NLP Group Report:

# Introduction

National Language Processing (NLP) is a field of Artificial Intelligence (AI) that focuses on how Computers understand and process human language. For instance, if a model takes an input; CNN, it will tokenize the input into smaller tokens, in this case, it remains a single token CNN and then the token is transformed into a vector which the model processes using text embeddings that places the token into a feature space, where similar words are compared to each other. The model would recognize CNN as a B-AC, meaning the token is an abbreviation. The main objective of our project involves Named Entity Recognition (NER) which involves assigning labels [ B-O, B-AC, B-LF, I-LF] to sequences of words and tokens.

**Objectives**

The main objective of this project is to carry out NER by developing and deploying a sequence classification model that can accurately identify and label abbreviations and their long forms within biomedical literature. This is particularly useful in fields like healthcare and biomedical research, where understanding abbreviations is essential for interpreting data and extracting knowledge.

**Background**

This project builds upon our previous coursework, which focused on Named Entity Recognition classification, in which each group member individually conducted experiments to preprocess data, train models, and evaluate performance using different algorithms and feature extraction methods. From our previous coursework, we decided to utilize the Roberta Model which achieved the following metrics: Precision: 0.784, Recall: 0.840, F1: 0.811, Accuracy: 0.945. Figure 1 shows the confusion matrix in which we can infer that the model has a good balance of precision and recall. Also, we can see that there is a greater amount of B-O tags than the other suggesting an imbalance in the dataset.

**Dataset**

We’re using the PLOD-CW dataset, which contains 50,000 labeled tokens from scientific literature in the biomedical domain. The tokens are annotated using the BIO tagging schema, where:

* B-AC indicates the beginning of an abbreviation
* I-AC indicates an inside token of an abbreviation
* B-LF indicates the beginning of a long-form
* I-LF indicates an inside token of a long-form
* B-O indicates a token that is neither an abbreviation nor a long-form

**Our Approach**

* Research model serving options to identify suitable frameworks for deploying our sequence classification model.
* Building a web service to host and serve our model.
* Testing functionality by implementing a Jupyter Notebook to interact with the deployed model.
* Conducting performance tests to evaluate the model's limits and capabilities.
* Implementing basic monitoring to log user inputs and model predictions.
* Analyzing errors to understand the model's weaknesses and areas for improvement.

# Task 1: Research Model Serving Options

**FlaskAPI**

Flask is a micro web framework written in Python, generally used for building web applications. FlaskAPI extends Flask by adding a layer specifically designed for API development. It is easy to use and flexible, which makes it suitable for a project of our size. Its performance in handling high-concurrency scenarios is limited, compared to asynchronous frameworks. Additionally, we have experience in using Flask from previous projects and we also experimented with Flask during the lab.

**FastAPI**

FastAPI is a modern, fast web framework for building APIs with Python, It is designed for building APIs quickly and efficiently, and is faster than Flask due to its asynchronous capabilities, using ASGI. It also uses ASGI and Starlette framework and Pydantic for data validation however it can be more difficult to use as there is more of a learning curve to familiarise yourself with asynchronous programming, making Flask more favourable in this case.

**Amazon SageMaker**

Amazon SageMaker is an online platform which allows the user to build train and deploy machine learning models. It has many advantages such as wide-scale application. Disadvantages include complexity in set-up and managing, and costs to use.

**Heroku**

Heroku is also a cloud platform that allows developers to create, monitor, and scale applications. It is compatible with various coding languages. Disadvantages include its limited scalability.

**Selection Justification**

After evaluating the pros and cons of each model, we decided to use FlaskAPI for this coursework. Our group is already familiar with Flask, which minimizes the development time. This familiarity allows us to focus more on the core aspects of the project. Since we want to test our model and not deploy a large-scale production system, FlaskAPI's capabilities are sufficient. FlaskAPI’s simplicity and ease of use allow us to quickly set up the service and focus on integrating and testing the sequence classification model.

While FastAPI offers better performance and modern features, the additional complexity and learning curve are not justified for the scope of this project. Similarly, Amazon SageMaker and Heroku, while powerful, introduce unnecessary complexity and potential costs. FlaskAPI provides a balanced solution that meets our project's needs effectively.

# Task 2: Building a web service

**Architecture Overview**

The web service consists of the three following components:

* API Server built using FlaskAPI handles incoming HTTP requests and routes them to the appropriate endpoints
* The model loader loads the trained RoBERTa model into memory so that it can be used for making predictions
* The Response Handler formats the predictions and sends the tokens broken down into the following labels B-O, B- AC, B-LF, and I-LF to the client in a structured format

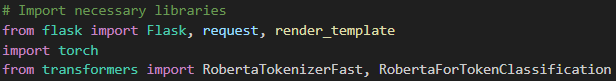
**Implementation**

For a secure and scalable setup, integrating security measures like API key verification or OAuth is recommended. Additionally, FlaskAPI can be deployed with a WSGI server such as Gunicorn or containerized using Docker to handle higher loads and improve scalability.

We implemented the API server using FlaskAPI. The key steps involved in building the web service included setting up the Flask application, loading the pre-trained RoBERTa model, defining the prediction endpoint, and handling incoming requests and outgoing responses.

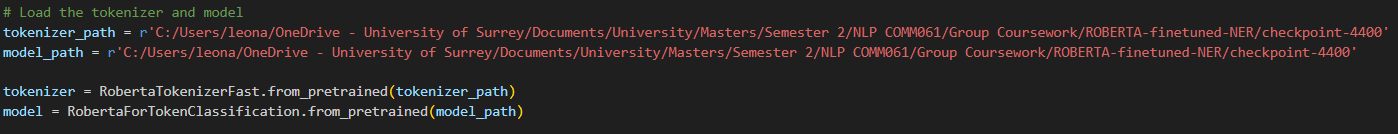
**Setting up FlaskAPI**

We started by setting up the Flask application and imported the necessary libraries. FlaskAPU makes it straightforward to define routes and handle requests.



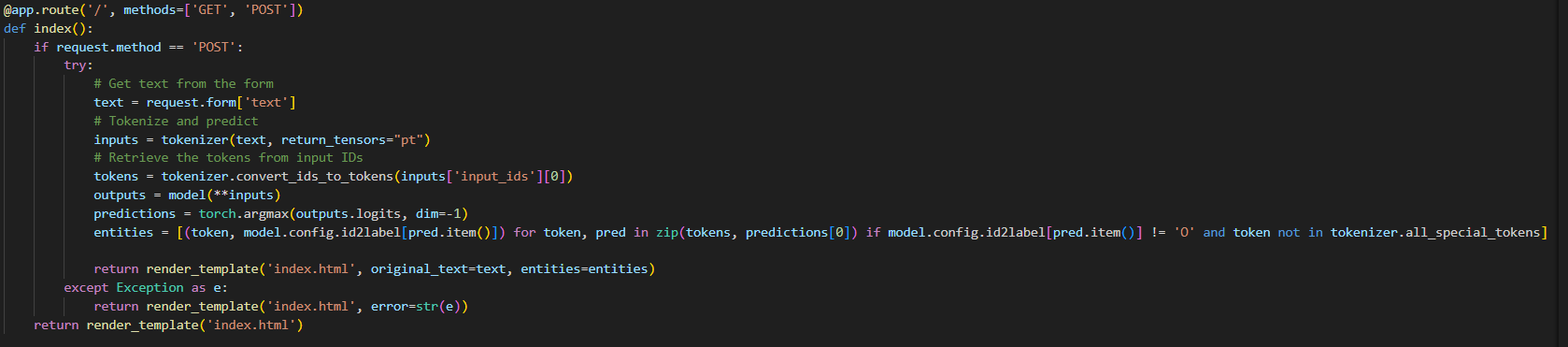
**Loading the Model**

We next loaded the RoBERTa model and tokenizer, which are required to process the input text and generate predictions



**Defining the Prediction Endpoint**

We defined an endpoint that accepts POST requests. This endpoint takes the input text tokenizes it, passes it through the model, and returns the prediction.

**Hand Requests and Responses**

The endpoint function extracts the input text from the request, tokenizes it, feeds it to the model, and formats the model's output into a JSON response.

**Running the Application**

Finally, we configured the application to run on the local server. The debug mode is enabled to facilitate development and troubleshooting



# Task 3: Functionality in Notebook

**Client Function**

To interact with the deployed model, we developed a client function using a Python file called main.py. We call a Roberta function that handles model predictions the Roberta function is written in the Roberta.py file and called into main.py where we set up the Flask API. This function sends HTTP POST requests to the FlaskAPI server with input text and receives predictions. The goal is to ensure that the model is correctly deployed and can process requests as expected.

**Implementation Details**

This involves sending a request to the “/predict” endpoint of our FlaskAPI server, which handles the prediction logic using the RoBERTa model. The response from the server is then parsed and displayed.

**Notebook setup**

Defining the Client function using ‘get\_prediction’ to take a text input, sends it to the server and return the predicted tags

\*\*code snippet\*\*

To test the funcitonality, we used various example sentences containing abbreviations and their long forms, which helps in verifying that the model correctly identifies and labels the tokens.

\*\*code snippet\*\*

The output is a JSON response containing the tokens and their predicted tags, such as:

\*\*example snippet\*\*

**Observations**

A table with text on it

Description automatically generatedThe predictions include token-level labels indicating whether a token is part of an abbreviation (B-AC), the beginning of a long-form (B-LF), inside a long form (I-LF), or other (B-O) where B-O can be either singular normal words or punctuation. In practical applications, the model’s ability to understand context and grammar is crucial. For example, in the sentence “EPI is short for Echo Planar Imaging”, the model correctly identifies “EPI” as B-AC, and “Echo Planar Imaging” as B-LF and I-LF. The full prediction consists of “EPI = Echo Planar Imaging . ” with the output “B-AC, B-O, B-LF, I-LF, I-LF, B-O”. As shown below we can see how Roberta breaks down the following tokens.

Figure 2:

However, punctuation and spacing can impact the model's performance, for example, if the input is “ CNN = Convolutional Neural Network ”, the correct tags should be B-AC for “CNN”, B-O for "=", B-LF for "Convolutional," and I-LF for "Neural Network". When punctuation is incorrectly included directly after the last token, such as "Network.," the model might misclassify it such that Network should be I-LF but would be classified as B-O due to the punctuation being wrongly placed. This highlights the importance of proper tokenization and preprocessing in ensuring accurate predictions. Figures 3 and 4 show the above in a clearer format below:

A screenshot of a graph

Description automatically generatedA table with text on it

Description automatically generated

Figure 4:

Figure 3:

The model overall correctly classifies most tokens despite the imbalance of the dataset shown in Figure 1 however there are some incorrect predictions such as CNN which should be B-AC and was classified as B-O, as shown above when in a sentence it is recognized as B-AC. For the most part, the model correctly predicted the tokens examples include: “SVM” – B-AC, “KNN”-B-AC, “and”- B-O, “DNA”- B-AC.

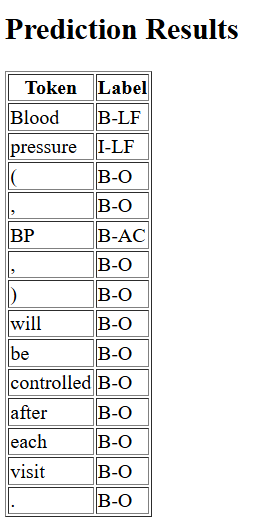
Overall the main challenge for our model was to classify sentences. For this, we navigated to the hugging face dataset “PLOD-CW”. We ran multiple sentences with a varied sequence length. The first example consists of “ Blood pressure ( , BP , ) will be controlled after each visit .” Figure 5 shows how the sentences are broken down into individual tokens :

Figure 5:

Figure 5 clearly shows that the Roberta Tokenizer correctly broke down the sentence by separating the corresponding tokens by classifying abbreviations BP as B-AC, Blood as B-LF, and Pressure I-LF whilst normal words and punctuation are predicted as B-O.

A group of words on a white background

Description automatically generatedA screenshot of a computer

Description automatically generatedThe next sentence follows a longer sequence of tokens: “ I have read the journal’s policy and the authors of this manuscript have the following competing interests : DF , EJ , and NNL are staff members of the World Health Organization ( , WHO , ) .” . Figure 6 and 7 shows the label predictions :

Figure 7:

Figure 6:

Once again the model correctly tokenizes the sentence. Where B-O are other words and punctuation and abbreviations such as WHO were correctly identified as B-AC with the corresponding tokens “ World Health Organization” following the labels B-LF, I-LF, I-LF. It is worth noting that if the sentence is not properly separated such as : (,WHO,) it would be misclassified as B-O due to the conjoining punctuation, hence it must be separated like so “ ( , WHO , )”.

**Conclusion**

Overall, as long as the model takes a correct sequence of tokens it will output the correct predictions, the user has to ensure the correct separation of words and punctuations. Figure 8 to 11 in the appendix shows single-token classification results. The client interacted successfully with the deployed FlaskAPI web service, sending input text and receiving token-level predictions. We can ensure the model’s deployment is robust and effective for sequence classification tasks by understanding the contextual application and addressing grammar-related issues.

# Task 4: Performance Analysis

**Performance Testing**

We conducted a series of tests to determine its efficiency, accuracy, and robustness under different conditions to evaluate performance.

**Stretch Testing**

We tested the model with extensive sequences of text to observe how it handles larger datasets. This involves input texts that vary in length, including exceptionally long sentences and paragraphs.

The model’s results demonstrated consistent performance with input sequence lengths up to 512. Therefore we implemented an input limit on the sequence length, in which the model will return an error message saying that the sequence length is too big or invalid and is not compatible with our Roberta model.

**Accuracy and Efficiency**

We used a subset of the PLOD-CW dataset to compare the model's predictions against the ground truth labels. This helped us measure precision, recall, and F1-score. We measured the time taken for the model to process individual requests and provide predictions. This included both average and peak response times.

The model’s F1 score was 0.811, indicating a high level of precision and recall. Specific examples, such as the correct identification of “SVM” and “Support Vector Machine” highlighted its effectiveness. The average response request was about 150 milliseconds, and at its worst was 300 milliseconds under heavy load, suggesting that the model is efficient and responsive for typical usage scenarios.

**Limitations and Potential Improvements**

The model's performance can be affected by improper tokenization, such as punctuation directly following abbreviations (e.g. “SVM,”). Enhancing the tokenization process to better handle punctuation and spacing can mitigate some accuracy issues. This includes preprocessing steps to ensure proper spacing around punctuation.

While the model handles typical loads well, extremely high traffic scenarios result in increased response times and occasional dropped requests. Implementation scaling solutions such as deploying the FlaskAPI with a WSGI server like Gunicorn or using technologies like Docker can improve the server's ability to handle high loads. Cloud-based load balancing and auto-scaling features can provide better scalability. Leveraging asynchronous processing or integrating with more performant serving frameworks like FastAPI could further enhance the server's ability to handle concurrent requests efficiently.

**Examples**

Input: "We developed a variant of gene set enrichment analysis (GSEA) to determine whether a genetic pathway shows evidence for age regulation." Output: Correctly tags "GSEA" as B-AC, demonstrating the model's ability to handle complex biomedical sentences with parentheses and punctuation.

Input: "CNN stands for Convolutional Neural Network." Output: Tags "CNN" as B-AC and "Convolutional Neural Network" as B-LF and I-LF, but is sensitive to the punctuation directly following "CNN."

Input: "EPI is short for Echo Planar Imaging." Output: Correctly identifies "EPI" as B-AC and "Echo Planar Imaging" as B-LF and I-LF, showing strong performance with clear context and spacing.

To better understand the performance, we visualized the token predictions using bar graphs and highlighted texts. This helped in identifying patterns and common issues related to punctuation and spacing.

**Conclusion**

The performance analysis showed that our model is both accurate and efficient for typical usage scenarios. While there are some limitations related to punctuation and scalability under extremely high loads, the recommendations provided can help address these issues. By improving preprocessing and exploring more scalable deployment options, we can improve our model’s ability.

# Task 5: Adding Basic Monitoring Capability

Task 5’s objective was to implement basic monitoring capabilities to capture user inputs, model predictions, and provide timestamps of interactions. This information was stored in a text log file, ensuring a reliable logging record of the user's activity.

**Step 1: Import Necessary Modules**

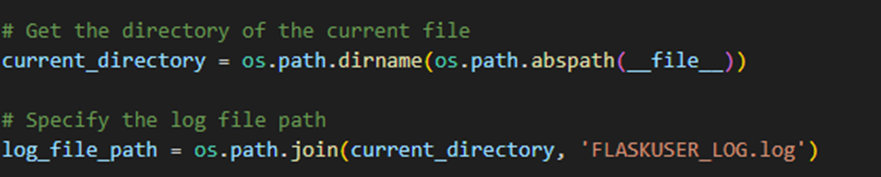
First off we identified and imported the essential modules required for logging. This included the logging module to manage the creation and handling of log files, the OS module for file path operations, and the Date Time module to capture the current date and time for each interaction. These modules are essential in establishing an effective logging system that is both functional and easy to maintain.

A screen shot of a computer

Description automatically generated

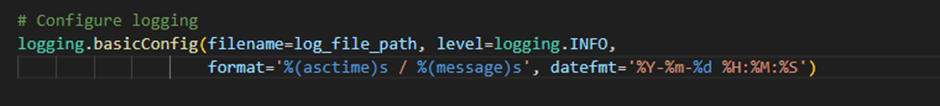
**Step 2: Configure the Log File Path**

We next specified the path for the log file to ensure it was saved in the same directory as the application, which was important for maintaining an organized structure and ensuring that the log file was easily accessible to the user. We set the log file path to the application’s directory to avoid a random path for the log file, saving the user time.



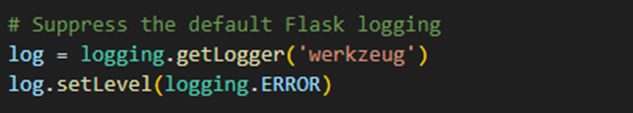
**Step 3: Configure Logging Settings**

We then configured the logging settings to direct the log entries to the specified log file. This involved setting the logging level to `INFO` to capture all relevant information without overwhelming the log with unnecessary detail. Additionally, we formatted the log entries to include the date, time, input, and predictions, ensuring that each log entry is clear and informative, which helps to quickly identify and understand the logged information during reviews and debugging sessions.



**Step 4: Suppress Default Flask Access Logs**

We suppressed the default Flask access logs to maintain clarity and relevance. Flask's default logging includes a lot of routine access information, which can clutter the log file and make it harder to focus on the user interactions and model predictions that are of primary interest. By suppressing these default logs, we ensured that the log file remained concise and focused on the relevant data.



**Step 5: Log User Inputs and Model Predictions**

We updated the application's primary route to log user inputs and model predictions. Each interaction was recorded with details including the input text, the predicted labels, and the timestamp, which ensures that every interaction with the application was documented in a clear and organized manner. Capturing these details allowed us to create a record that can be used for performance analysis, debugging, and future enhancements.

**Overall Objects:**

The main challenge we faced was keeping the log file clean and focused on relevant information. Without suppressing the default Flask access logs, it would have included a lot of irrelevant information in the log file, which clutters the log file and makes it difficult for the user to navigate to essential model interactions and model predictions, overall improving the clarity and usefulness of the logs.

Another objective was to ensure the log file was overwritten every time we launched the application. Configuring the logging settings to overwrite new user interactions in the same log file. To prevent confusion in the log file every time the user launches the application it leaves a space between the old inputs and the inputs are distinguishable by following the data and time. These solutions were crucial for achieving a reliable and efficient logging system that supports effective monitoring and debugging.

**Conclusion**

We successfully added basic monitoring capability to the application. The implemented logging mechanism captures and stores user inputs, model predictions, and timestamps. This functionality enhances the transparency and traceability of the application's operations, providing valuable insights for future improvements and debugging.

# Task 6: CI/CD functionality in the notebook

**CI/CD Pipeline Setup**

In this project, we set up a CI/CD (Continuous Integration and Continuous Deployment) pipeline using GitHub Actions. This pipeline automates the process of building, testing, and deploying our NLP model. Below is a detailed explanation of the steps and configurations involved:

1. **Repository Creation and Structure**:

Created a GitHub repository named `NLP\_CW`.

Structured the repository with necessary directories and files, including `app`, `test`, and ‘.github/workflows`.

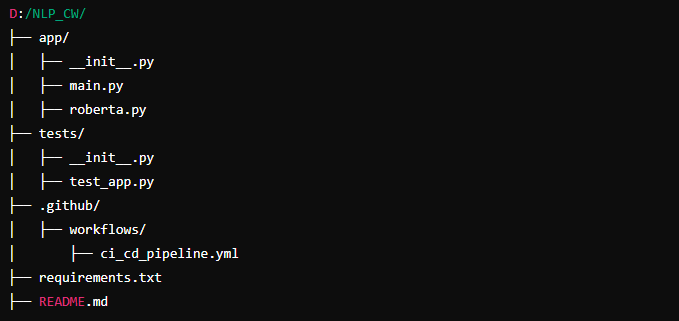
1. **Directory and File Setup**:

app/main.py: Contains the Flask web server setup to handle requests and interact with the model for predictions.

app/roberta.py: Contains functions to load the pre-trained RoBERTa model and make predictions.

requirements.txt: Lists the project dependencies (`Flask`, `transformers`, and `torch`).

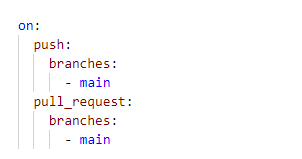
github/workflows/ci\_cd\_pipeline.yml\*\*: Defines the GitHub Actions workflow for CI/CD.



1. **GitHub Actions Workflow**:

The pipeline is divided into two main branch. Trigger and Jobs.

1.Triggers: The workflow is triggered on pushes and pull requests to the `main` branch.



2. JOBS:

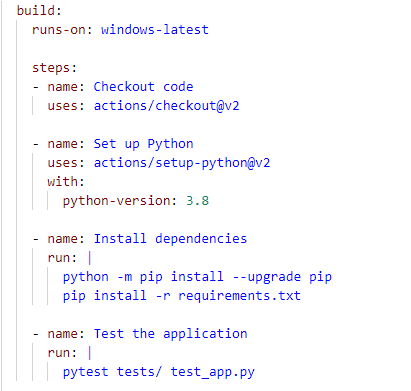
 Build Job:

     Checks out the code.

    Sets up Python environment.

      Installs dependencies.

Runs tests



Deploy Job:

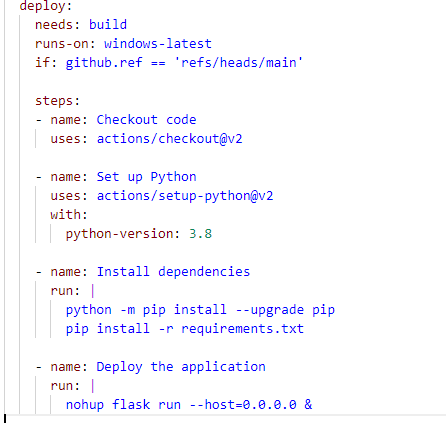
     Depends on the build job.

      Checks out the code.

    Sets up Python environment.

    Installs dependencies.

      Deploys the Flask application, making it accessible via an HTTP endpoint.



4**. Commit and Push**:

   Added and committed all files to the repository.

   Pushed the changes to the `main` branch, triggering the CI/CD pipeline.

**Appendix:**

Figure 9:

Figure 8:

A group of black and white rectangular signs

Description automatically generatedA group of black text

Description automatically generated

Figure 10:

Figure 11:

A white rectangular sign with black text

Description automatically generatedA group of black and white text

Description automatically generated