Final Report:

Sentiment Analysis for Early Detection of Mental Health Status

Introduction

The increasing prevalence of mental health disorders necessitates innovative approaches for early detection and timely intervention. Leveraging advancements in Natural Language Processing (NLP), this project aims to develop a multi-class classification model to identify various mental health statuses from textual data. By analyzing statements labeled with mental health conditions, the model will assist healthcare providers in recognizing signs of mental health issues, facilitating proactive measures for patient care.

Problem Statement

Accurate and early detection of mental health conditions is critical for effective treatment and intervention. Traditional methods of diagnosis can be time-consuming and subjective, often relying heavily on in-person assessments. By utilizing a comprehensive dataset of over 50,000 statements, this sentiment analysis aims to enable healthcare providers to recognize potential mental health issues and take appropriate actions as early as possible.

Objectives

1. **Develop a Multi-Class Classification Model**: Build and implement a predictive model to accurately classify various mental health statuses, such as "Normal," "Depression," "Suicidal," "Anxiety," "Bipolar," "Stress," and "Personality disorder," based on textual statements.
2. **Explore and Evaluate Machine Learning Algorithms**: Investigate and compare the performance of Bernoulli Naive Bayes, Logistic Regression, and Decision Tree Classifier to identify the most effective model for sentiment analysis in mental health.
3. **Facilitate Early Detection of Mental Health Issues**: Create a prototype tool that enables healthcare providers to monitor and trigger timely interventions based on predicted mental health statuses, ultimately improving patient care outcomes.

Dataset Overview

The dataset comprises over 50,000 statements derived from various sources and labeled with distinct mental health statuses. It includes the following features:

unique_id: A unique identifier for each entry.

statement: The textual data or post representing an individual's mental health status.

mental_health_status: Categorical labels indicating the mental health condition, including:

i. **Normal**, ii. **Depression**, iii. **Suicidal**, iv. **Anxiety**, v. **Bipolar**, vi. **Stress**, vii. **Personality Disorder**

The dataset has been compiled by **Suchintika Sarkar** and integrates information from multiple Kaggle datasets, ensuring a rich and diverse resource for analysis.

Data Wrangling:

The dataset initially contained 53,043 rows and two columns, "statement" and "status," with some missing entries. Upon examining the target variable, it became clear that the dataset is imbalanced, which requires attention before model building. As shown in the accompanying bar graph, the "Normal" status appears most frequently, with 16,351 instances, whereas "Personality Disorder" appears only 1,201 times, resulting in a 13.6:1 imbalance ratio between the most and least common categories.

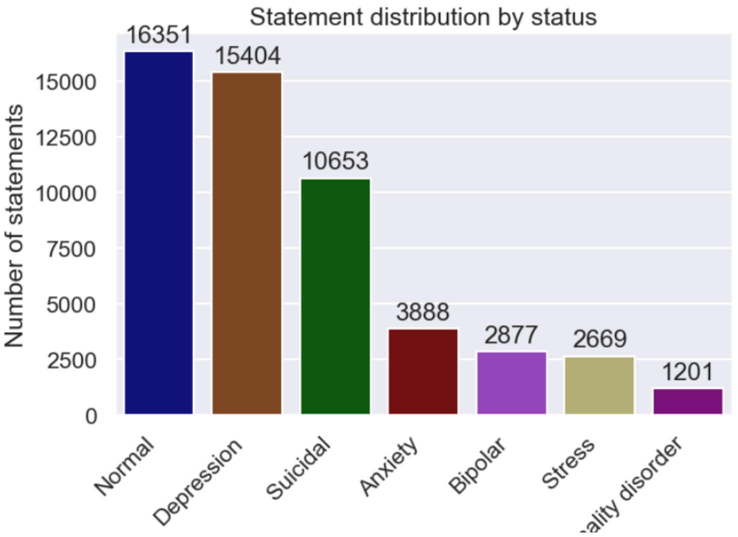


Figure 1. Statement distribution by status in Target Variables

Missing values and imputation:

Given the imbalance, I explored whether missing data might correlate with the less-represented statuses, specifically "Anxiety," "Bipolar," "Stress," and "Personality Disorder." This potential relationship suggested that missing entries might hold significance linked to a person's mental status. To investigate, I examined the distribution of missing data across these target variables. As shown in the figure below, the missing statements were disproportionately represented by "Anxiety" (1.21% of total), "Bipolar" (3.48%), "Stress" (3.07%), and "Personality Disorder" (10.32%). Given these patterns, I opted to impute the missing statements with a placeholder value of "unfilled".

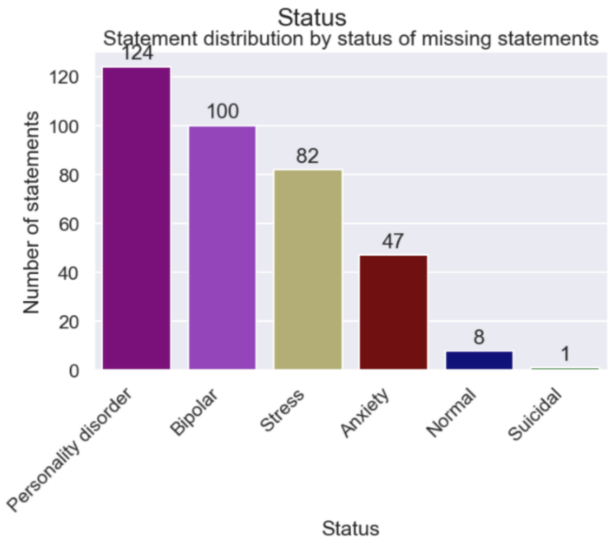


Figure 2. Statement distribution by status of missing statements

Preprocessing of data:

I developed a text preprocessing pipeline to clean and standardize text data in the statement column of the dataset. To ensure consistency, I first converted the text to lowercase, removed HTML tags, and handled accented characters. I then removed any URLs, usernames, and punctuation, with the exception of underscores. I also removed words containing numbers and reduced sequences of repeating characters to improve readability.

To handle common contractions, I applied a custom function that expanded contractions based on a predefined mapping. Additionally, I removed newline characters and extra whitespace. Once the cleaning process was complete, I applied the function to each entry in the statement column, resulting in a new cleaned column, "statement_cleaned", which contains standardized and easily interpretable text data.

Removing Stopwords:

I evaluated the impact of stopword removal on sentiment analysis accuracy. Initially, I considered that removing stopwords might negatively affect accuracy, as it could alter the meaning of statements like

"I like sleeping" versus "I do not like sleeping," leading to potentially identical interpretations. However, after testing both approaches, I observed an increase in accuracy when stopwords were removed. Consequently, I incorporated stopword removal into the preprocessing pipeline.

To implement this, I used a set of English stopwords and defined a function to remove them from each text entry in the “statement_cleaned”, column. This function tokenizes the text, filters out stopwords, and then rejoins the tokens. The cleaned statements were then updated in the “statement_cleaned”, column for further analysis.

Tokenization:

I tokenized each cleaned statement in the “statement_cleaned” column to further prepare the text data for analysis. Using the word_tokenize function, I split each statement into individual tokens and stored the resulting tokens in a new column, tokens, within the sentiment_df_cleaned DataFrame. This step will support subsequent analyses by enabling word-level processing of each statement.

Stemming and Lemmatization:

Stemming is the process of reducing words to their base or root form, typically by removing prefixes or suffixes. It’s a common technique in natural language processing (NLP) to help standardize words so that variations of a word (like "running," "ran," and "runner") are treated as the same base form ("run").

Popular Stemming Algorithms

Porter Stemmer: A widely used, rule-based algorithm that removes common suffixes. It’s simple and fast but can be aggressive, sometimes producing stems that aren’t real words (e.g., "univers" for "universe").

Snowball Stemmer: An extension of Porter with more rules, designed to handle different languages and offer better accuracy.

Lancaster Stemmer: A more aggressive algorithm that often reduces words to very short stems, which may not always be intuitive.

Lemmatization is the process of reducing words to their base or dictionary form, called a "lemma," while keeping the meaning of the word intact. Unlike stemming, which simply cuts off prefixes or suffixes, lemmatization considers the word's context, such as its part of speech, to produce grammatically correct root words. For instance, lemmatizing “running” would yield “run,” and “better” would become “good.”

Popular Lemmatization Techniques

SpaCy Lemmatizer: SpaCy is known for its speed and efficiency, especially for larger text datasets. It uses robust language models to automatically determine parts of speech, making it versatile and accurate for lemmatizing in context. It’s a great choice when processing larger datasets or requiring high accuracy without specifying parts of speech manually.

WordNet Lemmatizer (NLTK): This is one of the most widely used lemmatizers due to its simplicity and the extensive WordNet lexical database. While it needs the part of speech for optimal performance, it’s an excellent choice for small- to medium-sized datasets where specifying parts of speech is feasible.

Stanford CoreNLP Lemmatizer: This tool is highly accurate, making it ideal for complex NLP tasks that require high precision. Stanford CoreNLP’s lemmatizer uses probabilistic language models, ensuring quality lemmatization. It’s best suited for projects where accuracy outweighs speed and where resources support using Java-based tools.

To enhance text preprocessing, I applied both stemming and lemmatization to the tokenized statements. Using the Snowball Stemmer, I created a function to stem each token in the tokens column, then joined the stemmed words into a single string and stored the result in a new column, “tokens_joined”.

Similarly, I defined a lemmatization function with the WordNet Lemmatizer, which generated the “tokens_joined_lemma” column. After comparing the results of stemming and lemmatization, I found that stemming provided a slight improvement in performance. Therefore, I proceeded with stemming as part of the final preprocessing pipeline.

Feature Engineering:

I observed that the longest "words" in the statements were often not actual words but instead random gibberish, possibly entered either intentionally or due to technical error. While these entries might offer insight into the user's mental state or data entry quality, calculating features like character counts and word counts directly from the original, unprocessed statements allowed me to capture these nuances. This approach retained information about the current state of the data, which may reflect patterns in data quality or user behavior. Consequently, I decided to compute character count and similar features from the original statements to preserve nuanced information reflecting personal mental states, which could be valuable for sentiment analysis. I engineered several text-based features from these raw statements:

1. **Character Count**: Calculated the total number of characters in each statement.
2. **Word Count**: Counted the number of words in each statement.
3. **Sentence Count**: Determined the number of sentences by tokenizing each statement into sentences.
4. **Average Characters per Word**: Computed as the ratio of character count to word count.
5. **Average Characters per Sentence**: Calculated by dividing the character count by the sentence count.
6. **Average Words per Sentence**: Derived as the ratio of word count to sentence count.

After feature creation, I filtered out statements with a character count of 1 or less, sorted the data by character count in descending order, and reset the index for consistency. This approach ensured that I could analyze patterns in sentiment without losing subtle information contained in the text's original form.

Exploratory Data Analysis (EDA):

To effectively visualize the data while accounting for its skewness, long-tail entries were excluded from the analysis. This approach facilitated clearer visualizations of the distributions for character counts, word counts, sentence counts, and average metrics, allowing for a better understanding of the central tendencies and variations within the dataset which can be visualized by the figures below.

The analysis of skewness for the newly added features indicates significant right skewness across all metrics, with values exceeding 1. Notably, features such as sentence_count and avg_character_per_word exhibit extreme skewness, suggesting the presence of outliers or instances with exceptionally high values that disproportionately affect the mean.

To mitigate this skewness and achieve a more normalized distribution, I applied Log Transformation. This technique is particularly effective for reducing skewness in features characterized by positive values, helping to stabilize variance and improve the performance of subsequent analyses and models.

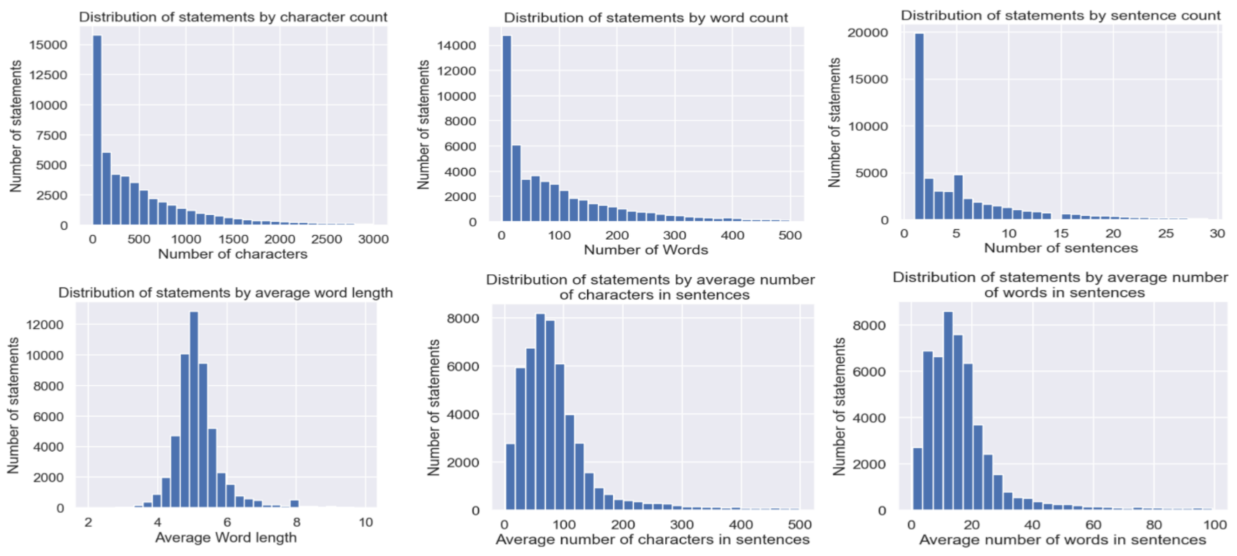


Figure 3: Distribution of newly added feature values

Target variable with respect to three newly generated features in 3D Scatterplot

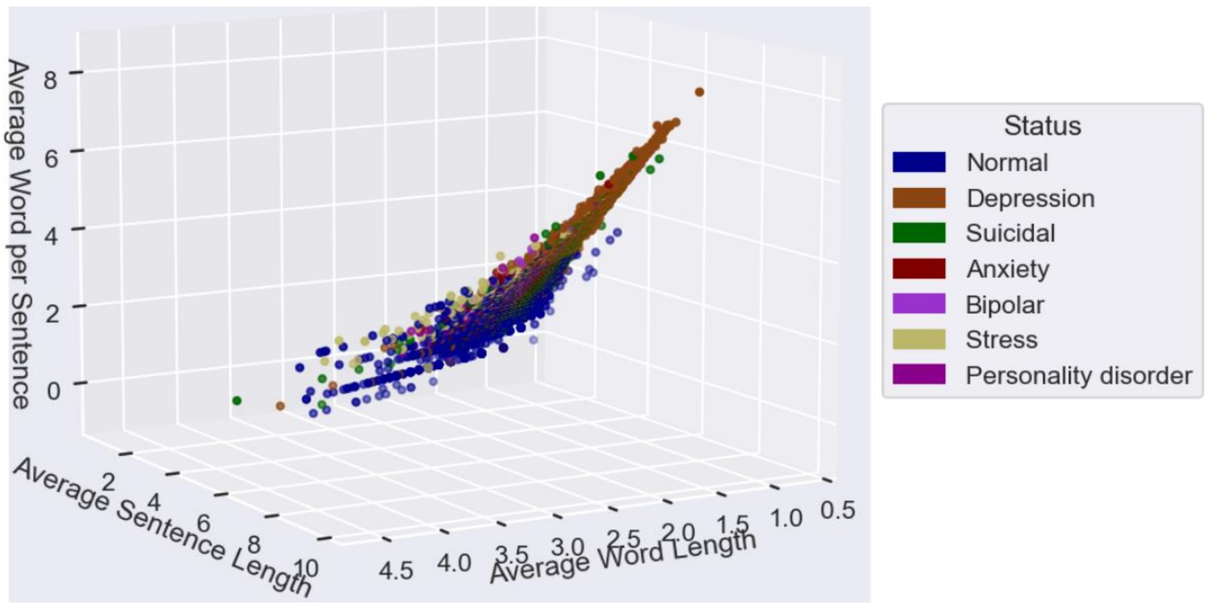


Figure 4: Target variable with respect to three newly generated features in 3D Scatterplot

To assess the significance of the newly added features, a 3D scatter plot was created using the logarithmic values of average word length, average sentence length, and average words per sentence. The plot revealed a distinct structure, with most entries categorized as "Normal" clustering at the lower regions, forming a nearly pyramid-like shape, while entries labeled as "Depression" were situated at the upper part of the plot. This spatial distribution highlights the relevance of the added features in differentiating between various mental health statuses, reinforcing their importance in the analysis. The color-coded representation, based on the mapped statuses, further elucidated the relationship between the features and the mental health conditions, emphasizing the utility of these features in the sentiment analysis.

Word Clouds to visualize the relative importance of words in each status category

Creating word clouds effectively visualizes the frequency and importance of words associated with different mental health statuses in the dataset. By displaying words in sizes that correspond to their prevalence, word clouds allow for quick identification of key terms within each status category.

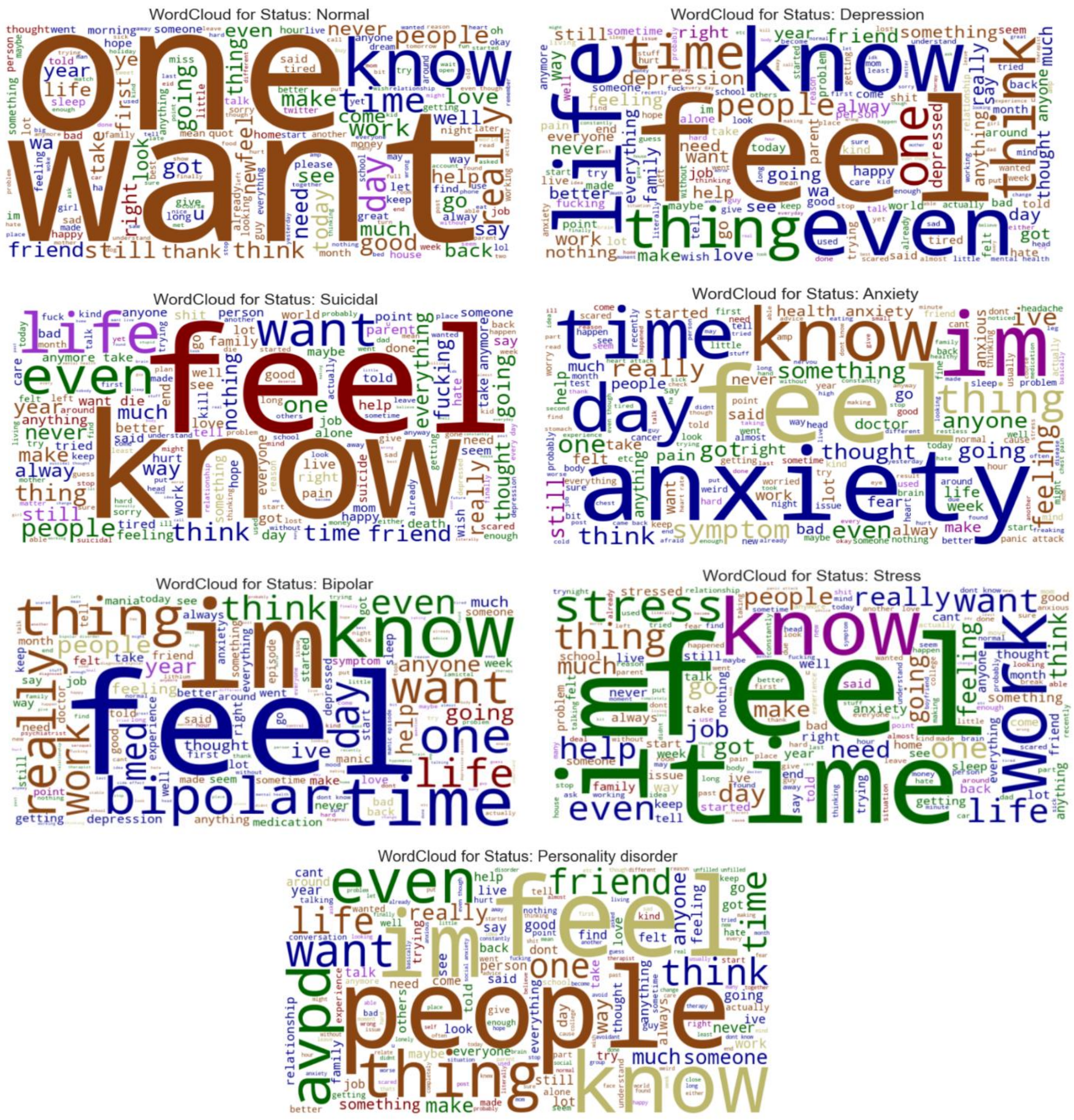


Figure 5: Word Clouds to visualize the relative importance of words in each status category

Many words such as "**feel**", "**know**" etc. are used with overlap across different statuses. It has to be taken care of while building a model. TF-IDF (Term Frequency-Inverse Document Frequency) is an excellent way to handle the overlap of words across different categories in NLP. TF-IDF helps distinguish words that are common across all categories from words that are more important and unique to specific categories, which can significantly improve the performance of classification models.

Model Development:

After separating target and features from the cleaned dataset, the target variable, “status”, was encoded using the LabelEncoder to convert categorical labels into numerical format. Subsequently, the dataset was split into training and testing sets using an 80-20 ratio, ensuring stratification based on the target variable to maintain the distribution of categories across both sets.

To handle the feature space containing both numerical and textual data, I employed the TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer to transform the tokenized text into a numerical representation. This approach accounts for the observed overlap in words across various target categories, as identified in the earlier word cloud analysis.

TF-IDF vectorization

TF-IDF is a powerful technique for text analysis that helps to identify the significance of words in documents relative to a larger corpus, making it invaluable for various natural language processing tasks. TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents (corpus). It is commonly used in text mining and information retrieval, particularly for tasks like document classification, clustering, and searching.

Term Frequency (TF): Measures how frequently a term occurs in a document.

$$TF(t,d) = \frac{Number\ of\ times\ term\ t\ appears\ in\ document\ d}{Total\ number\ of\ terms\ in\ document\ d}$$

A higher TF indicates that the term is more important in that specific document.

Inverse Document Frequency (IDF): Measures how important a term is across the entire corpus.

$$IDF(t) = \ ln\left(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ containing\ term\ t}\right)$$

A higher IDF value indicates that the term is rare across documents, making it more significant when it does appear.

The TF-IDF score combines these two metrics:

$$TF - IDF(t,d) = TF(t,d) \times IDF(t)$$

This means that common words across many documents (like "the", "is", "and") will have low TF-IDF scores, while rare words that appear frequently in a specific document will have high TF-IDF scores.

I configured the TF-IDF vectorizer with a range of n-grams from 1 to 2, set a maximum document frequency of 0.8, and limited the features to the top 50,000. After fitting the vectorizer to the training data (tokens_joined), I transformed both the training and testing sets accordingly.

Although I experimented with both stemmed and lemmatized data during this process, I ultimately chose to proceed with stemming due to its superior accuracy in model performance. This decision reflects a focus on optimizing the feature representation for better classification results.

To prepare the data for modeling, I scaled the numerical features using the StandardScaler, which standardizes the features by removing the mean and scaling to unit variance. This process was applied exclusively to the numeric columns, which include the log-transformed metrics such as character count, word count, sentence count, and averages of characters and words per sentence.

After scaling the numerical data, I combined the transformed textual features from the TF-IDF vectorizer with the scaled numerical features. This was accomplished using the hstack function, which horizontally stacks the two matrices together, resulting in a unified feature set for both the training and testing datasets:

- X_train_combined: A combination of the TF-IDF features and the scaled numeric features for the training set.
- X_test_combined: A similar combination for the testing set.

This combined feature set is now ready for use in the classification model, allowing the model to leverage both textual and numerical information effectively.

Resampling the data to fix the category imbalance

I experimented with different methods to address class imbalance in the dataset. While I initially applied the SMOTE (Synthetic Minority Over-sampling Technique) method, I found that the RandomOverSampler provided significantly better results in terms of model performance.

The RandomOverSampler effectively increased the number of instances in the minority classes by randomly duplicating existing examples, which allowed the model to better learn from these underrepresented classes. This approach ultimately led to improved accuracy and overall performance compared to the results obtained using SMOTE, which generates synthetic samples but may not always capture the underlying distribution effectively.

Model1: Bernoulli Naive Bayes

I performed hyperparameter tuning for the Bernoulli Naive Bayes model using GridSearchCV to optimize the alpha parameter, which controls the smoothing of the model. The parameter grid included a range of values: [0.0001, 0.001, 0.01, 0.1, 0.5, 1.0, 5.0, 10.0].

The best parameter found was alpha = 0.0001, resulting in a best cross-validated accuracy score of approximately 78.20%. After training the model with these optimal parameters, I evaluated its performance on the test set.

The final accuracy on the test set was around 63.76%. The detailed classification report indicated varying performance across the classes, with precision, recall, and f1-scores revealing challenges in classifying certain categories. Notably, classes with fewer samples, such as those corresponding to "Bipolar" and "Stress," had lower precision and recall scores, highlighting the model's struggle with imbalanced classes. Overall, the tuned Bernoulli Naive Bayes model demonstrated reasonable accuracy, but further enhancements could be beneficial, especially in improving the classification of less frequent categories.

Interestingly, while the GridSearchCV indicated that the optimal alpha was 0.0001 with a cross-validated accuracy of approximately 78.20%, the accuracy on the test set for this same alpha was significantly lower at 63.76%.

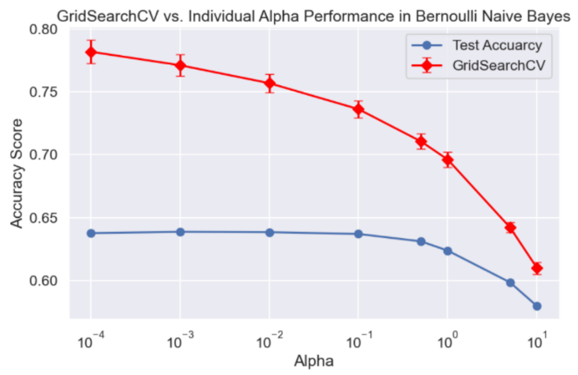


Figure 6: Accuracy Scores in GridSearchCV vs direct test score for different Alpha values.

This discrepancy suggests that the model may have been overfitting during cross-validation or that the training data used in GridSearchCV did not accurately represent the diversity of the test data. The results from individual testing of alpha values show a slight increase in accuracy for 0.001 and 0.01. This indicates that further investigation into model performance and potential overfitting may be necessary, particularly given the complexities of the dataset and class imbalances.

The confusion matrix for the Bernoulli Naive Bayes model indicates significant challenges in accurately classifying certain statuses, particularly "Depression" and "Suicidal".

The largest off-diagonal counts, where 575 instances of "Suicidal" were misclassified as "Depression", and 715 instances of "Depression" were misclassified as "Suicidal". This indicates a significant overlap in the features associated with these two categories. The model struggles particularly with distinguishing between these two statuses, which may suggest that the features used (especially in the TF-IDF representation) do not adequately capture the differences in language or context between "Depression" and "Suicidal". While other classes show a more balanced performance, the misclassifications can still lead to concerns, especially when considering the sensitive nature of these categories.

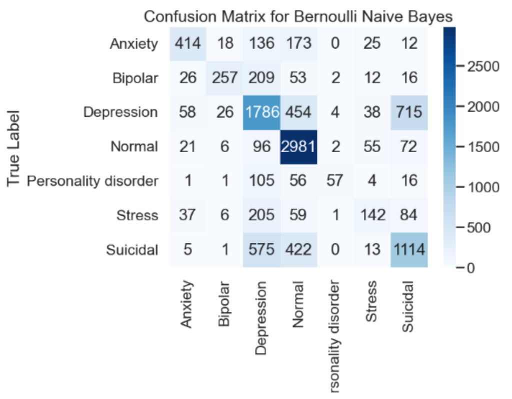


Figure 7: Confusion Matrix for Bernoulli Naïve Bayes Model for Alpha = 0.001.

Model2: Logistic Regression classifier

I performed hyperparameter tuning for the Logistic Regression model using GridSearchCV to optimize the regularization parameter C, which controls the strength of regularization in the model. The parameter grid included a range of values: [0.5,1.0,5.0,10.0,100.0] The best parameter found was C=10.0, resulting in a best cross-validated accuracy score of approximately **90.75%**.

After training the model with these optimal parameters, I evaluated its performance on the test set. The final accuracy on the test set was around **74%,** however just like Bernoulli Naive Bayes Model, the highest accuracy obtained from the individual evaluations was **74.89%** for **C=1.0**, significantly lower than the performance observed in GridSearchCV. This may be due to resampling process which can introduce variations in model performance, especially in models sensitive to class distributions.

The detailed classification report indicated varying performance across the classes, with precision, recall, and f1-scores revealing challenges in classifying certain categories. Notably, Class 3 ("Normal") exhibited strong performance with a precision of **90%** and recall of **91%**, while lower-performing classes,

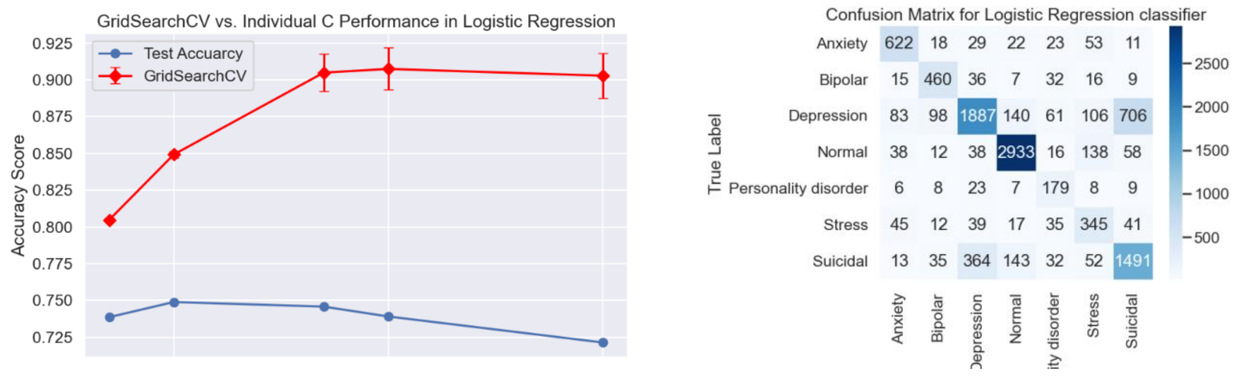


Figure 8: a) Accuracy Scores in GridSearchCV vs direct test score for different C values. b) Confusion Matrix for Logistic Regression classifier Model for C = 1.0.

such as Class 4 ("Personality disorder") and Class 5 ("Stress"), struggled with lower precision and recall scores, highlighting the model's difficulty in distinguishing these categories.

The confusion matrix reveals that the model has significant difficulty distinguishing between certain classes, particularly "Depression" and "Suicidal" (class 6), as indicated by the off-diagonal counts of 364 (575 in **Bernoulli Naive Bayes)** and 706 (715 in **Bernoulli Naive Bayes)**, respectively. Although this model improves vastly, it struggles with classifying these categories accurately.

In conclusion, while the tuned Logistic Regression model achieved reasonable performance on the test set, the confusion matrix indicates areas for improvement, especially in differentiating between overlapping categories. This highlights the potential need for further refinements, such as additional feature engineering or the exploration of alternative models, to enhance classification accuracy for less represented classes.

Model3: Decision Tree Classifier

I attempted to optimize the Decision Tree Classifier using GridSearchCV; however, I stopped midway because the computational time was too lengthy. To expedite the process, I first adjusted the min_samples_split parameter by testing various values, ultimately selecting min_samples_split = 5 based on its performance on test data.

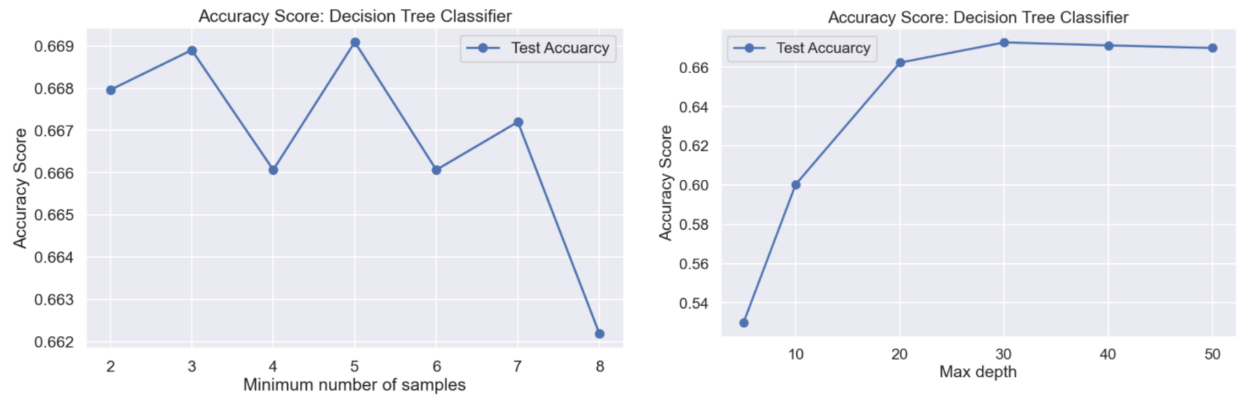
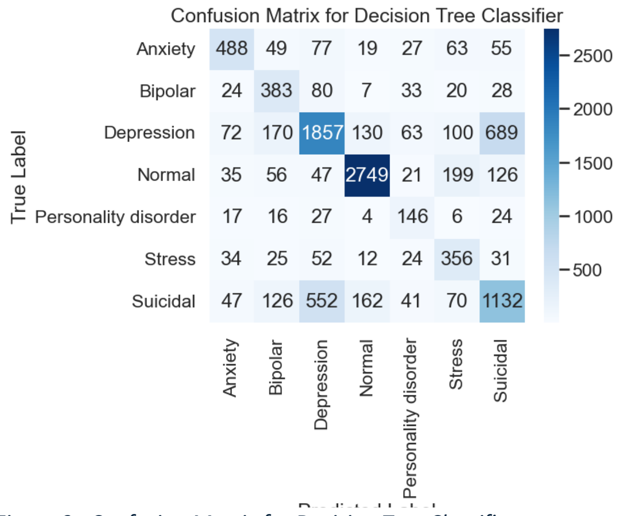


Figure 10: Optimization of min_samples_split and Max_depth for Decision Tree Classifier using accuracy score as metrics.

Following that, I shifted my focus to the max_depth parameter, evaluating several options to determine the optimal depth. The best accuracy was observed with max_depth = 30. This strategy allowed me to streamline the tuning process and improve the model's performance while managing computational resources effectively.

The Decision Tree Classifier showed improvements over the Tuned Bernoulli Naive Bayes but still encountered issues in distinguishing between the "Depression" and "Suicidal" categories, highlighting the need for further refinement in model training or the potential integration of additional features that could aid in better class separation.



	Anxiety	Bipolar	Depression	Normal	Personality disorder	Stress	Suicidal
Anxiety	488	49	77	19	27	63	55
Bipolar	24	383	80	7	33	20	28
Depression	72	170	1857	130	63	100	689
Normal	35	56	47	2749	21	199	126
Personality disorder	17	16	27	4	146	6	24
Stress	34	25	52	12	24	356	31
Suicidal	47	126	552	162	41	70	1132

Figure 9: Confusion Matrix for Decision Tree Classifier at min_samples_split = 5, and max_depth = 30.

Conclusions:

- **Model Performance and Accuracy**: The **Logistic Regression model** outperformed both the **Decision Tree Classifier** and the **Tuned Bernoulli Naive Bayes**, achieving the highest accuracy of approximately **74.89%**. This suggests that Logistic Regression is a more effective choice for this classification task, likely due to its ability to capture linear relationships and effectively handle the features present in the dataset.

- **Misclassification Challenges**: Both the Decision Tree Classifier and the Tuned Bernoulli Naive Bayes exhibited significant misclassification issues, particularly in differentiating between the "Depression" and "Suicidal" classes. This indicates a potential overlap in feature representation for these categories, suggesting that further feature engineering, additional data, or more complex models may be necessary to improve classification performance in these areas.

- **Computational Efficiency Considerations**: The decision to manually test various hyperparameters for the Decision Tree Classifier, rather than relying on GridSearchCV, highlights the importance of balancing model optimization with computational efficiency. While GridSearchCV can provide thorough parameter tuning, it can be time-consuming; thus, testing specific parameters directly can yield reasonably good results without excessive computational costs, as demonstrated by the results achieved with min_samples_split and max_depth values.

Future Recommendations:

- **Expand Dataset with Additional Data Collection**: Increase sample size for low-sampled classes by gathering data from trusted sources via web scraping, APIs, or open datasets, enhancing model robustness.
- **Increase Computational Resources**: Utilize cloud computing (AWS, Google Cloud) or distributed computing (e.g., Dask, Spark) to handle complex models and hyperparameter tuning, allowing for more extensive experimentation.
- **Leverage Pre-Trained Models with Fine-Tuning**: Use pre-trained NLP models (e.g., BERT, RoBERTa) and fine-tune them with your dataset, achieving high accuracy efficiently by capitalizing on existing language understanding.

Credits:

I am deeply grateful to **Suchintika Sarkar** for making this dataset publicly accessible, offering aspiring data scientist invaluable opportunities to practice and enhance their machine learning skills. My heartfelt thanks also go to my Springboard mentor, **Vinit Koshti**, for his insightful guidance and thoughtful support throughout this journey.