Analyzing Social Media Posts for Mental Health Disorder Detection

Submitted by

SOUMYADEEP NANDY (13000121033)

PRITHWISH SARKAR (13000121037)

SAGNIK MUKHOPADHYAY (13000121040)

ARKAPRATIM GHOSH (13000121058)

⟨ Group 29 ⟩

Final Year 8th Semester

⟨ June, 2025 ⟩

Submitted for the partial fulfillment for the degree of
Bachelor of Technology in
Computer Science and Engineering

Techno Main Salt Lake,
EM 4/1, Salt lake, Sector - V, Kolkata - 700091

Department of Computer Science and Engineering
Techno Main Salt Lake
Kolkata - 700 091
West Bengal, India

APPROVAL

This is to certify that the project entitled **"Analyzing Social Media Posts for Mental Health Disorder Detection"** prepared by **SOUMYADEEP NANDY** *(13000121033)*, **PRITHWISH SARKAR** *(13000121037)*, **SAGNIK MUKHOPADHYAY** *(13000121040)* and **ARKAPRATIM GHOSH** *(13000121058)* be accepted in partial fulfillment for the degree of Bachelor of Technology in Computer Science and Engineering.

It is to be understood that by this approval, the undersigned does not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn thereof, but approves the report only for the purpose for which it has been submitted.

. . . . . .
(Signature of the Internal Guide)

. . . . . .
(Signature of the HOD)

. . . . . .
(Signature of the External Examiner)

. . . . . .

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

. . . . . .

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our project guide in the department of Computer Science and Engineering. We are extremely thankful for the keen interest our guide took in advising us, for the books, reference materials and support extended to us.

Last but not the least we convey our gratitude to all the teachers for providing us the technical skill that will always remain as our asset and to all non-teaching staffs for the gracious hospitality they offered us.

Place: Techno Main Salt Lake
Date:

SOUMYADEEP NANDY (13000121033)

PRITHWISH SARKAR (13000121037)

SAGNIK MUKHOPADHYAY (13000121040)

ARKAPRATIM GHOSH (13000121058)

Table of Content

List of Figures

Abstract

This project aims to analyze social media posts for early detection of mental health disorders. Specifically, it focuses on using machine learning and deep learning algorithms to classify social media text data based on potential mental health issues. The problem lies in efficiently detecting patterns that indicate mental health conditions within large, unstructured datasets. Using methods like Logistic Regression , XGboost the project seeks to enhance the accuracy of detecting mental health concern with a specific probability. Expected results include a robust classifier capable of distinguishing between different mental health concerns with high accuracy. This project could provide a valuable tool for mental health monitoring on social platforms.

1 Introduction

1.1 Project Overview

Mental health has become a critical global issue, with millions of people affected by various mental disorders, including depression, anxiety, and others. With the increasing use of social media, these platforms have emerged as spaces where people often express their emotions and struggles, sometimes unknowingly revealing signs of mental health challenges. This project focuses on analyzing social media posts to detect mental health disorders using advanced machine learning techniques. By examining language patterns and contextual usage in text data, this project aims to classify posts that potentially indicate mental health issues. Such detection can facilitate early intervention and help direct individuals to appropriate mental health services.

1.2 Project Purpose

The main goal of this project is to leverage machine learning and deep learning to identify the best model and create a web applicatoin capable of identifying signs of mental health disorders from text, images and social media posts by giving username as input. This aligns with a broader goal of using technology to address public health concerns by enabling early detection through data analysis. Specifically, we use classification models such as Logistic Regression and XGboost to predict mental health issues based on text patterns. The project also addresses the technical challenges of processing large datasets and optimizing algorithms for accurate classification.

1.3 Technical Domain Specifications

This project falls within the intersection of natural language processing (NLP) and machine learning (ML), leveraging techniques such as text vectorization, and classification algorithms. Here are the key technical domain specifications:

- **Hardware** : The project does not require specialized hardware beyond a standard machine with adequate processing power. However, for larger datasets or complex model training, a machine equipped with a GPU (Graphics Processing Unit) could significantly reduce processing time. The project can be run on any system with at least 8GB of RAM and a multi-core processor.
- **Operating System** : The project is cross-platform and can be developed and executed on any modern operating system, including:
  - Windows 10/11
  - macOS
  - Linux distributions (Ubuntu, Linux Mint, etc.) A Linux-based system is often preferred in machine learning projects due to its stability and support for tools like TensorFlow, PyTorch, and other libraries used for model training.
- **Software** :
  - **Programming Languages** : Python 3.x will be the primary programming language, given its extensive libraries for machine learning, data analysis, and NLP.
  - **Libraries / Frameworks** :
    - **Scikit-learn** : Used for machine learning algorithms (k-NN, SVM) and model evaluation.
    - **Pandas** : For data manipulation and preprocessing.
    - **NumPy** : To handle large arrays and matrices, which are crucial for efficient numerical computations.
    - **NLTK and spaCy** : For text preprocessing and natural language understanding.
    - **Matplotlib and Seaborn** : For data visualization.
  - **Development Environment** :
    - **Jupyter Notebook** : For interactive development, experimentation, and visualization.
    - **Anaconda** : A distribution that simplifies package management and deployment.

    * **Google Colab** : For cloud-based execution when working with larger datasets or GPU-based model training.

1.4 Business Domain Specifications

From a business perspective, this project holds significant value across various sectors, particularly those that intersect with mental health monitoring, public health awareness, and social media governance. With the increasing prevalence of mental health issues globally, organizations within these industries are searching for innovative solutions to mitigate the growing mental health crisis. Leveraging machine learning for early detection of mental health disorders from social media data can revolutionize how mental health is addressed at both individual and societal levels. Below is a detailed exploration of how this project can impact different business domains:

- **Mental Health Services** : Mental health service providers—such as hospitals, therapy centers, and private practices—can greatly benefit from machine learning models capable of identifying early signs of mental health issues from social media data. In the traditional mental health setting, early detection of disorders like depression or anxiety often relies on self-reporting or clinical assessments, which may come too late in the progression of the disorder. By analyzing patterns in social media posts, these services can adopt a more proactive approach, reaching out to potential patients earlier in their mental health journey.

- **Social Media Platforms** : Social media platforms like Twitter, Facebook, Instagram, and others play an integral role in the public's expression of thoughts and feelings, including mental health struggles. These platforms face increasing pressure to safeguard the well-being of their users. This project's machine learning models can enable these companies to offer valuable services to users while adhering to ethical standards.

- **Public Health Organizations** : Public health organizations are tasked with monitoring and improving the mental well-being of the population on a large scale. For these organizations, access to real-time data from social media can provide a comprehensive view of the mental health landscape, identifying emerging trends and enabling data-driven interventions. Understanding how mental health is being discussed online can help public health organizations create more effective mental health awareness campaigns. Tailored messaging based on the language patterns identified by the model can lead to better engagement with individuals suffering from mental health issues.

1.5 Glossary / Keywords

Term	Definition
Machine Learning (ML)	A subset of artificial intelligence (AI) that enables computers to learn from data and make predictions or decisions without explicit programming.
Natural Language Processing (NLP)	A branch of artificial intelligence focused on the interaction between computers and humans through natural language, including tasks like text analysis.
Support Vector Machines (SVM)	A supervised learning algorithm used for classification or regression tasks, focusing on finding a hyperplane that best separates different classes.
Vectorization	The process of converting textual data into numerical form (such as a vector) so that it can be used as input for machine learning models.
Classifier	A machine learning model or algorithm that categorizes or labels data points into predefined classes.
Mental Health Disorder	A wide range of conditions that affect mood, thinking, and behavior, including depression, anxiety, schizophrenia, etc.
Data Preprocessing	The process of preparing raw data for analysis by cleaning, normalizing, and transforming it into a usable format for machine learning models.
Cross-validation	A model validation technique used to assess how well a model performs by dividing data into training and testing sets multiple times for better accuracy.
Precision	In the context of classification, precision refers to the accuracy of positive predictions, calculated as the ratio of true positives to the sum of true and false positives.
Recall	In classification, recall measures the ability of a model to identify all relevant instances within a dataset, calculated as the ratio of true positives to the sum of true positives and false negatives.
PRAW	PRAW (Python Reddit API Wrapper) is a Python library that provides a simple interface to interact with Reddit's API, allowing developers to easily access, retrieve, and analyze Reddit data, such as posts, comments, and user information.
TesseractOCR	TesseractOCR is an open-source Optical Character Recognition (OCR) engine that extracts text from images with high accuracy; it is widely used for various applications like scanning documents and digitalizing printed text.

Term	Definition
Depression	There is a difference between depression and mood swings or short-lived emotional reactions to daily experiments; A mental state causing painful symptoms adversely disrupts normal activities (e.g., sleeping).
Anxiety	Several behavioral disturbances are associated with anxiety disorders, including excessive fear and worry. Severe symptoms cause significant impairment in functioning cause considerable distress. Anxiety disorders come in many forms, such as social anxiety, generalized anxiety, panic, etc.
Bipolar Disorder	An alternating pattern of depression and manic symptoms is associated with bipolar disorder. An individual experiencing a depressive episode may feel sad, irritable, empty, or lose interest in daily activities. Emotions of euphoria or irritability, excessive energy, and increased talkativeness can all be signs of manic depression. Increased self-esteem, decreased sleep need, disorientation, and reckless behavior may also be signs of manic depression.
Post-Traumatic Stress Disorder (PTSD)	In PTSD, persistent mental and emotional stress can occur after an injury or severe psychological shock, characterized by sleep disturbances, constant vivid memories, and dulled response to others and the outside world. People who re-experience symptoms may have difficulties with their everyday routines and experience significant impairment in their performance.
DeepFace	DeepFace is a Python library for deep learning-based facial recognition and attribute analysis. It supports several pre-trained models and simplifies face recognition tasks, making it suitable for various applications in image analysis.
Transformers Module	The Transformers module in Python, developed by Hugging Face, is a library for natural language processing (NLP) tasks like text classification, translation, and summarization, using state-of-the-art models like BERT and GPT.
Gemini 1.5 Flash	Gemini 1.5 Flash is a cutting-edge AI model developed by Google, capable of performing advanced generative and analytical tasks across text, image, and other modalities.
FFmpeg	FFmpeg is a multimedia framework used for encoding, decoding, transcoding, streaming, and manipulating audio and video files, supporting a wide range of formats and codecs.
Hyperparameter Tuning	Hyperparameter tuning involves selecting the best parameters for a machine learning model to optimize its performance on a given task, using methods like grid search or random search.
Logistic Regression	Logistic Regression is a statistical method used for binary and multiclass classification tasks by modeling the relationship between input variables and probabilities using a sigmoid function.

Term	Definition
K-Nearest Neighbors (KNN)	KNN is a simple, instance-based machine learning algorithm used for classification and regression tasks, predicting outcomes based on the majority class of its k nearest neighbors.
Naive Bayes	Naive Bayes is a probabilistic classification algorithm based on Bayes' Theorem, assuming independence between features, making it efficient for text classification and other applications.
Random Forest	Random Forest is an ensemble machine learning algorithm that combines multiple decision trees to improve accuracy and prevent overfitting, commonly used for classification and regression tasks.
XGBoost	XGBoost (Extreme Gradient Boosting) is a powerful ensemble machine learning algorithm based on decision trees, designed for speed and performance in classification and regression tasks.
Long Short-Term Memory (LSTM)	LSTM is a type of recurrent neural network (RNN) architecture capable of learning and remembering patterns over long sequences of data, widely used in time-series analysis and NLP.

2 Related Studies

The intersection of social media analytics and mental health research has received increasing attention in recent years, leading to several important studies that highlight the potential for early detection and intervention. This section reviews key findings from various studies, emphasizing the relevance and applicability of social media data for identifying mental health disorders.

One of the seminal works in this domain is by Choudhury et al. (2013), who explored the predictive capabilities of social media content in identifying depression. They analyzed Twitter data and discovered that specific linguistic patterns, such as the use of negative emotion words, correlated strongly with self-reported depressive symptoms. This study demonstrated that social media could serve as a valuable resource for predicting mental health conditions, offering a potential tool for clinicians and researchers alike [2].

Similarly, Guntuku et al. (2017) conducted an integrative review that focused on detecting mental illness through social media. Their work synthesized various approaches and methodologies used in the field, providing insights into the effectiveness of different machine learning algorithms and sentiment analysis techniques. They found that social media platforms are rich sources of data that can reveal critical information about users' mental health, advocating for

the development of robust systems to analyze this data effectively [4].

A systematic review by Mathur et al. (2022) further emphasized the significance of mental health classification on social media. They examined various studies that utilized machine learning techniques for mental health detection, highlighting the success of these models in identifying depression and anxiety based on user-generated content. Their findings reinforced the notion that social media can be leveraged not only for individual assessments but also for broader epidemiological studies to understand population mental health trends [5].

In addition, Nadeem (2016) contributed to the discussion by investigating depression identification on Twitter. The study focused on developing algorithms that could discern emotional cues in tweets, indicating a user's mental state. The findings revealed that simple text analysis could lead to significant improvements in identifying at-risk individuals, further validating the potential of social media data in mental health monitoring [6].

Research by AlSagri and Ykhlef (2020) introduced a machine learning-based approach specifically for depression detection on Twitter. Their study incorporated both content and activity features, demonstrating that a combination of linguistic and behavioral analysis could enhance the accuracy of depression identification. This work illustrated the multifaceted nature of social media data and its ability to capture not just what users say but also how they interact online [1].

In a more recent study, Vaishnavi et al. (2022) investigated the application of various machine learning algorithms for predicting mental health illnesses. They found that certain algorithms outperformed others in classifying mental health conditions based on social media posts. This study provided a comparative analysis that could inform future research directions, emphasizing the importance of algorithm selection in the context of mental health detection [9].

Lastly, Safa et al. (2023) presented a roadmap for future development in predicting mental health using social media. Their work highlighted the ongoing challenges in the field, including ethical considerations and the need for improved data privacy measures. They emphasized that while social media offers rich data for mental health analysis, researchers must approach this opportunity with a strong ethical framework to ensure user safety and data security [7].

The paper titled "Single classifier vs. ensemble machine learning approaches for mental health prediction" (PMC9810771) explores the use of various machine learning techniques to predict mental health issues. Specifically, the study compares the performance of single classifiers (such as Logistic Regression, Gradient Boosting, Neural Networks, K-Nearest Neighbors, and

Support Vector Machine) with ensemble machine learning approaches, which combine multiple classifiers to improve prediction accuracy. The study is based on a dataset of survey responses collected by Open Sourcing Mental Illness (OSMI), and the goal is to classify mental health issues based on these responses. The authors also compare newer machine learning techniques like Extreme Gradient Boosting (XGBoost) and Deep Neural Networks (DNN). The results indicate that Gradient Boosting achieved the highest accuracy (88.80%), followed by Neural Networks (88.00%). The ensemble classifier, which combined multiple models, achieved an accuracy of 85.60%. Overall, the paper demonstrates that machine learning techniques, particularly ensemble models, show promise for automated prediction of mental health problems, providing valuable insights for early diagnosis and intervention in mental health care [3].

The paper titled "Ensemble of hybrid model based technique for early detecting of depression based on SVM and neural networks" (DOI: 10.1038/s41598-024-77193-0) presents a novel approach for early detection of depression using machine learning techniques. The study proposes an ensemble hybrid model combining Support Vector Machines (SVM) and Multilayer Perceptrons (MLP) to improve depression prediction accuracy. The DeprMVM hybrid model serves as a meta-learner, where the SVM and MLP networks act as level-0 learners. The model addresses class imbalance by applying the Synthetic Minority Over-sampling Technique (SMOTE) and cluster sampling, which improves detection accuracy and reduces the risk of overfitting. The study finds that the ensemble approach achieved an accuracy of 99.39% and an F1-score of 99.51%, outperforming previous models in depression detection. This paper highlights the potential of ensemble hybrid models for early detection of depression and their applicability in mental health care [8].

The paper titled "Survey of transformers and towards ensemble learning using transformers for natural language processing" (DOI: 10.1186/s40537-023-00842-0) provides a comprehensive review of transformer models in natural language processing (NLP). The study compares several prominent transformer-based models, including BERT, XLNet, RoBERTa, GPT-2, and ALBERT, evaluating their performance across multiple NLP tasks such as sentiment analysis, question answering, and text generation. The authors also introduce ensemble learning approaches using these models to enhance task performance. The results demonstrate that ensemble models outperform single classifier approaches, offering significant improvements for specific NLP tasks. This paper highlights the potential of ensemble learning with transformer models and their versatility in solving complex NLP challenges [10].

These studies collectively underscore the growing body of evidence supporting the integration of social media analytics and machine learning for mental health detection.

3 Problem Definition and Preliminaries

3.1 Context and Background

Mental health disorders have become a significant public health concern worldwide. The World Health Organization (WHO) estimates that approximately 1 in 8 people globally experience mental health disorders, which encompass conditions such as depression, anxiety, bipolar disorder, and post-traumatic stress disorder (PTSD). The rise of social media platforms has changed how individuals express their mental health struggles, share experiences, and seek support. Posts on platforms like Reddit and Twitter provide a wealth of data reflecting real-time sentiments, issues, and conversations surrounding mental health. However, this vast amount of unstructured textual data presents challenges in effectively identifying and categorizing specific mental health disorders.

3.2 Objective

The primary objective of this project is to develop a robust system that can automatically analyze social media posts, specifically from Reddit and Twitter, to detect various mental health disorders. By leveraging Natural Language Processing (NLP) techniques and machine learning algorithms, this project aims to:

- **Classify Posts** : Accurately classify social media posts based on the type of mental health disorder mentioned, including but not limited to depression, anxiety, bipolar disorder, and PTSD.
- **Data Driven Insights** : Provide valuable insights into the prevalence and expression of mental health issues on social media, helping researchers, mental health professionals, and policymakers understand trends and patterns.

3.3 Challenges

- **Data Variability** : Social media posts can vary significantly in structure, style, and length. Users may employ slang, abbreviations, and informal language, making it difficult for algorithms to accurately interpret and classify posts.
- **Imbalanced Data** : Certain mental health issues may be underrepresented in social media discussions, leading to an imbalanced dataset. This imbalance can adversely affect model training and performance, making it harder to detect less frequent disorders.
- **Cultural and Contextual Nuances** : Mental health perceptions and discussions can vary across different cultures and contexts. The model needs to account for these nuances to avoid misclassification and provide accurate insights.

- **Privacy and Ethical Considerations** : Analyzing social media data raises ethical concerns regarding user privacy. It is crucial to handle sensitive information responsibly and comply with data protection regulations.

3.4 Scope

The scope of the project ”Analyzing Social Media Posts for Mental Health Disorder Detection” delineates the specific aspects that will be covered, the methodologies employed, and the boundaries within which the research and analysis will occur. The project aims to harness the potential of machine learning and natural language processing (NLP) techniques to analyze social media sentiment and its correlation with mental health disorders, focusing specifically on a Reddit Dataset sourced using Python Reddit API Wrapper. This dataset contains user-generated content that reflects various emotional states, making it a valuable resource for this analysis.

- **Dataset Selection and Characteristics**
  The primary data source for this project is the top textual posts from Reddit. This dataset includes posts that are labeled with mental health issues (Normal, Anxiety, Depression, Bipolar, PTSD). The selection of this dataset is pivotal, as it encapsulates a wide range of mental health-related discussions expressed by individuals on social media. Key characteristics of the dataset include:
  - **User Anonymity** : To respect user privacy and adhere to ethical standards, the dataset does not contain personally identifiable information (PII) about the Twitter users. This ensures compliance with data protection regulations while allowing for robust analysis.
- **Analysis Objectives**
  The project will focus on several key objectives:
  - **Correlation with Mental Health Issues** : The analysis will explore the correlation between identified sentiments and specific mental health disorders, thereby contributing to the understanding of how social media discourse reflects mental health challenges.
  - **Trend Analysis** : By analyzing sentiment trends over time, the project seeks to identify patterns in public discourse surrounding mental health, including any potential spikes in negative sentiments during particular events or crises.
- **Methodologies** : The project will employ various methodologies to achieve its objectives, including:

- **Data Processing** : Cleaning and preparing the dataset to ensure that it is suitable for analysis. This includes tasks such as removing noise (e.g., URLs, hashtags), tokenization, and normalization of text.
- **Feature Extraction** : Utilizing techniques like Term Frequency-Inverse Document Frequency (TF-IDF) to convert textual data into numerical representations suitable for machine learning algorithms.
- **Machine Learning Techniques** : Implementing various machine learning algorithms, including Logistic Regression and XGboost to classify posts and evaluate their performance based on accuracy, precision, recall, and F1 score.
- **Data Visualizations** : Employing visualization tools to present findings clearly, including heat maps for confusion matrices and ROC AUC Curve.

3.5 Exclusions

In delineating the boundaries of the project "Analyzing Social Media Posts for Mental Health Disorder Detection," it is crucial to specify what is excluded from the scope of this research to maintain a clear focus on the primary objectives. This project will not encompass the direct collection or real-time monitoring of Twitter data via the Twitter API, as it is solely reliant on the pre-existing Twitter sentiment dataset obtained from Kaggle. Therefore, any analysis involving the dynamic aspects of social media engagement, such as real-time sentiment shifts in response to current events or trending topics, will be outside the project's purview. Furthermore, the study will not address the technicalities of Reddit's platform-specific features, such as hashtags, user mentions, or reposts, subreddits in detail, as these elements are not central to the primary research focus on mental issue classification of individual users. While the project aims to analyze posts surrounding mental health, it will also include mapping the mental issue with mental wellbeing. Additionally, the research will not explore the ethical implications of data ownership or the responsibilities of social media platforms regarding user-generated content, as the focus will be primarily on data analysis techniques and outcomes rather than the broader ethical landscape.

3.6 Assumptions

In the context of the project "Analyzing Social Media Posts for Mental Health Disorder Detection," several key assumptions underpin the research framework and methodologies employed. Firstly, it is assumed that the Reddit dataset obtained using PRAW is representative of broader social media discourse regarding mental health issues, capturing a diverse range of sentiments expressed by users on the platform. This assumption is critical as it establishes the foundation

for analyzing sentiment trends and their potential correlations with various mental health disorders. Secondly, it is presumed that the posts recorded in the dataset accurately reflect the users' true emotions and perspectives at the time of posting, thereby providing valid data for analysis. Furthermore, it is assumed that the textual data within the dataset can be effectively processed and interpreted through natural language processing (NLP) techniques, allowing for the accurate classification of sentiments and identification of patterns. Another assumption is that the selected machine learning algorithms, including Logistic Regression and XGboost, will perform optimally with the provided data, leading to reliable and interpretable results regarding sentiment analysis and mental health correlations. Additionally, it is assumed that the sentiments expressed in social media posts can serve as a valid proxy for understanding public perceptions of mental health, enabling insights into societal attitudes and the potential stigmatization associated with these disorders. The project also assumes that the cleaning and preprocessing steps applied to the data will sufficiently prepare the dataset for analysis, minimizing noise and irrelevant information that could skew the results. Lastly, it is presumed that the ethical considerations surrounding the use of publicly available social media data have been adequately addressed, ensuring that the research adheres to relevant ethical standards and does not compromise user privacy or data integrity. These assumptions serve as the bedrock for the project's analytical framework, guiding the research processes and interpretations that follow.

4 Proposed Solution

The proposed work centers around "Analyzing Social Media Posts for Mental Health Disorder Detection," leveraging advanced data analytics and machine learning techniques to provide insights into the sentiments expressed in social media discussions related to mental health issues. This research is significant due to the increasing prevalence of mental health disorders globally and the role social media plays in shaping public perception and discourse around these issues. The core objective of this project is to develop a systematic approach to classify sentiments in tweets, thereby enabling better understanding and awareness of mental health conditions through social media analysis. Below, I outline the specific contributions and reusable components deployed in this project.

4.1 Special Contributions

- **Dataset Acquisition and Preparation** : The initial step involved sourcing a high-quality dataset from Kaggle, specifically the Twitter sentiment dataset. This dataset comprises user-generated tweets containing sentiments related to various mental health issues, serving as the primary data source for analysis. The preparation phase included extensive data

cleaning and preprocessing, wherein missing values were handled, duplicate entries removed, and text normalization performed. This crucial step ensured the dataset's integrity and suitability for subsequent analysis, allowing for a more accurate representation of sentiments.

- **Text Vectorization** : To enable machine learning models to interpret textual data, TF-IDF was implemented. This approach involved converting posts into a numerical format by creating a matrix representation of word frequencies across the dataset. The Scikit-learn library was instrumental in this process, offering functions for text vectorization and feature extraction. The reusable components for text preprocessing and vectorization were packaged into functions, allowing for easy application to future datasets or similar projects.

- **Implementation of Machine Learning Algorithms** : The project employed several machine learning algorithms, focusing primarily on Logistic Regression and XGboost for smental health classification. The Logistic Regression algorithm was chosen for its simplicity and effectiveness in classifying data points based on proximity in the feature space. In addition to Logistic Regression, XGboost was implemented due to its robustness in handling high-dimensional data and multi class classification tasks.

- **Model Evaluation** : Comprehensive evaluation metrics were employed to assess the performance of the machine learning models. Metrics such as accuracy, precision, recall, and F1-score were calculated to provide a holistic view of model effectiveness in classifying sentiments related to mental health. This evaluation process not only demonstrated the models' capabilities but also highlighted areas for improvement, providing a foundation for future iterations of the project. The evaluation framework, including metrics calculations and visualization, was designed as reusable components to streamline future model assessments.

- **Insights and Recommendations** : A critical aspect of this project is the generation of actionable insights based on the analysis of social media sentiments. The findings from the classification can inform mental health professionals, researchers, and policymakers about public sentiment trends, potential stigma associated with mental health issues, and the effectiveness of awareness campaigns. Recommendations for mental health awareness strategies can be derived from understanding how sentiments vary across different demographics and regions. This interpretative analysis, combined with quantitative results, contributes valuable knowledge to the ongoing conversation about mental health in society.

- **Documentation and Reproducibility** : To enhance the usability and impact of the project, thorough documentation was maintained throughout the research process. This documentation includes detailed explanations of the methodologies employed, code snippets, and instructions for reproducing the results. The aim is to ensure that the components developed in this project can be easily utilized by other researchers and practitioners in the field. By documenting the code and methodologies, I am contributing to the open-source community, allowing for collaborative improvements and innovations in mental health sentiment analysis.

4.2 Reusable Components

- **Data Collection Functions** : Modular functions designed for data collection, which can be reused across different platforms.
- **Data preprocessing Module** : A component that does data cleaning to remove the duplicates and empty rows and add a seperate column for cleaned texts. This formatted dataset is then used further with TF-IDF to create a numerical matrix to be fed to the machine leanring algorithms.
- **Machine and Deep Learning Model Functions** : Functions for implementing Logistic Regression, Naive Bayes, Support Vector Machine, Random Forest, XGboost, Long Short Term Memory algorithm allowing for easy retraining on varying datasets. These also features various evaluation metrics, making it easy to assess different models' performances.
- **Deployment Function** : A seperate function that has the main python file for creating web based application on Streamlit Cloud. This also includes the requirements and package dependencies for deploying the application.

5 Project Planning

5.1 Software Life Cycle Model

In developing the project, I adopted an iterative approach to the Waterfall model, enabling a structured yet flexible framework for managing the various phases of the project. The project plan outlines specific tasks, dependencies, timelines, and milestones to ensure a systematic progression towards the final goal of analyzing social media posts for mental health disorder detection.

The Iterative Waterfall model was chosen for this project due to its structured approach while allowing for iterative revisions and refinements. Unlike the traditional Waterfall model, which emphasizes a linear progression through distinct phases, the iterative variant permits revisiting earlier stages based on findings and feedback. This flexibility is particularly beneficial in data-driven projects where insights gained during the analysis may necessitate adjustments to earlier stages, such as refining requirements or enhancing data preparation techniques.

In this project, the iterative nature of the Waterfall model facilitated ongoing improvement and adaptation throughout the development process. For example, initial results from the model evaluation phase may prompt a revisit to data preprocessing to enhance data quality or to explore alternative modeling techniques. This approach ultimately fosters a more robust final product, ensuring that the developed system meets the dynamic needs of mental health disorder detection in social media posts.

The key feature of the iterative approach is the feedback loop that exists between the phases. For instance, after completing the testing phase, if certain models do not meet performance expectations, the project can loop back to the implementation phase. This allows for modifications to the models, preprocessing techniques, or even revisiting the requirements to ensure alignment with the project's objectives.

The project is divided into distinct phases:

- **Requirement Gathering and Analysis** : This initial phase involved understanding the project's goals, objectives, and stakeholder expectations. It spanned approximately two weeks, culminating in a detailed requirements document that guided the subsequent stages.
- **Data Collection and Preparation** : Utilizing a Twitter sentiment dataset from Kaggle and Reddit API, the data collection phase was executed over a week. This included downloading the dataset, examining its structure, and performing data cleaning and prepossessing to ensure its suitability for analysis.
- **Model Development** : This phase, lasting about three weeks, included the creation of a Bag of Words model, splitting the dataset into training and test sets, and implementing various machine learning algorithms such as k-Nearest Neighbors (k-NN) and Support Vector Machines (SVM) for sentiment classification.
- **Model Evaluation** : Following model development, a week was allocated for rigorous testing and validation of the models, ensuring they met the required accuracy benchmarks. This phase involved using performance metrics such as accuracy, precision, recall, and F1-score to evaluate the models' effectiveness.

- **Final Deployment and Documentation** : The last phase, spanning two weeks, focused on deploying the best-performing model and creating comprehensive documentation. This included user manuals and technical documentation to facilitate future maintenance and enhancements.

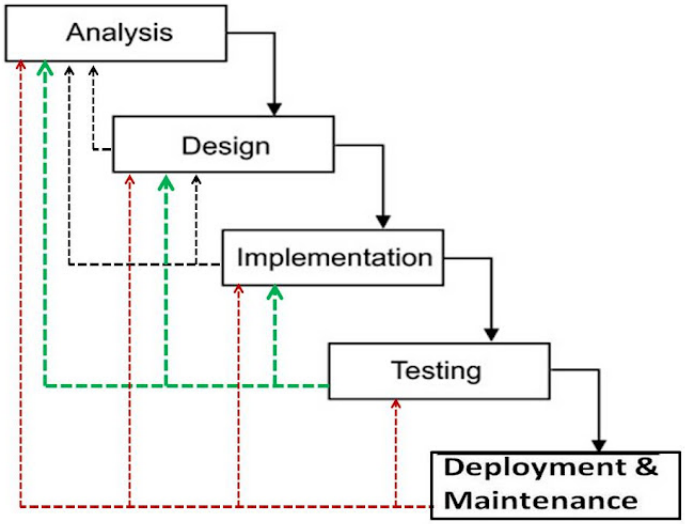


Figure 1: Iterative Waterfall Model

5.2 Dependencies and Milestones

Key dependencies were identified for successful project progression. For instance, completion of the data preparation phase was critical before proceeding to model development. Milestones were established at the end of each phase to ensure accountability and track progress. The successful completion of the requirement gathering phase marked the first milestone, followed by the data preparation phase, and so on.

5.3 Scheduling

Effective scheduling is crucial to the success of any project, as it establishes a clear timeline for tasks, milestones, and dependencies. In the context of our project on detecting mental health disorders through social media analysis, a detailed schedule has been developed to guide the project from inception to completion. This schedule includes specific tasks such as requirement gathering, data preprocessing, model implementation, testing, and deployment, each with clearly defined deadlines. The iterative nature of our chosen methodology allows for flexibility within the schedule, enabling adjustments based on testing outcomes and stakeholder feedback. Key milestones, such as the completion of data analysis, model validation, and user acceptance testing, have been identified to monitor progress and ensure timely delivery of the final product.

By utilizing project management tools, such as Microsoft Project, we can visualize and track the progress of tasks, manage resources effectively, and maintain open communication among team members, ensuring that the project stays on schedule and meets its objectives. This proactive approach to scheduling enhances our ability to deliver a high-quality solution that aligns with our goals and stakeholder expectations.

ID	Task Mode	Task Name	Duration	Start	Finish	Predecessors	% Complete
1		**GR29 ASMPFMHDD**	**243 days**	**Mon 01-07-24**	**Mon 02-06-25**		**100%**
2		**Phase 1: 7th Semester Activities**	**150 days**	**Mon 01-07-24**	**Wed 22-01-25**		**100%**
3		**Project Startup**	**20 days**	**Mon 01-07-24**	**Fri 26-07-24**		**100%**
4		Team Building	2 days	Mon 01-07-24	Tue 02-07-24		100%
5		Brainstorm on Project Topic	2 days	Wed 03-07-24	Thu 04-07-24	4	100%
6		Project agreed with Guide	2 days	Fri 05-07-24	Mon 08-07-24	5	100%
7		Related Study& Documentation	4 days	Mon 08-07-24	Thu 11-07-24	6	100%
8		Deliver Project Synopsis for Guide's review	2 days	Fri 12-07-24	Mon 15-07-24	7	100%
9		Close review feedbacks	9 days	Mon 15-07-24	Thu 25-07-24	8	100%
10		**Project Synopsis Finalized**	**1 day**	**Fri 26-07-24**	**Fri 26-07-24**	**9**	**100%**
11		**Requirement Analysis**	**17 days**	**Thu 01-08-24**	**Fri 23-08-24**		**100%**
12		Gather Requirements	7 days	Thu 01-08-24	Fri 09-08-24	10	100%
13		Prepare Draft Requirement Matrix	9 days	Mon 12-08-24	Thu 22-08-24	10	100%
14		**Requirement Matrix Finalized**	**1 day**	**Fri 23-08-24**	**Fri 23-08-24**	**13**	**100%**
15		**Design**	**46 days**	**Mon 26-08-24**	**Thu 24-10-24**	**14**	**100%**
16		**Detailed Design**	**25 days**	**Mon 26-08-24**	**Thu 26-09-24**	**14**	**100%**
17		Data Collection	3 days	Mon 26-08-24	Wed 28-08-24	14	100%
18		Data Preprocessing	4 days	Thu 29-08-24	Mon 02-09-24	17	100%
19		Model Training and Evaluation	18 days	Tue 03-09-24	Thu 26-09-24	18	100%
20		**Test Plan Preparation**	**21 days**	**Fri 27-09-24**	**Thu 24-10-24**	**19**	**100%**
21		Text Classification	4 days	Fri 27-09-24	Wed 02-10-24	19	100%
22		Image Classification	9 days	Thu 03-10-24	Tue 15-10-24	21	100%
23		Video Classification	4 days	Wed 16-10-24	Sat 19-10-24	22	100%
24		Reddit and Twitter User Analysis	4 days	Mon 21-10-24	Thu 24-10-24	23	100%
25		**Phase 1 Closure**	**59 days**	**Fri 01-11-24**	**Wed 22-01-25**	**3**	**100%**
26		Prepare 7th Semester Project Rep	14 days	Fri 01-11-24	Wed 20-11-24	20	100%
27		**Updated Requirement Matrix**	**2 days**	**Thu 21-11-24**	**Fri 22-11-24**	**26**	**100%**
28		**Updated Project Plan**	**1 day**	**Fri 22-11-24**	**Fri 22-11-24**	**27**	**100%**
29		Project Viva	2 days	Mon 20-01-25	Tue 21-01-25	28	100%
30		**Approved Project Report - 7th Semester**	**1 day**	**Wed 22-01-25**	**Wed 22-01-25**	**29**	**100%**
31		**Semester Gap**	**9 days**	**Thu 23-01-25**	**Tue 04-02-25**	**30**	**100%**
32		**Phase 2: 8th Semester Activities**	**84 days**	**Wed 05-02-25**	**Mon 02-06-25**	**31**	**100%**
33		**Coding & Únit Testing**	**27 days**	**Wed 05-02-25**	**Thu 13-03-25**	**31**	**100%**
34		Data Collection and Preprocessing	7 days	Wed 05-02-25	Thu 13-02-25	31	100%
35		Model Training and Evaluation	6 days	Fri 14-02-25	Fri 21-02-25	34	100%
36		Web Application Components	14 days	Mon 24-02-25	Thu 13-03-25	35	100%
37		**System Integration Testing**	**45 days**	**Fri 14-03-25**	**Thu 15-05-25**	**36**	**100%**
38		API Calls	23 days	Fri 14-03-25	Tue 15-04-25	36	100%
39		Deployment	22 days	Wed 16-04-25	Thu 15-05-25	38	100%
40		**Project Closure**	**14 days**	**Wed 14-05-25**	**Mon 02-06-25**	**39**	**100%**
41		Prepare 8th Semester Project Report	5 days	Wed 14-05-25	Tue 20-05-25	39	100%
42		**Updated Requirement Matrix**	2 days	Wed 21-05-25	Thu 22-05-25	41	100%
43		**Updated Project Plan**	5 days	Fri 23-05-25	Thu 29-05-25	42	100%
44		Review by Faculties	1 day	Fri 30-05-25	Fri 30-05-25	43	100%
45		**Approved Project Report - 7th Semester**	1 day	Mon 02-06-25	Mon 02-06-25	44	100%

Figure 2: Project Plan

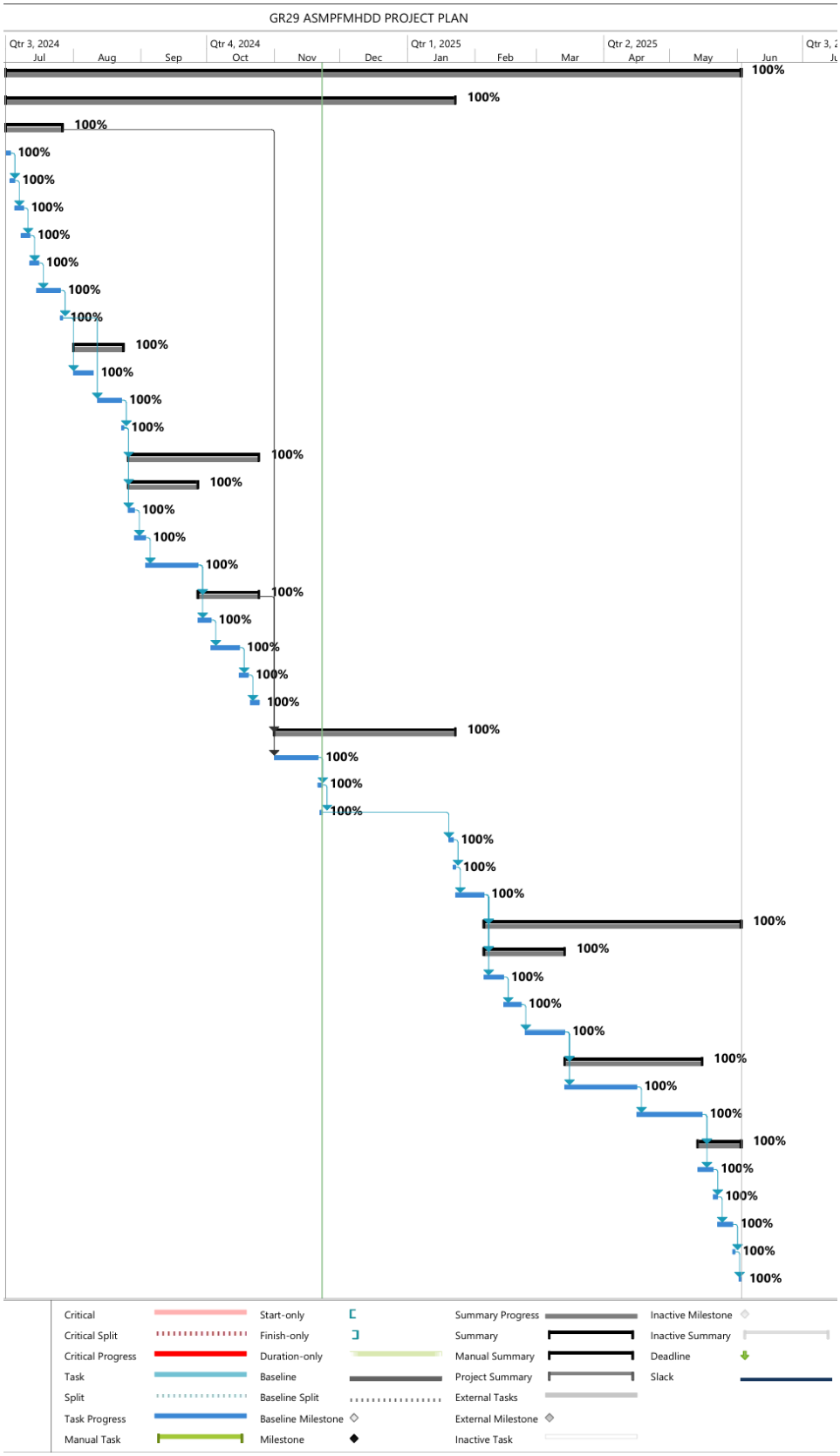


Figure 3: Gantt Chart

6 Requirement Analysis

6.1 Requirement Matrix

Rqmt ID	Requirement Item	Requirement Analysis Status	Design Module (As per Prototype folder structure)	Design Reference (section# under project Report)	Test Case Number (As per Prototype folder structure)	Technical Platform of Implementation	Prototype prepared ?	Name of Program / Component	Own code or Reusable component (with source reference)?	Test Results Reference
FR-001	Collect social media data from Reddit.	Completed	D01	8.2.1	D04T04	PYTHON, COLAB	Yes	Data Collection.ipynb	Own code	Data Collection.ipynb
FR-002	Implement data cleaning and preprocessing.	Completed	D02	8.2.2	D04T04	PYTHON, COLAB	Yes	Data Preprocessing.ipynb	Own code	Data Preprocessing.ipynb
FR-003	Train machine learning and deep learning models.	Completed	D03	8.2.3 - 8.2.10	D05DP01	PYTHON, COLAB	Yes	WebAppV7.ipynb	Own code	WebAppV7.ipynb
FR-004	Evaluate models using performance metrics (accuracy, recall, F1 Score, Support).	Completed	D03	9.2 - 9.9	D04T02, D04T03	PYTHON, COLAB	Yes	[Algorithm_name.ipynb]	Own code	[Algorithm_name].ipynb
NFR-001	Testing different features of the web application	Completed	D04	APPENDIX A - PROTOTYPE	D05DP02	PYTHON, STREAMLIT	Yes	WebAppV7.ipynb, WebAppV8.ipynb WebAppV9.ipynb, WebAppV10.ipynb	Own code	WebAppV7.ipynb, WebAppV8.ipynb WebAppV9.ipynb, WebAppV10.ipynb
NFR-002	Final Web Application Deployment	Completed	D05	APPENDIX A - PROTOTYPE	D05DP03	PYTHON, STREAMLIT	Yes	WebAppV11.ipynb	Own code	WebAppV11.ipynb

Figure 4: Requirement Matrix

The Requirement Matrix is a comprehensive tool used to track and manage the key requirements of a project. It systematically organizes each requirement with a unique identifier, description, priority, and category (such as functional or non-functional). The matrix also records the source of the requirement, its current status (e.g., in progress, completed), any dependencies on other requirements, and the team or individual responsible for fulfilling it. Additionally, it includes a verification method to ensure the requirement is met, such as through testing or review. This structured format helps ensure that all requirements are clearly defined, prioritized, and tracked, enabling effective project management and ensuring alignment with stakeholder expectations.

6.2 Requirement Elaboration

6.2.1 Functional Requirements

Requirement ID: FR-001
Description: Data Collection
Priority: High
Category: Functional
The system requires an ability to collect and ingest a large dataset of Reddit posts using Python Reddit API Wrapper. The data should include text content from tweets, associated sentiment labels, and other metadata such as timestamp and user details. This will serve as the primary source of information for mental health disorder detection. The system must ensure that the dataset is loaded correctly into the machine learning environment, and any discrepancies in the structure should be handled with pre-processing steps like cleaning, normalization.

Requirement ID: FR-002
Description: Data Cleaning and Preprocessing
Priority: High
Category: Functional
The system must include modules to clean the raw data, such as removing irrelevant characters, handling missing data, and tokenizing text. For the social media posts, it is essential to remove URLs, stopwords, and unnecessary punctuation. The pre-processing pipeline should also convert the text into a suitable numerical format using Term Frequency-Inverse Document Frequency (TF-IDF) for further analysis. Proper pre-processing ensures that the data is in a form that can be efficiently used by machine learning models.

Requirement ID: FR-003
Description: Training Machine and Deep Learning Models
Priority: High
Category: Functional
The system needs to implement machine learning algorithms such as Logistic Regression and XGboost to classify social media posts based on text and detect potential signs of mental health disorders. The system must be able to train these models on historical data and then apply them to predict the sentiment and detect mental health-related issues in new posts.

Requirement ID: FR-004
Description: Model Validation and Evaluation
Priority: High
Category: Functional
The system must evaluate the performance of the trained models by splitting the dataset into training and test sets. Various performance metrics like accuracy, precision, recall, and F1-score should be computed to assess the model's effectiveness in detecting mental health disorders. Based on the evaluation, the system should allow for model fine-tuning, such as adjusting hyperparameters, to improve the overall performance.

6.2.2 Non Functional Requirements

Requirement ID: NFR-001
Description: Testing
Priority: Medium
Category: Non Functional
The addition of testing as a non-functional requirement ensures that the system is not only usable in the short term but also maintainable and extensible over time. This guarantees that future

updates and improvements to the system can be implemented without disrupting existing functionality or requiring a steep learning curve for new developers or users.

Requirement ID: NFR-002
Description: Deployment
Priority: Medium
Category: Non Functional
The system must be deployed considering the potential growth in the amount of social media posts that need to be analyzed. The data processing pipeline should be scalable to handle increasing data size without significant degradation in performance. This could involve implementing parallel processing techniques or leveraging cloud-based infrastructure to ensure that processing large datasets remains feasible even as the dataset scales.

The system is designed to support comprehensive mental health disorder detection from social media posts, with key functional and non-functional requirements. It requires the ability to collect and ingest large datasets of Reddit posts using the Python Reddit API Wrapper, capturing text content, sentiment labels, timestamps, and user metadata. This data forms the foundation for analysis, necessitating robust pre-processing steps to ensure quality and consistency. The pre-processing pipeline must clean raw data by removing irrelevant characters, URLs, and stopwords, handling missing values, and tokenizing text. Additionally, text should be converted into numerical formats using techniques like Term Frequency-Inverse Document Frequency (TF-IDF) for model training. The system incorporates machine learning algorithms such as Logistic Regression and XGBoost to classify social media posts and detect potential mental health disorders. It must support training these models on historical data while applying them to predict and classify new posts. To ensure model reliability, the system splits data into training and test sets, evaluating performance using metrics like accuracy, precision, recall, and F1-score. Model fine-tuning, including hyperparameter adjustments, is essential for optimizing performance. Non-functional requirements focus on testing to ensure the system remains maintainable and extensible, enabling future updates without significant disruption. Additionally, deployment considerations emphasize scalability to process increasing volumes of social media posts, leveraging parallel processing and cloud infrastructure to maintain efficiency. Together, these requirements provide a robust framework for effective and scalable mental health detection.

7 Design

7.1 Technical Environment

The technical environment for the project ”Analyzing Social Media Posts for Mental Health Disorder Detection” comprises a combination of hardware, software, and tools that enable smooth data analysis, machine learning model training, and deployment. Below is a detailed overview of the minimum hardware configuration, software tools, and package details necessary to carry out this project effectively.

Minimum Hardware Configuration

Given the nature of the project, which involves processing textual data and training machine learning models, the hardware requirements are modest but significant enough to ensure optimal performance. The minimum configuration needed is:

- **Processor** : Intel Core i5 (or equivalent) with a base clock speed of at least 2.5 GHz. A multi-core processor is preferred as it helps in parallel processing, which is essential during model training and data preprocessing steps.
- **RAM** : 8 GB of RAM is recommended to handle the operations of data loading, cleaning, and transformation. Large datasets, like those used in this project, may require more memory to prevent memory overflow errors and reduce delays during processing. For larger datasets, 16 GB of RAM would be ideal.
- **Storage** : At least 256 GB of SSD storage is recommended. Faster storage access significantly impacts loading time for datasets and dependencies. SSD is preferred over traditional HDD because of its faster read/write speeds, which benefit large datasets like the Reddit-based social media posts used in this project.
- **Graphics Processing Unit (GPU)** : For basic machine learning tasks like Logistic Regression or SVM, a dedicated GPU is not necessary. However, if deep learning models or more complex neural networks were introduced later, a GPU like NVIDIA GTX 1060 with 4 GB VRAM or higher would be advantageous.
- **Operating System** : Windows 10 (64-bit) or higher, macOS 10.13 (High Sierra) or higher, or any stable Linux distribution (e.g., Ubuntu 18.04 or higher). The operating system should support all necessary machine learning libraries and be compatible with the tools required for the project.

Software Tools and Packages

For the software stack, the project leverages a set of well-established tools, platforms, and programming libraries to ensure smooth execution from data preprocessing to model deployment:

- **Python** : The primary programming language used for data processing, model training, and evaluation. Python is chosen due to its rich ecosystem of libraries and frameworks tailored for machine learning and data science.
- **Pandas** : A powerful library for data manipulation and analysis, essential for data preprocessing tasks, such as handling missing values and restructuring datasets.
- **scikit-learn** : A comprehensive machine learning library in Python used for implementing and comparing algorithms in classification, regression, and clustering, along with various model evaluation tools.
- **Streamlit** : An open-source Python framework that facilitates the deployment of machine learning models and interactive web applications.
- **Pyngrok** : A Python wrapper for ngrok, used to create secure tunnels to locally deployed applications, which is particularly useful for sharing Streamlit applications over the web.
- **Google Colab** : A cloud-based platform used for writing, executing, and sharing Python code, with access to GPU and TPU resources, beneficial for model training. It integrates seamlessly with libraries like TensorFlow and PyTorch.
- **PRAW (Python Reddit API Wrapper)** : A Python library that allows for easy interaction with the Reddit API to access, retrieve, and analyze Reddit data, such as posts, comments, and user information.
- **pytesseract** : A Python wrapper for Tesseract OCR, used to extract text from images. It's essential for converting image-based text data into a format suitable for processing.
- **Pillow** : A Python Imaging Library that adds support for image processing, which aids in handling image files for text recognition tasks with pytesseract.
- **joblib** : A library for efficient serialization and deserialization of Python objects, especially useful for saving and loading machine learning models during deployment.
- **protobuf** : A protocol buffer library by Google used for serializing structured data, helpful in efficient data exchange between applications.
- **deep-translator** : A library that facilitates easy translation across different languages, enabling multilingual processing of text data.

- **Requests** : A user-friendly library for making HTTP requests, used to retrieve data from APIs or web resources as part of data collection.
- **google-generativeai** : A Python client library for Google's generative AI models, providing tools to integrate and utilize AI functionalities within the project.

7.2 Hierarchy of Modules

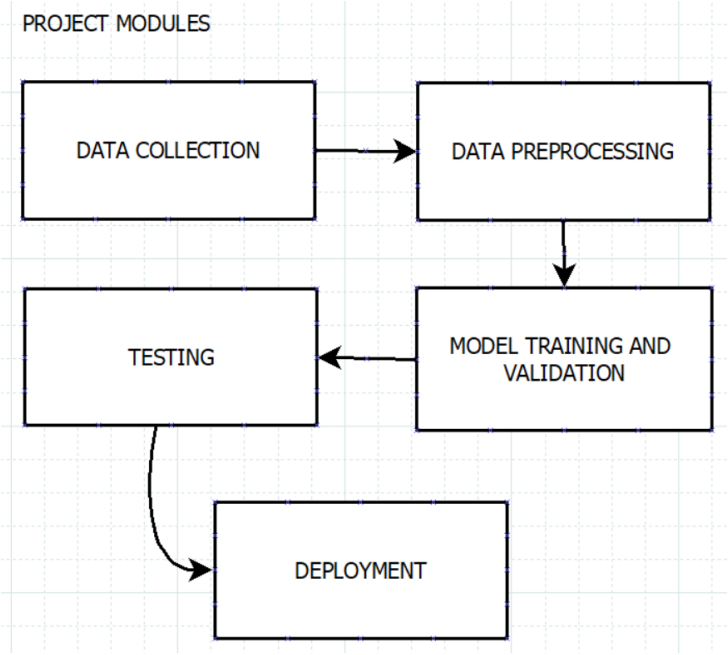


Figure 5: Project Modules

In this project, the system is structured into key modules to classify mental health issues based on text input effectively. The **Data Collection Module** gathers relevant text data, building a comprehensive dataset from sources like CSV files or platforms such as Reddit via PRAW. Next, the **Data Preprocessing Module** loads and cleans this data through tokenization, stopword removal, and lemmatization, preparing it for analysis. Following this, the **Model Training and Validation Module** converts the text into numerical features using techniques like TF-IDF, splitting the data into training and validation sets to test various machine learning models, including Logistic Regression, Naive Bayes, SVM, Random Forest, XGboost and LSTM. Finally, the **Testing and Deployment Module** allows real-time predictions by deploying the model on platforms like Streamlit Cloud, providing an accessible solution for practical applications.

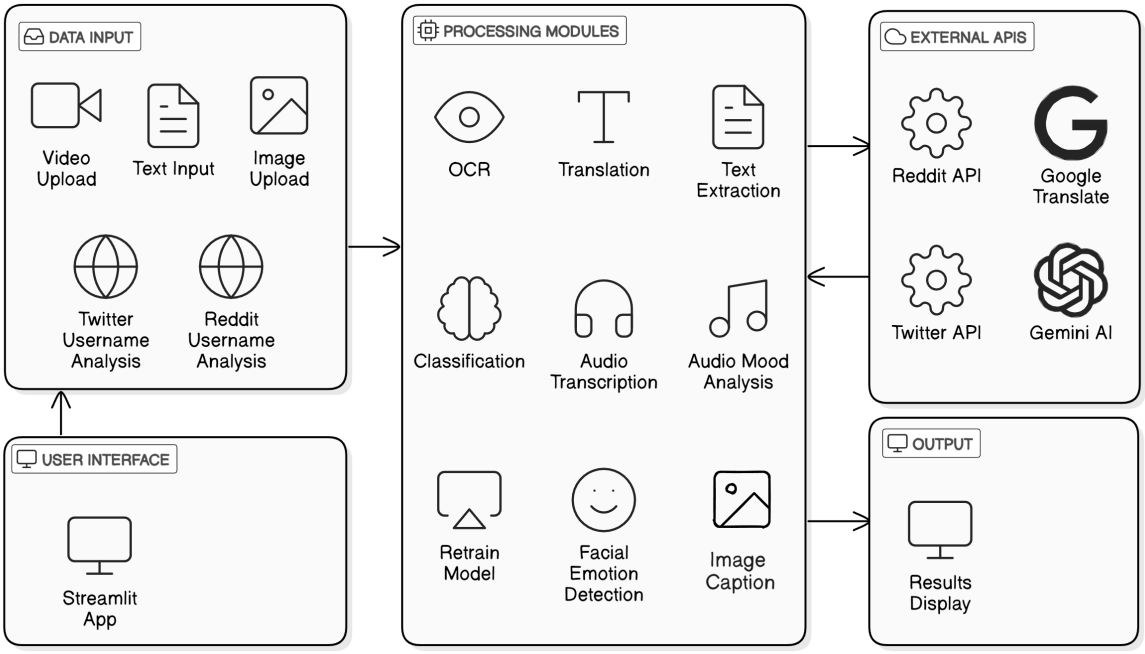


Figure 6: System Overview

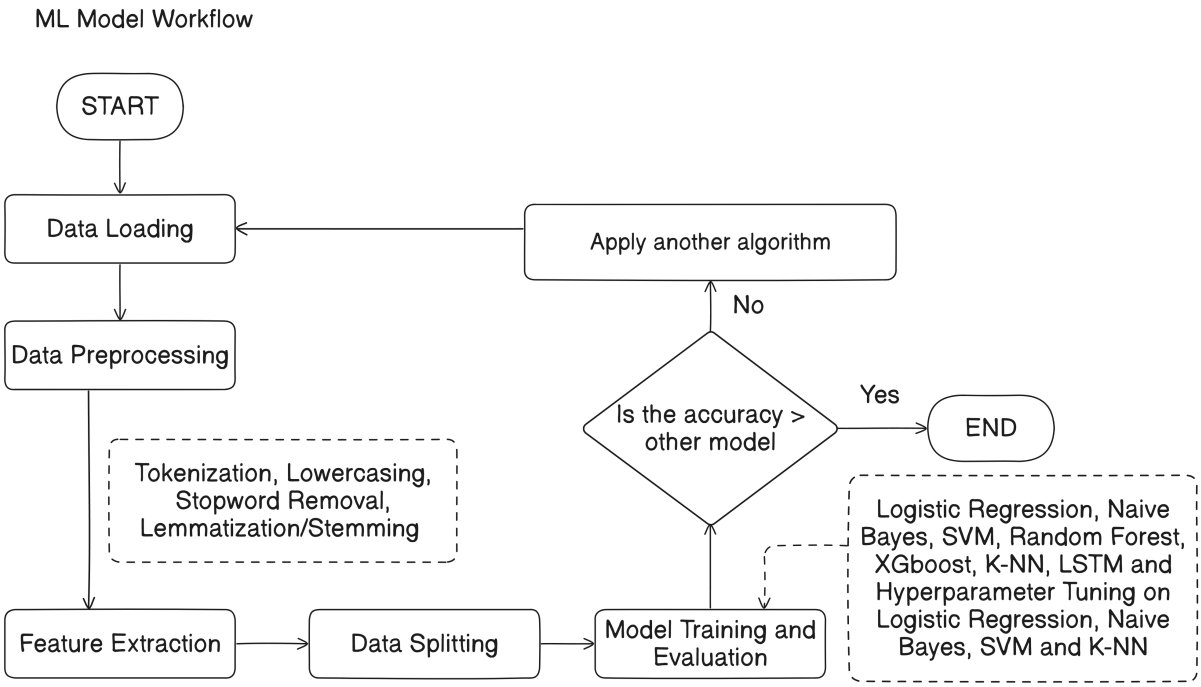


Figure 7: Model Workflow

7.3 Detailed Design

7.3.1 Data Loading and Preprocessing

The Data Loading and Preprocessing Module is the foundation of the system, responsible for ingesting and preparing the text data for analysis. This module begins by loading the dataset from the preprocessed_mental_health_text.csv file, which contains various mental health-related text entries. Once the data is loaded, a series of preprocessing steps are conducted to ensure the text is clean and ready for feature extraction. This includes tokenization, where the text is split into individual words or tokens, and lowercasing to maintain uniformity across the dataset. Additionally, stop-word removal is performed to eliminate common words that do not contribute to the meaning, such as ”and,” ”the,” and ”is.” Finally, lemmatization or stemming is applied to reduce words to their base or root forms. These preprocessing techniques are crucial as they help improve the quality of the input data, ultimately leading to better model performance.

7.3.2 Feature Extraction

In the Feature Extraction Module, the preprocessed text data is transformed into a numerical format that machine learning algorithms can process. This module allows for the selection between two primary feature extraction methods: Term Frequency-Inverse Document Frequency (TF-IDF). The TF-IDF approach evaluates the importance of words in the dataset by considering their frequency in individual documents relative to their overall occurrence across all documents. This helps in highlighting the most informative words. By converting text into numerical features, this module prepares the data for the subsequent training and validation stages, ensuring that the classification models can effectively interpret the input.

7.3.3 Model Training and Validation

The Model Training and Validation Module is critical to developing a robust classification system. In this module, the dataset is split into training and testing sets to evaluate the performance of the models accurately. Various classification algorithms are employed, including Logistic Regression, Naive Bayes, Support Vector Machines, Random Forest, XGboost, K-Nearest-Neighbours and LSTM. Each model is trained on the training set, which involves adjusting the model parameters based on the input features and their corresponding labels. Following training, the models undergo validation to assess their performance using various metrics such as accuracy, precision, recall, and F1-score. A decision point is included to determine if the achieved accuracy meets the project requirements. If the accuracy is deemed acceptable, the model is accepted.

7.3.4 Prediction

The Prediction Module is designed to provide real-time classification of new input text related to mental health issues. Upon receiving user input, this module initiates a preprocessing workflow that mirrors the steps applied during the training phase, including tokenization, lowercasing, stop-word removal, and lemmatization or stemming. Once the input text is preprocessed, it is fed into the trained classification models to generate predictions. Each model may provide a classification result, allowing for a comprehensive analysis of the input. This module not only delivers the predicted mental health issue but also ensures that users receive an informative output that reflects the confidence level of each prediction, enabling them to understand the model's reasoning.

7.3.5 Testing and Deployment

The module focuses on making the trained models accessible for real-time predictions. Once the models have been validated and selected based on their performance, this module prepares them for deployment on suitable platforms like Streamlit Cloud. This involves packaging the models and creating a user interface where users can input text and receive predictions. The deployment process also includes considerations for scaling, ensuring that the system can handle multiple requests simultaneously while maintaining responsiveness. By providing a free and efficient deployment solution, this module enables users to access the mental health classification service easily. The deployment of the models ensures that the insights generated from the analysis can be utilized effectively in real-world applications.

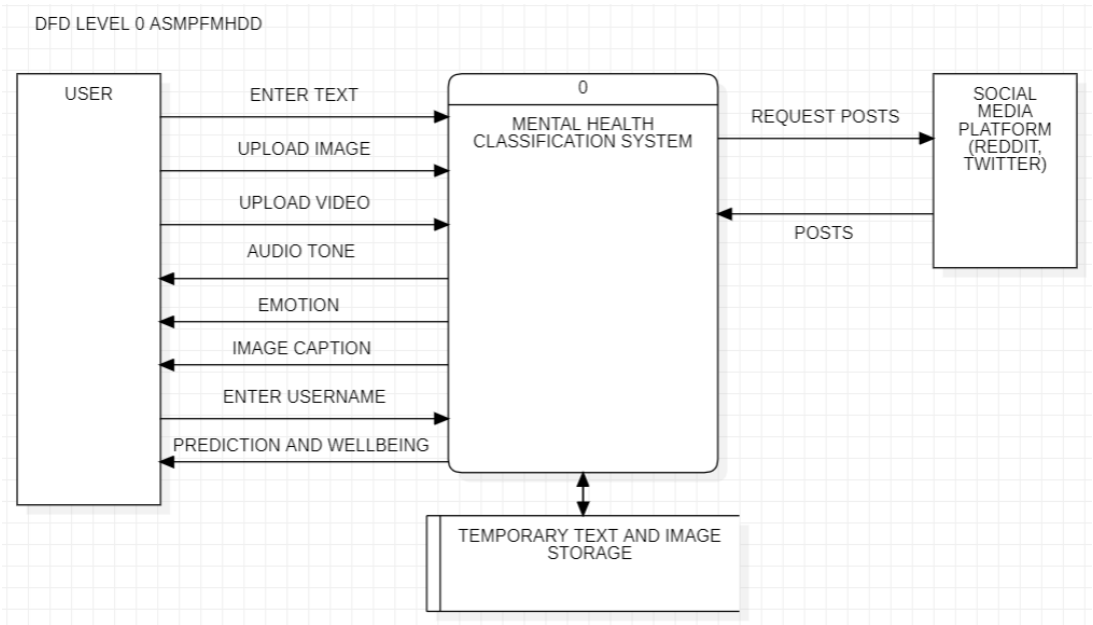


Figure 8: DFD Level 0 of the System

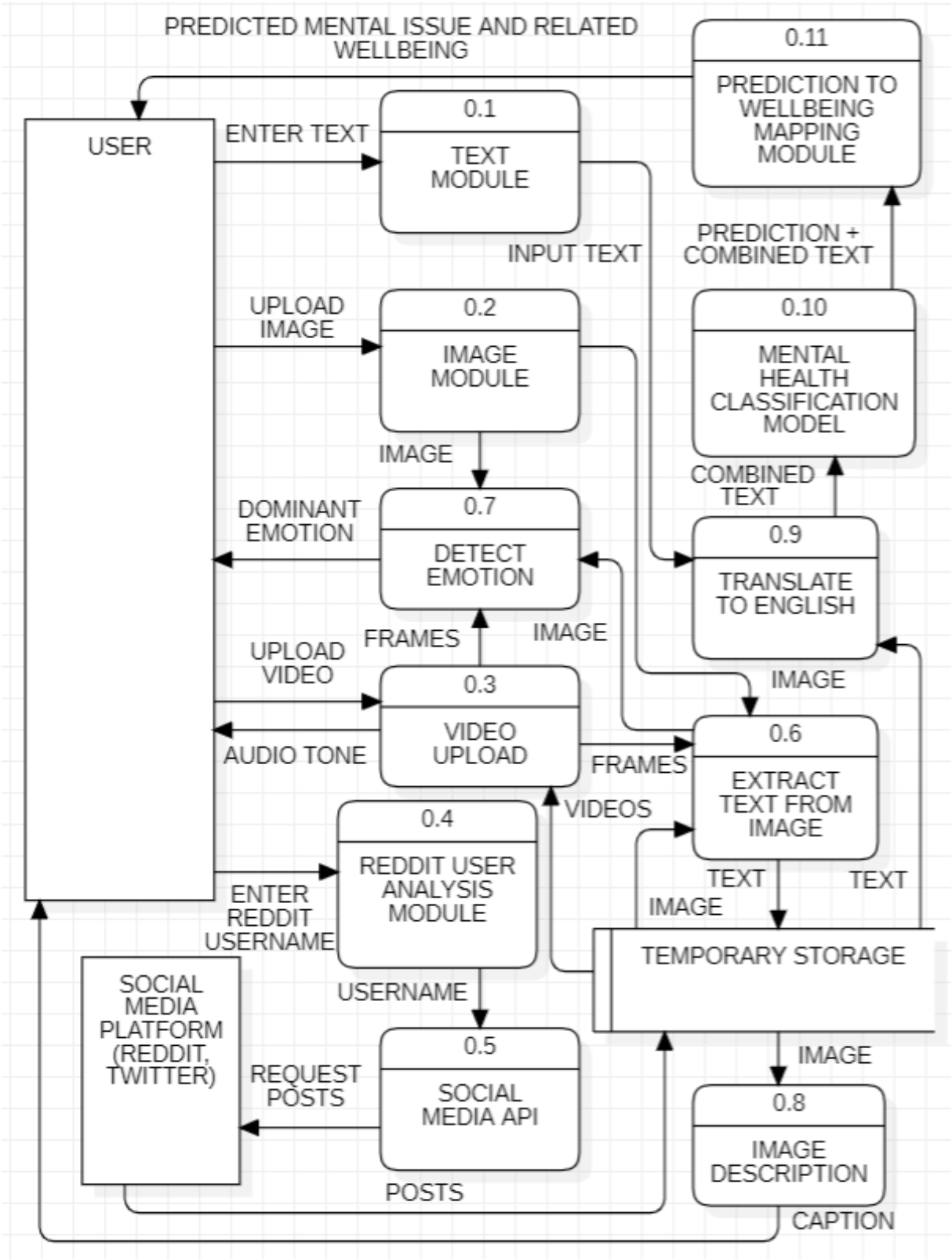


Figure 9: DFD Level 1 of the System

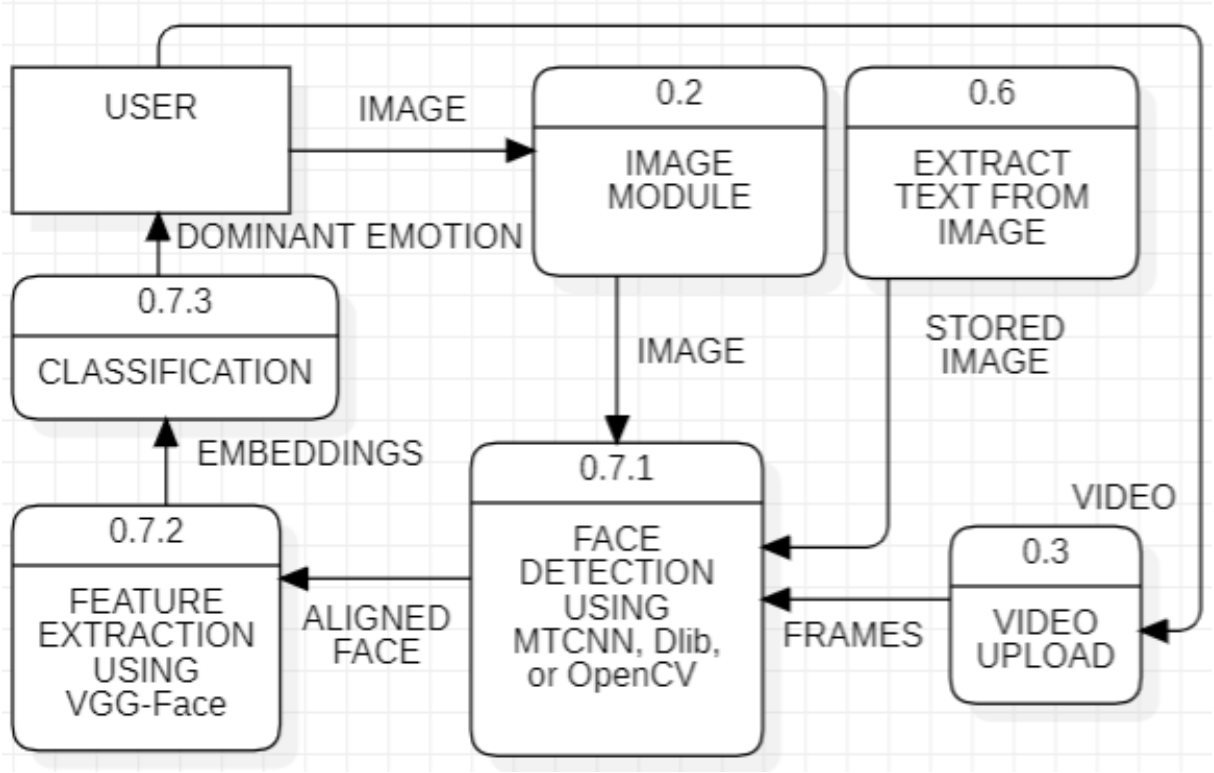


Figure 10: DFD Level 2 of Emotion Detection Functionality

7.4 Emotion Detection Functionality

DeepFace analysis involves several steps to process and analyze facial features from an uploaded image or video frame. The process begins with the user uploading the input, which is passed to DeepFace's pipeline. The first step is face detection, where models like MTCNN, Dlib, or OpenCV locate faces within the image. MTCNN (Multi-task Cascaded Convolutional Networks) detects faces using cascaded CNNs for high accuracy and alignment, while Dlib employs histogram-oriented gradients and machine learning techniques for robust detection. OpenCV, a widely-used computer vision library, uses Haar cascades or DNN-based approaches to detect faces. Once detected, the faces are cropped and aligned for consistency. Next is feature extraction, where pre-trained models like VGG-Face (a deep CNN trained specifically for face recognition) or Facenet (which uses triplet loss to generate highly discriminative face embeddings) convert the aligned face into numerical vectors called embeddings. These embeddings are compared using techniques like cosine similarity or Euclidean distance for tasks such as emotion detection(e.g., happiness, sadness), demographic analysis (e.g., age, gender), or face verification (matching two faces). Finally, the results, such as detected emotions or attributes, are displayed to the user, completing the analysis pipeline.

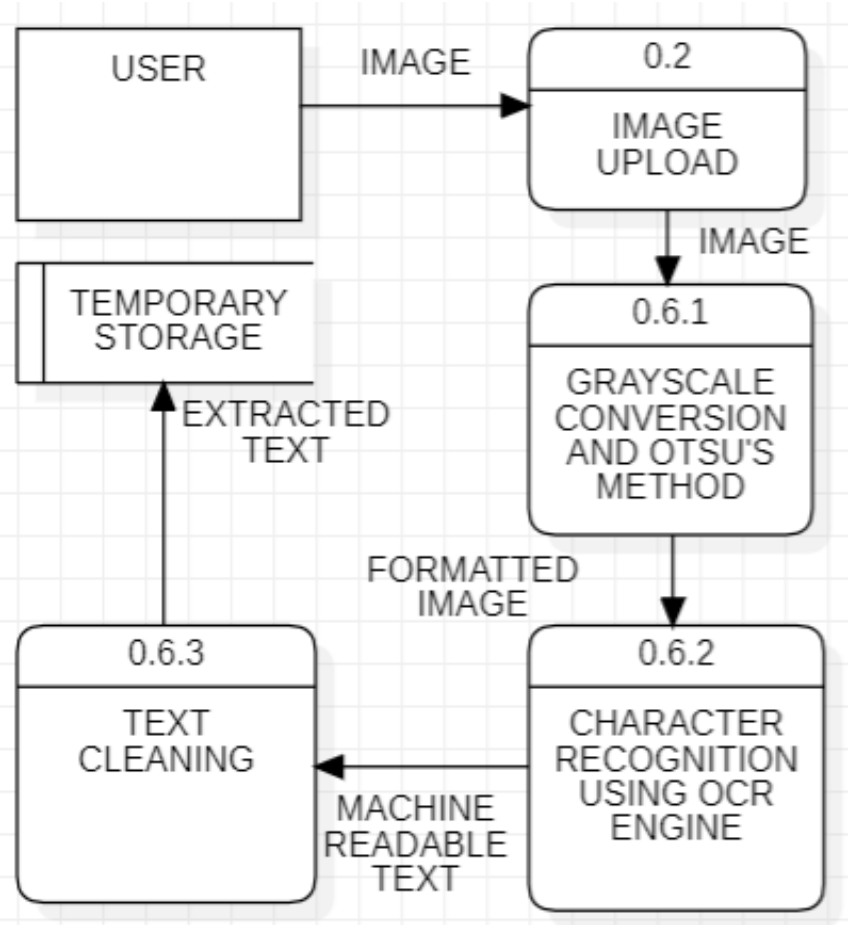


Figure 11: DFD Level 2 of Text Extraction from Image

7.5 Extract Text From Image

The process of extracting text from an image using Tesseract-OCR begins with the user uploading an image containing text, such as a scanned document or a photo of a sign. This raw input undergoes preprocessing to enhance the image for OCR recognition. The first step in preprocessing is grayscale conversion, where the image's RGB values are reduced to a single intensity value using a weighted formula. This simplifies the image by removing color information while retaining the text regions. Next, noise reduction is applied using techniques like Gaussian Blur or Median Blur, which smooth out artifacts while preserving the edges of text. After noise reduction, binarization is performed to convert the grayscale image into a binary format (black-and-white) using methods like Otsu's thresholding. This step isolates the text from the background. Region detection follows, where text-heavy areas are identified using contour detection or connected component analysis, filtering out non-text regions based on properties like aspect ratio and alignment. Once preprocessing is complete, the image is passed to Tesseract for character recognition. This involves segmenting the image into text lines and individual characters, which are then processed by Tesseract's LSTM-based neural network. The network

evaluates sequences of characters in context, improving the accuracy of word recognition. Recognized characters are combined into machine-readable text, leveraging language models and dictionary lookups to maintain the natural flow and correct structure of the text. The output is a string of recognized text, but it may still require postprocessing for better accuracy and readability. Postprocessing begins with error correction, where the recognized text is compared against dictionaries or spell-checking algorithms to identify and fix errors caused by OCR misrecognition. Rule-based corrections are also applied, such as replacing common misinterpreted characters (e.g., 0̀with Ò0̀r l̀with 1̀) depending on the context. After error correction, formatting is applied to structure the text into paragraphs, reconstruct line breaks, and retain alignments. This ensures that the extracted text is not only accurate but also well-organized for further use. Finally, the extracted text is either displayed to the user or saved to a file format such as ‹txt0̀r ‹docx‹ This structured and cleaned output can then be utilized for various applications, from document analysis to content storage. The entire pipeline, from preprocessing to postprocessing, ensures a high-quality extraction process by combining image enhancement, deep learning-based recognition, and advanced text correction methods.

7.6 Image Description

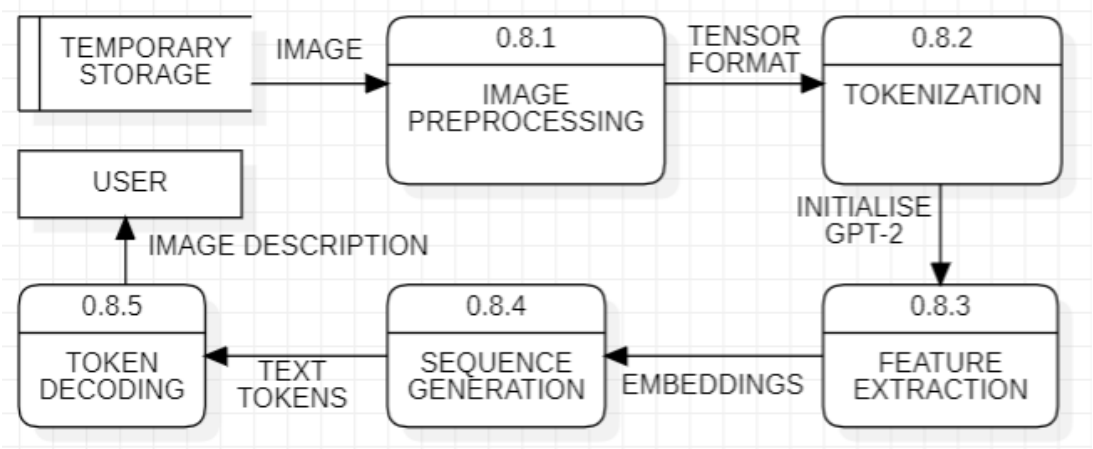


Figure 12: DFD Level 2 of Image Description

The process of generating an image description using Python's Transformers module and the ViT-GPT2 model begins with the user input, where the user uploads an image to be described. This image can be in any format (e.g., PNG, JPEG), and it is the raw data that will undergo a series of transformations before the description is generated. Once the image is received, it proceeds to the preprocessing stage. Preprocessing involves multiple steps: first, the image is resized to match the input dimensions required by the Vision Transformer (ViT) model, typically 224×224 pixels. This ensures uniformity across all input data. The pixel values are then normalized to fall within a standard range (e.g., between 0 and 1), which improves the efficiency

and stability of the model's computations. The image is subsequently transformed into a tensor format, which is the data structure required for processing by PyTorch or TensorFlow-based deep learning models. At this stage, the **GPT-2 tokenizer** is also initialized, preparing it to convert tokens into text during the sequence generation phase. Following preprocessing, the system moves to feature extraction, where the Vision Transformer (ViT) processes the image to extract meaningful features. ViT divides the image into smaller patches, typically 16×16 pixels, and flattens each patch into a vector. These vectors are embedded using a linear projection layer, and positional embeddings are added to encode the spatial arrangement of the patches. The embedded patches are then passed through several transformer layers, where self-attention mechanisms analyze relationships between patches to capture both local and global image features. The final output of ViT is a high-dimensional embedding vector that represents the visual content of the image. This embedding vector is then passed to the GPT-2 decoder for the sequence generation phase. GPT-2 uses the image embeddings as input and applies its pre-trained language model to generate a sequence of text tokens. This is achieved through an iterative process where GPT-2 predicts the next token in the sequence based on the embeddings and previously generated tokens. The model continues this process until it reaches a stopping condition, such as an end-of-sequence (EOS) token. The generated tokens are numerical representations of words and phrases. Next comes the postprocessing stage, where the numerical tokens generated by GPT-2 are decoded back into human-readable text using the GPT-2 tokenizer. This step involves mapping the numerical tokens to their corresponding words in the vocabulary. The decoded text is then cleaned up for readability, ensuring proper grammar and structure. Any formatting issues are addressed during this stage to produce coherent and meaningful sentences. Finally, the system outputs the image description to the user. The description is a concise and accurate textual representation of the image content, such as "A group of people playing soccer on a field." This description can be displayed on the application's interface or saved to a file for further use. By leveraging the advanced capabilities of the Vision Transformer for feature extraction and GPT-2 for language modeling, this pipeline provides an efficient and accurate solution for automated image captioning.

7.7 Translation to English

The process of translating text to English using DeepTranslator begins with the user providing input, either by typing text in a non-English language or uploading a file containing text. This input is handled by the system, which prepares it for translation. The text is first preprocessed to ensure optimal translation results. This involves cleaning the input by removing unwanted characters such as extra whitespace, punctuation errors, or HTML tags that could interfere with accurate translation. If the source language is not specified, a language detection step is performed. This step uses machine learning models trained on multilingual datasets to analyze pat-

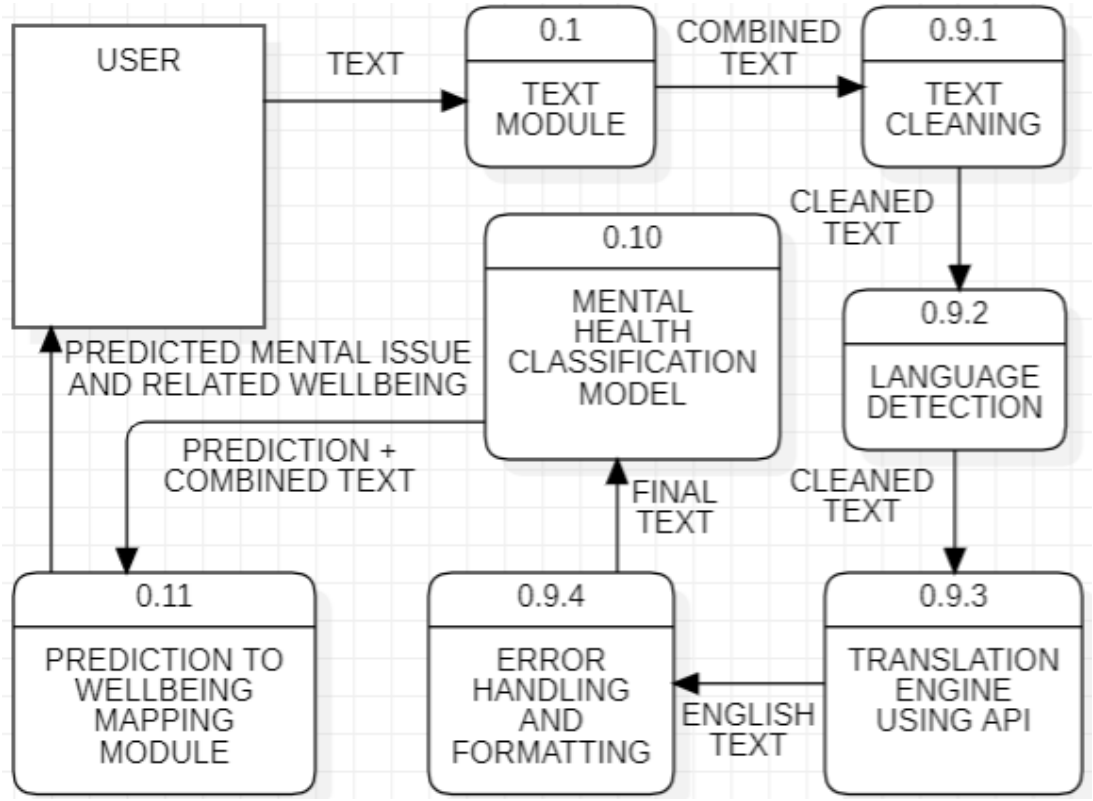


Figure 13: DFD Level 2 of Translation to English

terns in the text, such as character distribution and word frequency, to determine the language. Models like Google's Compact Language Detector employ statistical and supervised learning methods for this purpose. Once the text is prepared, it is passed to the DeepTranslator module, which interfaces with translation APIs like Google Translate or Microsoft Translator. These APIs rely on advanced neural machine translation (NMT) techniques to perform the translation. NMT models, which are based on deep learning, use sequence-to-sequence (seq2seq) architectures with encoder-decoder frameworks. The encoder processes the input text and converts it into a latent representation, a numerical vector capturing its meaning. This representation is then passed to the decoder, which generates the translated text in English. To ensure contextual accuracy, these models utilize attention mechanisms, which help focus on relevant parts of the input sentence during translation. Modern transformer-based architectures, like those used in Google Translate, enhance translation performance by enabling parallel processing of text and capturing long-range dependencies in sentences. After translation, the system postprocesses the output to ensure quality and readability. This includes validating the translated text for completeness, handling errors like untranslated phrases, and retrying failed requests if necessary. The output is formatted to resemble the original structure of the input, preserving elements like paragraphs, line breaks, and punctuation to maintain coherence. Finally, the translated text is displayed or saved, ensuring it is clean, accurate, and easy to understand. Throughout

this process, concepts from natural language processing (NLP), deep learning, and robust error handling are integrated to provide reliable and efficient translations.

7.8 Prediction to Wellbeing Mapping

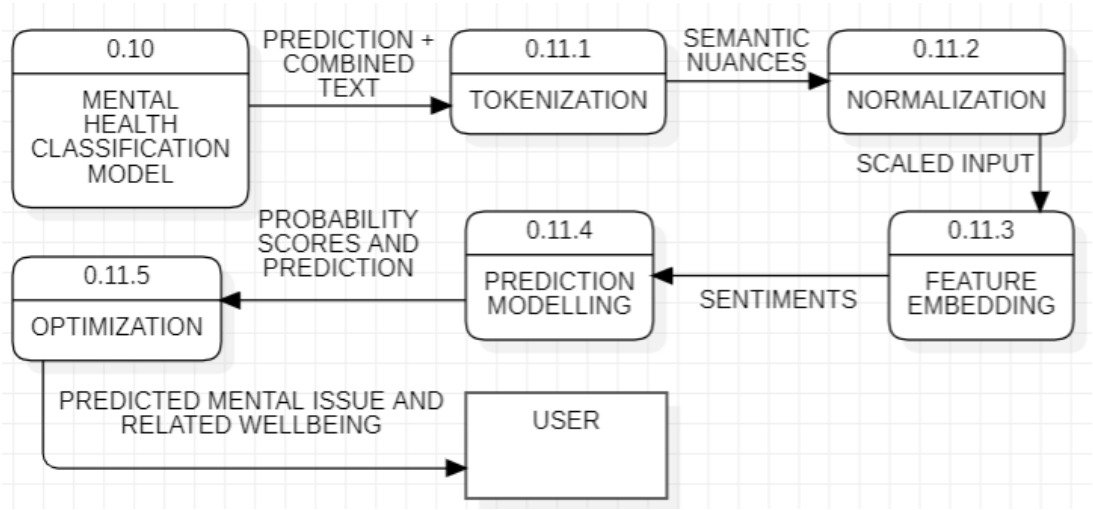


Figure 14: DFD Level 2 of Prediction to Wellbeing Mapping

The process begins when the user provides input data for prediction, which could be in the form of text, behavioral data, or health metrics. For instance, a user might input their mood and health conditions, share behavioral patterns, or upload relevant data such as physical activity logs or medical records. This data serves as the foundation for predicting the user's mental health or overall wellbeing. Once the data is provided, the next step is preprocessing, which involves cleaning and transforming the raw input. During the data cleaning phase, any noise, missing values, or redundant information are identified and removed. This ensures the dataset is clean and ready for processing. Additionally, feature engineering is performed to extract meaningful attributes or characteristics from the raw data. For example, from behavioral data, features like activity frequency or sleep patterns may be derived. This allows the model to work with relevant information that enhances the prediction process. The third step involves embedding generation, where the core model, GEMINI 1.5 FLASH, processes the preprocessed input to generate embeddings. Embeddings are high-dimensional vectors that capture the semantic and contextual properties of the data. For example, when analyzing text input, GEMINI 1.5 FLASH uses its deep learning algorithms to understand the context and meaning of the user's words, transforming them into embeddings that encapsulate both the surface-level meaning and the deeper nuances of the data. This representation of the input data is essential for subsequent prediction tasks. In the prediction model phase, the generated embeddings are passed to machine learning models, which use these embeddings to predict a user's mental health status or behavioral outcomes. These models may utilize various algorithms, such as decision trees,

support vector machines, or deep learning models, to make predictions. The system not only predicts the mental health condition but also maps these predictions to various wellbeing dimensions. These dimensions could include emotional wellbeing, such as happiness or stress; physical wellbeing, like energy levels or fatigue; and social wellbeing, including aspects like connectedness or isolation. By doing so, the system offers a comprehensive view of the user's overall wellbeing based on the predictions. Once predictions are made, postprocessing ensures the results are accurate and formatted properly. During the validation phase, the predictions are checked for any anomalies or inconsistencies, ensuring the output is reliable. This might involve techniques like consistency checks or outlier detection to verify that the predictions align with expected patterns. After validation, the results are formatted into a user-friendly structure. This might include organizing the wellbeing data into clear categories, such as emotional, physical, and social wellbeing, making it easy for the user to interpret. Finally, the system delivers the output, displaying or saving the prediction and associated wellbeing mapping. This output is presented in a way that helps the user understand the various aspects of their mental and physical health, giving them insights into their current state of wellbeing and providing actionable data for improvement.

7.9 Audio Mood Analysis

The analyze_audio_mood function begins with the user providing a path to a video file as input. This is the starting point of the process where the user supplies the video from which the audio will be extracted. In the preprocessing stage, the video file is passed to the extract_audio_from_video function. This function extracts the audio content from the video, separating it for further analysis. Once the audio is extracted, the next step involves loading the audio file into memory using the librosa library, which prepares the audio data for feature extraction. The core of the audio processing happens during the MFCC (Mel-frequency cepstral coefficients) feature extraction. The librosa.feature.mfcc method is used to compute the MFCCs of the audio. These MFCCs represent different frequency bands of the audio signal and are crucial for analyzing the characteristics of the audio that correspond to various emotional tones. By capturing frequency patterns, MFCCs provide a robust feature set for mood classification. After the MFCCs are extracted, the next step is feature segmentation. The MFCC array is divided into four distinct frequency bands: low frequencies (MFCCs 0, 1, and 2), mid-low frequencies (MFCCs 3 and 4), mid-high frequencies (MFCCs 5, 6, and 7), and high frequencies (MFCCs 8, 9, 10, 11, and 12). For each of these frequency bands, the scalar mean of the MFCC values is computed. This scalar mean represents the overall intensity of the audio in each frequency range, simplifying the data for classification purposes. Finally, the mood classification process takes place. The scalar means for each of the frequency bands are compared to determine the dominant mood of the audio. Based on these comparisons, the function classifies the audio as

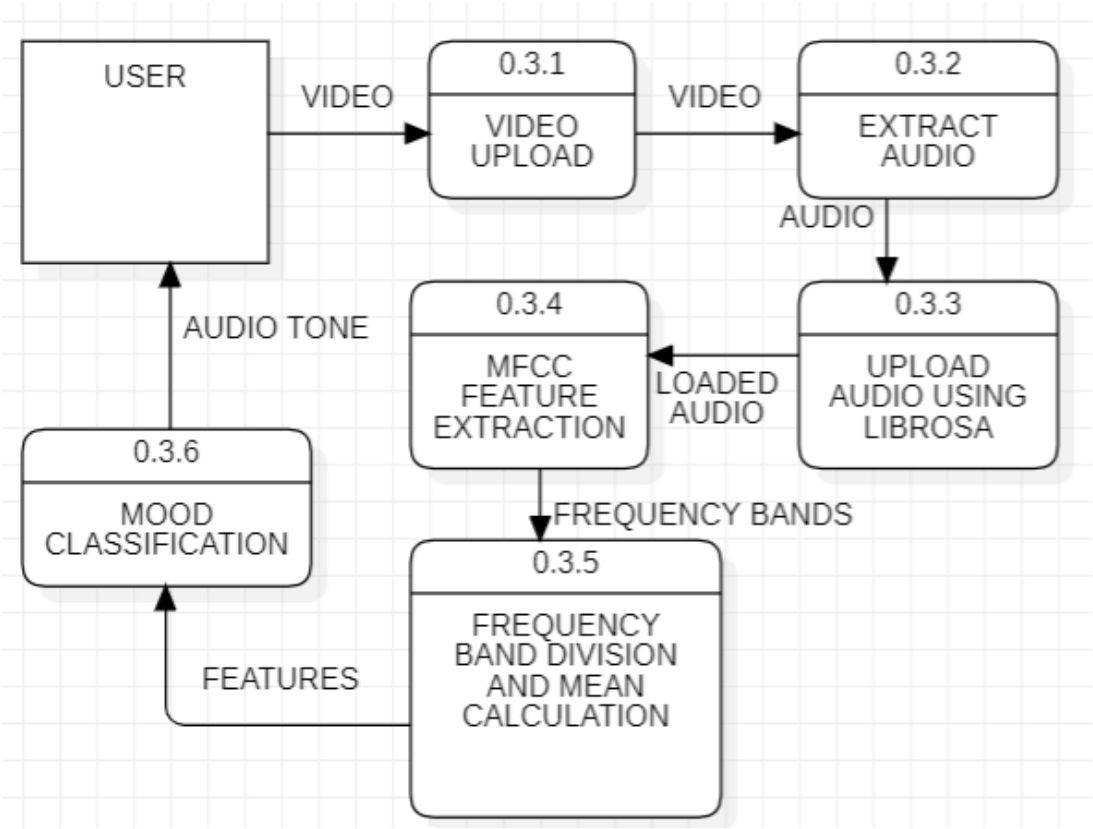


Figure 15: DFD Level 2 of Audio Mood Analysis

normal, neutral, melancholic, calm, anxious, or happy, depending on which frequency bands exhibit the strongest characteristics. This mood classification is then returned as the result of the analysis.

The various processes and technologies discussed throughout this conversation can play a pivotal role in enhancing the classification and prediction of mental health disorders. The use of machine learning (ML) and deep learning (DL) models enables the system to process and analyze large volumes of data, such as text, behavioral patterns, health metrics, and even multimedia inputs (e.g., images, audio, and video). By leveraging the DeepFace module for facial emotion recognition, the system can identify emotional cues from visual data, helping to assess a user's emotional state. This is complemented by text analysis models, such as those built with Logistic Regression, k-NN, SVM, and LSTM architectures, which are capable of classifying mental health issues from user-provided text, such as social media posts, personal narratives, or other forms of written communication.

Further enhancing the capability, the integration of DeepTranslator ensures that non-English input can be translated into English, allowing the system to support a wider variety of users and

ensuring the accuracy of the mental health analysis, even when the data comes from different languages. Meanwhile, behavioral data and health metrics are crucial inputs that can provide insights into a person's overall wellbeing. These data sources are analyzed using GEMINI 1.5 FLASH, which generates embeddings capturing the semantic and contextual properties of the input data, feeding into prediction models that classify users' mental health conditions and map them to various wellbeing dimensions. The classification of conditions like anxiety, depression, PTSD, and others can be made more accurate through these comprehensive data analyses.

By incorporating audio mood analysis through techniques such as MFCC for audio emotion detection and facial expression recognition for video inputs, the system further refines its ability to assess emotional and mental states. These audio and visual analyses provide additional layers of understanding that text alone might not fully capture. In conjunction with these features, streaming data from social media platforms like Reddit or Twitter allows for the monitoring of a user's public posts, providing real-time insights into their mental health as they interact with others online.

The combined use of these models and processes facilitates a holistic approach to mental health disorder classification. The system can classify mental health issues based on a range of factors, including written text, emotional tone from images, behavioral patterns, and audio cues. This multimodal approach helps create a more comprehensive, accurate, and nuanced prediction of mental health conditions, offering personalized insights into emotional, physical, and social wellbeing. As a result, users benefit from a system that not only detects potential mental health issues but also offers a deeper understanding of their overall wellbeing, which can inform targeted interventions, support, and wellbeing strategies.

8 Implementation

8.1 Features From RM

For the initial prototype development, a subset of the requirements from the Requirement Matrix (RM) was carefully selected to focus on implementing core functionalities and demonstrating proof of concept. The selection was based on a combination of high-priority functional requirements that form the backbone of the system, ensuring that critical features are built and validated before expanding the scope.

The requirements chosen for the prototype primarily involve data preprocessing, model training, and evaluation processes. These requirements were selected because they are fundamental to the project's success, ensuring that the data pipeline and model implementation work seamlessly together. This subset of features lays the groundwork for later integration with additional components and more complex functionality.

The filtered part of the RM focuses on the following high-priority requirements: data cleaning and feature extraction, training of machine learning models, and evaluation of model performance using standard metrics. These requirements were identified as crucial because they directly impact the system's ability to handle data, learn patterns, and provide meaningful outputs. Without successfully implementing these core features, the overall effectiveness of the solution would be significantly reduced.

Furthermore, these selected features align with the project's goals and provide a clear pathway for incremental development. By narrowing down the requirements to these foundational aspects, the development team can ensure that the prototype is not only functional but also extensible, providing a robust framework for future enhancements.

Rqmt ID	*Requirement Item*	*Requirement Analysis Status*
FR-001	Collect social media data from Reddit.	Completed
FR-002	Implement data cleaning and preprocessing.	Completed
FR-003	Train machine learning and deep learning models.	Completed
FR-004	Evaluate models using performance metrics (accuracy, recall, F1 Score, Support).	Completed
NFR-001	Testing different features of the web application	Completed
NFR-002	Final Web Application Deployment	Completed

Figure 16: Features from Requirement Matrix

8.2 Code Details and Output

8.2.1 Data Collection

Collecting Reddit Posts

```
import praw
import pandas as pd
import time
# Initialize Reddit API
reddit = praw.Reddit(client_id='YOUR_CLIENT_ID',
                     client_secret='YOUR_CLIENT_SECRET',
                     user_agent='Mental Health Classifier')
# Define subreddits and post types
subreddits = {'normal': ['news', 'AskReddit'],
              'depression': ['depression'],
              'ptsd': ['PTSD'],
              'anxiety': ['Anxiety'],
              'bipolar': ['BipolarReddit']}
post_types = ['hot', 'new', 'top']
posts_per_type = 100
# Collect and save posts
data = []
for label, subs in subreddits.items():
    for sub in subs:
        for post_type in post_types:
            posts = getattr(reddit.subreddit(sub), post_type)(
                limit=posts_per_type)
            for post in posts:
                data.append([post.title + " " + post.selftext,
                    label])
            time.sleep(1)
df = pd.DataFrame(data, columns=['text', 'label'])
df.to_csv(f'{label}_dataset.csv', index=False)
```

	text	mental_health_issue
0	2024 Election Due to the 2024 US Presidential ...	bipolar
1	Our most-broken and least-understood rules is ...	depression
2	Rules **UPDATED** 1. If you're here you must p...	normal
3	Happy Cakeday, r/Normal! Today you're 11 Let's...	normal
4	Happy Cakeday, r/Normal! Today you're 10 Let's...	normal
5	I am also a person I am very normal. Today I t...	normal
6	Happy Cakeday, r/Normal! Today you're 9 Let's ...	normal
7	I am a person I am a person	normal
8	Elections and Politics Hello friends!\n\nIt's ...	anxiety
9	You are more than just one emotion	ptsd

Figure 17: Obtained Dataset

Combining Collected Datasets

```
import pandas as pd
from sklearn.utils import shuffle

# Load datasets
bipolar_df = pd.read_csv("bipolar_dataset.csv")
depression_df = pd.read_csv("depression_dataset.csv")
normal_df = pd.read_csv("normal_dataset.csv")
anxiety_df = pd.read_csv("anxiety_dataset.csv")
ptsd_df = pd.read_csv("ptsd_dataset.csv")

# Minimum length for balanced classes
min_length = min(len(bipolar_df), len(depression_df), len(
   normal_df) // 6, len(anxiety_df), len(ptsd_df))

# Create balanced pattern
pattern_data = []
for i in range(min_length):
    pattern_data.extend([bipolar_df.iloc[i], depression_df.iloc[
       i]] +
                        normal_df.iloc[i*6:(i+1)*6].to_dict('
                           records') +
                        [anxiety_df.iloc[i], ptsd_df.iloc[i]])

pattern_df = pd.DataFrame(pattern_data)
remaining_data = shuffle(pd.concat([bipolar_df[min_length:],
   depression_df[min_length:],
                                    normal_df[min_length*6:],
                                       anxiety_df[min_length:],
                                       ptsd_df[min_length:]]))

final_df = pd.concat([pattern_df, remaining_data]).reset_index(
   drop=True)
final_df.to_csv("mental_health_combined.csv", index=False)
```

The first code snippet collects Reddit posts by connecting to specific subreddits associated with various mental health topics. Using the Reddit API, posts are retrieved from ‘hot‘, ‘new‘, and ‘top‘ categories for each mental health type, including general subreddits (for the ”normal” category). Each post’s title and content are combined, labeled, and saved into separate CSV files. The second code snippet merges these CSV files to form a combined dataset. It creates a balanced sample by selecting the minimum number of rows across each category and organizes the data in a structured pattern. Remaining rows are shuffled and appended, producing a final dataset suitable for machine learning applications in mental health classification.

The dataset for mental health classification was compiled from various subreddit communities.

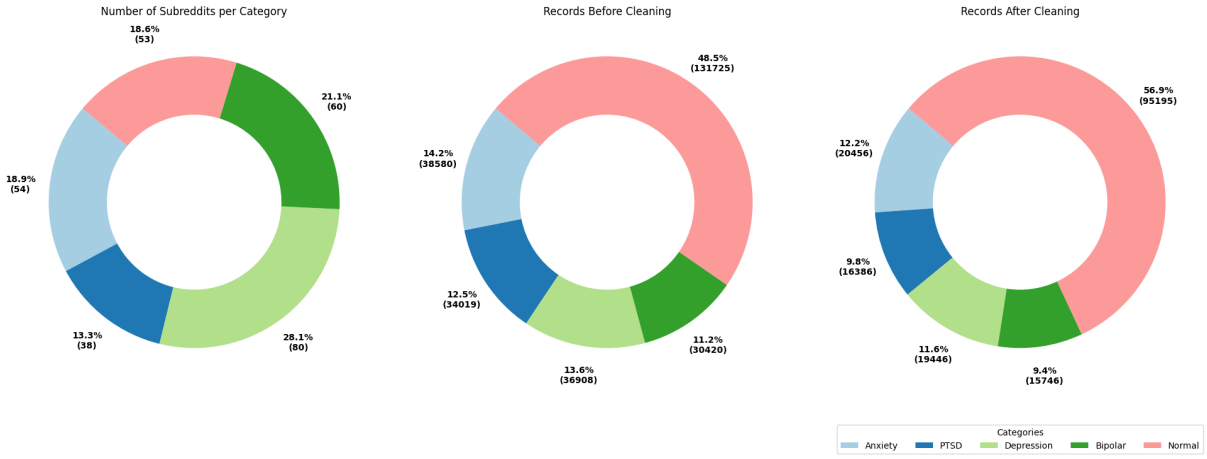


Figure 18: Collected Data Statistics

Initially, the dataset contained records from 54 subreddits for anxiety, 38 for PTSD, 80 for depression, 60 for bipolar disorder, and 53 for normal mental states. Before data cleaning, the records totaled 38580 for anxiety, 34019 for PTSD, 36908 for depression, 30420 for bipolar, and 131725 for normal, reflecting the distribution of raw data. After applying data cleaning techniques to ensure quality and relevance, the record counts were reduced to 20456 for anxiety, 16386 for PTSD, 19446 for depression, 15746 for bipolar, and 95195 for normal. A subset of this cleaned dataset containing 18596 records was used for further processing and analysis.

Given the large size of the original dataset, training and testing the model on the entire dataset would require significant computational resources, including memory and processing power. Such constraints can lead to prolonged training times and potentially infeasible execution on standard hardware. By using a subset of the cleaned dataset, we can significantly reduce the computational load while maintaining a representative sample of the data. This approach strikes a balance between efficiency and performance, enabling the development and evaluation of models in a more manageable and resource-friendly manner. Moreover, working with a subset ensures faster experimentation and fine-tuning cycles, which is crucial for iterative improvement in machine learning workflows.

8.2.2 Data Preprocessing

Text Preprocessing

```
import pandas as pd
import re
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk

# Download stopwords (if you haven't already)
nltk.download('stopwords')
nltk.download('punkt')

# Load the dataset
df = pd.read_csv('mental_health.csv')

# 1. Handling Missing Values
df.dropna(subset=['text'], inplace=True)

# 2. Removing duplicates (if any)
df.drop_duplicates(subset=['text'], inplace=True)

# 3. Text Preprocessing
negative_words = {"not", "no", "nor", "never", "nothing", "
   nowhere", "neither", "cannot", "n't", "without", "barely", "
   hardly", "scarcely"}

def clean_text(text):
    text = re.sub(r'http\S+', '', text)  # Remove URLs
    text = re.sub(r'@\w+', '', text)  # Remove mentions (
       @username)
    text = re.sub(r'[^a-zA-Z\s]', '', text)  # Remove special
       characters, numbers, and punctuations
    text = text.lower()  # Convert text to lowercase
    tokens = word_tokenize(text)  # Tokenize the text
    tokens = [word for word in tokens if word not in stopwords.
       words('english') or word in negative_words]  # Remove
       stopwords, but keep negative words
    clean_text = '␣'.join(tokens)  # Join the tokens back into a
        single string
    return clean_text

df['cleaned_text'] = df['text'].apply(clean_text)  # Apply the
   cleaning function to the 'text' column
```

Text Preprocessing

```
# 4. Feature Extraction using TF-IDF Vectorization
vectorizer = TfidfVectorizer(max_features=10000)  # Adjust the
    max_features
X = vectorizer.fit_transform(df['cleaned_text'])  # Fit and
    transform the cleaned text data
X_df = pd.DataFrame(X.toarray(), columns=vectorizer.
    get_feature_names_out())  # Convert the result to a DataFrame
     for easier understanding

# Save the preprocessed dataset (optional)
df.to_csv('preprocessed_mental_health.csv', index=False)
```

The code snippet above demonstrates the preprocessing steps for a mental health dataset. It handles missing values and duplicates in the text data, performs text cleaning by removing URLs, mentions, and special characters, and applies tokenization. The cleaned text is then vectorized using TF-IDF to convert the textual data into a format suitable for machine learning models. Finally, the preprocessed dataset is saved as a new CSV file for future use in classification tasks.

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
   aaa  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa   ab  abandon  abandoned  \
0  0.0                                     0.0  0.0      0.0        0.0
1  0.0                                     0.0  0.0      0.0        0.0
2  0.0                                     0.0  0.0      0.0        0.0
3  0.0                                     0.0  0.0      0.0        0.0
4  0.0                                     0.0  0.0      0.0        0.0

   abandoning  abandonment  abc  abilities  ability  ...  zemeckis  zemlja  \
0         0.0          0.0  0.0        0.0  0.00000  ...       0.0     0.0
1         0.0          0.0  0.0        0.0  0.03726  ...       0.0     0.0
2         0.0          0.0  0.0        0.0  0.00000  ...       0.0     0.0
3         0.0          0.0  0.0        0.0  0.00000  ...       0.0     0.0
4         0.0          0.0  0.0        0.0  0.00000  ...       0.0     0.0

   zero  zoloft  zombie  zombieland  zombies  zone  zoom  zuckerberg
0   0.0     0.0     0.0         0.0      0.0   0.0   0.0         0.0
1   0.0     0.0     0.0         0.0      0.0   0.0   0.0         0.0
2   0.0     0.0     0.0         0.0      0.0   0.0   0.0         0.0
3   0.0     0.0     0.0         0.0      0.0   0.0   0.0         0.0
4   0.0     0.0     0.0         0.0      0.0   0.0   0.0         0.0

[5 rows x 10000 columns]
```

Figure 19: Data Collection and Preprocessing

8.2.3 Vectorization Technqiues

Bag of Words

```
from sklearn.feature_extraction.text import CountVectorizer

# Initialize the CountVectorizer and fit/transform the cleaned
   text
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(dataset['cleaned_text'])
```

Bag of Words (BoW) converts text into a matrix of word counts. Each unique word in the corpus becomes a feature, and the value is the word's frequency in the document, ignoring grammar and word order.

TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Initialize the TfidfVectorizer and fit/transform the cleaned
   text
TFIDFvectorizer = TfidfVectorizer()
X = TFIDFvectorizer.fit_transform(dataset['cleaned_text'])
```

Term Frequency-Inverse Document Frequency (TF-IDF) weighs words by their importance in a corpus. Common words receive lower weights, while rare words are assigned higher values, capturing their significance in the document.

N-Gram

```
from sklearn.feature_extraction.text import CountVectorizer

# Initialize the CountVectorizer with n-grams (e.g., bi-grams
   and tri-grams)
vectorizer = CountVectorizer(ngram_range=(1, 3))
X = vectorizer.fit_transform(dataset['cleaned_text'])
```

N-Grams capture contiguous sequences of words to preserve some context. For example, bi-grams (2-grams) capture pairs of words like "machine learning," and trigrams (3-grams) capture sequences like "natural language processing."

LIWC (Empath)

```
from empath import Empath

# Initialize Empath
lexicon = Empath()

# Function to get Empath features
def get_empath_features(text):
    analysis = lexicon.analyze(text, normalize=True)  #
       Normalize for proportions
    return analysis

# Generate Empath features for each text
empath_features = dataset['cleaned_text'].apply(
   get_empath_features)

# Convert Empath features to a DataFrame
X = pd.DataFrame(empath_features.tolist())
```

LIWC (Linguistic Inquiry and Word Count) analyzes text by categorizing words into meaningful psychological or social dimensions. Empath is a Python library that replicates this functionality with customizable lexicons. Empath is a Python-based natural language processing tool inspired by LIWC (Linguistic Inquiry and Word Count), designed to analyze text and categorize words into psychologically meaningful dimensions such as emotions, cognitive processes, and social themes. This tool is particularly valuable for extracting semantic features that capture the psychological and emotional aspects of textual data, making it ideal for applications in sentiment analysis, mental health studies, and psycholinguistics. By matching words in the text to predefined categories, Empath assigns a proportional representation to each category, indicating the relative frequency of words that belong to it. In the provided code, an `Empath` object is initialized, which serves as the base for analyzing text against the predefined lexicons. A function named `get_empath_features` is defined to process individual texts from the dataset. This function leverages the `analyze` method of the `Empath` object to compute the distribution of words across various categories. The `normalize=True` parameter ensures that the output represents proportions rather than raw counts, making the results more interpretable and comparable across texts of varying lengths. The function is applied to each piece of cleaned text in the dataset, producing a structured output where each text is represented as a dictionary. Each key in this dictionary corresponds to an Empath category, and the associated value indicates the proportion of words from the text that fall into that category. The list of dictionaries generated through this process is then converted into a DataFrame, where each column represents a category and each row corresponds to a document in the dataset. This transformation allows the

extracted features to be seamlessly integrated into machine learning pipelines, where they can serve as predictors for tasks like sentiment analysis, mental health classification.

Word2Vec

```
from nltk.tokenize import word_tokenize
import gensim

# Tokenize the text data into words
def tokenize_text(text):
    return word_tokenize(text.lower())

dataset['tokens'] = dataset['cleaned_text'].apply(tokenize_text)

# Train a Word2Vec model using the tokenized data
word2vec_model = gensim.models.Word2Vec(sentences=dataset['
   tokens'], vector_size=100, window=5, min_count=1, workers=4)

# Function to average Word2Vec vectors for each document
def get_document_vector(tokens):
    valid_tokens = [word for word in tokens if word in
       word2vec_model.wv]
    if len(valid_tokens) == 0:
        return [0] * word2vec_model.vector_size  # Return zero
           vector for empty documents
    vectors = [word2vec_model.wv[word] for word in valid_tokens]
    return list(np.mean(vectors, axis=0))

# Convert the text data into document vectors
X = dataset['tokens'].apply(get_document_vector)
```

Word2Vec creates dense vector representations of words by training a neural network to predict words in context. These embeddings capture semantic meaning, and document vectors are computed by averaging word embeddings. The Python code implements a Word2Vec-based approach to transform text data into numerical vectors. First, the `tokenize_text` function is defined to tokenize each text in the dataset into lowercase words, creating a new column named `tokens`. Using these tokenized sentences, a Word2Vec model is trained with parameters such as a vector size of 100, a context window of 5, and a minimum word frequency of 1. The `get_document_vector` function computes a document-level vector by averaging the Word2Vec embeddings of valid tokens in each text, ensuring that empty documents return a zero vector. Finally, the document vectors are calculated and stored in `X`, enabling their use as input features for downstream machine learning tasks.

8.2.4 Logistic Regression Model for Classification

Logistic Regression for Mental Health Classification

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix

# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')

# Check if 'cleaned_text' and 'mental_health_issue' columns
   exist
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
   ' not in dataset.columns:
    raise ValueError("The dataset must have 'cleaned_text' and '
       mental_health_issue' columns.")

# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)

# Initialize the CountVectorizer and fit/transform the cleaned
   text
LRvectorizer = CountVectorizer()
X = LRvectorizer.fit_transform(dataset['cleaned_text'])

# Prepare the target variable
y = dataset['mental_health_issue']

# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)

# Initialize the Logistic Regression model
LRmodel = LogisticRegression(multi_class='ovr', max_iter=10000)

# Train the model
LRmodel.fit(X_train, y_train)

# Make predictions on the test set
y_pred = LRmodel.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

Logistic Regression for Mental Health Classification

```
# Print classification report
print("Classification Report:\n", classification_report(y_test,
    y_pred))

# Print confusion matrix
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

The provided code demonstrates how to apply Logistic Regression for mental health classification using a preprocessed dataset. First, the dataset is loaded, and a check is performed to ensure that it contains the necessary columns, specifically `cleaned_text` and `mental_health_issue`. If these columns are missing, an error is raised. The dataset is then cleaned by removing rows that have missing values in the `cleaned_text` column using the `dropna()` function. Next, the `CountVectorizer` is initialized and applied to the `cleaned_text` column to convert the text data into a numerical format suitable for machine learning. This is accomplished by transforming the text into a document-term matrix where each row represents a document (post) and each column represents a unique term (word). The target variable, `mental_health_issue`, is also extracted from the dataset.The dataset is then split into training and testing sets using `train_test_split()`, where 80% of the data is used for training, and 20% is reserved for testing. The `random_state` is set to ensure reproducibility of the results. A Logistic Regression model is initialized with a multi-class strategy (`multi_class='ovr'`) and a high number of iterations (`max_iter=10000`) to allow the model to converge. The model is then trained on the training data using the `fit()` method. After training, predictions are made on the test set using the `predict()` method. The model's performance is evaluated by calculating the accuracy score, which is printed as a percentage. Additionally, a detailed classification report is generated, which includes metrics such as precision, recall, and F1-score for each class. Lastly, a confusion matrix is printed to visualize the model's classification performance and to understand how well the model distinguishes between different mental health issues. This end-to-end pipeline is crucial for applying Logistic Regression to textual data, allowing for effective classification of mental health issues based on Reddit posts.

8.2.5 Naive Bayes for Classification

Naive Bayes for Mental Health Classification

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix

# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')
# Check if 'cleaned_text' and 'mental_health_issue' columns
   exist
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
   ' not in dataset.columns:
    raise ValueError("The dataset must have 'cleaned_text' and '
       mental_health_issue' columns.")

# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)
# Initialize the CountVectorizer and fit/transform the cleaned
   text
NBvectorizer = CountVectorizer()
X = NBvectorizer.fit_transform(dataset['cleaned_text'])

# Prepare the target variable
y = dataset['mental_health_issue']

# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)
# Initialize the Naive Bayes classifier
NBmodel = MultinomialNB()
# Fit the model
NBmodel.fit(X_train, y_train)
# Make predictions
y_pred = NBmodel.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

# Print classification report
print("Classification Report:\n", classification_report(y_test,
   y_pred))
# Print confusion matrix
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

The provided code demonstrates how to apply the Naive Bayes classifier (MultinomialNB) for mental health classification using a preprocessed dataset. First, the dataset is loaded, and a check is performed to ensure that it contains the necessary columns, specifically `cleaned_text` and `mental_health_issue`. If these columns are missing, an error is raised. The dataset is then cleaned by removing rows that have missing values in the `cleaned_text` column using the `dropna()` function. Next, the `CountVectorizer` is initialized and applied to the `cleaned_text` column to convert the text data into a numerical format suitable for machine learning. This is accomplished by transforming the text into a document-term matrix where each row represents a document (post) and each column represents a unique term (word). The target variable, `mental_health_issue`, is also extracted from the dataset. The dataset is then split into training and testing sets using `train_test_split()`, where 80% of the data is used for training, and 20% is reserved for testing. The `random_state` is set to ensure reproducibility of the results. A Naive Bayes model (`MultinomialNB`) is initialized and trained on the training data using the `fit()` method. After training, predictions are made on the test set using the `predict()` method. The model's performance is evaluated by calculating the accuracy score, which is printed as a percentage. Additionally, a detailed classification report is generated, which includes metrics such as precision, recall, and F1-score for each class. Lastly, a confusion matrix is printed to visualize the model's classification performance and to understand how well the model distinguishes between different mental health issues. This end-to-end pipeline is crucial for applying the Naive Bayes algorithm to textual data, allowing for effective classification of mental health issues based on Reddit posts.

8.2.6 Support Vector Machine for Classification

Support Vector Classifier Implementation

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix

# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')
# Check if 'cleaned_text' and 'mental_health_issue' columns
   exist
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
   ' not in dataset.columns:
    raise ValueError("The dataset must have 'cleaned_text' and '
       mental_health_issue' columns.")
```

Support Vector Classifier Implementation

```
# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)
# Initialize the CountVectorizer and fit/transform the cleaned
   text
SVMvectorizer = CountVectorizer()
X = SVMvectorizer.fit_transform(dataset['cleaned_text'])

# Prepare the target variable
y = dataset['mental_health_issue']
# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)

# Initialize the Support Vector Classifier
SVMmodel = SVC(kernel='linear', C=1, random_state=42,
   probability=True)  # You can adjust parameters as needed

# Train the model
SVMmodel.fit(X_train, y_train)
# Make predictions on the test set
y_pred = SVMmodel.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

# Print classification report
print("Classification Report:\n", classification_report(y_test,
   y_pred))
# Print confusion matrix
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

The provided code demonstrates how to apply the Support Vector Classifier (SVC) for mental health classification using a preprocessed dataset. First, the dataset is loaded, and a check is performed to ensure that it contains the necessary columns, specifically `cleaned_text` and `mental_health_issue`. If these columns are missing, an error is raised. The dataset is then cleaned by removing rows that have missing values in the `cleaned_text` column using the `dropna()` function.Next, the `CountVectorizer` is initialized and applied to the `cleaned_text` column to convert the text data into a numerical format suitable for machine learning. This is accomplished by transforming the text into a document-term matrix where each row represents a document (post) and each column represents a unique term (word). The target variable, `mental_health_issue`, is also extracted from the dataset. The dataset is then split into training and testing sets using `train_test_split()`, where 80% of the data

is used for training, and 20% is reserved for testing. The `random_state` is set to ensure reproducibility of the results. A Support Vector Classifier model (`SVC`) is initialized with a linear kernel (`kernel='linear'`), regularization parameter `C=1`, and the probability flag set to `True` to enable probability estimates. The model is then trained on the training data using the `fit()` method. After training, predictions are made on the test set using the `predict()` method. The model's performance is evaluated by calculating the accuracy score, which is printed as a percentage. Additionally, a detailed classification report is generated, which includes metrics such as precision, recall, and F1-score for each class. Lastly, a confusion matrix is printed to visualize the model's classification performance and to understand how well the model distinguishes between different mental health issues. This end-to-end pipeline is crucial for applying the Support Vector Machine (SVM) algorithm to textual data, allowing for effective classification of mental health issues based on Reddit posts.

8.2.7 Random Forest for Classification

Random Forest Classifier Implementation

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix

# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')
# Check if 'cleaned_text' and 'mental_health_issue' columns
   exist
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
   ' not in dataset.columns:
    raise ValueError("The␣dataset␣must␣have␣'cleaned_text'␣and␣'
       mental_health_issue'␣columns.")
# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)
# Initialize the CountVectorizer and fit/transform the cleaned
   text
RFvectorizer = CountVectorizer()
X = RFvectorizer.fit_transform(dataset['cleaned_text'])
# Prepare the target variable
y = dataset['mental_health_issue']
# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)
```

Random Forest Classifier Implementation

```
# Initialize the Random Forest Classifier with added parameters
RFmodel = RandomForestClassifier(n_estimators=3000, max_depth=
   None, min_samples_split=20, min_samples_leaf=1, max_features=
   'sqrt', bootstrap=False, random_state=42)

# Train the model
RFmodel.fit(X_train, y_train)
# Make predictions on the test set
y_pred = RFmodel.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

# Print classification report
print("Classification Report:\n", classification_report(y_test,
   y_pred))
# Print confusion matrix
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

The provided code demonstrates how to apply the Random Forest Classifier for mental health classification using a preprocessed dataset. First, the dataset is loaded, and a check is performed to ensure that it contains the necessary columns, specifically `cleaned_text` and `mental_health_issue`. If these columns are missing, an error is raised. The dataset is then cleaned by removing rows that have missing values in the `cleaned_text` column using the `dropna()` function. Next, the `CountVectorizer` is initialized and applied to the `cleaned_text` column to convert the text data into a numerical format suitable for machine learning. This is accomplished by transforming the text into a document-term matrix where each row represents a document (post) and each column represents a unique term (word). The target variable, `mental_health_issue`, is also extracted from the dataset. The dataset is then split into training and testing sets using `train_test_split()`, where 80% of the data is used for training, and 20% is reserved for testing. The `random_state` is set to ensure reproducibility of the results. A Random Forest Classifier model is initialized with several hyperparameters. Specifically, `n_estimators=3000` defines the number of decision trees in the forest. The `max_depth=None` means the trees are allowed to grow until all leaves are pure or contain fewer than the minimum samples required to split a node. The `min_samples_split=20` and `min_samples_leaf=1` specify the minimum number of samples required to split an internal node and the minimum number of samples required to be at a leaf node, respectively. The `max_features='sqrt'` sets the maximum number of features to consider for the best split at each node to be the square root of the total number of features. `bootstrap=False` disables bootstrapping, meaning the entire dataset is used to build each tree. After initialization,

the model is trained using the `fit()` method on the training data. After training, predictions are made on the test set using the `predict()` method. The model's performance is evaluated by calculating the accuracy score, which is printed as a percentage. Additionally, a detailed classification report is generated, which includes metrics such as precision, recall, and F1-score for each class. Lastly, a confusion matrix is printed to visualize the model's classification performance and to understand how well the model distinguishes between different mental health issues. This end-to-end pipeline is essential for applying the Random Forest algorithm to textual data, allowing for effective classification of mental health issues based on Reddit posts.

8.2.8 XGBoost for Classification

XGBoost Classifier Implementation

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix
import xgboost as xgb

# Load the dataset
data = pd.read_csv('preprocessed_mental_health.csv')
# Separate features and target
X = data['text']
y = data['mental_health_issue']
# Encode target labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)

# Convert text data to numerical data using TF-IDF Vectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(max_features=5000)
X_train = tfidf.fit_transform(X_train)
X_test = tfidf.transform(X_test)

# Define the XGBoost classifier
xgb_clf = xgb.XGBClassifier(objective='multi:softmax', num_class
   =5, eval_metric='mlogloss', use_label_encoder=False)
# Train the model
xgb_clf.fit(X_train, y_train)
# Predict on the test set
y_pred = xgb_clf.predict(X_test)
```

XGBoost Classifier Implementation

```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=
   label_encoder.classes_)

print(f"Accuracy: {accuracy * 100:.2f}%")
print("Classification Report:\n", report)
# Print confusion matrix
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

The provided code demonstrates how to apply XGBoost for mental health classification using a preprocessed dataset. First, the dataset is loaded, and the target variable (`mental_health_issue`) is separated from the feature variable (`text`). The target variable is then encoded using `LabelEncoder()` to convert the categorical labels into numerical values, which are required for machine learning algorithms. Next, the dataset is split into training and testing sets using `train_test_split()`, where 80% of the data is used for training, and 20% is reserved for testing. The `random_state` is set to ensure reproducibility of the results. The text data is then transformed into numerical features using the `TfidfVectorizer()`. This vectorizer converts the raw text into a matrix of TF-IDF (Term Frequency-Inverse Document Frequency) features, which are more suitable for training a machine learning model. The `max_features=5000` parameter limits the number of features to the 5000 most frequent terms in the dataset. The XGBoost classifier is then initialized with several hyperparameters. The `objective='multi:softmax'` indicates that the task is a multi-class classification problem. The `num_class=5` parameter specifies the number of classes for the classification task, and `eval_metric='mlogloss'` is set to use the log loss metric for evaluation. `use_label_encoder=False` disables the automatic label encoding of target labels, as we have already manually encoded the labels. After initialization, the model is trained on the training data using the `fit()` method. After training, predictions are made on the test set using the `predict()` method. The model's performance is evaluated by calculating the accuracy score, which is printed as a percentage. Additionally, a detailed classification report is generated, which includes metrics such as precision, recall, and F1-score for each class. Lastly, a confusion matrix is printed to visualize the model's classification performance and to understand how well the model distinguishes between different mental health issues. This pipeline demonstrates the use of XGBoost in combination with TF-IDF vectorization for text classification, enabling effective classification of mental health issues based on Reddit posts.

8.2.9 K Nearest Neighbours for Classification

k-NN Classifier Implementation for Mental Health Classification

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix

# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')
# Check if 'cleaned_text' and 'mental_health_issue' columns
   exist
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
   ' not in dataset.columns:
    raise ValueError("The␣dataset␣must␣have␣'cleaned_text'␣and␣'
       mental_health_issue'␣columns.")
# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)
# Initialize the CountVectorizer and fit/transform the cleaned
   text
KNNvectorizer = CountVectorizer()
X = KNNvectorizer.fit_transform(dataset['cleaned_text'])
# Prepare the target variable
y = dataset['mental_health_issue']
# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)

# Initialize the k-NN classifier
KNNmodel = KNeighborsClassifier(n_neighbors=5)  # You can adjust
    the number of neighbors
# Fit the model
KNNmodel.fit(X_train, y_train)
# Make predictions
y_pred = KNNmodel.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy:␣{accuracy␣*␣100:.2f}%')

# Print classification report
print("Classification␣Report:\n", classification_report(y_test,
   y_pred))
# Print confusion matrix
print("Confusion␣Matrix:\n", confusion_matrix(y_test, y_pred))
```

The code provided demonstrates how to implement a k-Nearest Neighbors (k-NN) classifier for mental health issue classification based on text data. Initially, the dataset is loaded from a CSV file, and a check is performed to ensure the presence of the necessary columns, namely `cleaned_text` and `mental_health_issue`. If any of these columns are missing, the program raises an error. The next step involves removing rows with missing text data from the `cleaned_text` column using `dropna()`. The `CountVectorizer` from `sklearn` is used to transform the textual data into a matrix of token counts, which is suitable for machine learning. This transformation converts each post (document) into a vector of word occurrences, with each unique word in the dataset being represented by a column. The target variable, `mental_health_issue`, is extracted, and the dataset is split into training and test sets using `train_test_split()`. The training set consists of 80% of the data, and the remaining 20% is used for testing. A k-NN classifier is initialized with 5 neighbors (`n_neighbors=5`), though this number can be adjusted for tuning the model. The classifier is trained on the training data using the `fit()` method. Once the model is trained, it is used to predict mental health issues for the test set, and the accuracy of the predictions is calculated and printed. In addition to the accuracy score, a detailed classification report is generated, which includes metrics like precision, recall, and F1-score for each class. Finally, a confusion matrix is printed to help visualize the model's performance in terms of how well it classifies different mental health issues. This end-to-end pipeline demonstrates how to apply a k-NN classifier to a text classification task, specifically identifying mental health issues based on user-generated content such as social media posts.

8.2.10 Long Short Term Memory based Classification

LSTM Model Implementation

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix,
   classification_report
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import
   pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense,
   Dropout
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
```

LSTM Model Implementation

```
# Load the dataset
data = pd.read_csv('preprocessed_mental_health.csv')
# Separate features and target
X = data['text']
y = data['mental_health_issue']
# Encode target labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
y = to_categorical(y)  # Convert labels to one-hot encoded
   format for multi-class classification
# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)
# Tokenize and pad the text sequences
vocab_size = 10000  # Set a vocabulary size
max_length = 100    # Set a max length for padding

tokenizer = Tokenizer(num_words=vocab_size, oov_token="<OOV>")
tokenizer.fit_on_texts(X_train)
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)
X_train_padded = pad_sequences(X_train_seq, maxlen=max_length,
   padding='post', truncating='post')
X_test_padded = pad_sequences(X_test_seq, maxlen=max_length,
   padding='post', truncating='post')
# Build the LSTM model
model = Sequential([ Embedding(vocab_size, 128, input_length=
   max_length), LSTM(128,eturn_sequences=True), Dropout(0.2),
   LSTM(64), Dropout(0.2), Dense(64,activation='relu'), Dense(y.
   shape[1],activation='softmax')])

model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
# Train the model
history = model.fit(X_train_padded, y_train, epochs=10,
   batch_size=32, validation_data=(X_test_padded, y_test))

# Evaluate the model on test data
test_loss, test_accuracy = model.evaluate(X_test_padded, y_test)
print(f"Test␣Accuracy:␣{test_accuracy␣*␣100:.2f}%")
# Generate predictions and convert back from one-hot encoding
y_pred = model.predict(X_test_padded)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)
```

This code demonstrates how to apply a Long Short-Term Memory (LSTM) neural network for

text classification, specifically for predicting mental health issues based on preprocessed Reddit data. The dataset is loaded and the target variable (`mental_health_issue`) is encoded into a one-hot format for multi-class classification. The dataset is split into training and testing sets. Text data is tokenized and padded to ensure uniform input lengths for the LSTM model. The model consists of embedding layers, LSTM layers with dropout for regularization, and dense layers for classification. The model is trained on the training data, and performance is monitored using validation data. The training and validation accuracy and loss are plotted. The model is evaluated on the test data, and performance metrics such as accuracy and classification report are displayed. A confusion matrix is plotted to visualize model performance.

```
Epoch 1/10
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
465/465 ━━━━━━━━━━━━━━━━━━━━ 135s 280ms/step - accuracy: 0.6046 - loss: 1.1142 - val_accuracy: 0.6841 - val_loss: 0.7390
Epoch 2/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 131s 281ms/step - accuracy: 0.6681 - loss: 0.7686 - val_accuracy: 0.6634 - val_loss: 0.8092
Epoch 3/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 144s 286ms/step - accuracy: 0.6499 - loss: 0.8130 - val_accuracy: 0.6680 - val_loss: 0.7538
Epoch 4/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 139s 280ms/step - accuracy: 0.6368 - loss: 0.8657 - val_accuracy: 0.6605 - val_loss: 0.7481
Epoch 5/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 130s 280ms/step - accuracy: 0.6949 - loss: 0.6574 - val_accuracy: 0.6922 - val_loss: 0.6711
Epoch 6/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 141s 278ms/step - accuracy: 0.7132 - loss: 0.6056 - val_accuracy: 0.6933 - val_loss: 0.6697
Epoch 7/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 147s 289ms/step - accuracy: 0.7304 - loss: 0.5676 - val_accuracy: 0.7081 - val_loss: 0.6606
Epoch 8/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 131s 282ms/step - accuracy: 0.7483 - loss: 0.5433 - val_accuracy: 0.7516 - val_loss: 0.6319
Epoch 9/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 142s 282ms/step - accuracy: 0.8079 - loss: 0.4523 - val_accuracy: 0.7484 - val_loss: 0.6037
Epoch 10/10
465/465 ━━━━━━━━━━━━━━━━━━━━ 142s 282ms/step - accuracy: 0.8564 - loss: 0.3823 - val_accuracy: 0.8355 - val_loss: 0.5368
```

Figure 20: Output for LSTM Epochs

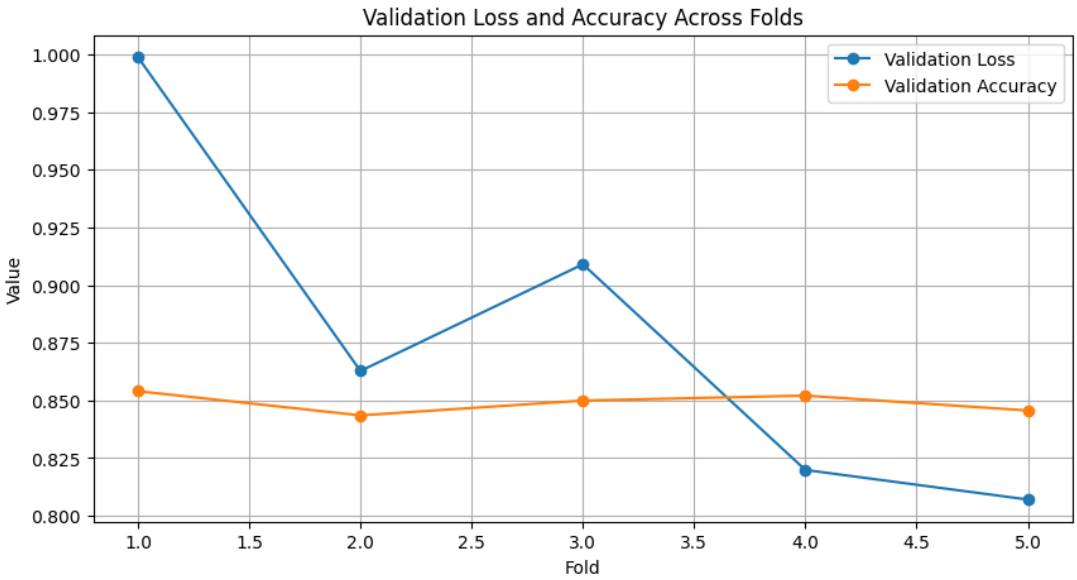


Figure 21: LSTM Validation loss and accuracy

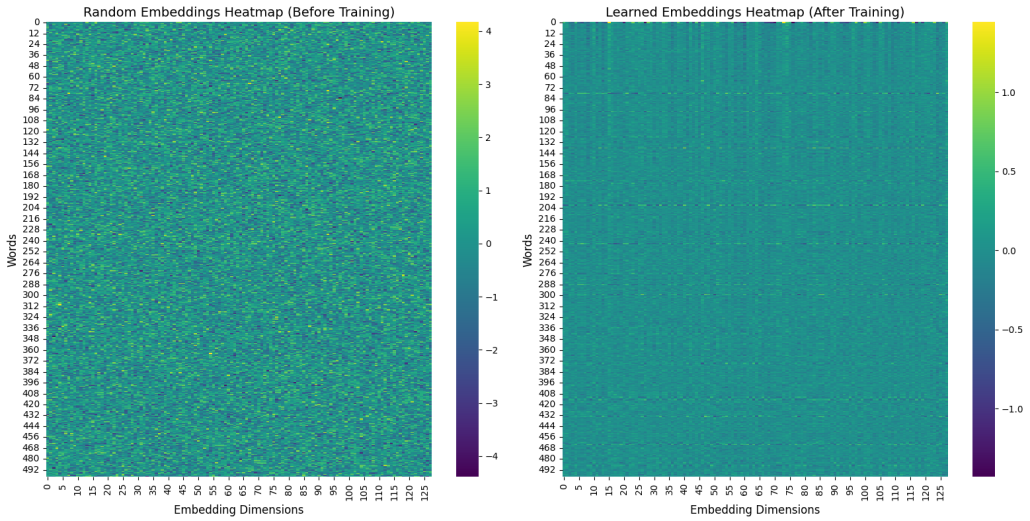


Figure 22: LSTM Random and Learned Embeddings

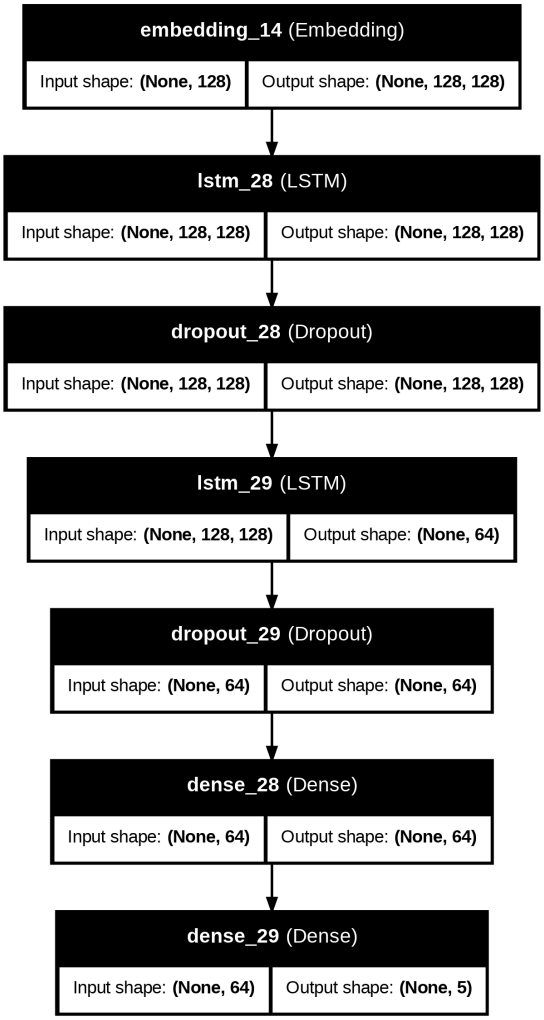


Figure 23: LSTM Model Architecture

8.2.11 Hyperparamter Tuning Using RandomizedSearchCV

Logistic Regression

```
import pandas as pd
from sklearn.model_selection import train_test_split,
   RandomizedSearchCV
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,
   classification_report

# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
   ' not in dataset.columns:
    raise ValueError("The dataset must have 'cleaned_text' and '
       mental_health_issue' columns.")
# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)

# Initialize the CountVectorizer and fit/transform the cleaned
   text
HPLRvectorizer = CountVectorizer()
X = HPLRvectorizer.fit_transform(dataset['cleaned_text'])

# Prepare the target variable
y = dataset['mental_health_issue']
# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)

# Initialize the Logistic Regression model
HPLRmodel = LogisticRegression(max_iter=200)
# Define the hyperparameter grid for Randomized Search
param_distributions = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100], 'penalty': ['l1', 'l2',
       'elasticnet', 'none'], 'solver': ['liblinear', 'saga']}

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=HPLRmodel,
   param_distributions=param_distributions, n_iter=10, scoring='
   accuracy', cv=5, n_jobs=-1, random_state=42)

random_search.fit(X_train, y_train)
print("Best Hyperparameters:", random_search.best_params_)
best_modelLR = random_search.best_estimator_
y_pred = best_modelLR.predict(X_test)
```

Logistic Regression

```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

# Print classification report
print("Classification Report:\n", classification_report(y_test,
    y_pred))
```

The above code demonstrates a machine learning workflow for classifying mental health issues using `LogisticRegression` with hyperparameter tuning. The dataset is loaded using `pandas`, and a check ensures the presence of `'cleaned_text'` and `'mental_health_issue'` columns. Missing values in `'cleaned_text'` are dropped to prevent processing errors. The textual data is transformed into a numerical format using `CountVectorizer`, which converts text into a bag-of-words representation. The dataset is then split into training and testing sets using `train_test_split`. A logistic regression model is initialized, and the hyperparameter grid is defined, including parameters like `C` (inverse of regularization strength), `penalty`, and `solver`. Hyperparameter tuning is performed using `RandomizedSearchCV`, which searches for the best model configuration by sampling parameter combinations. The best model is used to make predictions on the test set, and evaluation metrics such as accuracy and a classification report are printed.

K Nearest Neighbours

```
import pandas as pd
from sklearn.model_selection import train_test_split,
    RandomizedSearchCV
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score,
    classification_report, confusion_matrix

# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')
# Check if 'cleaned_text' and 'mental_health_issue' columns
    exist
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
    ' not in dataset.columns:
     raise ValueError("The dataset must have 'cleaned_text' and '
        mental_health_issue' columns.")
```

K Nearest Neighbours

```
# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)

# Initialize the CountVectorizer and fit/transform the cleaned
   text
HPTKNNvectorizer = CountVectorizer()
X = HPTKNNvectorizer.fit_transform(dataset['cleaned_text'])

# Prepare the target variable
y = dataset['mental_health_issue']

# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)

# Initialize the k-NN classifier
knn = KNeighborsClassifier()

# Define the hyperparameter grid for Randomized Search
param_distributions = {
    'n_neighbors': [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 14,
        15, 16, 17, 18, 19, 20],
        # Different values for number of neighbors
    'metric': ['euclidean', 'manhattan', 'chebyshev', 'minkowski
       '],  # Different distance metrics
    'weights': ['uniform', 'distance']  # Weighing options for
       neighbors
}

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=knn,
   param_distributions=param_distributions,
                                   n_iter=10, scoring='accuracy
                                      ', cv=5, n_jobs=-1,
                                      random_state=42)

# Fit RandomizedSearchCV
random_search.fit(X_train, y_train)

# Best hyperparameters from Random Search
print("Best Hyperparameters:", random_search.best_params_)

# Best model from Random Search
best_knn = random_search.best_estimator_

# Make predictions using the best model
y_pred = best_knn.predict(X_test)
```

K Nearest Neighbours

```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

# Print classification report
print("Classification Report:\n", classification_report(y_test,
    y_pred))

# Print confusion matrix
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

This code implements a k-Nearest Neighbors (k-NN) classifier for classifying mental health issues from text data using hyperparameter tuning via RandomizedSearchCV. First, it loads a preprocessed dataset, ensuring the existence of critical columns, `cleaned_text` (features) and `mental_health_issue` (target). Rows with missing values in the `cleaned_text` column are removed to maintain data integrity. The text is vectorized into numerical format using `CountVectorizer`, converting text into a bag-of-words model. The features (X) and target (y) are split into training (80%) and test (20%) datasets to ensure proper model evaluation. The k-NN model is initialized without predefined parameters, and a hyperparameter grid is defined for tuning. This grid explores various values for the number of neighbors (`n_neighbors`), distance metrics (`metric`), and neighbor weighting options (`weights`). `RandomizedSearchCV` is then used to optimize these parameters by training multiple k-NN models with combinations from the grid, evaluating them with 5-fold cross-validation, and scoring based on accuracy. The best combination of hyperparameters is identified and used to configure the final k-NN model. The model is then tested on unseen data (test set), and its performance is evaluated by calculating the accuracy, a classification report detailing precision, recall, F1-score for each class, and a confusion matrix showing true vs. predicted classifications. This approach balances computational efficiency with robustness, enabling the selection of an optimized k-NN model for mental health classification.

Support Vector Machine

```
import pandas as pd
from sklearn.model_selection import train_test_split,
    RandomizedSearchCV
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
    classification_report
```

Support Vector Machine

```
# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')

# Check if 'cleaned_text' and 'mental_health_issue' columns
   exist
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
   ' not in dataset.columns:
    raise ValueError("The dataset must have 'cleaned_text' and '
       mental_health_issue' columns.")

# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)

# Initialize the CountVectorizer and fit/transform the cleaned
   text
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(dataset['cleaned_text'])

# Prepare the target variable
y = dataset['mental_health_issue']

# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)

# Initialize the SVC model
model = SVC()

# Define the hyperparameter grid for Randomized Search
param_distributions = {
    'C': [0.1, 1, 10, 100],              # Regularization
       parameter
    'kernel': ['linear', 'rbf', 'poly'],  # Kernel types
    'gamma': ['scale', 'auto', 0.1, 1],   # Kernel coefficient
       for 'rbf', 'poly', and 'sigmoid'
}

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=model,
   param_distributions=param_distributions,
                                   n_iter=10, scoring='accuracy
                                      ', cv=5, n_jobs=-1,
                                      random_state=42)

# Fit RandomizedSearchCV
random_search.fit(X_train, y_train)
```

Support Vector Machine

```
# Best hyperparameters from Random Search
print("Best␣Hyperparameters:", random_search.best_params_)

# Best model from Random Search
best_model = random_search.best_estimator_

# Make predictions using the best model
y_pred = best_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy:␣{accuracy␣*␣100:.2f}%')

# Print classification report
print("Classification␣Report:\n", classification_report(y_test,
   y_pred))
```

This code implements a Support Vector Classifier (SVC) for classifying mental health issues from text data using hyperparameter tuning via RandomizedSearchCV. It begins by loading a preprocessed dataset and ensuring that essential columns, `cleaned_text` (features) and `mental_health_issue` (target), are present. Any rows with missing values in the `cleaned_text` column are removed to ensure clean data. The text is then vectorized into a numerical format using `CountVectorizer`, transforming it into a bag-of-words representation. The features (X) and target (y) are split into training (80%) and test (20%) sets for model validation. The SVC model is initialized without predefined hyperparameters, and a hyperparameter grid is defined for tuning. This grid includes different values for the regularization parameter (`C`), kernel types (`kernel`), and the kernel coefficient (`gamma`). `RandomizedSearchCV` is employed to find the best combination of these parameters by evaluating multiple models with various hyperparameter combinations through 5-fold cross-validation, using accuracy as the scoring metric. The best set of hyperparameters is selected to configure the final SVC model. The model is then evaluated on the test set by making predictions, and its performance is measured by calculating the accuracy, as well as generating a classification report that includes precision, recall, and F1-score for each class. This approach ensures the selection of the most optimized SVC model for the classification of mental health issues from text data.

Naive Bayes

```
import pandas as pd
from sklearn.model_selection import train_test_split,
   RandomizedSearchCV
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix
from scipy.stats import uniform

# Load the preprocessed dataset
dataset = pd.read_csv('preprocessed_mental_health.csv')

# Check if 'cleaned_text' and 'mental_health_issue' columns
   exist
if 'cleaned_text' not in dataset.columns or 'mental_health_issue
   ' not in dataset.columns:
    raise ValueError("The dataset must have 'cleaned_text' and '
       mental_health_issue' columns.")

# Remove rows with missing values in 'cleaned_text' column
dataset.dropna(subset=['cleaned_text'], inplace=True)

# Initialize the CountVectorizer and fit/transform the cleaned
   text
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(dataset['cleaned_text'])

# Prepare the target variable
y = dataset['mental_health_issue']

# Split the dataset into Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)

# Initialize the Naive Bayes model
naive_bayes_model = MultinomialNB()

# Define the parameter distribution for Randomized Search
param_distributions = {
    'alpha': uniform(0.001, 5.0)  # Sampling alpha from a
       uniform distribution
}
```

Naive Bayes

```
# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=naive_bayes_model,
   param_distributions=param_distributions,
                                   n_iter=10, scoring='accuracy
                                      ', cv=5, n_jobs=-1,
                                      random_state=42)

# Fit RandomizedSearchCV
random_search.fit(X_train, y_train)

# Best hyperparameters from Random Search
print("Best␣Hyperparameters:", random_search.best_params_)

# Best model from Random Search
best_model = random_search.best_estimator_

# Make predictions using the best model
y_pred = best_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy:␣{accuracy␣*␣100:.2f}%')

# Print classification report
print("Classification␣Report:\n", classification_report(y_test,
   y_pred))

# Print confusion matrix
print("Confusion␣Matrix:\n", confusion_matrix(y_test, y_pred))
```

This code implements a Naive Bayes classifier using the MultinomialNB algorithm to classify mental health issues from text data, with hyperparameter tuning via RandomizedSearchCV. The dataset is first loaded, ensuring the required columns, `cleaned_text` (features) and `mental_health_issue` (target), exist. Rows with missing values in the `cleaned_text` column are removed to maintain data quality. The text is vectorized using `CountVectorizer`, converting it into a bag-of-words representation. The features (X) and target (y) are then split into training (80%) and test (20%) sets to facilitate proper evaluation. The Naive Bayes model is initialized, and a parameter distribution for `alpha` is defined using a uniform distribution, sampling values between 0.001 and 5.0. `RandomizedSearchCV` is utilized to explore different values for `alpha` over 10 iterations, using 5-fold cross-validation and accuracy as the scoring metric. The best hyperparameters are identified and used to configure the final Naive Bayes model. The model is then tested on the unseen data (test set).

8.2.12 Tranformer based model for classification

Importing Libraries

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix,
   ConfusionMatrixDisplay
from tensorflow.keras.layers import MultiHeadAttention, Input,
   Dense, Embedding, GlobalAveragePooling1D, LayerNormalization,
    Layer
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import TextVectorization
import matplotlib.pyplot as plt
from tensorflow.data import Dataset
from tensorflow import convert_to_tensor, string, float32, shape
   , range, reshape
```

The provided Python code imports a set of essential libraries and modules used for data processing, machine learning, and neural network construction. NumPy and pandas are used for numerical computations and data manipulation, while scikit-learn provides tools like label encoding and confusion matrix evaluation. TensorFlow modules enable the construction of advanced deep learning architectures with layers such as multi-head attention and embeddings, alongside utilities for text vectorization and categorical encoding. Matplotlib is utilized for visualizations, and TensorFlow's Dataset API facilitates efficient data handling and pipeline creation for training models.

Load Dataset

```
from google.colab import files

# Upload the file
uploaded = files.upload()
# Load dataset
file_path = 'preprocessed_mental_health.csv'
data = pd.read_csv(file_path)

# Drop rows with missing cleaned_text
data = data.dropna(subset=['cleaned_text'])
# Extract features and labels
texts = data['cleaned_text'].astype(str).values
labels = data['mental_health_issue'].astype(str).values
```

This code snippet uses Google Colab's 'files' module to upload datasets interactively, enabling easy integration with cloud-based environments. It then loads a dataset in CSV format using pandas and preprocesses the data by dropping rows with missing values in the 'cleaned_text' column. The 'cleaned_text' column is used to extract the textual features, and the 'mental_health_issue' column provides the corresponding labels for classification tasks, both stored as NumPy arrays.

Processing Labels and Text Vectorization

```
# Encode the labels
label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(labels)
categorical_labels = to_categorical(encoded_labels)

# Get class names
class_names = label_encoder.classes_
print("Classes:", class_names)
# Define parameters
vocab_size = 25000
sequence_length = 300

# Vectorization layer
vectorize_layer = TextVectorization(max_tokens=vocab_size,
   output_sequence_length=sequence_length)

# Adapt vectorizer to texts
vectorize_layer.adapt(Dataset.from_tensor_slices(texts))
# Vectorize text data
texts_vectorized = vectorize_layer(convert_to_tensor(texts,
   dtype=string))
```

This code snippet encodes textual labels into numerical format using the 'LabelEncoder' from scikit-learn and transforms them into categorical labels suitable for deep learning models. The class names are retrieved and displayed for reference. A 'TextVectorization' layer is defined with specified vocabulary size and sequence length to preprocess text data into numerical format. The vectorization layer is adapted to the text dataset, and the texts are vectorized into fixed-length sequences for model input.

Define Transformer Model

```
class EmbeddingLayer(Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim):
        super(EmbeddingLayer, self).__init__()
        self.word_embedding = Embedding(input_dim=vocab_size,
           output_dim=embed_dim)
        self.position_embedding = Embedding(input_dim=
           sequence_length, output_dim=embed_dim)

    def call(self, tokens):
        sequence_length = shape(tokens)[-1]
        positions = range(start=0, limit=sequence_length, delta
           =1)
        positions_encoding = self.position_embedding(positions)
        words_encoding = self.word_embedding(tokens)
        return positions_encoding + words_encoding

class EncoderLayer(Layer):
    def __init__(self, total_heads, total_dense_units, embed_dim
       ):
        super(EncoderLayer, self).__init__()
        self.multihead = MultiHeadAttention(num_heads=
           total_heads, key_dim=embed_dim)
        self.nnw = Sequential([Dense(total_dense_units,
           activation="relu"), Dense(embed_dim)])
        self.normalize_layer = LayerNormalization()

    def call(self, inputs):
        attn_output = self.multihead(inputs, inputs)
        normalize_attn = self.normalize_layer(inputs +
           attn_output)
        nnw_output = self.nnw(normalize_attn)
        final_output = self.normalize_layer(normalize_attn +
           nnw_output)
        return final_output

# Model parameters
embed_dim = 64
num_heads = 2
total_dense_units = 60

# Define the model
inputs = Input(shape=(sequence_length,))
embedding_layer = EmbeddingLayer(sequence_length, vocab_size,
   embed_dim)
encoder_layer = EncoderLayer(num_heads, total_dense_units,
   embed_dim)
```

Define Transformer Model

```
emb = embedding_layer(inputs)
enc = encoder_layer(emb)
pool = GlobalAveragePooling1D()(enc)
dense = Dense(total_dense_units, activation="relu")(pool)
outputs = Dense(len(class_names), activation="softmax")(dense)

transformer_model = Model(inputs=inputs, outputs=outputs)
transformer_model.compile(optimizer="adamw", loss="
   categorical_crossentropy", metrics=['accuracy'])
transformer_model.summary()
```

This code snippet defines a transformer-based model for text classification. The 'EmbeddingLayer' class creates embeddings for both words and their positions in the sequence, allowing the model to consider positional context. The 'EncoderLayer' class implements an attention mechanism using multi-head attention and dense layers, followed by normalization for improved training stability. The model incorporates these layers to process tokenized input sequences, aggregates information with global pooling, and predicts class probabilities through dense layers with softmax activation. The model is compiled with the AdamW optimizer and categorical crossentropy loss for multi-class classification tasks.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 300)	0
embedding_layer (EmbeddingLayer)	(None, 300, 64)	1,619,200
encoder_layer (EncoderLayer)	(None, 300, 64)	41,148
global_average_pooling1d (GlobalAveragePooling1D)	(None, 64)	0
dense_2 (Dense)	(None, 60)	3,900
dense_3 (Dense)	(None, 5)	305

Total params: 1,664,553 (6.35 MB)
Trainable params: 1,664,553 (6.35 MB)
Non-trainable params: 0 (0.00 B)

Figure 24: Transformer Model Summary

Train Transformer Model

```
# Split data into train and validation
from sklearn.model_selection import train_test_split
import numpy as np # Import numpy

# Convert the TensorFlow tensor to a NumPy array
texts_vectorized_np = texts_vectorized.numpy()

# Now perform the split
X_train, X_val, y_train, y_val = train_test_split(
    texts_vectorized_np, categorical_labels, test_size=0.2,
       random_state=42
)

# Train the model
history = transformer_model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    batch_size=32,
    epochs=5
)
```

This code performs the crucial steps of splitting the dataset into training and validation sets, followed by training the transformer-based model. The `train_test_split` function from the `sklearn.model_selection` module is used to divide the data, ensuring that 20% of the data is set aside for validation while maintaining a random state for reproducibility. Before splitting, the vectorized text data, originally in TensorFlow tensor format, is converted to a NumPy array using the `numpy()` method. This conversion facilitates seamless integration with scikit-learn functions. The training process involves the `fit` method of the transformer model. Here, the training features (`X_train`) and labels (`y_train`) are passed to the model, alongside the validation data (`X_val` and `y_val`) for evaluating the model's performance on unseen data during training. A batch size of 32 ensures that the data is processed in manageable chunks, balancing computational efficiency and learning stability. The training is conducted over 5 epochs, allowing the model to iteratively adjust its weights to minimize the loss function. The resulting `history` object captures the training and validation metrics for each epoch, which can be used for performance analysis and visualization.

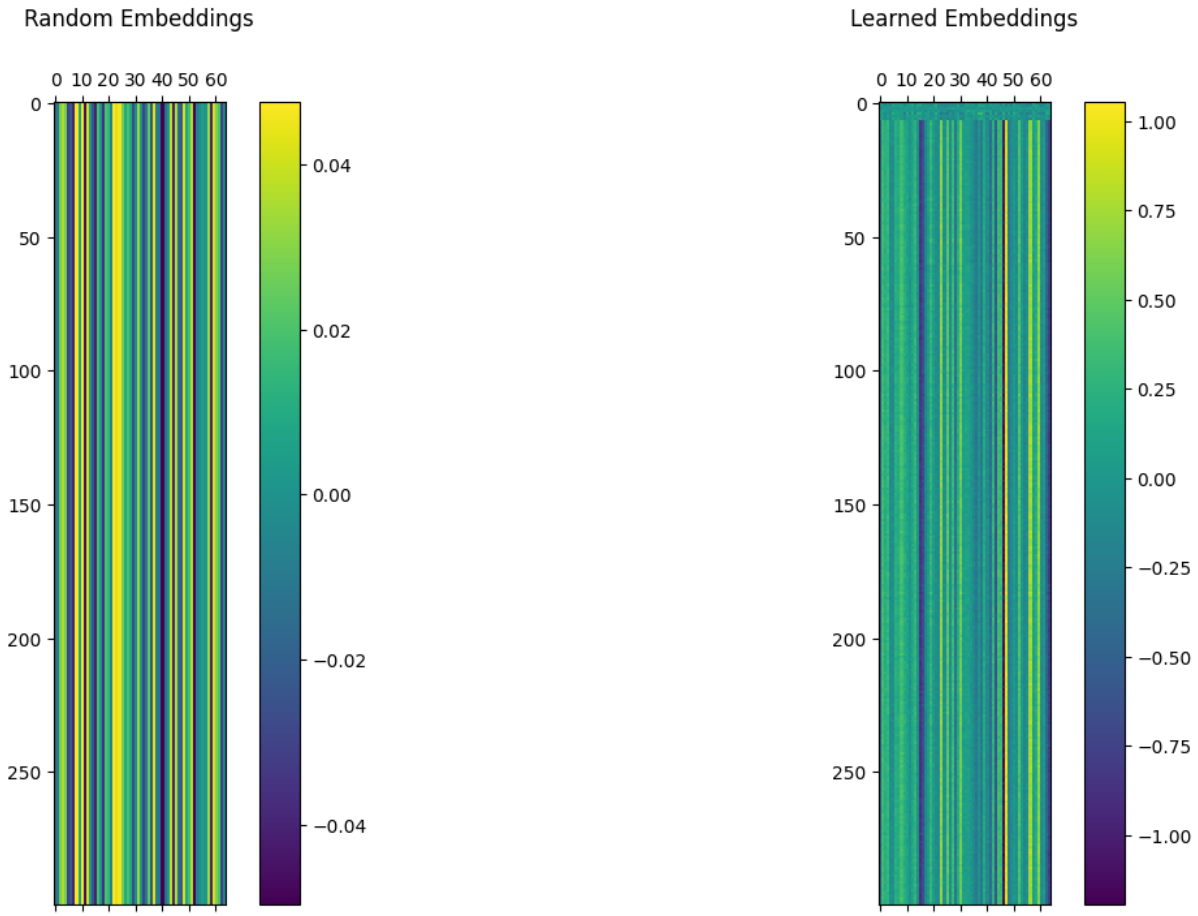


Figure 25: Transformer Model Random and Learned Embeddings

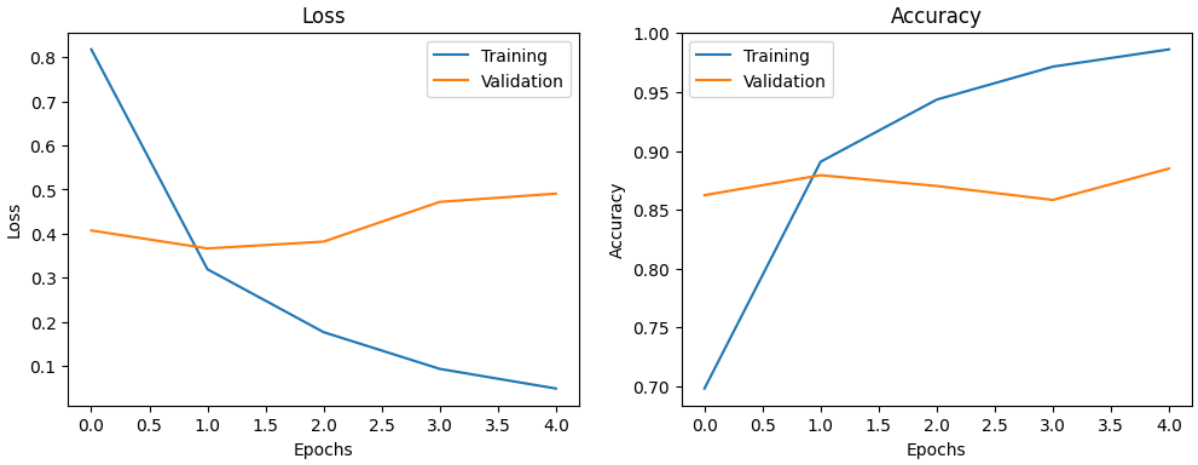


Figure 26: Transformer Epoch, Loss, Accuracy

8.2.13 Ensemble Model 1 (Stacking with Meta-Learner : Logistic Regression)

Base Model : Logistic Regression, XGBoost.

Ensemble Model 1 : Loading and Preprocessing

```
import pickle
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,
   classification_report
import xgboost as xgb
import pandas as pd

# Load the Logistic Regression model and vectorizer
with open('LRmodel.pkl', 'rb') as file:
    lr_model = pickle.load(file)
with open('LRvectorizer.pkl', 'rb') as file:
    lr_vectorizer = pickle.load(file)
# Load the XGBoost model, label encoder, and TF-IDF vectorizer
with open('xgb_model.pkl', 'rb') as file:
    xgb_model = pickle.load(file)
with open('label_encoder.pkl', 'rb') as file:
    label_encoder = pickle.load(file)
with open('tfidf_vectorizer.pkl', 'rb') as file:
    tfidf_vectorizer = pickle.load(file)
```

This Python code is centered around preparing a machine learning workflow by loading pre-trained models and their corresponding preprocessing tools. It begins by importing essential libraries for model evaluation and stacking. The `pickle` library is specifically used to load serialized Python objects, enabling the restoration of previously trained models and utilities saved as `.pkl` files. The first part of the code loads a Logistic Regression model along with its associated vectorizer. The Logistic Regression model is deserialized from the `LRmodel.pkl` file, which represents a trained model capable of predicting outcomes based on vectorized text data. The corresponding vectorizer, stored in `LRvectorizer.pkl`, is also loaded. This vectorizer, likely a `CountVectorizer`, converts raw text into numerical feature vectors, which serve as the input for the Logistic Regression model. By loading these components, the workflow ensures consistency with the preprocessing and modeling steps used during training. In the next section, the code loads an XGBoost model alongside its associated tools. The XGBoost model is deserialized from `xgb_model.pkl` and represents another trained model designed to classify text data. To facilitate this, the code also loads a label encoder from `label_encoder.pkl`. The label encoder is responsible for mapping textual class labels (e.g., "depression" or "normal") to numerical values required by machine learning models.

Furthermore, the `tfidf_vectorizer.pkl` is loaded to preprocess text data using the Term Frequency-Inverse Document Frequency (TF-IDF) approach, which captures the importance of words in a document relative to a corpus. This ensures that the input text is transformed in a manner consistent with the original training setup of the XGBoost model. The design of this code emphasizes efficiency and consistency by leveraging pre-trained components. Instead of retraining models, it restores them from their serialized forms, enabling immediate use for predictions or integration into ensemble learning setups. This approach also ensures that text preprocessing remains identical to the training phase, avoiding discrepancies in input data handling. By combining these tools, the foundation is laid for complex tasks such as making predictions or building advanced models like stacked classifiers.

Ensemble Model 1 : Data Preparation

```
# Load the test dataset
data = pd.read_csv('preprocessed_mental_health.csv')

# Check if 'cleaned_text' column exists
if 'cleaned_text' not in data.columns:
    raise ValueError("The dataset must have a 'cleaned_text'
       column.")

# Remove rows with missing values in 'cleaned_text'
data.dropna(subset=['cleaned_text'], inplace=True)

# Split features and target
X_test = data['cleaned_text']
y_test = data['mental_health_issue']

# Encode target labels
y_test = label_encoder.transform(y_test)

# Transform the text using the respective vectorizers
X_test_lr = lr_vectorizer.transform(X_test)  # Logistic
   Regression vectorizer
X_test_xgb = tfidf_vectorizer.transform(X_test)  # XGBoost
   vectorizer
```

This code snippet focuses on preparing the test dataset for evaluation with pre-trained machine learning models, specifically Logistic Regression and XGBoost. First, the test dataset is loaded from a CSV file named `preprocessed_mental_health.csv` using `pd.read_csv`. The dataset is expected to have been preprocessed earlier, containing features and corresponding labels. The code then checks for the existence of a critical column named `cleaned_text`. If this column is missing, a `ValueError` is raised, ensuring that the dataset meets the re-

quired format for subsequent steps. Next, the code removes rows with missing values in the `cleaned_text` column using `data.dropna`. This step ensures the dataset is clean and free of incomplete rows that could potentially cause errors during text transformation or model evaluation. After this cleaning process, the dataset is split into features (`X_test`) and target labels (`y_test`). The features contain the preprocessed text, while the target labels represent the mental health issues corresponding to each text entry. The target labels in `y_test` are then encoded into numerical format using a preloaded `label_encoder`. This encoding is crucial because machine learning models require numerical representations of categorical labels for classification tasks. By transforming the labels to integers, the encoded labels are consistent with the classes used during the model's training phase. Finally, the text data in `X_test` is vectorized using two pre-trained vectorizers. The `lr_vectorizer`, likely a `CountVectorizer` or similar, is applied to produce feature vectors suitable for the Logistic Regression model. Simultaneously, the `tfidf_vectorizer` is applied to transform the same text data into feature vectors optimized for the XGBoost model. These transformations ensure that the test data aligns with the feature representations learned by the respective models during training. This setup allows for seamless evaluation of the Logistic Regression and XGBoost models on the same dataset, maintaining consistency with their training configurations.

Ensemble Model 1 : Combine Base Model Predictions

```
import numpy as np

# Get predictions from the base models
lr_predictions_proba = lr_model.predict_proba(X_test_lr)
xgb_predictions_proba = xgb_model.predict_proba(X_test_xgb)

# Combine predictions as new features
stacked_features = np.hstack((lr_predictions_proba,
   xgb_predictions_proba))
```

This Python code demonstrates a fundamental step in stacking ensemble learning by combining the prediction probabilities of base models into a single feature set. The process begins with obtaining predictions from two pre-trained models: a Logistic Regression model (`lr_model`) and an XGBoost model (`xgb_model`). These models generate class probability distributions for the test dataset, which are then used as input features for a meta-learner in an ensemble framework. First, the code uses the `predict_proba` method of each model to calculate the probabilities associated with each class for every test instance. For the Logistic Regression model, the feature set (`X_test_lr`) is passed to `lr_model.predict_proba`, producing a 2D array (`lr_predictions_proba`) where each row contains the class probabilities for a corresponding instance. Similarly, the XGBoost model processes its input fea-

tures (X_test_xgb) to generate a comparable 2D array (xgb_predictions_proba) of class probabilities. Next, the code combines these probability arrays to form a new feature set, referred to as stacked_features. This is accomplished using the np.hstack function from the NumPy library, which horizontally stacks the arrays column-wise. The resulting stacked_features array contains concatenated probability distributions from both models for each instance. For example, if there are three classes and 100 instances, the combined feature set will have 100 rows and six columns (three from each model). This stacked feature set serves as input to a meta-learner, enabling it to make final predictions by leveraging the strengths of both base models. By integrating predictions in this manner, the stacking approach aims to improve classification performance by reducing errors associated with individual models. The meta-learner effectively learns how to optimally combine these predictions to produce more accurate outcomes.

Ensemble Model 1 : Train Meta-Learner

```
# Train meta-learner using combined features (optional step if
   not pre-trained)
X_train_meta, y_train_meta = stacked_features, y_test  # Example
    using test data as meta-training data

meta_learner = LogisticRegression(max_iter=5000)
meta_learner.fit(X_train_meta, y_train_meta)

# Save the trained meta-learner
with open('meta_learner.pkl', 'wb') as file:
    pickle.dump(meta_learner, file)
```

This Python code focuses on training a meta-learner in the context of stacking ensemble learning. A meta-learner is a secondary machine learning model trained on the outputs of base models, with the goal of combining their predictions to achieve higher accuracy and robustness. The first step involves defining the training data for the meta-learner. The variable X_train_meta is set to the combined feature set (stacked_features), which contains the concatenated prediction probabilities from base models like Logistic Regression and XGBoost. These features represent the likelihood of each class for every instance, as estimated by the base models. The variable y_train_meta holds the true labels (y_test), which are used as the target variable for training the meta-learner. The code then initializes a Logistic Regression model as the meta-learner. The parameter max_iter=5000 ensures that the optimization process has sufficient iterations to converge, particularly for complex datasets or large feature sets. The fit method is called to train the meta-learner using the stacked features and corresponding labels. During this training process, the meta-learner learns how to optimally

combine the prediction probabilities from the base models to produce more accurate final predictions. After training, the code saves the meta-learner to a file named `meta_learner.pkl` using the `pickle.dump` function. This step serializes the trained model, enabling its reuse for future predictions without needing to retrain it. By saving the meta-learner, the workflow becomes more efficient, as the combined expertise of the base models and the meta-learner can be leveraged in subsequent tasks. This approach exemplifies the flexibility and power of stacking ensemble learning, where a meta-learner synthesizes the strengths of multiple base models to enhance predictive performance. The ability to save the meta-learner ensures that this improved model can be deployed in practical applications with minimal computational overhead.

Ensemble Model 1 : Evaluate Meta-Learner and Ensemble Model

```
from sklearn.metrics import confusion_matrix

# Load the pre-trained meta-learner
with open('meta_learner.pkl', 'rb') as file:
    meta_learner = pickle.load(file)

# Predict using the meta-learner
final_predictions = meta_learner.predict(stacked_features)

# Evaluate the ensemble model
accuracy = accuracy_score(y_test, final_predictions)
report = classification_report(y_test, final_predictions,
   target_names=label_encoder.classes_)

print(f"Accuracy: {accuracy * 100:.2f}%")
print("Classification Report:\n", report)
# Print confusion matrix
print("Confusion Matrix:\n", confusion_matrix(y_test,
   final_predictions))
```

This Python code evaluates the performance of a stacking ensemble model, specifically focusing on the meta-learner. The workflow begins by loading the pre-trained meta-learner from a file named `meta_learner.pkl`. Using the `pickle.load` function, the serialized meta-learner model is deserialized and prepared for making predictions. The next step involves generating predictions with the meta-learner. The model uses the stacked features, which are the combined prediction probabilities from the base models (e.g., Logistic Regression and XGBoost). The `predict` method of the meta-learner outputs the final predicted labels for the test data. To evaluate the ensemble model's effectiveness, the code computes its accuracy using the `accuracy_score` function. Accuracy is expressed as the percentage of correctly classified instances relative to the total instances in the test set. Additionally, the

`classification_report` function is used to generate a detailed performance summary. This report includes precision, recall, F1-score, and support for each mental health class. The `target_names` parameter ensures that the report associates the predicted numerical labels with their corresponding class names, as encoded by the label encoder. The evaluation also involves generating a confusion matrix, which provides an in-depth view of the classification performance. The `confusion_matrix` function compares the true labels (`y_test`) with the predicted labels (`final_predictions`), resulting in a matrix that shows the counts of true positives, true negatives, false positives, and false negatives for each class. Finally, the accuracy, classification report, and confusion matrix are printed to the console. This comprehensive evaluation allows the user to assess the ensemble model's overall performance, identify any weaknesses (e.g., misclassifications for specific classes), and confirm the effectiveness of the meta-learner in combining predictions from base models. This structured approach to evaluation underscores the advantages of stacking ensemble learning, demonstrating how the combined insights from multiple base models and a meta-learner can lead to robust predictions and meaningful performance metrics.

In ensemble learning, combining different models like Logistic Regression and XGBoost often leads to improved results due to their complementary strengths. Logistic Regression, being a linear model, excels in problems where the relationship between features is linear. It is simple, interpretable, and computationally efficient, making it effective for text-based classification tasks when the relationship between the features (such as word counts or TF-IDF scores) and the target variable is linear. However, Logistic Regression has some limitations: it struggles with capturing complex non-linear relationships and can perform poorly when the data is highly imbalanced or when there are interactions between features that the model cannot capture. On the other hand, XGBoost (Extreme Gradient Boosting) is a powerful tree-based model that can capture complex, non-linear relationships and interactions between features, making it particularly effective for handling large, noisy, and high-dimensional datasets like text. It builds an ensemble of decision trees and iteratively improves the model by minimizing errors. However, XGBoost also has its drawbacks. It requires careful tuning of hyperparameters (e.g., learning rate, number of trees, and tree depth) and can be computationally expensive, especially for large datasets. Additionally, while it can model non-linear relationships well, it lacks the interpretability of simpler models like Logistic Regression. By combining Logistic Regression and XGBoost in an ensemble learning setup, the two models can complement each other. Logistic Regression can provide a robust baseline when the relationships between features are linear, while XGBoost can enhance the model's ability to capture more complex patterns. This hybrid approach leverages the strengths of both models—Logistic Regression's simplicity and speed, alongside XGBoost's power in modeling complex interactions.

8.2.14 Ensemble Model 2 (Stacking and Boosting with Meta-Learner : XGBoost)

Base Models : Logistic Regression, Naive Bayes, SVM, LSTM, XGBoost.

Evaluate Meta-Learner and Ensemble Model 2

```
import pickle
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import pad_sequences
from xgboost import XGBClassifier

# Load Logistic Regression model and vectorizer
with open('LRmodel.pkl', 'rb') as file:
    lr_model = pickle.load(file)
with open('LRvectorizer.pkl', 'rb') as file:
    lr_vectorizer = pickle.load(file)
# Load SVM model and vectorizer
with open('SVMmodel.pkl', 'rb') as file:
    svm_model = pickle.load(file)
with open('SVMvectorizer.pkl', 'rb') as file:
    svm_vectorizer = pickle.load(file)
# Load XGBoost model, vectorizer, and label encoder
with open('xgb_model.pkl', 'rb') as file:
    xgb_model = pickle.load(file)
with open('tfidf_vectorizer.pkl', 'rb') as file:
    tfidf_vectorizer = pickle.load(file)
with open('label_encoder.pkl', 'rb') as file:
    label_encoder = pickle.load(file)
# Load LSTM model, tokenizer, and label encoder
lstm_model = load_model('lstm_model.h5')
with open('LSTM_tokenizer.pkl', 'rb') as file:
    lstm_tokenizer = pickle.load(file)
# Load Naive Bayes model and vectorizer
with open('NBmodel.pkl', 'rb') as file:
    nb_model = pickle.load(file)
with open('NBvectorizer.pkl', 'rb') as file:
    nb_vectorizer = pickle.load(file)
# Load the test dataset
data = pd.read_csv('preprocessed_mental_health.csv')

# Check if 'cleaned_text' column exists
if 'cleaned_text' not in data.columns:
    raise ValueError("The␣dataset␣must␣have␣a␣'cleaned_text'␣
       column.")
```

Evaluate Meta-Learner and Ensemble Model 2

```
# Remove rows with missing values in 'cleaned_text'
data.dropna(subset=['cleaned_text'], inplace=True)
# Split features and target
X_test = data['cleaned_text']
y_test = data['mental_health_issue']

# Encode target labels
y_test = label_encoder.transform(y_test)

# Process the text for each model
X_test_lr = lr_vectorizer.transform(X_test)  # Logistic
   Regression vectorizer
X_test_svm = svm_vectorizer.transform(X_test)  # SVM vectorizer
X_test_xgb = tfidf_vectorizer.transform(X_test)  # XGBoost
   vectorizer
X_test_nb = nb_vectorizer.transform(X_test)  # Naive Bayes
   vectorizer
X_test_lstm = lstm_tokenizer.texts_to_sequences(X_test)  # LSTM
   tokenizer

# Pad sequences for LSTM
X_test_lstm = pad_sequences(X_test_lstm, maxlen=100, padding='
   post', truncating='post')

# Get predictions from the base models
lr_predictions_proba = lr_model.predict_proba(X_test_lr)  #
   Logistic Regression probabilities
svm_predictions_proba = svm_model.predict_proba(X_test_svm)  #
   SVM probabilities
xgb_predictions_proba = xgb_model.predict_proba(X_test_xgb)  #
   XGBoost probabilities
nb_predictions_proba = nb_model.predict_proba(X_test_nb)  #
   Naive Bayes probabilities
lstm_predictions_proba = lstm_model.predict(X_test_lstm)  # LSTM
    probabilities

# Stack the predictions of all models to create the feature
   matrix for the meta-learner
stacked_features = np.hstack((
    lr_predictions_proba,
    svm_predictions_proba,
    xgb_predictions_proba,
    nb_predictions_proba,
    lstm_predictions_proba
))
```

Evaluate Meta-Learner and Ensemble Model 2

```
# Train the meta-learner (XGBoost as the meta-learner)
meta_learner_xgb = XGBClassifier(use_label_encoder=False,
   eval_metric='mlogloss')
meta_learner_xgb.fit(stacked_features, y_test)

# Save the trained XGBoost meta-learner
with open('meta_learner_xgb.pkl', 'wb') as file:
    pickle.dump(meta_learner_xgb, file)

# Predict using the XGBoost meta-learner
final_predictions_xgb = meta_learner_xgb.predict(
   stacked_features)

# Evaluate the XGBoost ensemble model
accuracy_xgb = accuracy_score(y_test, final_predictions_xgb)
report_xgb = classification_report(y_test, final_predictions_xgb
   , target_names=label_encoder.classes_)

print(f"XGBoost␣Meta-Learner␣Accuracy:␣{accuracy_xgb␣*␣100:.2f}%
   ")
print("Classification␣Report␣(XGBoost):\n", report_xgb)
print("Confusion␣Matrix␣(XGBoost):\n", confusion_matrix(y_test,
   final_predictions_xgb))
```

This script is designed to load several pre-trained machine learning models, apply them to a dataset of mental health-related text data, combine their predictions using a meta-learner approach, and evaluate the performance of the meta-learner model. The models used in this pipeline include logistic regression, support vector machine (SVM), XGBoost, Naive Bayes, and a deep learning-based Long Short-Term Memory (LSTM) model. The purpose of this setup is to leverage the individual strengths of different models to improve classification accuracy for predicting mental health issues based on text data. First, the script imports various necessary libraries like `pickle`, `numpy`, `pandas`, and machine learning tools from `sklearn` and `xgboost`. These libraries help load, transform, and evaluate models and datasets. It then proceeds to load each pre-trained model and associated preprocessing objects (like vectorizers and tokenizers) from disk using `pickle.load()`. Specifically, models for logistic regression, SVM, Naive Bayes, XGBoost, and LSTM are loaded along with their respective vectorizers or tokenizers. These models are already trained and saved previously, allowing the script to reuse them for making predictions on new data. Next, the script loads the test dataset, which contains preprocessed mental health-related text in a column called `cleaned_text`. It ensures that no rows with missing values are present in this column by dropping such rows. The target variable, `mental_health_issue`, which indicates whether a mental health issue is present, is also

extracted. This target variable is then encoded using a label encoder that was previously saved. Encoding is essential because machine learning models require numerical labels rather than categorical ones. For each model, the script prepares the test data differently. The text data is processed using the corresponding vectorizer or tokenizer to convert the text into numerical features that the models can understand. For models like logistic regression, SVM, Naive Bayes, and XGBoost, the test data is transformed into feature matrices using their respective vectorizers, such as `lr_vectorizer` for logistic regression and `svm_vectorizer` for the SVM model. In contrast, the LSTM model requires tokenized sequences, so the `lstm_tokenizer` is used to convert the text into sequences of integers. Since the LSTM model requires fixed-length input, the sequences are padded to a maximum length of 100 tokens using `pad_sequences`.

After transforming the data, the script proceeds to generate predictions from each model. The predictions for each model are given in the form of probabilities (instead of class labels), using methods like `predict_proba()`. These probabilities represent the model's confidence in each class for a given input. For the LSTM model, the prediction probabilities are generated using `model.predict()`, which outputs the probability distribution across possible classes. With these prediction probabilities in hand, the script constructs a feature matrix for the meta-learner, which is an ensemble learning method that combines the predictions of the base models. This meta-learner is an XGBoost classifier, which is a powerful machine learning algorithm known for its efficiency and predictive accuracy. The probabilities predicted by each base model are stacked together using `np.hstack()`, creating a combined feature vector for each sample in the dataset. These stacked features are then used to train the XGBoost meta-learner model. The trained meta-learner, now stored in `meta_learner_xgb`, is used to make final predictions. These predictions represent the ensemble decision, which ideally benefits from the strengths of all the base models. The script then evaluates the performance of the meta-learner using accuracy, classification report, and confusion matrix. Accuracy gives an overall percentage of correctly classified instances, while the classification report provides detailed metrics like precision, recall, and F1-score for each class. The confusion matrix helps visualize the performance of the classifier, showing how many samples were correctly or incorrectly classified for each class. Finally, the trained meta-learner is saved to disk using `pickle.dump()` for future use, so it can be reused to make predictions on new data without retraining. This entire process ensures that the ensemble model, which combines the strengths of different classifiers, is capable of making robust predictions on the mental health issue dataset.

8.2.15 Ensemble Model 3 (Stacking with Meta Learner : Random Forest)

Base Model : Logistic Regression, Naive Bayes, SVM, LSTM, XGBoost

Evaluate Meta-Learner and Ensemble Model 3

```
import pickle
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import pad_sequences

# Load models and vectorizers
with open('LRmodel.pkl', 'rb') as file:
    lr_model = pickle.load(file)
with open('LRvectorizer.pkl', 'rb') as file:
    lr_vectorizer = pickle.load(file)
with open('SVMmodel.pkl', 'rb') as file:
    svm_model = pickle.load(file)
with open('SVMvectorizer.pkl', 'rb') as file:
    svm_vectorizer = pickle.load(file)
with open('XGBmodel.pkl', 'rb') as file:
    xgb_model = pickle.load(file)
with open('XGBvectorizer.pkl', 'rb') as file:
    tfidf_vectorizer = pickle.load(file)
with open('XGBlabel_encoder.pkl', 'rb') as file:
    label_encoder = pickle.load(file)
lstm_model = load_model('LSTMmodel.h5')
with open('LSTMtokenizer.pkl', 'rb') as file:
    lstm_tokenizer = pickle.load(file)
with open('NBmodel.pkl', 'rb') as file:
    nb_model = pickle.load(file)
with open('NBvectorizer.pkl', 'rb') as file:
    nb_vectorizer = pickle.load(file)

# Load the test dataset
data = pd.read_csv('preprocessed_mental_health.csv')
data.dropna(subset=['cleaned_text'], inplace=True)
X_test = data['cleaned_text']
y_test = data['mental_health_issue']
y_test = label_encoder.transform(y_test)

# Preprocess the text
X_test_lr = lr_vectorizer.transform(X_test)
X_test_svm = svm_vectorizer.transform(X_test)
```

Evaluate Meta-Learner and Ensemble Model 3

```
X_test_nb = nb_vectorizer.transform(X_test)
X_test_xgb = tfidf_vectorizer.transform(X_test)
X_test_lstm = lstm_tokenizer.texts_to_sequences(X_test)
X_test_lstm = pad_sequences(X_test_lstm, maxlen=100, padding='
   post', truncating='post')

# Get individual model probabilities
lr_predictions_proba = lr_model.predict_proba(X_test_lr)
svm_predictions_proba = svm_model.predict_proba(X_test_svm)
nb_predictions_proba = nb_model.predict_proba(X_test_nb)
xgb_predictions_proba = xgb_model.predict_proba(X_test_xgb)
lstm_predictions_proba = lstm_model.predict(X_test_lstm)

# Stack the predictions to create the feature matrix for the
   meta-learner
stacked_features = np.hstack((
    lr_predictions_proba, svm_predictions_proba,
       nb_predictions_proba, xgb_predictions_proba,
       lstm_predictions_proba
))

# Train Random Forest as the meta-learner
meta_learner_rf = RandomForestClassifier(
    max_depth=None, min_samples_split=20, min_samples_leaf=1,
       max_features='sqrt', bootstrap=False, random_state=42
)
meta_learner_rf.fit(stacked_features, y_test)

# Save the trained Random Forest meta-learner
with open('meta_learner_rf.pkl', 'wb') as file:
    pickle.dump(meta_learner_rf, file)

# Predict using the Random Forest meta-learner
final_predictions_rf = meta_learner_rf.predict(stacked_features)

# Evaluate the Random Forest ensemble model
accuracy_rf = accuracy_score(y_test, final_predictions_rf)
report_rf = classification_report(y_test, final_predictions_rf,
   target_names=label_encoder.classes_)

print(f"Random␣Forest␣Meta-Learner␣Accuracy:␣{accuracy_rf␣*␣
   100:.2f}%")
print("Classification␣Report␣(Random␣Forest):\n", report_rf)
print("Confusion␣Matrix␣(Random␣Forest):\n", confusion_matrix(
   y_test, final_predictions_rf))
```

In this section, we evaluate the performance of the Random Forest meta-learner model in the ensemble. The code loads various pre-trained models and their respective vectorizers, processes the test data, and then stacks the predictions from each model to form a feature matrix. The Random Forest classifier is then trained as a meta-learner using this stacked feature matrix. After training, the model's accuracy is calculated, and the classification report and confusion matrix are displayed to assess its performance. This ensemble approach combines the strengths of individual models, improving the overall classification accuracy for mental health issue prediction.

8.2.16 Ensemble Model 4 (Bagging)

Base Model : Logistic Regression, Naive Bayes, SVM, LSTM, XGBoost

Evaluate Bagging Meta-Learner and Ensemble Model 4

```
import pickle
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix
from sklearn.ensemble import BaggingClassifier
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import pad_sequences

# Load models and vectorizers
with open('LRmodel.pkl', 'rb') as file:
    lr_model = pickle.load(file)
with open('LRvectorizer.pkl', 'rb') as file:
    lr_vectorizer = pickle.load(file)
with open('SVMmodel.pkl', 'rb') as file:
    svm_model = pickle.load(file)
with open('SVMvectorizer.pkl', 'rb') as file:
    svm_vectorizer = pickle.load(file)
with open('xgb_model.pkl', 'rb') as file:
    xgb_model = pickle.load(file)
with open('tfidf_vectorizer.pkl', 'rb') as file:
    tfidf_vectorizer = pickle.load(file)
with open('label_encoder.pkl', 'rb') as file:
    label_encoder = pickle.load(file)
lstm_model = load_model('lstm_model.h5')
with open('LSTM_tokenizer.pkl', 'rb') as file:
    lstm_tokenizer = pickle.load(file)
with open('NBmodel.pkl', 'rb') as file:
    nb_model = pickle.load(file)
with open('NBvectorizer.pkl', 'rb') as file:
    nb_vectorizer = pickle.load(file)
```

Evaluate Bagging Meta-Learner and Ensemble Model 4

```
# Load the test dataset
data = pd.read_csv('preprocessed_mental_health.csv')
data.dropna(subset=['cleaned_text'], inplace=True)
X_test = data['cleaned_text']
y_test = data['mental_health_issue']
y_test = label_encoder.transform(y_test)

# Preprocess the text
X_test_lr = lr_vectorizer.transform(X_test)
X_test_svm = svm_vectorizer.transform(X_test)
X_test_nb = nb_vectorizer.transform(X_test)
X_test_xgb = tfidf_vectorizer.transform(X_test)
X_test_lstm = lstm_tokenizer.texts_to_sequences(X_test)
X_test_lstm = pad_sequences(X_test_lstm, maxlen=100, padding='
   post', truncating='post')

# Get individual model probabilities
lr_predictions_proba = lr_model.predict_proba(X_test_lr)
svm_predictions_proba = svm_model.predict_proba(X_test_svm)
nb_predictions_proba = nb_model.predict_proba(X_test_nb)
xgb_predictions_proba = xgb_model.predict_proba(X_test_xgb)
lstm_predictions_proba = lstm_model.predict(X_test_lstm)

# Stack the predictions to create the feature matrix for the
   meta-learner
stacked_features = np.hstack((
    lr_predictions_proba, svm_predictions_proba,
       nb_predictions_proba, xgb_predictions_proba,
       lstm_predictions_proba
))

# Train BaggingClassifier as the ensemble model
meta_learner_bagging = BaggingClassifier(
    n_estimators=50,            # Number of base models
    max_samples=0.8,            # Fraction of the dataset for
       each model
    max_features=0.8,           # Fraction of features for each
       model
    bootstrap=True,             # Whether to bootstrap samples
    random_state=42             # For reproducibility
)
meta_learner_bagging.fit(stacked_features, y_test)

# Save the trained Bagging meta-learner
with open('meta_learner_bagging.pkl', 'wb') as file:
    pickle.dump(meta_learner_bagging, file)
```

Evaluate Meta-Learner and Ensemble Model 4

```
# Predict using the Bagging meta-learner
final_predictions_bagging = meta_learner_bagging.predict(
    stacked_features)

# Evaluate the Bagging ensemble model
accuracy_bagging = accuracy_score(y_test,
    final_predictions_bagging)
report_bagging = classification_report(y_test,
    final_predictions_bagging, target_names=label_encoder.
    classes_)

print(f"Bagging␣Meta-Learner␣Accuracy:␣{accuracy_bagging␣*␣
    100:.2f}%")
print("Classification␣Report␣(Bagging):\n", report_bagging)
print("Confusion␣Matrix␣(Bagging):\n", confusion_matrix(y_test,
    final_predictions_bagging))
```

In this section, the performance of the Bagging ensemble model is evaluated. The process begins by loading pre-trained models and their vectorizers, along with the test dataset. After preprocessing the text data for each model, the individual model probabilities are obtained. These predictions are then stacked to form a feature matrix that is fed into the BaggingClassifier, which is trained as a meta-learner. The accuracy, classification report, and confusion matrix are computed to assess the performance of the Bagging meta-learner on the test data.

8.2.17 Ensemble Model 5 (Blending with Meta-Learner : Random Forest)

Base Model : Logistic Regression, Naive Bayes, SVM, LSTM, XGBoost

Blending with Random Forest as Meta-Learner

```
import pickle
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.metrics import accuracy_score,
    classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import pad_sequences
from tensorflow.keras.utils import to_categorical
```

Blending with Random Forest as Meta-Learner

```
# Load models and vectorizers
with open('LRmodel.pkl', 'rb') as file:
    lr_model = pickle.load(file)
with open('LRvectorizer.pkl', 'rb') as file:
    lr_vectorizer = pickle.load(file)
with open('SVMmodel.pkl', 'rb') as file:
    svm_model = pickle.load(file)
with open('SVMvectorizer.pkl', 'rb') as file:
    svm_vectorizer = pickle.load(file)
with open('xgb_model.pkl', 'rb') as file:
    xgb_model = pickle.load(file)
with open('tfidf_vectorizer.pkl', 'rb') as file:
    tfidf_vectorizer = pickle.load(file)
with open('label_encoder.pkl', 'rb') as file:
    label_encoder = pickle.load(file)
lstm_model = load_model('lstm_model.h5')
with open('LSTM_tokenizer.pkl', 'rb') as file:
    lstm_tokenizer = pickle.load(file)
with open('NBmodel.pkl', 'rb') as file:
    nb_model = pickle.load(file)
with open('NBvectorizer.pkl', 'rb') as file:
    nb_vectorizer = pickle.load(file)

# Load the test dataset
data = pd.read_csv('preprocessed_mental_health.csv')
data.dropna(subset=['cleaned_text'], inplace=True)
X = data['cleaned_text']
y = data['mental_health_issue']
y = label_encoder.transform(y)

# Split data into training and holdout sets (for blending)
X_train, X_holdout, y_train, y_holdout = train_test_split(X, y,
   test_size=0.2, random_state=42)
# One-hot encode the labels
y_train_one_hot = to_categorical(y_train, num_classes=len(
   label_encoder.classes_))
y_holdout_one_hot = to_categorical(y_holdout, num_classes=len(
   label_encoder.classes_))

# Preprocess the text for the training set
X_train_lr = lr_vectorizer.transform(X_train)
X_train_svm = svm_vectorizer.transform(X_train)
X_train_nb = nb_vectorizer.transform(X_train)
X_train_xgb = tfidf_vectorizer.transform(X_train)
X_train_lstm = lstm_tokenizer.texts_to_sequences(X_train)
X_train_lstm = pad_sequences(X_train_lstm, maxlen=100, padding='
   post', truncating='post')
```

Blending with Random Forest as Meta-Learner

```
# Train base models
lr_model.fit(X_train_lr, y_train)
svm_model.fit(X_train_svm, y_train)
nb_model.fit(X_train_nb, y_train)
xgb_model.fit(X_train_xgb, y_train)

# Recompile the LSTM model to reset the optimizer
lstm_model.compile(optimizer='adam', loss='
   categorical_crossentropy', metrics=['accuracy'])

# Train the LSTM model with one-hot encoded labels
lstm_model.fit(X_train_lstm, y_train_one_hot, epochs=3,
   batch_size=32)  # Adjust epochs and batch size as needed

# Preprocess the text for the holdout set
X_holdout_lr = lr_vectorizer.transform(X_holdout)
X_holdout_svm = svm_vectorizer.transform(X_holdout)
X_holdout_nb = nb_vectorizer.transform(X_holdout)
X_holdout_xgb = tfidf_vectorizer.transform(X_holdout)
X_holdout_lstm = lstm_tokenizer.texts_to_sequences(X_holdout)
X_holdout_lstm = pad_sequences(X_holdout_lstm, maxlen=100,
   padding='post', truncating='post')

# Get predictions from base models on the holdout set
lr_predictions_proba = lr_model.predict_proba(X_holdout_lr)
svm_predictions_proba = svm_model.predict_proba(X_holdout_svm)
nb_predictions_proba = nb_model.predict_proba(X_holdout_nb)
xgb_predictions_proba = xgb_model.predict_proba(X_holdout_xgb)
lstm_predictions_proba = lstm_model.predict(X_holdout_lstm)

# Stack the predictions to create the feature matrix for the
   meta-learner (blending)
stacked_features = np.hstack((
    lr_predictions_proba, svm_predictions_proba,
       nb_predictions_proba, xgb_predictions_proba,
       lstm_predictions_proba
))

# Train Random Forest as the meta-learner
meta_learner_rf = RandomForestClassifier(
    max_depth=None, min_samples_split=20, min_samples_leaf=1,
       max_features='sqrt', bootstrap=False, random_state=42
)

# Fit the meta-learner using the predictions from the base
   models (holdout set)
meta_learner_rf.fit(stacked_features, y_holdout)
```

Blending with Random Forest as Meta-Learner

```
# Save the trained Random Forest meta-learner
with open('meta_learner_rf_blending.pkl', 'wb') as file:
    pickle.dump(meta_learner_rf, file)

# Predict using the Random Forest meta-learner on the holdout
   set
final_predictions_rf = meta_learner_rf.predict(stacked_features)

# Evaluate the Random Forest blending model
accuracy_rf = accuracy_score(y_holdout, final_predictions_rf)
report_rf = classification_report(y_holdout,
   final_predictions_rf, target_names=label_encoder.classes_)

print(f"Random␣Forest␣Meta-Learner␣Accuracy␣(Blending):␣{
   accuracy_rf␣*␣100:.2f}%")
print("Classification␣Report␣(Random␣Forest):\n", report_rf)
print("Confusion␣Matrix␣(Random␣Forest):\n", confusion_matrix(
   y_holdout, final_predictions_rf))
```

This code implements a blending approach to improve classification performance by combining the predictions of multiple base models through a Random Forest meta-learner. Initially, several machine learning models (Logistic Regression, SVM, Naive Bayes, XGBoost, and LSTM) are trained individually using the provided dataset. The dataset is split into training and holdout sets to facilitate the blending process. The models are then evaluated on the holdout set, and their prediction probabilities are obtained. These prediction probabilities from each model are stacked together to create a feature matrix, which is then used as input to a Random Forest classifier, acting as the meta-learner. The goal is to leverage the diversity in predictions from the base models to make more accurate final predictions. The Random Forest meta-learner is trained using this stacked feature matrix, and its performance is evaluated on the holdout set. The final output includes the accuracy, classification report, and confusion matrix of the Random Forest meta-learner, which represents the blended model's effectiveness. By blending the base models' predictions, the Random Forest meta-learner aims to capture the strengths of each model and improve overall classification performance, reducing the likelihood of errors from any single model.

8.2.18 Ensemble Model 6 (Weighted Voting)

Base Model : Logistic Regression, Naive Bayes, SVM, LSTM, XGBoost

Weighted Voting Ensemble for Classification

```
import pickle
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import pad_sequences

# Load models and vectorizers
with open('LRmodel.pkl', 'rb') as file:
    lr_model = pickle.load(file)
with open('LRvectorizer.pkl', 'rb') as file:
    lr_vectorizer = pickle.load(file)
with open('SVMmodel.pkl', 'rb') as file:
    svm_model = pickle.load(file)
with open('SVMvectorizer.pkl', 'rb') as file:
    svm_vectorizer = pickle.load(file)
with open('xgb_model.pkl', 'rb') as file:
    xgb_model = pickle.load(file)
with open('tfidf_vectorizer.pkl', 'rb') as file:
    tfidf_vectorizer = pickle.load(file)
with open('label_encoder.pkl', 'rb') as file:
    label_encoder = pickle.load(file)
lstm_model = load_model('lstm_model.h5')
with open('LSTM_tokenizer.pkl', 'rb') as file:
    lstm_tokenizer = pickle.load(file)
with open('NBmodel.pkl', 'rb') as file:
    nb_model = pickle.load(file)
with open('NBvectorizer.pkl', 'rb') as file:
    nb_vectorizer = pickle.load(file)

# Load the test dataset
data = pd.read_csv('preprocessed_mental_health.csv')
data.dropna(subset=['cleaned_text'], inplace=True)
X_test = data['cleaned_text']
y_test = data['mental_health_issue']
y_test = label_encoder.transform(y_test)

# Preprocess the text
X_test_lr = lr_vectorizer.transform(X_test)
X_test_svm = svm_vectorizer.transform(X_test)
```

Weighted Voting Ensemble for Classification

```
X_test_nb = nb_vectorizer.transform(X_test)
X_test_xgb = tfidf_vectorizer.transform(X_test)
X_test_lstm = lstm_tokenizer.texts_to_sequences(X_test)
X_test_lstm = pad_sequences(X_test_lstm, maxlen=100, padding='
   post', truncating='post')

# Get individual model probabilities
lr_predictions_proba = lr_model.predict_proba(X_test_lr)
svm_predictions_proba = svm_model.predict_proba(X_test_svm)
nb_predictions_proba = nb_model.predict_proba(X_test_nb)
xgb_predictions_proba = xgb_model.predict_proba(X_test_xgb)
lstm_predictions_proba = lstm_model.predict(X_test_lstm)

# Define weights for models (based on prior performance or
   experimentation)
weights = {
    'lr': 0.25,  # Logistic Regression
    'svm': 0.2,  # SVM
    'nb': 0.15,  # Naive Bayes
    'xgb': 0.25, # XGBoost
    'lstm': 0.15 # LSTM
}

# Weighted sum of probabilities
final_predictions_proba = (
    weights['lr'] * lr_predictions_proba + weights['svm'] *
       svm_predictions_proba + weights['nb'] *
       nb_predictions_proba + weights['xgb'] *
       xgb_predictions_proba + weights['lstm'] *
       lstm_predictions_proba
)

# Final predictions based on maximum weighted probability
final_predictions = np.argmax(final_predictions_proba, axis=1)

# Evaluate weighted voting ensemble
accuracy = accuracy_score(y_test, final_predictions)
report = classification_report(y_test, final_predictions,
   target_names=label_encoder.classes_)

print(f"Weighted␣Voting␣Ensemble␣Accuracy:␣{accuracy␣*␣100:.2f}%
   ")
print("Classification␣Report␣(Weighted␣Voting):\n", report)
print("Confusion␣Matrix:\n", confusion_matrix(y_test,
   final_predictions))
```

This code demonstrates the use of a weighted voting ensemble to classify mental health issues based on text data. The main goal of this approach is to combine the outputs from multiple base models—Logistic Regression, SVM, Naive Bayes, XGBoost, and LSTM—into a single, stronger prediction. Each of these models provides predicted probabilities for each class, and these probabilities are then combined using a weighted sum. The weights are assigned based on the individual performance of each model, which can be adjusted based on prior experimentation or evaluation. The ensemble's final prediction is made by taking the class with the highest probability after the weighted sum. This is a form of soft voting where the model probabilities, rather than just the final class labels, are combined. This method allows for more flexibility and the potential for better overall performance compared to simply choosing the most frequent class among the base models (hard voting). The text input is preprocessed for each model using appropriate vectorizers or tokenizers. Logistic Regression, SVM, Naive Bayes, and XGBoost models use TF-IDF or other feature extraction techniques, while the LSTM model processes the text sequences directly. After obtaining predictions from all models, the weighted sum of their probabilities is computed. The final class predictions are derived by selecting the class with the highest weighted probability. The ensemble model's performance is evaluated on the test set using accuracy, classification report, and confusion matrix. This method takes advantage of the strengths of each individual model while mitigating their weaknesses, aiming for better generalization and accuracy in classifying mental health issues from text.

8.2.19 Ensemble Model 7 (Stacking and using Transformer)

Ensemble Model 7

```
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from tensorflow.keras.models import load_model, Model
from tensorflow.keras.utils import pad_sequences,
   custom_object_scope
from tensorflow.keras.layers import MultiHeadAttention, Input,
   Dense, Embedding, GlobalAveragePooling1D, LayerNormalization,
    Layer
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split,
   cross_val_score
from tensorflow.keras.optimizers import Adam
```

Ensemble Model 7

```
# load all pickle files
# Load the saved transformer model with custom objects
custom_objects = {"EmbeddingLayer": EmbeddingLayer,"EncoderLayer
    ": EncoderLayer}
transformer_model = load_model('Ttransformer_model.h5',
    custom_objects=custom_objects)
# Preprocess the text for the transformer model
# Get probabilities from the transformer model
transformer_predictions_proba = transformer_model.predict(
    X_test_transformer)
# Stack the predictions to create the feature matrix for the
    meta-learner
```

This code integrates a transformer-based model into an existing ensemble learning approach, enhancing its predictive capabilities. Initially, the required libraries are imported, including TensorFlow, NumPy, pandas, and scikit-learn modules for machine learning operations. The pre-trained transformer model is loaded using the `load_model` function, ensuring compatibility by passing a dictionary of custom objects, which includes the `EmbeddingLayer` and `EncoderLayer`. The input test data (`X_test`) is preprocessed using the vectorization layer (`t_vectorize_layer`) to convert raw text into a numerical representation suitable for the transformer model. The predictions from the transformer model are then obtained as class probabilities, using the `predict` method. Next, these predictions are stacked with probabilities from other machine learning models, such as logistic regression, support vector machines, Naive Bayes, XGBoost, and LSTM, to create a comprehensive feature matrix (`stacked_features`). This matrix serves as input for the meta-learner in the ensemble model. By keeping the existing ensemble model (referred to as *Ensemble Model 3*) unchanged and integrating the transformer model's predictions, a new ensemble model is constructed. This modification results in improved accuracy, demonstrating the powerful generalization capabilities of transformer-based models when combined with traditional and deep learning methods.

The ensemble model for web application consists of several base models, each utilizing different techniques for feature extraction and training. Logistic Regression is implemented using the Bag of Words model for vectorization. Naive Bayes is optimized through hyperparameter tuning with Random Search, also utilizing the Bag of Words model for feature extraction. The Long Short-Term Memory (LSTM) model processes the text sequences directly, while XGBoost uses the Term Frequency-Inverse Document Frequency (TF-IDF) method for feature extraction. Support Vector Machine (SVM) is implemented using the Bag of Words model as well. These base models along with transformer are then combined through a Random Forest meta-learner.

8.2.20 Hierarchical Ensemble Model

Approach 1 : One Ensemble for One Subset → Final Ensemble

Hierarchical Ensemble Model 1

```
import pickle
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold,
   cross_val_score
from sklearn.metrics import accuracy_score,
   classification_report, confusion_matrix
from tensorflow.keras.models import load_model, Model
from tensorflow.keras.utils import pad_sequences,
   custom_object_scope
from tensorflow.keras.layers import MultiHeadAttention, Input,
   Dense, Embedding, GlobalAveragePooling1D, LayerNormalization,
    Layer
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split,
   cross_val_score
from tensorflow.keras.optimizers import Adam
import tensorflow as tf

# Define the custom layers
class EmbeddingLayer(Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim,
       **kwargs):
        super(EmbeddingLayer, self).__init__(**kwargs)
        self.word_embedding = Embedding(input_dim=vocab_size,
           output_dim=embed_dim)
        self.position_embedding = Embedding(input_dim=
           sequence_length, output_dim=embed_dim)

    def call(self, tokens):
        sequence_length = tf.shape(tokens)[-1]
        positions = tf.range(start=0, limit=sequence_length,
           delta=1)
        positions_encoding = self.position_embedding(positions)
        words_encoding = self.word_embedding(tokens)
        return positions_encoding + words_encoding
```

Hierarchical Ensemble Model 1

```
class EncoderLayer(Layer):
    def __init__(self, total_heads, total_dense_units, embed_dim
       , **kwargs):
        super(EncoderLayer, self).__init__(**kwargs)
        self.multihead = MultiHeadAttention(num_heads=
           total_heads, key_dim=embed_dim)
        self.nnw = Sequential([Dense(total_dense_units,
           activation="relu"), Dense(embed_dim)])
        self.normalize_layer = LayerNormalization()

    def call(self, inputs):
        attn_output = self.multihead(inputs, inputs)
        normalize_attn = self.normalize_layer(inputs +
           attn_output)
        nnw_output = self.nnw(normalize_attn)
        final_output = self.normalize_layer(normalize_attn +
           nnw_output)
        return final_output

# Helper function to load models and vectorizers
def load_models_and_vectorizers(prefix):
    with open(f'LRmodel_{prefix}.pkl', 'rb') as file:
        lr_model = pickle.load(file)
    with open(f'LRvectorizer_{prefix}.pkl', 'rb') as file:
        lr_vectorizer = pickle.load(file)
    with open(f'NBmodel_{prefix}.pkl', 'rb') as file:
        nb_model = pickle.load(file)
    with open(f'NBvectorizer_{prefix}.pkl', 'rb') as file:
        nb_vectorizer = pickle.load(file)
    with open(f'SVMmodel_{prefix}.pkl', 'rb') as file:
        svm_model = pickle.load(file)
    with open(f'SVMvectorizer_{prefix}.pkl', 'rb') as file:
        svm_vectorizer = pickle.load(file)
    with open(f'xgb_model_{prefix}.pkl', 'rb') as file:
        xgb_model = pickle.load(file)
    with open(f'tfidf_vectorizer_{prefix}.pkl', 'rb') as file:
        tfidf_vectorizer = pickle.load(file)
    with open(f'label_encoder_{prefix}.pkl', 'rb') as file:
        label_encoder = pickle.load(file)
    lstm_model = tf.keras.models.load_model(f'lstm_model_{prefix
       }.h5')
    with open(f'LSTM_tokenizer_{prefix}.pkl', 'rb') as file:
        lstm_tokenizer = pickle.load(file)
    with open(f'Tlabel_encoder_{prefix}.pkl', 'rb') as file:
        t_label_encoder = pickle.load(file)
```

Hierarchical Ensemble Model 1

```
    with open(f'Tvectorize_layer_{prefix}.pkl', 'rb') as file:
    t_vectorize_layer = pickle.load(file)
    transformer_model = tf.keras.models.load_model(f'
       Ttransformer_model_{prefix}.h5', custom_objects={
        "EmbeddingLayer": EmbeddingLayer,
        "EncoderLayer": EncoderLayer
    })

    return (lr_model, lr_vectorizer, nb_model, nb_vectorizer,
       svm_model, svm_vectorizer,
            xgb_model, tfidf_vectorizer, label_encoder,
               lstm_model, lstm_tokenizer,
            t_label_encoder, t_vectorize_layer,
               transformer_model)

# Function to create meta learner for a given dataset
def create_meta_learner(dataset_path, prefix, meta_model_path):
    data = pd.read_csv(dataset_path)
    data.dropna(subset=['cleaned_text'], inplace=True)

    X = data['cleaned_text']
    y = data['mental_health_issue']

    models = load_models_and_vectorizers(prefix)

    lr_model, lr_vectorizer, nb_model, nb_vectorizer, svm_model,
        svm_vectorizer, \
    xgb_model, tfidf_vectorizer, label_encoder, lstm_model,
       lstm_tokenizer, \
    t_label_encoder, t_vectorize_layer, transformer_model =
       models

    y = label_encoder.transform(y)

    # Preprocess text for each model
    X_lr = lr_vectorizer.transform(X)
    X_nb = nb_vectorizer.transform(X)
    X_svm = svm_vectorizer.transform(X)
    X_xgb = tfidf_vectorizer.transform(X)
    X_lstm = lstm_tokenizer.texts_to_sequences(X)
    X_lstm = pad_sequences(X_lstm, maxlen=100, padding='post',
       truncating='post')
    X_transformer = t_vectorize_layer(X)
```

Hierarchical Ensemble Model 1

```
    # Get individual model probabilities
    stacked_features = np.hstack((
        lr_model.predict_proba(X_lr),
        nb_model.predict_proba(X_nb),
        svm_model.predict_proba(X_svm),
        xgb_model.predict_proba(X_xgb),
        lstm_model.predict(X_lstm),
        transformer_model.predict(X_transformer)
    ))

    # Train meta-learner
    X_train, X_test, y_train, y_test = train_test_split(
        stacked_features, y, test_size=0.2, stratify=y,
           random_state=42
    )

    meta_learner = RandomForestClassifier(
        max_depth=None, min_samples_split=20, min_samples_leaf
           =1,
        max_features='sqrt', bootstrap=False, random_state=42
    )
    meta_learner.fit(X_train, y_train)

    # Save meta-learner
    with open(meta_model_path, 'wb') as file:
        pickle.dump(meta_learner, file)

    return stacked_features, y

# Create meta learners for all subsets
meta_features = []
meta_labels = []

for i in range(1, 7):
    dataset_path = f'preprocessed_mental_health_{i}.csv'
    prefix = f'{i}'
    meta_model_path = f'meta_learner_rf_{i}.pkl'

    features, labels = create_meta_learner(dataset_path, prefix,
        meta_model_path)
    meta_features.append(features)
    meta_labels.append(labels)

# Combine meta features and labels for final meta-learner
meta_features = np.vstack(meta_features)
meta_labels = np.concatenate(meta_labels)
```

Hierarchical Ensemble Model 1

```
# Train final meta-learner
final_meta_learner = RandomForestClassifier(
    max_depth=None, min_samples_split=20, min_samples_leaf=1,
    max_features='sqrt', bootstrap=False, random_state=42
)
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = cross_val_score(final_meta_learner, meta_features,
   meta_labels, cv=skf, scoring='accuracy')
final_meta_learner.fit(meta_features, meta_labels)
with open('final_meta_learner_rf.pkl', 'wb') as file:
    pickle.dump(final_meta_learner, file)
print(f"Cross-Validation␣Accuracies:␣{cv_scores}")
print(f"Mean␣Validation␣Accuracy:␣{np.mean(cv_scores)␣*␣100:.2f
   }%")
print(f"Validation␣Accuracy␣Std␣Dev:␣{np.std(cv_scores)␣*␣100:.2
   f}%")
```

The provided code implements a hierarchical ensemble model for classifying mental health issues from text data. The approach involves multiple base models, a meta-learner for each subset of the dataset, and a final meta-learner to combine the results of all subsets. The primary goal is to leverage the strengths of different machine learning and deep learning models for robust predictions. Initially, the necessary libraries for machine learning, deep learning, and data manipulation are imported. Custom TensorFlow/Keras layers, such as `EmbeddingLayer` and `EncoderLayer`, are defined to support the transformer model. The `EmbeddingLayer` combines word embeddings and positional encodings to capture the contextual representation of tokens, while the `EncoderLayer` implements the transformer encoder block using multi-head attention and feedforward neural networks with layer normalization for stability and better training dynamics. The function `load_models_and_vectorizers` is responsible for loading pre-trained machine learning models (e.g., Logistic Regression, Naive Bayes, SVM, XGBoost), their respective vectorizers, and deep learning models (LSTM and transformer). Each model and its vectorizer are uniquely identified by a prefix to support multiple dataset subsets. This modular loading design ensures that the framework can handle various datasets and model configurations dynamically. The `create_meta_learner` function processes an individual subset of the dataset. It begins by loading the data from a CSV file, ensuring no null values exist in the `cleaned_text` column. The `cleaned_text` column represents preprocessed text, while the `mental_health_issue` column contains class labels. The labels are encoded into numerical values using a label encoder to make them compatible with machine learning algorithms. The text data is then transformed into appropriate feature representations using the vectorizers for each machine learning model. For deep learning models, the LSTM uses tok-

enized sequences padded to a fixed length, and the transformer model utilizes a vectorization layer.

The models' predictions are gathered as probabilities, representing the confidence levels of each model for all possible classes. These probabilities are horizontally stacked to create a feature matrix, where each row corresponds to an instance, and the columns represent the combined outputs from all base models. This stacked representation serves as input to a meta-learner, which is trained to make final predictions by identifying patterns in the base models' outputs. A `RandomForestClassifier` is used as the meta-learner, chosen for its robustness and ability to handle diverse input features. The training data is split into training and test sets to evaluate the meta-learner's performance. This process is repeated for all subsets of the dataset, and the resulting meta-learners are saved using the `pickle` library for future use. Once all meta-learners are trained, their outputs (features) and labels are combined across subsets to form a comprehensive dataset. This aggregated dataset is used to train a final meta-learner, which combines the knowledge from all subset-specific meta-learners. The final meta-learner is trained using a stratified K-fold cross-validation approach, ensuring that the training and validation splits preserve the class distribution. This technique provides a robust estimate of the model's performance across various data partitions. The final model is then trained on the entire aggregated dataset, and its accuracy is evaluated and saved for deployment. The hierarchical structure of the model allows the integration of diverse base models, including both traditional machine learning and neural network-based approaches. This ensemble leverages the strengths of each model type, such as the interpretability of machine learning models and the capacity of deep learning models to capture complex patterns in the data. The use of meta-learners ensures that the final predictions are informed by the combined knowledge of all base models, leading to a more accurate and robust classification system for mental health issues.

Approach 2 : One Ensemble for one type of base model → Final Ensemble Model
The below code expands upon the hierarchical ensemble approach by constructing a multi-layered ensemble model. The primary goal is to group models of the same type, such as Logistic Regression, SVM, Naive Bayes, XGBoost, LSTM, and Transformer, from different data subsets into dedicated ensemble models. Each ensemble model is trained using Logistic Regression as the meta-learner. First, the dataset is divided into multiple subsets, each containing preprocessed data for training individual models. For every subset, models of various types are trained, and their vectorizers or tokenizers preprocess the input data. The function create_ensemble_for_model_type then groups models of the same type, collecting predictions across all subsets. These predictions are used to train a Logistic Regression meta-learner for each model type, resulting in six meta-learners, one for each model type. In the second step,

Hierarchical Ensemble Model 2

```
def create_ensemble_for_model_type(data_subsets, model_type):
    all_predictions = []
    all_labels = []
    for i, (dataset_path, prefix) in enumerate(data_subsets):
        data = pd.read_csv(dataset_path)
        data.dropna(subset=['cleaned_text'], inplace=True)
        X = data['cleaned_text']
        y = data['mental_health_issue']
        models = load_models_and_vectorizers(prefix)
        lr_model, lr_vectorizer, nb_model, nb_vectorizer,
           svm_model, svm_vectorizer, \
        xgb_model, tfidf_vectorizer, lstm_model, lstm_tokenizer,
            transformer_model = models

        # Preprocess text and get predictions for the specific
           model type
        if model_type == 'LR':
            X_transformed = lr_vectorizer.transform(X)
            predictions = lr_model.predict_proba(X_transformed)
        elif model_type == 'NB':
            X_transformed = nb_vectorizer.transform(X)
            predictions = nb_model.predict_proba(X_transformed)
        elif model_type == 'SVM':
            X_transformed = svm_vectorizer.transform(X)
            predictions = svm_model.predict_proba(X_transformed)
        elif model_type == 'XGB':
            X_transformed = tfidf_vectorizer.transform(X)
            predictions = xgb_model.predict_proba(X_transformed)
        elif model_type == 'LSTM':
            X_transformed = lstm_tokenizer.texts_to_sequences(X)
            X_transformed = pad_sequences(X_transformed, maxlen
               =100, padding='post', truncating='post')
            predictions = lstm_model.predict(X_transformed)
        elif model_type == 'Transformer':
            # Preprocess the text for the Transformer using the
               TextVectorization layer
            with open(f'Tvectorize_layer_{prefix}.pkl', 'rb') as
                vectorizer_file:
                t_vectorize_layer = pickle.load(vectorizer_file)
            X_transformed = t_vectorize_layer(tf.
               convert_to_tensor(X, dtype=tf.string))
            # Make predictions
            predictions = transformer_model.predict(
               X_transformed)
    all_predictions.append(predictions)
    all_labels.append(y)
```

Hierarchical Ensemble Model 2

```
    # Stack predictions and labels
    stacked_features = np.vstack(all_predictions)
    stacked_labels = np.concatenate(all_labels)
    # Train Logistic Regression meta-learner
    meta_learner = RandomForestClassifier(max_depth=None,
       min_samples_split=20, min_samples_leaf=1, max_features='
       sqrt', bootstrap=False, random_state=42)
    meta_learner.fit(stacked_features, stacked_labels)
    return meta_learner

# Create meta-learners for each model type
data_subsets = [(f'preprocessed_mental_health_{i}.csv', f'{i}')
   for i in range(1, 7)]

lr_meta_learner = create_ensemble_for_model_type(data_subsets, '
   LR')
nb_meta_learner = create_ensemble_for_model_type(data_subsets, '
   NB')
svm_meta_learner = create_ensemble_for_model_type(data_subsets,
   'SVM')
xgb_meta_learner = create_ensemble_for_model_type(data_subsets,
   'XGB')
lstm_meta_learner = create_ensemble_for_model_type(data_subsets,
    'LSTM')
transformer_meta_learner = create_ensemble_for_model_type(
   data_subsets, 'Transformer')

# Create final meta-learner using Random Forest
final_predictions = []
final_labels = []
for data_path, prefix in data_subsets:
data = pd.read_csv(data_path)
data.dropna(subset=['cleaned_text'], inplace=True)
X = data['cleaned_text']
y = data['mental_health_issue']
models = load_models_and_vectorizers(prefix)
lr_model, lr_vectorizer, nb_model, nb_vectorizer, svm_model,
   svm_vectorizer, \
xgb_model, tfidf_vectorizer, lstm_model, lstm_tokenizer,
   transformer_model = models
with open(f'Tvectorize_layer_{prefix}.pkl', 'rb') as
   vectorizer_file:
    t_vectorize_layer = pickle.load(vectorizer_file)
```

Hierarchical Ensemble Model 2

```
# Get predictions from each meta-learner
X_lr = lr_vectorizer.transform(X)
X_nb = nb_vectorizer.transform(X)
X_svm = svm_vectorizer.transform(X)
X_xgb = tfidf_vectorizer.transform(X)
X_lstm = lstm_tokenizer.texts_to_sequences(X)
X_lstm = pad_sequences(X_lstm, maxlen=100, padding='post',
   truncating='post')
# Preprocess the input for the Transformer
X_transformer = t_vectorize_layer(tf.convert_to_tensor(X, dtype=
   tf.string))  # Use the text vectorizer layer
transformer_predictions = transformer_model.predict(
   X_transformer)

features = np.hstack([
    lr_meta_learner.predict_proba(lr_model.predict_proba(X_lr)),
        nb_meta_learner.predict_proba(nb_model.predict_proba(
       X_nb)), svm_meta_learner.predict_proba(svm_model.
       predict_proba(X_svm)), xgb_meta_learner.predict_proba(
       xgb_model.predict_proba(X_xgb)), lstm_meta_learner.
       predict_proba(lstm_model.predict(X_lstm)),
       transformer_meta_learner.predict_proba(
       transformer_predictions)
])

final_predictions.append(features)
final_labels.append(y.values[:features.shape[0]])
final_features = np.vstack(final_predictions)
final_labels = np.concatenate(final_labels)
```

predictions from these meta-learners are combined into a feature set for each dataset subset, capturing the information from all model types. These combined predictions and the labels from all subsets are used to train a Random Forest as the final meta-learner, creating the ultimate ensemble model. This approach improves upon the previous model by providing a hierarchical organization, where first-level meta-learners group models by type, and a second-level meta-learner combines these ensembles into a robust final model. The flexibility of this approach allows fine-tuning at each level, such as optimizing the Logistic Regression for type-specific ensembles and Random Forest for the final ensemble. Additionally, by leveraging the strengths of various model architectures and combining them hierarchically, the model is better equipped to generalize to unseen data.

9 Test Plans, Results and Analysis

Test Case Plan Table

Test Case ID	Description	Expected Result	Actual Result	Status
T01	User provides input in the text area.	Text is accepted for further processing.	Text is accepted for further processing.	✓
T02	Display mental health issues with probabilities for classes.	Probabilities for each mental health class are shown.	Probabilities for each mental health class are shown.	✓
T03	Highlight the mental health issue with the highest probability.	Correct issue with the highest probability is displayed.	Correct issue with the highest probability is displayed.	✓
T04	Accept username input and provide prediction.	Prediction is displayed based on the provided username.	Prediction is displayed based on the provided username.	✓
T05	Translate multiple language inputs to English.	Non-English text is translated correctly to English.	Non-English text is translated correctly to English.	✓
T06	Extract text and detect emotion from image.	Extracted text and detected emotions are displayed accurately.	Extracted text and detected emotions are displayed accurately.	✓
T07	Pass a prompt to the system and retrieve a valid response.	Correct response is generated based on the prompt.	Correct response is generated based on the prompt.	✓
T08	Perform a combined test using multiple inputs.	Predictions for all inputs are displayed correctly.	Predictions for all inputs are displayed correctly.	✓
T09	Analyze uploaded audio files and transcribe them into text.	Audio transcription and analysis results are displayed.	Audio transcription and analysis results are displayed.	✓
T10	Extract frames, analyze emotions, audio from video.	Frame emotions and audio transcription are displayed.	Frame emotions and audio transcription are displayed.	✓
T11	Extract tweets and related media using a Twitter username.	Text and media are extracted and analyzed correctly.	Text and media are extracted and analyzed correctly.	✓
T12	Generate captions for uploaded images or video frames.	Captions are generated for images or frames.	Captions are generated for images or frames.	✓

The metrics used for evaluating the performance of the classification models include Precision, Recall, F1-Score, and Support, along with the Confusion Matrix. These metrics are crucial for assessing how well the models are able to differentiate between various classes, providing insight into their accuracy, ability to capture relevant instances, and the overall balance between precision and recall. By analyzing these metrics, a comprehensive understanding of the model's performance can be obtained, enabling informed decisions for further optimization and tuning.

9.1 Classification Metrics and Confusion Matrix

- **Precision:** Precision is the ratio of correctly predicted positive observations to the total predicted positives. It can be calculated as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

  where TP represents True Positives, and FP represents False Positives.

- **Recall:** Recall is the ratio of correctly predicted positive observations to all observations in the actual class:

$$\text{Recall} = \frac{TP}{TP + FN}$$

  where TP is True Positives, and FN is False Negatives.

- **F1-Score:** F1-Score is the harmonic mean of Precision and Recall, calculated as:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Support:** Support refers to the number of actual occurrences of each class in the dataset:

$$\text{Support} = \text{Number of samples in the true class}$$

- **Confusion Matrix:** A confusion matrix is used to evaluate the performance of a classification model. It is structured as follows:

$$\begin{bmatrix} TP & FP \\ FN & TN \end{bmatrix}$$

  where:

  - TP = True Positives
  - FP = False Positives
  - FN = False Negatives
  - TN = True Negatives

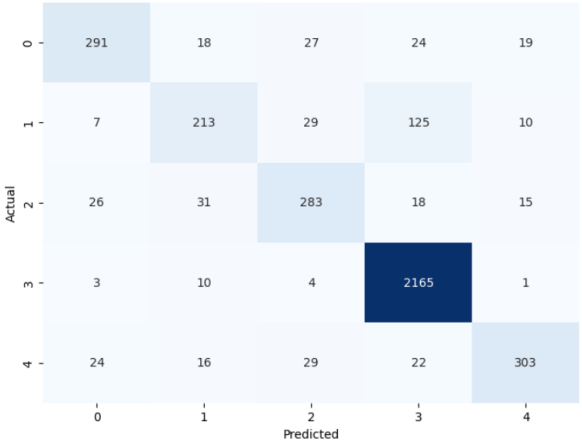
9.2 Results of Logistic Regression

Logistic Regression Classification Report

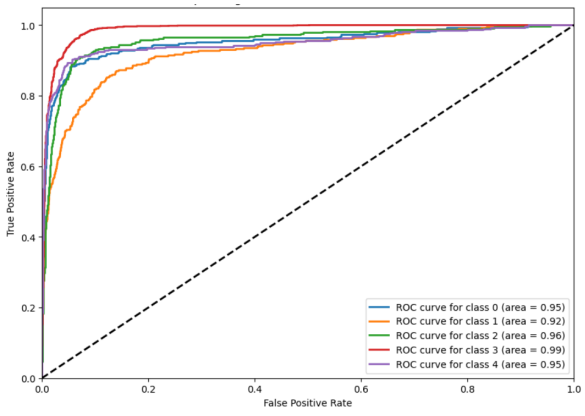
Class	Precision	Recall	F1-Score	Support
Anxiety	0.83	0.77	0.80	379
Bipolar	0.74	0.55	0.63	384
Depression	0.76	0.76	0.76	373
Normal	0.92	0.99	0.95	2183
PTSD	0.87	0.77	0.82	394
Accuracy	87.66%			
Macro Avg	0.82	0.77	0.79	3713
Weighted Avg	0.87	0.88	0.87	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.95
Bipolar	0.92
Depression	0.96
Normal	0.99
PTSD	0.95



(a) Confusion Matrix (Logistic Regression)



(b) ROC AUC (Logistic Regression)

Figure 27: Evaluation Results for Logistic Regression

The Logistic Regression model performed well with an overall accuracy of 87.66%, indicating that the model correctly classified the majority of the instances. The classification report shows high precision, recall, and F1-scores for the 'Normal' class, which was expected due to its large number of instances. However, the 'Bipolar' and 'Anxiety' classes have lower recall and F1-scores, suggesting that the model struggles more with these classes. The confusion matrix highlights the misclassifications. For example, 'Anxiety' is often confused with 'Depression'

and 'PTSD,' while the 'Normal' class is well-separated from the other classes. The large number of instances in the 'Normal' class could have contributed to the high accuracy but also to the imbalance in performance across other classes. The ROC curve AUC scores indicate that the model performs well in distinguishing between the classes. The 'Normal' class has the highest AUC (0.99), which is expected due to the large proportion of 'Normal' instances. Other classes, like 'Anxiety' and 'Depression,' also have high AUC scores (0.95 and 0.96, respectively), indicating that the model is capable of distinguishing them effectively.

9.3 Results of Naive Bayes

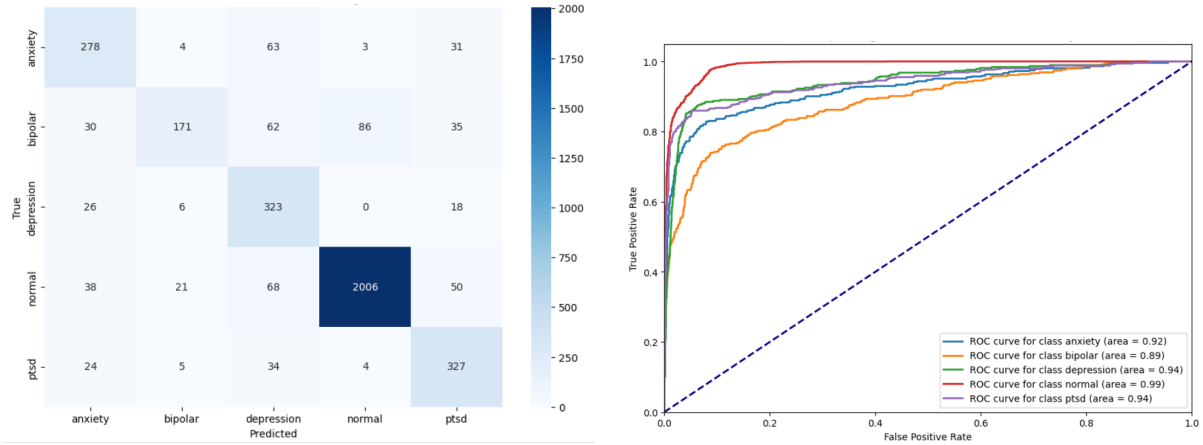
Naive Bayes Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.70	0.73	0.72	379
Bipolar	0.83	0.45	0.58	384
Depression	0.59	0.87	0.70	373
Normal	0.96	0.92	0.94	2183
PTSD	0.71	0.83	0.76	394
Accuracy	83.63%			
Macro Avg	0.76	0.76	0.74	3713
Weighted Avg	0.85	0.84	0.84	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.92
Bipolar	0.89
Depression	0.94
Normal	0.99
PTSD	0.94

The Naive Bayes model achieved an overall accuracy of 83.63%, indicating a reasonable performance. The classification report shows strong results for the Normal class, with a precision of 0.96 and a recall of 0.92, resulting in an F1-score of 0.94. However, the Bipolar class exhibits much lower recall (0.45) and F1-score (0.58), suggesting that the model struggles to accurately identify Bipolar instances. Misclassifications are more frequent for Bipolar, often being confused with Depression and PTSD. The Anxiety class, although having decent precision (0.70), shows a lower recall (0.73), indicating that it also faces challenges in classification. The Confusion Matrix highlights that the model has difficulty distinguishing Bipolar and Depression from each other, with a large number of misclassifications across these classes. On the other hand, the Normal class is well-separated and correctly classified, with 2006 true positives, which likely contributes to the high overall accuracy. PTSD also shows relatively good classification results,



(a) Confusion Matrix (Naive Bayes)

(b) ROC AUC (Naive Bayes)

Figure 28: Evaluation Results for Naive Bayes

with misclassifications being less frequent. The ROC AUC Curve Areas show that the model performs well for most classes, with the Normal class having the highest AUC score (0.99), followed by Depression and PTSD with AUCs of 0.94. While the model performs well for distinguishing between classes like Normal and Depression, the lower AUC for Bipolar (0.89) indicates that there may still be room for improvement in distinguishing this class from others.

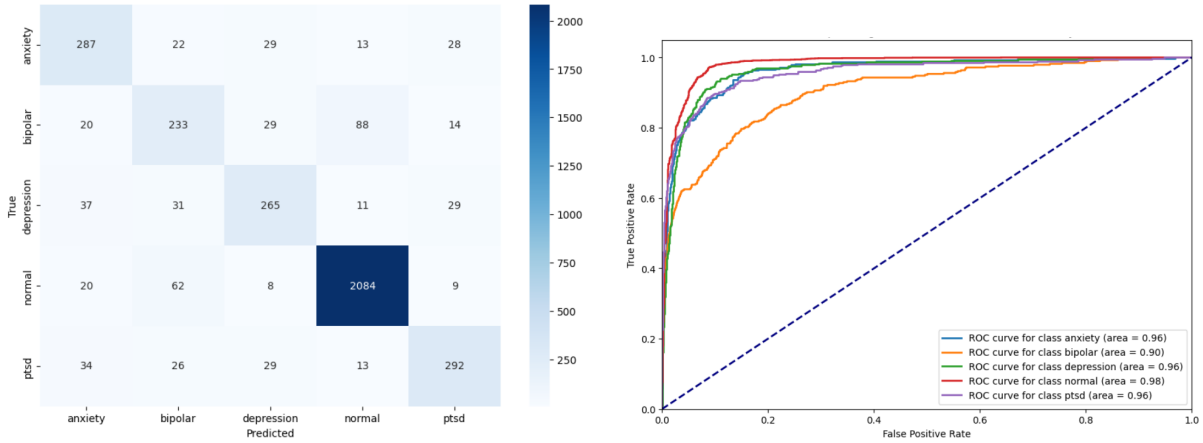
9.4 Results of Support Vector Machine

SVM Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.72	0.76	0.74	379
Bipolar	0.62	0.61	0.61	384
Depression	0.74	0.71	0.72	373
Normal	0.94	0.95	0.95	2183
PTSD	0.78	0.74	0.76	394
Accuracy	85.13%			
Macro Avg	0.76	0.75	0.76	3713
Weighted Avg	0.85	0.85	0.85	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.96
Bipolar	0.90
Depression	0.96
Normal	0.98
PTSD	0.96



(a) Confusion Matrix (SVM)

(b) ROC AUC (SVM)

Figure 29: Evaluation Results for SVM

The Support Vector Machine (SVM) model achieved an accuracy of 85.13%, demonstrating strong performance in classifying the various mental health conditions. The classification report indicates that the Normal class has the highest precision (0.94) and recall (0.95), resulting in an F1-score of 0.95, showing that the model is particularly good at identifying instances of normal behavior. However, the Bipolar class shows lower performance, with a precision of 0.62 and recall of 0.61, suggesting that the model struggles more with identifying bipolar disorder instances. The recall for the Anxiety class is relatively high (0.76), though precision is somewhat lower (0.72), indicating a balanced classification performance for this condition. The confusion matrix reveals that the model is generally accurate, with the Normal class being correctly classified most of the time (2084 true positives). However, some misclassifications occur for classes like Bipolar, Depression, and PTSD, with notable misclassifications of Bipolar as Depression and Normal. These misclassifications are likely affecting the overall recall for certain classes. The ROC AUC curve areas demonstrate that the model has excellent performance in distinguishing between most classes. The Normal class has the highest AUC of 0.98, followed by Anxiety, Depression, and PTSD with AUCs of 0.96. Bipolar, although still high at 0.90, lags behind the other classes, reflecting the model's challenge in distinguishing Bipolar from other conditions.

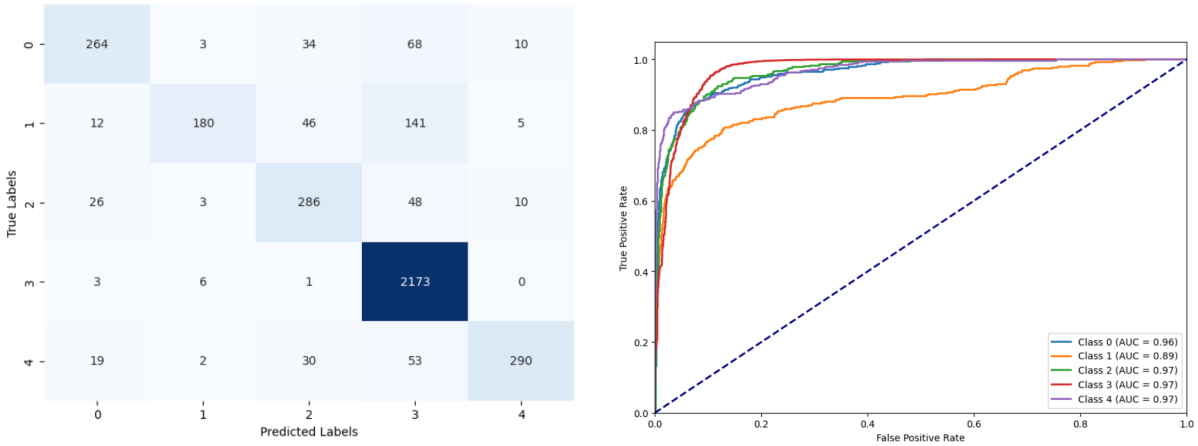
9.5 Results of Random Forest

Random Forest Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.81	0.70	0.75	379
Bipolar	0.93	0.47	0.62	384
Depression	0.72	0.77	0.74	373
Normal	0.88	1.00	0.93	2183
PTSD	0.92	0.74	0.82	394
Accuracy	86.00%			
Macro Avg	0.85	0.73	0.77	3713
Weighted Avg	0.86	0.86	0.85	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.96
Bipolar	0.89
Depression	0.97
Normal	0.97
PTSD	0.97



(a) Confusion Matrix (Random Forest)

(b) ROC AUC (Random Forest)

Figure 30: Evaluation Results for Random Forest

The Random Forest model achieved an accuracy of 86.00%, indicating a strong performance in classifying the mental health conditions. The classification report shows that the Normal class achieved the highest recall (1.00) and a precision of 0.88, resulting in a high F1-score of 0.93. This reflects the model's ability to correctly classify the majority of the Normal instances. The Bipolar class, however, shows a much lower recall (0.47), which indicates that the model has difficulty identifying instances of Bipolar disorder, as reflected by its F1-score of 0.62. The

Anxiety and PTSD classes have relatively balanced performance, with moderate precision and recall values. The confusion matrix illustrates the distribution of misclassifications. The Normal class is correctly classified almost entirely (2173 true positives), while other classes like Bipolar and PTSD exhibit significant misclassifications, particularly Bipolar, which is often misclassified as Depression and Normal. The ROC AUC scores for all classes are quite high, indicating that the model is effective at distinguishing between the classes. The Anxiety, Depression, Normal, and PTSD classes each have AUC values above 0.96, with the Normal and Depression classes achieving 0.97. Bipolar has the lowest AUC at 0.89, which corresponds to its lower classification performance. These AUC values suggest that the Random Forest model is proficient in distinguishing between most of the classes, though it faces challenges in identifying Bipolar disorder.

9.6 Results of XGBoost

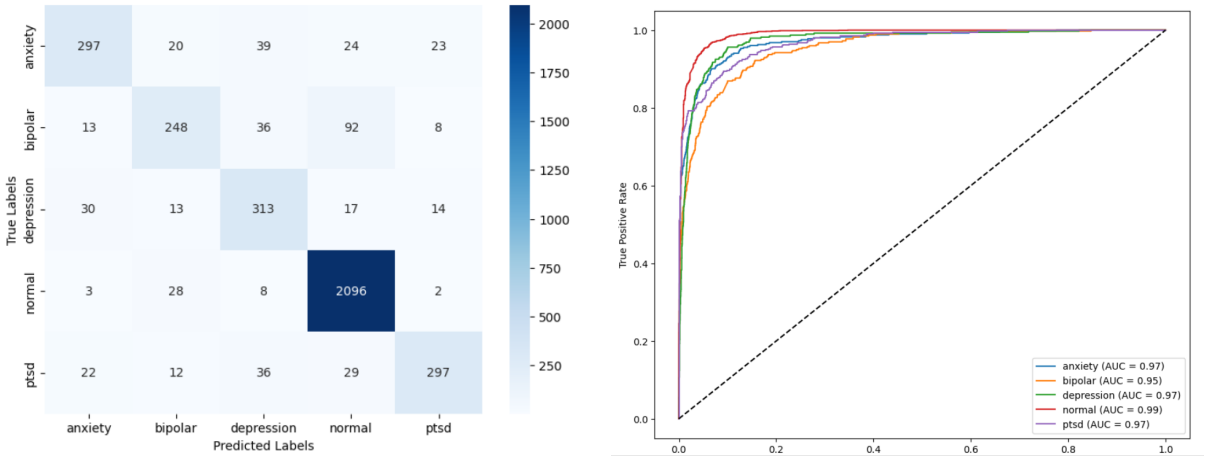
XGBoost Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.81	0.74	0.77	403
Bipolar	0.77	0.62	0.69	397
Depression	0.72	0.81	0.76	387
Normal	0.93	0.98	0.95	2137
PTSD	0.86	0.75	0.80	396
Accuracy	87.39%			
Macro Avg	0.82	0.78	0.80	3720
Weighted Avg	0.87	0.87	0.87	3720

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.97
Bipolar	0.95
Depression	0.97
Normal	0.99
PTSD	0.97

The XGBoost model achieved an accuracy of 87.39%, demonstrating strong overall performance. The classification report indicates high precision and recall for the 'Normal' class, which is likely due to its substantial representation in the dataset. The 'Anxiety' and 'PTSD' classes also showed reasonable results, with the 'Anxiety' class achieving a precision of 0.81 and a recall of 0.74. However, the 'Bipolar' class exhibited a lower recall (0.62), suggesting that the model struggles more with this class. The confusion matrix highlights a well-separated 'Normal' class, while the 'Bipolar' and 'PTSD' classes experience more confusion with other



(a) Confusion Matrix (XGBoost)

(b) ROC AUC (XGBoost)

Figure 31: Evaluation Results for XGBoost

categories. The ROC AUC scores further emphasize the model's good discriminatory capability, with AUC values of 0.97 for 'Anxiety,' 'Depression,' and 'PTSD,' and 0.99 for 'Normal.' These scores indicate that the model is proficient at distinguishing between different mental health conditions in the dataset.

9.7 Results of KNN

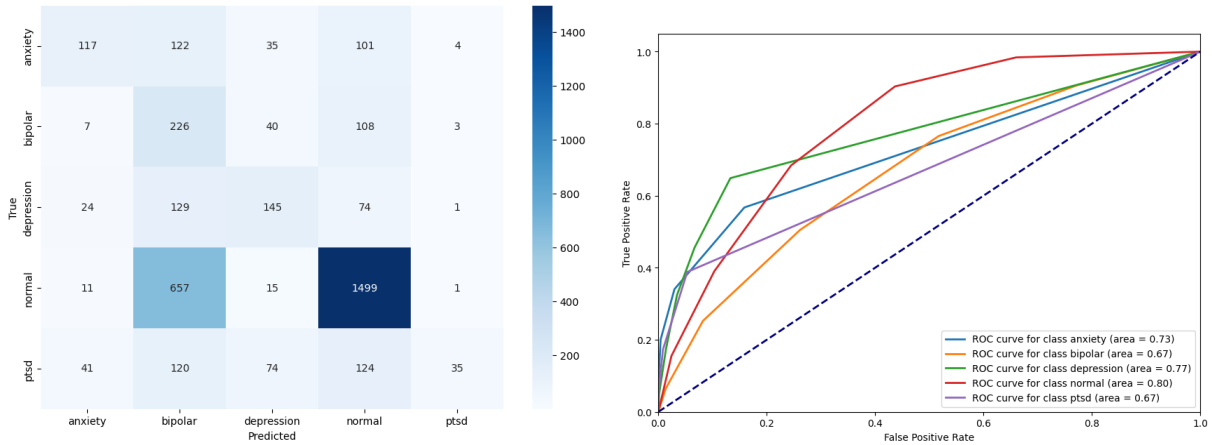
KNN Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.58	0.31	0.40	379
Bipolar	0.18	0.59	0.28	384
Depression	0.47	0.39	0.43	373
Normal	0.79	0.69	0.73	2183
PTSD	0.80	0.09	0.16	394
Accuracy	54.46%			
Macro Avg	0.56	0.41	0.40	3713
Weighted Avg	0.67	0.54	0.56	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.73
Bipolar	0.67
Depression	0.77
Normal	0.80
PTSD	0.67

The KNN model achieved an accuracy of 54.46%, which is considerably lower than the XGBoost model. The classification report reveals a significant imbalance in performance across



(a) Confusion Matrix (KNN)

(b) ROC AUC (KNN)

Figure 32: Evaluation Results for KNN

classes. The 'Normal' class achieved relatively high precision (0.79) and recall (0.69), reflecting the model's tendency to perform well with dominant classes. However, the 'Anxiety' and 'PTSD' classes performed poorly, especially with PTSD having a very low recall of 0.09, indicating substantial misclassification. The confusion matrix suggests that the model struggles with distinguishing between some classes, especially 'Anxiety' and 'Bipolar,' where confusion with other categories is high. The ROC AUC scores are relatively lower compared to the XGBoost model, with 'Normal' having the highest AUC of 0.80. These results indicate that while the KNN model offers a lower overall performance, it still provides a reasonable level of discrimination for the 'Normal' class.

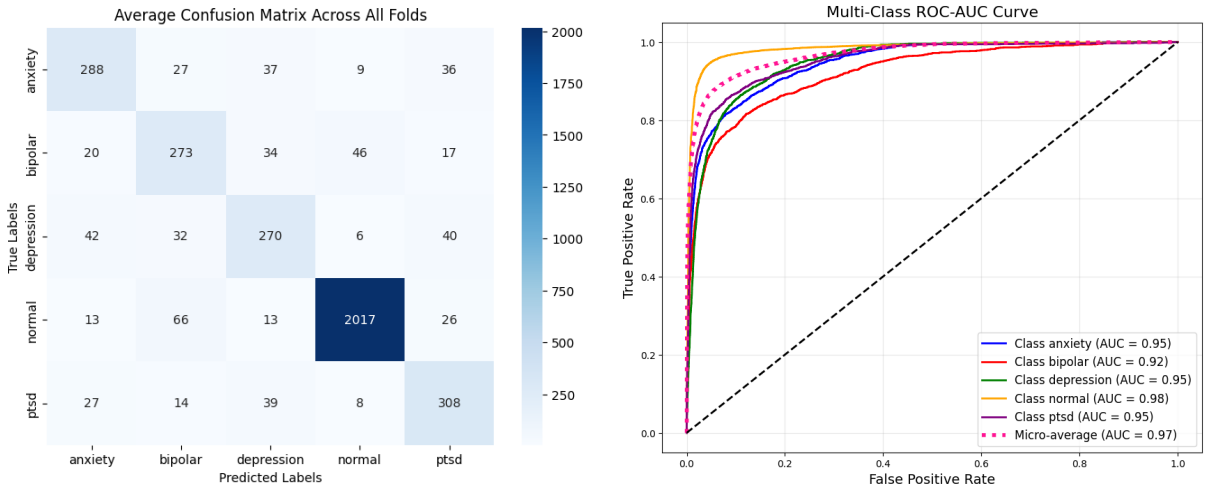
9.8 Results of LSTM

LSTM Classification Report

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Anxiety	0.74	0.72	0.73	1999
Bipolar	0.66	0.70	0.68	1964
Depression	0.68	0.69	0.69	1959
Normal	0.97	0.94	0.95	10688
PTSD	0.72	0.78	0.75	1987
Accuracy	84.91%			
Macro Avg	0.75	0.77	0.76	18597
Weighted Avg	0.85	0.85	0.85	18597

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.95
Bipolar	0.92
Depression	0.95
Normal	0.98
PTSD	0.95



(a) Confusion Matrix (LSTM)

(b) ROC AUC (LSTM)

Figure 33: Evaluation Results for LSTM

The LSTM model achieved an accuracy of 84.91%, indicating strong performance. The classification report shows high precision, recall, and F1-scores for the 'Normal' class, which dominates the dataset. The 'Anxiety' class has reasonable performance, with a precision of 0.74 and recall of 0.72, while the 'Bipolar' and 'PTSD' classes have slightly lower precision and recall. The 'Bipolar' class shows a precision of 0.66 and recall of 0.70, while 'PTSD' has a precision of 0.72 and recall of 0.78. The confusion matrix reflects this, with 'Bipolar' and 'PTSD' misclassified with other classes, while 'Normal' is well-separated. The ROC AUC scores highlight the model's ability to distinguish between classes effectively, with 'Normal' scoring the highest.

9.9 Results of Hyperparameter Tuning

Logistic Regression

Hyperparameter Tuning on Logistic Regression

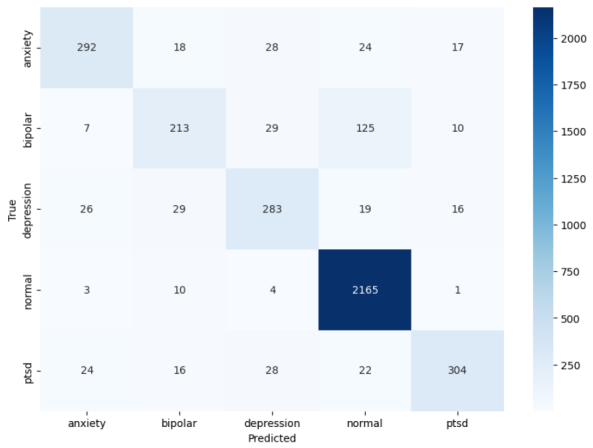
Best Hyperparameters	Value
Solver	liblinear
Penalty	l2
C	1

Classification Report

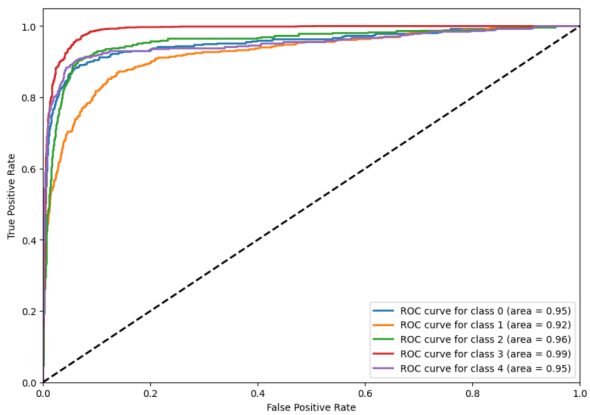
Class	Precision	Recall	F1-Score	Support
Anxiety	0.83	0.77	0.80	379
Bipolar	0.74	0.55	0.64	384
Depression	0.76	0.76	0.76	373
Normal	0.92	0.99	0.95	2183
PTSD	0.87	0.77	0.82	394
Accuracy	87.72%			
Macro Avg	0.83	0.77	0.79	3713
Weighted Avg	0.87	0.88	0.87	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.95
Bipolar	0.92
Depression	0.96
Normal	0.99
PTSD	0.95



(a) Classification Matrix after Hyperparameter Tuning (Logistic Regression)



(b) ROC AUC after Hyperparameter Tuning (Logistic Regression)

Figure 34: Evaluation Results after Hyperparameter Tuning (Logistic Regression)

The results of hyperparameter tuning on the Logistic Regression model show that the best hyperparameters were a solver of `liblinear`, a penalty of `l2`, and a regularization parameter (`C`) of 1. The model achieved an overall accuracy of 87.72%, with high performance across most classes. The classification report reveals that the model performs well on `normal` class with an accuracy of 92% in predicting this class, whereas performance on `bipolar` is slightly lower $64\% F1 - score$. The ROC AUC scores indicate that the model is highly effective in distinguishing between different classes, with `normal` achieving a near-perfect AUC of 0.99, followed closely by `depression` at 0.96. These results suggest that the model is well-tuned, particularly for classes like `normal` and `anxiety`.

K-Nearest Neighbours

Hyperparameter Tuning on k-NN

Best Hyperparameters	Value
Weights	distance
n_neighbors	10
Metric	euclidean

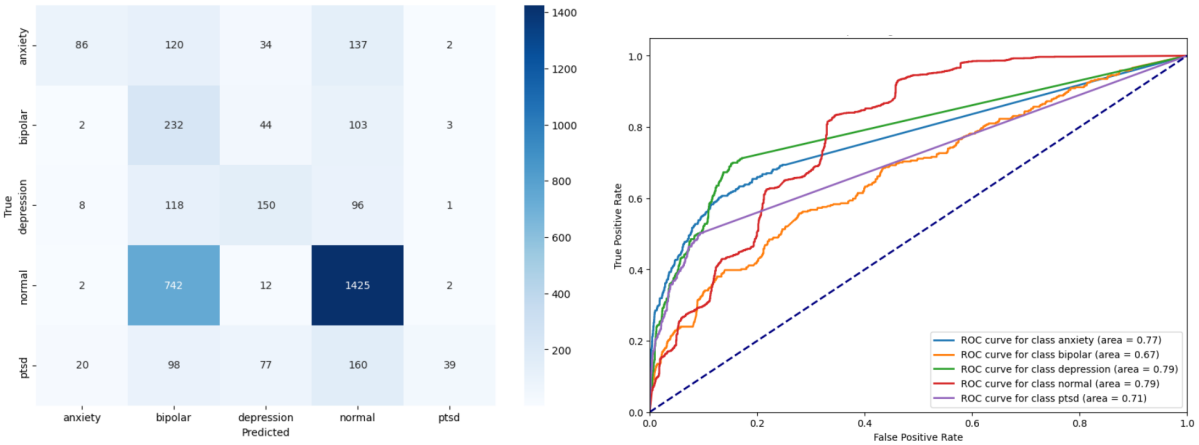
Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.73	0.23	0.35	379
Bipolar	0.18	0.60	0.27	384
Depression	0.47	0.40	0.43	373
Normal	0.74	0.65	0.69	2183
PTSD	0.83	0.10	0.18	394
Accuracy	52.03%			
Macro Avg	0.59	0.40	0.39	3713
Weighted Avg	0.66	0.52	0.53	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.77
Bipolar	0.67
Depression	0.79
Normal	0.79
PTSD	0.71

The results of hyperparameter tuning on the k-Nearest Neighbors (k-NN) model show that the best hyperparameters were `weights` set to `distance`, `n_neighbors` set to 10, and `metric` set to `euclidean`. Despite these optimizations, the model achieved an overall accuracy of 52.03%, which is relatively low. The classification report indicates that the model struggled to effectively predict the `anxiety` and `PTSD` classes, with particularly low recall



(a) Classification Matrix after Hyperparameter Tuning (KNN)

(b) ROC AUC after Hyperparameter Tuning (KNN)

Figure 35: Evaluation Results after Hyperparameter Tuning (KNN)

values (0.23 and 0.10, respectively). However, it performed reasonably well on the `normal` class with an F1-score of 0.69. The ROC AUC scores reflect this by showing relatively poor performance for `bipolar` (0.67) and `PTSD` (0.71), while `normal` and `depression` had slightly better values at 0.79 each. These results suggest that k-NN is not the best model for this dataset, as it fails to consistently classify the minority classes effectively.

Support Vector Machine

Hyperparameter Tuning on SVM

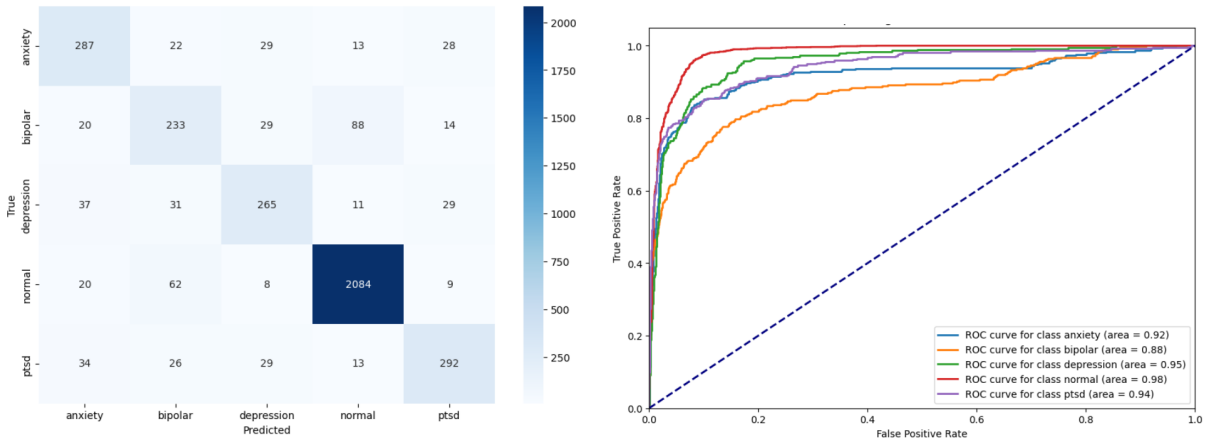
Best Hyperparameters	Value
Kernel	linear
Gamma	scale
C	1

Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.72	0.76	0.74	379
Bipolar	0.62	0.61	0.61	384
Depression	0.74	0.71	0.72	373
Normal	0.94	0.95	0.95	2183
PTSD	0.78	0.74	0.76	394
Accuracy	85.13%			
Macro Avg	0.76	0.75	0.76	3713
Weighted Avg	0.85	0.85	0.85	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.92
Bipolar	0.88
Depression	0.95
Normal	0.98
PTSD	0.94



(a) Classification Matrix after Hyperparameter Tuning (SVM)

(b) ROC AUC after Hyperparameter Tuning (SVM)

Figure 36: Evaluation Results after Hyperparameter Tuning (SVM)

The results of hyperparameter tuning on the Support Vector Machine (SVM) model reveal that the best hyperparameters were `kernel` set to `linear`, `gamma` set to `scale`, and `C` set to 1. With these settings, the model achieved an overall accuracy of 85.13%. The classification report indicates solid performance across most classes, with `normal` achieving the highest F1-score of 0.95, followed by `depression` at 0.72 and `anxiety` at 0.74. The `bipolar` class had a slightly lower F1-score of 0.61, indicating that the model was less effective in distinguishing this class. The ROC AUC values were strong for most classes, with `normal` reaching 0.98 and `depression`, `PTSD`, and `anxiety` all scoring above 0.90, showing that the SVM model effectively discriminates between the classes. Overall, the SVM model performed well on this dataset, particularly for the `normal` and `anxiety` classes, making it a strong contender for mental health issue classification.

Naive Bayes

Hyperparameter Tuning on Naive Bayes

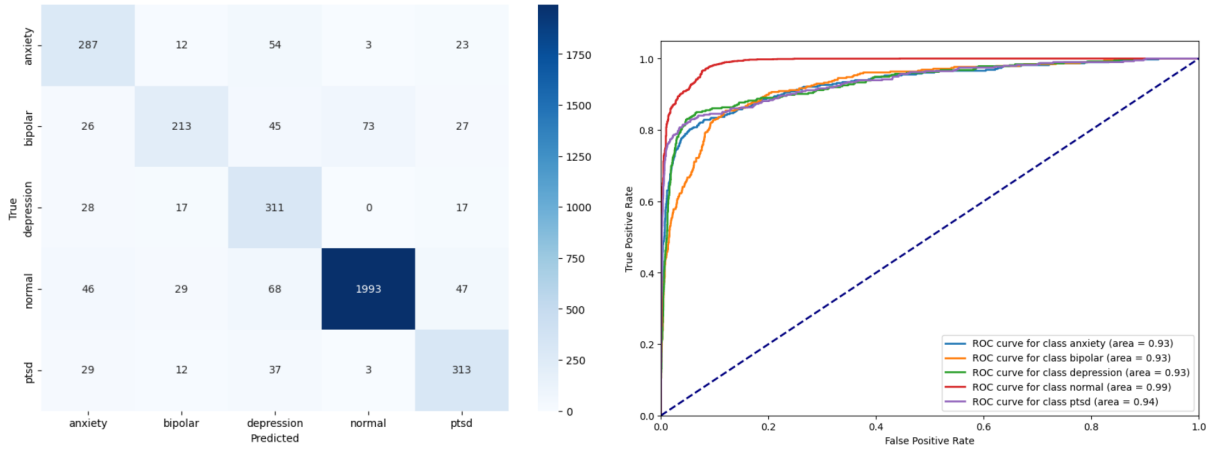
Best Hyperparameters	Value
Alpha	0.2914180608409973

Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.69	0.76	0.72	379
Bipolar	0.75	0.55	0.64	384
Depression	0.60	0.83	0.70	373
Normal	0.96	0.91	0.94	2183
PTSD	0.73	0.79	0.76	394
Accuracy	83.95%			
Macro Avg	0.75	0.77	0.75	3713
Weighted Avg	0.85	0.84	0.84	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.93
Bipolar	0.93
Depression	0.93
Normal	0.99
PTSD	0.94



(a) Classification Matrix after Hyperparameter Tuning (Naive Bayes)

(b) ROC AUC after Hyperparameter Tuning (Naive Bayes)

Figure 37: Evaluation Results after Hyperparameter Tuning (Naive Bayes)

The results of hyperparameter tuning on the Naive Bayes model show that the best hyperparameter was `alpha` set to 0.2914. With this optimal value, the model achieved an accuracy of 83.95%. The classification report reveals that the `normal` class performed the best, with a high F1-score of 0.94, followed by `PTSD` at 0.76. The `anxiety` class achieved a solid F1-score of 0.72, while `bipolar` and `depression` had lower scores. The ROC AUC curve areas demonstrate strong performance across all classes, with `normal` reaching a perfect 0.99 and the other classes scoring above 0.90. Overall, the Naive Bayes model performed well, especially in identifying `normal` and `PTSD`, making it a good candidate for mental health

issue classification. After performing hyperparameter tuning using RandomizedSearchCV, Logistic Regression emerged as the best-performing model for mental health classification. The model achieved an accuracy of 87.72%, demonstrating its strong ability to correctly classify mental health issues based on text data. The best hyperparameters found during tuning were `'solver': 'liblinear', 'penalty': 'l2', 'C': 1`, which contributed to the model's robustness and efficiency. In terms of classification performance, Logistic Regression showed a high recall and precision, especially for classes like *normal*, with a recall of 0.99, indicating its proficiency in identifying the most prevalent class in the dataset.

K-Nearest Neighbors (KNN) performs significantly poorly on this dataset compared to other algorithms, both before and after hyperparameter tuning. The dataset used, which includes Reddit posts with corresponding mental health labels, presents challenges for KNN due to its reliance on distance-based metrics. Specifically, KNN uses the distance between data points to classify them, but this approach is sensitive to the scale and distribution of features. In the case of text data, the feature space can be sparse and high-dimensional, making it difficult for KNN to find meaningful relationships between the posts and their associated mental health labels. After hyperparameter tuning, KNN was optimized with parameters such as **weights='distance'**, **n_neighbors=10**, and **metric='euclidean'**, yet its performance remains suboptimal. This suggests that KNN struggles with the complexities inherent in text classification tasks, where the relationships between features (such as words or phrases in a Reddit post) are often non-linear and context-dependent. Despite tuning, the algorithm's performance metrics—accuracy, precision, recall, and F1-score—remain notably lower than those of other algorithms like Logistic Regression, Naive Bayes, or SVM. Additionally, KNN's inability to capture these subtleties in language means it performs poorly, particularly in distinguishing between categories like anxiety, bipolar disorder, or PTSD, where the text patterns may be more nuanced. Overall, KNN's reliance on proximity without considering the richer context of textual data likely contributes to its lower performance on this mental health classification task.

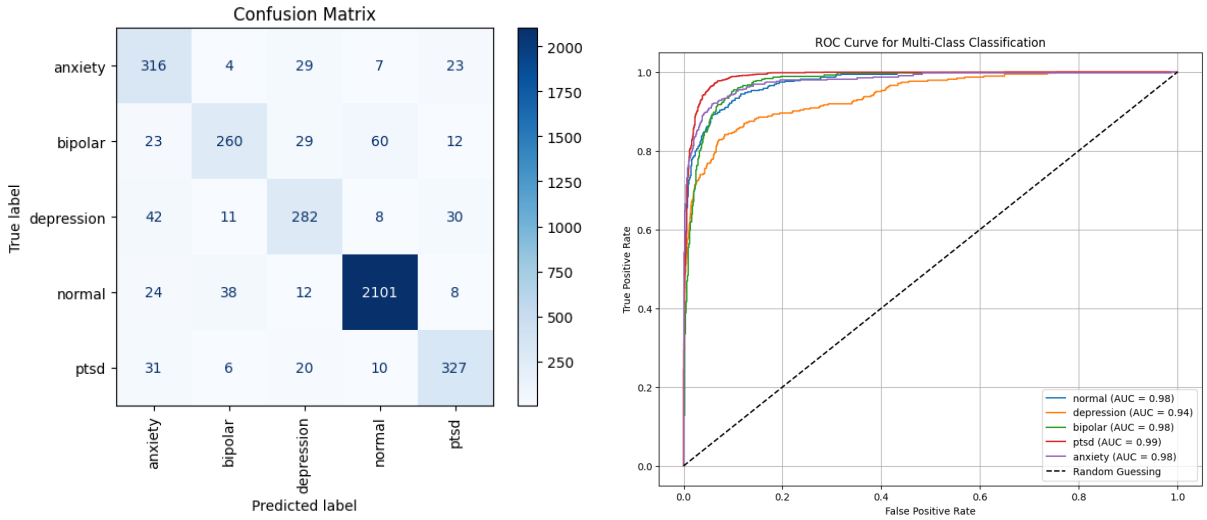
9.10 Results from Transformer based model

Classification Report

Class	Precision	Recall	F1-Score	Support
Anxiety	0.75	0.78	0.76	379
Bipolar	0.76	0.69	0.73	384
Depression	0.82	0.72	0.77	373
Normal	0.95	0.97	0.96	2183
PTSD	0.79	0.80	0.79	394
Accuracy	88.5%			
Macro Avg	0.81	0.79	0.80	3713
Weighted Avg	0.88	0.88	0.88	3713

ROC Curve Areas for Each Class

Class	ROC AUC
Anxiety	0.97
Bipolar	0.94
Depression	0.98
Normal	0.99
PTSD	0.97



(a) Confusion Matrix for Transformer based Model (b) ROC AUC for Transformer based Model

Figure 38: Evaluation Results for Transformer based Model

The classification report and ROC AUC scores for your transformer-based model demonstrate strong performance across all classes, reflecting its ability to effectively handle the nuances of mental health text. The model achieved high precision, recall, and F1-scores, particularly for the ”Normal” and ”Anxiety” classes, where the precision and recall values were very high. This indicates that the model is good at identifying these conditions with minimal false positives and negatives. The high ROC AUC scores, ranging from 0.94 to 0.99 for each class, further highlight the model’s ability to distinguish between different mental health conditions, with ”Normal” and ”Anxiety” showing particularly strong discriminatory power. The transformer-based architecture has likely contributed to these results by enabling the model to capture the complex relationships and contextual cues present in the text, which traditional models might struggle to identify. The pre-training on large datasets and fine-tuning for specific tasks has likely improved the model’s ability to generalize across various mental health issues, contributing to the high accuracy of 88.5%. This model’s ability to balance performance across both major and minor classes is reflected in its macro and weighted averages, which show that it handles the class imbalance well and performs effectively on all categories. The transformer model’s ability to process long-range dependencies and understand contextual relationships in

the text has clearly enhanced its ability to classify mental health issues accurately, resulting in improved overall performance.

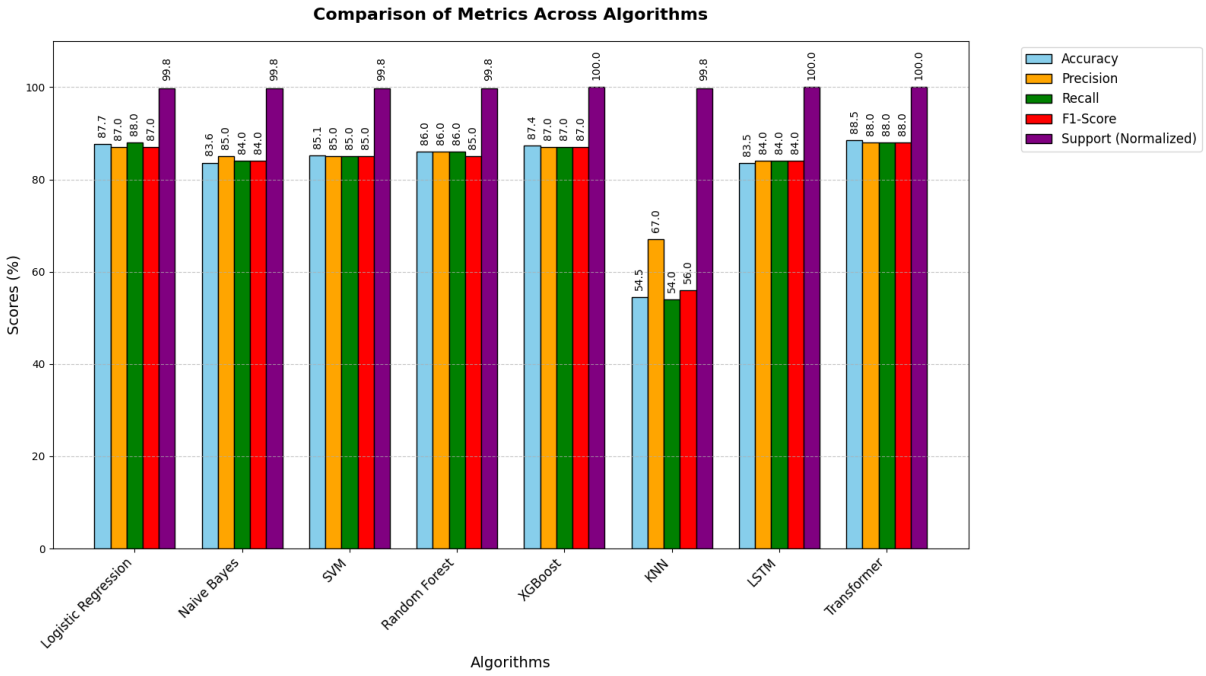


Figure 39: Result Comparison of the Algorithms

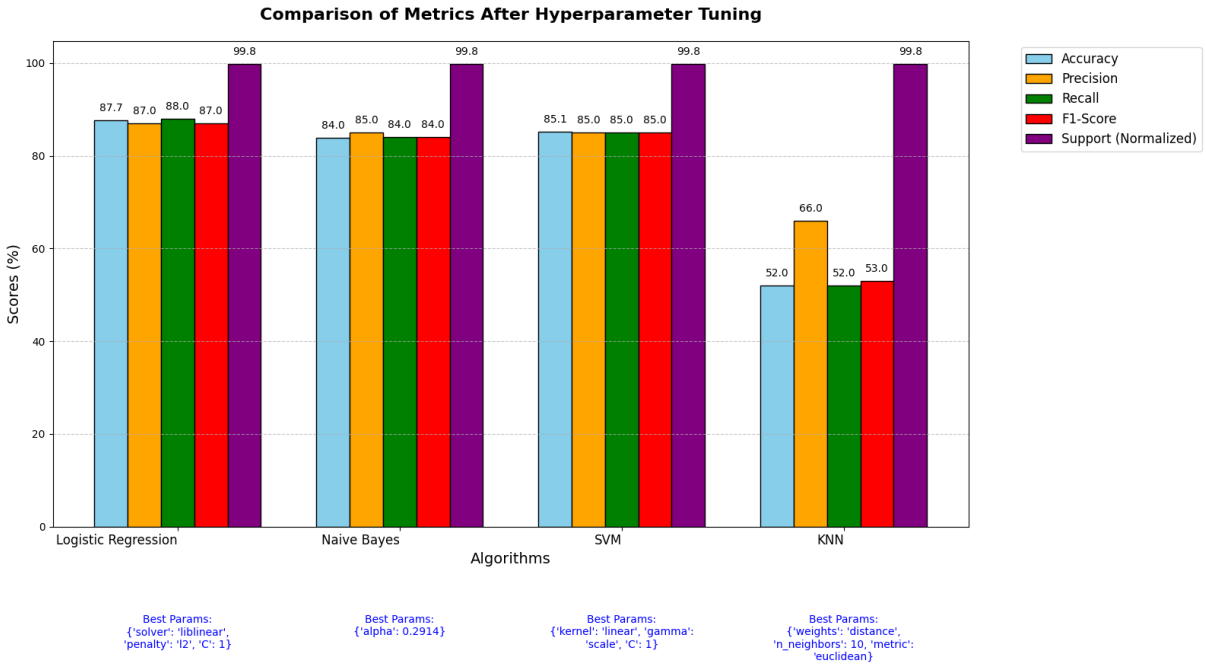


Figure 40: Result Comparison after Hyperparameter Tuning

9.11 Comparison of different tokenizations

Model Performance Comparison

Model	BoW	TFIDF	LIWC	Word2Vec	N-Gram
Logistic Regression	87.66	86.02	66.36	79.40	87.13
KNN	54.56	75.06	70.70	75.52	43.06
SVC	85.13	88.26	68.14	77.89	84.33
Naive Bayes	83.63	79.42	59.63	60.44	81.71
Random Forest	85.32	85.73	76.73	79.88	79.02
XGB	87.42	87.39	78.56	81.01	87.96

In ensemble learning, the goal is to combine multiple base models to improve the overall performance of the system. However, using models like Logistic Regression (LR), Naive Bayes (NB), K-Nearest Neighbors (KNN), Support Vector Machine (SVM), Long Short-Term Memory (LSTM), XGBoost (XGB), and Random Forest (RF) all together as base models in an ensemble is not always feasible. Each of these models has its strengths and weaknesses, and the combination of them can lead to various challenges, particularly in terms of computational efficiency, model size, and accuracy.

For instance, Logistic Regression, Naive Bayes, and SVM work well with simpler vectorization methods like Bag of Words (BoW) because these methods are efficient and provide good results in terms of classification accuracy. On the other hand, more complex models like XGBoost and Random Forest can handle feature extraction methods such as TFIDF (Term Frequency-Inverse Document Frequency) better due to their ability to manage more intricate feature spaces. However, the problem arises when using these models together in a single ensemble. Although SVM with TFIDF can provide high accuracy on its own, it can negatively affect the performance of the final ensemble model. This is likely because the TFIDF features, while effective for models like SVM, may not align well with other base models, such as LR and NB, which rely more on the simpler, sparse representation of the data.

XGBoost, when used with N-Gram features, tends to give the highest accuracy individually, but the model size and computational demands are significantly higher, making it impractical to include in an ensemble. The larger size of the model can lead to slower inference times and increased memory usage, which is particularly problematic in real-time systems or when deploying the model at scale. Additionally, KNN is another model that is not considered as a base model in the ensemble due to its computational intensity. KNN requires the entire training dataset to be stored and does not generalize well to unseen data, and its performance deteriorates as the dataset grows, leading to slow predictions. Thus, including KNN would only further increase the computational cost and reduce the overall performance of the ensemble model.

Random Forest is also omitted as a base model in the ensemble for similar reasons. While Random Forest can perform well as an individual model, it is computationally intensive due to its multiple decision trees and large size, which can lead to longer training and prediction times. When included in an ensemble, its computational overhead would outweigh the benefits, especially if the ensemble consists of already complex models.

The reason for selecting Bag of Words (BoW) for LR, NB, and SVM, and TFIDF for XGBoost lies in the nature of these feature extraction methods and the models themselves. BoW is a simple, yet effective method for converting text data into numerical form. It represents each word as a feature in a vector space, where the value is the frequency of the word in the document. This works well for models like LR, NB, and SVM because they perform better when the features are sparse and straightforward. The simplicity of BoW ensures that these models are not overwhelmed by the sheer number of features, which helps maintain interpretability and reduces computational cost.

On the other hand, TFIDF takes into account not only the frequency of words in a document but also their importance in the entire corpus. Words that occur frequently across many documents are weighted lower, while words that are more specific to a particular document are given more importance. This approach is particularly useful for models like XGBoost, which can handle the more complex and rich feature sets that TFIDF generates. The increased dimensionality of TFIDF features provides a better representation of the data, which XGBoost leverages to improve accuracy. Combining BoW for LR, NB, and SVM with TFIDF for XGBoost ensures that the strengths of both simpler and more complex feature extraction methods are utilized. The simpler methods prevent the models from being overwhelmed by excessive features, while TFIDF provides more nuanced features for XGBoost, leading to improved accuracy.

Ultimately, the combination of **BoW** and **TFIDF** strikes a balance between simplicity and complexity, ensuring that each model works efficiently and effectively within the ensemble without causing undue computational strain. This balanced approach helps to maintain both high accuracy and manageable computational requirements, making it the best choice for feature extraction in this scenario.

9.12 Results from Ensemble Model Training and Testing

9.12.1 Ensemble Model 1

Cross-Validation Accuracy Scores

Cross-Validation Accuracy Scores	Value
1	0.96364126
2	0.95744681
3	0.96175599
4	0.95933208
5	0.96202532
Mean Validation Accuracy	**96.08%**

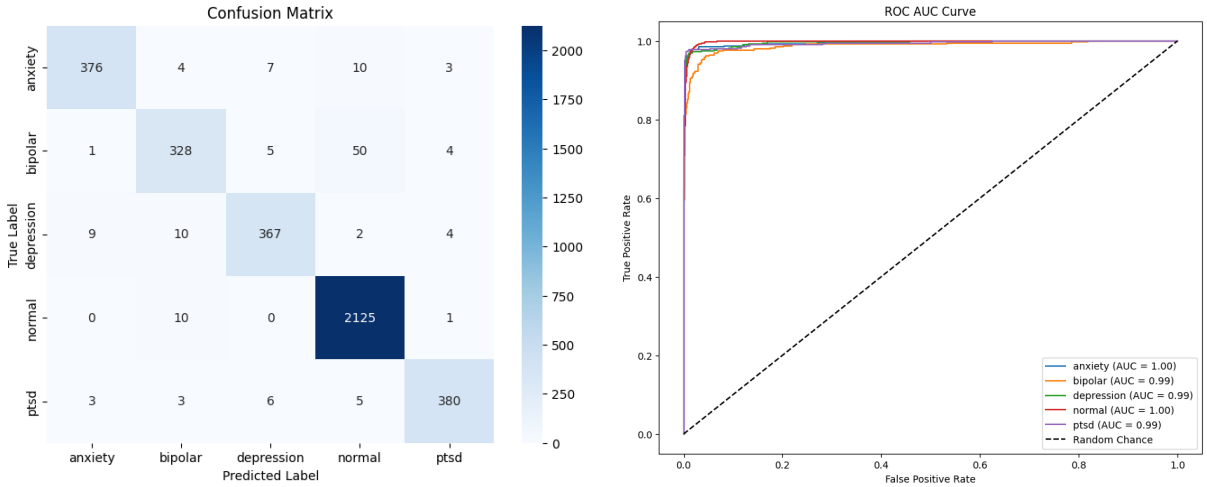
Classification Report for Ensemble Model1

	Precision	Recall	F1-Score	Support
Anxiety	0.97	0.94	0.95	400
Bipolar	0.92	0.85	0.88	388
Depression	0.95	0.94	0.94	392
Normal	0.97	0.99	0.98	2136
PTSD	0.97	0.96	0.96	397
Accuracy	**96.08%**			
Macro avg	0.96	0.93	0.95	3713
Weighted avg	0.96	0.96	0.96	3713

ROC AUC Scores for Each Class

Class	ROC AUC
Anxiety	1.0
Bipolar	0.99
Depression	0.99
Normal	1.0
PTSD	0.99

The classification report provides an insightful summary of the ensemble model's performance, which combines Logistic Regression and XGBoost to classify mental health issues. Precision is a key metric that measures the proportion of correctly predicted positive cases out of all the positive predictions made by the model. For example, a precision of 0.97 for anxiety means that when the model predicts anxiety, it is correct 97% of the time. Recall, on the other hand, reflects the model's ability to identify all actual positive cases in the dataset. A recall of 0.94 for anxiety suggests that the model correctly identifies 94% of all true anxiety cases. The F1-score, which balances precision and recall, is also an important metric. With an F1-score of 0.95 for anxiety, the model strikes a good balance between precision and recall, indicating that it performs well across both metrics. Support represents the number of actual instances of a class in the dataset, and for anxiety, there are 1999 instances in the test set. The model's overall



(a) Confusion Matrix for Ensemble Model 1 (b) ROC AUC Curve for Ensemble Model 1

Figure 41: Evaluation Results for Ensemble Model 1

accuracy is 96.08%, meaning that it correctly predicts the mental health condition for 96.08% of the test instances, reflecting the overall effectiveness of the model. The macro average, which takes the unweighted mean of precision, recall, and F1-scores across all classes, shows an average precision of 0.96, recall of 0.93, and F1-score of 0.95. This provides a general idea of the model's performance across all classes without accounting for class imbalance. The weighted average, however, adjusts for the class distribution by giving more weight to classes with more instances. The weighted averages for precision, recall, and F1-score are all 0.96, reflecting strong overall performance, especially when considering the differing class sizes. The ROC AUC scores are another key performance indicator that evaluates the model's ability to distinguish between different classes. The AUC score ranges from 0 to 1, with 1 representing perfect classification. A perfect AUC score of 1.0 for anxiety, normal, and PTSD indicates that the model can perfectly differentiate between these classes and others. The AUC scores for bipolar and depression are 0.99, which is almost perfect, further demonstrating the model's strong performance across all classes. Overall, these metrics indicate that the ensemble model is highly effective at classifying mental health issues, with robust precision, recall, and the ability to distinguish between the different conditions. The confusion matrix will provide insight into the number of false positives, false negatives, true positives, and true negatives for each class. By looking at the confusion matrix, you can better understand where the model might be misclassifying certain classes, such as when it confuses anxiety with depression or bipolar with PTSD. For example, a high number of false positives in the anxiety class would mean the model is often predicting anxiety when it's actually a different mental health condition. Conversely, a high number of false negatives would mean the model is missing anxiety cases. The combination of logistic regression and XGBoost allows the ensemble model to leverage

the strengths of both algorithms. While logistic regression is simple and interpretable, it might struggle with non-linearities in the data. XGBoost, on the other hand, is a powerful gradient boosting algorithm that can capture complex patterns in the data. By stacking their predictions, the ensemble model benefits from both simplicity and complexity, achieving higher accuracy and robustness in text-based classification tasks.

This ensemble model, combining Logistic Regression and XGBoost, is highly effective for the mental health classification web application due to its ability to leverage the strengths of both algorithms. Logistic Regression provides a simple yet powerful linear approach to classification, offering interpretability and efficiency, while XGBoost excels at handling complex, non-linear relationships with its gradient boosting framework, enabling better accuracy and robustness. By stacking these models, the ensemble approach mitigates the individual weaknesses of each algorithm—Logistic Regression's limitations in capturing non-linear patterns and XGBoost's susceptibility to overfitting on small datasets—resulting in a more balanced, accurate, and generalizable model. This synergy ensures that the mental health classification web application delivers reliable, high-precision predictions across diverse text-based inputs.

9.12.2 Ensemble Model 2

Cross-Validation Accuracy Scores

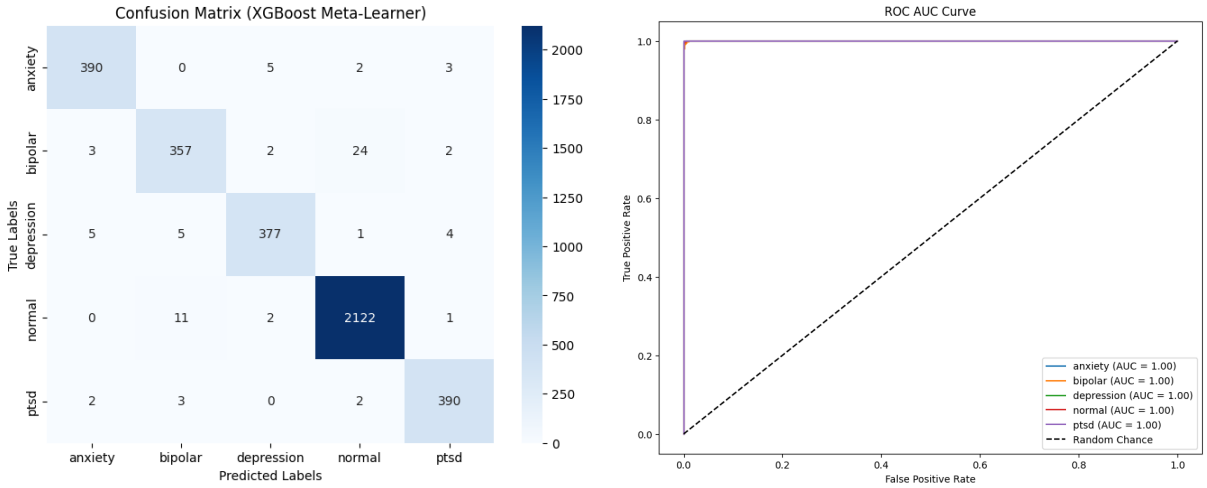
Cross-Validation Accuracy Scores	Value
1	0.980878
2	0.97629949
3	0.98249394
4	0.97522219
5	0.97710746
Mean Validation Accuracy	**97.84%**

Classification Report for Ensemble Model2

	Precision	Recall	F1-Score	Support
Anxiety	0.97	0.97	0.97	400
Bipolar	0.95	0.92	0.93	388
Depression	0.98	0.96	0.97	392
Normal	0.99	0.99	0.99	2136
PTSD	0.97	0.98	0.98	397
Accuracy	**97.93%**			
Macro avg	0.97	0.97	0.97	3713
Weighted avg	0.98	0.98	0.98	3713

ROC AUC Scores for Each Class

Class	ROC AUC
Anxiety	1.0
Bipolar	1.0
Depression	1.0
Normal	1.0
PTSD	1.0



(a) Confusion Matrix for Ensemble Model 2 (b) ROC AUC for Ensemble Model 2

Figure 42: Evaluation Results for Ensemble Model 2

Each row of the matrix represents the true class, while each column represents the predicted class. The diagonal elements (390, 357, 377, 2122, 390) show the number of correct predictions for each class (anxiety, bipolar, depression, normal, PTSD), while the off-diagonal elements represent misclassifications. For example, the model misclassified 2 samples from the anxiety class as depression and 1 sample from the bipolar class as normal, but overall, the model made very few misclassifications. An ROC AUC score of 1.0 indicates perfect classification performance for each class, meaning that the model has perfectly separated the classes based on the probability predictions. This reinforces the outstanding performance of the XGBoost model.

Using XGBoost as a meta-learner in an ensemble model is a strategic decision that brings several advantages, both in terms of performance and interpretability. To understand why XGBoost is an excellent choice, it is important to first appreciate the concept of a meta-learner and how ensemble learning works. Ensemble learning involves combining multiple base models to improve prediction accuracy by leveraging their individual strengths. In a typical ensemble method, the predictions of several base models (e.g., logistic regression, SVM, Naive Bayes, LSTM) are used as input features for a meta-learner, which then learns how to best combine these predictions to make a final decision. The meta-learner's role is to determine which base model's predictions are more reliable for each sample, thus improving the overall performance

of the ensemble. XGBoost, a gradient boosting algorithm, has several features that make it an ideal candidate for a meta-learner in this context. First, XGBoost is known for its ability to handle complex relationships in data, especially in terms of interactions between features. Unlike simpler models like logistic regression or Naive Bayes, XGBoost builds decision trees iteratively, where each new tree is focused on correcting the errors made by the previous trees. This iterative approach allows XGBoost to capture non-linear patterns and subtle interactions that might be missed by individual base models. When used as a meta-learner, XGBoost can intelligently combine the predictions from the base models by learning the optimal way to weigh the output of each base model. This is especially important in a multi-model setting where different classifiers might have complementary strengths—XGBoost can effectively ”learn” when to trust each model's prediction more, depending on the characteristics of the input data. Another advantage of using XGBoost as a meta-learner is its regularization capabilities. XGBoost includes both L1 and L2 regularization, which helps prevent overfitting. In the context of ensemble learning, this is particularly useful because, with multiple models contributing to the final prediction, the meta-learner needs to ensure it doesn't overly rely on any one base model, especially if that model is prone to overfitting. By applying regularization, XGBoost encourages the model to generalize well, ensuring that the ensemble is robust and does not become overly sensitive to the training data or to noise.

Additionally, XGBoost is highly efficient in terms of both computation and memory usage. It is designed to work well with large datasets, and its ability to handle sparse data effectively makes it a good fit for a wide variety of tasks, including those involving textual data or other high-dimensional inputs. In ensemble models, where multiple base models generate a large number of features (e.g., prediction probabilities from each base model), XGBoost's efficiency in processing these features allows it to scale well even as the number of base models or the size of the dataset grows. Another key benefit is XGBoost's ability to handle imbalanced datasets. Many classification tasks, especially in medical or mental health data, suffer from class imbalances, where some classes (e.g., “Normal”) are more frequent than others (e.g., “Bipolar” or “PTSD”). XGBoost has built-in features to adjust for class imbalances, such as using class weights or employing custom loss functions. This is important in a meta-learner context because the base models might not be equally effective across all classes. XGBoost can ensure that the ensemble model doesn't bias its predictions toward the majority class, which is crucial for improving the overall classification performance, especially for underrepresented classes. Furthermore, XGBoost provides excellent interpretability through feature importance scores. In an ensemble model, where several base models contribute to the final prediction, understanding how each base model influences the outcome is valuable for both improving the model and for explaining its behavior. XGBoost offers built-in tools to analyze the importance of individual features (in this case, the predictions of the base models), allowing practitioners to understand

how the meta-learner is making decisions and which base models are contributing the most to its predictions. Lastly, XGBoost's performance in various machine learning competitions has demonstrated its robustness. It is one of the top-performing algorithms in tasks such as classification and regression, often outperforming other methods, including random forests and neural networks. Its popularity and track record make it a safe and reliable choice for a meta-learner.

While using models like XGBoost, SVM, Naive Bayes, and LSTM as meta-learners in ensemble modeling can provide high accuracy, these models can also be prone to **overfitting**, especially when they are trained on the predictions of multiple base models. Overfitting occurs when the model learns the noise or patterns specific to the training data that do not generalize well to unseen data. For XGBoost, although it is a powerful and highly flexible model, its ability to capture complex relationships between features can lead to overfitting, especially when the base models are already strong. When combined with a large number of features from multiple base models, the XGBoost meta-learner may excessively adjust to small fluctuations in the training data, leading to high performance on the training set but poor generalization to new data. This risk is heightened if the base models already provide overlapping or correlated predictions, which can reinforce irrelevant patterns for the meta-learner. Similarly, SVM as a meta-learner can overfit in ensemble setups. SVM, especially when using a non-linear kernel, is very sensitive to the scale of features and may become highly specialized to the training data if not properly tuned. When stacking the output of multiple base models, each with its own biases and prediction tendencies, the SVM meta-learner may overemphasize specific instances that are not representative of the general data distribution. This can lead to poor performance on new, unseen data despite strong accuracy on the training set. Naive Bayes is another model that can overfit when used as a meta-learner. Although it assumes independence between features, stacking it with predictions from base models may violate this assumption, leading to overly optimistic performance during training. Naive Bayes models are particularly sensitive to correlated features, and in an ensemble, where the base models might produce similar predictions, the meta-learner might end up fitting to these correlations too closely. This results in high training accuracy but a significant drop in performance when applied to unseen data. Finally, LSTM models, although excellent at capturing long-range dependencies in sequential data, can also overfit when used as a meta-learner. LSTMs are prone to memorizing the training data, especially when the number of parameters in the model is large relative to the size of the training set. In ensemble settings, the LSTM meta-learner might overly adapt to the specific features of the training data, particularly when the base models' predictions are very similar, reinforcing the same patterns in the data. This can cause the LSTM meta-learner to perform well on training data but fail to generalize effectively to new data.

9.12.3 Ensemble Model 3

Cross-Validation Accuracy Scores

Cross-Validation Accuracy Scores	Value
1	0.98195529
2	0.97549152
3	0.98060867
4	0.97603016
5	0.97629949
Mean Validation Accuracy	**97.81%**

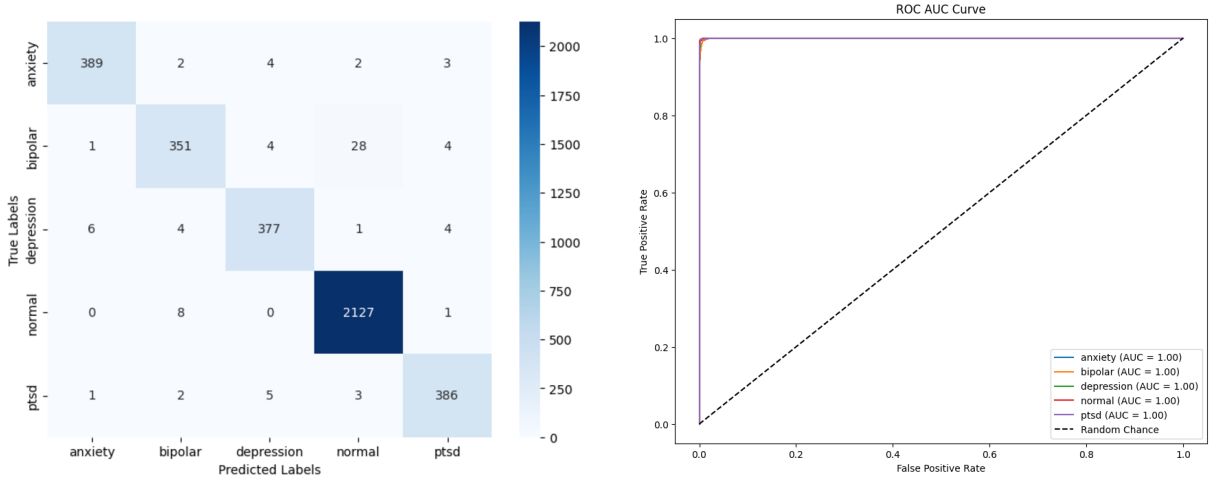
Classification Report for Ensemble Model3

	Precision	Recall	F1-Score	Support
Anxiety	0.98	0.97	0.98	400
Bipolar	0.96	0.90	0.93	388
Depression	0.97	0.96	0.96	392
Normal	0.98	1.00	0.99	2136
PTSD	0.97	0.97	0.97	397
Accuracy	**97.76%**			
Macro avg	0.97	0.96	0.97	3713
Weighted avg	0.98	0.98	0.98	3713

ROC AUC Scores

Anxiety	1.00
Bipolar	1.00
Depression	1.00
Normal	1.00
PTSD	1.00

The confusion matrix for the Random Forest model shows the performance of the classifier for each of the five mental health issues. For anxiety, there were 389 true positives, 2 false positives, 4 false negatives, and 2 misclassifications into other classes, demonstrating the model's high precision and recall for this category. Similarly, the bipolar disorder class had 351 true positives, 1 false positive, and 28 misclassifications into other classes, indicating strong but slightly lower recall compared to other categories. Depression exhibited robust performance with 377 true positives, only 6 misclassifications into other classes, and minimal false negatives. The normal class had an outstanding result with 2127 true positives, 8 false positives, and only 1 false negative, highlighting the model's exceptional accuracy for the majority class. For the PTSD class, there were 386 true positives, 5 misclassifications into other classes, and minimal false positives and negatives, showcasing excellent performance. Overall, the confusion matrix indicates that the Random Forest model is highly effective across all classes, with very few errors. The consistent performance across all classes underscores the model's ability to distinguish between different mental health conditions with high accuracy and reliability.



(a) Confusion Matrix for Ensemble Model 3

(b) ROC AUC for Ensemble Model 3

Figure 43: Evaluation Results for Ensemble Model 3

Random Forest is particularly effective at mitigating overfitting because it uses techniques like bootstrapping and random feature selection when constructing decision trees. This ensures that individual trees focus on different aspects of the data, reducing the risk of over-reliance on noise or specific patterns that might lead to overfitting. In contrast, linear models like Logistic Regression are inherently more sensitive to overfitting in high-dimensional spaces or when the data contains multicollinearity. Logistic Regression assumes a linear relationship between features and the target variable, making it less flexible in capturing complex patterns and more prone to memorizing specific data points, especially when regularization techniques are not carefully tuned. Despite experimenting with alternative configurations, such as N-Gram for XGBoost or TF-IDF for SVM, the selected ensemble model remains one of the optimal choices. The minor accuracy gains offered by the selected model are complemented by its computational efficiency, making it a practical solution for deployment. Random Forest's robust handling of overfitting and ability to generalize effectively across different inputs further solidify its suitability as the meta-learner, ensuring the web application remains both efficient and reliable in its predictive capabilities.

9.12.4 Ensemble Model 4

Cross-Validation Accuracy Scores

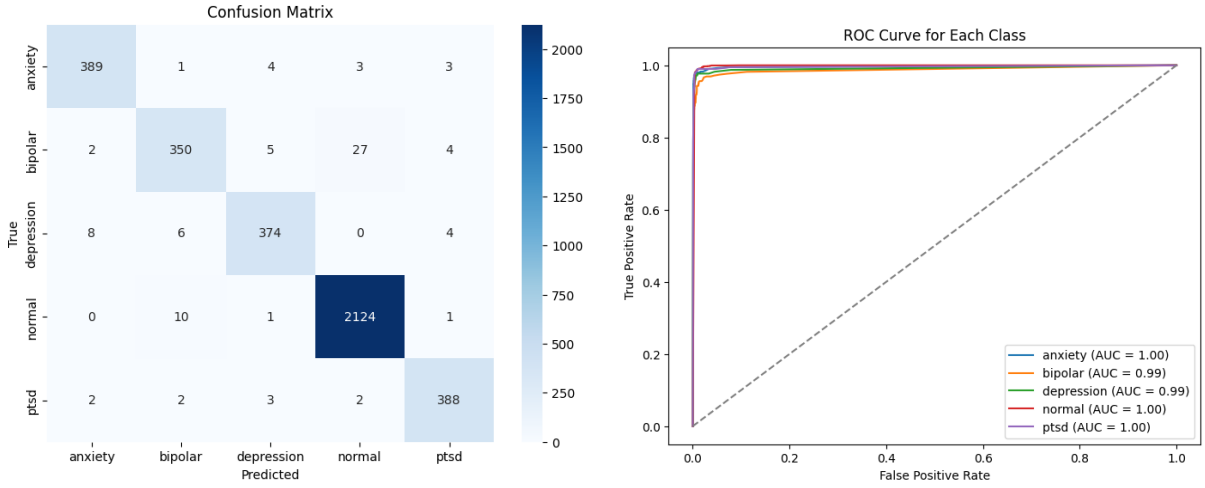
Cross-Validation Accuracy Scores	Value
1	0.97683814
2	0.96821977
3	0.97872340
4	0.96821977
5	0.97225963
Mean Validation Accuracy	**97.29%**

Classification Report for Ensemble Model4

Class	Precision	Recall	F1-Score	Support
Anxiety	0.97	0.97	0.97	400
Bipolar	0.95	0.90	0.92	388
Depression	0.97	0.95	0.96	392
Normal	0.99	0.99	0.99	2136
PTSD	0.97	0.98	0.97	397
Accuracy	**97.63%**			
Macro Avg	0.97	0.96	0.96	3713
Weighted Avg	0.98	0.98	0.98	3713

ROC AUC Scores

Class	ROC AUC Score
Anxiety	1.00
Bipolar	0.99
Depression	0.99
Normal	1.00
PTSD	1.00



(a) Confusion Matrix for Ensemble Model 4 (b) ROC AUC for Ensemble Model 4

Figure 44: Evaluation Results for Ensemble Model 4

The classification results for the Bagging model demonstrate high performance, with an accuracy of 97.63%. The classification report highlights strong precision, recall, and F1-scores across all classes, confirming the model's capability to accurately identify and classify mental health states. Notably, the "Normal" class exhibits near-perfect classification with precision, recall, and F1-scores all close to 0.99. Other classes, such as "Anxiety," "Bipolar," and "PTSD," also maintain high performance with minimal errors. The confusion matrix reveals minimal misclassifications. For example, in the "Anxiety" class, only a small number of instances are misclassified into other categories, while the "Bipolar" class shows a slightly higher degree

of misclassification but still achieves a recall of 0.90 and an F1-score of 0.92. These results emphasize the robustness of the Random Forest model in handling complex datasets.
Bagging classifiers, while highly effective at improving predictive accuracy, have a tendency to overfit the training data when used as meta-learners in ensemble models. This is due to the way bagging works, which involves training multiple base models on bootstrapped subsets of the training data and averaging their predictions. Although this method reduces variance in individual models, it can inadvertently lead to overfitting. Bagging classifiers aggregate outputs from high-variance base models, and when used as a meta-learner, the bagging process leverages outputs from already strong individual classifiers such as SVM, XGBoost, and LSTM. This additional layer of aggregation may amplify the model's capacity to memorize the training data, particularly when the meta-learner is exposed to intricate patterns that are not generalizable to unseen data. This can result in high accuracy on the training set but poor generalization to new data. The Bagging classifier can achieve an impressive accuracy, such as 99.77%, but this performance might not hold in real-world scenarios due to its reduced ability to generalize. When the model encounters slight variations in the testing dataset, its overfitting nature can cause the model to fail in maintaining consistent accuracy, leading to unreliable predictions. Bagging classifiers also do not inherently introduce the same level of regularization that Random Forest does. Random Forest uses feature-level randomness, selecting a subset of features for splitting at each node, which helps reduce overfitting by preventing any single feature from dominating the splits. Bagging classifiers, on the other hand, utilize all available features for training, which increases the likelihood of the model fitting noise in the training data, especially in high-dimensional datasets. Moreover, Bagging classifiers can be unstable and sensitive to minor changes in input data or individual base model outputs. This instability can further reduce the reliability of the model in practical applications, where consistency in predictions is crucial. In contrast, Random Forest, despite having slightly lower accuracy than the Bagging classifier, is better suited as a meta-learner. Random Forest introduces feature-level randomness, which prevents overfitting and enables the model to generalize well to unseen data. The bias-variance tradeoff is better balanced in Random Forest, which allows it to perform consistently across different datasets. Another significant advantage of using Random Forest as a meta-learner is its ability to maintain stability when retrained with new data inputs. For example, when retraining with user inputs, the accuracy fluctuations of the Random Forest meta-learner are minimal, indicating its robustness and ensuring stable performance over time. Additionally, Random Forest classifiers are computationally more efficient than Bagging classifiers, making them a more practical choice for real-time systems, such as those in web applications.

9.12.5 Ensemble Model 5

Cross-Validation Accuracy Scores

Cross-Validation Accuracy Scores	Value
1	0.96768765
2	0.97273645
3	0.97306397
4	0.96801347
5	0.96767677
Mean Validation Accuracy	**96.98%**

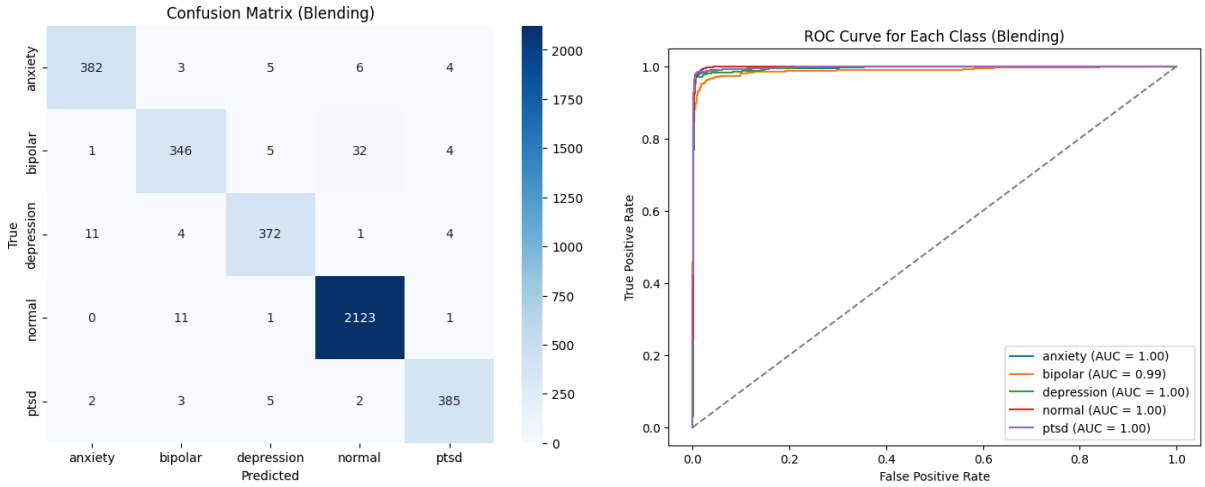
Classification Report for Ensemble Model5

	Precision	Recall	F1-Score	Support
Anxiety	0.96	0.95	0.96	400
Bipolar	0.94	0.89	0.92	388
Depression	0.96	0.95	0.95	392
Normal	0.98	0.99	0.99	2136
PTSD	0.97	0.97	0.97	397
Accuracy	**97.17%**			
Macro avg	0.96	0.95	0.96	3713
Weighted avg	0.97	0.97	0.97	3713

ROC AUC Scores

Class	AUC Score
Anxiety	1.00
Bipolar	0.99
Depression	1.00
Normal	1.00
PTSD	1.00

The Blending Meta-Learner model exhibits strong performance with a test accuracy of 97.17%. The classification report highlights solid precision, recall, and F1-scores across all classes, confirming the model's ability to accurately classify mental health conditions. For anxiety, the precision, recall, and F1-score are 0.96, 0.95, and 0.96 respectively, indicating high accuracy in classifying anxiety. The bipolar class shows a precision of 0.94 and recall of 0.89, with an F1-score of 0.92. Depression has an excellent performance with precision, recall, and F1-score of 0.96, 0.95, and 0.95, respectively. The normal class shows near-perfect classification with precision of 0.98, recall of 0.99, and F1-score of 0.99, highlighting the model's ability to accurately predict the majority class. PTSD shows a precision of 0.97, recall of 0.97, and F1-score of 0.97, reflecting strong performance. The accuracy across all classes is 97%, with the macro average F1-score being 0.96, and the weighted average F1-score being 0.97. The confusion matrix reveals minimal misclassifications, with anxiety, bipolar, and depression showing only a few misclassified instances, and the normal class showing very few errors. The ROC



(a) Confusion Matrix for Ensemble Model 5 (b) ROC AUC for Ensemble Model 5

Figure 45: Evaluation Results for Ensemble Model 5

AUC scores indicate excellent model performance in distinguishing between classes, with perfect scores of 1.00 for anxiety, depression, normal, and PTSD, and a score of 0.99 for bipolar. Cross-validation results show consistency, with accuracies ranging from 96.77% to 97.31% and a mean validation accuracy of 96.98%. The standard deviation of validation accuracy is 0.25%, demonstrating the stability of the model across different folds. These results emphasize the effectiveness and reliability of the Blending Meta-Learner in predicting mental health conditions with minimal misclassifications and high consistency. However, despite these impressive results, stacking ensemble using Random Forest as the meta-learner is often preferred over this Blending model. The main reason for this is that stacking tends to be more effective in generalizing to unseen data because it leverages multiple models for training the meta-learner, which reduces the risk of overfitting. In contrast, Blending may result in overfitting as it typically uses a simpler approach of directly training the meta-learner on the predictions from base models without incorporating a cross-validation step.

9.12.6 Ensemble Model 6

Cross-Validation Accuracy Scores

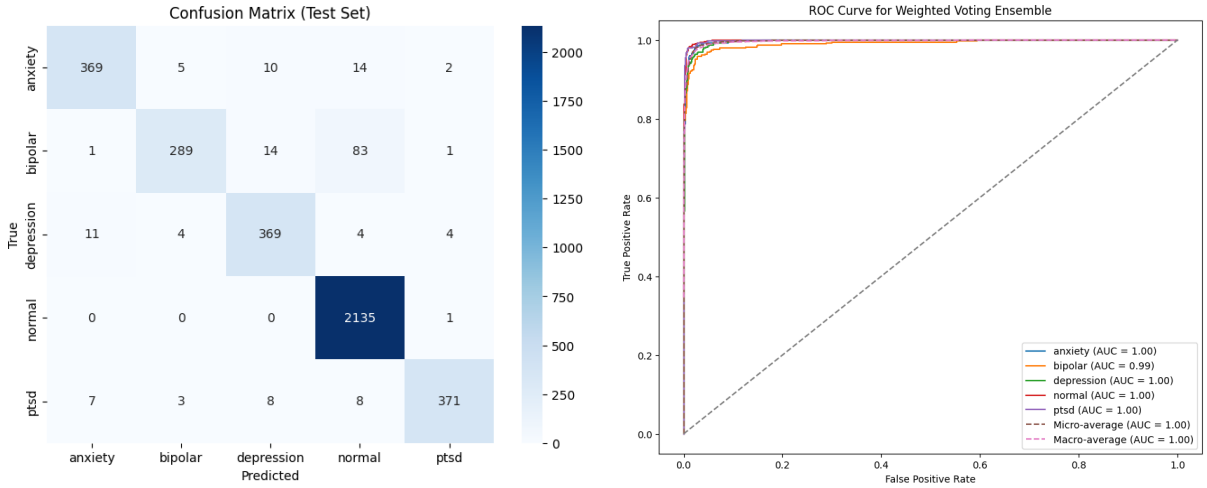
Cross-Validation Accuracy Scores	Value
1	0.9582547805009426
2	0.9477511446269863
3	0.9453272286560732
4	0.9464045246431457
5	0.9488284406140587
Mean Validation Accuracy	**94.93%**

Classification Report for Ensemble Model6

	Precision	Recall	F1-Score	Support
Anxiety	0.95	0.92	0.94	400
Bipolar	0.96	0.74	0.84	388
Depression	0.92	0.94	0.93	392
Normal	0.95	1.00	0.97	2136
PTSD	0.98	0.93	0.96	397
Accuracy	**95.15%**			
Macro avg	0.95	0.91	0.93	3713
Weighted avg	0.95	0.95	0.95	3713

ROC AUC Scores

Class	ROC AUC
Anxiety	1.00
Bipolar	0.99
Depression	1.00
Normal	1.00
PTSD	1.00



(a) Confusion Matrix for Ensemble Model 6 (b) ROC AUC for Ensemble Model 6

Figure 46: Evaluation Results for Ensemble Model 6

The Weighted Voting model achieved an accuracy of 95.15% on the test set, showcasing solid performance across multiple mental health categories. The classification report reveals that the model performed particularly well in identifying ”Normal” cases, with perfect recall (1.00) and high precision (0.95). While the ”Bipolar” class showed the lowest recall (0.74) and precision (0.96), indicating some challenges in correctly identifying bipolar disorder, other classes like ”Anxiety” and ”PTSD” exhibited strong performance with high F1-scores (0.94 and 0.96, respectively). The confusion matrix further highlights a few misclassifications, with a notable

number of "Bipolar" instances misclassified as other classes, particularly "Normal" and "Depression." The ROC AUC score of 0.99 for PTSD indicates that the model performed exceptionally well in distinguishing PTSD cases from the rest, though other classes had lower AUC values. Overall, the Weighted Voting model effectively handled the classification task with robust performance for most classes, but some room for improvement remains in distinguishing between certain mental health issues, particularly bipolar disorder.

While the Weighted Voting ensemble model shows a respectable accuracy of 94.93% and performs well across all classes, it is not preferred over other models like the Stacking ensemble with Random Forest as the meta-learner. The primary reason for this is that the Weighted Voting model does not fully take advantage of the strength of individual base models in the same way as the Stacking ensemble does. In Weighted Voting, the predictions of base models are combined based on their weighted contribution, but this method does not allow for the optimal learning of inter-model relationships that Stacking does. In the Stacking model, the Random Forest meta-learner is specifically designed to handle interactions between different base models, learning how to combine their strengths in a way that minimizes overfitting while maintaining high accuracy. In contrast, Weighted Voting simply assigns a weight to each model based on its performance, without a deeper mechanism to adjust for overfitting or to learn from the diversity of base models. This lack of fine-tuning in model interaction limits its ability to generalize as effectively as Stacking. Furthermore, the Weighted Voting model's performance may suffer when dealing with more complex relationships between features, as it relies on the simplicity of averaging the weighted predictions. While it is computationally efficient, it does not provide the robustness that Stacking offers through a more sophisticated meta-learning approach. As a result, despite the relatively high accuracy, the Weighted Voting ensemble model is considered less optimal for the given task when compared to more powerful models like Stacking.

9.12.7 Ensemble Model 7

Cross-Validation Accuracy Scores

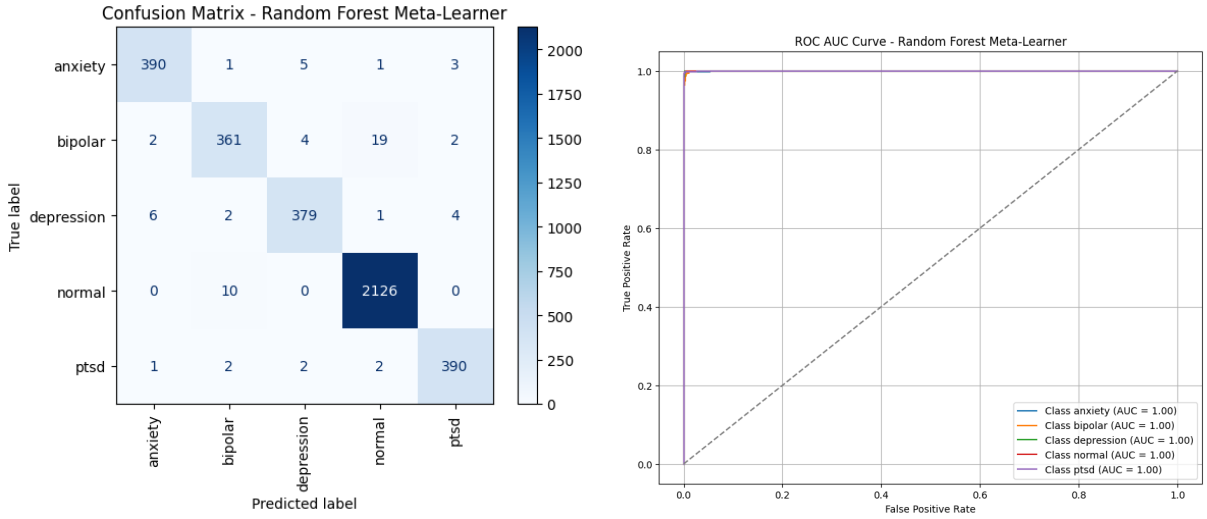
Cross-Validation Accuracy Scores	Value
1	0.980878
2	0.97899273
3	0.98410988
4	0.98168597
5	0.97764611
Mean Validation Accuracy	**98.03%**

Classification Report for Ensemble Model7

	Precision	Recall	F1-Score	Support
Anxiety	0.98	0.97	0.98	400
Bipolar	0.96	0.93	0.95	388
Depression	0.97	0.97	0.97	392
Normal	0.99	1.00	0.99	2136
PTSD	0.98	0.98	0.98	397
Accuracy	**98.20%**			
Macro avg	0.98	0.97	0.97	3713
Weighted avg	0.98	0.98	0.98	3713

ROC AUC Scores

Class	ROC AUC Score
Anxiety	1.00
Bipolar	1.00
Depression	1.00
Normal	1.00
PTSD	1.00



(a) Confusion Matrix for Ensemble Model 7 (b) ROC AUC for Ensemble Model 7

Figure 47: Evaluation Results for Ensemble Model 7

The ensemble model, which leverages a Transformer-based model as the base learner along with Logistic Regression, Naive Bayes, SVM, LSTM, XGBoost, and Random Forest as the meta-learner, has achieved an impressive accuracy of 98.03% on the test set. The classification report indicates high precision, recall, and F1-scores across all classes, with the model performing particularly well in detecting ”Normal” and ”PTSD” cases, where precision and recall are both very high. The confusion matrix reveals minimal misclassifications, further demonstrating the effectiveness of the ensemble approach in handling complex classification tasks. The

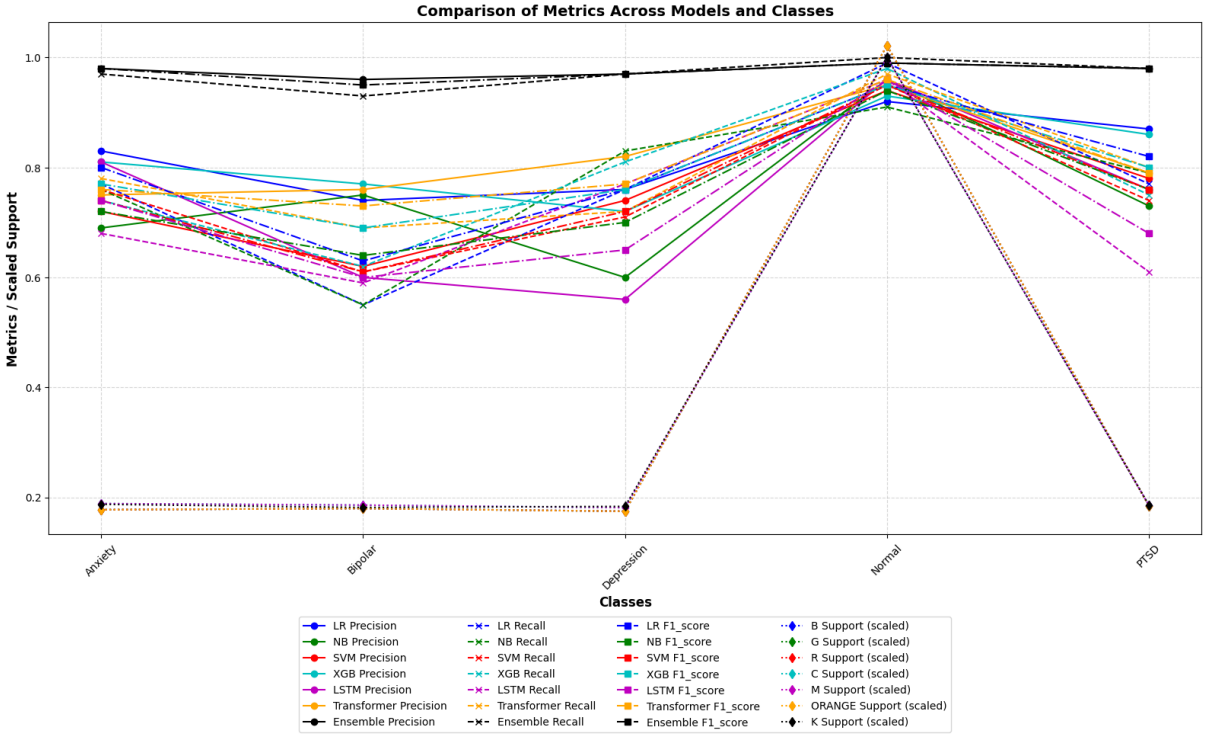


Figure 48: Comparison of Base Models and Ensemble Model7

perfect ROC AUC scores of 1.00 across all classes suggest that the model is able to distinguish between different mental health conditions with near-perfect specificity and sensitivity. This robust performance highlights the advantage of combining multiple models through ensemble learning, where the strengths of each model contribute to the overall improvement in accuracy.

Incorporating a custom Transformer-based model as part of the ensemble brings significant advantages in the classification task. Transformer models are particularly effective in handling sequential data due to their self-attention mechanism, which allows the model to focus on important features and long-range dependencies within the input data. This makes them particularly suited for tasks like natural language processing or time-series classification, where context plays a crucial role. By integrating a Transformer as a base model in an ensemble, the model can capture complex patterns and relationships that other models, such as Logistic Regression or Naive Bayes, may overlook. The ensemble approach benefits from the Transformer's ability to learn sophisticated representations, while the meta-learner, in this case, Random Forest, effectively combines the outputs of various base models to enhance overall performance. This combination not only boosts accuracy but also improves the robustness and generalization capabilities of the model, making it more reliable for real-world applications. The Transformer architecture is fundamentally different from traditional models due to its self-attention mechanism, which allows the model to weigh the importance of different tokens in a sequence regardless of their distance from each other. Unlike recurrent neural networks (RNNs) or long

short-term memory (LSTM) networks, which process data sequentially, Transformers are capable of processing entire sequences in parallel, which significantly speeds up training. In a Transformer, each input token is transformed into an embedding, and the self-attention mechanism computes a weighted sum of all tokens, allowing the model to capture contextual relationships between distant elements. This allows the Transformer to learn both local and global patterns effectively. For classification tasks, this ability to understand long-range dependencies in the data is crucial. For example, in text classification, words in a sentence may carry meaning only when considered in the context of other words. A Transformer can efficiently model these relationships, regardless of how far apart the words are in the input sequence. In an ensemble, the Transformer's ability to learn complex, high-dimensional feature representations complements the more straightforward models, which may not be able to capture such intricate patterns. When used as a base model, the Transformer generates rich feature vectors, which are then processed by the meta-learner. In this case, Random Forest, a robust ensemble method, takes the output of the base models, including the Transformer, and uses it to make the final classification decision. The meta-learner helps in reducing overfitting and improving generalization by aggregating the predictions of multiple models, leading to better performance across different data distributions. By leveraging the strengths of the Transformer's ability to understand complex relationships and the Random Forest's ability to combine multiple models, the ensemble approach becomes highly effective in complex classification tasks.

Below is a comparison of all the ensemble models for reference

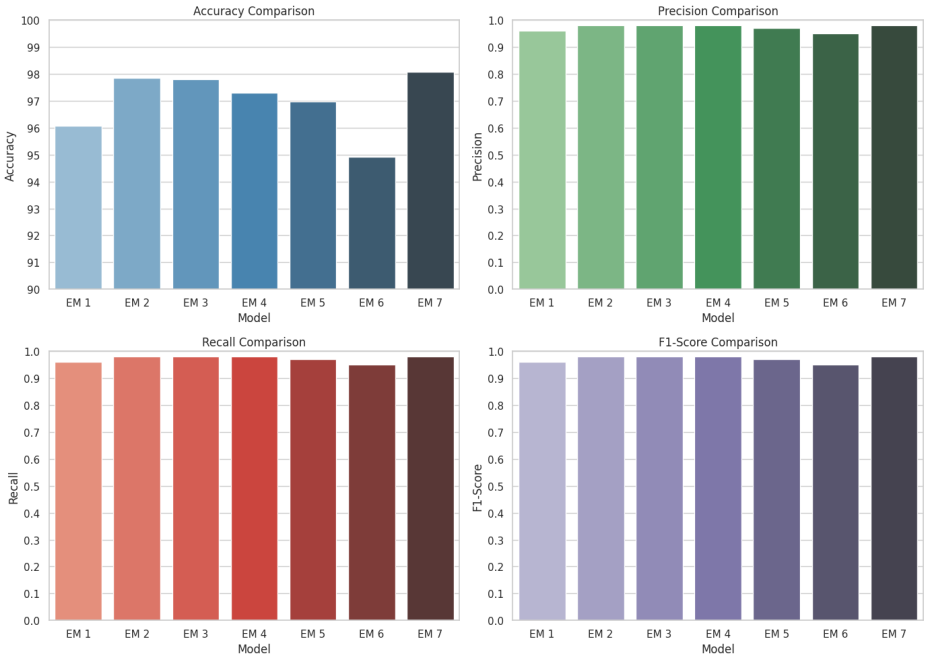


Figure 49: Comparison of all Ensemble Models

9.13 Result from hierarchical Ensemble Models

Performance Comparison of Models

Model	Subset 1	Subset 2	Subset 3	Subset 4	Subset 5	Subset 6	ENSEMBLE
Logistic Regression	87.66	85.07	86.74	90.61	88.17	72.62	**90.90**
Naive Bayes	83.63	81.71	83.50	84.44	85.93	64.33	**84.15**
SVM	85.13	82.30	83.50	88.34	85.45	67.81	**91.79**
XGBoost	87.39	85.79	86.78	91.37	86.32	70.95	**67.67**
LSTM	84.91	81.63	81.83	87.22	85.38	67.74	**87.48**
Transformer	88.50	84.50	86.50	89.35	87.62	72.40	**91.53**
ENSEMBLE	**98.03**	**94.85**	**96.96**	**97.79**	**96.86**	**92.57**	**96.24** / **96.25**

Note: The final ensemble models from two different architectures achieved accuracies of **96.24%** and **96.25%** respectively.

The hierarchical ensemble model implemented in this project addresses the challenge of handling large datasets effectively by leveraging a modular and scalable approach. The dataset, consisting of millions of records, is divided into manageable subsets, with each subset treated as an independent data unit. For each subset, multiple base models such as Logistic Regression, Naive Bayes, SVM, LSTM, XGBoost, and Custom Transformers are trained and tested. These base models are then combined into a subset-specific ensemble using Random Forest as the meta learner. This process is repeated for all subsets, resulting in several independent ensemble models. Finally, the outputs from these subset-specific ensembles are aggregated to form a final ensemble model, representing the comprehensive learning from the entire dataset. This approach not only ensures better performance but also provides a framework for handling large-scale data efficiently. While the current implementation follows a fully sequential approach for testing and validation purposes, the design is inherently scalable to a distributed system architecture. In a distributed setup, the dataset can be partitioned and processed in parallel, with each subset assigned to a separate computational node or worker. Each node trains base models, creates an ensemble for its subset, and transmits the results to a central controller. The controller aggregates the subset-specific ensembles into a global ensemble model, ensuring a streamlined process that benefits from parallelism and modularity. This architecture mirrors distributed computing principles, making it highly suitable for large-scale deployments and real-world applications.

The hierarchical structure of this approach aligns with distributed systems by effectively implementing data parallelism and task parallelism. By dividing the dataset into subsets, the computational load is distributed, preventing memory bottlenecks and ensuring resource optimization. Each subset is processed independently, enabling a modular design where tasks are decoupled. This design also enhances fault tolerance, as the failure of a single subset's processing does not affect the others. Moreover, the scalability of the approach allows it to adapt

to varying resource constraints. Adding more computational nodes makes it feasible to handle additional subsets or process the data faster, showcasing its adaptability and efficiency. The project also emphasizes the potential of this design to be implemented on distributed computing frameworks such as Apache Spark, TensorFlow Distributed, or PyTorch Distributed. These frameworks provide tools to split datasets, distribute tasks, and aggregate results seamlessly across multiple nodes. The distributed nature of the architecture ensures high throughput and reduces training time, making it suitable for scenarios involving large-scale datasets. The hierarchical ensemble model design is a future-ready solution that can transition seamlessly to distributed platforms, leveraging cloud services or on-premises high-performance computing clusters. The methodology combines the strengths of ensemble learning with the efficiencies of distributed systems to create a robust and scalable solution for modern data science challenges.

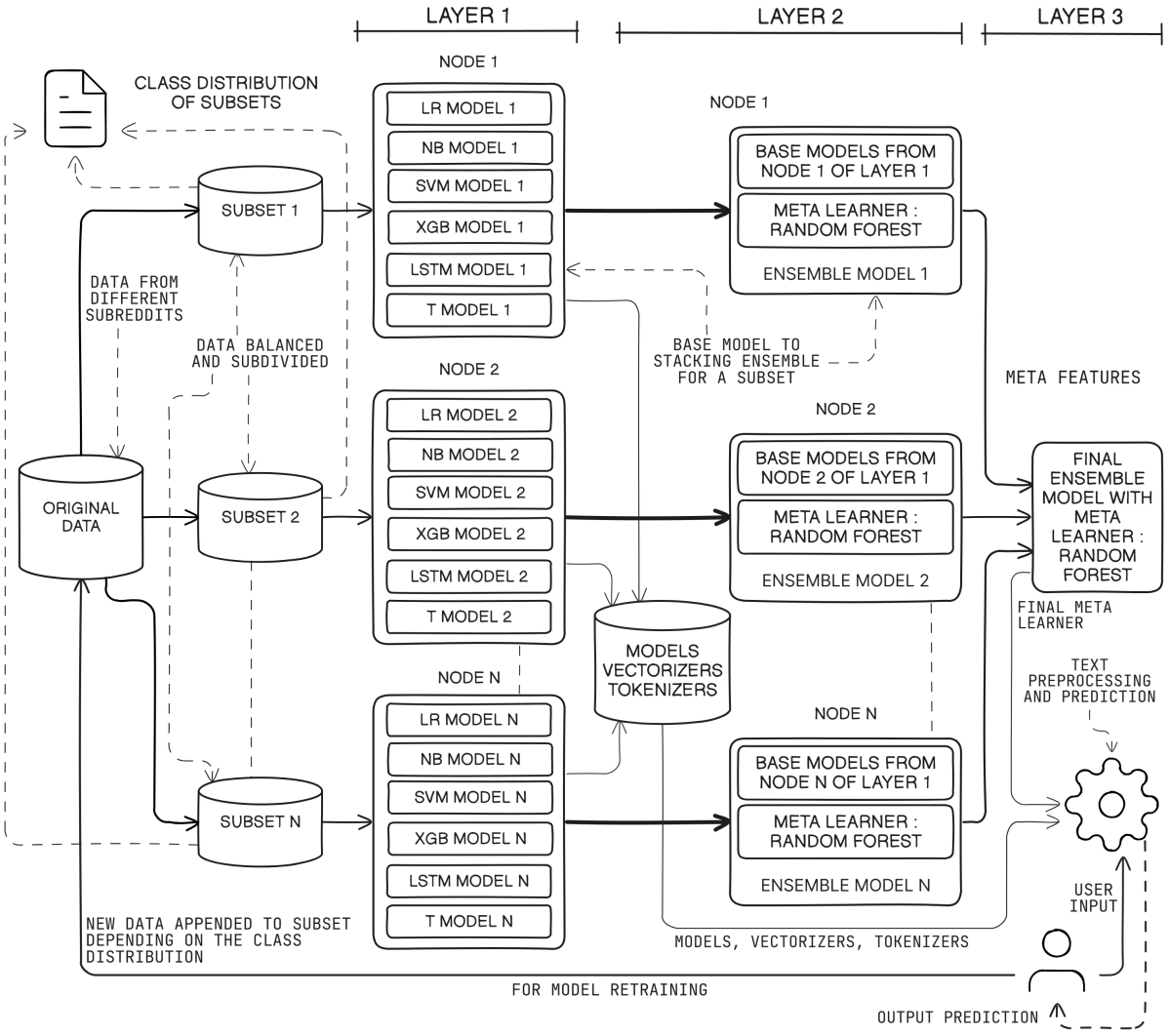


Figure 50: Scalable Distributed Architecture 1

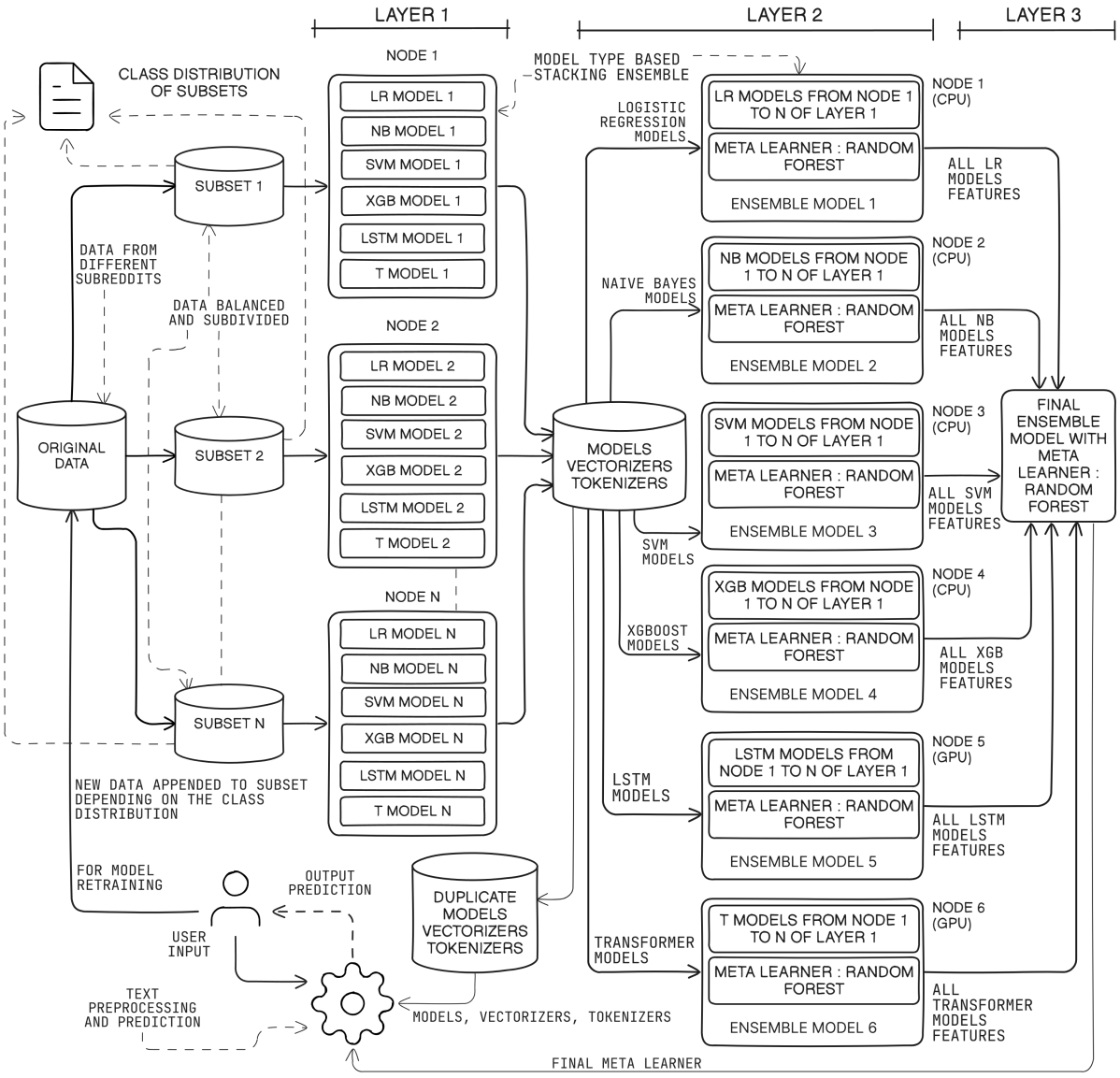


Figure 51: Scalable Distributed Architecture 2

The second architecture optimizes the hierarchical ensemble approach by restructuring the model aggregation process to enhance scalability and computational efficiency. Instead of creating ensemble models for each of the n subsets independently, models of the same type from all subsets are combined into a single ensemble for that type using Random Forest (or Logistic Regression) as the meta learner. This results in six intermediate ensemble models, one for each model type: Logistic Regression, Naive Bayes, SVM, XGBoost, LSTM, and Transformer. These intermediate ensemble models are then used to form the final ensemble model, where Random Forest acts as the meta learner. This architectural adjustment significantly reduces the number of intermediate ensemble models from n to six while preserving the hierarchical structure, thereby maintaining a consistent three-layer design. The second architecture achieved an

accuracy of 96.25%. However, its design provides distinct advantages when dealing with much larger datasets or a higher number of subsets. As the dataset size increases, the number of subsets and computational nodes may also grow. In the previous architecture, this scaling would lead to a proportional increase in the number of intermediate ensemble models, which could strain computational resources and introduce inefficiencies in model aggregation. In contrast, the second architecture fixes the number of intermediate ensemble models at six, irrespective of the number of subsets, ensuring that the computational load remains manageable while scalability is maintained. Another significant improvement lies in the efficient use of hardware resources. In the second architecture, GPUs are utilized exclusively for training the LSTM and Transformer-based ensemble models, rather than being partially engaged across all intermediate models as in the first architecture. This selective usage of GPUs streamlines resource allocation, ensuring that the most computationally intensive models benefit from accelerated hardware while reducing unnecessary overhead for other model types. The use of Logistic Regression as the meta learner in the intermediate ensemble models further contributes to this efficiency. Logistic Regression is computationally lightweight and linear, making it an excellent choice for aggregating predictions across subsets without introducing excessive complexity. Its linearity ensures faster training while maintaining strong predictive performance, particularly for well-separated data classes. Random Forest as a meta-learner is a better option for considering overfitting issues. A key addition to the second architecture is the implementation of an efficient data bus for transferring models of the same type from different subsets to the respective intermediate nodes responsible for creating the type-specific ensemble. This structured flow of data and models ensures seamless communication between subsets and intermediate nodes, reducing latency and synchronization overhead. By fixing the number of intermediate ensemble models at six, the architecture limits the number of aggregation layers while still leveraging the diversity and complementary strengths of all base models. This design strikes a balance between computational efficiency and predictive performance, making it particularly advantageous for large-scale datasets. The second architecture's streamlined design, efficient resource utilization, and fixed number of intermediate ensemble models make it a robust and scalable solution for modern data science challenges. By addressing the potential bottlenecks of the first architecture while maintaining comparable performance, it provides a practical alternative for real-world applications where scalability and computational efficiency are critical.

Architecture 1 takes more time in cross-validation compared to Architecture 2 due to its decentralized design. In Architecture 1, each subset independently creates its intermediate ensemble model, resulting in a larger number of models being trained and validated. This increases the computational workload and requires more synchronization across subsets. On the other hand, Architecture 2 aggregates models of the same type into a single intermediate ensemble model, reducing the number of intermediate ensembles. This streamlining minimizes redundancy and

allows for faster cross-validation while maintaining predictive performance. In Architecture 2, models of the same type are used as base models in the intermediate ensemble to ensure consistency and simplify the aggregation process. Mixing different types of models, such as Logistic Regression from one subset and SVM from another, would introduce heterogeneity in predictions, making the meta-learning process more complex. By grouping models of the same type, the meta-learner can better capture specific patterns and variations unique to each model type. This design also ensures that the strengths of each algorithm are utilized optimally, allowing the final ensemble to benefit from their complementary capabilities. The observation that SMOTE is not required and yields the same results as stratified K-fold cross-validation indicates that the dataset is either balanced or sufficiently robust to handle minor class imbalances. Stratified K-fold cross-validation ensures that each fold maintains the same class distribution as the original dataset, effectively mitigating the impact of class imbalance during training and testing. This suggests that the dataset's inherent balance, combined with the model's robustness, eliminates the need for synthetic oversampling techniques like SMOTE. Logistic Regression with L1 or L2 regularization is less suited for Architecture 2 because it takes more time to train on the larger, aggregated datasets at the intermediate level. Regularization adds computational overhead by optimizing penalty terms alongside the model's weights, which can slow down training. Furthermore, Logistic Regression may struggle with the diverse, high-dimensional feature space created by aggregated predictions from multiple subsets. Random Forest, as a meta-learner, is a better choice in Architecture 2 because it is less prone to overfitting, can capture nonlinear relationships, and handles high-dimensional data effectively without extensive parameter tuning. Its ensemble-based nature ensures better generalization, making it more efficient and reliable for intermediate and final aggregation.

To determine n, the number of subsets, based on the size of the original dataset, it is crucial to consider computational efficiency, memory constraints, and sequential execution. Given that the dataset has $D = 167,229$ records and is processed sequentially in Google Colab with 12GB of RAM per node, the number of subsets n must strike a balance between memory usage and model performance. In this case, you chose $n = 6$ subsets, which implies a subset size S of:

$$S = \frac{D}{n} = \frac{167,229}{6} \approx 27,872 \text{ records per subset.}$$

The choice of $n = 6$ is reasonable given the following factors:

1. **Memory Constraints**: Google Colab provides 12GB of RAM. Each subset must fit within this memory while accommodating the model's requirements for training and validation. Processing approximately 27,833 records at a time is well-suited to this memory limit for most

machine learning models, including Logistic Regression, SVM, LSTM, and Transformer-based models.
2. **Sequential Execution**: Since the architectures are implemented sequentially, the number of subsets n does not need to align with the number of computational nodes. Instead, the goal is to divide the dataset into manageable chunks that reduce training time and memory overhead for each subset.
3. **Performance and Aggregation**: With $n = 6$, both architectures remain computationally feasible. In Architecture 1, $n = 6$ leads to six independent intermediate ensemble models, which are aggregated into the final ensemble. In Architecture 2, models of the same type from all six subsets are combined into six intermediate ensemble models, one for each base model type.

The choice of $n = 6$ also ensures that the dataset is sufficiently divided to prevent memory bottlenecks while allowing each subset to contribute meaningful insights to the ensemble. Larger values of n would reduce the subset size but increase the number of models to train, leading to longer execution times. Conversely, smaller values of n would increase the subset size, risking memory overflow in Colab. Here, the RAM usage per record is approximated based on empirical testing, and n can be dynamically adjusted if needed. Given that $S \approx 27,833$ records per subset fits comfortably within 12GB of RAM, the choice of $n = 6$ is both optimal and practical.

10 Conclusion

10.1 Project Benefits

The project on detecting mental health disorders through social media analysis offers a wide array of significant benefits, both immediate and long-term, across multiple dimensions. First and foremost, it addresses a critical issue in mental health care—early detection and intervention. Social media has become a ubiquitous platform where people express their thoughts, feelings, and emotional states, often unconsciously. By leveraging the vast amounts of data available on social media platforms, our project seeks to tap into this resource to identify early signs of mental health disorders such as anxiety, depression, and more severe conditions like bipolar disorder or schizophrenia. The ability to detect mental health issues through real-time social media data is a game-changer for public health systems, mental health practitioners, and even individuals who may not realize they are at risk. Early detection enables timely intervention, reducing the overall burden of mental health disorders on society by preventing escalation into more severe conditions that often lead to hospitalization, self-harm, or even suicide. In this sense, the project aligns with global health initiatives that emphasize early diagnosis and preventive care.

Moreover, this project holds significant potential for improving the accuracy and efficiency of mental health diagnostics. Traditional diagnostic methods are often time-consuming, subjective, and reliant on self-reporting, which can lead to underdiagnosis or misdiagnosis. By utilizing machine learning algorithms and natural language processing techniques, our project automates the process of sentiment and behavioral analysis on social media platforms, offering a more objective and data-driven approach. This automated system can process large volumes of data much faster than human professionals, providing insights that would be impossible to glean from manual analysis. The algorithms developed as part of this project can be easily scaled to analyze millions of social media posts, enabling a broader reach in monitoring public mental health trends. Additionally, the project offers practical benefits for mental health professionals, allowing them to focus on treatment and intervention rather than diagnosis. It provides a tool that can be integrated into telehealth systems, offering mental health screening at scale, which is particularly valuable in underserved or rural areas where access to mental health professionals is limited.

From a technological standpoint, the project offers a host of reusable components and methodologies. The machine learning models developed, the sentiment analysis tools, and the overall data pipeline are designed to be scalable and modular. These components can be adapted and extended to other domains beyond mental health, such as market sentiment analysis, public opinion monitoring, or even detecting harmful behavior like cyberbullying and harassment online. By advancing the state of the art in social media analytics, this project contributes to the

growing field of AI-driven health care solutions. Furthermore, it provides a blueprint for future interdisciplinary work that integrates data science, psychology, and public health.

10.2 Future Scope for Improvements

While this project offers numerous immediate benefits, there is substantial room for future enhancements that can broaden its applicability, accuracy, and effectiveness. Currently, the project focuses on analyzing Reddit data using PRAW, which is limited to a specific social media platform and dataset. In the future, incorporating data from other platforms like Facebook, Instagram, Twitter, and even niche forums could provide a more comprehensive understanding of an individual's mental health status. Different platforms cater to different demographics and social behaviors, and expanding the dataset will allow for a more holistic analysis of mental health indicators across various user bases. Additionally, expanding the dataset to include multilingual posts or integrating language translation capabilities could make the system applicable to a global audience, helping to identify mental health issues in non-English speaking populations.

Moreover, future improvements could focus on integrating real-time data analysis capabilities. Currently, our project is based on batch processing of historical data. However, in future iterations, the system could be developed to perform real-time monitoring, offering immediate feedback and potentially alerting health professionals or loved ones when someone shows signs of mental distress. This real-time capability would be invaluable in emergency situations, allowing for immediate intervention. Developing a mobile application or a web-based interface where users can voluntarily connect their social media accounts to monitor their mental health status could also increase user engagement and provide individuals with direct feedback on their well-being.

Another significant future enhancement could involve incorporating ethical considerations and improving user privacy. As mental health is a sensitive subject, ensuring that the system is designed with robust privacy protections is critical. Future work could focus on using differential privacy or other anonymization techniques to ensure that user data remains confidential while still allowing for effective analysis. Moreover, collaborating with psychologists, ethicists, and legal experts could help refine the system to ensure it adheres to ethical guidelines and avoids potential harm, such as misdiagnosis or privacy violations.

Lastly, the future scope of this project could include expanding its use in clinical settings. While the current system is primarily designed as a research tool, future iterations could be developed in collaboration with mental health professionals to ensure that it meets clinical standards. In conclusion, the hierarchical ensemble model demonstrates an innovative and scalable approach

to handling large datasets by combining the strengths of ensemble learning and distributed systems principles. By dividing the data into subsets, training independent models, and aggregating their outputs, the approach ensures efficiency, modularity, and adaptability. While the current implementation is sequential, its design is inherently distributed, making it future-ready for large-scale deployments. This framework not only addresses computational challenges effectively but also establishes a foundation for leveraging advanced distributed computing technologies to enhance performance and scalability in real-world applications.

Probable Method for implementation of Distributed Architecture

The dataset is divided into subsets for parallel processing in a distributed system. Data partitioning can be performed using distributed data storage systems such as Hadoop HDFS, Amazon S3, or Google Cloud Storage. Tools like Apache Spark or Dask are particularly effective for splitting datasets into logical subsets based on criteria such as size, record count, or specific data attributes like time ranges, regions, or categories. Each subset is then assigned to a worker node for processing. Alternatively, data sharding can be used at the database level, where SQL queries or shard keys in databases like MongoDB or Cassandra create logical partitions of the dataset. Each worker node is responsible for training base models on its assigned subset. Worker nodes are typically implemented using containers such as Docker or virtual machines, each equipped with necessary machine learning frameworks like Scikit-learn, TensorFlow, or PyTorch. A central master node, acting as the orchestrator, distributes tasks such as training models like Logistic Regression, SVM, and others to these workers. Frameworks such as Apache Spark MLlib or TensorFlow Distributed Strategy facilitate this process. Each worker independently processes its subset, training multiple base models in parallel, while periodically reporting training statuses back to the master node. After training the base models for each subset, a subset-specific ensemble is created. This is typically done by employing a meta learner like Random Forest to aggregate the outputs of the base models within each worker. Frameworks like Scikit-learn or XGBoost can be utilized locally on the workers for this purpose. The trained subset-specific ensembles are then saved to distributed storage systems such as S3 or HDFS for later aggregation into a final ensemble model. To create the final ensemble model, predictions or model weights from all subset-specific ensembles are collected by the master node. The aggregation process can use techniques such as weighted averaging or stacking, implemented through another meta learner. In cases where the number of subset-specific ensembles is large, the aggregation itself can be distributed using paradigms like MapReduce. The final model is saved to shared storage to facilitate deployment. Once the final ensemble model is created, it is deployed using distributed inference platforms like TensorFlow Serving, Kubernetes, or AWS SageMaker, ensuring scalability by serving the model across multiple instances. Monitoring tools such as Prometheus or Grafana are employed to track the performance and resource utilization of the

distributed system, ensuring reliable operation. For implementing such a system, various technologies are available. Distributed computing frameworks like Apache Spark or Dask are ideal for parallel data processing and machine learning. TensorFlow Distributed Strategy or PyTorch Distributed are suitable for scalable training of deep learning models. Task orchestration can be managed using tools like Apache Airflow or Prefect, while Kubernetes is used for container orchestration and scaling worker nodes. Storage and sharding can be handled by systems like Hadoop HDFS, Amazon S3, or databases like Cassandra and MongoDB. For managing communication between nodes, message-passing systems such as RabbitMQ or Kafka are employed. Network overhead must be minimized to avoid communication bottlenecks between the master node and workers. Fault tolerance is critical, and distributed frameworks with built-in recovery mechanisms should be used.

The implementation of threading in the web application presents a significant opportunity for future improvements, particularly in enhancing the scalability and efficiency of the system. Threading can address several real-world challenges by allowing the application to handle multiple tasks concurrently, thereby improving responsiveness and reducing bottlenecks. Functions involving heavy I/O operations, such as downloading videos or images, extracting audio, or transcribing speech, can be threaded to overlap network and disk operations. This would minimize the idle time caused by waiting for these tasks to complete, allowing the system to serve multiple user requests simultaneously. Furthermore, tasks related to data processing, such as extracting frames from videos, detecting emotions from images, or analyzing audio mood, can also benefit from threading. These operations, while computationally intensive, often involve processing independent data units such as individual video frames or audio segments. By distributing these tasks across multiple threads, the workload can be parallelized, significantly reducing the processing time for large datasets. For model inference tasks like image captioning, text classification, or emotion analysis, threading can be employed to execute predictions on different inputs concurrently. This is particularly advantageous when the application processes multiple user inputs or large batches of data. By leveraging threading, the application can scale more effectively, accommodating higher user demand without compromising performance. In a real-world scenario, threading can also enhance the system's ability to handle simultaneous user interactions, a critical factor for a scalable web application. For instance, concurrent video downloads or transcription tasks can ensure faster response times for users accessing the same services. Similarly, threading image analysis or captioning workflows can support batch processing while maintaining system responsiveness.

11 References

[1] Hatoon S AlSagri and Mourad Ykhlef. Machine learning-based approach for depression detection in twitter using content and activity features. *IEICE Transactions on Information and Systems*, 103(8):1825–1832, 2020.

[2] Munmun De Choudhury, Michael Gamon, Scott Counts, and Eric Horvitz. Predicting depression via social media. *Proceedings of the International AAAI Conference on Web and Social Media*, 2013.

[3] Jetli Chung and Jason Teo. Single classifier vs. ensemble machine learning approaches for mental health prediction. *Brain Informatics*, 10(1):1, jan 2023.

[4] Sharath Chandra Guntuku, David Bryce Yaden, Margaret L. Kern, Lyle H. Ungar, and Johannes C. Eichstaedt. Detecting depression and mental illness on social media: an integrative review. *Current Opinion in Behavioral Sciences*, 18:43–49, 2017.

[5] Priya Mathur, Amit Kumar Gupta, and Abhishek Dadhich. Mental health classification on social-media: Systematic review. *Proceedings of the 4th International Conference on Information Management & Machine Intelligence*, 2022.

[6] Moin Nadeem. Identifying depression on twitter. *arXiv preprint arXiv:1607.07384*, 2016.

[7] Ramin Safa, S. A. Edalatpanah, and Ali Sorourkhah. Predicting mental health using social media: A roadmap for future development, 2023.

[8] Dip Kumar Saha, Tuhin Hossain, Mejdl Safran, Sultan Alfarhood, M. F. Mridha, and Dunren Che. Ensemble of hybrid model based technique for early detecting of depression based on svm and neural networks. *Scientific Reports*, 14(1):25470, oct 2024.

[9] Konda Vaishnavi, U Nikhitha Kamath, B Ashwath Rao, and N V Subba Reddy. Predicting mental health illness using machine learning algorithms. *Journal of Physics: Conference Series*, 2161(1):012021, jan 2022.

[10] Hongzhi Zhang and M. Omair Shafiq. Survey of transformers and towards ensemble learning using transformers for natural language processing. *Journal of Big Data*, 11(1):25, feb 2024.

APPENDIX A - Prototype

System Dependencies Installation

```
!apt-get install -y ffmpeg libsm6 libxext6
!apt-get install -y tesseract-ocr
!apt-get install -y portaudio19-dev
```

This code installs essential system-level dependencies required for multimedia processing, Optical Character Recognition (OCR), and audio manipulation tasks. The ffmpeg library provides robust capabilities for processing video and audio data, supporting operations like format conversion, compression, and extraction. The libsm6 and libxext6 libraries are X Window System components needed for graphical processing and enabling compatibility with multimedia frameworks like OpenCV. The tesseract-ocr package installs Tesseract, a powerful OCR engine used for extracting text from images and documents. Lastly, portaudio19-dev is a development package for the PortAudio library, which enables cross-platform audio processing and is often required for speech recognition and audio streaming applications. These installations ensure that the environment is properly configured to handle complex data processing workflows.

Import Libraries

```
import streamlit as st
import joblib
import pandas as pd
import praw
from PIL import Image
from deep_translator import GoogleTranslator
import requests
from io import BytesIO
from collections import Counter
import google.generativeai as genai
import cv2
import numpy as np
import whisper
import tempfile
import os
from pydub import AudioSegment
import subprocess
import re
import librosa
import librosa.display
import tensorflow as tf
import pytesseract
```

This code snippet includes various library imports essential for building a multi-functional application that processes and analyzes multimedia data, text, and machine learning tasks. Streamlit is used for building interactive web apps. Joblib assists with object serialization, and pandas handles structured data manipulation. PRAW enables interaction with Reddit's API, while Pillow facilitates image processing. GoogleTranslator supports text translation, and requests fetches data from web sources. BytesIO handles in-memory byte streams, and collections.Counter aids in counting elements in datasets. Google Generative AI tools are also incorporated for advanced AI tasks. For media processing, OpenCV and NumPy handle image and video manipulations. Whisper is an ASR (Automatic Speech Recognition) tool, and tempfile manages temporary files. Pydub processes audio, and subprocess executes shell commands. Re provides regular expressions for text parsing, Librosa supports audio feature extraction and visualization, and TensorFlow enables building machine learning models. Lastly, pytesseract extracts text from images using OCR (Optical Character Recognition), completing a robust toolset for diverse computational tasks.

Configuration and Model Initialization

```
# Configure Tesseract and FFMPEG
pytesseract.pytesseract.tesseract_cmd = '/usr/bin/tesseract'
os.environ["FFMPEG_BINARY"] = "/usr/bin/ffmpeg"
# Load Whisper model for audio transcription
whisper_model = whisper.load_model("base")
# Load the saved logistic regression model and vectorizer
model = joblib.load('LRmodel.pkl')
vectorizer = joblib.load('LRvectorizer.pkl')
# Initialize Reddit API
reddit = praw.Reddit(client_id='<CLIENT_ID>',
client_secret='<CLIENT_SECRET_KEY>',user_agent='Mental␣Health')
# Configure Gemini API for wellbeing insights
genai.configure(api_key="<GEMINI_API_KEY>")
generation_config = {
    "temperature": 1, "top_p": 0.95, "top_k": 40,          "
       max_output_tokens": 8192, "response_mime_type": "text/
       plain",
}
gemini_model = genai.GenerativeModel(
    model_name="gemini-1.5-flash",        generation_config=
       generation_config,
)
```

This code initializes and configures various tools and models for a complex data-processing pipeline. The Tesseract OCR engine is configured by specifying its executable path to enable text extraction from images. The FFMPEG binary is similarly set to facilitate multimedia

processing tasks. The Whisper model, a speech-to-text solution, is loaded with its base configuration for audio transcription. Saved machine learning artifacts, a logistic regression model, and a vectorizer are loaded using joblib, providing a pre-trained setup for text classification tasks. The Reddit API is initialized using PRAW, with credentials to interact with Reddit's platform for data retrieval. For generative AI tasks, the Gemini API is configured with an API key and generation parameters such as temperature, top-p sampling, and maximum output token limit. A generative model is instantiated using these configurations, designed to provide wellbeing insights. This setup creates a cohesive framework for multimedia processing, natural language processing, and machine learning applications.

Fetching Reddit User Text Posts

```
# Function to fetch text-based posts from Reddit
def fetch_user_text_posts(username):
    try:
        user = reddit.redditor(username)
        posts = [post.title + "␣" + post.selftext for post in
           user.submissions.new(limit=20)]
        return posts
    except Exception as e:
        st.write(f"Error␣fetching␣text␣posts:␣{e}")
        return []
```

This function, `fetch_user_text_posts`, is designed to fetch text-based posts from a specified Reddit user. The function takes a `username` as an input and attempts to retrieve the most recent 20 text-based posts from that user's Reddit submissions. Using the PRAW library, the `reddit.redditor(username)` method is called to access the user's posts, and the function iterates over these posts, concatenating the title and the content (selftext) of each post into a single string. These concatenated post details are then stored in a list. If an error occurs during this process (such as network issues or invalid user input), the function catches the exception and displays an error message using Streamlit's `st.write()`. If an exception is caught, an empty list is returned. This function is useful for collecting Reddit text data for further analysis or processing.

Fetching Image-Based Posts from Reddit and Performing OCR

```
# Function to fetch image-based posts from Reddit and perform
   OCR
def fetch_user_images_and_extract_text(username):
    try:
        user = reddit.redditor(username)
        images = [post.url for post in user.submissions.new(
           limit=20) if post.url.endswith(('.jpg', '.jpeg', '.
           png', '.webp', '.bmp', '.tiff'))]

        extracted_texts = []
        for image_url in images:
            try:
                response = requests.get(image_url)
                image = Image.open(BytesIO(response.content))
                st.image(image, caption="Fetched␣Image",
                   use_column_width=True)

                extracted_text = extract_text_from_image(image)
                if extracted_text.strip():
                    translated_text = GoogleTranslator(source='
                       auto', target='en').translate(
                       extracted_text)
                    extracted_texts.append(translated_text)
                    st.write("Extracted␣and␣Translated␣Text␣from
                       ␣Image:")
                    st.text(translated_text)
            except Exception as e:
                st.write(f"Error␣processing␣image␣{image_url}:␣{
                   e}")

        return extracted_texts
    except Exception as e:
        st.write(f"Error␣fetching␣images:␣{e}")
        return []
```

This function, `fetch_user_images_and_extract_text`, is designed to fetch image-based posts from a specified Reddit user and perform Optical Character Recognition (OCR) to extract text from the images. The function first attempts to retrieve the most recent 20 submissions from the specified Reddit user using the PRAW library. It filters the posts to include only those with image URLs that match common image file formats (e.g., `.jpg`, `.jpeg`, `.png`, etc.). For each valid image URL, the function fetches the image using the `requests` library and then processes it by opening the image with `Pillow`'s `Image.open()` method. The image is displayed on the Streamlit app using `st.image()` with the option to show it in the appropriate

column width. After displaying the image, the function calls `extract_text_from_image`, which performs OCR using Tesseract (assumed to be defined elsewhere in the code). If any text is successfully extracted, it is translated into English using the `GoogleTranslator` from the `deep_translator` library, and the translated text is displayed on the app using `st.write()` and `st.text()`. In case of errors (e.g., issues fetching the image or processing the OCR), exceptions are caught and an error message is displayed via Streamlit's `st.write()`. The function returns a list of extracted and translated texts from the images, or an empty list if any errors occur during the process. This function is helpful for gathering, processing, and translating text from image posts on Reddit.

Classifying Text and Displaying Results

```
# Function to classify text and display result
def classify_text(text):
    input_vectorized = vectorizer.transform([text])
    prediction_proba = model.predict_proba(input_vectorized)

    issue_labels = model.classes_
    proba_df = pd.DataFrame(prediction_proba, columns=
       issue_labels).T
    proba_df.columns = ['Probability']

    top_issue = proba_df['Probability'].idxmax()
    top_probability = proba_df['Probability'].max()

    st.write(f"The␣most␣likely␣mental␣health␣concern␣is:␣{
       top_issue}␣with␣a␣probability␣of␣{top_probability:.2%}")

    get_wellbeing_insight(text, top_issue)
```

This function, `classify_text`, is used to classify a given text and display the most likely mental health concern along with its probability. The function first vectorizes the input `text` using the `vectorizer.transform()` method, which converts the text into a format suitable for the machine learning model. The `model.predict_proba()` method is then called to get the probabilities for each possible class label, which represents different mental health issues. The class labels (issues) are retrieved using `model.classes_`, and a Pandas DataFrame (`proba_df`) is created to display the predicted probabilities for each issue. The DataFrame is transposed and renamed to give a clearer view, with a column for the probability values. The function then identifies the issue with the highest probability using `idxmax()` and retrieves the maximum probability value using `max()`. Finally, the most likely issue and its associated probability are displayed on the Streamlit app using `st.write()`. Additionally, the function calls `get_wellbeing_insight`, passing the text and the top issue, likely to generate further

insights into the mental health concern identified. This function is integral to classifying text data for mental health analysis and providing actionable insights based on the results.

Getting Wellbeing Insights from Gemini Model

```
# Function to get wellbeing insights from Gemini model
def get_wellbeing_insight(text, top_issue):
    try:
        chat_session = gemini_model.start_chat(history=[])
            prompt = f"<Prompt␣to␣get␣the␣well␣being␣based␣on␣
               Ryff␣Scale␣Six␣Factor␣Model>"

        response = chat_session.send_message(prompt)

        st.write("###␣Wellbeing␣Insight:")
        st.write(response.text)
    except Exception as e:
        st.write(f"Error␣retrieving␣wellbeing␣insights:␣{e}")
```

The function `get_wellbeing_insight` interacts with the Gemini AI model to generate insights related to a mental health issue based on the Ryff Scale of Psychological Well-Being. The function starts by initializing a chat session with the Gemini model using `gemini_model.start_chat()`. It then constructs a detailed prompt that outlines the six factors of well-being: autonomy, environmental mastery, personal growth, positive relations with others, purpose in life, and self-acceptance. These factors are crucial for evaluating an individual's psychological well-being. The prompt includes specific example statements related to each factor, offering context for how the mental health issue (`top_issue`) might affect the individual in each area. The function sends this prompt to the Gemini model and retrieves the response, which provides detailed advice and reflections on how the issue impacts each well-being factor. The response includes short paragraphs for each of the six factors, analyzing the potential effects of the issue on the individual's ability to function in these areas. After receiving the response, the function uses Streamlit's `st.write()` method to display the wellbeing insights. In case of any errors, such as issues with the Gemini model or the chat session, an error message is displayed using `st.write()`. This function is valuable for generating personalized psychological insights and offering practical advice based on the impact of a specific mental health issue.

Getting Video Audio from Reddit and combining them

```
def download_video(video_url, save_path):
    try:
        video_data = requests.get(video_url)
        with open(save_path, 'wb') as f:
            f.write(video_data.content)
        return save_path
    except Exception as e:
        st.write(f"Error downloading video: {e}")
        return None

def download_audio(audio_url, save_path):
    try:
        audio_data = requests.get(audio_url)
        with open(save_path, 'wb') as f:
            f.write(audio_data.content)
        return save_path
    except Exception as e:
        st.write(f"Error downloading audio: {e}")
        return None

def combine_video_audio(video_path, audio_path, output_path):
    try:
        # FFmpeg command to combine video and audio
        ffmpeg_command = [
            "/usr/bin/ffmpeg",
            "-i", video_path,  # Input video file
            "-i", audio_path,  # Input audio file
            "-c:v", "libx264",  # Use libx264 codec for video
            "-c:a", "aac",  # Use AAC codec for audio
            "-strict", "experimental",  # Allow experimental AAC
                encoding
            "-shortest",  # Use the shortest length (video or
               audio) to determine the output length
            output_path  # Output file path
        ]

        # Run FFmpeg command
        subprocess.run(ffmpeg_command, check=True)
        return output_path
    except Exception as e:
        st.write(f"Error combining video and audio: {e}")
        return None
```

Getting Video Audio from Reddit and combining them

```
def get_user_posts_with_videos(username, max_items=10):
    try:
        # Attempt to fetch the user's posts
        user = reddit.redditor(username)

        post_data = []
        for submission in user.submissions.new(limit=max_items):
            videos = []

            # Check if the post is a direct video
            if submission.is_video:
                # Get the URL of the hosted video (Reddit video
                   URL)
                video_url = submission.media['reddit_video']['
                   fallback_url']

                # Dynamically generate the audio URL by
                   replacing the resolution part with _AUDIO_128
                   .mp4
                audio_url = video_url.split("DASH_")[0] + "
                   DASH_AUDIO_128.mp4"
                videos.append({'video_url': video_url, '
                   audio_url': audio_url})

            # Only add posts with videos
            if videos:
                post_data.append({"text": submission.title, "
                   videos": videos})

        return post_data

    except praw.exceptions.RedditAPIException as e:
        st.error(f"Error␣fetching␣data␣from␣user␣'{username}':␣{
           e}")
        return []
    except Exception as e:
        st.error(f"Error␣fetching␣user␣data:␣{e}")
        return []
```

The provided code is designed to download audio and video files from a Reddit user's profile and then combine them into a single file. The first two functions, `download_video` and `download_audio`, handle downloading the video and audio files, respectively, by fetching the content from the given URLs and saving them to local files. These functions use the `requests` library to make HTTP requests and handle exceptions to ensure errors are logged

appropriately. The `combine_video_audio` function uses the FFmpeg library to merge the downloaded video and audio files into a single multimedia file. FFmpeg is a powerful tool for multimedia processing, and the command specifies the codecs to use for video and audio encoding. The combined file is saved at the specified output path. If an error occurs during this process, it is logged for debugging. The `get_user_posts_with_videos` function fetches the most recent posts from a specified Reddit user's profile. Using the PRAW library, it identifies posts containing videos, extracts their video and audio URLs, and prepares a dataset containing the post titles and associated media. The function also dynamically constructs audio URLs based on the Reddit video URL format. Finally, the data is returned for further processing, and any exceptions during the data fetch are logged. This system integrates various components such as HTTP requests, media processing, and Reddit API interactions, showcasing a seamless workflow for downloading, combining, and handling multimedia content programmatically.

Extract Text from Image Using Tesseract

```
# Function to extract text from image using Tesseract
def extract_text_from_image(image):
    extracted_text = pytesseract.image_to_string(image)
    return extracted_text.splitlines()

# Function to extract text from an image using Tesseract
def extract_text_from_image_video(image):
    extracted_text = pytesseract.image_to_string(image)
    return extracted_text if extracted_text else ""  # Return
       empty string if no text is found
```

The provided code consists of two functions designed to extract text from an image using the Tesseract OCR (Optical Character Recognition) engine.

The first function, `extract_text_from_image(image)`, takes an image object as input, applies the `pytesseract.image_to_string(image)` method to extract text from the image, and then splits the resulting text into individual lines using the `splitlines()` method. This allows the function to return a list where each element corresponds to a line of extracted text, making it easier to handle multiline results. The second function, `extract_text_from_image_video(image)`, operates similarly by extracting text from the image, but it includes an additional check to determine whether any text was successfully extracted. If no text is found, the function returns an empty string (`""`), ensuring that it always returns a consistent type rather than `None`, which could potentially cause issues in other parts of the code. Both functions rely on the `pytesseract` library, which is a Python wrapper for the open-source Tesseract OCR engine, to perform the text extraction. These functions are useful for extracting text from images, which can be helpful in various use cases such as doc-

ument scanning, analyzing images with embedded text, or processing video frames to capture text content.

Extract Frames from Video File

```
# Function to extract 20 frames from a video file
def extract_frames(video_path, num_frames=20):
    cap = cv2.VideoCapture(video_path)
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    frames = []
    frame_interval = total_frames // num_frames

    for i in range(num_frames):
        cap.set(cv2.CAP_PROP_POS_FRAMES, i * frame_interval)
        ret, frame = cap.read()
        if ret:
            frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            frames.append(frame)
    cap.release()
    return frames
```

The `extract_frames(video_path, num_frames=20)` function extracts a specified number of frames from a given video file. The function uses the OpenCV library, specifically the `cv2.VideoCapture()` method, to load the video from the file path provided by the `video_path` argument. First, it calculates the total number of frames in the video using `cap.get(cv2.CAP_PROP_FRAME_COUNT)`. Next, it calculates the frame interval by dividing the total number of frames by the desired number of frames to be extracted (`num_frames`). For each of the frames, the function sets the position of the video playback to a specific frame using `cap.set(cv2.CAP_PROP_POS_FRAMES, i * frame_interval)`. It then reads the frame, converts the color from BGR (Blue-Green-Red) to RGB using `cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)`, and appends the frame to a list. This process is repeated for the number of frames specified. Finally, the video capture object is released with `cap.release()`, and the list of frames is returned. This function can be useful in scenarios where frame extraction is needed from videos for further processing, such as image analysis, video summarization, or even object detection tasks.

Transcribe Audio from Video

```
def transcribe_audio_from_video(video_file):
    try:
        # Save the uploaded video file to a temporary file
        with tempfile.NamedTemporaryFile(delete=False, suffix=".
           mp4") as temp_video_file:
            temp_video_file.write(video_file.read())
            temp_video_path = temp_video_file.name

        audio_path = tempfile.NamedTemporaryFile(suffix=".wav",
           delete=False).name

        # Extract audio from video using subprocess
        subprocess.run(["ffmpeg", "-i", temp_video_path, "-q:a",
            "0", "-map", "a", audio_path, "-y"])
        audio = AudioSegment.from_file(audio_path)

        # Use Whisper to transcribe the audio
        result = whisper_model.transcribe(audio_path)

        # Get the transcribed text and translate if necessary
        transcribed_text = result["text"]
        translated_text = GoogleTranslator(source="auto", target
           ="en").translate(transcribed_text)

        # Clean up temporary files
        os.remove(temp_video_path)
        os.remove(audio_path)

        return translated_text

    except Exception as e:
        # Display a user-friendly message if the video is too
           long or another error occurs
        if "duration" in str(e).lower() or "length" in str(e).
           lower():
            return "The␣video␣is␣too␣long␣to␣process.␣Please␣
               upload␣a␣shorter␣video."
        else:
            return f"An␣error␣occurred:␣{e}"
```

The `transcribe_audio_from_video(video_file)` function processes a video file by first saving it to a temporary file on disk. It then extracts the audio using the `ffmpeg` command-line tool, saving the extracted audio as a WAV file. The extracted audio is processed using the Whisper model for transcription, which outputs the transcribed text. Additionally, the function utilizes the `GoogleTranslator` to translate the text into English if necessary. Temporary

files created during processing are deleted to manage resources efficiently. In case of errors, the function returns a user-friendly message, especially handling scenarios where the video duration exceeds allowable limits. This function is useful for converting spoken content in videos to text for further processing.

Translate Text Using DeepL

```
# Function to translate text using DeepL
def translate_text(text, target_lang="en"):
    try:
        if text:
            translated_text = GoogleTranslator(source="auto",
               target=target_lang).translate(text)
            return translated_text
        return ""  # Return empty string if text is empty or
           None
    except Exception as e:
        return f"Error translating text: {str(e)}"
```

The `translate_text(text, target_lang="en")` function provides an interface to translate a given string of text into a specified target language, with English (`"en"`) set as the default. Using the `GoogleTranslator` library, the function automatically detects the source language (`source="auto"`) and translates the text to the desired target language. If the input text is empty or `None`, the function gracefully returns an empty string. In case of any errors during the translation process, such as network issues or invalid input, an error message containing the exception details is returned. This function is particularly useful for applications requiring multilingual support, enabling seamless translation of text between languages.

Extract Audio from Video File

```
def extract_audio_from_video(video_path):
    try:
        audio_path = tempfile.NamedTemporaryFile(delete=False,
           suffix=".wav").name
        # Use FFmpeg to extract audio from video
        subprocess.run(["ffmpeg", "-i", video_path, "-q:a", "0",
            "-map", "a", audio_path, "-y"])
        return audio_path
    except Exception as e:
        return f"Error extracting audio: {str(e)}"
```

The `extract_audio_from_video(video_path)` function facilitates the extraction of audio content from a video file. It first generates a temporary file path with a `.wav` suffix

to store the extracted audio. The function uses the `ffmpeg` command-line tool to process the video file specified by `video_path`, extracting the audio stream with high quality (`-q:a 0`) and saving it to the temporary file. The path to the extracted audio file is returned for further use. In the event of an error, such as invalid file paths or processing issues, the function catches the exception and returns a descriptive error message. This function is useful in multimedia applications where the separation of audio from video content is needed, such as transcription or audio analysis workflows.

Analyze Audio Mood Based on Extracted Audio

```
# Function to analyze audio mood based on extracted audio
def analyze_audio_mood(video_path):
    try:
        # Extract audio from the video (assuming
           extract_audio_from_video is implemented)
        audio_path = extract_audio_from_video(video_path)

        # Load the audio file using librosa
        y, sr = librosa.load(audio_path)

        # Extract MFCCs (Mel-frequency cepstral coefficients)
           from the audio signal
        mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)

        # Divide the MFCC array into 4 frequency bands and
           calculate scalar mean for each band

        # Low Frequencies: MFCC 0, 1, 2
        low_freq_mfcc = np.mean(mfcc[0:3], axis=1)
        mean_low = np.mean(low_freq_mfcc)  # Scalar mean for low
            frequencies

        # Mid-Low Frequencies: MFCC 3, 4
        mid_low_freq_mfcc = np.mean(mfcc[3:5], axis=1)
        mean_mid_low = np.mean(mid_low_freq_mfcc)  # Scalar mean
            for mid-low frequencies

        # Mid-High Frequencies: MFCC 5, 6, 7
        mid_high_freq_mfcc = np.mean(mfcc[5:8], axis=1)
        mean_mid_high = np.mean(mid_high_freq_mfcc)  # Scalar
           mean for mid-high frequencies

        # High Frequencies: MFCC 8, 9, 10, 11, 12
        high_freq_mfcc = np.mean(mfcc[8:13], axis=1)
        mean_high = np.mean(high_freq_mfcc)  # Scalar mean for
           high frequencies
```

Analyze Audio Mood Based on Extracted Audio

```
        # Now use these scalar means for classification

        if mean_high <= mean_low and mean_high <= mean_mid_low
           and mean_high <= mean_mid_high:
            return "Audio␣sounds␣normal,␣with␣no␣dominant␣
               emotion␣detected"

        elif mean_mid_high <= mean_low and mean_mid_high <=
           mean_mid_low and mean_mid_high <= mean_high:
            return "Audio␣sounds␣neutral,␣calm,␣or␣peaceful"

        elif mean_mid_low <= mean_low and mean_mid_low <=
           mean_mid_high and mean_mid_low <= mean_high:
            return "Audio␣sounds␣slightly␣melancholic␣or␣neutral
               "

        elif mean_low <= mean_mid_low and mean_low <=
           mean_mid_high and mean_low <= mean_high:
            return "Audio␣sounds␣calm␣or␣melancholic,␣with␣less␣
               intensity"

        elif mean_high > mean_low and mean_high > mean_mid_low
           and mean_high <= mean_mid_high:
            return "Audio␣sounds␣depressive␣or␣anxious␣in␣nature
               "

        else :
            return "Audio␣sounds␣upbeat␣and␣energetic␣(Happy)"

    except Exception as e:
        return f"Error␣analyzing␣audio␣mood:␣{str(e)}"
```

The `analyze_audio_mood(video_path)` function determines the mood of an audio segment extracted from a given video file. It begins by extracting audio using the `extract_audio_from_video` function and loading the audio file with `librosa`. The function computes Mel-frequency cepstral coefficients (MFCCs), which are features widely used in audio signal processing for mood or emotion analysis. The MFCCs are divided into four frequency bands (low, mid-low, mid-high, high), and scalar means are computed for each band. Based on these scalar values, a series of conditions classify the mood as normal, neutral, melancholic, calm, depressive, or upbeat. In case of an exception, an error message with details is returned. This function can be utilized in applications such as multimedia content analysis, emotion recognition, or mood-based music recommendations.

Twitter API call and post extraction

```
# Initialize Twitter API
BEARER_TOKEN = "<TWITTER-BEARER-TOKEN>"
client = tweepy.Client(bearer_token=BEARER_TOKEN)

# Twitter
def fetch_image_content(image_url):
    """Fetch and process an image from a URL."""
    try:
        response = requests.get(image_url, timeout=10)
        response.raise_for_status()  # Ensure the request was
           successful
        return Image.open(BytesIO(response.content))
    except Exception as e:
        st.write(f"Error fetching image: {e}")
        return None

def get_latest_tweets_with_images(username, max_items=10):
    """Fetch latest tweets with text and associated images."""
    # Fetch user details to get user ID
    user = client.get_user(username=username)
    if not user.data:
        return [], []

    user_id = user.data.id

    # Fetch the latest tweets (exclude retweets and replies)
    response = client.get_users_tweets(
        id=user_id,
        tweet_fields=["attachments"],
        expansions=["attachments.media_keys"],
        media_fields=["url"],
        exclude=["retweets", "replies"],
        max_results=max_items
    )

    tweet_data = []

    if response.data:
        for tweet in response.data:
            # Extract text
            text = tweet.text
```

Twitter API call and post extraction

```
            # Extract images if available
            images = []
            if hasattr(tweet, "attachments") and tweet.
               attachments is not None:
                if "media_keys" in tweet.attachments:
                    for media_key in tweet.attachments["
                       media_keys"]:
                        media = next(
                            (media for media in response.
                               includes.get("media", []) if
                               media["media_key"] == media_key),
                                None
                        )
                        if media and media.type == "photo":
                            images.append(media.url)

            # Append tweet data
            tweet_data.append({"text": text, "images": images})

    return tweet_data
```

The above code snippet initializes the Twitter API using the `tweepy` library, specifically with a `BEARER_TOKEN` for authentication. It defines two functions: `fetch_image_content` and `get_latest_tweets_with_images`. The `fetch_image_content` function retrieves an image from a provided URL using the `requests` library. It ensures successful HTTP requests via `response.raise_for_status()` and opens the image using the `Pillow` library's `Image.open`, handling errors gracefully by returning `None` if an exception occurs. The `get_latest_tweets_with_images` function fetches the latest tweets from a specified username, extracting text and associated image URLs. It first retrieves the user's unique Twitter ID using `client.get_user`. It then fetches tweets using `client.get_users_tweets`, excluding retweets and replies, and includes `attachments.media_keys` to identify images. For each tweet, it extracts the text and resolves media URLs by matching `media_keys` with `response.includes` media data, appending only those of type `photo`. The final result is a list of dictionaries, each containing tweet text and a list of image URLs.

Streamlit Mental Health Disorder Detection App

```
# Define the Streamlit app
def run_app():
    st.title("Mental Health Disorder Detection")

    option = st.sidebar.selectbox(
        "Choose an option",
        ["Text Input", "Image Upload", "Video Upload", "Reddit
           Username Analysis"]
    )

    # Text Input
    if option == "Text Input":
        st.subheader("Enter Text to Classify Mental Health Issue
           ")
        input_text = st.text_area("Enter your text here:")

        if st.button("Classify Text"):
            if input_text.strip() == "":
                st.write("Please enter some text to classify.")
            else:
                translated_text = GoogleTranslator(source='auto'
                   , target='en').translate(input_text)
                st.write("Translated Text (to English):")
                st.write(translated_text)
                classify_text(translated_text)

    # Image Upload
    elif option == "Image Upload":
        st.subheader("Upload an Image to Extract and Classify
           Text")
        uploaded_image = st.file_uploader("Upload an Image",
           type=["jpg", "jpeg", "png", "webp", "bmp", "tiff"])

        if uploaded_image is not None:
            image = Image.open(uploaded_image)
            st.image(image, caption="Uploaded Image",
               use_column_width=True)

            extracted_text = extract_text_from_image(image)
            translated_text = GoogleTranslator(source='auto',
               target='en').translate("\n".join(extracted_text))

            st.subheader("Translated Text (to English)")
            st.text(translated_text)
```

Streamlit Mental Health Disorder Detection App

```
            if st.button("Classify␣Extracted␣Text"):
                if not translated_text or translated_text.strip
                    () == "":
                    st.write("It␣is␣normal␣with␣probability␣100%
                        ")
                else:
                    classify_text(translated_text)

    # Video Upload
    elif option == "Video␣Upload":
        st.subheader("Upload␣a␣Video␣to␣Extract␣and␣Classify␣
            Text")
        video_file = st.file_uploader("Choose␣a␣video␣file",
            type=["mp4", "mov", "avi"])

        if video_file:
            video_path = "/tmp/uploaded_video.mp4"
            with open(video_path, "wb") as f:
                f.write(video_file.getbuffer())

            st.video(video_file)

            frames = extract_frames(video_path)
            combined_text = ""

            st.write("Extracting␣frames␣from␣video␣...")
            for idx, frame in enumerate(frames):
                st.image(frame, caption=f"Frame␣{idx␣+␣1}",
                    use_column_width=True)
                text_from_frame = extract_text_from_image_video(
                    frame)
                if text_from_frame and text_from_frame not in
                    combined_text:
                    combined_text += text_from_frame + "␣"

            st.write("Text␣Extracted␣from␣Video␣Frames:")
            st.text(combined_text)

            transcribed_audio_text = transcribe_audio_from_video
                (video_file)
            st.write("Transcribed␣Audio␣Text:")
            st.text(transcribed_audio_text)

            full_combined_text = combined_text + "␣" +
                transcribed_audio_text
            translated_combined_text = translate_text(
                full_combined_text)
```

Streamlit Mental Health Disorder Detection App

```
            st.write("Translated␣Combined␣Text␣(Frames␣+␣Audio):
               ")
            st.text(translated_combined_text)

            st.write("Analyzing␣Audio␣Mood...")
            mood_result = analyze_audio_mood(video_path)
            st.write(mood_result)

            if st.button("Classify␣Extracted␣Text"):
                classify_text(translated_combined_text)

    # Reddit Username Analysis
    elif option == "Reddit␣Username␣Analysis":
        st.subheader("Enter␣Reddit␣Username␣for␣Analysis")
        username = st.text_input("Enter␣Reddit␣username:")

        if st.button("Analyze"):
            text_posts = fetch_user_text_posts(username)
            image_texts = fetch_user_images_and_extract_text(
               username)

            all_text = text_posts + image_texts
            if all_text:
                predictions = [model.predict(vectorizer.
                   transform([text]))[0] for text in all_text]
                issue_counts = Counter(predictions)
                top_issue, top_count = issue_counts.most_common
                   (1)[0]
                st.write(f"The␣most␣frequent␣issue:␣{top_issue}␣
                   ({(top_count␣/␣len(predictions))␣*␣100:.2f}%)
                   ")
                issue_distribution = pd.DataFrame(issue_counts.
                   items(), columns=['Mental␣Health␣Issue', '
                   Count'])
                st.write("Mental␣health␣issue␣distribution␣
                   across␣posts:")
                st.write(issue_distribution)

                # Call the Gemini model to get well-being
                   insights
                get_wellbeing_insight("␣".join(all_text),
                   top_issue)
            else:
                st.write("No␣valid␣text␣found␣for␣analysis.")
```

Streamlit Mental Health Disorder Detection App

```
    # Twitter Username Analysis
    elif option == "Twitter␣Username␣Analysis":
        st.subheader("Enter␣Twitter␣Username␣for␣Analysis")
        username = st.text_input("Enter␣Twitter␣username:")

        if st.button("Analyze"):
            if username.strip() == "":
                st.write("Please␣enter␣a␣Twitter␣username.")
            else:
                # Fetch the latest tweets with associated images
                tweets_with_images =
                   get_latest_tweets_with_images(username)

                # Extract text content from tweets
                text_posts = [tweet['text'] for tweet in
                   tweets_with_images if tweet['text']]
                st.write("Recent␣Text␣Posts␣from␣Tweets:")
                st.write(text_posts[:3])  # Display a few posts
                   for review

                # Extract and process text from associated
                   images
                image_texts = []
                for tweet in tweets_with_images:
                    for image_url in tweet['images']:
                        image = fetch_image_content(image_url)
                        if image:
                            st.image(image, caption=f"Image␣from
                               ␣Tweet", use_column_width=True)
                        if image:
                            extracted_text =
                               extract_text_from_image(image)  #
                                Assuming a text extraction
                               function is defined
                            if extracted_text:
                                image_texts.append(
                                   extracted_text)

                # Combine text from both tweet text and
                   extracted image text
                all_text = text_posts + image_texts

                # Ensure all entries in all_text are strings
                all_text = [str(text) for text in all_text if
                   text]
```

Streamlit Mental Health Disorder Detection App

```
                if all_text:
                    predictions = []
                    for text in all_text:
                        try:
                            # Vectorize and classify each text
                            input_vectorized = vectorizer.
                               transform([text])
                            prediction = model.predict(
                               input_vectorized)
                            predictions.append(prediction[0])
                        except Exception as e:
                            st.write(f"Error␣processing␣text:␣{
                               text[:50]}...␣-␣{e}")
                            continue

                    # Count the most common mental health issue
                    issue_counts = Counter(predictions)
                    top_issue, top_count = issue_counts.
                       most_common(1)[0]
                    top_percentage = (top_count / len(
                       predictions)) * 100

                    st.write(f"The␣most␣frequently␣detected␣
                       mental␣health␣concern␣is:␣{top_issue},␣
                       appearing␣in␣{top_percentage:.2f}%␣of␣
                       analyzed␣text.")
                    issue_distribution = pd.DataFrame(
                       issue_counts.items(), columns=['Mental␣
                       Health␣Issue', 'Count'])
                    st.write("Mental␣health␣issue␣distribution␣
                       across␣posts:")
                    st.write(issue_distribution)

                    # Call the Gemini model to get well-being
                       insights
                    get_wellbeing_insight("␣".join(all_text),
                       top_issue)
                else:
                    st.write("No␣valid␣text␣found␣for␣analysis."
                       )

# Run the app
if __name__ == '__main__':
    run_app()
```

The `run_app()` function defines a Streamlit application to detect mental health disorders using multiple input methods: text input, image upload, video upload, and Reddit username analysis. Users can interact with a sidebar to choose their preferred mode of input. Each mode processes the data accordingly, utilizing pre-defined helper functions for text translation, text extraction from images, audio transcription, and Reddit user data retrieval. Depending on the mode, the app combines extracted and processed data to classify mental health concerns or display insights. It leverages libraries like `GoogleTranslator`, `librosa`, and `FFmpeg` for processing and analysis. The results include mood analysis, classification probabilities, and potential mental health issues. This app provides a comprehensive tool for mental health-related data analysis. For the Twitter analysis mode, the app allows users to input a Twitter username, fetches the latest tweets from the user, and extracts both text content and any associated images. It uses the `tweepy` library to interact with the Twitter API and retrieve tweets that are not retweets or replies. For each tweet, the app extracts the text and checks for any media attachments, specifically images. If images are found, it fetches and processes them using the `fetch_image_content` function, which downloads the image and uses OCR (Optical Character Recognition) to extract text from the image. Both the tweet text and any extracted text from the images are combined to create a comprehensive dataset for analysis. This data is then passed through a pre-trained machine learning model to classify the mental health concerns present in the posts. The model generates predictions, and the app displays the most frequently detected mental health issue along with its percentage occurrence. It also shows the distribution of mental health issues across the posts and provides insights using a separate well-being model. This feature enables the app to offer a detailed analysis of the mental health state inferred from Twitter posts, based on the extracted text and images. Additionally, the app integrates the Gemini API for advanced mood analysis and mental well-being insights. The Gemini API, accessed through a Flask server, provides an external service that can process large volumes of text data for sentiment analysis, mood classification, and mental health predictions. The Gemini API is a machine learning-powered API that allows users to analyze textual data by passing in relevant text input. In this case, the app sends the combined text from Twitter posts (including extracted text from images) to the Gemini API for analysis. The API returns insights such as sentiment scores, emotional tones, and relevant mental health classifications based on the input. The Flask server acts as an intermediary between the Streamlit app and the Gemini API. It handles HTTP requests from the Streamlit app, passes the text data to the Gemini API, and processes the responses before sending them back to the Streamlit interface. When a user submits their data, the app makes an HTTP request to the Flask server, which in turn queries the Gemini API. The Flask server ensures that the data is properly formatted and can handle various types of inputs, whether they are text, audio, or other formats. The Gemini API processes this data and returns a structured response that contains mood-related insights and mental health predictions.

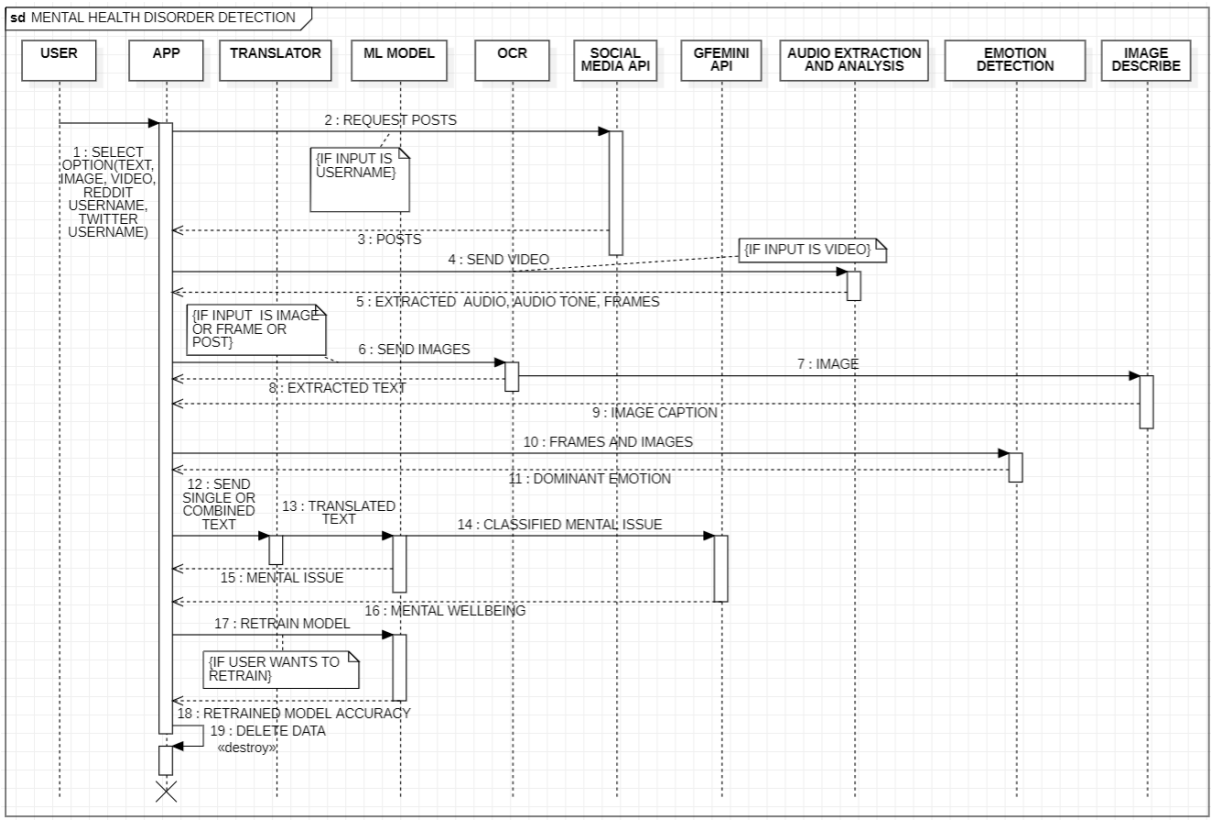


Figure 52: Sequence Diagram of the Application

Below are some screenshots from the web application.

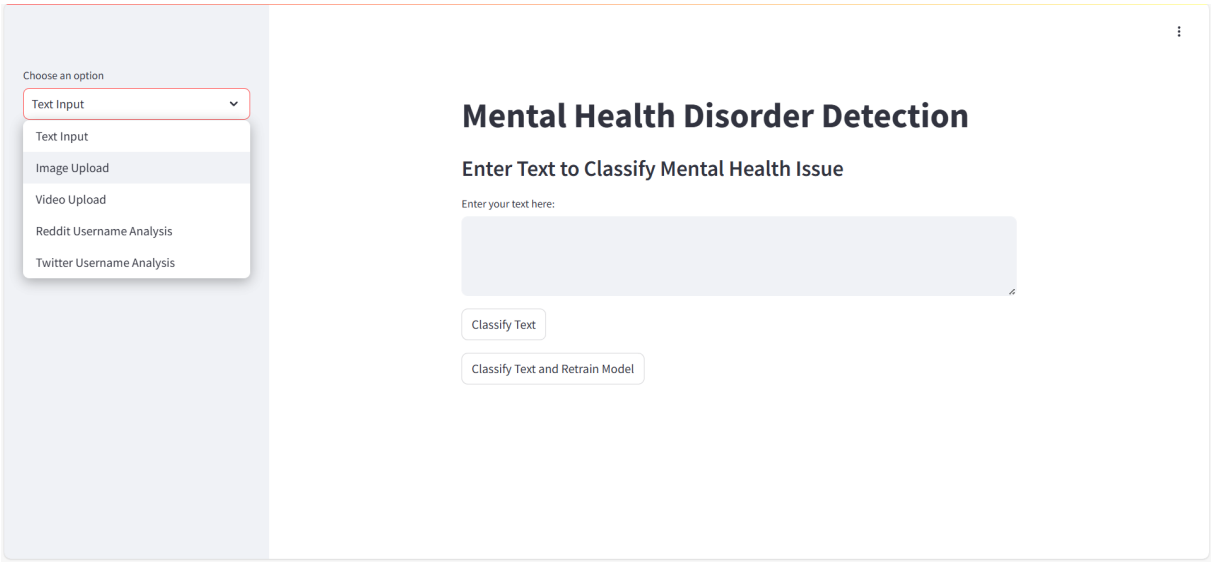


Figure 53: Website with all options

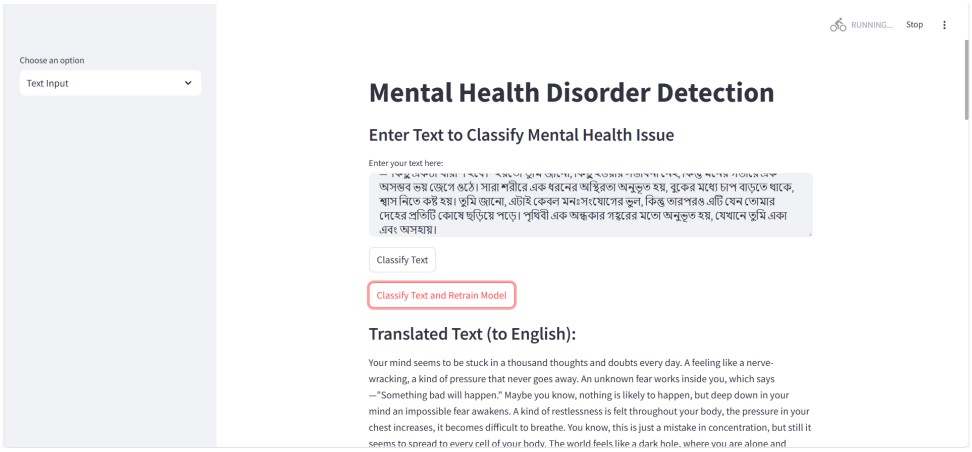


Figure 54: Entering Text for classification

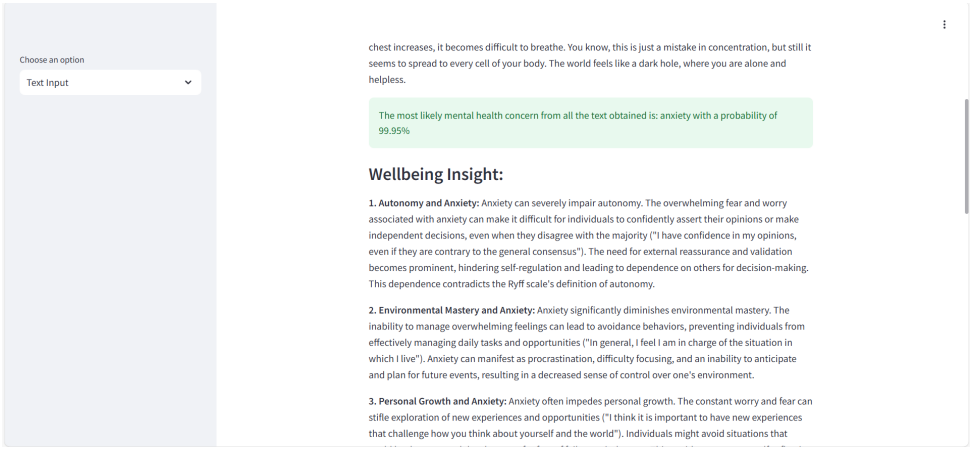


Figure 55: Text Classification Result

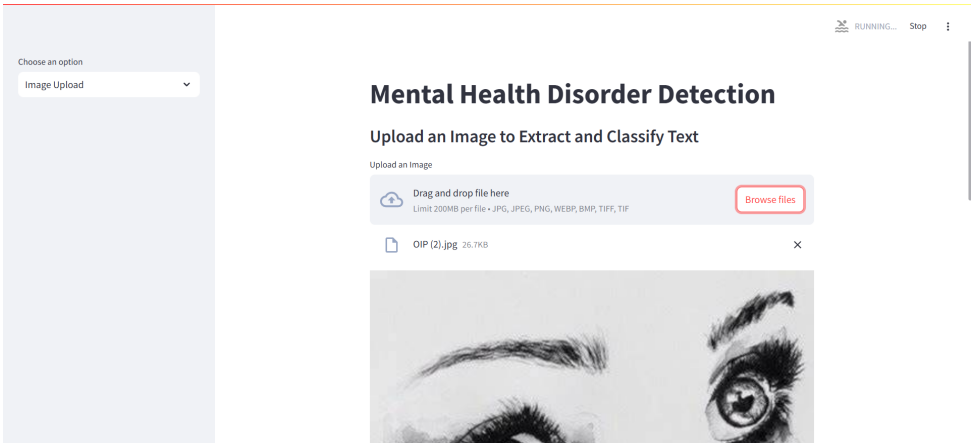


Figure 56: Upload Image

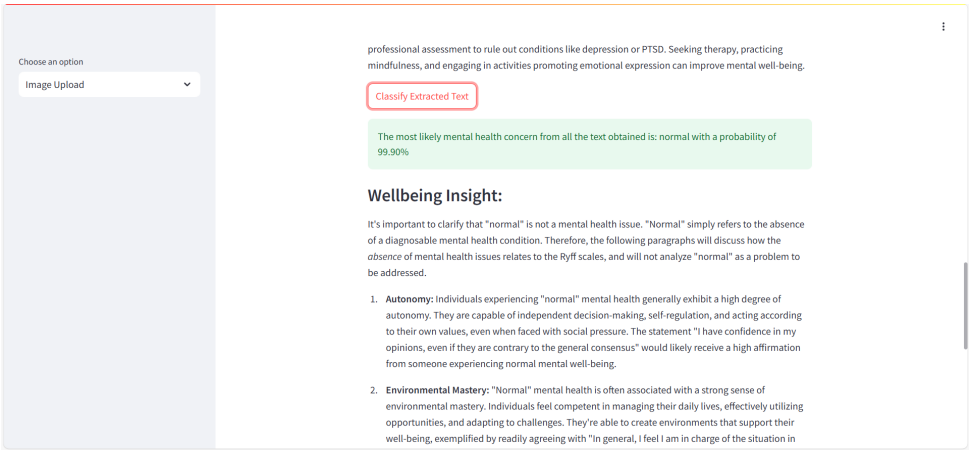


Figure 57: Image Classification Result

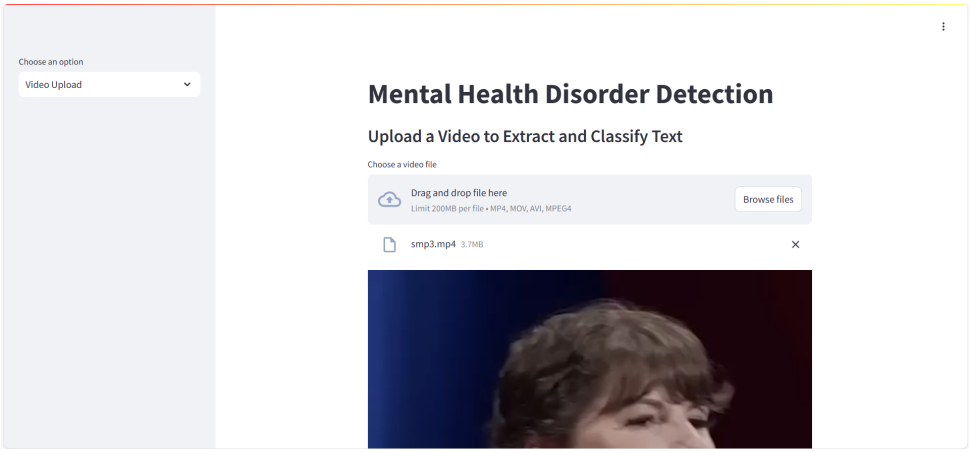


Figure 58: Upload Video

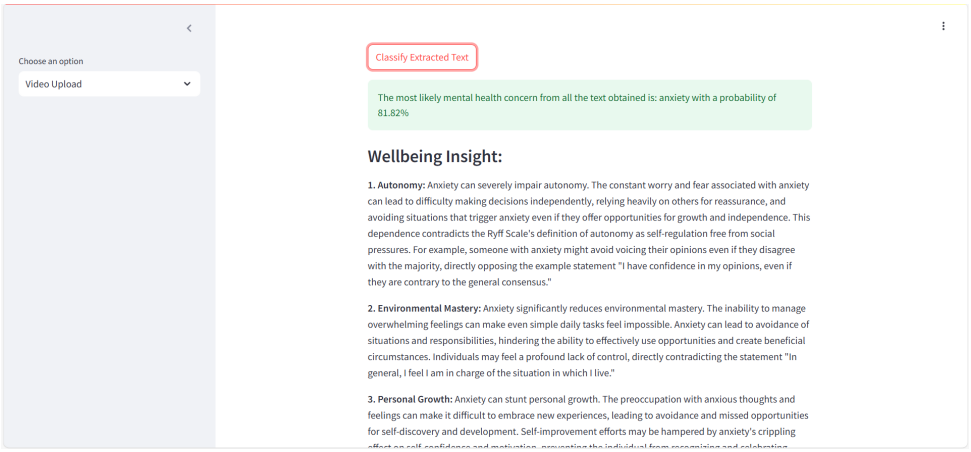


Figure 59: Video Classification Result

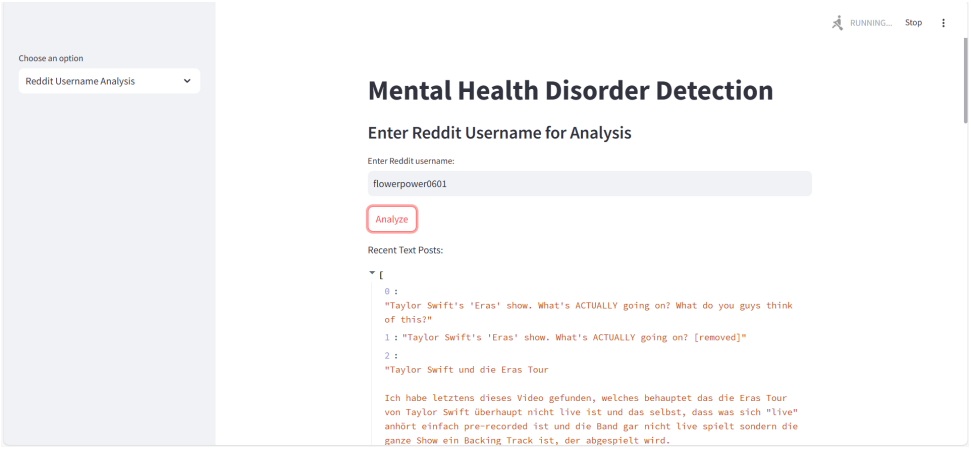


Figure 60: Reddit User Analysis

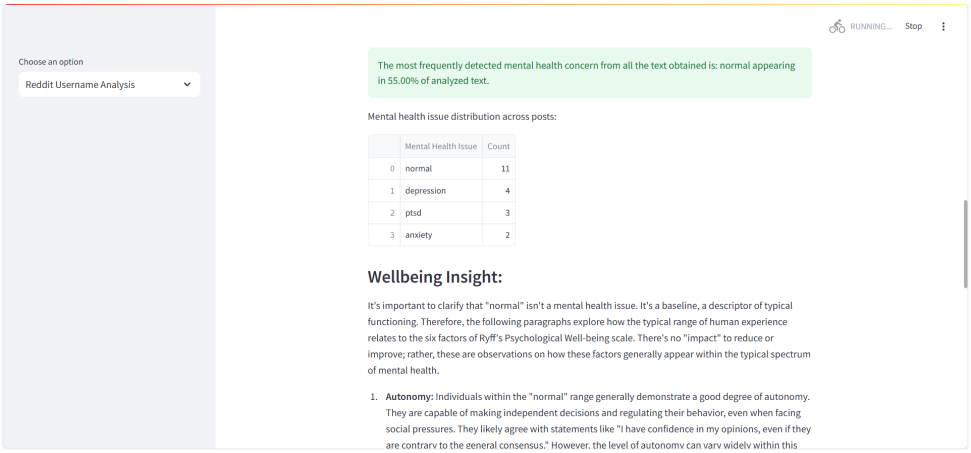


Figure 61: Result from Reddit Posts Analysis

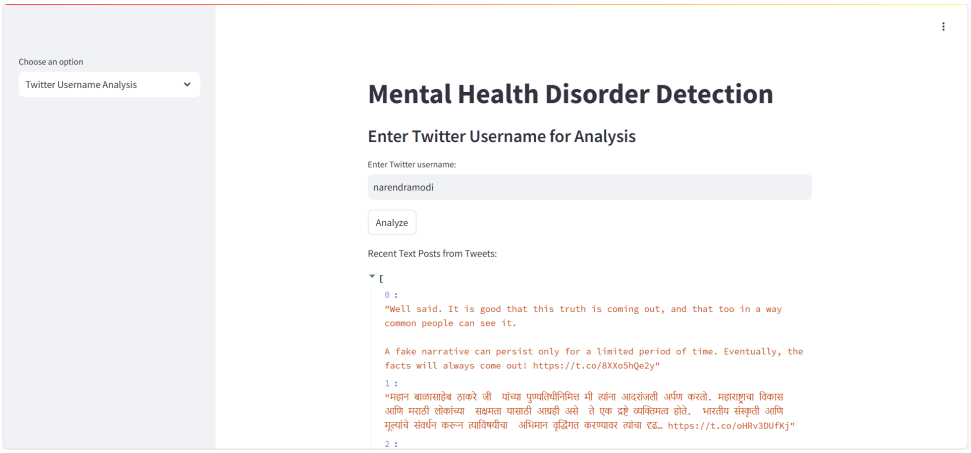


Figure 62: Twitter User Analysis

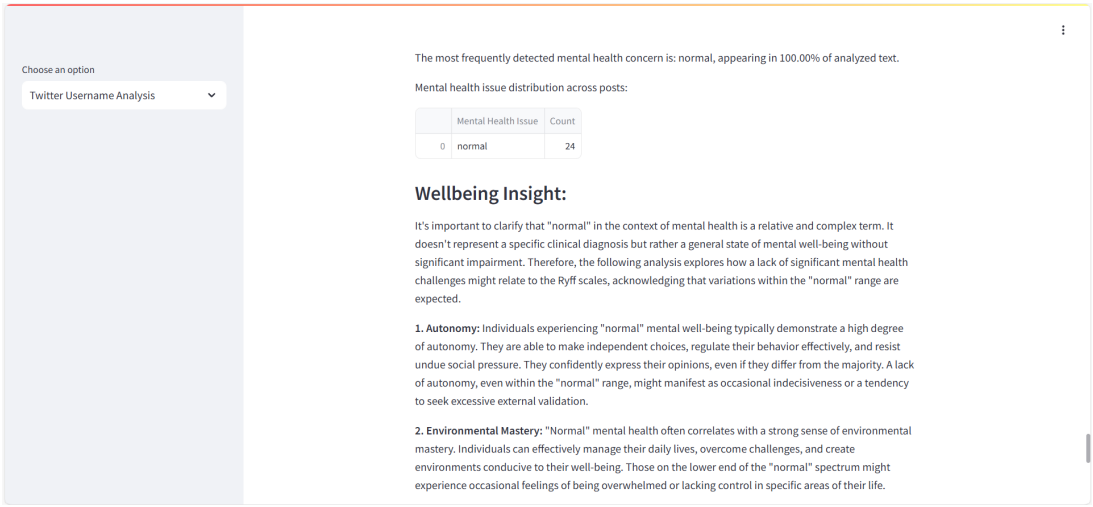


Figure 63: Result from Twitter Posts Analysis

Retraining the Ensemble Model

```
def retrain_model():
    # Load Logistic Regression model and vectorizer
    with open('LRmodel.pkl', 'rb') as file:
        lr_model = pickle.load(file)
    with open('LRvectorizer.pkl', 'rb') as file:
        lr_vectorizer = pickle.load(file)

    # Load SVM model and vectorizer
    with open('SVMmodel.pkl', 'rb') as file:
        svm_model = pickle.load(file)
    with open('SVMvectorizer.pkl', 'rb') as file:
        svm_vectorizer = pickle.load(file)

    # Load XGBoost model, vectorizer, and label encoder
    with open('xgb_model.pkl', 'rb') as file:
        xgb_model = pickle.load(file)
    with open('tfidf_vectorizer.pkl', 'rb') as file:
        tfidf_vectorizer = pickle.load(file)
    with open('label_encoder.pkl', 'rb') as file:
        label_encoder = pickle.load(file)

    # Load LSTM model, tokenizer, and label encoder
    lstm_model = load_model('lstm_model.h5')
    with open('LSTM_tokenizer.pkl', 'rb') as file:
        lstm_tokenizer = pickle.load(file)
```

Retraining the Ensemble Model

```
# Load Naive Bayes model and vectorizer
with open('NBmodel.pkl', 'rb') as file:
    nb_model = pickle.load(file)
with open('NBvectorizer.pkl', 'rb') as file:
    nb_vectorizer = pickle.load(file)

# Load the test dataset
data = pd.read_csv('preprocessed_mental_health.csv')

# Check if 'cleaned_text' column exists
if 'cleaned_text' not in data.columns:
    raise ValueError("The dataset must have a 'cleaned_text'
       column.")

# Remove rows with missing values in 'cleaned_text'
data.dropna(subset=['cleaned_text'], inplace=True)

# Split features and target
X_test = data['cleaned_text']
y_test = data['mental_health_issue']

# Encode target labels
y_test = label_encoder.transform(y_test)

# Process the text for each model
X_test_lr = lr_vectorizer.transform(X_test)  # Logistic
   Regression vectorizer
X_test_svm = svm_vectorizer.transform(X_test)  # SVM
   vectorizer
X_test_xgb = tfidf_vectorizer.transform(X_test)  # XGBoost
   vectorizer
X_test_nb = nb_vectorizer.transform(X_test)  # Naive Bayes
   vectorizer
X_test_lstm = lstm_tokenizer.texts_to_sequences(X_test)  #
   LSTM tokenizer

# Pad sequences for LSTM
X_test_lstm = pad_sequences(X_test_lstm, maxlen=100, padding
   ='post', truncating='post')

# Get predictions from the base models
lr_predictions_proba = lr_model.predict_proba(X_test_lr)  #
   Logistic Regression probabilities
svm_predictions_proba = svm_model.predict_proba(X_test_svm)
    # SVM probabilities
```

Retraining the Ensemble Model

```
xgb_predictions_proba = xgb_model.predict_proba(X_test_xgb)
    # XGBoost probabilities
nb_predictions_proba = nb_model.predict_proba(X_test_nb)  #
   Naive Bayes probabilities
lstm_predictions_proba = lstm_model.predict(X_test_lstm)  #
   LSTM probabilities

# Stack the predictions of all models to create the feature
   matrix for the meta-learner
stacked_features = np.hstack((lr_predictions_proba,
   svm_predictions_proba, xgb_predictions_proba,
   nb_predictions_proba, lstm_predictions_proba))

# Define the Logistic Regression model for the meta-learner
meta_learner_lr = LogisticRegression(max_iter=5000)

# Train the Logistic Regression meta-learner on the full
   dataset
meta_learner_lr.fit(stacked_features, y_test)

# Save the trained Logistic Regression meta-learner
with open('meta_learner_lr.pkl', 'wb') as file:
    pickle.dump(meta_learner_lr, file)

# Predict using the Logistic Regression meta-learner
final_predictions_lr = meta_learner_lr.predict(
   stacked_features)

# Evaluate the Logistic Regression ensemble model
accuracy_lr = accuracy_score(y_test, final_predictions_lr)

return meta_learner_lr, accuracy_lr
```

The above function, `retrain_model`, is designed to retrain an ensemble learning model (similar for adding transformer by loading the transformer files) using an updated dataset. Initially, it loads multiple base models (Logistic Regression, SVM, XGBoost, Naive Bayes, and LSTM) along with their respective vectorizers, tokenizers, and label encoder from previously saved pickle files. The function then loads and preprocesses the updated dataset, ensuring it contains the necessary text column (`cleaned_text`) and splitting it into features (`X_test`) and target labels (`y_test`). The target labels are encoded using the label encoder, while the text features are transformed using the corresponding vectorizers or tokenizers for each model. Predictions from each model are then collected, stacked together, and used to retrain the meta-learner (RandomForestClassifier). Once trained, the meta-learner is saved for future use.

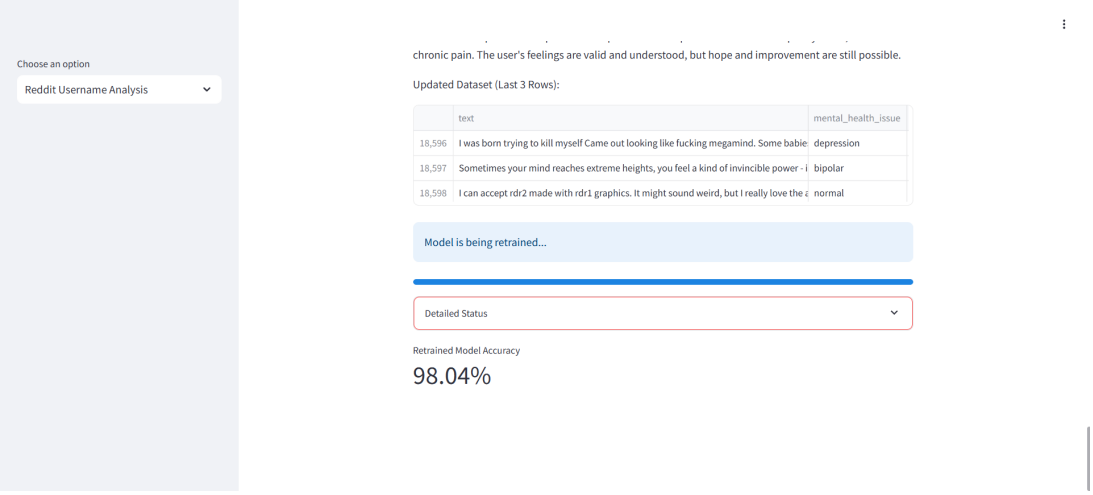


Figure 64: Result from Reddit Analysis and Model Retraining

Emotion Analysis and Mental Health Insights

```
def detect_emotions_from_frame(frame):
    try:
        # Use DeepFace to analyze emotions
        result = DeepFace.analyze(frame, actions=['emotion'],
            enforce_detection=False)
        return result[0]['dominant_emotion']
    except Exception as e:
        print(f"No expression or error detecting emotion: {e}")
        return None

def analyze_emotions_from_frames(frames):
    # Initialize an emotion counts dictionary
    emotion_counts = {'happy': 0, 'sad': 0, 'angry': 0, 'disgust
        ': 0, 'fear': 0, 'surprise': 0, 'neutral': 0}
    frame_emotions = []

    for idx, frame in enumerate(frames):
        # Detect emotions from the current frame
        emotion = detect_emotions_from_frame(frame)
        if emotion:
            frame_emotions.append(emotion)
            if emotion in emotion_counts:
                emotion_counts[emotion] += 1

    return emotion_counts, frame_emotions
```

Emotion Analysis and Mental Health Insights

```
def display_emotion_summary(emotion_counts):
    # Convert the emotion counts to a DataFrame for display
    emotion_df = pd.DataFrame(list(emotion_counts.items()),
       columns=['Emotion', 'Count'])
    st.write("Emotion␣Analysis␣Summary:")
    st.table(emotion_df)
    return max(emotion_counts, key=emotion_counts.get)

def analyze_with_gemini(dominant_emotion, emotion_counts):
    try:
        # Start a chat session with the Gemini API
        chat_session = gemini_model.start_chat(history=[])

        # Summarize emotion counts for the prompt
        emotion_summary = ",␣".join([f"{emotion}:␣{count}" for
           emotion, count in emotion_counts.items()])
        # Create a prompt for the Gemini API
        prompt = (
            f"The␣detected␣dominant␣emotion␣is␣'{
               dominant_emotion}'.␣"
            f"{emotion_summary}.␣Based␣on␣this␣information,␣
               analyze␣the␣potential␣implications␣for␣mental␣
               health␣"
            f"conditions␣such␣as␣depression,␣anxiety,␣PTSD,␣or␣
               bipolar␣disorder.␣Provide␣actionable␣insights␣in␣
               three␣lines."
        )

        # Send the prompt and get a response
        response = chat_session.send_message(prompt)
        st.write(response.text)
    except Exception as e:
        print(f"Error␣in␣Gemini␣API␣call:␣{e}")
        return "An␣error␣occurred␣while␣communicating␣with␣the␣
           Gemini␣API.␣Please␣try␣again␣later."
```

The functions in the code work together to analyze emotions from video frames. The detect_emotions_from_frame(frame) function utilizes the DeepFace library to detect the dominant emotion in a given video frame, returning None if no emotion is detected or an error occurs. The analyze_emotions_from_frames(frames) function iterates over multiple video frames, applying detect_emotions_from_frame to each and counting the frequency of detected emotions in a dictionary. The display_emotion_summary(emotion_counts) function uses Streamlit to display a summary of detected emotions in a table format and identifies the dominant emotion with the highest count. Lastly, analyze_with_gemini(dominant_emotion, emotion_counts) communicates

with the Gemini API by preparing a prompt based on the emotion analysis, sending it to the API, and displaying the response in Streamlit, while also handling any potential errors during the API interaction.

The DeepFace Python module is a comprehensive framework for facial recognition and facial attribute analysis using deep learning. It provides a high-level interface to process and analyze facial data through a variety of pre-trained deep learning models. The module is designed to make facial recognition, emotion detection, age estimation, and gender detection straightforward by abstracting the complexities of deep learning. DeepFace supports multiple state-of-the-art face recognition models, including VGG-Face, Google FaceNet, OpenFace, Facebook DeepFace, and Dlib. Users can choose any of these models depending on their specific requirements. It also integrates with various facial attribute models for emotion analysis, enabling it to classify emotions such as happiness, sadness, anger, and more. The module uses TensorFlow and Keras as its backend frameworks to run deep learning operations efficiently. At its core, DeepFace operates by detecting and aligning faces in an image using a pre-trained facial detector, ensuring consistent positioning before feeding the face into recognition or analysis pipelines. This preprocessing step is critical because it eliminates variability caused by head tilt, lighting, and facial orientation. For emotion detection specifically, DeepFace uses a softmax classifier at the final layer of its models to predict probabilities for different emotions. DeepFace is also known for its flexibility and simplicity. Users can invoke facial analysis or recognition with a single line of code. Additionally, the module includes an **analyze** function for emotion, age, and gender detection, as well as a **verify** function to compare two images for identity matching. It also allows users to toggle between facial detectors, including OpenCV, SSD, and MTCNN, depending on their hardware and processing needs. The module emphasizes usability with optional parameters for tuning accuracy, speed, and error handling. For example, users can enable or disable facial detection enforcement, which is useful in scenarios where non-human faces or partial occlusions are present. The ability to batch process multiple images and integrate seamlessly with large datasets makes DeepFace a preferred choice for production-level applications. Overall, DeepFace abstracts the complexities of implementing deep learning models for facial analysis, providing a user-friendly and powerful tool for a variety of facial recognition and emotion detection tasks.

In a mental health disorder detection application, DeepFace plays a pivotal role in extracting facial expressions from images and videos, which can be critical for assessing emotional states and providing insights into potential mental health concerns. The module leverages deep learning models trained to recognize a wide array of facial expressions, such as happiness, sadness, anger, surprise, fear, and disgust. These expressions are vital indicators of a person's emotional state, which, when analyzed over time, can offer valuable clues about their mental health.

DeepFace first processes the input image or video by detecting faces through a facial recognition model, ensuring that only human faces are analyzed. Once faces are detected, the module aligns and normalizes the images to account for variations in lighting, angles, and head positions, ensuring that the analysis remains accurate regardless of how the subject is positioned or oriented. DeepFace's emotion detection model then works by analyzing the subtle changes in facial muscle movements that correspond to different emotional expressions. For instance, a furrowed brow and tight lips may indicate anger, while a smiling face may suggest happiness. These expressions are captured and classified using convolutional neural networks (CNNs), which are adept at identifying patterns in visual data. DeepFace outputs these emotions as probability scores, allowing the application to interpret the likelihood of each emotion being present in a given frame. In a video context, the module processes each frame sequentially, enabling the tracking of emotional transitions over time, which is particularly valuable in detecting mood fluctuations or episodes commonly associated with mental health conditions like depression, anxiety, or bipolar disorder. The ability to extract and analyze these facial expressions in real-time or from pre-recorded content empowers mental health disorder detection systems to better understand the emotional well-being of individuals. By assessing facial expressions in conjunction with other data points, such as speech or text analysis, DeepFace helps build a comprehensive profile of a person's emotional state. This can aid in detecting mental health issues early, identifying stress triggers, or tracking the effectiveness of therapeutic interventions. In this way, DeepFace's emotion detection capabilities become an essential tool in a mental health disorder detection application, providing critical insights that can inform diagnosis and personalized care plans.

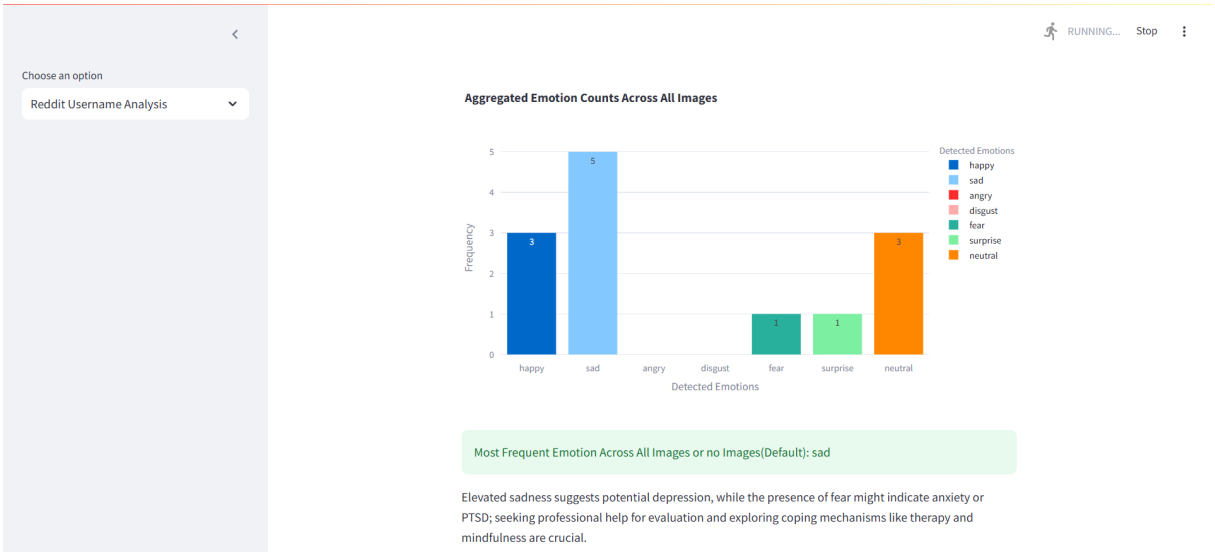


Figure 65: Result from the emotion analysis of facial expression

Image Captioning

```
# Function to load the model (cached for efficiency)
@st.cache_resource
def load_model():
    model_name = "nlpconnect/vit-gpt2-image-captioning"
    model = VisionEncoderDecoderModel.from_pretrained(model_name
       )
    feature_extractor = ViTImageProcessor.from_pretrained(
       model_name)
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    return model, feature_extractor, tokenizer

# Load the model
IDmodel, IDfeature_extractor, IDtokenizer = load_model()

# Set device to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "
   cpu")
IDmodel.to(device)

# Function to generate caption
def generate_caption(image):
    # Preprocess the image
    if image.mode != "RGB":
        image = image.convert(mode="RGB")
    pixel_values = IDfeature_extractor(images=image,
       return_tensors="pt").pixel_values
    pixel_values = pixel_values.to(device)

    # Generate caption (you can adjust max_length and num_beams
       as needed)
    with torch.no_grad():
        output_ids = IDmodel.generate(pixel_values, max_length
           =16, num_beams=4)
    caption = IDtokenizer.decode(output_ids[0],
       skip_special_tokens=True)
    return caption
```

The provided code implements two key functionalities: image captioning and mental health text classification. For image captioning, it loads a pre-trained VisionEncoderDecoder model (vit-gpt2-image-captioning) using Hugging Face tools, processes input images, and generates captions. The **vit-gpt2-image-captioning** model combines two advanced architectures, the Vision Transformer (ViT) and GPT-2, to generate descriptive captions for images. It functions as an encoder-decoder system where ViT acts as the encoder to process and extract meaningful features from the image, and GPT-2 acts as the decoder to generate coherent natural language

captions based on those features. The Vision Transformer (ViT) is a transformer-based model for image recognition. Unlike traditional convolutional neural networks (CNNs), which process images in a grid-like fashion, ViT splits the image into smaller patches (e.g., 16x16 pixels). These patches are flattened into vectors and passed through a transformer encoder. This encoder treats each patch as a token, similar to how words are tokens in natural language processing. By leveraging self-attention mechanisms, ViT captures both local and global dependencies in the image, resulting in a comprehensive feature representation. These features are passed to the decoder for further processing. GPT-2, a pre-trained generative language model, serves as the decoder. It uses the features extracted by ViT to generate captions word by word. Transformers like GPT-2 rely heavily on self-attention and positional encodings to understand the relationships between words in a sequence. In this context, GPT-2 uses the visual feature embeddings as the initial context and generates text sequentially. At each step, it predicts the next word in the caption based on the previous words and the image features, ensuring that the captions are contextually relevant and grammatically accurate. Transformers are crucial here because they allow the model to handle complex dependencies and interactions in both image and text data. ViT uses transformers to learn rich visual representations by modeling relationships between patches in the image, while GPT-2 leverages transformers to generate fluent and coherent text based on the encoded image features.

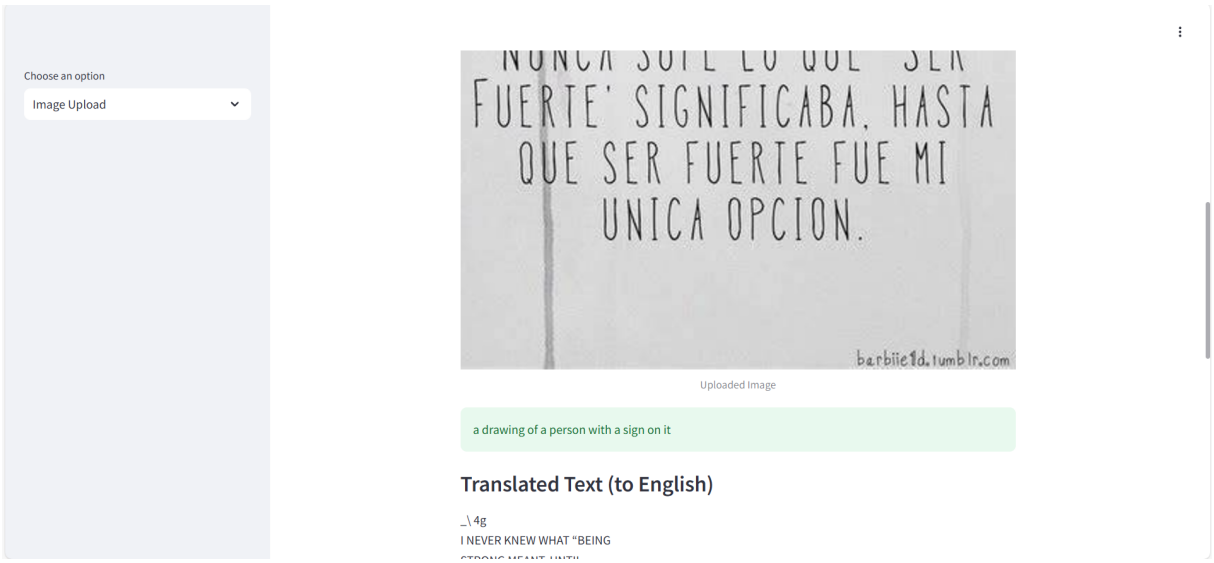


Figure 66: Generate Image Caption

Knowledge Graph Creation

```
def create_knowledge_graph(input_text, classifications,
   probabilities):
    # Initialize a directed graph
    graph = nx.DiGraph()
    # Add the central node (input text)
    graph.add_node("Input␣Text", size=1500, color="#ADD8E6")  #
       Light blue for the central node
    # Normalize probabilities for better edge length scaling
    max_prob = max(probabilities)
    min_prob = min(probabilities)
    prob_scaled = [(1 - (p - min_prob) / (max_prob - min_prob))
       + 0.1 for p in probabilities]  # Invert probabilities for
        distances

    # Add nodes for classifications and connect them to the
       input text
    for classification, probability, scaled_prob in zip(
       classifications, probabilities, prob_scaled):
        prob_percentage = f"{probability␣*␣100:.2f}%"
        graph.add_node(classification, size=1000, color="#E6E6FA
           ")  # Light lavender for classification nodes
        graph.add_edge("Input␣Text", classification, weight=
           scaled_prob, label=prob_percentage)

    # Extract node colors and sizes
    node_colors = [data["color"] for _, data in graph.nodes(data
       =True)]
    node_sizes = [data["size"] for _, data in graph.nodes(data=
       True)]
    # Compute positions using spring layout, scaling edge
       lengths with inverted probabilities
    pos = nx.spring_layout(graph, seed=42, weight='weight')

    # Draw the graph
    plt.figure(figsize=(12, 8))
    nx.draw(
        graph, pos, with_labels=True, node_size=node_sizes,
           node_color=node_colors,
        font_size=10, font_weight="bold", edge_color="gray")

    # Add edge labels for probabilities
    edge_labels = nx.get_edge_attributes(graph, "label")
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=
       edge_labels, font_color="red")
    # Display the plot in Streamlit
    st.pyplot(plt)
```

This code defines a function to create a knowledge graph that visualizes the relationships between an input text and mental health classifications along with their associated probabilities. It uses the NetworkX library to construct a directed graph, where the input text forms the central node connected to classification nodes, each representing a mental health issue. Probabilities are used to determine edge lengths dynamically, ensuring that classifications with higher probabilities are closer to the central node. The Matplotlib library is used to render the graph, with color-coded nodes for improved readability. The final visualization is displayed within a Streamlit application, allowing interactive exploration of the graph.

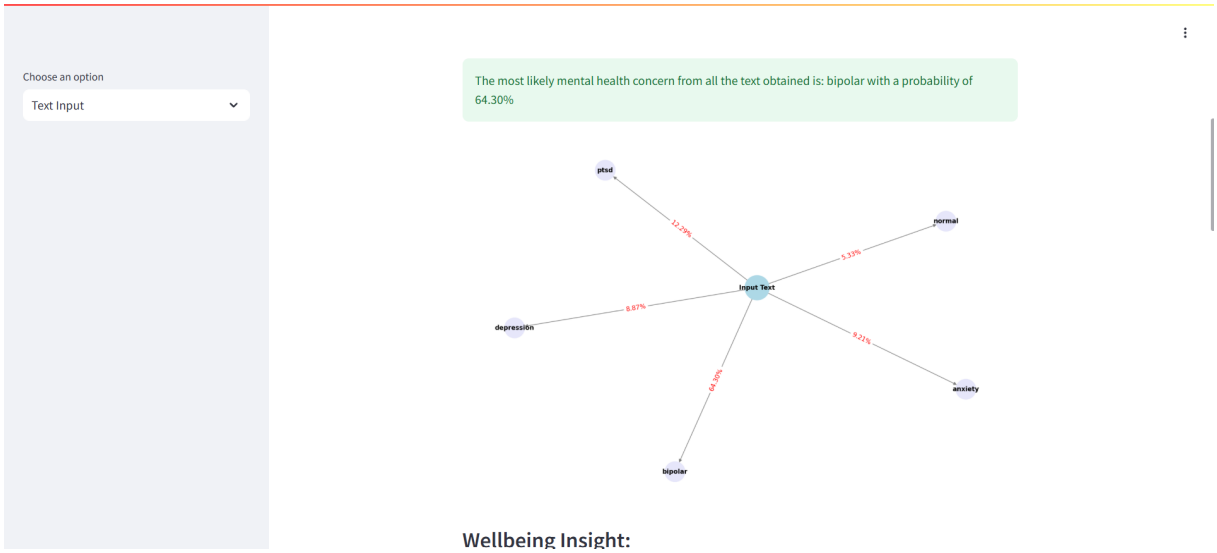


Figure 67: Knowledge Graph from classification