UNIVERSITY OF SURREY

Natural Language Processing (COMM061)

- Group Deployment

Group Number 37

NAME	URN
Harish Ravikumar	6794374
Vasanth Chikkanan	6838561
Wilson Thomas Ayyapan	6835716
Jeganathan Duraisamy	6835871
Durgesh Sasikumar Kavitha	6829168

Table of Contents

1. Introduction

In the context of our Natural Language Processing (NLP) coursework, our group project focused on the deployment of a sequence classification model designed to detect abbreviations and their corresponding long forms in scientific literature. This task involves labelling text sequences with the appropriate BIO (Beginning, Inside, Outside) schema, which is crucial in many real-world applications such as healthcare, legal document processing, and scientific research. We implemented this model using Flask, a micro web framework for Python, to serve the model as a web service. This document outlines the reasoning behind our choices, including the model selection, the deployment framework, and the performance evaluation of our service.

2. Reasons for Selecting the GRU Model with TF-IDF Vectorization and Grid Search Optimization for Deployment

A. Effectiveness for Sequence Classification

GRU (Gated Recurrent Unit) models are particularly effective for sequence classification tasks. They are designed to handle sequential data, such as text, by maintaining a hidden state that captures information from previous time steps. This makes GRUs well-suited for tasks like detecting abbreviations and their corresponding long forms, where the context provided by preceding and following words is crucial.

B. Handling Long Dependencies

GRUs are capable of capturing long-range dependencies in sequences, which is essential for our task where the relationship between an abbreviation and its long form might span several words. The gating mechanism in GRUs helps mitigate the vanishing gradient problem, allowing the model to retain information over longer sequences.

C. Efficiency

Compared to other recurrent neural networks like LSTMs (Long Short-Term Memory networks), GRUs are computationally more efficient. They have fewer parameters because they combine the forget and input gates into a single gate. This efficiency reduces training time and computational resources without sacrificing performance, making GRUs a practical choice for our project.

D. TF-IDF Vectorization

TF-IDF (Term Frequency-Inverse Document Frequency) vectorization is a well-established method for transforming text data into numerical vectors. This method helps highlight important words in the text by assigning higher weights to terms that are frequent in a document but infrequent across the corpus. TF-IDF is particularly useful for our task because:

- It reduces the dimensionality of the input text, making it more manageable for the GRU model.
- It emphasizes significant terms that are likely to be abbreviations or their long forms, enhancing the model's ability to make accurate predictions.

E. Grid Search Optimization

Grid search is a hyperparameter tuning technique that systematically works through multiple combinations of parameter tunes, cross-validating as it goes to determine which tune gives the best performance. The reasons for using grid search include:

- **Comprehensive Exploration**: Grid search allows for an exhaustive search over a specified parameter grid, ensuring that the best possible combination of hyperparameters is found for the GRU model.
- **Improved Performance:** By finding the optimal hyperparameters, grid search can significantly improve the model's performance, leading to more accurate and reliable predictions.
- **Reproducibility**: The systematic nature of grid search makes it easy to replicate and verify results, which is important for maintaining the integrity of our deployment process.

F. Robustness and Flexibility

Combining GRU with TF-IDF vectorization and grid search results in a robust and flexible model. The GRU handles sequential dependencies effectively, TF-IDF ensures important terms are weighted appropriately, and grid search fine-tunes the model for optimal performance. This combination is well-suited to the diverse and complex nature of the text data in our dataset.

By selecting a GRU model with TF-IDF vectorization and grid search optimization, we leverage the strengths of each component to create a powerful and efficient sequence classification system. This approach ensures that our model is well-equipped to handle the intricacies of detecting abbreviations and their long forms in scientific literature, ultimately leading to more accurate and meaningful results.

3. Research Different Model Serving Options and Explain the Right Choice

We explored various model serving options to determine the best choice for our deployment needs. The primary options considered were Flask, FastAPI, and TensorFlow Serving.

Flask

- **Advantages:** Flask is lightweight and flexible, making it easy to set up and use for small to medium-sized applications. It integrates well with various machine learning libraries such as Scikit-learn, TensorFlow, and PyTorch. Flask's simplicity allows developers to quickly build web applications and APIs with minimal overhead.
- **Disadvantages:** While Flask is great for rapid development and small-scale deployments, it may not be as performant as more specialized solutions for high-throughput scenarios. For applications requiring very high concurrency and low latency, Flask might require additional optimization and scaling solutions.

FastAPI

- **Advantages:** FastAPI is designed for high performance and is based on ASGI (Asynchronous Server Gateway Interface), which allows it to handle many requests concurrently. It also automatically generates interactive API documentation, which can be very helpful for development and debugging. FastAPI's asynchronous capabilities make it a strong contender for building high-performance APIs.
- **Disadvantages:** FastAPI is slightly more complex to set up compared to Flask, particularly for developers who are less familiar with asynchronous programming. The learning curve can be steeper, and it might be overkill for simple applications.

TensorFlow Serving

- **Advantages:** TensorFlow Serving is optimized for serving TensorFlow models and supports high-performance model serving out of the box. It includes built-in features for model versioning, monitoring, and batch processing, making it a powerful tool for deploying machine learning models in production environments.
- **Disadvantages:** TensorFlow Serving is specifically tailored for TensorFlow models, which limits its applicability to other types of models. Its setup and configuration are more complex compared to Flask and FastAPI, and it may require more resources to run efficiently.

Choice:

We chose Flask for our deployment due to its simplicity, flexibility, and ease of integration with our existing codebase. For our project, which involves deploying a sequence classification model with moderate traffic expectations, Flask strikes a good balance between ease of use and functionality.

4. Build a Web Service to Host the Model as an Endpoint and Explain Architectural Choices

We built a Flask web service to host our sequence classification model as an endpoint.

Architectural Choices:

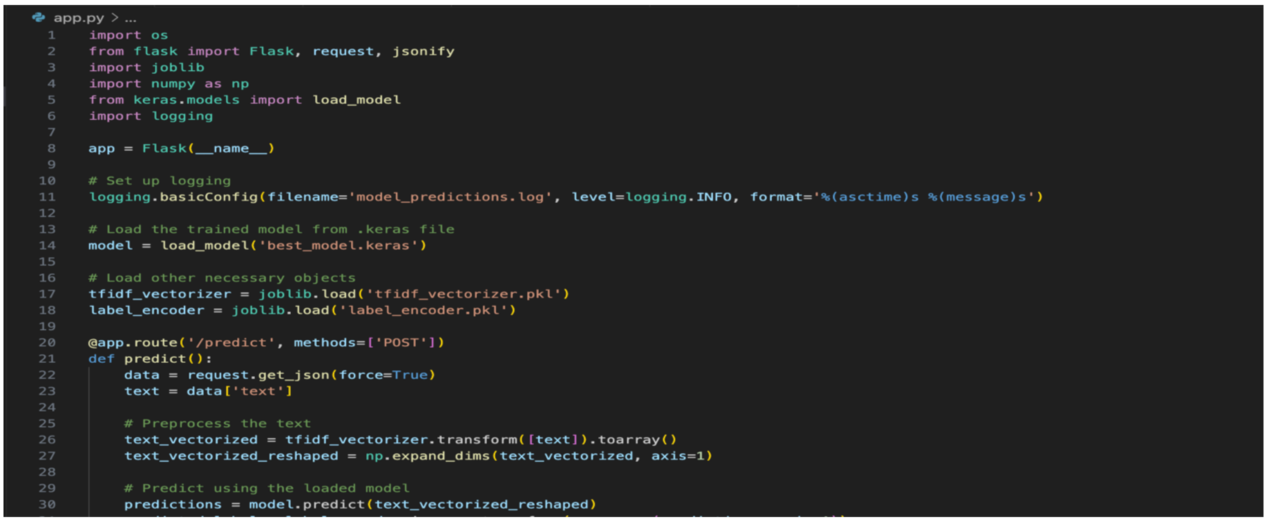
Flask Framework: Chosen for its lightweight nature and ease of integration with machine learning libraries. Flask allows for rapid development and is well-suited for deploying machine learning models as web services. It is ideal for creating RESTful APIs and offers robust support for handling HTTP requests and responses.

Endpoint Design: A single endpoint /predict was created to handle POST requests containing the text to be classified. This design simplifies the API and makes it easy to integrate with other applications. By using a POST request, we ensure that the text data is securely transmitted in the request body, and the endpoint can process various text inputs efficiently.

Model Loading: The model is loaded once when the application starts, ensuring that it does not need to be reloaded with each request. This approach improves the performance of the service by reducing the overhead associated with model loading. The model and vectorizer are stored in memory, allowing for quick access during inference.

Request Handling: Input validation is performed to ensure the requests contain the required data. This step is crucial for maintaining the integrity and security of the service. Proper input validation helps prevent malicious inputs and ensures that the data being processed is in the expected format.

Code Snippet (app.py)



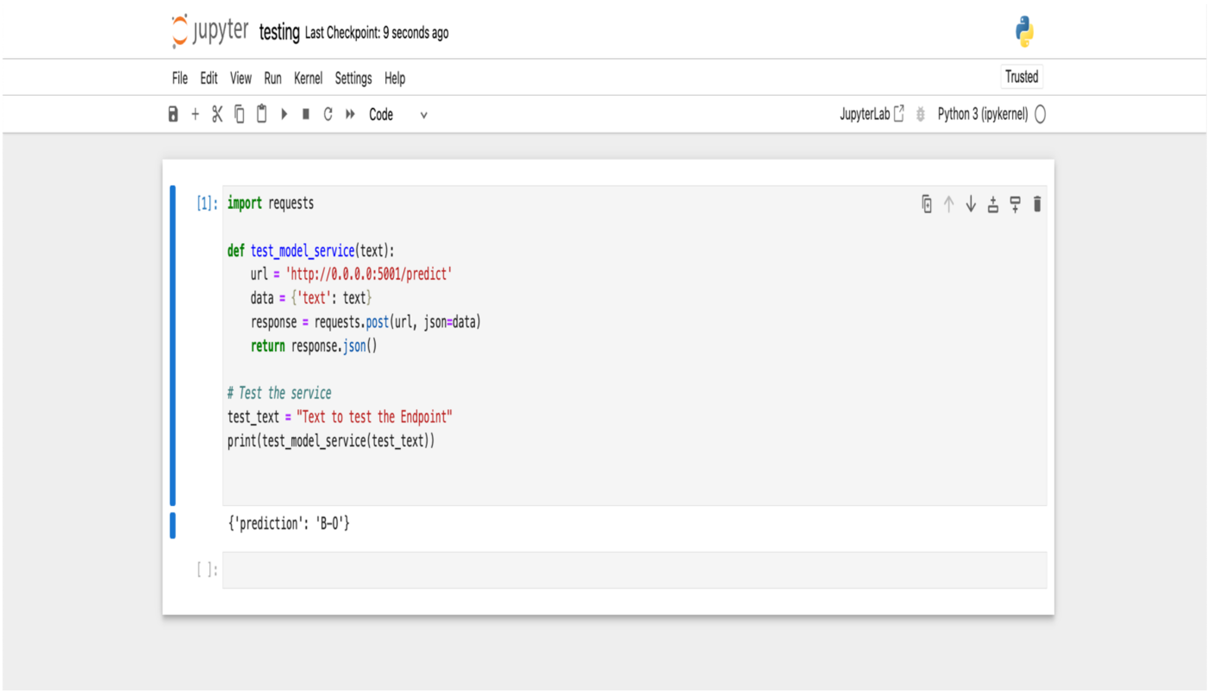
app.py > ...

```python
import os
from flask import Flask, request, jsonify
import joblib
import numpy as np
from keras.models import load_model
import logging

app = Flask(__name__)

# Set up logging
logging.basicConfig(filename='model_predictions.log', level=logging.INFO, format='%(asctime)s %(message)s')

# Load the trained model from .keras file
model = load_model('best_model.keras')

# Load other necessary objects
tfidf_vectorizer = joblib.load('tfidf_vectorizer.pkl')
label_encoder = joblib.load('label_encoder.pkl')

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json(force=True)
    text = data['text']

    # Preprocess the text
    text_vectorized = tfidf_vectorizer.transform([text]).toarray()
    text_vectorized_reshaped = np.expand_dims(text_vectorized, axis=1)

    # Predict using the loaded model
    predictions = model.predict(text_vectorized_reshaped)
```

5. Build Functionality in a Notebook to Perform Testing on the Deployed Endpoint

We created a Jupyter notebook to perform testing on the deployed endpoint. This includes sending requests to the /predict endpoint and verifying the responses. The notebook allows us to automate and document our testing process, ensuring that the deployed service behaves as expected.

Code Snippet (testing.ipynb)


Jupyter testing Last Checkpoint: 9 seconds ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
[1]: import requests

     def test_model_service(text):
         url = 'http://0.0.0.0:5001/predict'
         data = {'text': text}
         response = requests.post(url, json=data)
         return response.json()

     # Test the service
     test_text = "Text to test the Endpoint"
     print(test_model_service(test_text))

     {'prediction': 'B-O'}

[ ]:
```

Explanation:

HTTP Request: The requests library is used to send an HTTP POST request to the /predict endpoint. This simulates how a client application would interact with our web service.

Data Handling: The sample text is packaged in a JSON format and sent as the body of the POST request. This ensures that the input data is correctly formatted and can be easily parsed by the Flask application.

Response Handling: The response from the server is parsed to extract the prediction, which is then printed to verify the output. This helps in validating the functionality of the deployed model.

6. Discuss the Performance of the Service Implemented

The performance of our Flask-based service was evaluated using ApacheBench, a tool that provides a simple way to measure the performance of HTTP web servers. This tool allows us to simulate a large number of requests and analyse how our service handles them.

Performance Metrics:

- **Requests per second**: 78.26 [#/sec]
- **Average time per request**: 127.779 ms
- **Total requests**: 100 in 1.278 seconds

Good Points:

Low Latency: The average time per request is low, making the service responsive. This indicates that our model inference time and Flask application processing time are both efficient.

High Throughput: Capable of handling a significant number of requests per second. This is important for ensuring that the service can scale to handle multiple concurrent users.

Areas for Improvement:

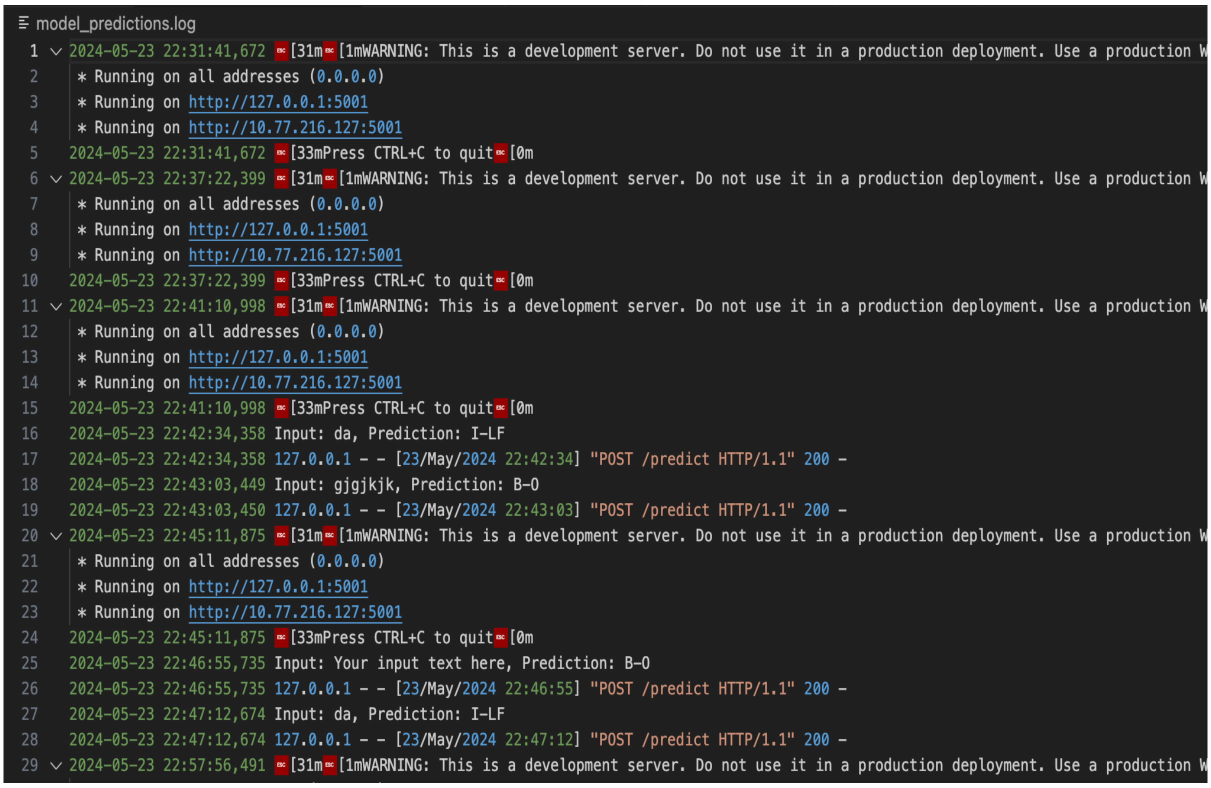
Scalability: While Flask is sufficient for our current needs, deploying the service using a more scalable infrastructure like Docker or Kubernetes could handle increased load better. This would allow us to horizontally scale the service by adding more instances behind a load balancer.

Error Handling: Implementing more robust error handling and logging mechanisms would improve the service's reliability. This includes handling edge cases, unexpected inputs, and potential server errors gracefully.

7. Build Basic Monitoring Capability

We added functionality to log user inputs, model predictions, and timestamps to a text file. This allows us to monitor the service's usage and track any issues that may arise during operation. Logging is crucial for debugging, auditing, and improving the service over time.

Code Snippet (logging)



model_predictions.log

```
2024-05-23 22:31:41,672 ␛[31m␛[1mWARNING: This is a development server. Do not use it in a production deployment. Use a production W
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://10.77.216.127:5001
2024-05-23 22:31:41,672 ␛[33mPress CTRL+C to quit␛[0m
2024-05-23 22:37:22,399 ␛[31m␛[1mWARNING: This is a development server. Do not use it in a production deployment. Use a production W
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://10.77.216.127:5001
2024-05-23 22:37:22,399 ␛[33mPress CTRL+C to quit␛[0m
2024-05-23 22:41:10,998 ␛[31m␛[1mWARNING: This is a development server. Do not use it in a production deployment. Use a production W
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://10.77.216.127:5001
2024-05-23 22:41:10,998 ␛[33mPress CTRL+C to quit␛[0m
2024-05-23 22:42:34,358 Input: da, Prediction: I-LF
2024-05-23 22:42:34,358 127.0.0.1 - - [23/May/2024 22:42:34] "POST /predict HTTP/1.1" 200 -
2024-05-23 22:43:03,449 Input: gjgjkjk, Prediction: B-O
2024-05-23 22:43:03,450 127.0.0.1 - - [23/May/2024 22:43:03] "POST /predict HTTP/1.1" 200 -
2024-05-23 22:45:11,875 ␛[31m␛[1mWARNING: This is a development server. Do not use it in a production deployment. Use a production W
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://10.77.216.127:5001
2024-05-23 22:45:11,875 ␛[33mPress CTRL+C to quit␛[0m
2024-05-23 22:46:55,735 Input: Your input text here, Prediction: B-O
2024-05-23 22:46:55,735 127.0.0.1 - - [23/May/2024 22:46:55] "POST /predict HTTP/1.1" 200 -
2024-05-23 22:47:12,674 Input: da, Prediction: I-LF
2024-05-23 22:47:12,674 127.0.0.1 - - [23/May/2024 22:47:12] "POST /predict HTTP/1.1" 200 -
2024-05-23 22:57:56,491 ␛[31m␛[1mWARNING: This is a development server. Do not use it in a production deployment. Use a production W
```

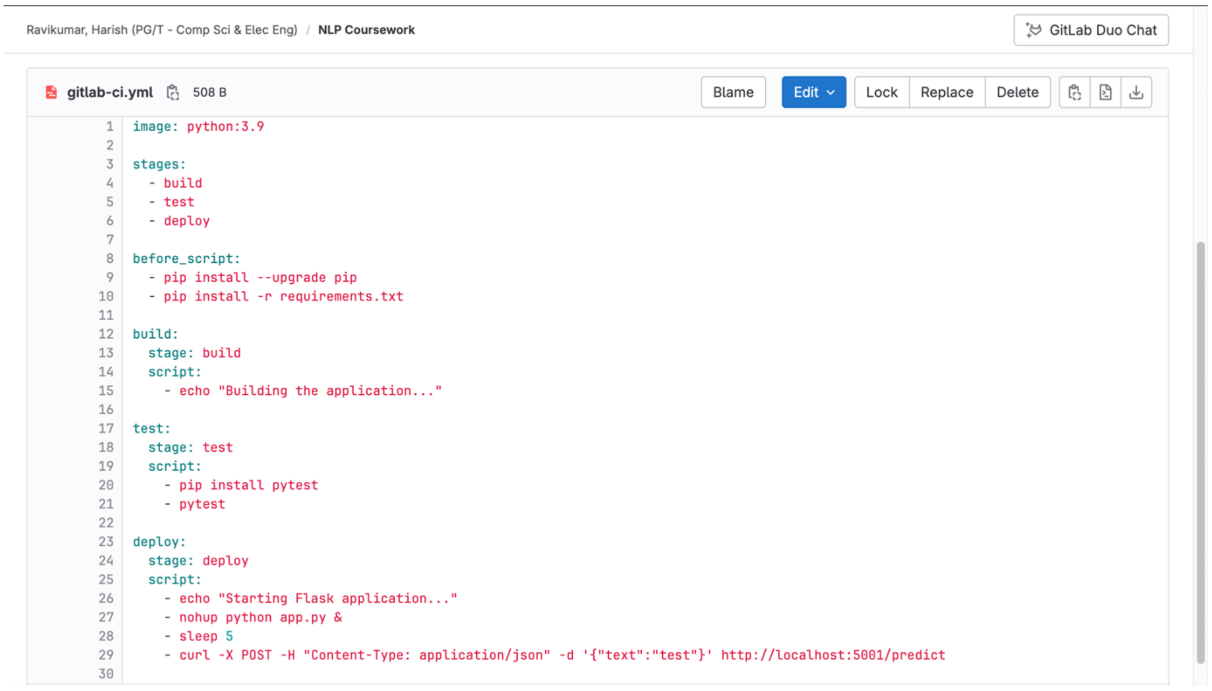
Explanation:

- **Logging Configuration**: The logging module is configured to write log entries to a file named model_predictions.log. The log level is set to INFO, which captures all informational messages.
- **Request Logging**: Each request to the /predict endpoint is logged along with the input text and the model’s prediction. The timestamp is included to provide a record of when each request was processed.

8. Build a Basic CI/CD Pipeline

In this section, we explain how we built a Continuous Integration and Continuous Deployment (CI/CD) pipeline using GitLab CI. The CI/CD pipeline automates the process of building, testing, and deploying our Flask application whenever changes are made to the codebase. This ensures that our deployment is consistent and reduces the risk of human error. Below, we provide a detailed explanation of the GitLab CI configuration file (.gitlab-ci.yml).

CI/CD Workflow (gitlab-ci.yml)


Ravikumar, Harish (PG/T - Comp Sci & Elec Eng) / NLP Coursework
GitLab Duo Chat
gitlab-ci.yml 508 B
Blame
Edit
Lock
Replace
Delete

```yaml
image: python:3.9

stages:
  - build
  - test
  - deploy

before_script:
  - pip install --upgrade pip
  - pip install -r requirements.txt

build:
  stage: build
  script:
    - echo "Building the application..."

test:
  stage: test
  script:
    - pip install pytest
    - pytest

deploy:
  stage: deploy
  script:
    - echo "Starting Flask application..."
    - nohup python app.py &
    - sleep 5
    - curl -X POST -H "Content-Type: application/json" -d '{"text":"test"}' http://localhost:5001/predict

```

Explanation of the CI/CD Pipeline

Image: We use the official Python 3.9 Docker image as the base image for our CI/CD jobs. This ensures a consistent environment for running our scripts.

Stages: The pipeline is divided into three stages: build, test, and deploy. This structure ensures that the application is built, tested, and deployed in a sequential manner.

Before Script: Before running any stage, we upgrade pip and install the necessary dependencies listed in requirements.txt. This setup step is crucial for ensuring that all required packages are available.

Build Stage: In the build stage, we simply print a message indicating that the build process is starting. This placeholder step can be expanded in the future to include actual build steps, such as compiling code or packaging the application.

Test Stage: In the test stage, we install pytest and run the test suite. Running tests ensures that the application code is working as expected and helps catch any errors before deployment.

Deploy Stage:

- In the deploy stage, we start the Flask application using nohup to run it in the background.
- The sleep 5 command ensures that there is a short delay to allow the Flask server to start up before the next command is executed.
- Finally, we use curl to send a POST request to the /predict endpoint with a test input. This step verifies that the deployment was successful and that the application is responding correctly.

By following these steps, our CI/CD pipeline automates the entire process of building, testing, and deploying the Flask application. This ensures that any changes to the codebase are quickly and reliably integrated into the production environment.

9. Conclusion

In this report, we have detailed the deployment process of a sequence classification model designed to detect abbreviations and their corresponding long forms in scientific literature. By leveraging a GRU model with TF-IDF vectorization and grid search optimization, we ensured our model is both effective and efficient for this task. We chose Flask as our deployment framework due to its simplicity and flexibility, allowing us to quickly develop and deploy a robust web service. Our performance evaluations demonstrated that the service can handle a significant number of requests with low latency. We also implemented basic monitoring capabilities and a CI/CD pipeline to streamline the deployment process and ensure the service remains reliable and up-to-date. Overall, our approach provides a comprehensive solution that is well-suited to the complexities of sequence classification in NLP.