Twitter Sentiment Analysis - Deep learning project

This document outlines the iterative process used to develop a robust sentiment analysis model and describes the production-ready infrastructure implemented for high-concurrency inference.

The goal was to classify text sentiment accurately and deploy the model in an environment capable of handling high user traffic, specifically benchmarking against **500 concurrent users**.

1. Model Development Iterations

The project began by exploring various classification scopes and fine-tuning strategies using Hugging Face's Transformers library, primarily focusing on TFBertForSequenceClassification and TFDistilBertForSequenceClassification.

Attempt	**Classification Scope**	**Model & Strategy**	**Outcome & Challenge**
1	**4 Classes** (Positive, Negative, Neutral, Irrelevant)	TFBertForSequenceClassification + Class Weights	High complexity due to large parameter count; infeasible for rapid iteration and deployment resource constraints. Dropped due to the size of the BERT model.
2	**4 Classes** (Positive, Negative, Neutral, Irrelevant)	TFDistilBertForSequenceClassification + Full Fine-tuning + Class Weights	Training was excessively slow due to full fine-tuning, despite using the smaller DistilBERT model.
3	**2 Classes** (Positive, Negative)	TFDistilBertForSequenceClassification + Full Fine-tuning	Accuracy dropped significantly (), indicating severe overfitting. Removing less-represented classes (Neutral, Irrelevant) did not immediately solve the generalization issue with full fine-tuning.

4	**2 Classes** (Positive, Negative)	TFDistilBertForSequenceClassification + Frozen Weights	Introduced a frozen weight strategy (transfer learning) with a moderate learning rate (). Model continued to overfit, yielding low test accuracy.
5 (Final)	**2 Classes** (Positive, Negative)	TFDistilBertForSequenceClassification + Frozen Weights + Regularization	Successfully mitigated overfitting using a multi-pronged approach, achieving the required performance target.

2. Final Model Configuration (Approach 5)

The final, successful approach focused on preventing overfitting in the pre-trained DistilBERT architecture, resulting in a high-accuracy, deployable model:

Parameter	**Value / Description**
Base Model	TFDistilBertForSequenceClassification
Classification Scope	**2 Labels** (Positive, Negative)
Fine-tuning Strategy	**Frozen Weights Training** (Only top layers are trainable)
Learning Rate	Reduced to **1e-4** (from 5e-4)
Regularization	**Dropout Layer** with a rate of **0.4**
Optimization	**Adam** Optimizer
Weight Decay	**L2 Regularization** with weight decay 0.0001
Training Controls	**Early Stopping** and **Model Checkpoint** (to save only the best performing model)
Test Accuracy	Achieved **80.97%**
Precision for class 0 and 1	`0.8056 , 0.8146`

Recall for class 0 and 1	`0.8356, 0.7817`
F1-score for class 0 and 1	`0.8203, 0.7978`

3. Deployment, Optimization, and Load Testing

The trained model was containerized and deployed using a scalable architecture designed for high availability and low latency inference.

Prediction API (FastAPI)

- **Framework: FastAPI** was chosen for its modern, asynchronous nature, enabling high performance.
- **Optimization:**
    - **Single Model Load:** The model is loaded only **once** into memory upon application startup, eliminating disk I/O latency for every prediction request.
    - **Threading for Inference:** Standard Python threading was implemented to prevent resource allocation conflicts during model inference, ensuring non-blocking execution and smoother parallel processing.

Deployed API Endpoint

The prediction API is deployed and accessible at the following URL:

https://dac7a9adfa2824d729f097cdc2dd2cf65.clg07azjl.paperspacegradient.com/predict

Sample Request Payload (POST):

```
{
  "text": "I am really frustrated by your service!"
}
```

Sample Response:

```
{
   "sentiment": "Negative",
   "confidence_score": 0.7034
}
```

Dockerization and Scalability

- **Environment:** The entire application (FastAPI + Model) was containerized using **Docker**.

- **Hardware:** Deployed on a cloud machine with **8 CPU cores**, **30 GB RAM**, **GPU support**, and **auto-scaling** capability.
- **Concurrency Handling (Docker Compose):** A docker-compose setup was implemented with **7 worker containers** running the FastAPI service, allowing the prediction API to handle high concurrent load by leveraging multiple cores.

Key Docker Commands:

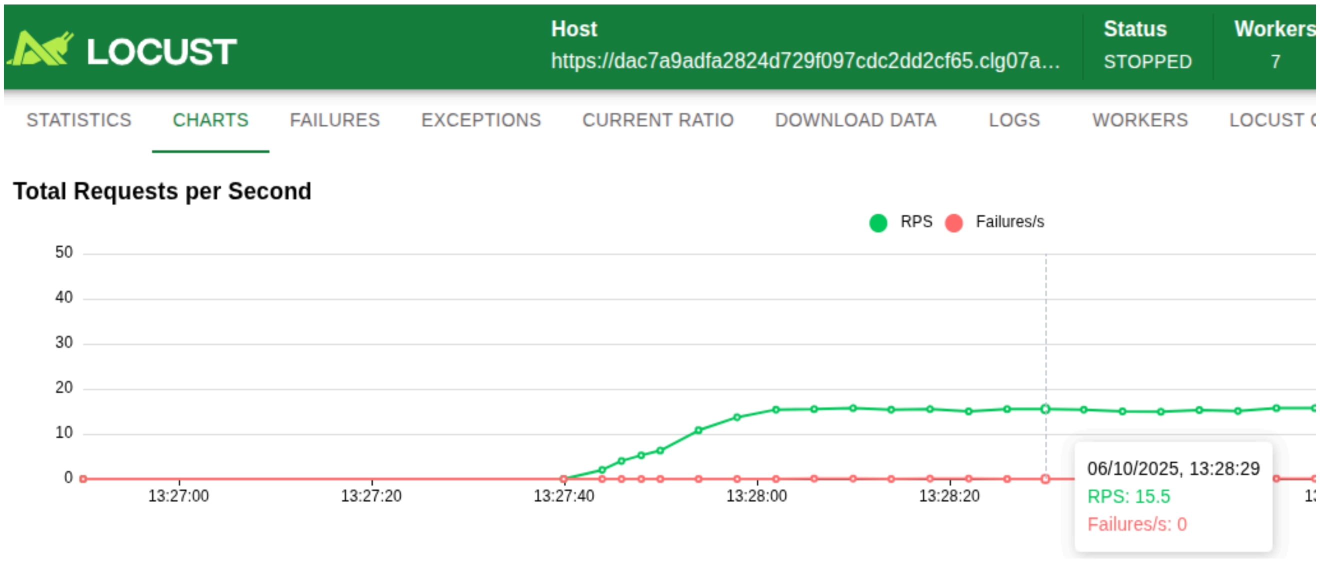
Command	**Purpose**
docker pull afrahthahir/sentiment-api:latest	Pulls the docker image from the dockerhub
docker compose up -d --scale worker=7	Builds, creates, and starts the **7 worker containers** specified in docker-compose.yml in detached mode.
docker-compose down	Stops and removes the containers, networks, and volumes defined in the docker-compose.yml file.

Load Testing (Locust)

- **Tool: Locust** was used to perform performance and load testing against the deployed API.
- **Benchmark Goal:** The system was successfully tested to ensure stable performance and responsiveness when subjected to a load of **500 concurrent users**.

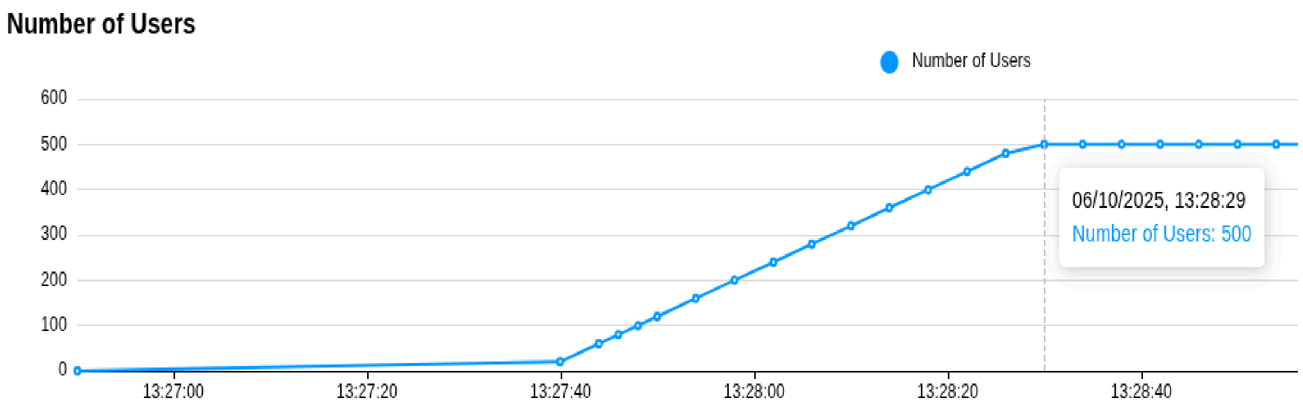
Key Locust Commands:

Command	**Purpose**
Once the **docker compose up -d --scale worker=7** is executed, Go to localhost port **8089**. You will see the locust UI	Starts the Locust testing framework with the multiple workers to handle the distributed api load in your localhost port 8089
Locust Web UI Settings	Configuration used to simulate the target load: - Number of users: 500 - Spawn rate: 50 users/second


Host
https://dac7a9adfa2824d729f097cdc2dd2cf65.clg07a…
Status
STOPPED
Workers
7

Total Requests per Second

06/10/2025, 13:28:29
RPS: 15.5
Failures/s: 0


Number of Users

06/10/2025, 13:28:29
Number of Users: 500

When the number of concurrent users is 500, the Request Per Second is 15.5, and the Request failed is 0. This shows that the API is scaled to handle 500 concurrent users.

Speech-to-Text Service Implementation for Real-Time Application - R&D Skills:

I have conducted an extensive search for both well-established commercial services and promising open-source solutions suitable for real-time, low-latency applications.

STT Models Comparison Report

My observations and comparisons are recorded in this following report.
STT Models Comparison Report

Criteria for Evaluation

The following custom criteria were developed to evaluate candidates based on the application's non-negotiable requirement for a robust, real-time experience:

1. **Accuracy (Word Error Rate - WER):** The paramount metric. The solution must provide industry-leading accuracy in various acoustic environments.
2. **Real-Time Latency (**≤500ms**):** Must support a streaming API with end-to-end latency below 500ms to ensure a fluid, conversational user experience.
3. **Cost (Commercial Only):** Evaluation of pricing structures (per minute/second) and tiers against the total cost of ownership (TCO) compared to the effort of managing an open-source solution.