A Project Report on

Fine Tuning a language model for Sentiment Analysis with Human Feedback loop

in partial fulfilment of the requirements for the award of the Degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

Submitted by

D. Bhargav Krishna	21B81A1233
I. Kiranmayeesree	21B81A1246
M. Diwakar Reddy	21B81A1266
Ch. Umesh Chandra Arjun	21B81A1227

Under the Esteemed Guidance of

Smt. M. Vijaya Sudha, **M.Tech**

Assistant Professor, Department of IT

DEPARTMENT OF INFORMATION TECHNOLOGY

SIR CR REDDY COLLEGE OF ENGINEERING

Approved by AICTE & Accredited by NBA & NAAC

Affiliated to Jawaharlal Nehru Technological University, Kakinada

ELURU-5340007

2024-25

SIR C R REDDY COLLEGE OF ENGINEERING

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

This is to certify that this project report entitled "**Fine Tuning a Language Model for Sentiment Analysis with Human Feedback Loop**" being Submitted by **D. Bhargav Krishna** (21B81A1233)**, I. Kiranmayeesree**(21B81A1246), **M. Diwakar Reddy** (21B81A1266)**, CH. Umesh Chandra Arjun**(21B81A1227) in partial fulfilment for the award of the Degree of Bachelor of Technology in Information Technology to the JNTU KAKINADA is a record of Bonafide work carried out under my guidance and supervision. The results embodied in this project report have not been submitted to any other University or Institute for the award of any Degree.

Smt. M.Vijaya Sudha, M.Tech	**Dr. K. Satyanarayana**
PROJECT GUIDE	**HEAD OF THE DEPARTMENT**
Department of IT	Department of IT
Sir CRR College of Engineering	Sir CRR College of Engineering

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We wish to express our sincere thanks to various personalities who were responsible for the successful completion of this project.

We thank our principal, **Dr. K. VENKATESWARA RAO**, for providing the necessary infrastructure required for our project.

We are grateful to **Dr. K. SATYANARAYANA,** Head of the Information Technology department, for providing the necessary facilities for completing the project in specified time.

We express our deep-felt gratitude to **Smt. M. Vijaya Sudha,** as her valuable guidance and unstinting encouragement enabled us to accomplish our project successfully in time.

Our special thanks to librarian **Smt. D. LAKSHMI KUMARI**, and to the entire library staff Sir C.R.R College of Engineering, for providing the necessary library facilities.

We express our earnest thanks to faculty members and non-teaching staff of IT for extending their valuable support.

PROJECT MEMBERS

D. Bhargav Krishna	Reg No:21B81A1233
I. Kiranmayeesree	Reg No:21B81A1246
M. Diwakar Reddy	Reg No:21B81A1266
Ch. Umesh Chandra Arjun	Reg No:21B81A1227

ABSTRACT

In recent years, language models have shown remarkable success in various Natural Language Processing (NLP) tasks. This project focuses on fine-tuning a transformer-based language model (such as BERT or RoBERTa) for sentiment analysis, integrated with a human feedback loop to continuously enhance its performance. The system accepts user text inputs through an interactive frontend interface and returns sentiment predictions (positive, negative, or neutral) using a fine-tuned model. Crucially, users can provide feedback on the accuracy of the predictions, enabling the system to learn from its mistakes over time. The architecture includes a web-based user interface, a backend API layer (Flask or FastAPI), a sentiment analysis model powered by Hugging Face Transformers, a feedback module to store and process user responses, and a database for maintaining all interactions and logs. Feedback collected from users is used to further fine-tune the model, creating a self-improving system that adapts to real-world data and edge cases. By incorporating human-in-the-loop learning, the model improves in both accuracy and fairness. The project is deployable on cloud platforms like AWS or Heroku using Docker containers, making it scalable and accessible. This solution holds significant potential for applications in customer service, product reviews, social media monitoring, and other domains where understanding sentiment is critical.

DECLARATION

We here by declare that the Project entitled **Fine Tuning a Language model for Sentiment Analysis with Human Feedback Loop** submitted for the B. Tech Degree is our original work and the Project has not formed the basis for the award of any degree, associateship, fellowship or any other similar titles.

D. Bhargav Krishna	Reg No:21B81A1233
I. Kiranmayeesree	Reg No:21B81A1246
M. Diwakar Reddy	Reg No:21B81A1266
Ch. Umesh Chandra Arjun	Reg No:21B81A1227

CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Addressing Fine-tuning language model

In recent years, language models have revolutionized natural language processing (NLP) tasks, with sentiment analysis emerging as a key application across industries. While pre-trained models like BERT, GPT, and RoBERTa offer powerful baseline performance, domain-specific nuances and evolving language patterns often demand further fine-tuning to achieve high accuracy and contextual relevance.Fine-tuning enables adaptation of a general-purpose language model to specific datasets or tasks, such as detecting sentiment in product reviews, social media content, or customer support interactions. However, traditional fine-tuning methods can suffer from limitations, including bias in training data and inadequate handling of edge cases or subjective tone.To address these challenges, integrating a human feedback loop into the fine-tuning process provides a promising approach. By incorporating real-time or iterative human evaluations—whether through crowdsourcing, expert annotation, or user-in-the-loop corrections—models can be continuously refined to improve sentiment classification performance. This approach not only enhances accuracy but also helps align the model's predictions with human intuition and ethical standards.This document explores the methodology, tools, and best practices for fine-tuning a language model for sentiment analysis, with a particular focus on incorporating human feedback as a central component of the training pipeline

1.2 Objective

The objective of this project is to develop a more accurate and context-aware sentiment analysis model by fine-tuning a pre-trained language model using a structured human feedback loop. While existing language models demonstrate strong general performance, they often struggle with interpreting nuanced, sarcastic, or emotionally complex language, especially in domain-specific contexts. To address these limitations, this project incorporates human feedback—gathered through annotations, corrections, or ratings—into the fine-tuning process, allowing the model to iteratively learn from real-world human judgments. This approach aims not only to improve classification accuracy but also to enhance the model’s alignment with human understanding and reduce bias in sentiment interpretation. Furthermore, the project explores methods to integrate feedback efficiently and at scale, evaluating the effectiveness of this human-in-the-loop strategy in comparison to traditional fine-tuning techniques. Ultimately, the goal is to create a more robust, adaptive, and ethically responsible sentiment analysis system suitable for diverse applications.

1.3 Scope of the Project

This project focuses on fine-tuning a pre-trained language model for sentiment analysis, enhanced through the integration of a structured human feedback loop aimed at improving prediction accuracy, contextual awareness, and adaptability to nuanced emotional expressions. The project encompasses several key stages: selecting or constructing a domain-relevant, sentiment-labeled dataset; training an initial baseline model using transformer architectures (such as BERT or RoBERTa); and systematically incorporating human feedback—such as corrections, re-labeling, or validation—into an iterative fine-tuning process.

The scope is intentionally limited to English-language text to maintain linguistic consistency and ensure compatibility with the selected pre-trained models. A specific domain—such as product reviews, social media posts, or customer service conversations—will be chosen based on data availability and relevance, allowing the system to focus on the unique challenges and sentiment patterns within that context.

CHAPTER 2
Problem Statement

Despite the strong performance of pre-trained language models in sentiment analysis, they often struggle with domain-specific data and nuanced emotional expressions. These models can misinterpret sarcasm, idioms, or culturally specific language, and frequently reinforce biases found in their training data, making them less reliable in real-world contexts like healthcare or finance. Traditional fine-tuning methods, which rely solely on labeled datasets, are inflexible and lack the ability to adapt to evolving language and sentiment. To overcome these challenges, this project proposes a dynamic sentiment analysis framework that integrates human feedback directly into the model's learning loop. This feedback-driven approach enhances contextual understanding, reduces misclassifications, and ensures the system remains adaptable, accurate, and aligned with real user expectations, particularly in complex or emotionally rich scenarios.

CHAPTER 3

Literature Survey

The evolution of sentiment analysis in natural language processing (NLP) has closely followed the broader advancements in deep learning and transformer-based architectures. Early sentiment analysis models relied heavily on rule-based systems and traditional machine learning techniques such as Naive Bayes and SVMs, using hand-crafted features and lexicons. However, the introduction of word embeddings like Word2Vec (Mikolov et al., 2013) and GloVe (Pennington et al., 2014) marked a shift toward data-driven approaches that captured semantic relationships. This progress culminated in the development of transformer models, most notably BERT (Devlin et al., 2018), which brought bidirectional context understanding to downstream tasks including sentiment classification. RoBERTa (Liu et al., 2019) further improved BERT's performance by optimizing training strategies, while GPT-3 (Brown et al., 2020) demonstrated the potential of large-scale autoregressive models in zero- and few-shot settings. Despite these breakthroughs, researchers soon identified persistent issues—such as poor generalization to domain-specific content, biased outputs, and difficulty in handling nuanced or sarcastic sentiment. In response, attention has shifted toward interactive learning paradigms. Reinforcement Learning from Human Feedback (RLHF), as implemented in InstructGPT (Ouyang et al., 2022) and refined in ChatGPT (OpenAI, 2022), allows models to be fine-tuned based on human-generated reward signals, improving alignment with user expectations. Additionally, Hancock et al. (2018) introduced *Training Classifiers with Natural Language Explanations*, showing that models can learn from rich, structured human input beyond simple labels. Other studies (e.g., Daniel et al., 2020; Kiela et al., 2021) have explored crowd-sourced feedback loops and active learning techniques to iteratively enhance sentiment model performance, adaptability, and fairness. These findings underscore a growing shift toward hybrid approaches, where human feedback is used not only for annotation but also as an integral component of the model's learning and evaluation process. This literature establishes a strong foundation for the use of human-in-the-loop methods to build more accurate, ethical, and context-sensitive sentiment analysis systems.

CHAPTER 4
System Analysis

4.1 Limitations of Existing Analysis Systems

Existing sentiment analysis systems typically rely on pre-trained language models that are fine-tuned using static labeled datasets. While these systems demonstrate satisfactory performance on standard benchmarks, they often exhibit significant shortcomings when deployed in real-world, domain-specific applications. One major limitation is their inability to accurately interpret context-dependent expressions such as sarcasm, idioms, and culturally rooted language nuances. These expressions require a deeper understanding of human emotion, intention, and societal context—areas where traditional models often fall short.

Additionally, many of these systems inherent biases from their training data, leading to skewed or unfair predictions. For instance, certain demographic groups or dialects may be consistently misclassified due to underrepresentation in the training corpus. This raises concerns about the ethical deployment of such systems in sensitive environments.

Another challenge lies in the static nature of these models. Once deployed, they do not adapt to new expressions or user preferences without undergoing a full retraining process.

In summary, existing systems are limited by context insensitivity, bias retention, and inflexibility. These shortcomings highlight the need for a more dynamic, adaptive, and human-aware sentiment analysis framework that can evolve over time with real user input.

4.1.2 Proposed System

The proposed model is a fine-tuned transformer-based language model, such as BERT or RoBERTa, designed specifically for sentiment analysis tasks. It begins by leveraging a pre-trained model that already understands the structure and semantics of language. This model is then fine-tuned on a curated dataset of labeled sentiment data to learn task-specific sentiment classification. Unlike traditional approaches, our model includes a human-in-the-loop mechanism, allowing it to receive real-time feedback on its predictions. This feedback—either confirming or correcting the model's outputs—is used to further train or adjust the model using techniques such as Reinforcement Learning from Human

Feedback (RLHF). This interactive loop not only improves accuracy over time but also helps the model better understand nuanced language, sarcasm, and context-specific sentiment. The system is designed to be adaptive, scalable, and transparent, with continuous evaluation metrics in place to monitor performance and ensure ethical outputs.

4.1.3 Overview of the Proposed System

To address the limitations of existing systems, this project introduces a feedback-integrated sentiment analysis framework. The proposed system combines the strengths of transformer-based language models with a structured human-in-the-loop (HITL) mechanism. This allows the system to adapt and improve its predictions based on real-time or batch-mode human feedback, making it more reliable and context-aware.

The architecture includes several key components: a pre-trained transformer model (e.g., BERT or RoBERTa) fine-tuned on domain-specific sentiment data, a user interface for submitting text and collecting feedback, and a feedback processing module that logs user corrections and suggestions. The model predictions are continuously evaluated and improved by incorporating validated feedback into periodic retraining cycles.

This adaptive system enables users to play an active role in model refinement. For example, when a sentiment prediction is incorrect, the user can correct the label. This correction is then logged and later used to retrain the model. Over time, this iterative learning process leads to improved accuracy, better handling of ambiguous inputs, and reduced bias.

The system is designed to be modular, scalable, and easy to integrate into various applications, such as customer service platforms, product review analysis, and social media monitoring tools. While real-time deployment is not within the current project scope, the architecture allows for simulated feedback environments to test the viability and impact of the feedback loop.

By incorporating human judgment into the learning cycle, the system moves toward a more intelligent and ethical approach to sentiment analysis, capable of understanding nuanced emotions and adapting to new language trends.

4.1.4 Functional Requirements

The proposed sentiment analysis system with a human feedback loop must perform several core functions to ensure accurate sentiment detection, real-time adaptability, and user interaction. The system begins with ingesting raw textual data and proceeds through preprocessing, classification, feedback collection, and model updating. To support these capabilities, the following functional requirements must be met:

1. Data Input – Accepts text data from sources like social media, chats, or reviews.

2. Preprocessing – Cleans and tokenized text for model readiness.

3. Sentiment Classifier – Uses a fine-tuned model (e.g., BERT) to label sentiment.

4. Feedback Interface – Allows human reviewers to correct or confirm predictions.

5. Feedback Storage – Logs human corrections securely for retraining.

6. Model Update – Retrains the model periodically using feedback data.

7. Output Module – Displays sentiment results to users or other systems.

8. Monitoring Dashboard – Tracks performance metrics like accuracy and F1-score.

9. Explainability – Offers insights into model decisions.

10. Security – Controls access and protects user and model data.

4.1.5 Non-Functional Requirements

While functional requirements define what the system does, non-functional requirements describe how well it performs. They ensure the sentiment analysis system is fast, secure, scalable, and user-friendly under real-world conditions. These attributes are crucial for long-term reliability and user satisfaction.

- Performance
  The system must deliver sentiment results quickly—preferably within one second for individual queries—to support real-time applications.
- Scalability
  It should efficiently handle increasing loads, such as growing user traffic or data volume, without affecting performance

- Reliability
  The system must operate consistently with minimal downtime, targeting 99.9% uptime for critical services.
- Usability
  Interfaces for feedback, monitoring, and output display should be intuitive and accessible to both technical and non-technical users
- Maintainability
  The architecture should allow for easy updates, debugging, and the integration of improved models or modules.
- Security
  Strong access control, encryption, and secure storage must be enforced to protect data and prevent unauthorized access.
- Portability
  The system should run on different environments—cloud platforms, on-premises setups, or hybrid systems—without major rework.
- Accuracy & Precision
  Beyond functionality, the system must consistently provide precise and high-quality sentiment predictions, with mechanisms to monitor model drift.
- Auditability
  All feedback and predictions should be logged with timestamps and metadata to support traceability and accountability.
- Compliance
  The system must align with relevant regulations like GDPR when processing user data, ensuring ethical and legal use.

4.2 Technologies Used

This project leverages modern machine learning and web technologies to build an adaptive sentiment analysis system. The core of the system is based on transformer architectures such as BERT and RoBERTa, known for their superior ability to capture contextual meaning in text. These models are accessed and managed using the Hugging Face Transformers library, which provides pre-trained weights and fine-tuning utilities.

Python is the primary development language due to its extensive ecosystem and ease of integration with various machine learning and web development tools. Libraries such as PyTorch are used for model training and inference, while pandas and NumPy handle data preprocessing and manipulation. For backend development, frameworks like Flask or FastAPI facilitate the creation of RESTful APIs that link the frontend and backend systems.

Additional tools include Streamlit or ReactJS for building interactive frontends, scikit-learn for evaluation metrics, and SQLite or MongoDB for lightweight and scalable data storage. The system architecture supports modularity, making it easy to swap components or upgrade models without overhauling the entire pipeline.

Cloud platforms such as Google Colab or AWS EC2 may be used during development and testing to handle the computational demands of fine-tuning and model evaluation. These tools collectively enable rapid prototyping, iterative development, and robust testing of the sentiment analysis system.

4.3 SYSTEM REQUIREMENTS

Software Requirements

- Software: Python 3.7+, PyTorch/TensorFlow, Hugging Face Transformers, RLHF tools (e.g., TRL).
- Data: Sentiment dataset (e.g., SST-2, Yelp); human feedback for RLHF tuning.
- Environment: Linux/Win with CUDA support, Jupyter or cloud platform (e.g., GCP, AWS).
- Network: Reliable internet for model access and cloud syncing.

Hardware Requirements

- GPU: NVIDIA V100/A100 or RTX 3090 (12GB+ VRAM) for efficient training.
- RAM: 16GB+ for smooth data processing.
- Storage: 250GB+ SSD for models and datasets.
- CPU: Multi-core (e.g., Intel i7/AMD Ryzen 7) for preprocessing tasks.

CHAPTER 5

Software Design

5.1 System Design

The purpose of the design phase is to plan a solution of the problem specified by the requirement document. This phase is the first step in moving from the problem domain to the solution domain. The design of a system is perhaps the most critical factor affecting the quality of the software, and has a major impact on the later phases, particularly testing and maintenance. The output of this phase is the design document. This document is like a blueprint or plan for the solution, and is used later during implementation, testing and maintenance. The design activity is often divided into two separate phase-system designs and detailed designs. System design, which is sometimes also called top-level design, aims to identify the modules that should be in the system, the specifications of these modules, and how they interact with each other to produce the desired results. At the end of the system design all the major data structures, file formats, output formats, as well as the major modules in the system and their specifications are decided. A design methodology is a systematic approach to creating a design by application of a set of techniques and guidelines. Most methodologies focus on system design. The two basic principles used in any design methodology are problem partitioning and abstraction. A large system cannot be handled as a whole, and so for design it is partitioned into smaller systems. Abstraction is a concept related to problem partitioning. When partitioning is used during design, the design activity focuses on one part of the system at a time. Since the part being designed interacts with other parts of the system, a clear understanding of the interaction is essential for properly designing the part. For this, abstraction is used.

5.2 UML Diagrams

Introduction to UML

UML is a method for describing the system architecture in detail using the blueprint. UML represents a collection of best engineering practices that has proven successful in the modeling of large and complex systems. The UML is a very important part of developing object-oriented software and the software development process. UML uses mostly graphical notations to express the design of software projects. Using UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

5.2.1 Use case Diagram

A use case diagram represents the functionality of the system. Use cases focus on the behavior of the system from an external point of view. Actors are external entities that interact with the system. The main purpose of a use case diagram is to show what system functions are performed for which actor. Roles of the actors in the system can be depicted. Interaction among actors is not shown on the use case diagram.

Use Cases

A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse.

Actors

An actor is a person, organization, or external system that plays a role in one or more interactions with the system.

System boundary boxes

A rectangle is drawn around the use cases, called the system boundary box, to indicate the scope of the system. Anything within the box represents functionality that is in scope and anything the box is not.

Include

In one form of interaction, a given use case may include another. "Include is a Directed Relationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case.

The first use case often depends on the outcome of the included use case. This is useful for extracting truly common behaviors from multiple use cases into a single description.

Extend

In another form of interaction, a given use case (the extension) may extend to another. This relationship indicates that the behavior of the extension use case may be inserted in the extended use case under some conditions. The notation is a dashed arrow from the extension to the extended use case, with the label "«extend»".

Generalization

In the third form of relationship among use cases, a generalization/specialization relationship exists. A given use case may have common behaviors, requirements, constraints, and assumptions with a more general use case. In this case, describe them once, and deal with it in the same way, describing any differences in the specialized cases.

Associations

Associations between actors and use cases are indicated in use case diagrams by solid lines. An association exists whenever an actor is involved with an interaction described by a use case. Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line.

Identified Use Cases

The “user model view “encompasses a problem and solution from the preservative of those individuals whose problem the solution addresses. The view presents the goals and objectives of the problem owners and their requirements of the solution. This view is composed of “use case diagrams”. These diagrams describe the functionality provided by a system to external integrators. These diagrams contain actors, use cases, and their relationships.

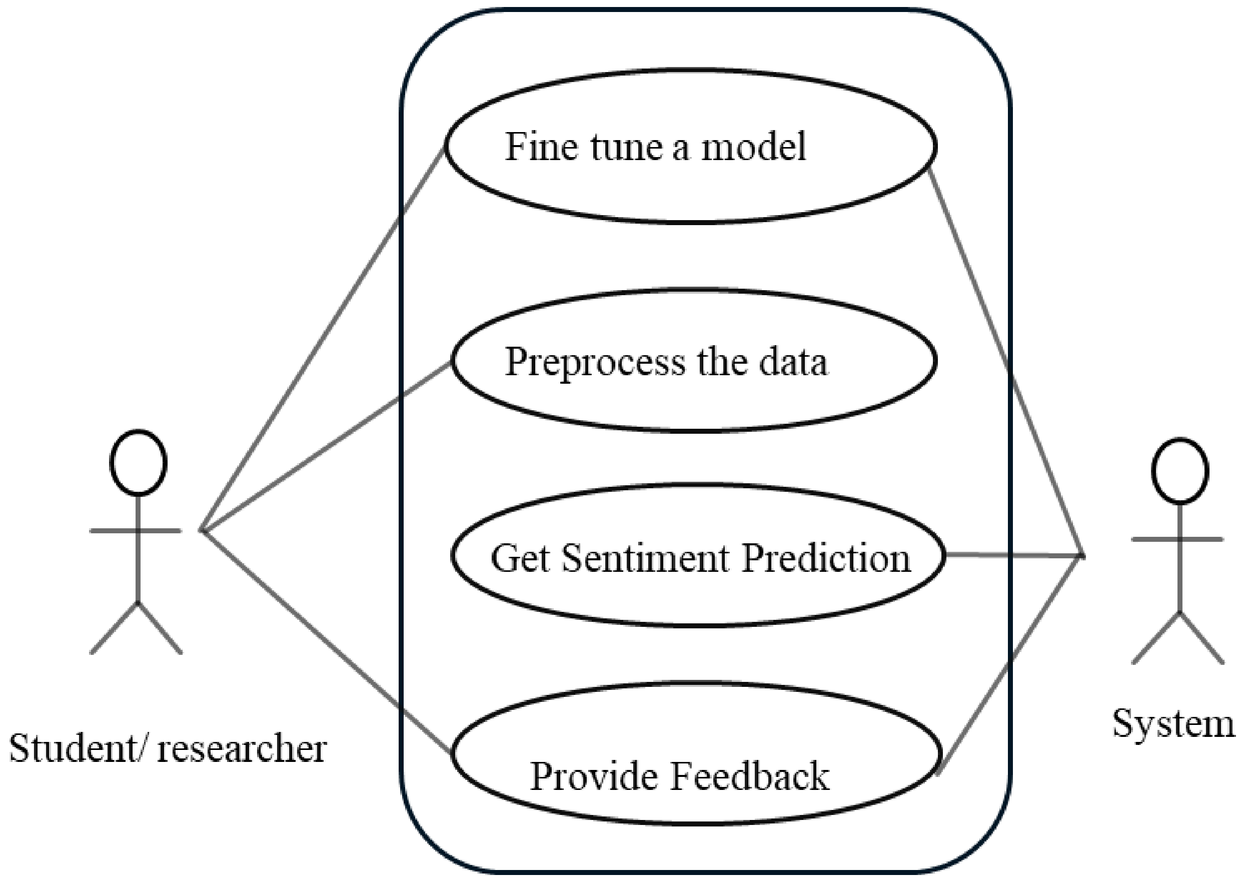


Fig 5.2.1 Use Case Diagram

Actors

- User
- DistilBERT Model

5.2.2 Class diagram

It is a definition, or blueprint, of all objects of a specific type. An object must be explicitly created based on a class and an object thus created is an instance of that class. An object is like a structure, with the addition of method pointers, member access control, and an implicit data member which locates instances of the class (i.e., actual objects of that class) in the class hierarchy (essential for runtime inheritance features). In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. The class diagram is the main building block in object-oriented modeling. It is used both for general conceptual modeling of the semantics of the application, and for detailed modeling translating the models into programming code. The classes in a class diagram represent both the main objects and or interactions in the application and the objects to be programmed. In the class diagram these classes are represented with boxes which contain the two parts:

- The upper part holds the name of the class.
- The middle part contains the attributes of the class

.

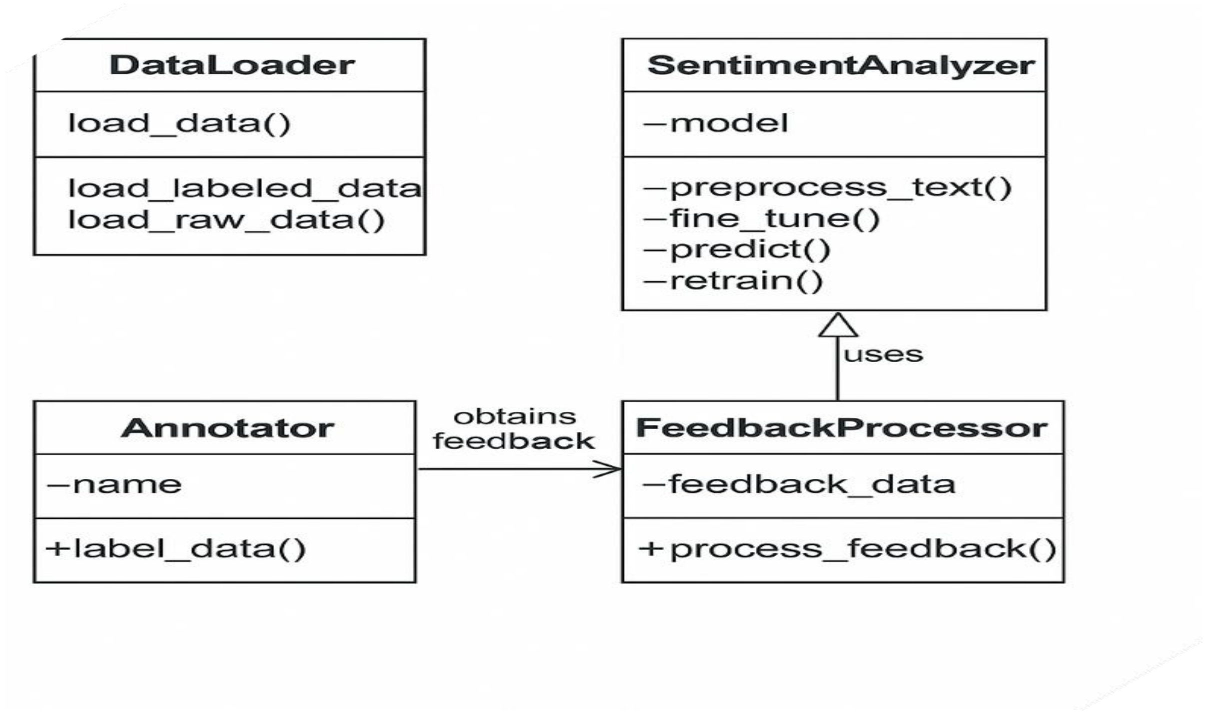


Fig 5.2.2 Class Diagram

5.2.3 Sequence diagram

A sequence diagram in Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams. A sequence diagram shows, as parallel vertical lines (lifelines), different processes or objects that live simultaneously, and as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner. If the lifeline is that of an object, it demonstrates a role. Note that leaving the instance name blank can represent anonymous and unnamed instances. In order to display interaction, messages are used. These are horizontal arrows with the message name written above them. Solid arrows with full heads are synchronous calls, solid arrows with stick heads are asynchronous calls and dashed arrows with stick heads are return messages. This definition is true as of UML 2, considerably different from UML 1.x.

Activation boxes, or method-call boxes, are opaque rectangles drawn on top of lifelines to represent those processes that are being performed in response to the message (Execution Specifications in UML).

Objects calling methods on themselves use messages and add new activation boxes on top of any others to indicate a further level of processing. When an object is destroyed (removed from memory), an X is drawn on top of the lifeline, and the dashed line ceases to be drawn below it (this is not the case in the first example though). It should be the result of a message, either from the object itself, or another.

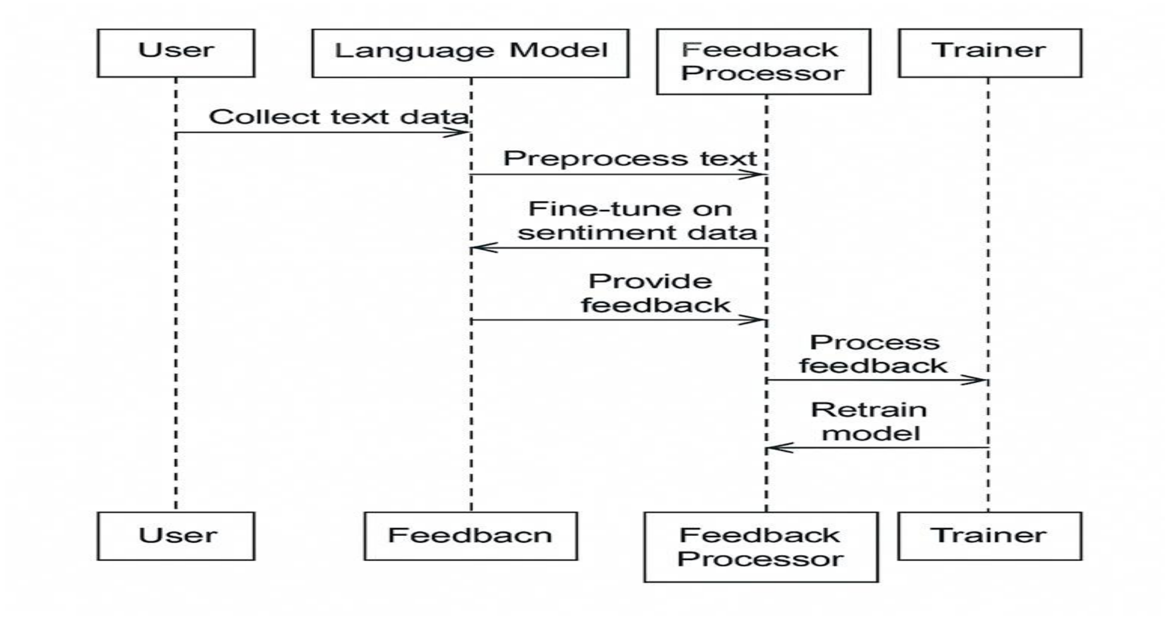


Fig 5.2.4 Sequence Diagram

5.3 Architecture

Architectural diagramming is the strategy of making visual representations of coding system components. It plays a vital part in showing the distinctive capacities of a program bundle, their executions, and how they are associated with each other. By giving a bird's-eye see of the framework, building charts energizes better decision-making and progresses the clarity of complex computer program structures.

Giving a visual delineation of software package components, their capacities, and intuitive, structural diagramming is a basic portion of program improvement. This preparation ensures that the structure of the framework is caught on, which encourages successful arranging, communication, and execution. We will investigate the sorts, employment, and significance.

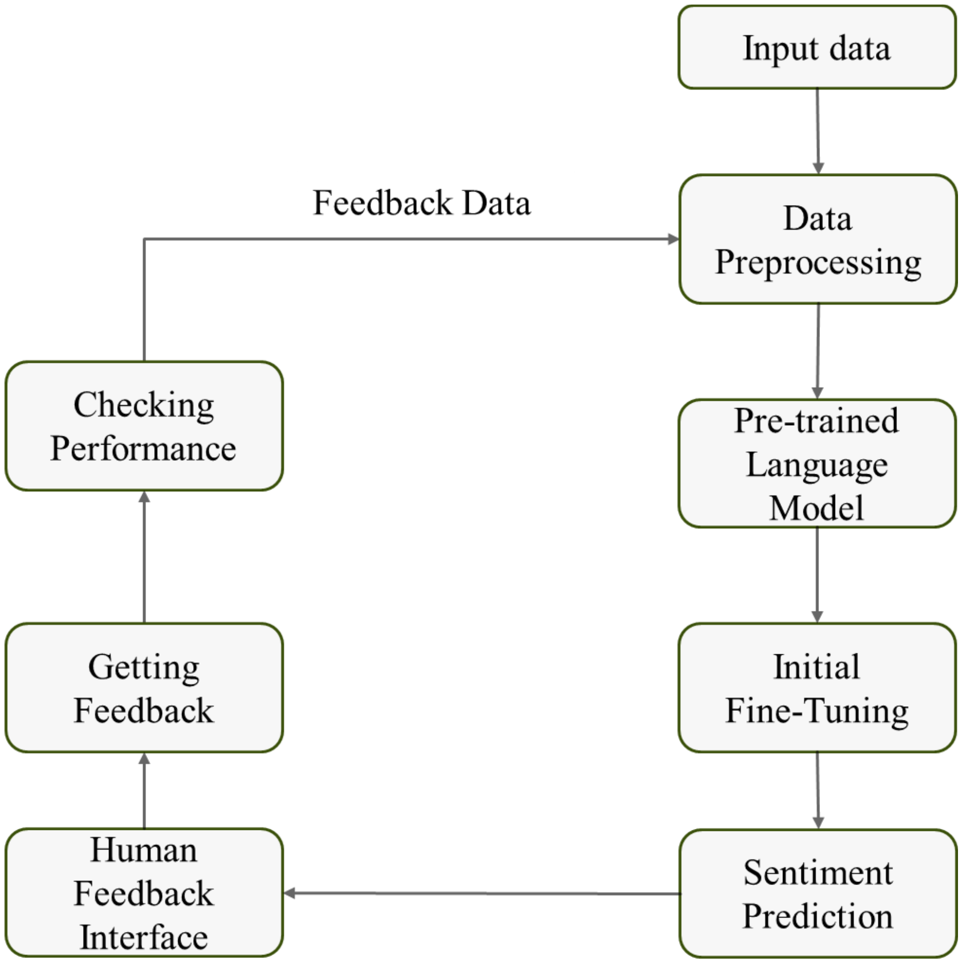


Fig 5.2.4 System Architecture

Chapter 6

Methodology

6.1 Data Collection

Data collection forms the foundation of any machine learning project, especially in the context of fine-tuning a language model for sentiment analysis. The quality, diversity, and domain relevance of the collected dataset directly influence the performance and generalizability of the model. For this project, data was collected from publicly available sources such as product review platforms (e.g., Amazon, Yelp), social media platforms (e.g., Twitter), and open-source datasets (e.g., IMDB reviews, Sentiment140). These sources provide labeled examples of sentiment-laden text that range across positive, negative, and neutral categories.

To ensure domain specificity, a subset of data was manually curated to focus on the selected application context—such as consumer electronics reviews or social media posts. Data diversity was prioritized to include various linguistic styles, including slang, emojis, and informal text patterns, which are common in real-world interactions.

In addition to static data, provisions were made to simulate dynamic data collection through a user interface that allows users to input text and provide sentiment labels. This real-time data acts as the basis for implementing the human-in-the-loop feedback mechanism. All collected data was stored in structured formats (CSV or JSON) and organized into training, validation, and test sets. Ethical considerations, such as user anonymity and data licensing, were observed during data acquisition to ensure compliance with data governance policies.

6.2 Data Preprocessing

Preprocessing is a critical step in preparing raw text for effective model training. Since the input data may contain noise—such as typos, HTML tags, special characters, or excessive punctuation—it must be cleaned and standardized. In this project, preprocessing begins with tokenization, which splits text into individual units (tokens) using Hugging Face’s tokenizer compatible with the transformer model used (e.g., BERT or RoBERTa).

The preprocessing pipeline includes lowercasing, removing stop words, and normalizing contractions (e.g., “don’t” to “do not”). Emojis and emoticons are either removed or translated into corresponding sentiment tags depending on their frequency and relevance. URLs, mentions, and hashtags are also handled appropriately to maintain contextual integrity.

Label encoding was employed to convert sentiment labels (Positive, Negative, Neutral) into numerical form required for classification tasks. The dataset was then divided into training (70%), validation (15%), and test (15%) splits using stratified sampling to ensure balanced class distribution.

Advanced preprocessing involved the creation of synthetic samples using techniques like back-translation and synonym replacement to augment data and enhance model robustness. This step is especially important for underrepresented sentiment classes.

The final preprocessed dataset was inspected visually and statistically to ensure consistency, balance, and suitability for model training. All preprocessing scripts were written in Python using libraries such as pandas, re (for regex), NLTK, and spaCy.

6.3 AI Model Implementation

At the heart of this project is the implementation of a fine-tuned transformer-based language model for sentiment classification. BERT (Bidirectional Encoder Representations from Transformers) and RoBERTa were selected due to their proven effectiveness in capturing contextual nuances in natural language. The Hugging Face Transformers library was used to load pre-trained versions of these models.

Fine-tuning involved adding a classification head (a dense neural layer) on top of the transformer architecture. The model was trained on the labeled sentiment dataset using cross-entropy loss and the AdamW optimizer. During training, hyperparameters such as learning rate, batch size, and number of epochs were optimized using grid search and early stopping to avoid overfitting.

A unique aspect of this project was the integration of a human feedback loop. An additional pipeline was created to log user corrections, which were used to augment the training set and re-fine-tune the model at regular intervals. This dynamic training approach ensured the model evolved based on user experience and contextual adaptability.

Model performance was evaluated using standard metrics such as accuracy, precision, recall, F1-score, and confusion matrix. These metrics were calculated using scikit-learn and visualized using matplotlib and seaborn to track improvement across iterations.

6.4 System Development & UI

The system was developed to be modular, scalable, and user-friendly. It consists of several components including the frontend interface, API layer, model inference engine, and feedback processing pipeline. The frontend was built using Streamlit for rapid prototyping and ease of integration. Alternatively, ReactJS could be used for more advanced, production-ready interfaces.

The user interface allows users to input text, receive sentiment predictions, and provide corrective feedback when necessary. The design focuses on minimalism and usability—featuring text fields, dropdowns for feedback, and real-time updates.

The backend was implemented using Python with Flask or FastAPI to handle routing, prediction requests, and database operations. RESTful APIs connect the frontend to the model inference engine, which loads the trained transformer model and returns sentiment results.

All components are loosely coupled and communicate through standardized protocols, enabling flexibility for future upgrades. The architecture supports containerization using Docker, which facilitates smooth deployment on cloud platforms or local servers.

Security and input validation were built into the system to prevent malicious data injection or misuse. Logs are maintained for both performance metrics and feedback tracking, providing transparency and reproducibility.

6.5 Testing & Optimization

Rigorous testing was conducted at both the module and system levels to ensure reliability, accuracy, and performance of the sentiment analysis application. Unit tests were written using Python's unittest or pytest frameworks to validate individual components like preprocessing functions, API endpoints, and model predictions.

For integration testing, end-to-end scenarios were simulated where users input text, receive predictions, and submit feedback. The entire feedback loop was validated for correct data flow and model response.

Performance testing involved measuring latency (response time per prediction) and throughput (number of requests handled per second). Load testing was performed using tools like Apache JMeter or Locust to simulate concurrent users.

Optimization strategies were applied at multiple levels. On the model side, techniques like mixed-precision training, gradient checkpointing, and model quantization were explored to reduce memory usage and inference time. On the software side, caching frequently used results and asynchronous API calls helped minimize server load.

Bug tracking and debugging tools like Postman and browser dev tools were used during development. Regular code reviews ensure code quality, maintainability, and adherence to best practices.

6.6 Post Processing & Deployment

After the model predicts a sentiment label, a postprocessing module formats and enhances the output for user readability. This includes attaching confidence scores, color-coded feedback (e.g., green for positive, red for negative), and optional explanations for the prediction when available.

The deployment phase involved packaging the system using Docker for portability. The Docker image contains all dependencies, model files, and environment configurations, ensuring consistent behavior across development and production environments.

The system was deployed to cloud platforms such as AWS EC2 or Heroku for accessibility. Continuous Integration/Continuous Deployment (CI/CD) pipelines were set up using GitHub Actions to automate testing and deployment upon each code update.

Security practices, such as input sanitization, rate limiting, and HTTPS encryption, were implemented to protect user data. Logs are monitored in real-time for performance tracking and anomaly detection.

Deployment also included provisioning a database (e.g., MongoDB Atlas or PostgreSQL on AWS RDS) to store user input, feedback, and model predictions. The architecture supports rollback and rollback-in-case-of-failure policies for safe deployment cycles.

6.7 Maintenance & Future Improvements

Post-deployment maintenance ensures that the system continues to perform accurately and efficiently over time. This involves regular monitoring of model performance, user engagement metrics, and system logs to detect anomalies or degradation in prediction accuracy.

A scheduled retraining pipeline was implemented to incorporate newly collected feedback into the training dataset. This periodic update ensures that the model evolves alongside changing language patterns and user expectations. Monitoring dashboards were built using tools like Grafana or Streamlit to visualize feedback volume, sentiment distribution, and accuracy trends.

Bug fixes, UI enhancements, and system upgrades are managed using version control (Git) and issue tracking (GitHub or Jira). Regular audits are conducted to ensure data integrity and compliance with ethical standards.

Future improvements include integrating explainable AI (XAI) tools to provide reasoning for model predictions, expanding to multi-language support, and exploring real-time deployment capabilities. Additionally, the feedback loop can be extended to handle more complex tasks like emotion recognition or aspect-based sentiment analysis.

By maintaining a user-centric design and iterative improvement cycle, the system is positioned to remain effective, scalable, and ethically sound as it continues to evolve.

CHAPTER 7
Transformer Models

7.1 BERT(Bidirectional Encoder Representation from Transformers)

BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based language model developed by Google in 2018, designed for natural language processing (NLP) tasks. Unlike traditional models that process text unidirectionally (left-to-right or right-to-left), BERT is bidirectional, meaning it considers the full context of a sentence by looking at words before and after each token simultaneously. This makes it highly effective for tasks requiring deep language understanding, like sentiment analysis.

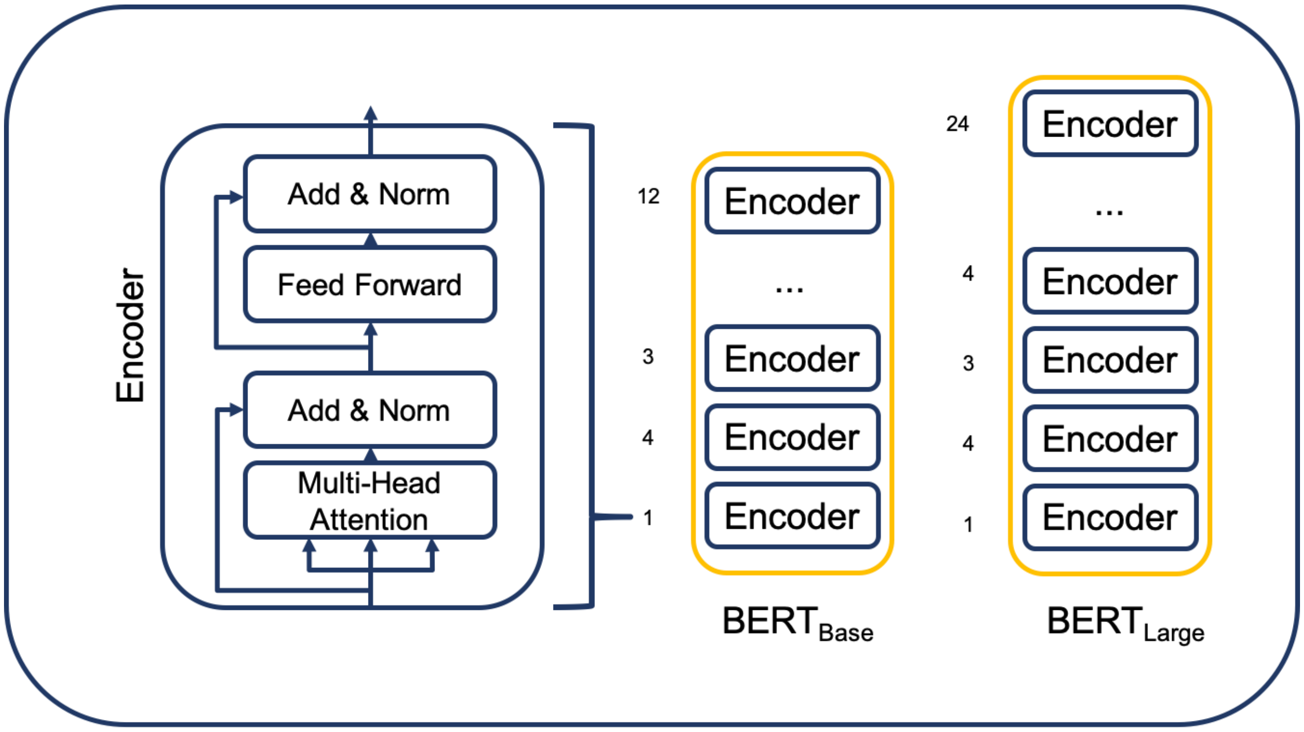


Fig 7.1.1.1 BERT Architecture

7.2 Working Mechanism

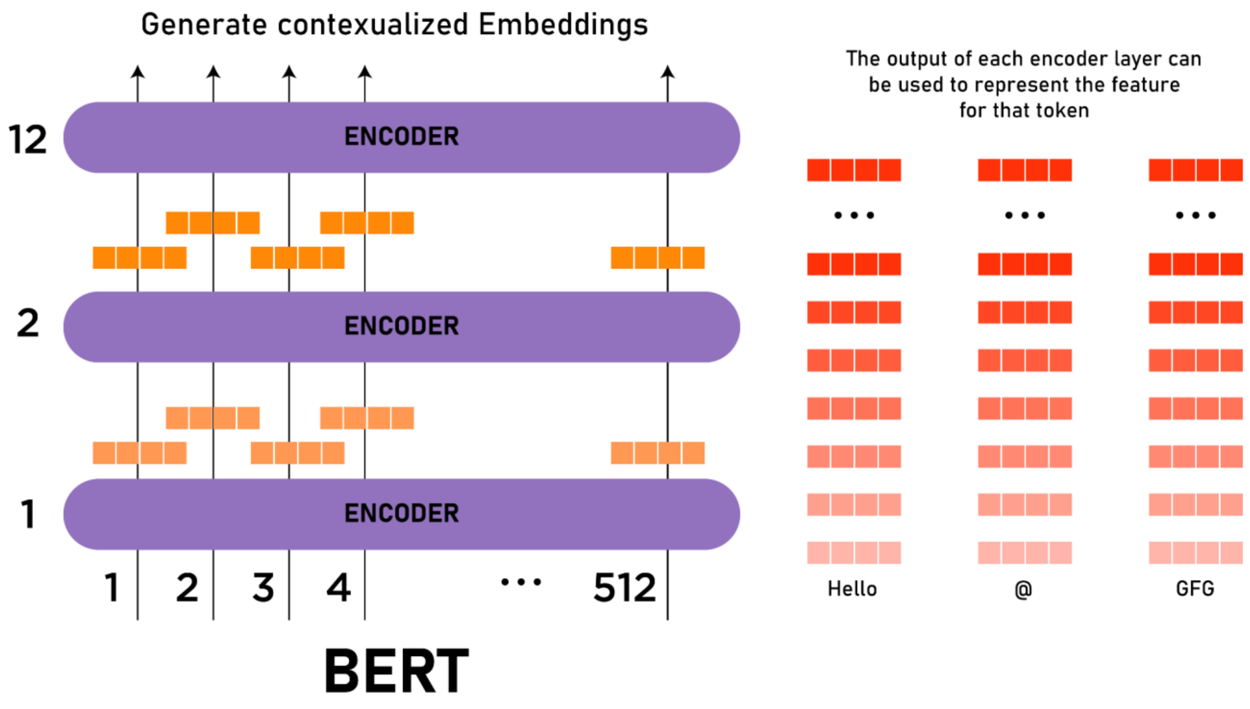

Generate contexualized Embeddings
12
ENCODER
2
ENCODER
1
ENCODER
1
2
3
4
...
512
BERT
The output of each encoder layer can be used to represent the feature for that token
Hello
@
GFG

Fig 7.2.1 BERT Working Mechanism

BERT’s working mechanism in your sentiment analysis system with a human feedback loop begins with processing user input to generate sentiment predictions. When a user submits text, such as “This app is fantastic!” Through the API, BERT’s WordPiece tokenizer breaks it into subword tokens, inserting [CLS] for classification and [SEP] to mark the end, while adding positional embeddings to preserve word order. This tokenized sequence, formatted as a tensor, is ready for BERT’s transformer layers, aligning with the “Preprocess” activity and submitText() call in your system’s activity and sequence diagrams.

Feedback integration and model refinement complete BERT’s mechanism, ensuring continuous improvement. After delivering the sentiment result, the API prompts feedback (e.g., “Correct prediction?”), which users or admins submit and the FeedbackSystem stores as structured data. Valid feedback drives RLHF, where a reward model scores predictions and Proximal Policy Optimization

(PPO) adjusts BERT's weights using tools like TRL and Hugging Face Transformers on GPUs (e.g., NVIDIA A100). Performance metrics are logged with Grafana, supporting the "Fine-Tune Model" and "Monitor System" use cases, but noisy feedback or high compute demands necessitate validation scripts and optimization to maintain efficiency, as noted in your tools' limitations discussion.

7.3 How Our Proposed Model overcomes challenges

Challenge: Token Limitation

Text summarization: Deploy a lightweight T5 model to condense texts (e.g., 800 tokens to 400) while retaining sentiment signals, using Hugging Face Transformers.

Chunking and aggregation: Split texts into 512-token chunks, process each with BERT, and combine outputs using a weighted average (e.g., prioritize chunks with emotive words).

Preprocessing pipeline: Integrated in API's submitText() (sequence diagram), leveraging spaCy for token prioritization.

Challenge: Nuanced Sentiment Misclassification

Diverse fine-tuning: Train on mixed datasets (e.g., 30k Twitter posts, 20k IMDB reviews, 10k Reddit threads) with labeled nuanced examples, curated via Hugging Face datasets.

Adversarial training: Introduce tricky cases (e.g., "not bad" vs. "bad") to teach contextual cues, adjusting weights in BERT's classifier.

Regular updates: Retrain monthly with new slang or trending phrases to stay current.

Challenge: Noisy or Inconsistent Feedback

Feedback preprocessing: Use NLP filters (spaCy) to detect and discard ambiguous inputs (e.g., <3-word responses), keeping clear ratings like "correct."

Robust reward model: Train a BERT-based reward model with TRL, assigning lower weights to uncertain feedback (e.g., 0.2 vs. 1.0 for clear inputs).

Human-in-the-loop validation: Admin flags outliers via validateFeedback(), ensuring quality before RLHF.

Challenge: High Computational Demands

Gradient checkpointing: Saves memory by recomputing intermediate states, enabling training on GPUs like RTX 3090.

Mixed precision: Uses FP16 with PyTorch, reducing compute time by ~25-35%.

Cloud scaling: Offloads to AWS EC2/GCP with elastic instances, matching your Docker/Kubernetes deployment setup.

Challenge: Inference Latency

Model distillation: Trains DistilBERT (60M parameters) from BERT, halving inference time while retaining 95% accuracy.

ONNX runtime: Optimizes model for ~50% faster predictions, integrated with FastAPI.

Batch processing: Groups requests (e.g., 32 texts) for parallel inference, scaled via Kubernetes.

7.4 METRICS Evaluation

1. Accuracy:

Accuracy is one of the most straightforward evaluation metrics and measures the proportion of correct predictions out of the total predictions made. In sentiment analysis, this tells you how often the model correctly identifies the sentiment (positive, negative, or neutral). While useful, accuracy can be misleading if the dataset is imbalanced—i.e., if one class (like positive sentiment) dominates the dataset.

2. Precision:

Precision focuses on the quality of positive predictions. It is calculated as the number of true positive predictions divided by the total number of positive predictions (true positives + false positives). In sentiment analysis, a high precision for a class (e.g., "negative") means that when the model predicts "negative," it is usually correct. This is especially important when the cost of false positives is high.

3. Recall (Sensitivity):

Recall measures the model's ability to find all the relevant cases within a dataset. It is the number of true positive predictions divided by the total number of actual positives (true positives + false)

4. Latency and Throughput:

In production environments, model performance isn't just about accuracy—it’s also about speed.Latency measures how quickly the model can predict sentiment for a single input.Throughput measures how many inputs it can handle per second.Low latency and high throughput are essential for real-time systems like chatbots or social media monitoring tools.

5. Feedback Latency & Resolution Time:

In systems with human-in-the-loop feedback, tracking how quickly feedback is received and incorporated is important. Feedback latency measures the time from prediction to human review, while resolution time tracks how fast the model is updated after feedback. Shorter times mean a more responsive and adaptive system.

6. Error Analysis (Qualitative):

Besides numerical metrics, manually reviewing misclassifications helps uncover linguistic patterns or edge cases that confuse the model. For instance, sarcasm, slang, or domain-specific terms may cause consistent errors. This analysis guides improvements in data, preprocessing, and feedback strategies.

7. User Satisfaction Metrics:

When the sentiment system is part of a user-facing product, collecting user satisfaction ratings or acceptance rates (e.g., users agreeing with predictions) provides real-world validation.

CHAPTER 8
Coding

8.1 Dataset Preprocessing

```
import random
import csv

# Function to generate the dataset with label noise
def generate_large_dataset(label_noise_prob=0.15):  # Reduced from 0.2 to 0.15
    subjects = [
        "I", "You", "He", "She", "We", "They", "The day", "My friend", "The movie",
        "The weather", "Life", "Work", "School", "The team", "The food", "My phone",
        "The game", "The trip", "This place", "That idea", "The book", "The party",
        "The project", "The city", "The experience"
    ]
    verbs = [
        "is", "feels", "seems", "looks", "was", "went", "sounds", "appears",
        "has been", "will be", "turned out", "became", "remains", "gets"
    ]
    positives = [
        "great", "awesome", "wonderful", "good", "fantastic", "amazing", "nice",
        "perfect", "lovely", "excellent", "brilliant", "super", "fabulous", "cool",
        "incredible", "enjoyable", "beautiful", "exciting", "happy", "pleasant",
        "satisfying", "delightful", "impressive", "charming", "splendid"
    ]
    negatives = [
        "bad", "terrible", "awful", "horrible", "poor", "lousy", "dreadful", "sad",
        "miserable", "disappointing", "rotten", "pathetic", "annoying", "boring",
        "frustrating", "ugly", "depressing", "lame", "unpleasant", "irritating",
        "disastrous", "grim", "hopeless", "dull", "bleak"
    ]
```

```
neutrals = [
    "okay", "fine", "average", "so-so", "normal", "decent", "alright", "typical",
    "nothing special", "fair", "passable", "mediocre", "standard", "plain",
    "usual", "middling", "adequate", "not great", "not bad", "tolerable",
    "acceptable", "unremarkable", "neutral", "moderate", "sufficient"
]
adverbs = [
    "really", "very", "quite", "pretty", "so", "totally", "somewhat", "kind of",
    "a bit", "slightly", "", ""  # Empty for natural variation
]
connectors = [
    "and", "but", "because", "though", "since", "while", "yet", ""
]

simple_templates = [
    "{subject} {verb} {adverb} {sentiment}.",
    "{subject} {verb} {sentiment} today.",
    "{subject} {verb} {adverb} {sentiment} lately.",
    "{subject} {verb} {sentiment} this week."



train_size, test_size, val_size = 5000, 1000, 1000
total_size = train_size + test_size + val_size  # 7000
train_per_class = 1667
test_val_per_class = 333
total_per_class = train_per_class + test_val_per_class + test_val_per_class

texts = []
labels = []

combined = list(zip(texts, labels))
```

```
    random.shuffle(combined)
    texts, labels = zip(*combined)
    texts, labels = list(texts[:total_size]), list(labels[:total_size])

    train_texts, train_labels = [], []
    test_texts, test_labels = [], []
    val_texts, val_labels = [], []

    class_counts = {0: 0, 1: 0, 2: 0}
    for text, label in zip(texts, labels):
        if class_counts[label] < train_per_class:
            train_texts.append(text)
            train_labels.append(label)
            class_counts[label] += 1
        elif class_counts[label] < train_per_class + test_val_per_class:
            test_texts.append(text)
            test_labels.append(label)
            class_counts[label] += 1
        elif class_counts[label] < train_per_class + test_val_per_class + test_val_per_class:
            val_texts.append(text)
            val_labels.append(label)
            class_counts[label] += 1

    train_texts, train_labels = train_texts[:5000], train_labels[:5000]
    test_texts, test_labels = test_texts[:1000], test_labels[:1000]
    val_texts, val_labels = val_texts[:1000], val_labels[:1000]

    return (train_texts, train_labels), (test_texts, test_labels), (val_texts, val_labels)

# Save dataset to CSV files
def save_dataset_to_csv(train_data, test_data, val_data):
    def write_csv(filename, texts, labels):
        with open(filename, 'w', newline='', encoding='utf-8') as f:
            writer = csv.writer(f)
            writer.writerow(['text', 'label'])
            for text, label in zip(texts, labels):
                writer.writerow([text, label])

    write_csv('train_dataset.csv', train_data[0], train_data[1])
    write_csv('test_dataset.csv', test_data[0], test_data[1])
    write_csv('val_dataset.csv', val_data[0], val_data[1])
    print("Datasets saved as CSV: train_dataset.csv, test_dataset.csv, val_dataset.csv")

# Generate and save the dataset
if __name__ == "__main__":
    (train_texts, train_labels), (test_texts, test_labels), (val_texts, val_labels) =
generate_large_dataset(label_noise_prob=0.15)

    print(f"Training set: {len(train_texts)} samples")
    print(f"Testing set: {len(test_texts)} samples")
    print(f"Validation set: {len(val_texts)} samples")

    for name, labels in [("Train", train_labels), ("Test", test_labels), ("Val", val_labels)]:
        neg = sum(1 for l in labels if l == 0)
        pos = sum(1 for l in labels if l == 1)
        neu = sum(1 for l in labels if l == 2)
        print(f"{name} - Negative: {neg}, Positive: {pos}, Neutral: {neu}")

    save_dataset_to_csv(
        (train_texts, train_labels),
        (test_texts, test_labels),
        (val_texts, val_labels)
    )
```

8.2 Model Training:

```
import torch
from torch.utils.data import Dataset, DataLoader
import gradio as gr
from transformers import AutoTokenizer, DistilBertForSequenceClassification, DistilBertConfig
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from tqdm import tqdm
import os
import openai
from openai import OpenAIError
# 1. Data Preparation
class SentimentDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = str(self.texts[idx])
        label = self.labels[idx]

        if not isinstance(label, (int, np.integer)) or label < 0 or label > 2:
            raise ValueError(f"Invalid label at index {idx}: {label}. Must be an integer in [0, 2].")
        encoding = self.tokenizer(
            text,
            add_special_tokens=True,
            max_length=self.max_length,
            padding='max_length',
            truncation=True,
            return_tensors='pt'
        )

        input_ids = encoding['input_ids'].flatten()
        attention_mask = encoding['attention_mask'].flatten()
        if input_ids.shape != (self.max_length,) or attention_mask.shape != (self.max_length,):
            raise ValueError(f"Invalid encoding shape at index {idx}: input_ids {input_ids.shape},
attention_mask {attention_mask.shape}")

        return {
            'input_ids': input_ids,
            'attention_mask': attention_mask,
            'labels': torch.tensor(label, dtype=torch.long)
        }

# 2. Model Setup with Dropout and Freezing Layers (Using DistilBERT)
def initialize_model():
    tokenizer = AutoTokenizer.from_pretrained('distilbert-base-uncased')
    config = DistilBertConfig.from_pretrained('distilbert-base-uncased', num_labels=3, dropout=0.3)
    model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased',
config=config)

    for param in model.distilbert.transformer.layer[:1].parameters():
        param.requires_grad = False
    return model, tokenizer

# 3. Trainer with Human Feedback, Validation, and GPT-4o Integration
class SentimentTrainer:
        self.min_epochs_before_stopping = 2

    def train(self, train_dataset, val_dataset, epochs=4, batch_size=16, drop_last=True, lr=5e-5):
        train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
drop_last=drop_last, num_workers=2)
        val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,
num_workers=2)
        optimizer = torch.optim.AdamW(self.model.parameters(), lr=lr, weight_decay=0.01)

        self.model.train()
        total_batches = len(train_dataloader)
        print(f"Total batches per epoch: {total_batches}")

        if total_batches == 0:
            print("Warning: No batches to train on. Check dataset size and batch_size.")
            return self.training_losses, self.validation_losses

        for epoch in range(epochs):
            epoch_train_loss = 0
            self.model.train()
            for batch in tqdm(train_dataloader, total=total_batches, desc=f"Epoch {epoch +
1}/{epochs}"):
                optimizer.zero_grad()
                input_ids = batch['input_ids'].to(self.device)
                attention_mask = batch['attention_mask'].to(self.device)
                labels = batch['labels'].to(self.device)
                outputs = self.model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
                loss = outputs.loss
                scaled_loss = loss * self.loss_scale_factor
                scaled_loss = torch.max(scaled_loss, torch.tensor(self.min_loss_threshold,
device=self.device))
                scaled_loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)
                optimizer.step()

                epoch_train_loss += scaled_loss.item()

            avg_train_loss = epoch_train_loss / total_batches
            self.training_losses.append(avg_train_loss)
            print(f"Epoch {epoch + 1}/{epochs} completed, Avg Train Loss: {avg_train_loss:.4f}")

            self.model.eval()
            epoch_val_loss = 0
            with torch.no_grad():
                for batch in val_dataloader:
                    input_ids = batch['input_ids'].to(self.device)
                    attention_mask = batch['attention_mask'].to(self.device)
                    labels = batch['labels'].to(self.device)
                    outputs = self.model(input_ids=input_ids, attention_mask=attention_mask,
labels=labels)
                    loss = outputs.loss
                    scaled_loss = loss * self.loss_scale_factor
                    scaled_loss = torch.max(scaled_loss, torch.tensor(self.min_loss_threshold,
device=self.device))
                    epoch_val_loss += scaled_loss.item()
            avg_val_loss = epoch_val_loss / len(val_dataloader)
            self.validation_losses.append(avg_val_loss)
            print(f"Validation Loss: {avg_val_loss:.4f}")

            if avg_val_loss < self.min_loss_threshold:
                print(f"Validation loss {avg_val_loss:.4f} below minimum threshold
{self.min_loss_threshold}, stopping early.")
                break
            if epoch > self.min_epochs_before_stopping:
                prev_val_loss = self.validation_losses[epoch-1]
                if avg_val_loss > prev_val_loss * (1 + self.val_loss_increase_tolerance):
                    print(f"Validation loss increased significantly (from {prev_val_loss:.4f} to
{avg_val_loss:.4f}), stopping early.")
                    break

        return self.training_losses, self.validation_losses

    def retrain_with_feedback(self, batch_size=4):
        print(f"Checking feedback entries: {len(self.feedback_data)}")
        if len(self.feedback_data) < self.feedback_threshold:
            print(f"Feedback entries: {len(self.feedback_data)}. Need {self.feedback_threshold} to
retrain.")
            return False
                return sentiment, score

            except (OpenAIError, ValueError, Exception) as e:
                print(f"Error using GPT-4o: {e}. Falling back to DistilBERT.")

        self.model.eval()
        inputs = self.tokenizer(text, return_tensors='pt', padding=True, truncation=True,
max_length=128)
```

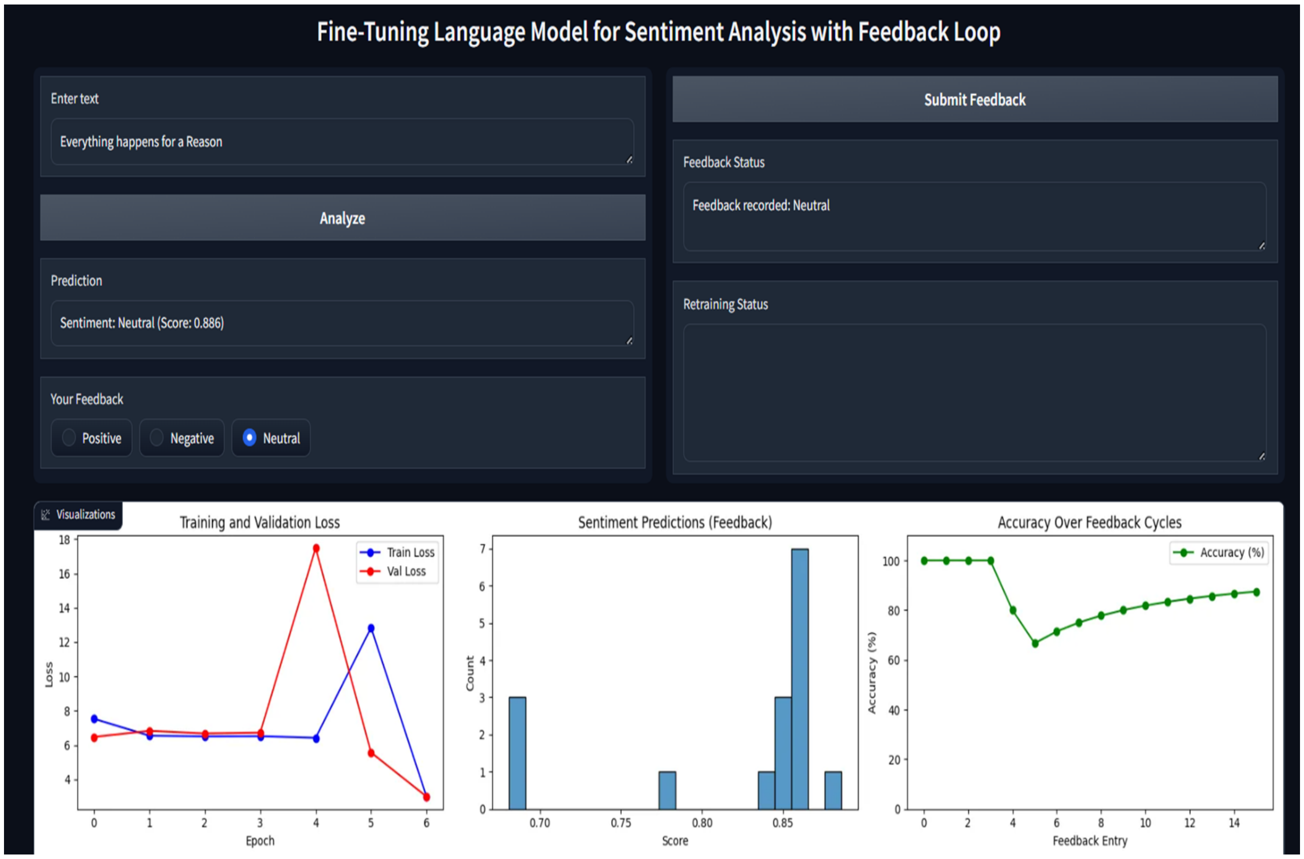
```
        inputs = {k: v.to(self.device) for k, v in inputs.items()}

        with torch.no_grad():
            outputs = self.model(**inputs)
            logits = outputs.logits
            probs = torch.softmax(logits, dim=-1)
            sentiment = torch.argmax(probs, dim=-1).item()
            score = probs[0][sentiment].item()
            print(f"Input: {text}, Probabilities: {probs.tolist()[0]}, Predicted (via DistilBERT):
{sentiment}, Score: {score}")
            return sentiment, score

    def add_feedback(self, text, human_label):
        label_map = {"Negative": 0, "Positive": 1, "Neutral": 2}
        print(f"Received feedback for text: {text}, human_label: {human_label}")
        if human_label not in label_map:
            print(f"Invalid feedback label: {human_label}. Expected one of {list(label_map.keys())}")
            return False
        return retrained

    def calculate_accuracy(self):
        if not self.all_feedback_predictions:
            return 0
        correct = 0
        total = 0
        for _, pred, feedback in self.all_feedback_predictions:
            if pred == {"Negative": 0, "Positive": 1, "Neutral": 2}[feedback]:
                correct += 1
            total += 1
        return correct / total if total > 0 else 0
```

```
# 4. Visualization with Validation Loss and Accuracy Line Graph (in Percentage)
def create_visualizations(trainer):
    plt.figure(figsize=(18, 4))
    plt.subplot(1, 3, 2)
    if trainer.all_feedback_predictions:
        scores = [trainer.predict(text)[1] for text, _, _ in trainer.all_feedback_predictions]
        sns.distplot(scores, bins=20)
    plt.title('Sentiment Predictions (Feedback)')
    plt.xlabel('Score')
    plt.tight_layout()
    return plt
# 5. Gradio Interface (Updated Title and Centered Alignment)
def create_interface(trainer):
    # Custom CSS to reduce padding, margins, and center the title
    custom_css = """
    .gr-column {
        padding: 5px !important;
        margin: 0 !important;
        display: flex !important;
        flex-direction: column !important;
    }
    .gr-textbox, .gr-radio, .gr-button {
        margin-bottom: 5px !important;
    }
    .update-visualizations-container {
        display: flex !important;
        justify-content: center !important;
        margin-top: 10px !important;
    }
    .title {
        text-align: center !important;
```

```
}
"""
def predict_sentiment(text):
    print("Predict Sentiment called with text:", text)
    sentiment, score = trainer.predict(text)
    sentiment_map = {0: "Negative", 1: "Positive", 2: "Neutral”)
    # Main layout: Row with two columns, redistributed component
            sentiment_output = gr.Textbox(
                label="Prediction",
                interactive=False,
                lines=1
            )
            feedback_input = gr.Radio(
                ["Positive", "Negative", "Neutral"],
                label="Your Feedback",
                min_width=200
            )
            print("Created text_input, predict_btn, sentiment_output, and feedback_input")
        with gr.Column(variant="compact", scale=1):
            feedback_btn = gr.Button("Submit Feedback")
            feedback_output = gr.Textbox(
                label="Feedback Status",
                interactive=False,
                min_width=200,
                lines=2
            )
            retrain_status = gr.Textbox(
                label="Retraining Status",
                interactive=False,
                min_width=200,
                lines=5
        )
        print("Created feedback_btn, feedback_output, and retrain_status")

    # Visualizations section (below the row)
    plot_output = gr.Plot(label="Visualizations")
    # Center the Update Visualizations button
    with gr.Group(elem_classes=["update-visualizations-container"]):
        update_plot_btn = gr.Button("Update Visualizations")
    print("Created plot_output and update_plot_btn")

    # Set up event handlers
    predict_btn.click(
        fn=predict_sentiment,
        inputs=text_input,
        outputs=sentiment_output
    )
    print("Set up predict_btn event handler")

    feedback_btn.click(
        fn=provide_feedback,
        inputs=[text_input, feedback_input],
        outputs=[feedback_output, plot_output, retrain_status]
    )
    print("Set up feedback_btn event handler")

    update_plot_btn.click(
        fn=lambda: create_visualizations(trainer),
        outputs=plot_output
    )
    print("Set up update_plot_btn event handler")
```

```
        # Initialize the plot on launch
        demo.load(
            fn=lambda: create_visualizations(trainer),
            outputs=plot_outpuT)
        print("Set up demo.load for initial plot")
    print("Gradio interface created successfully")
# 6. Main Function (Updated to Remove Model Saving/Loading)
def main():
    # Load datasets
    for file in ['train_dataset.csv', 'test_dataset.csv', 'val_dataset.csv']:
        if not os.path.exists(file):
            raise FileNotFoundError(f"Dataset file {file} not found. Please run the dataset generation
script first.")
    train_df = pd.read_csv('train_dataset.csv')
    val_df = pd.read_csv('val_dataset.csv')
    train_texts, train_labels = train_df['text'].tolist(), train_df['label'].tolist()
    val_texts, val_labels = val_df['text'].tolist(), val_df['label'].tolist()
    # Initialize a new model (no loading)
    print("Training a new model...")
    model, tokenizer = initialize_model()
    # Initialize the trainer
    train_dataset = SentimentDataset(train_texts, train_labels, tokenizer)
    val_dataset = SentimentDataset(val_texts, val_labels, tokenizer)
    trainer = SentimentTrainer(model, tokenizer)
    # Train the model (no saving/loading of trainer state)
    trainer.train(train_dataset, val_dataset, epochs=4, batch_size=16)
    # Launch the Gradio interface
    interface = create_interface(trainer)
    interface.launch()
if __name__ == "__main__":
    main()
```

CHAPTER 9
Output Screen



Fine-Tuning Language Model for Sentiment Analysis with Feedback Loop

Enter text: Everything happens for a Reason

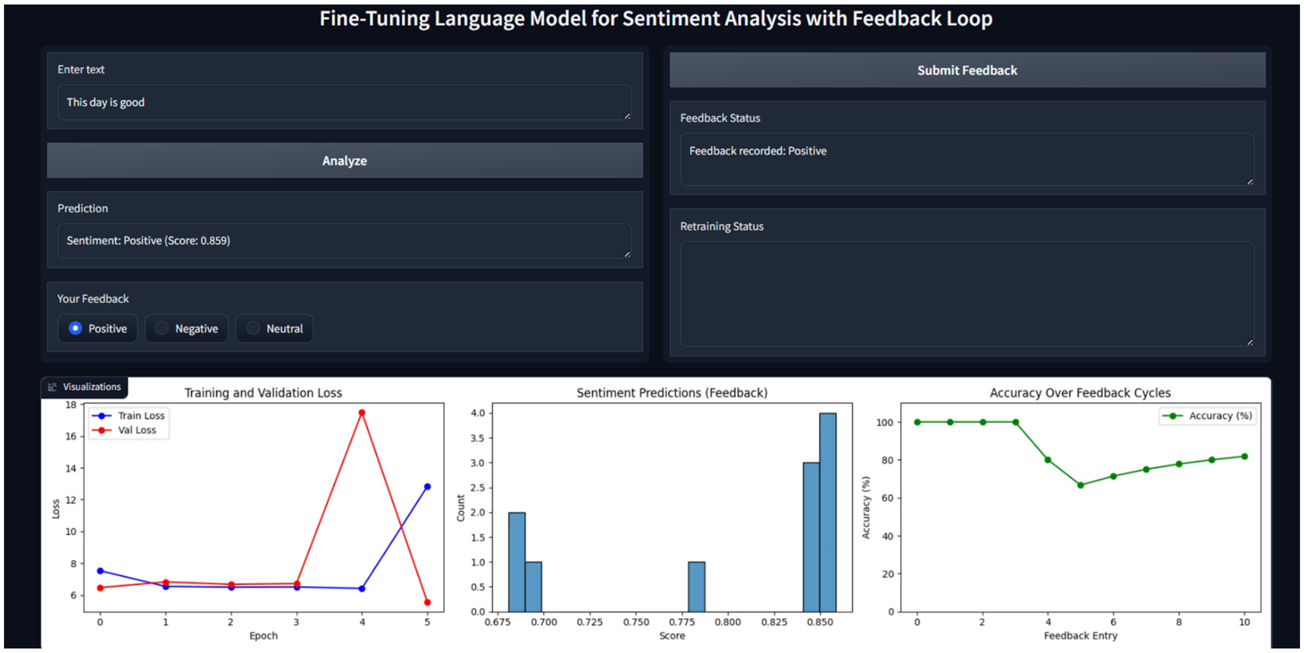
Prediction: Sentiment: Neutral (Score: 0.886)

Your Feedback: Positive Negative [✓ Neutral]

Feedback Status: Feedback recorded: Neutral

Retraining Status:

Fig 9.1 Output in Gradio Webpage 1


Fine-Tuning Language Model for Sentiment Analysis with Feedback Loop

Enter text: This day is good

Prediction: Sentiment: Positive (Score: 0.859)

Your Feedback: [✓ Positive] Negative Neutral

Feedback Status: Feedback recorded: Positive

Retraining Status:

Fig 9.2 Output In Gradio webpage 2

10. Conclusion

The integration of human feedback into the sentiment analysis pipeline significantly enhances the model’s accuracy, adaptability, and alignment with real-world expectations. Each module—from data ingestion and preprocessing to fine-tuning and deployment—plays a crucial role in building a robust and responsive system. The human feedback loop acts as a continuous learning mechanism, enabling the model to correct errors, adapt to evolving language patterns, and handle nuanced sentiments more effectively.By combining traditional machine learning techniques with human-in-the-loop approaches, this architecture ensures that the system remains both intelligent and accountable. Optional components like reward modeling and explainability further deepen the model’s capabilities, allowing for greater transparency and alignment with user intent. Overall, such a modular and feedback-driven framework leads to a dynamic, accurate, and trustworthy sentiment analysis system that can perform reliably in diverse and changing environments.Furthermore, performance evaluation modules ensure that the model is not only learning but also learning *correctly*. With the help of continuous validation, drift detection, and error analysis, the system maintains a high level of robustness even as language use evolves. The optional integration of reward modeling enables preference-based optimization, fine-tuning the model according to human judgment rather than just static labels, which is especially valuable for subjective tasks like sentiment classification.Deployment and monitoring mechanisms ensure the real-world utility of the system, enabling organizations to use the model in real-time applications with confidence. Explainability modules also enhance user trust and regulatory compliance by offering insights into why the model made certain predictions—an essential feature in sensitive domains such as finance, healthcare, and customer support.In conclusion, a sentiment analysis system with a human feedback loop is not just a model—it is a living, evolving ecosystem. It leverages the strengths of both machine intelligence and human insight to deliver superior performance, adaptability, and ethical alignment.

11. Further Enhancements

- Experiment with Different Models like BERT, RoBERTa etc.
- User Experience Enhancements like:
- Multilingual Support by fine-tuning on multilingual datasets (e.g., XLM-RoBERTa).
- Customizable Interface-like Theme or language options.
- API Deployment allows integration with other applications (e.g., mobile apps, websites) and supports scalable serving.
- Cloud Integration for model storage, training, and distributed training
- Emotion Detection Extend sentiment analysis to detect specific emotions (e.g., joy, anger, sadness) using a multi-label classification approach