
Abbreviation Detection – Deployment Documentation

Abstract

This report presents the development and deployment of an Abbreviation Detection system using NLP techniques. The implementation includes a Flask-based web application that serves the model's predictions through a user-friendly interface. The project also incorporates testing the deployed endpoints, the logging and log parsing strategies, and the CI/CD pipeline setup.

I. Research on Model Serving Options

In the Group Deployment phase of our coursework, selecting the appropriate model serving option is crucial for deploying our machine learning model effectively. This report evaluates various model serving options and justifies the choice of Flask for our project.

A. Criteria for Choosing a Model Serving Option

When selecting a model serving option, we considered the following criteria:

- **Scalability:** The ability to handle increasing loads.
- **Latency:** The time it takes to get a response from the model.
- **Ease of Use:** How easy it is to set up and manage.
- **Integration:** Compatibility with other tools and frameworks
- **Cost:** The overall cost of deployment.

B. Comparison of Model Serving Options

- **Flask**

Flask is a lightweight web framework for Python that is easy to set up and use. It is suitable for small

to medium sized projects and provides flexibility in designing the API endpoints. Flask is not as scalable as some other options but is ideal for projects that do not require handling many concurrent requests.

- **FastAPI**

FastAPI is a modern Python web framework for creating APIs, known for its speed and efficiency. It supports Python 3.6+ and uses type hints for easier coding. FastAPI automatically generates interactive API documentation, making development and testing simpler. It's ideal for high-performance, large-scale projects requiring quick response times.

- **Django**

Django is a high-level Python web framework that promotes quick development and clean design. It has many built-in features like an ORM, authentication, and an admin interface. Django is great for large, robust, and scalable projects but might be too much for simpler applications.

- **AWS SageMaker**

AWS SageMaker is a fully managed service that allows developers and data scientists to quickly build, train, and deploy machine learning models. It is highly scalable, offers low latency for real-time inference, and integrates well with the AWS ecosystem. SageMaker follows a pay-as-you-go pricing model and adheres to high security standards, making it a robust option for model serving. However, it can be costly, especially for large-scale projects.

CRITERIA	Flask	FastAPI	Django	AWS SageMaker	TensorFlow Serving	TorchServe
Scalability	Medium	High	High	Very High	Very High	High
Latency	Low	Very Low	Medium	Very Low	Very Low	Low
Ease of Use	High	High	Medium	Medium	Medium	Medium
Integration	High	High	High	Very High	High	High
Cost	Low	Low	Medium	Pay-as-you-go	Medium	Medium

Table 1: Comparing Model Serving Options

- **TensorFlow Serving**
TensorFlow Serving is a high-performance system for serving machine learning models in production. It primarily supports TensorFlow models but can be extended to serve other types. It is highly scalable, offers low latency, and is ideal for large-scale deployments, though it may require more setup and management than simpler frameworks. While the software itself is free, operational costs can add up, especially for large-scale or complex deployments.
- **TorchServe**
TorchServe is an open-source framework for serving PyTorch models in production. It simplifies deployment and management, offering features like multi-model serving and RESTful APIs. Great for PyTorch projects needing scalability, though operational costs should be considered.

C. Chosen Model Serving Option: FLASK

After evaluating various model serving options, we have chosen Flask for our project. The reasons for this choice are as follows:

- *Simplicity*: Flask is easy to set up and use, making it ideal for our project requirements.
- *Flexibility*: Flask provides the flexibility to design custom API endpoints tailored to our needs.
- *Ease of Integration*: Flask integrates well with other Python libraries and tools we are using in our project.
- *Cost-Effective*: Flask is open-source and does not incur additional costs for deployment and maintenance.
- *Sufficient for Our Needs*: Given the scope of our project, Flask's scalability and performance are sufficient to handle the expected load.

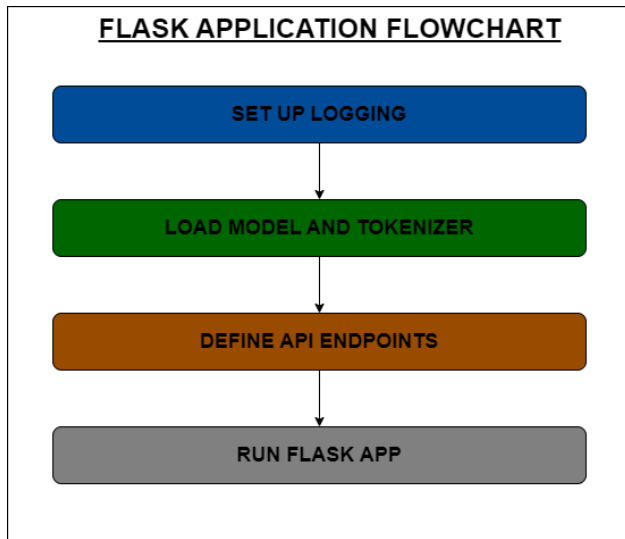
II. Building the Web Service

In this section, we describe the implementation of a web service to host our Natural Language Processing (NLP) model as an endpoint using Flask. The purpose of this web service is to allow clients to send text data and receive predictions from our pre-trained model, enabling real-time processing and analysis of text inputs.

A. Architectural Choices

- The Flask application serves as the interface for our machine learning model, allowing users to send text inputs and receive predictions. The application leverages the **transformers** library to load a pre-trained model and perform token classification.
- We chose the fine-tuned RoBERTa model for predicting acronyms and their long forms because it performed the best. RoBERTa had the highest macro F1 score, meaning it balanced precision and recall well. This makes it reliable and effective for our tasks. Using RoBERTa and the PLOD filtered dataset improves our system's accuracy and reliability.
- We used the Hugging Face Transformers library for text processing and tokenization. Specifically, we utilized the `AutoModelForTokenClassification` class and the `RobertaTokenizerFast` tokenizer. The `AutoModelForTokenClassification` class provides a pre-trained model suitable for token classification tasks, simplifying model handling and ensuring compatibility with various transformer models. The `RobertaTokenizerFast` efficiently processes input text for the RoBERTa model, ensuring accurate predictions by preserving structure and context with necessary prefixes and special tokens.
- For the first development stage, we decided to deploy the service locally. This has benefits like easier testing and debugging, without the hassle of using a remote server. Local deployment also sets the stage for transitioning to cloud environments later once everything's stable and tested.
- First, we set up an endpoint to show the UI page wherever the server is run. Then, we made another endpoint (`/predict`) to handle POST requests with text data in JSON format. Here's what it does:
 - Extracts the input text from the request body sent by the client.
 - Preprocess the text into tokens suitable for the RoBERTa model using the tokenizer.
 - Passes the tokenized text through the model to get predictions for each token.
 - Filters out special tokens added by the tokenizer.
 - Returns the predictions in a JSON response, showing the predicted classes.
 - for each token in an easy-to-understand format.

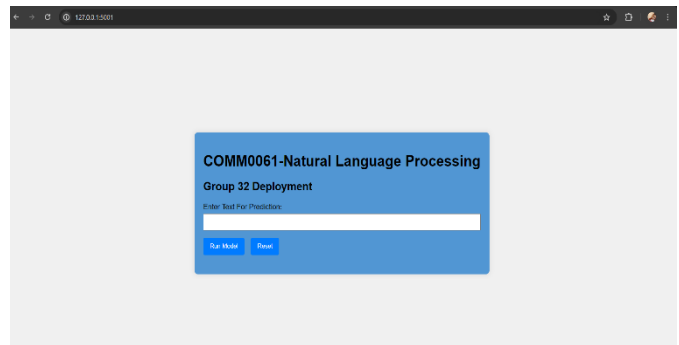
B. Implementation of Flask Application



- We started by setting up the Flask application and loading the necessary components for the web service. We initialized the model and tokenizer to process text inputs. The tokenizer preprocesses the text and converts it into a format suitable for the model. The model is loaded with its trained weights from CW1 and configurations to ensure accurate predictions from the PLOD dataset.
- To ensure efficient logging, we set up a **RotatingFileHandler** to capture important events and errors. This helps in monitoring the application's performance and diagnosing issues.
- The model is set to evaluation mode using **model.eval()**, which disables dropout layers and ensures that the model behaves correctly during inference.
- The service endpoint is designed to handle incoming POST requests containing text data. Upon receiving a request, the service extracts the input text and preprocesses it using the tokenizer. The pre-processed text is then passed through the model to obtain predictions. The model generates logits for each token, which are further processed to generate the final predictions.
- Special tokens added during tokenization are filtered out to ensure that only meaningful tokens are included in the response. This filtering step removes any unnecessary tokens that may not

contribute to the final predictions. Finally, the predictions are formatted into a JSON response and returned to the client.

- For frontend integration, we created a simple user interface where users can input text for prediction. The interface includes a text input field and a "Run" button. Users enter their text and click "Run" to send it to the backend for processing. The predictions are then displayed below the input field. There is also a "Reset" button to clear both the input and predictions, allowing users to start fresh with new text inputs.



III. Testing the Deployed Endpoint

Our testing process encompassed the following key phases:

Sample Data Testing: Initially, we conducted testing using data present in the PLOD dataset. By sending requests to the endpoint with sample data inputs and analysing the corresponding predictions, we aimed to verify the basic functionality and accuracy of the service.

Custom Data Testing: In addition to sample data, we performed testing with custom text inputs. These inputs were specifically designed to check how well the endpoint can handle different types of text. We varied factors such as text length, complexity, language, and format to mimic real-world scenarios. This helped us evaluate the endpoint's ability to handle diverse input types effectively.

A. Testing with a Jupyter Notebook

- To facilitate testing within Jupyter Notebook, we integrated the locally deployed URL of the endpoint. This setup allowed seamless communication between the notebook and the service. By defining the endpoint URL in the notebook, we established a direct connection, enabling us to interact with the endpoint programmatically.

- We created client functions in the notebook to interact with the deployed endpoint. These functions simplified HTTP request handling and response parsing, making testing easier. They were designed to send text data to the endpoint and retrieve the predictions from the service.

```
# Example usage
text = (
    "UOS = University of surrey "
)

predictions = get_predictions(text)
print(predictions)

[{"predictions": [{"UOS": "B-AC"}, {"is": "B-O"}, {"University": "B-LF"}, {"of": "I-LF"}, {"surrey": "I-LF"}]}
```

Figure 1. Prediction with Jupyter notebook

```
{'predictions': [{'UOS': 'B-AC'},
{'is': 'B-O'},
{'University': 'B-LF'},
{'of': 'I-LF'},
{'surrey': 'B-O'},
{'is': 'B-O'},
{'a': 'B-O'},
{'great': 'B-O'},
{'college': 'B-O'},
{'to': 'B-O'},
{'study': 'B-O'}]}
```

Figure 2. Prediction with Jupyter

• B. Testing with ‘curl’ command

We also tested the Flask application using **curl** commands in the terminal. This method is useful for quickly verifying the API's functionality from the command line.

```
PS C:\Users\naima\Downloads\Abbreviation_Detection-main\Abbreviation_Detection-main>
curl.exe -X POST -H "Content-Type: application/json" -d '{"text": "\nEPI = Echo
Planar Imaging\n"}' http://127.0.0.1:5001/predict
{"predictions":[{"EPI":"B-AC"}, {"is":"B-O"}, {"Echo":"B-LF"}, {"Planar":"I-LF"}, {"Imaging":"I-LF"}]}
```

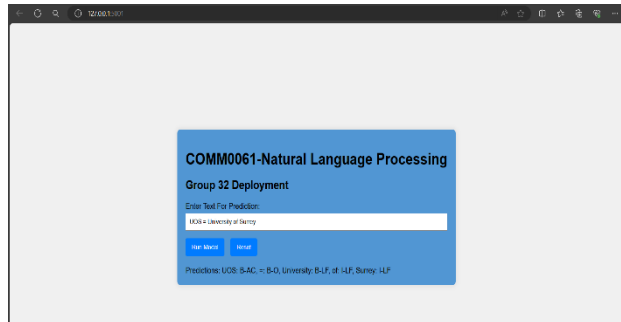
Figure 3. Using curl command

C. Testing with an HTML UI

- To provide a user-friendly interface for testing, we created an HTML page that allows users to input text and receive predictions from the API.

D. Findings

Our rigorous testing and evaluation of the deployed endpoint within the Jupyter Notebook environment yielded the following notable findings:



Accuracy: Across both sample and custom data inputs, the endpoint consistently produced accurate predictions, reflecting the efficacy of the underlying NLP model in capturing and understanding the nuances of the provided text.

Performance: The endpoint exhibited satisfactory performance in terms of response time and processing speed. We observed minimal latency in receiving predictions, indicating efficient processing and response handling by the deployed service. The breaking point and the load testing is done in detail in the below sections

Robustness: The endpoint demonstrated robustness in processing a wide range of text inputs, including inputs of varying lengths, complexities, and formats. The weakness of the model was noted where it has difficulties for longer tokens in the long forms where it was accurately predicting when the long form was analyzed alone.

IV. Performance Analysis

There are several ways to ensure the robustness and scalability of the Abbreviation detection application. In this Abbreviation detection system, we have implemented load testing using Locust. The primary focus was to analyse the performance of our Flask application, which serves as the backend for abbreviation prediction using the Roberta model and identify its demerits.

A. Locust Load Testing

Locust is an open-source load testing tool that allows us to simulate user behavior and measure the performance of web applications. It is particularly helpful for analyzing and identifying how an application handles concurrent requests on varying loads.



Figure 5. Locust Chart

B. Test Setup

The locustfile.py script defines the user behaviour for sending POST requests to our prediction endpoint. In locust script we simulate user behaviour by creating tasks that repeatedly send requests to the server. It will help us to measure the application's performance under load. We conducted load testing using Locust with a maximum of 1000 concurrent users and a ramp-up rate of 10 users per second. The / predict endpoint was tested using a JSON payload containing text for abbreviation detection from the locustfile-.py

C. Observations

Up to 150 users, the application handled requests efficiently, reaching a maximum response per second of 25 which is indicating good performance under moderate load. Beyond 150 users, the RPS started to decrease. The application stabilised at around 5 RPS with 500 users. However, the system started failing at 700 users, with significantly increased response times and many requests timing out. The average response time is increased proportionally with the increased load. The 95th percentile response time also rose significantly, indicating some requests took much longer to process. The system started failing to handle requests efficiently at 700 users, as evidenced by a spike in failed requests and increased response times. Our system performs well under moderate load but starts failing after reaching the threshold of 700 users.

D. Analysis

The application performs very well under moderate load conditions (up to 150 users), with high RPS and stable response times, indicating effective resource utilisation. However, the application struggles under high loads, indicating potential resource limitations (CPU, memory, I/O). Performance degrades significantly with increased

load, suggesting bottlenecks in the application or infrastructure. High concurrency leads to increased latency and request failures.

E. Recommendations

Improve code efficiency and handling of concurrent requests. Implement caching strategies to reduce server load and speed up response times. Streamline the model's inference pipeline for better performance under load. Consider using a lighter model or quantizing the current model. Deploy multiple instances of the Flask application and use a load balancer to distribute the load evenly across servers. Implement comprehensive monitoring and logging to proactively identify and address performance issues. Use tools like Prometheus and Grafana for real-time monitoring and alerting.

V. Building Basic Monitoring Capability

Logging is a crucial aspect of any application as it helps in monitoring the application's behavior, debugging issues, and maintaining a record of events. In our Flask application, we have implemented logging to capture important events, such as incoming requests and their corresponding predictions.

A. Logging Setup

We set up logging in the Flask application using Python's built-in **logging** module. The **RotatingFileHandler** is used to manage log files efficiently, ensuring that the log files do not grow indefinitely.

B. Logging Requests and Predictions

In the Flask application, we log each incoming request along with the input text and the model's predictions. This helps in tracking the inputs processed by the model and the corresponding outputs.

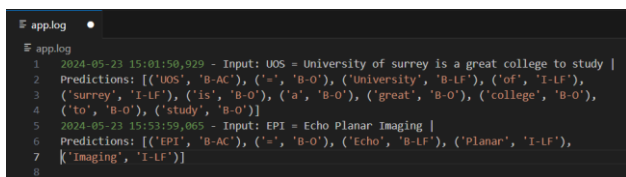
C. Detailed Explanation of Log Components

Timestamp: The timestamp is recorded in a precise format that includes the year, month, day, hour, minute, second, and milliseconds. This level of precision is essential for accurately tracing interactions and understanding the exact sequence of events. For instance, a timestamp like "2024-05-23 12:36:33,700" indicates that the interaction occurred on May 23, 2024, at 12:36:33 and 700 milliseconds. Including the timestamp in the logs is crucial for maintaining a chronological record of all activities. It allows us to track when each request was processed, correlate logs with external events or other system logs, and diagnose issues that may be time-dependent, such as those related to specific periods of high usage or versions of the service.

User Input: Logged as provided by the client, preserving the exact context and phrasing used, user input data integrity is maintained for future analysis. Detailed input records inform future improvements, feature developments, and identification of potential issues.

Model Predictions: Logged predictions show what the model thinks about each word, helping check how well it's doing, track its progress, and tweak it if needed.

Example Log Entries:



```
app.log
app.log
1 2024-05-23 15:01:50,929 - Input: UOS = University of surrey is a great college to study |
2 Predictions: [['UOS', 'B-AC'), ('=', 'B-O'), ('University', 'B-LF'), ('of', 'I-LF'),
3 ('surrey', 'I-LF'), ('is', 'B-O'), ('a', 'B-O'), ('great', 'B-O'), ('college', 'B-O'),
4 ('to', 'B-O'), ('study', 'B-O')]
5 2024-05-23 15:53:59,065 - Input: EPI = Echo Planar Imaging |
6 Predictions: [['EPI', 'B-AC'), ('=', 'B-O'), ('Echo', 'B-LF'), ('Planar', 'I-LF'),
7 ('Imaging', 'I-LF')]
```

Figure 6. Log Entries

D. Parsing Log Files

To analyze the log files, we created a script (**logparse.py**) that parses the log entries and extracts relevant information such as timestamps, input texts, and predictions.

E. Benefits of Logging

Implementing this detailed logging mechanism provides several significant benefits:

Traceability: Tracking every interaction with the web service creates a detailed record. It helps make sure requests are handled correctly and the service works as it should. It also helps follow rules about managing data and checking things later.

Debugging: Logs are like clues to solve problems. They show exactly what happened when something went

wrong. For example, if the model makes a mistake, the log tells us what it was given and what it predicted, helping us figure out what went wrong and how to fix it.

Performance Analysis: Looking at logs helps understand how well the model is doing over time. We can see patterns in what people ask, check if the predictions are accurate, and spot any changes in how well it works. This helps decide if the model needs to be updated or changed.

Usage Monitoring: Logs show us how people are using the service, like how many requests it gets, when it's busiest, and what people ask for most often. This helps improve the service to better match what people need and plan.

Example Explained: If someone sends in a long message about different things, the log records when it happened, exactly what was said, and what the model guessed for each word. This helps us understand how well the model works in different situations.

By implementing logging in the Flask application, we can effectively monitor and debug the application. The logs provide valuable insights into the inputs processed by the model and the corresponding predictions, helping us ensure the application's accuracy and reliability.

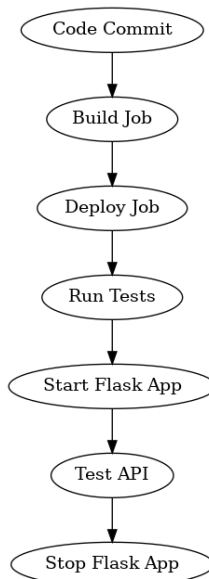
VI. CI/CD Pipeline

Continuous Integration and Continuous Deployment (CI/CD) are crucial practices in modern software development. They help automate the process of integrating code changes, running tests, and deploying applications, ensuring that the software is always in a deployable state. In this section, we describe the CI/CD pipeline set up for our Flask application using GitHub Actions.

A. Pipeline Configuration

The CI/CD pipeline is configured using a YAML file that defines the workflow for building, testing, and deploying the application. The pipeline is triggered on pushes and pull requests to the **main** branch.

B. Pipeline Stages



1. Triggering the Pipeline

The pipeline is triggered by two events:

- **Push:** When code is pushed to the **main** branch.
- **Pull Request:** When a pull request is made to the **main** branch.

2. Build Job

The **build** job is responsible for setting up the environment, installing dependencies, and running tests.

- **Runs-on:** The job runs on the latest version of Ubuntu (**ubuntu-latest**).
- **Steps:**
 - **Checkout** **Code:** Uses the actions/checkout@v2 action to check out the repository's code.
 - **Set Up Python:** Uses the actions/setup-python@v2 action to set up Python 3.12.
 - **Install Dependencies:** Upgrades pip and installs the required dependencies from requirements.txt.
 - **Run Tests:** Executes the tests using pytest to ensure the code is functioning correctly.

3. Deploy Job

The **deploy** job is responsible for deploying the application. It depends on the successful completion of the **build** job.

- **Runs-on:** The job runs on the latest version of Ubuntu.

- **Needs:** Specifies that this job depends on the **build** job.
- **Steps:**
 - **Checkout** **Code:** Uses the actions/checkout@v2 action to check out the repository's code.
 - **Set Up Python:** Uses the actions/setup-python@v2 action to set up Python 3.12.
 - **Install Dependencies:** Upgrades pip and installs the required dependencies from requirements.txt.
 - **Run Flask App:** Starts the Flask application in the background using nohup python app.py &.
 - **Wait for Flask App to Start:** Introduces a delay of 10 seconds to allow the Flask app to start.
 - **Test API:** Runs a script (test_api.py) to test the API and ensure it is functioning correctly.
 - **Stop Flask App:** Stops the Flask application using pkill -f "python app.py".

The CI/CD pipeline ensures that the Flask application is automatically built, tested, and deployed whenever changes are pushed to the **main** branch, or a pull request is made. This automation helps maintain the quality and reliability of the application, reducing the risk of errors and ensuring that the application is always in a deployable state.

VII. References

1. Flask. Flask Documentation. Retrieved from <https://flask.palletsprojects.com/en/2.0.x/>
2. FastAPI. FastAPI Documentation. Retrieved from <https://fastapi.tiangolo.com/>
3. Django. Django Documentation. Retrieved from <https://www.djangoproject.com/>
4. TensorFlow Serving. TensorFlow Serving Overview. Retrieved from <https://www.tensorflow.org/tfx/serving/overview>
5. AWS SageMaker. Amazon SageMaker Documentation. Retrieved from <https://aws.amazon.com/sagemaker/>
6. TorchServe. PyTorch Serve Documentation. Retrieved from <https://pytorch.org/serve/>
7. Hugging Face Transformers. Transformers Documentation. Retrieved from <https://huggingface.co/transformers/>
8. Locust. Locust Documentation. Retrieved from <https://locust.io/>
9. GitHub Actions. GitHub Actions Documentation. Retrieved from <https://docs.github.com/en/actions>