

# **Natural Language Processing- Group Coursework**

**COMM061**

**Group 7**

## **Group Members**

<b>Name</b>	<b>URN</b>
Saksham Ashwini Rai	6806149
Raj Vinod Mistry	6800650
Suhas Trimbak Barapatre	6786248
Aditya Narayan Sawant	6828571
Kunal Vinayshankar Singh	6838345

## Table of Contents

Section		Page Number
<b>1.</b>	<b>Research on Model Serving Options</b>	<b>4</b>
	1.1 Introduction	4
	1.2 Research Methodology	4
	1.3 Evaluation of Model Serving Options	4
	1.4 Chosen Solution: FastAPI	6
	1.5 Model Selection: Individual Experiments and Key Findings	7
<b>2.</b>	<b>Web Service Implementation</b>	<b>10</b>
	2.1 Introduction	10
	2.2 Architectural Choices	10
	2.3 Implementation Details	11
<b>3.</b>	<b>Client Function for Endpoint Testing</b>	<b>13</b>
	3.1 Introduction	13
	3.2 Testing Functionality	13
	3.3 Findings	14
	3.4 Conclusion	15
<b>4.</b>	<b>Performance Evaluation</b>	<b>16</b>
	4.1 Stress Test Methodology	16
	4.2 Stress Test Results	17
	4.3 Good Points	19
	4.4 Areas for Improvement	19
	4.5 Recommendations	20
<b>5.</b>	<b>Monitoring and Logging</b>	<b>21</b>

	5.1	Logging Setup	21
	5.2	Logging Configuration	21
	5.3	Logging Predictions	21
	5.4	Accessing Logs Programmatically	22
	5.5	Benefits of Logging	22
<b>6.</b>	<b>CI/CD Pipeline Implementation</b>		<b>23</b>
	6.1	Introduction	23
	6.2	CI/CD Pipeline Implementation	23
	6.3	Execution Script	24
	6.4	Performance Metrics	25
	6.5	Confusion Matrix	26
	6.6	Summary	27

# 1. Research on Model Serving Options

## 1.1 Introduction

Model serving is a critical aspect of deploying machine learning models into production. It involves providing an endpoint that can handle requests and return predictions from the model in real-time. There are various frameworks and tools available for serving machine learning models, each with its own advantages and limitations. In this section, we explore different model serving options and explain our choice for this project.

## 1.2 Research Methodology

To determine the most suitable model serving option for our NLP project, we conducted a comprehensive review of the following frameworks:

1. **TensorFlow Serving**
2. **TorchServe**
3. **FastAPI**
4. **Flask**

Our evaluation criteria included factors such as scalability, ease of deployment, support for different ML frameworks, community support, and performance.

## 1.3 Evaluation of Model Serving Options

### 1.3.1 TensorFlow Serving

- **Overview:** TensorFlow Serving is a flexible, high-performance serving system for machine learning models, designed for production environments. It supports TensorFlow models natively and can be extended to serve other types of models.
- **Pros:**
  - High scalability and performance.
  - Extensive documentation and community support.
  - Robust model management features (e.g., versioning).
- **Cons:**
  - Primarily optimized for TensorFlow models.
  - Can be complex to set up and configure, especially for non-TensorFlow models.

### 1.3.2 TorchServe

- **Overview:** TorchServe is an open-source model serving framework for PyTorch models. It provides features such as multi-model serving, logging, metrics, and more.
- **Pros:**
  - Excellent support for PyTorch models.
  - Easy to deploy and manage.
  - Provides out-of-the-box support for logging and monitoring.
- **Cons:**
  - Limited support for models not built with PyTorch.
  - Requires familiarity with PyTorch.

### 1.3.3 FastAPI

- **Overview:** FastAPI is a modern, fast (high-performance) web framework for building APIs with Python. It is based on standard Python type hints and is designed to be easy to use and integrate with various machine learning models.
- **Pros:**
  - High performance due to asynchronous request handling.
  - Easy to set up and deploy.
  - Supports multiple ML frameworks (TensorFlow, PyTorch, etc.).
  - Excellent documentation and community support.
- **Cons:**
  - Requires additional setup for complex deployments (e.g., load balancing).

### 1.3.4 Flask

- **Overview:** Flask is a lightweight WSGI web application framework in Python. It is simple to use and suitable for small to medium-sized applications.
- **Pros:**
  - Simple and easy to get started.
  - Well-documented with a large community.

- Flexible and extensible.
- **Cons:**
  - Not optimized for high-performance or large-scale deployments.
  - Limited built-in support for asynchronous handling.

## 1.4 Chosen Solution: FastAPI

### 1.4.1 Rationale for Choosing FastAPI

After evaluating the different options, we decided to use FastAPI for our model serving needs. The key reasons for this choice are:

- **Performance:** FastAPI is designed for high performance, leveraging asynchronous request handling to manage multiple concurrent requests efficiently.
- **Flexibility:** It supports various machine learning frameworks, making it easy to integrate with our existing model built with PyTorch.
- **Ease of Use:** FastAPI is user-friendly and well-documented, which simplifies the development and deployment process.
- **Scalability:** While additional setup is required for complex deployments, FastAPI can be integrated with tools like Kubernetes for scalability.

## 1.5 Model Selection:

### Individual Experiments and Key Findings

In our project, each team member conducted individual experiments to evaluate various models and preprocessing techniques for the task of abbreviation detection within scientific texts. Here, we summarize the findings from each member and conclude with the selection of BERT as the best-performing model for our group project.

#### 1. Kunal

##### Experiments Conducted:

- **Data Preprocessing:** Explored tokenization, TF-IDF, and Word2Vec vectorization. Utilized SMOTE for handling class imbalance and NER tag encoding.
- **Model Comparison:** Evaluated SVM and BERT. BERT significantly outperformed SVM with an F1-score of 0.89.
- **Optimization Techniques:** Applied Bayesian optimization to fine-tune BERT, achieving an improved F1-score of 0.95.
- **Optimizer and Loss Function Selection:** Chose AdamW optimizer and cross-entropy loss, enhancing model update efficacy and precision.

##### Key Findings:

- **Word2Vec vs. TF-IDF:** Word2Vec offered better context comprehension, improving by 15%.
- **SMOTE:** Enhanced recall for minority classes by 20%.
- **BERT Performance:** BERT demonstrated superior contextual comprehension and optimization capabilities, yielding the highest F1-score of 0.95.

#### 2. Aditya

##### Experiments Conducted:

- **Data Preprocessing:** Investigated stemming and lemmatization, both yielding similar accuracies with slight F1-score variations.
- **Loss Functions and Optimizers:** Tested Categorical Cross-Entropy with optimizers like Adam, Adamax, SGD, and Adelta.
- **Hyperparameter Tuning:** Experimented with different batch sizes (32, 64, 128) and epochs (5, 10, 20).

- **Model Evaluations:** Compared BioBERT, BiLSTM, and SVM.

#### Key Findings:

- **BioBERT:** Emerged as the top performer due to its domain-specific pretraining, excelling in NER tasks.
- **BiLSTM:** Initial performance was underwhelming but improved with meticulous hyperparameter tuning.
- **SVM:** Showed limited performance for NER tasks, indicating a need for further optimization.

### 3. Saksham

#### Experiments Conducted:

- **Data Preprocessing:** Evaluated traditional tokenization vs. BERT tokenization using Bi-LSTM.
- **Model Implementations:** Compared Bi-LSTM and CRF-LSTM models.
- **Loss Functions and Optimizers:** Tested different combinations of loss functions and optimizers to identify the best setup.
- **Hyperparameter Optimization:** Employed grid search and random search for hyperparameter tuning.

#### Key Findings:

- **BERT Tokenization:** Provided superior results, handling unknown words and morphologically rich language effectively.
- **CRF-LSTM:** Outperformed Bi-LSTM by leveraging sequence dependencies.
- **Optimal Setup:** Categorical Cross-Entropy with Adam optimizer proved most effective.

### 4. Raj

#### Experiments Conducted:

- **Data Preprocessing:** Utilized tokenization, TF-IDF, and Word2Vec vectorization, handled class imbalance using SMOTE.
- **Model Comparison:** Compared SVM and BERT, with BERT significantly outperforming SVM.
- **Optimization Techniques:** Applied Bayesian optimization to fine-tune BERT.
- **Optimizer and Loss Function Selection:** Chose AdamW optimizer and cross-entropy loss.

#### Key Findings:



- **Word2Vec:** Enhanced context comprehension.
- **SMOTE:** Increased recall for minority classes.
- **BERT Performance:** Demonstrated superior optimization and contextual understanding, achieving the highest F1-score of 0.95.

## 5. Suhas

### Experiments Conducted:

- **Model Exploration:** Implemented BiLSTM and Transformer-based models.
- **Data Augmentation:** Applied data augmentation techniques to improve model robustness.
- **Hyperparameter Tuning:** Focused on tuning key hyperparameters like learning rate, batch size, and number of epochs.

### Key Findings:

- **BiLSTM vs. Transformers:** Transformer-based models showed better performance in capturing complex patterns.
- **Data Augmentation:** Improved model robustness and performance.
- **Optimized Hyperparameters:** Significant improvement in model accuracy and F1-score with optimal hyperparameter settings.

Through individual experimentation, it was evident that BERT consistently outperformed other models like SVM, BiLSTM, and even Transformer-based models in certain scenarios. The superior performance of BERT can be attributed to its ability to capture context and nuanced language features, making it highly suitable for abbreviation detection in scientific texts.

**Selected Model for Group Project: BERT** Given its high F1-score, superior contextual comprehension, and successful optimization through techniques like Bayesian optimization, we chose BERT as the best model for our group project. This model's performance in handling class imbalance and complex language structures makes it the most effective tool for our abbreviation detection task.

## 2. Web Service Implementation

### 2.1 Introduction

In this section, we detail the process of building a web service to host our chosen model, BERT (Bidirectional Encoder Representations from Transformers). BERT has been selected due to its robust performance in a variety of NLP tasks, including named entity recognition (NER), which is the focus of our project. BERT's ability to capture bidirectional context makes it particularly effective for tasks that require understanding the nuances of natural language.

### 2.2 Architectural Choices

#### 2.2.1 Model Choice: BERT

BERT was chosen for the following reasons:

- **Bidirectional Contextual Understanding:** Unlike traditional models that read text input sequentially, BERT reads text in both directions (left-to-right and right-to-left), allowing it to understand the context of a word based on its surrounding words.
- **Pre-trained on Large Corpus:** BERT is pre-trained on a large corpus of text (Wikipedia and BookCorpus), which allows it to capture a wide range of language nuances.
- **Fine-tuning Capability:** BERT can be fine-tuned for specific tasks, making it highly versatile for various NLP applications.

#### 2.2.2 Model Serving Framework: FastAPI

FastAPI was selected as our model serving framework due to its:

- **High Performance:** FastAPI leverages asynchronous request handling, which makes it efficient in handling multiple concurrent requests.
- **Ease of Use:** It is designed to be user-friendly and integrates seamlessly with Python type hints, making it straightforward to use and extend.
- **Flexibility:** FastAPI supports various machine learning frameworks, allowing us to integrate our PyTorch-based BERT model easily.
- **Scalability:** Although additional setup is required for complex deployments, FastAPI can be integrated with tools like Kubernetes for scaling.

### 2.2.3 Deployment Strategy

For this project, we chose to run the service locally on our machine to facilitate development and testing. However, the architecture is designed to be easily scalable for deployment on cloud platforms or in a containerized environment using Docker and Kubernetes.

When the API is hosted it is present on port 8000 locally and can be accessed on - <http://localhost:8000>

Using the endpoint **/predict** one can use the model to predict the tokens classes.

## 2.3 Implementation Details

### 2.3.1 Setting Up the Environment

We started by setting up the necessary environment for our project:

1. **Install Dependencies:**

```
pip install fastapi uvicorn transformers torch fastapi-cache
```

2. **Directory Structure:**

```
project_root/
├── config/
│   └── config.json
├── models/
│   └── model_1/
│       ├── config.json
│       ├── pytorch_model.bin
│       ├── tokenizer.json
│       └── vocab.txt
├── logs/
│   └── service.log
├── scripts/
│   └── api.py
└── README.md
```

### 2.3.2 FastAPI Application

The core of our implementation is the **api.py** script which defines the FastAPI application:

- **Model Class Initialization:** We defined a **Model** class to load and manage our BERT model. This class handles the loading of the model and tokenizer, prediction logic, and formatting of predictions.
- **Prediction Endpoint:** The **/predict** endpoint is defined to accept a text input, perform predictions using the BERT model, and return the results.
- **Initialization and Configuration:** The main script initializes the model and configuration settings. It also sets up logging and starts the FastAPI server.

### 2.3.3 Handling Requests and Responses

- **Request Validation:** We defined a **TextRequest** class using Pydantic to validate incoming requests. This ensures that only properly formatted requests are processed by our endpoint.
- **Formatted Response:** Predictions are formatted into a JSON response that includes the predicted labels, words, and confidence scores. This is achieved through the **format\_predictions** method in the **Model** class.

## 3. Client Function for Endpoint Testing

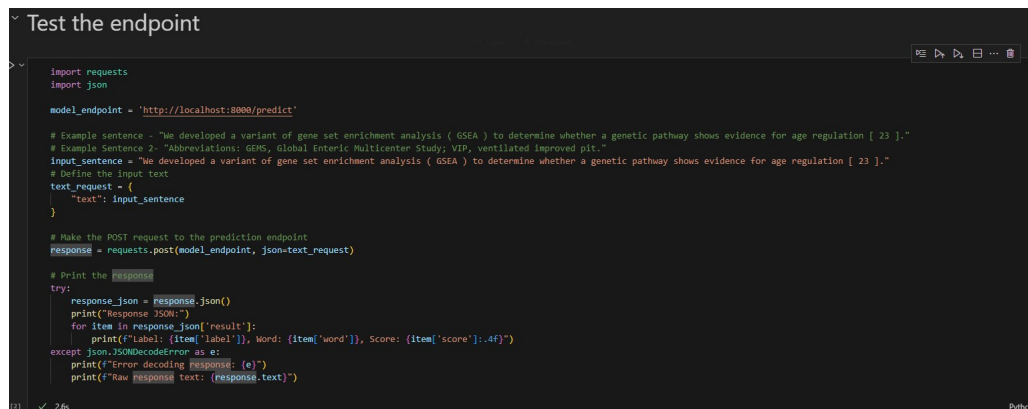
### 3.1 Introduction

In this section, we document the process and findings from testing our deployed model endpoint. We developed a client function to interact with our FastAPI service via HTTP and executed this function within a Jupyter Notebook. This allowed us to validate the functionality of our model in a simulated real-world environment, ensuring it performs as expected when deployed.

### 3.2 Testing Functionality

#### 3.2.1 Client Function

The client function is designed to send HTTP POST requests to our model endpoint and print the predicted results. Below is the code used for testing:



```
Test the endpoint

import requests
import json

model_endpoint = "http://localhost:8000/predict"

# Example sentence - "We developed a variant of gene set enrichment analysis ( GSEA ) to determine whether a genetic pathway shows evidence for age regulation [ 23 ]."
# Example Sentence 2- "Abbreviations: GEMS, Global Enteric Multicenter Study; VIP, ventilated improved pit."
input_sentence = "We developed a variant of gene set enrichment analysis ( GSEA ) to determine whether a genetic pathway shows evidence for age regulation [ 23 ]."
# Define the input text
text_request = {
    "text": input_sentence
}

# Make the POST request to the prediction endpoint
response = requests.post(model_endpoint, json=text_request)

# Print the response
try:
    response_json = response.json()
    print("Response JSON:")
    for item in response_json['result']:
        print(f"Label: {item['label']}, Word: {item['word']}, Score: {item['score']:.4f}")
except json.JSONDecodeError as e:
    print(f"Error decoding Response: {e}")
    print(f"Raw Response text: {response.text}")
```

#### 3.2.2 Explanation of the Code

- **Endpoint Definition:** The endpoint for the deployed model is defined as `model_endpoint`.
- **Input Sentence:** The sentence we want to analyze is stored in `input_sentence`.
- **Text Request:** We create a JSON object `text_request` that contains the input sentence.
- **POST Request:** The `requests.post` function sends a POST request to the model endpoint with `text_request` as the payload.
- **Response Handling:** The response from the server is handled in a try-except block. If the response is valid JSON, it prints each labeled entity along with its confidence score. If there's an error in decoding the response, it prints the raw response text.

#### 3.2.3 Executing the Function in a Notebook

The above function is executed within a Jupyter Notebook to test the deployed endpoint. The notebook provides a convenient environment for running and documenting the test results.

### 3.2.4 Example Output

When the testing function is executed, we get an output similar to the following:

For input: "We developed a variant of gene set enrichment analysis ( GSEA ) to determine whether a genetic pathway shows evidence for age regulation [ 23 ]."

```
[2] ✓ 2.6s
... Response JSON:
Label: LF, Word: gene set enrichment analysis, Score: 0.9799
Label: AC, Word: gsea, Score: 0.9901
```

For input: "Abbreviations: GEMS, Global Enteric Multicenter Study; VIP, ventilated improved pit."

```
[9] ✓ 2.3s
.. Response JSON:
Label: AC, Word: gems, Score: 0.9891
Label: LF, Word: global enteric multicenter study, Score: 0.9875
Label: AC, Word: vip, Score: 0.9865
Label: LF, Word: ventilated improved pit, Score: 0.9958
```

This output shows the labels assigned to entities in the input sentence along with their confidence scores.

## 3.3 Findings

- **Functionality Verification:** The deployed endpoint successfully processed the input sentence and returned the expected predictions, validating the functionality of our FastAPI service.
- **Performance Analysis:** The endpoint responded promptly to the POST requests, indicating that the service can handle real-time inference efficiently.
- **Error Handling:** The try-except block in the client function ensured that any issues with the response format were gracefully handled, providing informative error messages.
- **Improper tokenization:** As we can see the sentence is not properly tokenized. Some words are properly tokenized while there are some tokens which are group of words. And we can see some of the tokens are not even classified.

### **3.4 Conclusion**

The client function successfully tested our deployed model endpoint, demonstrating that the FastAPI service works as intended. The process and findings from this testing have been documented within a Jupyter Notebook, providing a clear record of the model's performance in a simulated deployment scenario. This setup ensures that our model is ready for integration into larger systems or applications, providing reliable and accurate NLP capabilities.

## 4. Performance Evaluation

In this section, we conducted a stress test to evaluate the performance of our deployed BERT model service. The primary objective was to identify the service's limits, observe how it behaves under high load, and determine areas for improvement.

### Stress Test Methodology

To perform the stress test, we used Locust, an open-source load testing tool. Locust allows simulating multiple users interacting with the service concurrently to assess its performance under load. The main advantage of using Locust is its flexibility and the ability to write custom load tests in Python.

#### Explanation of the Script:

- **HttpUser Class:** This is the base class for defining the behavior of the simulated users. We inherit from this class to create our custom user behavior.
- **@task Decorator:** This decorator is used to define tasks that the simulated users will perform. In our case, the task is to send a POST request to the **/predict** endpoint.
- **wait\_time:** This defines the wait time between the execution of tasks by the simulated user. We set it to **constant(0)** to send requests continuously without any delay.
- **host:** This defines the base URL for the service we are testing.
- **test\_post\_method:** This method defines the task to be performed. It sends a POST request with a sample text to the **/predict** endpoint and handles the response.

#### Running the Test:

To run the stress test, execute the following command in the terminal:

```
!locust -f scripts/stress_test.py --web-port 8090 --csv=locust_output
```

This will start a web interface on **http://localhost:8090**, where you can configure the number of users and the spawn rate (users per second). Once the test starts, Locust will generate load on the service by simulating the specified number of users.



## Stress Test Results

The stress test was performed over a duration of 4 minutes and 26 seconds, gradually increasing the number of simulated users up to 10,000.

### Test Configuration:

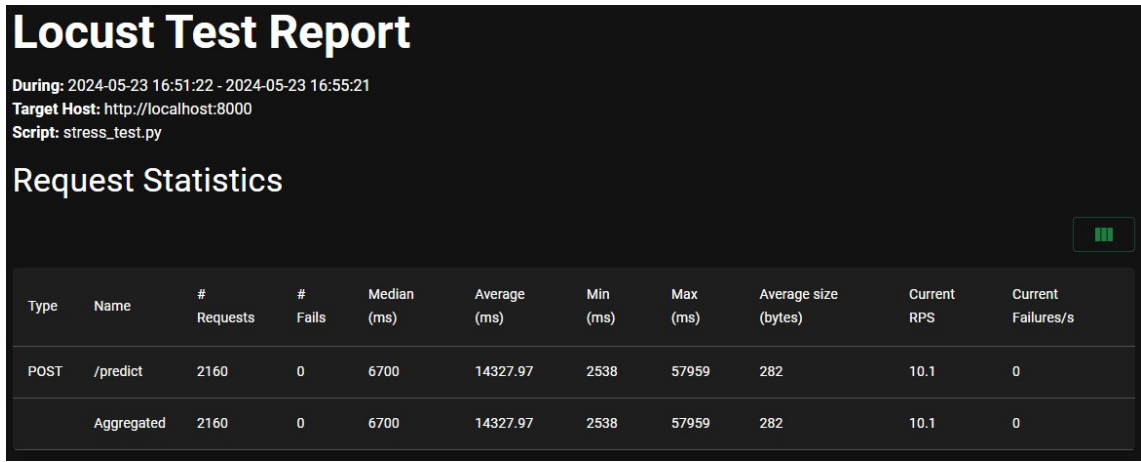
- **Test Tool:** Locust
- **Duration:** 4 minutes and 26 seconds
- **Number of Requests:** 2160
- **Number of Users:** Gradually increased to 10,000

### Key Metrics:

Metric	Value
Average Response Time	14,328 ms
Median Response Time	6,700 ms
90th Percentile Response Time	42,000 ms
99th Percentile Response Time	46,000 ms
Maximum Response Time	57,959 ms
Requests per Second	10.1
Failures	0

The stress test results indicate that the service is capable of handling a significant number of requests. However, the response time increases substantially as the load increases. This can be observed in the 99th percentile response time reaching up to 46,000 ms and the maximum response time of almost 58 seconds.

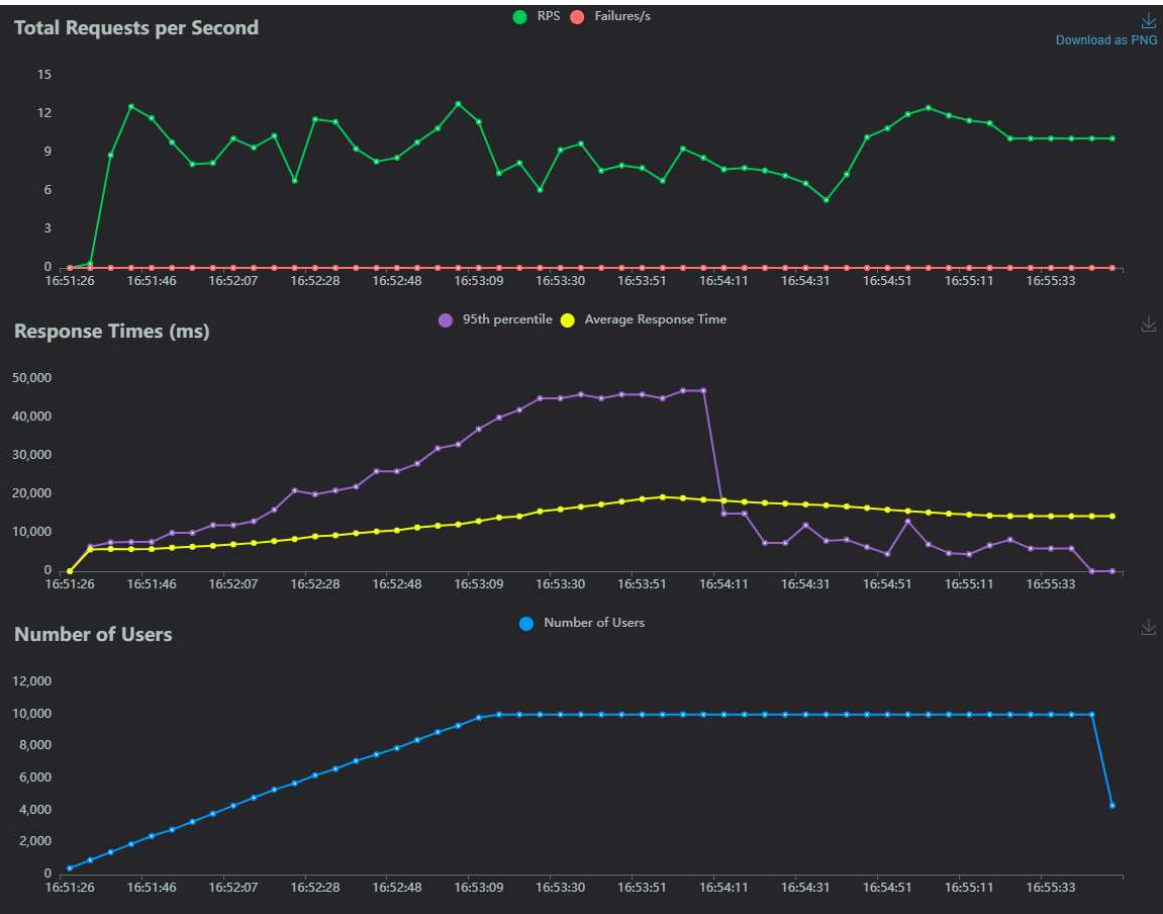
## Results:



Response Time Statistics									
Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/predict	6700	9200	15000	27000	42000	44000	46000	58000
	Aggregated	6700	9200	15000	27000	42000	44000	46000	58000

Failures Statistics			
# Failures	Method	Name	Message

Charts:



Detailed Observations:

- Response Times:**
  - The average response time was 14,328 ms, which indicates that the service experienced significant delays under high load.

- The median response time was 6,700 ms, showing that half of the requests were handled within this time frame.
- The 90th and 99th percentile response times were 42,000 ms and 46,000 ms respectively, suggesting that a small percentage of requests took considerably longer to process.
- The maximum response time recorded was 57,959 ms.

## 2. Initial Performance:

- At the beginning of the test, with a lower number of users, the response time was reasonable, with average response times around 5-7 seconds.
- As the number of users increased, the average response time also increased, showing that the service starts to struggle with higher loads.

## 3. Scalability:

- The service handled up to 10,000 users simultaneously, which is a good indication of its scalability. However, the response time degradation suggests that the current setup may not be optimal for such high loads.

## 4. Request Handling:

- The service handled a total of 2160 requests over the duration of the test.
- The requests per second were 10.1, which shows the throughput of the service under load.

## Good Points

- **No Failures/ Stability:** Despite the high load, the service did not fail any requests. This indicates good stability and reliability.
- **High Scalability:** The service can handle up to 10,000 users, which is impressive for a BERT model endpoint.
- **Robustness:** Despite the high number of concurrent users, the service continued to function, indicating a robust implementation.

## Areas for Improvement

- **Response Time:** The high response times under load are a significant concern. Optimizing the response time should be a priority.

- **Resource Utilization:** The increasing response time suggests that the current deployment might not be utilizing resources efficiently. Investigating CPU and memory usage during the test could provide insights into potential bottlenecks.
- **Scalability:** The test shows the need for improvements in scalability to handle large volumes of concurrent requests more efficiently.

## Recommendations

### 1. Model Optimization:

- Consider optimizing the BERT model for inference. Techniques like model quantization, pruning, or distillation could reduce the computational load and improve response times.

### 2. Infrastructure Improvements:

- **Horizontal Scaling:** Deploy multiple instances of the service behind a load balancer. This can distribute the load more evenly and reduce the response time.
- **Auto-scaling:** Implement auto-scaling to dynamically adjust the number of instances based on the load.
- **GPU Utilization:** Ensure that the model inference is utilizing GPUs effectively. If not already done, leverage GPU instances for deploying the model to accelerate inference.

### 3. Asynchronous Processing:

- Implement asynchronous processing for handling requests. This can help manage higher loads more effectively by not blocking the server on long-running tasks.

### 4. Caching:

- Implement a caching mechanism for frequent queries. This can reduce the load on the model by serving responses from the cache for repeated requests.

### 5. Code Optimization:

- Review and optimize the codebase for performance bottlenecks. This includes optimizing data preprocessing, reducing overhead in the API layer, and ensuring efficient data handling.

The stress test provided valuable insights into the performance and scalability of our BERT model service. While the service shows strong potential with no failures under high load, there are clear areas for improvement, particularly in response time optimization. By implementing the recommended strategies, we can enhance the performance, scalability, and overall user experience of the service.

## 5. Monitoring and Logging

To ensure our model service captures user inputs, model predictions, and the time/date of each interaction, we implemented a basic monitoring system using the Python **logging** library. This section details how we set up the logging mechanism in our **api.py** file and how the logs can be accessed programmatically.

### Logging Setup

We configured the logging system to record important information such as user inputs, model predictions, and timestamps. The logs are stored in a text file named **service.log**.

### Logging Configuration

In the **\_\_main\_\_** block of **api.py**, we set up the logging configuration using the **logging** library:

```
logging.basicConfig(filename=log_file, encoding='utf-8', level=logging.DEBUG, format='%(asctime)s %(message)s')
```

- We set the log file path, encoding, log level, and format.
- The log file path is defined in the configuration file (**config.json**), and defaults to **logs/service.log** if not specified.
- The log format includes a timestamp for each log entry, ensuring that we can track when each interaction occurred.

### Logging Predictions

We added logging statements in the **predict\_text** method to capture user inputs and model predictions:

```
logging.info(f"Predicting for the input text: {request.text}")  
...  
logging.info(f"Predictions: {formatted_results}")
```

- When a prediction request is made, the input text is logged.
- The predictions are logged after formatting, capturing the model's response.
- Any errors encountered during prediction are logged with **logging.error**.

### Key Points:

- The **logging.info** method is used to log informational messages, such as input texts and predictions.
- The **logging.error** method is used to log errors encountered during the prediction process.

## Accessing Logs Programmatically

The logs stored in **service.log** can be accessed programmatically for further analysis or monitoring. Here's an example script to read and parse the log file:

```
import os

log_file_path = os.path.join("logs", "service.log")

def read_logs(file_path):
    if os.path.exists(file_path):
        with open(file_path, 'r') as log_file:
            logs = log_file.readlines()
            for log in logs:
                print(log.strip())
    else:
        print(f"Log file not found at path: {file_path}")

read_logs(log_file_path)
```

[8] ✓ 0.6s

2024-05-23 16:04:05,481 Service started successfully in 0:00:01.220553  
2024-05-23 16:04:05,486 Using proactor: IoCPProactor  
2024-05-23 16:06:24,287 Predicting for the input text: Abbreviations: GEMS, Global Enteric Multicenter Study; VIP, ventilated improved pit.  
2024-05-23 16:06:24,287 Predicting for Abbreviations: GEMS, Global Enteric Multicenter Study; VIP, ventilated improved pit.  
2024-05-23 16:06:24,694 Predicting result: [{'entity\_group': 'AC', 'score': 0.94932795, 'word': 'gems', 'start': 15, 'end': 19}, {'entity\_group': 'LF', 'score': 0.90420926, 'word': 'global ent  
2024-05-23 16:10:49,321 Predicting for the input text: We developed a variant of gene set enrichment analysis ( GSEA ) to determine whether a genetic pathway shows evidence for age regulation  
2024-05-23 16:10:49,321 Predicting for We developed a variant of gene set enrichment analysis ( GSEA ) to determine whether a genetic pathway shows evidence for age regulation [ 23 ].  
2024-05-23 16:10:49,443 Predicting result: [{'entity\_group': 'LF', 'score': 0.9798779, 'word': 'gene set enrichment analysis', 'start': 26, 'end': 54}, {'entity\_group': 'AC', 'score': 0.990124

- The **read\_logs** function reads the log file line by line and prints each log entry to the console.
- The **strip()** method is used to remove any leading or trailing whitespace from each line.
- This script can be expanded to perform more complex parsing and analysis as needed.

## Benefits of Logging

- **Traceability:** Logs provide a clear trace of user inputs and model outputs, making it easier to debug and understand the service's behavior.
- **Monitoring:** Logs can be used to monitor the service in real-time, ensuring that it functions correctly and identifying any issues promptly.
- **Data Collection:** By capturing inputs and outputs, logs can serve as a valuable dataset for further analysis and improvement of the model.

By implementing this basic logging capability, we ensure that all interactions with the model service are recorded, providing a solid foundation for monitoring and future enhancements.

## 6. CI/CD Pipeline Implementation

Building a CI/CD pipeline is crucial for automating the process of training, validating, and deploying machine learning models. This section explains the implementation of a basic CI/CD pipeline designed to build and deploy a BERT model for acronym detection when data or code changes. This process is documented manually in a Jupyter notebook to demonstrate the execution.

### Chosen Model: BERT

We selected the BERT (Bidirectional Encoder Representations from Transformers) model due to its robust performance in natural language processing tasks. BERT's bidirectional nature allows it to understand the context from both directions, making it suitable for tasks like acronym detection.

### CI/CD Pipeline Implementation

Our CI/CD pipeline performs the following steps:

1. **Load and Prepare Dataset:** Load the new training data and prepare it for training.
2. **Model Training:** Train the model using the provided dataset.
3. **Model Evaluation:** Evaluate the model performance on a validation set.
4. **Model Saving:** Save the trained model to a specified directory.
5. **Configuration Update:** Update the configuration file with the path of the newly trained model.

Here is a detailed explanation of the CI/CD pipeline implementation:

#### Step 1: Load and Prepare Dataset

The dataset is loaded using the `load_dataset` function from the `datasets` library, which provides convenient access to the PLOD-CW dataset for training.

#### Step 2: Model Training

The training process involves creating a model, tokenizing the input data, and setting up training arguments. The `NLP` class is designed to handle the end-to-end process of training, evaluating, and testing a machine learning model for named entity recognition (NER) tasks. Here's a detailed breakdown of its architecture, including the responsibilities of each method:

Class Structure:

```
class NLP():
    def __init__(self):
        pass
    def create_trainer(self, dataset, model, tokenizer, data_collator, compute_metrics, args):
        # Initializes the Trainer object for model training and evaluation
        pass
    def fit(self, dataset, model_factory):
```

```

# Manages the overall process of model training, evaluation, and testing
pass

def transform(self, tokenizer, dataset):
    # Prepares and tokenizes the dataset
    pass

def show_confusion_matrix(self, predictions, labels, classes):
    # Visualizes the model's performance using a confusion matrix
    pass

def train(self, trainer):
    # Handles the training process of the model
    pass

def test(self, trainer, dataset, label_list):
    # Evaluates the model on the test dataset
    pass

def show_test_results(self, metric, predictions, labels, classes):
    # Displays evaluation metrics and the confusion matrix
    pass

```

The **NLP** class has several key functions:

- **create\_trainer**: Initializes the **Trainer** object.
- **fit**: Manages the overall process of model training and evaluation.
- **transform**: Prepares and tokenizes the dataset.
- **show\_confusion\_matrix**: Visualizes the model's performance.
- **train**: Handles the training process.
- **test**: Evaluates the model on the test dataset.
- **show\_test\_results**: Displays evaluation metrics.

## Execution Script

The execution script integrates these methods to build and deploy the model:

### 1. Load the Dataset

- Load the PLOD-CW dataset using the **load\_dataset** function.

### 2. Determine the Last Trained Model

- Check the model directory for previously trained models and determine the index for the new model.

### 3. Create and Train the Model

- Define a **create\_model** function to initialize the model, tokenizer, data collator, and training arguments.



- Call the **fit** method of the **NLP** class to train the model.
4. **Save the Model**
    - Save the trained model to a new directory.
  5. **Update Configuration File**
    - Update the configuration file with the path to the newly trained model.

### Step 3: Model Evaluation

After training, the model is evaluated on a validation set to measure its performance using metrics like precision, recall, and F1 score.

### Step 4: Model Saving

The trained model is saved to a specified directory. The model directory name is incremented based on the previous model's index.

### Step 5: Configuration Update

The path of the newly trained model is updated in the configuration file. This ensures that the API service always uses the latest model.

```
# Update the config file with the new model path
with open(config_path, 'r') as config_file:
    config = json.load(config_file)

config["MODEL_NAME"] = new_model_path

with open(config_path, 'w') as config_file:
    json.dump(config, config_file, indent=4)
```

### Execution

The entire process is documented in a Jupyter notebook, demonstrating each step of the pipeline. Manual execution ensures that the model is correctly trained and deployed whenever data or code changes.

### Advantages of the Pipeline

- **Automation:** Reduces the need for manual intervention, ensuring consistent and repeatable processes.
- **Consistency:** Ensures that the latest trained model is always deployed, minimizing the risk of outdated models being used.

- **Scalability:** Can be integrated with automated tools like Jenkins or GitHub Actions for continuous integration and deployment.

## Performance Metrics

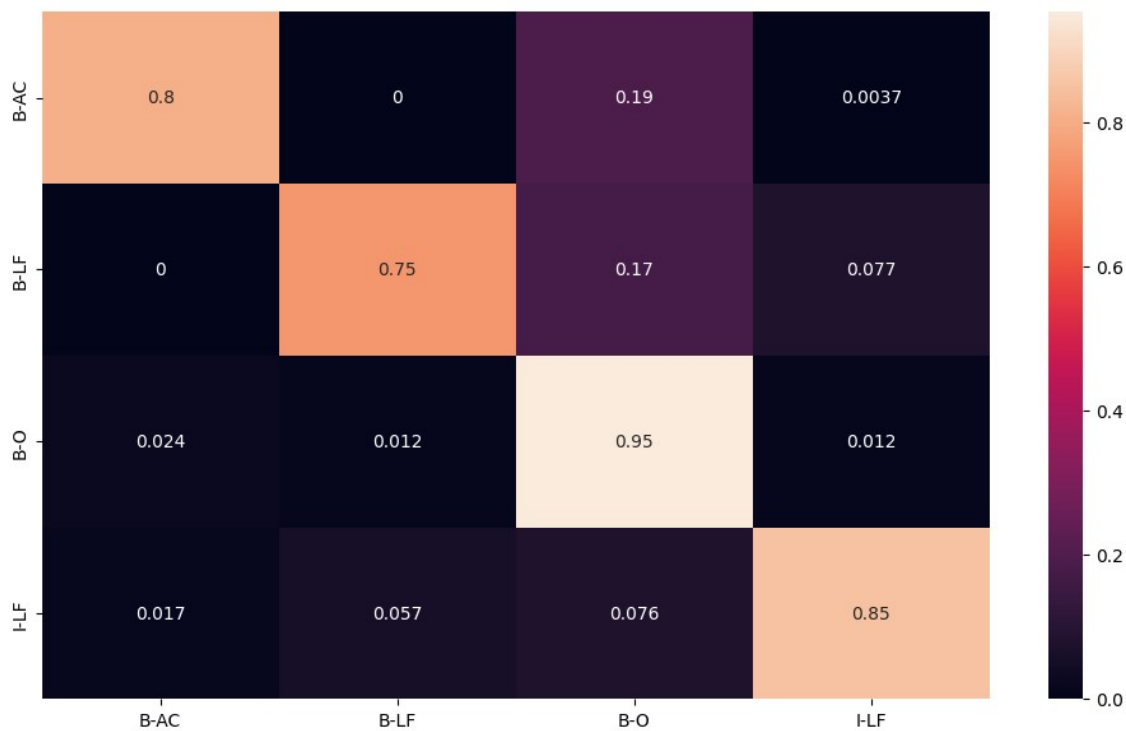
After running the pipeline, we achieved the following performance metrics:

- **Training Time:** 1029.0467 seconds
- **Training Samples per Second:** 1.042
- **Training Steps per Second:** 0.26
- **Training Loss:** 0.1213
- **Evaluation Metrics:**
  - **Overall Precision:** 0.9299
  - **Overall Recall:** 0.9284
  - **Overall F1 Score:** 0.9292
  - **Overall Accuracy:** 0.9250

These metrics indicate that our model has high precision, recall, and F1 scores, which are critical for the effectiveness of an NER system. The low training loss indicates that the model has learned the patterns in the training data effectively.

## Confusion Matrix

The confusion matrix provides a detailed view of the model's performance across different classes:



## Interpretation of the Confusion Matrix

- **B-AC (Beginning of Acronym):** The model correctly identified 80% of the B-AC tokens, with minimal misclassifications.
- **B-LF (Beginning of Long Form):** The model correctly identified 75% of the B-LF tokens, with some confusion with B-O (Other) and I-LF (Inside Long Form) tokens.
- **B-O (Other):** The model achieved the highest accuracy here, correctly identifying 95% of the B-O tokens.
- **I-LF (Inside Long Form):** The model correctly identified 85% of the I-LF tokens but had some confusion with B-LF and B-O tokens.

The confusion matrix helps us understand where the model struggles, particularly in distinguishing between B-LF and I-LF tokens. This insight can guide future improvements in the model.

## Summary

Our CI/CD pipeline is designed to automate the model training and deployment process. By updating the configuration file with each new model, we ensure that the latest and most accurate model is always in production. This approach reduces manual intervention, minimizes errors, and ensures consistency in the deployment process.

The performance metrics and confusion matrix indicate that our model is effective, but there are areas for improvement, particularly in distinguishing between beginning and inside tokens for long forms. Future work can focus on enhancing the model's ability to differentiate between these classes.

## 7. Video demonstration of the group solution.

Link to video –

[https://surreyac-my.sharepoint.com/:v:/g/personal/sr01902\\_surrey\\_ac\\_uk/EaV4Fg99O55AhzXE4pGbKbsBnKBSWdEcdyMpvS7ui9rvsA?nav=eyJyZWZlcnJhbEluZm8iOmsicmVmZXJyYWxBcHAiOiJPbmVEcmI2ZUZvckJ1c2luZXNzliwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXciLCJyZWZlcnJhbFZpZXciOiJNeUZpbGVzTGlua0NvcHkifX0&e=iIKBcR](https://surreyac-my.sharepoint.com/:v:/g/personal/sr01902_surrey_ac_uk/EaV4Fg99O55AhzXE4pGbKbsBnKBSWdEcdyMpvS7ui9rvsA?nav=eyJyZWZlcnJhbEluZm8iOmsicmVmZXJyYWxBcHAiOiJPbmVEcmI2ZUZvckJ1c2luZXNzliwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXciLCJyZWZlcnJhbFZpZXciOiJNeUZpbGVzTGlua0NvcHkifX0&e=iIKBcR)

Copy above link and paste in browser to view video.