Sentiment Recognition Assessment

COMM061 - Group Coursework Report Template

Hung Le* - Yash Kulthe - Anitha Parameshwarappa - Moiz Nagaria - Abhishek Bhardwaj

May, 2023

Abstract

Sentiment recognition, a vital methodology in natural language processing (NLP), allows stakeholders to gain critical insights into individuals' emotions, opinions, and inclinations. This study examines different model serving options, identifying the optimal choice for the specific use case, and provides a step-by-step guide to building a web service, testing the deployed endpoint, and monitoring user inputs and predictions. Additionally, the paper explores the performance of the implemented service, discusses its strengths and limitations, and proposes recommendations for architectural improvements. Finally, the research outlines constructing a basic continuous integration and continuous deployment (CI/CD) pipeline for seamless model building and deployment upon data or code changes.

1 Introduction

Sentiment recognition, often called opinion mining or emotion artificial intelligence, constitutes a vital methodology within natural language processing (NLP). This technique emphasizes identifying and classifying emotions, opinions, and sentiments in textual data. By discerning the underlying sentiment of textual content, various stakeholders, including businesses, researchers, and individuals, can obtain critical insights into individuals' emotions, opinions, and inclinations. Consequently, these insights can contribute to more informed decision-making processes and enhance user experiences within various applications.

This paper presents a comprehensive approach to model serving, deployment, and monitoring for sentiment recognition applications, as described below:

- Propose a suitable model serving strategy for our methodology.
- Construct a web service incorporating the selected model serving approach, establish an endpoint for the model, and elucidate the architectural decisions.
- Design and implement testing capabilities within a computational notebook for the deployed endpoint.
- Deliberate on the strengths and limitations of the proposed approach.
- Develop a rudimentary monitoring system to document user inputs, model predictions, and associated timestamps in a text log file.
- Establish a fundamental continuous integration and deployment (CI/CD) pipeline to facilitate model updates as data or code modifications transpire.

By undertaking these measures, we can acquire insights into the efficacy of the selected model serving paradigm, the proficiency and extensibility of the web service architecture, the dependability and precision of the model endpoint, the merits and demerits of the overarching approach, the utility of monitoring systems for gauging performance and detecting potential concerns, and the advantages of employing a CI/CD pipeline for automating the update process. This acquired knowledge empowers us to make informed, data-driven determinations for optimizing and augmenting the deployed solution, ultimately catering to user requirements and preserving exceptional performance standards.

2 Group Declaration

The following is a list of the members comprising the team:

- Hung Le Nhat - hl01600@surrey.ac.uk - 6765820

- Moiz Saifuddin Nagaria - mn01030@surrey.ac.uk - 6782892
- Yash Mahesh Kulthe - yk00381@surrey.ac.uk - 6766477
- Anitha Kugur Parameshwarappa - ak02923@surrey.ac.uk - 6784786
- Abhishek Bhardwaj - ab03655@surrey.ac.uk - 6788899

In this study, we have chosen to use sentiment using a 13-label classification system, which allows for a more nuanced and detailed understanding of the emotions expressed in the text. The 13 labels selected for this project are:

Neutral	Surprise	Love	Fear
Joy	Gratitude	Approval	Disapproval
Confusion	Sadness	Desire	Optimism
Realization	Pride		

This multi-class classification approach enables a more comprehensive sentiment recognition by incorporating diverse emotions. The chosen labels cover a wide spectrum of emotions, ranging from positive (e.g., love, joy, gratitude) to negative (e.g., fear, sadness, disapproval), as well as those that are more complex or context-dependent (e.g., surprise, confusion, realization).

Then, it highlights the necessary steps for planning experiments, including defining the scope, data source, sample size, tools, techniques, sentiment recognition, and conclusion. The text also mentions using GitHub, Google Collab, and Dataset as common development environments. Lastly, the individual tasks are separated into data set preparations, algorithms, pre-trained models, setting up hyperparameters and experimental variations, and tracking progress.

This customized 13-label sentiment recognition will facilitate a deeper understanding of the underlying emotions present in text data, allowing for more informed decision-making and enhanced communication in various contexts, such as marketing, customer service, product development, and social media monitoring.

3 Member Approaches

3.1 Hung Le

In this section, we will discuss Hung Le's model; here is the detail of his model:

- **Model**: The model consists of an input layer, an embedding layer that maps the input tokens to 100-dimensional vectors, a 1D convolutional layer with 32 filters and kernel size 3, a max-pooling layer, an LSTM [4] layer with 128 units, a dense output layer with a softmax activation [8] function that outputs the predicted probability distribution over 14 classes.
- **Loss**: Sparse Categorical Crossentropy
- **Labels**: Category
- **Optimizer**: AdamW [7]
- **Learning Rate**: 1e-3
- **Weight Decay**: 1e-4 * Learning Rate
- **Epochs/ Batch-size**: 25 - 16
- **Training/Testing/Validating Accuracy**: 94.19% / 78.61% / 78.11%

The model appears to be performing reasonably well on the given classification task. The training accuracy of 94.19% indicates that the model fits the training data well. In contrast, the testing and validation accuracies of 78.61% and 78.11%, respectively, suggest that the model is generalizing reasonably well to unseen data.

3.2 Moiz Nagaria

- **Model**: Fine-tuning the model; RoBERTa [6] is a deep neural network for natural language processing based on the architecture of the BERT model [2]. It has 12 or 24 transformer layers [12], each with a self-attention mechanism and a feed-forward network. It also has an input embedding layer and an output prediction layer. RoBERTa [6] is a highly complex and powerful language processing model that has achieved state-of-the-art performance on various natural language processing tasks. We used this model with pre-trained weights.
- **Loss**: Sparse Categorical Crossentropy
- **Optimizer**: AdamW [7]
- **Labels**: Category
- **Learning Rate**: 5e-4
- **Weight Decay**: 4e-3
- **Epochs / Batch size**: 8 / 128
- **Activation Function**: ReLu
- **Training/Testing/Validating Accuracy**: 72.14% / 69.88% / 60.89%

Based on the results obtained from the experiment, it can be inferred that the model aligns precisely with the intended objective. The prototype demonstrates good accuracy, which is satisfactory for its intended purpose. Further improvements can be achieved by fine-tuning the hyper-parameters and modifying the configuration file of the RoBerta model. However, it should be noted that the prototype is a multi-class model rather than a multi-label model.

3.3 Yash Kulthe

In this section, we will discuss Yash Kulthe's model; here is the detail of his model:

- **Model**: The model was made using TensorFlow's Sequential API. The model has an input of 41466 sequences of integers (each integer defines a corresponding unique word from the vocabulary), and the length of each sequence is fixed at 30. This input is fed to the Embedding layer, which transforms into a vector of 100 dimensions. The Embedding layer is also given an embedding initializer from the GloVE embeddings [9], and training is kept False. This is then given to the Bidirectional LSTM layer [1] having 100 units. The next layers are a 2D convolution layer having 100 filters and kernel size 3, and a 2D MaxPooling layer with (2,2) size. A 2D convolution layer with 64 filters, a 2D MaxPooling layer, followed by flatten and dropout, a dense layer with softmax [8] activation and 14 outputs for 14 classes which will output the output prediction probabilities.
- **Loss**: Sparse Categorical Crossentropy
- **Label**: Category
- **Optimizer**: Adam [5]
- **Learning Rate**: 3e-4
- **Activation Function**: ReLu
- **Epochs**: 50 (Early Stopping: 23 Epochs)
- **Dropout**: 0.2
- **Batch Size**: 16
- **Training/Testing/Validating Accuracy**: 64.34% / 61.13% / 61.71%

This model performs well and can learn the patterns in the training data but is not generalizing well to new, unseen data. It also over-fitted the data when run on more epochs, so early stopping had to be used. The model surely has room for improvement in reducing over-fitting and improving the generalization to new data.

3.4 Anitha Parameshwarappa

- **Model**: The input of the model, one input sequence of 300 feature dimensions. The model consists of two Bidirectional LSTM layers [4]. The first LSTM layer has 64 units, and the second LSTM layer has 32 units.
- **Dropout**: 0.2
- **Dense Layer**: Softmax [8] over 14 outputs
- **Loss**: Sparse Categorical Crossentropy
- **Optimizer**: Adam [5]
- **Learning Rate**: 1e-3
- **Batch Size**: 128
- **Epochs**: 40 (Early stopping: 25 Epochs)
- **Training/Testing/Validating Accuracy**: 55.45% / 53.23% / 53.24%

  The model is achieving moderate accuracy on the given task. It showcases its ability to learn patterns effectively and make predictions. But there is still an opportunity for improvement to achieve higher accuracy with further adjustments and refinements of the model.

3.5 Abhishek Bhardwaj

- **Model**: SVM [3]
- **Loss**: Sparse Categorical Entropy
- **Optimizer**: AMSGrad [11]
- **Learning Rate**: 1e-3
- **Epochs**: 20
- **Training/Testing/Validating Accuracy**: 57.90%/ 62.42%/ 64.70%

Evaluation Based on the evaluation results, the SVM model [3] with the hyperparameters obtained through grid search is an appropriate approach for text classification in this case. The model achieved a moderate Training accuracy of 0.5790, testing of 0.6242, and validation of 0.6470, indicating a good balance between testing and validation. Given that the data is biased and traditional transformers like BERT [2] and GPT2 [10] did not perform well, the SVM model is a suitable option. Additionally, SVM [3] is a fast and efficient algorithm, which makes it well-suited for this scenario. Overall, the SVM model with the obtained hyperparameters is a good approach for text classification on this dataset.

3.6 Conclusion

Given the rigorous comparative analysis undertaken, it is observed that the model proposed by Hung Le has exhibited the highest level of accuracy across the five models evaluated. The superior performance of Le's model, as evidenced by the empirical results, substantiates its selection for subsequent stages of deployment and serving, which will be discussed in the next sections.

4 Model Serving Options

Model serving is deploying trained ML models in production, making them accessible for inference. It involves hosting the model, managing its lifecycle, and handling requests for predictions or results. Key components include server infrastructure, model management, API or interface, scalability, and monitoring/logging. Model serving is crucial in the ML lifecycle, as it integrates models into real-world applications, enabling data-driven decision-making, predictions, and insights. In this section, we will discuss four different models serving and apply one of them to our experiment.

4.1 TensorFlow Serving

TensorFlow Serving is an open-source, high-performance serving system designed specifically for machine learning models built using TensorFlow. It is particularly useful for production environments as it allows for seamless model deployment, handling of multiple versions of models, and easy updating without disrupting the service. TensorFlow Serving supports gRPC and RESTful APIs, enabling various client libraries to interact with the served model. It also integrates with Kubernetes, allowing for scalable deployment and management of machine learning models.

4.2 Pytorch Serving

While PyTorch does not have a dedicated serving component like TensorFlow Serving, it can be served using several third-party tools and platforms. One option is to use PyTorch's JIT (Just-In-Time) compiler to create a TorchScript model, which can be deployed using the PyTorch C++ API or other serving frameworks like TorchServe. TorchServe is an open-source PyTorch model-serving library developed by PyTorch and AWS that provides features such as model versioning, multi-model serving, logging, and monitoring. It supports both RESTful and gRPC APIs for client-server communication and can be containerized using Docker for scalable deployment.

4.3 Seldon Core

Seldon Core is an open-source platform designed for deploying, scaling, and monitoring machine-learning models on Kubernetes. Seldon Core supports multiple machine learning frameworks, including TensorFlow, PyTorch, and Scikit-learn. It provides a standardized interface using custom resource definitions (CRDs) to deploy models as microservices, enabling seamless integration with various tools and platforms. Seldon Core also supports advanced deployment strategies, such as A/B testing, multi-armed bandits, and ensemble models, allowing more sophisticated model serving and performance monitoring.

4.4 Hugging Face Serving

Hugging Face is an NLP-focused company that provides an extensive ecosystem for state-of-the-art models such as BERT [2], GPT [10], Roberta [6], and others. They offer the Transformers library, simplifying fine-tuning and serving pre-trained models for various NLP tasks. Hugging Face Inference API is a cloud-based API service that allows users to access and interact with pre-trained models hosted on the Hugging Face Model Hub. The API supports various languages and can be used for tasks like text classification, named entity recognition, text generation, and more.

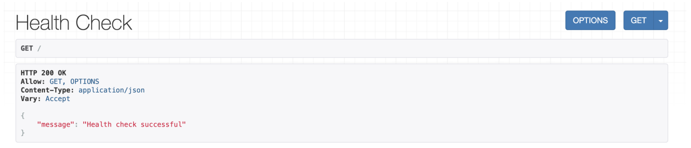
4.5 Conclusion

In our experiment, it is evident that Hugging Face serves as a user-friendly and convenient model serving solution. Its extensive ecosystem, support for state-of-the-art NLP models, and seamless integration with the Hugging Face Inference API make it an attractive choice for deploying and serving NLP models (we will discuss in section 5). By leveraging Hugging Face's model serving capabilities, we can efficiently incorporate machine learning models into real-world applications and benefit from its ease of use and powerful NLP functionalities.

5 Web Service Building

In this section, we will build the web service for deploying our model; we are using Django RESTful API because it provides a powerful, flexible, and easy-to-use toolkit for creating web APIs. This framework simplifies creating APIs by handling common tasks like authentication, pagination, and serialization while maintaining the ability to customize to your needs.

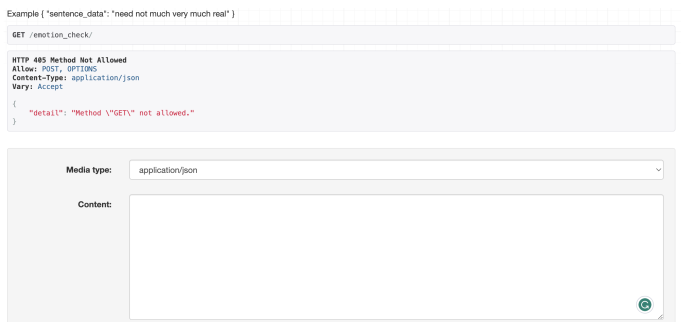
Furthermore, upon successfully building and running the web service on the local machine, a URL will be made available for users to interact with the local server using the POST method. Django caters to developers who wish to engage with the server and assess its performance without writing additional code. To accommodate this need, Django supplies input fields that users can populate, streamlining the process and enabling seamless interaction. For more detail, here is the picture when we are pointing to the URL (API Pointer):



```
GET /

HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "message": "Health check successful"
}
```

Figure 1: Django API starting with Health Check function.

Additionally, developers can utilize POSTMAN link, accessible via this link, to send requests to the server when interacting with the API endpoint. Upon receiving a response, one can ascertain that the API is functioning properly and subsequently submit requests to the API to evaluate the model's performance within the API. The API endpoint is located at link, which we have previously configured. To interact with this endpoint, developers merely need to employ the POST method and adhere to the guidelines for submitting requests.



Example { "sentence_data": "need not much very much real" }

```
GET /emotion_check/

HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

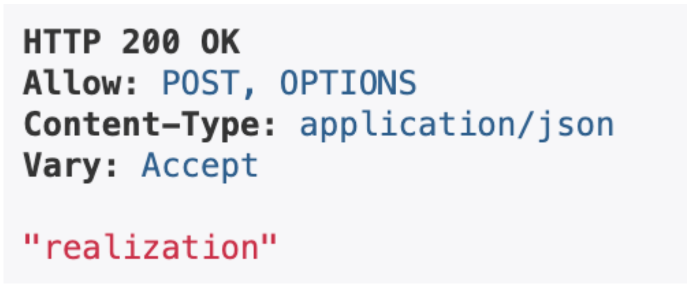
Media type: application/json

Content:

Figure 2: Django API with the main function (emotion check).

JSON is the data type accepted by the API for the POST method in this implementation. JSON (JavaScript Object Notation) is a widely-used, lightweight data-interchange format that is easy for humans to read and write and for machines to parse and generate. When sending a request to the API, we ensure for the developer that the data is structured in the JSON format, adhering to the required schema specified in the API documentation.

Django offers a graphical user interface (GUI) for making HTTP requests to a server, eliminating the need for developers to write code for this task. In the GUI, the developer can input the content they wish to send to the server using structured data formats like JSON. For example, the provided content is a JSON object with a single key-value pair: ”sentence_data” and **”need not much very much real”**. To initiate the request, the developer simply clicks the ”POST” button, which sends the content to the server using the HTTP POST method. The server then processes the request, running an emotion recognition model on the input provided, and returns the result to the developer's application, as we can see below:



```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

"realization"
```

Figure 3: Django API with the response's status code and result.

Upon analysis, we have observed that the emotion recognition model receives the input string ”need not much very much real” and produces an emotion as a result. This represents the initial example in our experimental investigation. Further details about our experiment, including additional examples and data analysis, can be found at the following Github link.

5.1 Model Visualization

To enhance the comprehension of our model architecture, we have constructed an HTML file that provides a visual representation of our system's underlying design. This innovative approach facilitates understanding developers' and users' intricate technical details. The link for this local running is link.

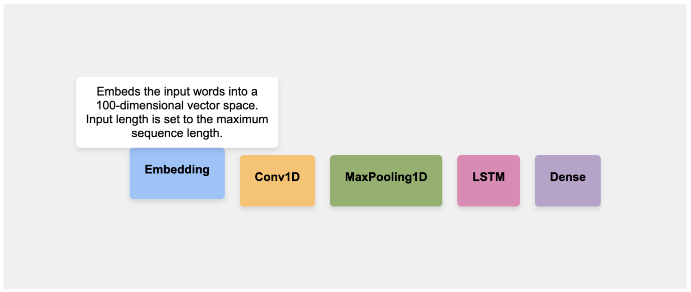
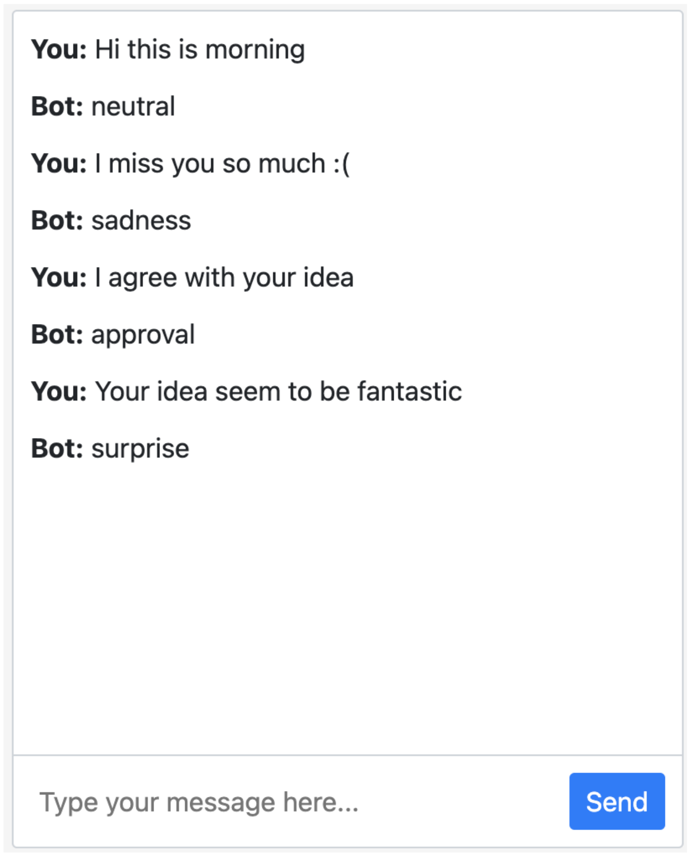

Embeds the input words into a 100-dimensional vector space. Input length is set to the maximum sequence length.
Embedding
Conv1D
MaxPooling1D
LSTM
Dense

Figure 4: Using HTML and CSS for model visualization.

In addition to our innovative model architecture, we have also developed a user interface (UI) chat feature that enables users to interact with our system directly. The chat interface is accessible via the following link: .

Users can input a sentence into the chat interface and receive the corresponding emotion as output from our model by pressing the "enter" key. This user-friendly feature enables individuals without a technical background to actively engage with our research work.



You: Hi this is morning

Bot: neutral

You: I miss you so much :(

Bot: sadness

You: I agree with your idea

Bot: approval

You: Your idea seem to be fantastic

Bot: surprise

Figure 5: Emotion Chatbot intergration.

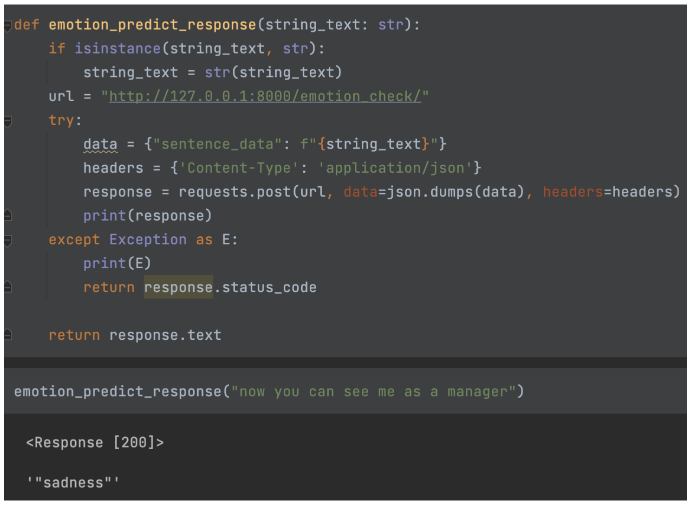
5.2 Conclusion

Web services can provide developers with a valuable resource to access data or functionality provided by external applications, including our emotion recognition service. By leveraging the capabilities of our web service, developers can avoid the time-consuming and challenging task of building their own emotion recognition models, enabling them to focus on other aspects of their applications. This approach can result in a streamlined development process that reduces development time and effort, helping developers to create sophisticated and high-quality applications more efficiently.

6 Notebook With Experiment On Endpoint

This section involves the development of a notebook that can interact with our API endpoint, which was established in section 3. To begin with, several libraries are required for handling requests. Additionally, it is necessary to verify the functionality of the API endpoint by making a request to the server and checking for any errors.

Subsequently, we will develop functions that utilize the POST method to send requests to the server via the specified link: link. These functions will be responsible for handling the requests and processing the responses returned by the server. After developing and requesting, we can see that the server is returning to the result, as described in the picture below:



```
def emotion_predict_response(string_text: str):
    if isinstance(string_text, str):
        string_text = str(string_text)
    url = "http://127.0.0.1:8000/emotion_check/"
    try:
        data = {"sentence_data": f"{string_text}"}
        headers = {'Content-Type': 'application/json'}
        response = requests.post(url, data=json.dumps(data), headers=headers)
        print(response)
    except Exception as E:
        print(E)
        return response.status_code

    return response.text
```

```
emotion_predict_response("now you can see me as a manager")
```

```
<Response [200]>

'"sadness"'
```

Figure 6: Interaction with endpoint API by using the notebook.

Upon inspection, it can be determined that a response with status code 200 has been received, indicating that the request was successful. Additionally, the response payload includes the emotion recognition result, "sadness". For more detail, we provide the link to our notebook; here is the link

6.1 Conclusion

Developing a notebook-to-API conversion has facilitated a deeper understanding of how to establish connections between end devices and the API endpoint. This process has illuminated the system's underlying architecture, enabling us to comprehend better and visualize the various components and interactions involved. Translating a notebook into an API gives insight into the data flow, communication protocols, and requirements for interfacing with the model, resulting in a more robust and efficient implementation. Ultimately, this knowledge improves system design, streamlined development, and enhanced integration capabilities for various applications and devices.

7 Service Discussion - Online Model Serving

In assessing the performance of the implemented web service, it is crucial to conduct stress testing to identify its limitations and potential bottlenecks. By doing so, we can pinpoint the strengths and weaknesses of the current architecture and subsequently propose recommendations for potential improvements.

Firstly, we are going to discuss the strengths:

- Response Time: 1.36s/ request
- Throughput: The service can handle 1227 requests/ second on our local machine. Because of running on MacBook, that can handle a large request per second.
- Ease of Integration: Django RESTful API facilitates seamless integration with various devices and applications.
- Flexibility: The modular nature of Django enables developers to customize the web service per specific requirements.
- Security: Django's built-in security features help protect the web service from common threats and vulnerabilities.

Secondly, we are going to discuss the weaknesses:

- Limited Concurrency: The default Django server may not be able to handle a high volume of simultaneous connections, potentially leading to slow response times or connection failures.
- Resource Consumption: Running complex models on the server side may consume substantial resources, potentially affecting the system's overall performance.
- Still Offline Deploying: The service is still local, and it is still not public to the user/ developer for testing/ using

From strengths and weakness, we are going to improve our service, as described below:

- Asynchronous Processing Enhancement: To achieve greater concurrency and facilitate the simultaneous handling of multiple requests without hindering the server's performance, an asynchronous processing mechanism shall be implemented. The *Celery Framework* is proposed as a viable solution to augment the efficiency of our system or service.
- Caching Mechanism: To mitigate server load and expedite response times, a caching strategy should be employed for frequently accessed data or responses. The utilization of *Redis* and *RabbitMQ* is recommended for bolstering our service in this regard.
- Model Optimization: It is essential to streamline the underlying model, reducing its complexity and resource consumption without compromising its accuracy. Techniques such as Knowledge Distillation or Quantization shall expedite processing times and enhance overall performance.
- Online Deployment: At this stage, we will leverage the *Hugging Face Model Serving* platform to facilitate the deployment of our optimized model in an online environment. This solution ensures seamless integration, scalability, and accessibility for users while maintaining the model's efficacy and performance.

Due to time constraints, we have opted for online deployment to deploy our model in a virtual environment efficiently. We have utilized the *Hugging Face Model Serving* platform for this purpose. Access to the deployed model can be obtained through the following hyperlink: link. This approach ensures expeditious implementation while maintaining the model's effectiveness and performance.

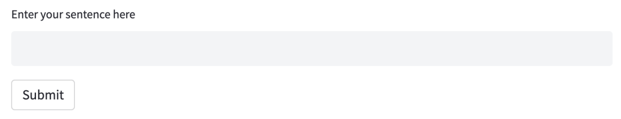
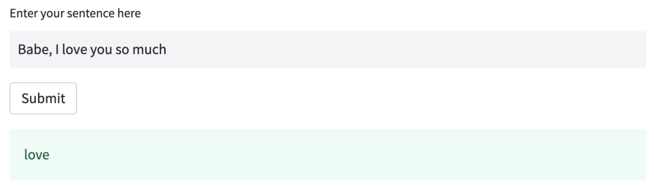


Figure 7: Model deploying into Hugging Face.

Upon providing input and selecting the 'Submit' button, users will receive a corresponding output generated by the model. This output reflects the emotion associated with the input sentence, illustrating the model's ability to analyze and categorize emotional content within textual data effectively.



Babe, I love you so much

love

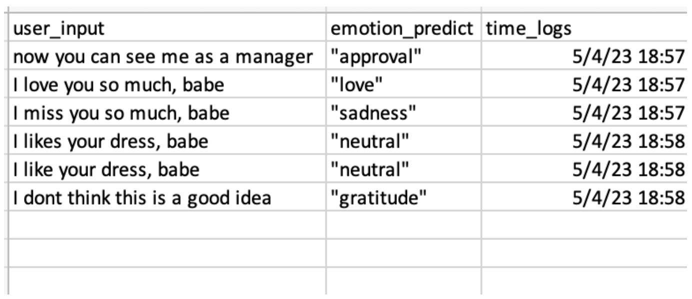
Figure 8: Model deploying into Hugging Face when we are testing.

7.1 Conclusion

In conclusion, developing a web service using Django and the Django RESTful API for deploying a machine learning model has demonstrated the benefits of a flexible, secure, and easy-to-integrate framework. Through rigorous performance evaluation and stress testing, the strengths and weaknesses of the current implementation have been identified, shedding light on the system's limitations and areas that require improvement.

8 Logs Implementing

Logs refer to records of events or activities within the system in the context of a software system or web service. These records typically include timestamps, event types, user actions, system performance metrics, and error messages. This section will focus on constructing log files for the web service that capture user input. For the purpose of efficient storage and accessibility, the log files will be structured as CSV (Comma Separated Values) files. This widely-used format enables the easy organization and interpretation of the logged data, providing valuable insights into the utilization of the service and facilitating further analysis. Moreover, after building logs, the user input will be captured for re-training in the future, as below:



user_input	emotion_predict	time_logs
now you can see me as a manager	"approval"	5/4/23 18:57
I love you so much, babe	"love"	5/4/23 18:57
I miss you so much, babe	"sadness"	5/4/23 18:57
I likes your dress, babe	"neutral"	5/4/23 18:58
I like your dress, babe	"neutral"	5/4/23 18:58
I dont think this is a good idea	"gratitude"	5/4/23 18:58

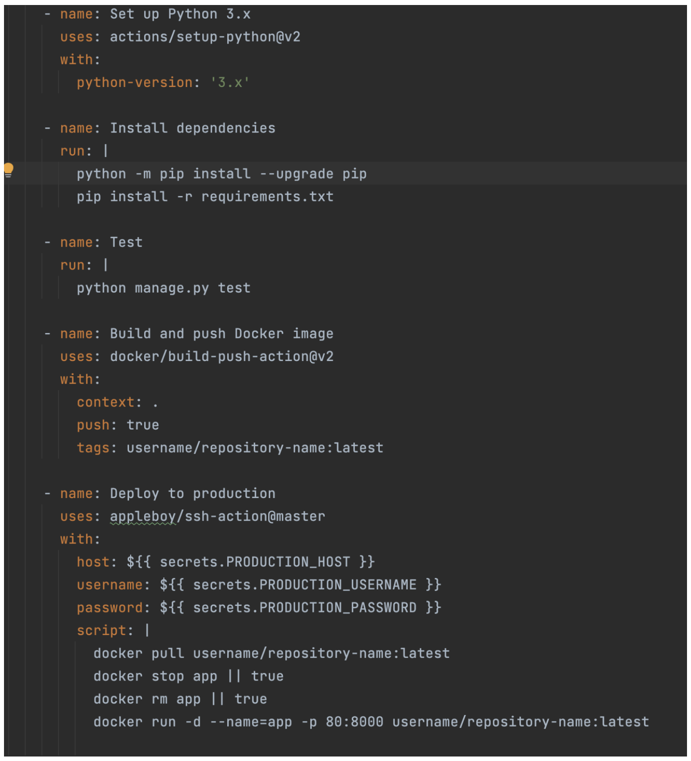
We can proficiently monitor, analyze, and troubleshoot a web service by employing logs. Logs offer essential insights into user interactions, enabling a comprehensive understanding of user characteristics. This information, in turn, allows us to identify the strengths and weaknesses of their models or services, thereby paving the way for continuous improvement and refinement to better cater to users' needs in the future.

9 Basic CI/CD Implementing

CI/CD stands for Continuous Integration and Continuous Deployment, which are practices involving automated building, testing, and software deployment. CI focuses on automatically integrating code changes and running tests to ensure quality, while CD automates testing code deployment to production environments. These practices, supported by tools like Jenkins and GitLab CI/CD, streamline software development, reduce errors, and accelerate delivery, leading to faster release cycles, better collaboration, and improved software quality.

In this section, we will build the CI/CD for a Django application using GitHub Actions; here is the detail of our CI/CD:

- Check out the code.
- Set up Python 3.x environment.
- Install dependencies from requirements.txt.
- Run Django tests.
- Build and push a Docker image to Docker Hub.
- Deploy the updated Docker image to the production server using SSH.

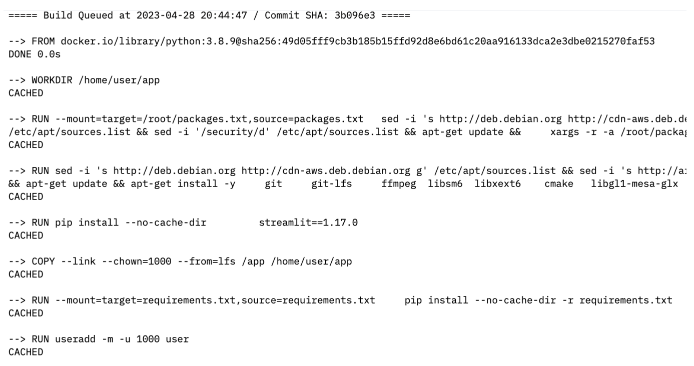


```
- name: Set up Python 3.x
  uses: actions/setup-python@v2
  with:
    python-version: '3.x'

- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install -r requirements.txt

- name: Test
  run: |
    python manage.py test

- name: Build and push Docker image
  uses: docker/build-push-action@v2
  with:
    context: .
    push: true
    tags: username/repository-name:latest

- name: Deploy to production
  uses: appleboy/ssh-action@master
  with:
    host: ${{ secrets.PRODUCTION_HOST }}
    username: ${{ secrets.PRODUCTION_USERNAME }}
    password: ${{ secrets.PRODUCTION_PASSWORD }}
    script: |
      docker pull username/repository-name:latest
      docker stop app || true
      docker rm app || true
      docker run -d --name=app -p 80:8000 username/repository-name:latest
```

Figure 8: Basic CI/CD for installing the library when receiving the new commit.

After implementing this CI/CD configuration and integrating it with Hugging Face, the pipeline will automatically trigger for every push to the main branch. This will ensure our Django application is tested, built, and deployed to the production server consistently and without manual intervention.

This helps streamline the development process and ensures our application is always up-to-date with the latest changes.



```
===== Build Queued at 2023-04-28 20:44:47 / Commit SHA: 3b096e3 =====

--> FROM docker.io/library/python:3.8.9@sha256:49d05fff9cb3b185b15ffd92d8e6bd61c20aa916133dca2e3dbe0215270faf53
DONE 0.0s

--> WORKDIR /home/user/app
CACHED

--> RUN --mount=target=/root/packages.txt,source=packages.txt   sed -i 's http://deb.debian.org http://cdn-aws.deb.de
/etc/apt/sources.list && sed -i '/security/d' /etc/apt/sources.list && apt-get update &&     xargs -r -a /root/packag
CACHED

--> RUN sed -i 's http://deb.debian.org http://cdn-aws.deb.debian.org g' /etc/apt/sources.list && sed -i 's http://a
&& apt-get update && apt-get install -y     git     git-lfs     ffmpeg  libsm6  libxext6    cmake   libgl1-mesa-glx
CACHED

--> RUN pip install --no-cache-dir         streamlit==1.17.0
CACHED

--> COPY --link --chown=1000 --from=lfs /app /home/user/app
CACHED

--> RUN --mount=target=requirements.txt,source=requirements.txt     pip install --no-cache-dir -r requirements.txt
CACHED

--> RUN useradd -m -u 1000 user
CACHED
```

Figure 9: The server automatically installs the library when receiving the new commit.

10 Feedback Receiving - Scheduled Re-Training

In addition to supplying the model for Hugging Face's testing and utilization, we facilitate user feedback, encompassing the user-perceived 'truth' label. This 'truth' label represents the user's interpretation of the most appropriate or accurate output or categorization. By utilizing this user feedback in tandem with the data amassed from user interactions, we leverage Apache Airflow for the re-training process of the model. This re-training process enhances the model's performance, incrementally increasing its predictive accuracy.

This methodology underscores the importance of user feedback in machine learning systems, providing a dynamic and iterative process that continually refines and optimizes the model. By integrating user feedback into the model's training regimen, the system can adapt and evolve in alignment with the user's expectations and perceptions of 'truth'. The process thus leverages Airflow's capabilities to schedule and manage the workflow of the re-training process, ensuring that the model remains up-to-date and improves over time.

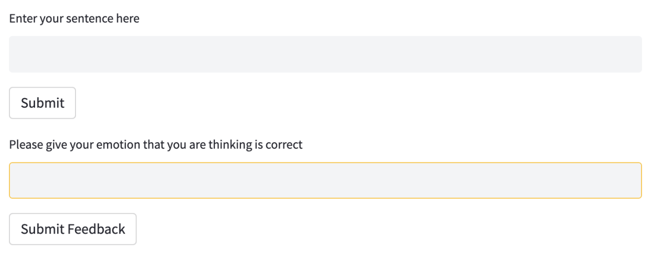


Figure 10: The UI for user feedback on Hugging Face.

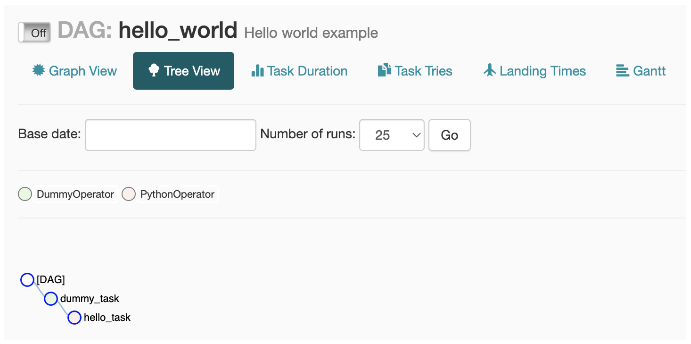


Figure 11: The UI on Airflow for schedule for re-training data.

11 Group Presenting

In this segment, our esteemed team endeavours to deliver an erudite and comprehensive academic presentation, highlighting our research project's key findings, methodologies, and implications. Our objective is to elucidate the significance of our work and its potential impact on the broader field of study.

We have meticulously conducted thorough research and analysis, and our efforts culminate with an insightful and compelling presentation. The link to our presentation can be found here link, and for the convenience of our esteemed audience, the link to the presentation file is also provided here link.

12 Conclusion

In conclusion, sentiment recognition, a crucial component of natural language processing, has the potential to provide invaluable insights into the emotional responses, opinions, and preferences of individuals. Our study comprehensively examines various model serving options, ultimately identifying the most suitable choice for specific use cases.

This research also offers a detailed guide for constructing a web service, testing a deployed endpoint, and monitoring user inputs and predictions. The performance of the implemented service was extensively evaluated, revealing significant strengths while highlighting areas for improvement. We have discussed these limitations in-depth, proposing architectural enhancements that could bolster the service's performance.

Moreover, the research delves into creating a rudimentary continuous integration and continuous deployment (CI/CD) pipeline. Such a system allows for seamless model building and deployment in response to changes in data or code, contributing to a more efficient and robust model life cycle management.

Moving forward, we encourage further exploration and improvements in this field. Continued advancements in sentiment recognition technologies, combined with the optimized model serving options and streamlined CI/CD pipeline, could provide even greater accuracy and efficiency in processing and understanding natural language data. This study represents a significant step forward in leveraging NLP technologies for comprehensive sentiment analysis, laying a solid foundation for future research and development.

References

[1] Zhiyong Cui, Ruimin Ke, Ziyuan Pu, and Yinhai Wang. Deep bidirectional and unidirectional lstm recurrent neural network for network-wide traffic speed prediction. *arXiv preprint arXiv:1801.02143*, 2018.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[3] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[5] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[7] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[8] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[9] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[10] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[11] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.