week9_code

March 29, 2025

1 Week 2 Ingesting and Exploring the Dataset

[1]:
```
# install wordcloud
!pip install wordcloud
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: wordcloud in /home/jupyter-
geean/.local/lib/python3.12/site-packages (1.9.4)
Requirement already satisfied: numpy>=1.6.1 in
/opt/tljh/user/lib/python3.12/site-packages (from wordcloud) (1.26.4)
Requirement already satisfied: pillow in /opt/tljh/user/lib/python3.12/site-
packages (from wordcloud) (11.1.0)
Requirement already satisfied: matplotlib in /opt/tljh/user/lib/python3.12/site-
packages (from wordcloud) (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in
/opt/tljh/user/lib/python3.12/site-packages (from matplotlib->wordcloud) (1.3.1)
Requirement already satisfied: cycler>=0.10 in
/opt/tljh/user/lib/python3.12/site-packages (from matplotlib->wordcloud)
(0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/opt/tljh/user/lib/python3.12/site-packages (from matplotlib->wordcloud)
(4.55.3)
Requirement already satisfied: kiwisolver>=1.3.1 in
/opt/tljh/user/lib/python3.12/site-packages (from matplotlib->wordcloud) (1.4.8)
Requirement already satisfied: packaging>=20.0 in
/opt/tljh/user/lib/python3.12/site-packages (from matplotlib->wordcloud) (24.1)
Requirement already satisfied: pyparsing>=2.3.1 in
/opt/tljh/user/lib/python3.12/site-packages (from matplotlib->wordcloud) (3.2.1)
Requirement already satisfied: python-dateutil>=2.7 in
/opt/tljh/user/lib/python3.12/site-packages (from matplotlib->wordcloud)
(2.9.0.post0)
Requirement already satisfied: six>=1.5 in /opt/tljh/user/lib/python3.12/site-
packages (from python-dateutil>=2.7->matplotlib->wordcloud) (1.17.0)
```

[2]:
```
# import packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from wordcloud import WordCloud
%matplotlib inline

import warnings
warnings.filterwarnings("ignore")
```

[3]:
```
# change working directory
import os
os.getcwd()  # Get current working directory
os.chdir('..')  # Move up one directory level from notebooks
print(os.getcwd())
#os.chdir('../data') # change to the data folder
```

```
/home/jupyter-geean/cookiecutter-data-science/{{ cookiecutter.repo_name }}
```

[4]:
```
# load the data
df = pd.read_csv('data/Combined Data.csv', index_col=0)
```

[5]:
```
# make a copy and get rid of the missing values
df1 = df.copy()
df1.dropna(inplace = True)
# see the top head of the data
df1.head()
```

[5]:

	statement	status
0	oh my gosh	Anxiety
1	trouble sleeping, confused mind, restless hear…	Anxiety
2	All wrong, back off dear, forward doubt. Stay …	Anxiety
3	I've shifted my focus to something else but I'…	Anxiety
4	I'm restless and restless, it's been a month n…	Anxiety

[6]:
```
# number of missing values
missing_values = df.isnull().sum()

print(missing_values)
```

```
statement    362
status         0
dtype: int64
```

[7]:
```
# get the rows and columns of all of the data
rows,columns = df.shape
print(f"Number of rows: {rows}")
print(f"Number of columns: {columns}")
```

```
Number of rows: 53043
Number of columns: 2
```

[8]:
```
# calculate the number of missing values
rows_with_missing = df[df.isnull().any(axis=1)]
print(rows_with_missing)
```

```
      statement   status
293         NaN  Anxiety
572         NaN  Anxiety
595         NaN  Anxiety
1539        NaN   Normal
2448        NaN   Normal
…          …       …
52838       NaN  Anxiety
52870       NaN  Anxiety
52936       NaN  Anxiety
53010       NaN  Anxiety
53031       NaN  Anxiety

[362 rows x 2 columns]
```

The dataset contains 362 missing values in the 'Statement' column and no missing values for 'Status'.

2 Missing Values -Week 3

Many of the rows have NaNs and represent anxiety and normal. Since there are 53,043 values and there are only 362 rows where there is missing values. We feel that it is best to drop these rows since they represent only 0.7% of the data and as you will see later we have an abundance of "normal" and "anxiety" labeled data.

The dataset includes 52,681 rows and 2 columns after removing missing values.

[9]:
```
# get the rows and columns of the data that drops the missing values
rows,columns = df1.shape
print(f"Number of rows: {rows}")
print(f"Number of columns: {columns}")
```

```
Number of rows: 52681
Number of columns: 2
```

We want to add a column to explore the length of each statement. This can help us quantify the user's input and support further analysis. This will give us an idea on how to preprocess the text and determine tokenization especially for transformer models. Many NLP models, especially those based on deep learning, have limitations on input length so determining the length is important.

[10]:
```
# create a new column that gives the length of each statement
df1['statement_len'] = df1['statement'].apply(lambda x: len(x.split(' ')))
df1.head()
```

[10]:
```
                                           statement   status  statement_len
0                                         oh my gosh  Anxiety              3
1  trouble sleeping, confused mind, restless hear…  Anxiety             10
2  All wrong, back off dear, forward doubt. Stay …  Anxiety             14
3  I've shifted my focus to something else but I'…  Anxiety             11
4  I'm restless and restless, it's been a month n…  Anxiety             14
```

From the output, we can see that this dataset includes 2 variables: statement and status.

The statement variable is a text variable that contains different user inputs.

The status variable represents different emotional statuses, which contain different categories.

The next step is to explore dataset

[11]:
```python
# information about the dataset
'''The class type of the DataFrame.
The range of the index.
The number of columns and their names.
The count of non-null values in each column.
The data type of each column.
The memory usage of the DataFrame.'''

print(df1.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 52681 entries, 0 to 53042
Data columns (total 3 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   statement      52681 non-null  object
 1   status         52681 non-null  object
 2   statement_len  52681 non-null  int64
dtypes: int64(1), object(2)
memory usage: 1.6+ MB
None
```

Statement and status column are object data types. The statement_len column is an integer/numeric datatype.

[12]:
```python
# descriptive statistics
'''count is the number of non-null entries.
unique is the number of unique values.
top is the most frequent value.
freq is the frequency of the most frequent value.'''
df1.describe(include='object').T
```

[12]:
```
           count unique                top   freq
statement  52681  51073  what do you mean?     22
status     52681      7             Normal  16343
```

The 'Statement' column contains 51,073 unique values, indicating that most user inputs are unique. The most frequently appeared statement is "What do you mean?" and occurred 22 times in the dataset. The frequent occurrence of "What do you mean?" suggests significant communication gaps or misunderstandings, indicating areas where individuals feel confused or need more clarity, which is crucial in mental health discussions. This phrase often reflects a state of uncertainty or anxiety, signaling important emotional states. It could also indicate active engagement and a desire for better understanding and it could indicate that individuals need more support or reassurance, aiding in tailoring mental health resources effectively.

The 'Status' column contains 7 unique values and represents different emotion statuses. The most common status is "Normal", suggesting that over 30% of the statements in the dataset fall under this category.

[13]:
```
# Get summary statistics for the 'statement_len' column
summary_statistics = df1['statement_len'].describe()
print(summary_statistics)
```

```
count    52681.000000
mean       113.035914
std        163.501877
min          1.000000
25%         15.000000
50%         62.000000
75%        148.000000
max       6300.000000
Name: statement_len, dtype: float64
```

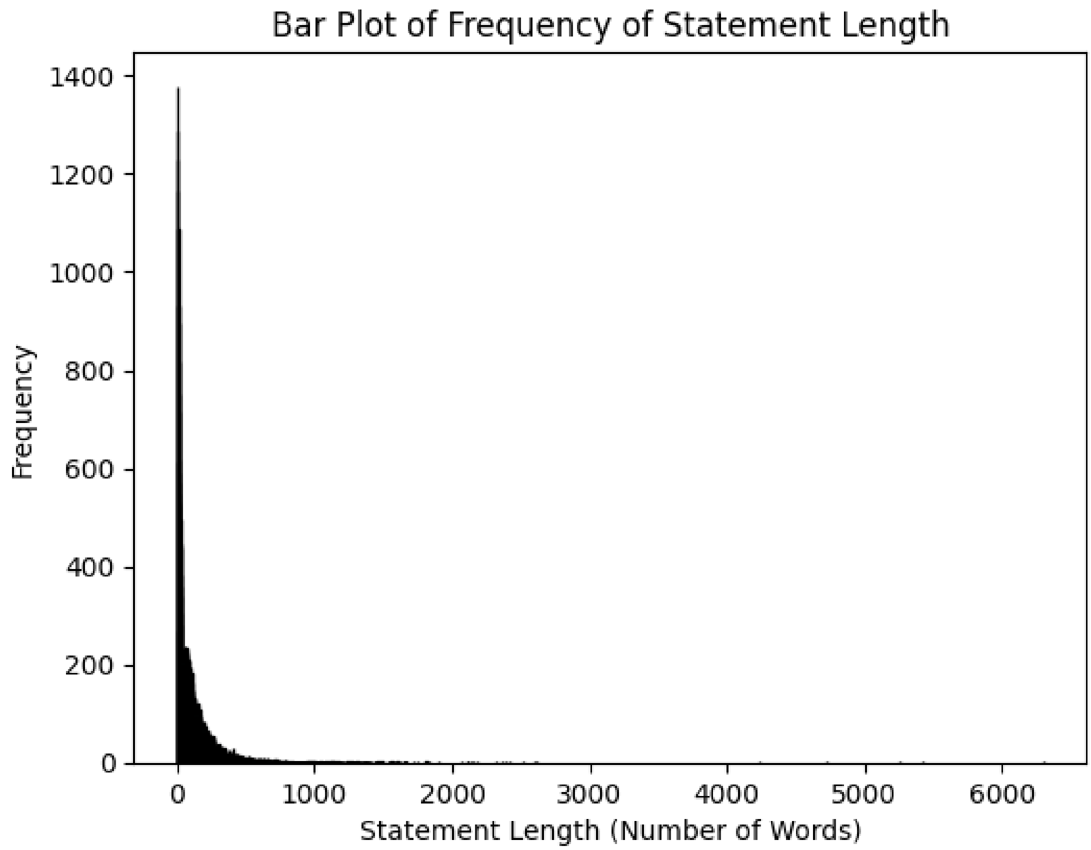
[14]:
```
# Calculate the mode of the 'statement_len' column
mode_value = df1['statement_len'].mode()[0]

print(f"The mode of the 'statement_len' column is: {mode_value}")
```

```
The mode of the 'statement_len' column is: 5
```

The summary statistics for the 'Statement_len' column show the distribution of statment lengths. The average statement contains 113 words with a standard deviation of 163.5 words. The shortest statement only has 1 word, while the longest contains 6300 words. The most frequent statement length is 5 words, indicating that short phrases are commonly used.

The following bar plot of the frequency of statement length visualizes the previous statement.

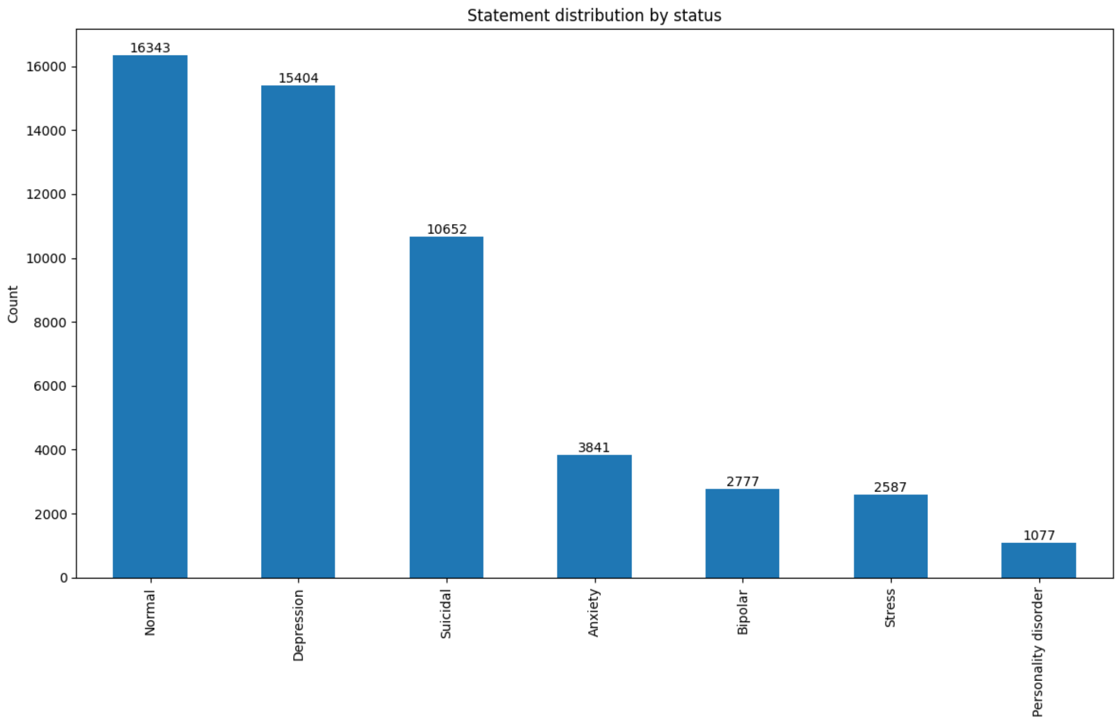
[15]:
```
# Create a bar plot of the frequency of the 'statement_len' column
statement_len_counts = df1['statement_len'].value_counts()

plt.bar(statement_len_counts.index, statement_len_counts.values,␣
    ↪edgecolor='black')
plt.xlabel('Statement Length (Number of Words)')
plt.ylabel('Frequency')
plt.title('Bar Plot of Frequency of Statement Length')
plt.show()
```



The histogram shows that it is a right skewed distribution, which most of the statement length under 1000 words. This means that when we focus on the output length, we should set it to be under 1000.

[16]:
```
# Histogram of Frequency of Statements by Status
plt.figure(figsize=(12,8))

# get the unique status values and their counts
status_counts = df1['status'].value_counts()

# create the bar plot
ax = status_counts.plot(kind='bar')

# add the count labels on top of each bar
for i, v in enumerate(status_counts):
    ax.text(i, v, str(v), ha='center', va='bottom')

plt.title('Statement distribution by status')
plt.xlabel('Status')
plt.ylabel('Count')
```

```
plt.tight_layout()
plt.show()
```



Here is a plot showing distribution by status. Normal is the most common status and contains 16343 data, followed by depression and suicidal, which are the 2nd and 3rd largest portions of the dataset. Personality disorder is the most rare one, which contains 1077 data.

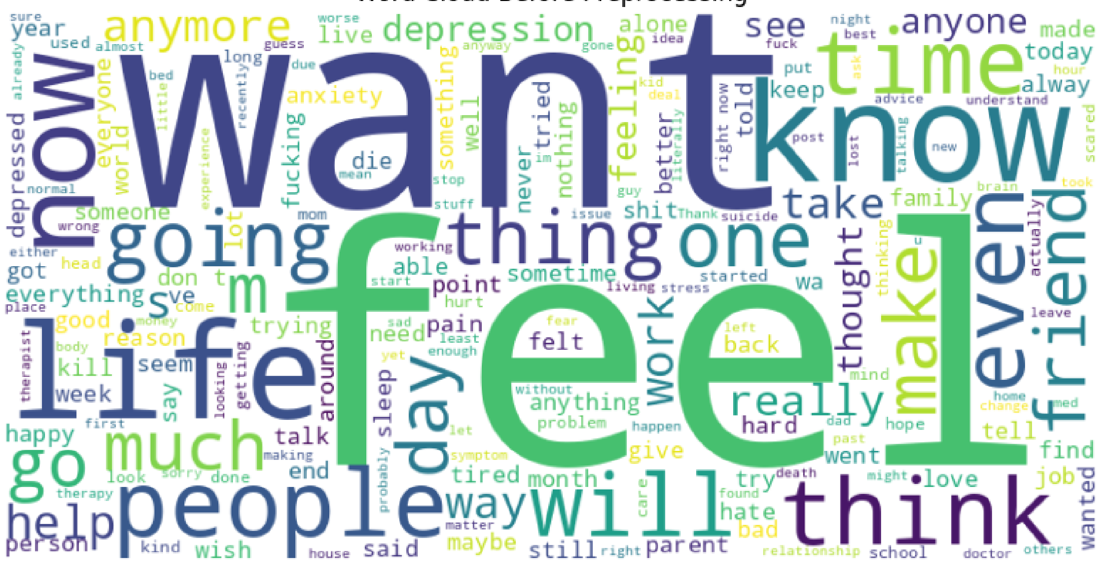
The ratio between different statuses suggests about 70% of the user's input falls under the negative status category.

The target variable in our dataset is unbalanced in favor of depression, suicidal and normal. This imbalance could affect our model's performance, so we'll need to address it later to ensure accurate and fair predictions especially when predicting sentiment analysis for anxiety, bipolar, stress and personality disorder.

[17]:
```
# Word Cloud Before Preprocessing
# Combine all statements into a single string
text = ' '.join(df1['statement'].dropna())

# Create a word cloud
wordcloud = WordCloud(width=800, height=400, background_color='white').
 ↪generate(text)

# Display the word cloud
```

```
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud Before Preprocessing')
plt.show()
```

Word Cloud Before Preprocessing



Here is the word cloud for Statement before data processing, which will be used to compare with the data after processing. The word cloud shows that the most frequently used words are “feel,” “want,” “know,” and “life.”. The observation is reasonable considering verbs and similar expressions that reflect personal thoughts would be the biggest part of user inputs. We can also see words like “depression,” “tired,” and “anxiety” in the word cloud even before data processing, which matches our observation of the status distribution above.

This means that these words may be key indicators for determining sentiment. We will explore more by dividing word clouds into status.

2.0.1 Week 3 Code- EDA and Train-Test Split

We have already done some EDA in the previous code but we will expand on it here. We will also do a train-test split.

[18]:
```
# Taking a look at the long messages
df1[df1['statement_len'] > 1500]
```

[18]:
```
                                                statement  \
7851  So introduction I guess.. my name is Michael a…
8221  do not really know where to start so I am goin…
9504  Hello everyone,I rarely post on Reddit but…
```

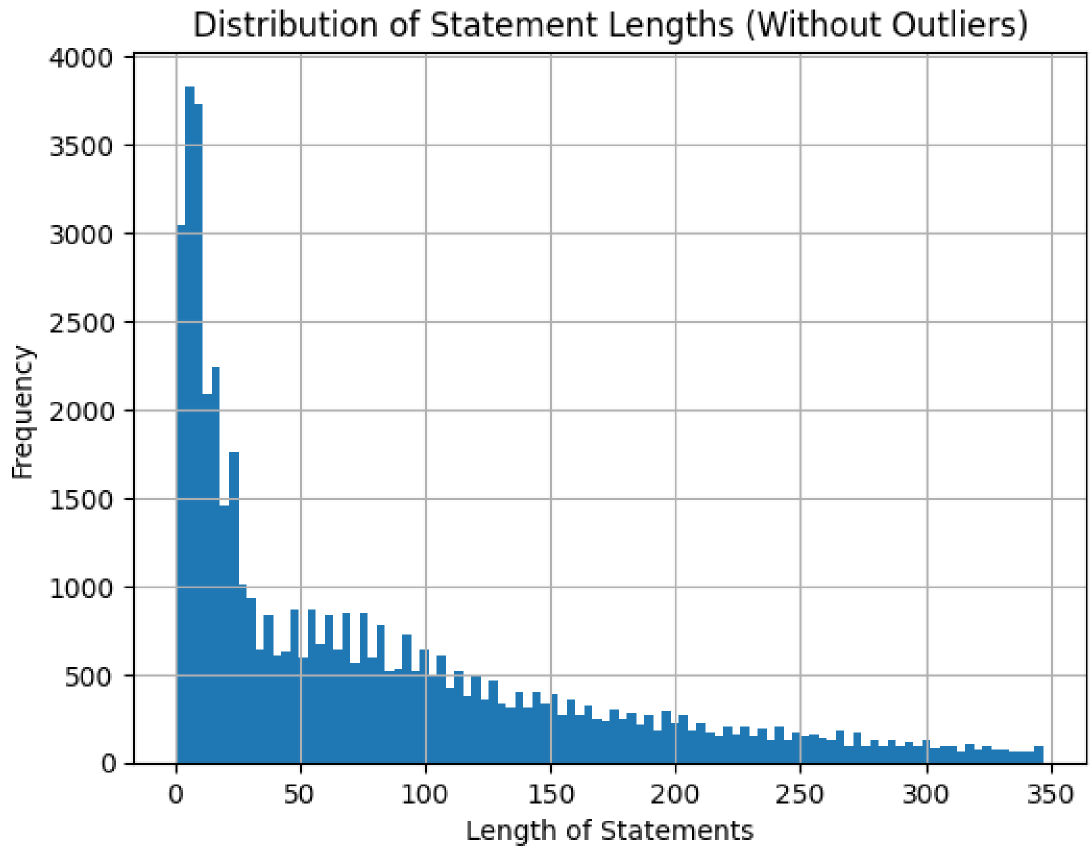
```
10743  ThrowawayIm female, 20 years old. Ever since I…
10834  I have only 1 person I can somewhat open to bu…
11537  The title is not meant to discourage others, b…
11581  I no longer know what else to do but write thi…
11636  And has life gotten better?&amp;#x200B;No. Eve…
11831  Sorry this is long but I doubt anyone will eve…
13188  I am frustrated. that is the constant theme wi…
13293  I cannot TAKE IT ANYMORE. I cannot TAKE IT ANY…
13577  I am very sick and tired, both mentally and ph…
14602  I am 27 years old and have grown deeper into a…
16061  Bear with me please, this may be extremely len…
16498  Hey, this is goodbye note. it is most likely g…
18215  I am someone living in Turkey. My age is proba…
18323  I am going to be turning 30 in a couple weeks…
19321  This happened a little while ago but it still …
19701  If there is a more beneficial sub please lmk s…
20867  Apologies for length. there is a *lot* to expl…
21285  First I am going to present you with a few que…
21396  will i ever be noticed? is my life worth anyth…
21858  I constantly repeat to myself that I have neve…
22243  I do not expect anyone to read this rambly mes…
22351  This is a lengthy post but its a summary of my…
22563  I have been thinking about posting online for …
23195  My entire life has spontaneously combusted ove…
23366  I wish I knew what was wrong with me. So many …
23820  I need support or encouragement. I (29M) reall…
23845  This is a a vent. I (29M) really do not know w…
24276  I guess it all started when I was I guess 11, …
38083  this is my first reddit post also my first tim…
38255  i m at a very weird place in my life right now…
38579  hello thank you for reading my post and any ad…
39579  we ve been seeing a worrying increase in pro s…
39582  for starter i never really had a childhood whe…
39752  it doesn t matter anymore i m going to copy an…
40028  this is a long story i m sorry me and my ex br…
40208  i m at a very weird place in my life right now…
40293  i have come to the conclusion that i am just n…
40371  hello thank you for reading my post and any ad…
46660  DEPRESSION HAS A PURPOSE: HOW TO USE IT RIGHT …
47949  Don't know what to do anymore Back when I was …
48915  I think I'm in the middle of a nervous breakdo…
50253  Manic for 6 months ending up in jail where I h…
51396  Please help me understand what I went through …
52775  I don't know what to do. I don't know how to d…

                  status  statement_len
7851          Depression           2153
```

8221	Depression	1602
9504	Depression	2139
10743	Depression	1537
10834	Suicidal	5248
11537	Depression	2391
11581	Depression	2612
11636	Depression	2415
11831	Depression	2187
13188	Depression	1832
13293	Suicidal	6300
13577	Suicidal	1811
14602	Depression	1809
16061	Depression	1558
16498	Suicidal	1566
18215	Suicidal	2066
18323	Suicidal	1559
19321	Depression	1902
19701	Depression	1661
20867	Depression	1625
21285	Depression	1559
21396	Depression	2510
21858	Depression	2599
22243	Suicidal	2364
22351	Depression	1551
22563	Suicidal	2319
23195	Depression	1818
23366	Depression	1654
23820	Depression	2105
23845	Suicidal	2108
24276	Suicidal	1539
38083	Depression	1559
38255	Depression	1584
38579	Depression	1537
39579	Depression	1747
39582	Depression	1653
39752	Depression	4239
40028	Depression	1726
40208	Depression	1584
40293	Depression	1656
40371	Depression	1537
46660	Bipolar	4727
47949	Depression	1663
48915	Stress	1601
50253	Bipolar	1664
51396	Personality disorder	5419
52775	Anxiety	1586

Many of the longest messages are those with depression and suicidal tendencies. This will help us since if we shorten the output length when preprocessing the data, we are not reducing the number of data points for those that do not have very many data points such as anxiety, bipolar, stress and personality disorder.

Now we want to see a clearer distribution without these outliers so that we can determine the best output length for preprocessing the text.

[19]:
```
# Statement Length Distribution Without Outliers
# Calculate Q1 (25th percentile) and Q3 (75th percentile)
Q1 = df1['statement_len'].quantile(0.25)
Q3 = df1['statement_len'].quantile(0.75)
IQR = Q3 - Q1

# Define the lower and upper bound for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out the outliers
filtered_df = df1[(df1['statement_len'] >= lower_bound) & (df1['statement_len']␣
↪<= upper_bound)]
# Plot the distribution of statement lengths without outliers
filtered_df['statement_len'].hist(bins=100)
plt.title('Distribution of Statement Lengths (Without Outliers)')
plt.xlabel('Length of Statements')
plt.ylabel('Frequency')
plt.show()
```



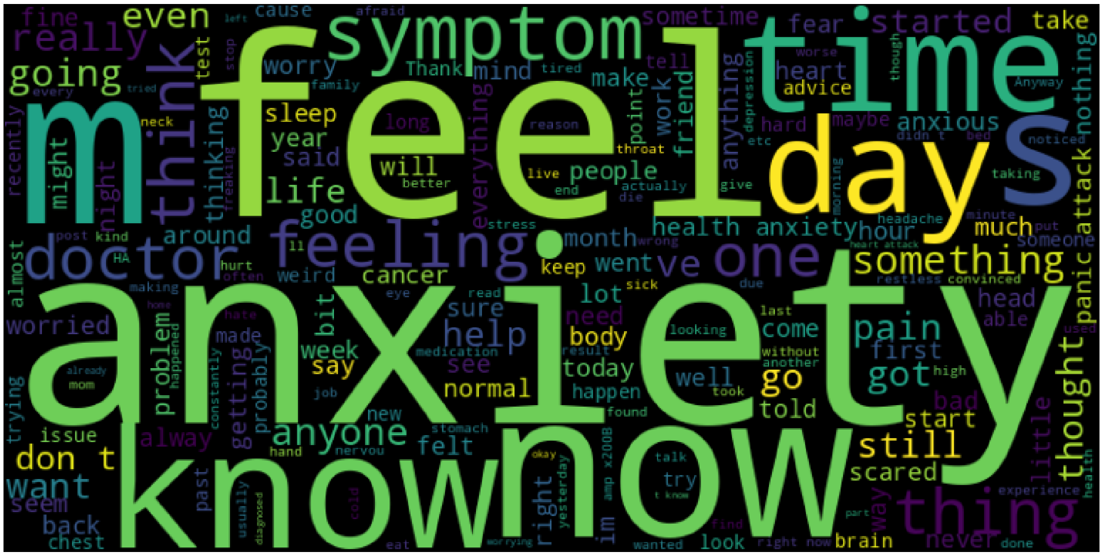
This distrubtion still shows a right-skewed data distribution. We now have a much clearer distribution where approximately 50% of the statements have 0-50 word lengths, especially with a spike at approximately 25 words with approximately 3700 statements. This will help us immensely to determine the best statement length to run our transformer models to save computational resources and time but not decrease model performance.

Now we want to take a closer look at the word clouds for each status since it will give us even more information about the possible word indicators for each status.

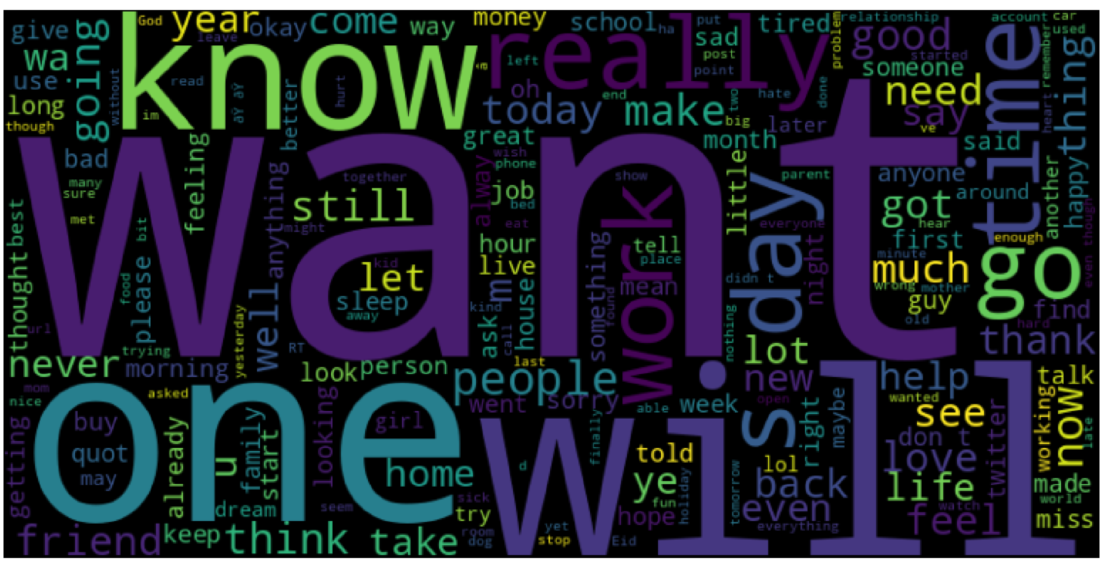
[20]:
```
# Create a function to generate and display a word cloud
def generate_word_cloud(text, title):
    wordcloud = WordCloud(width=800, height=400).generate(text)
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.title(title)
    plt.axis('off')
    plt.show()

# Generate word clouds for each status
statuses = df1['status'].unique()
```

```python
for status in statuses:
    status_text = ' '.join(df1[df1['status'] == status]['statement'])
    generate_word_cloud(status_text, title=f'Word Cloud for {status}')
```

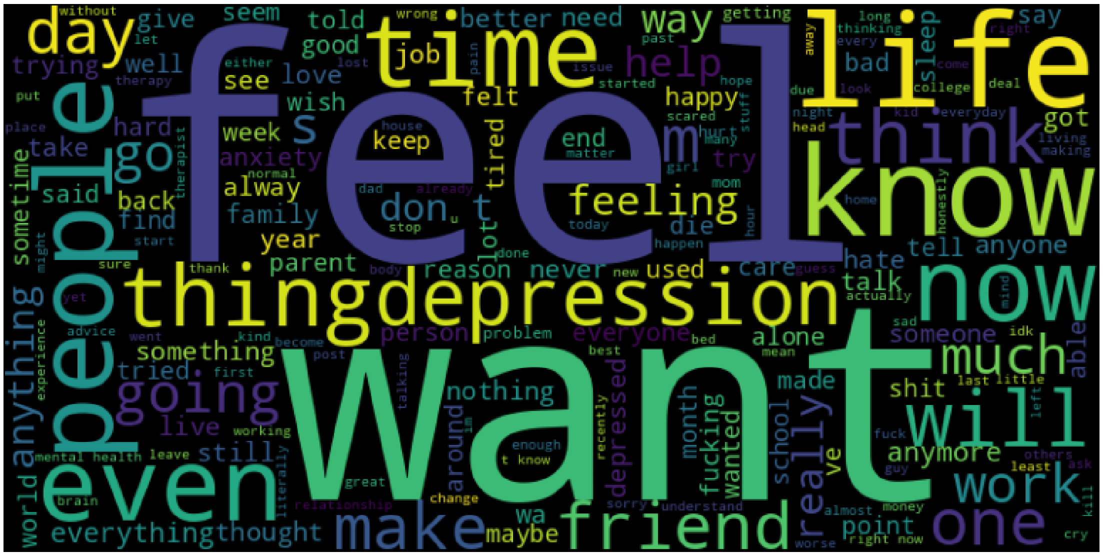
Word Cloud for Anxiety



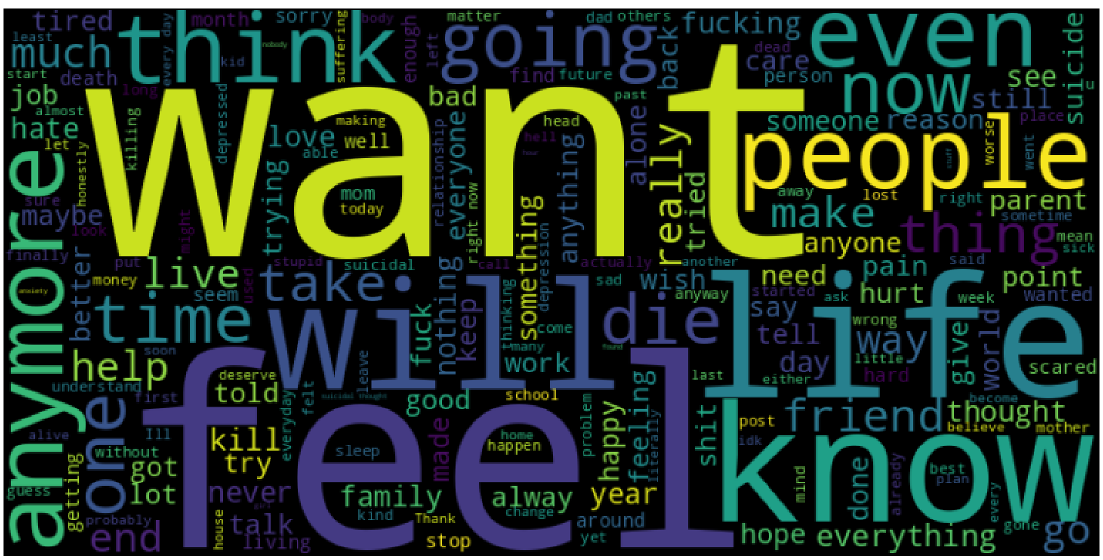
Word Cloud for Normal



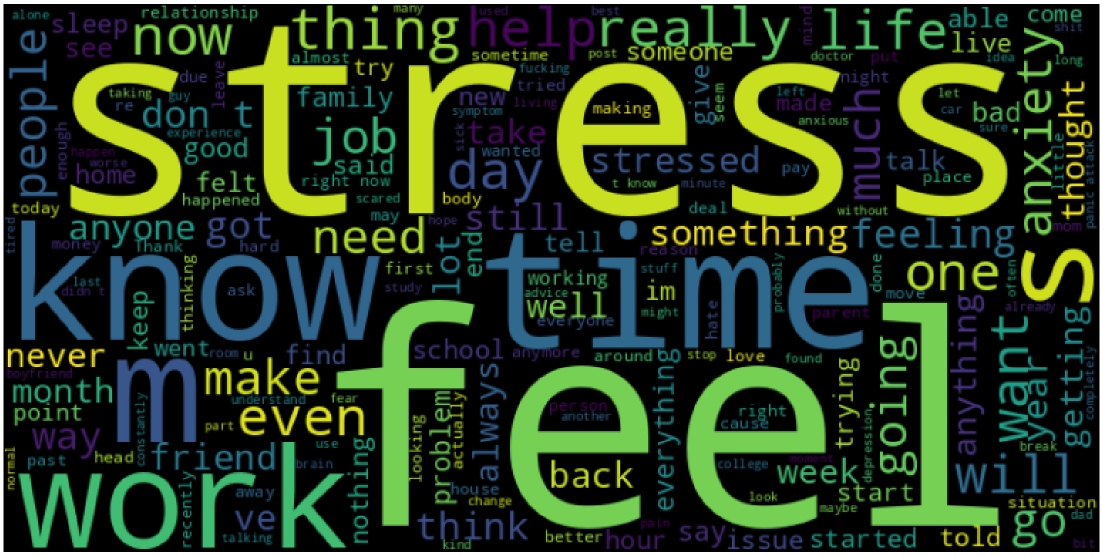
Word Cloud for Depression



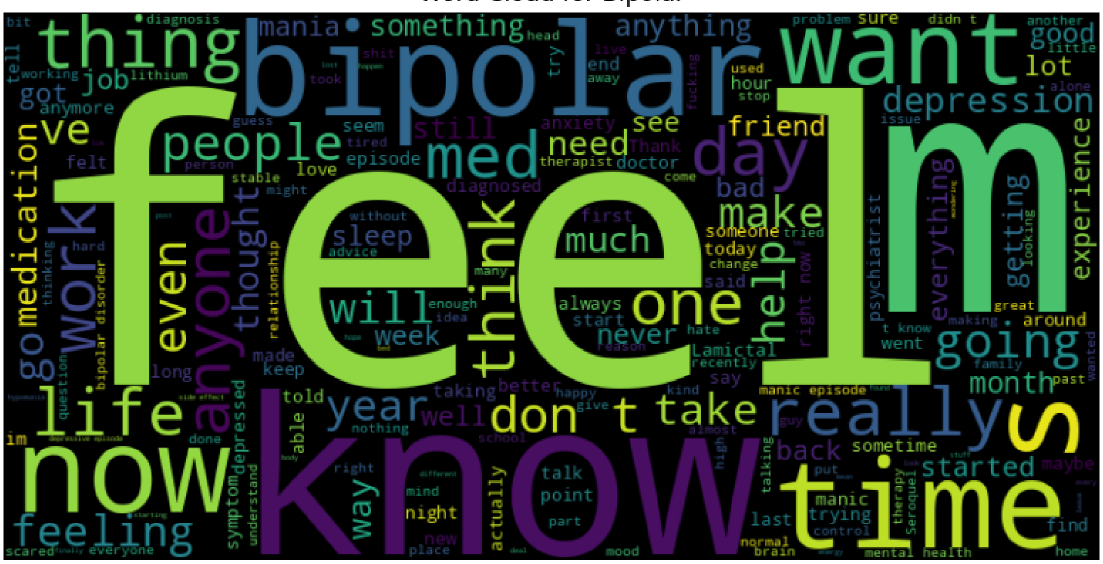
Word Cloud for Suicidal



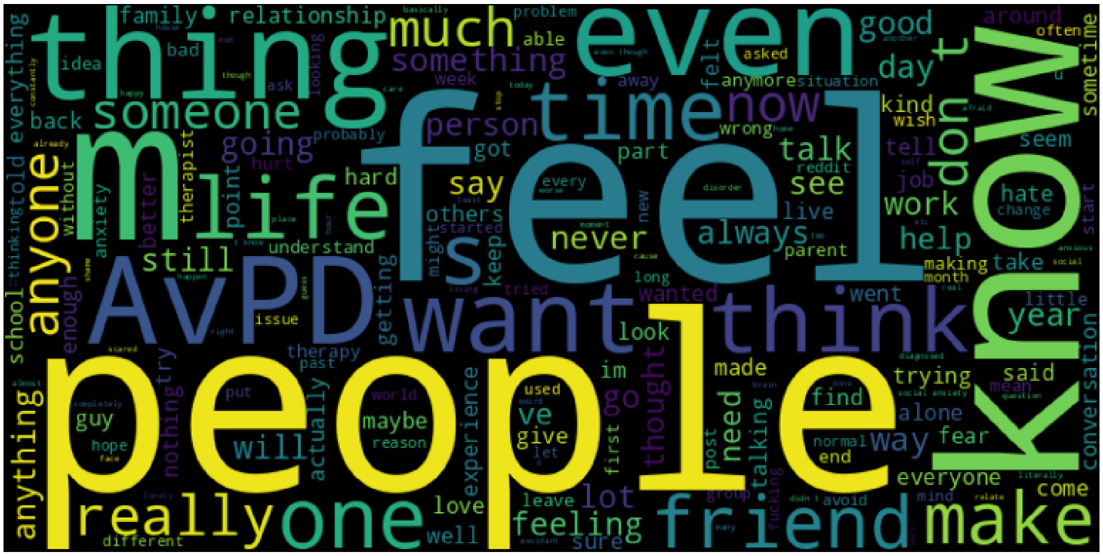
Word Cloud for Stress



Word Cloud for Bipolar



Word Cloud for Personality disorder



We can see for anxiety, stress, and bipolar disorder that “anxiety” and “stress”, “bipolar” are the most popular words. “AVPD”, “people”, and “feel” are the most common words for personality disorder. These words tend to make sense since the words are the literal statuses. Avoidant Personality Disorder (AVPD) is a mental health condition marked by chronic feelings of inadequacy, extreme sensitivity to criticism, and a strong desire to avoid social interactions due to fear of rejection and is a medical condition for a personality disorder.

“Will”, “want”, “know” are the most common words for normal. “Life”, “feel”, “want” are the most common words for suicidal. Many people with suicidal tendencies tend to talk about their lives. The words for “normal” status tend to have positive connotations especially “want” and “will”, especially for a “will” to live.

These words seem to be preliminary indicators to help determine the status based on their statement. We can compare once we calculate sentiment scores.

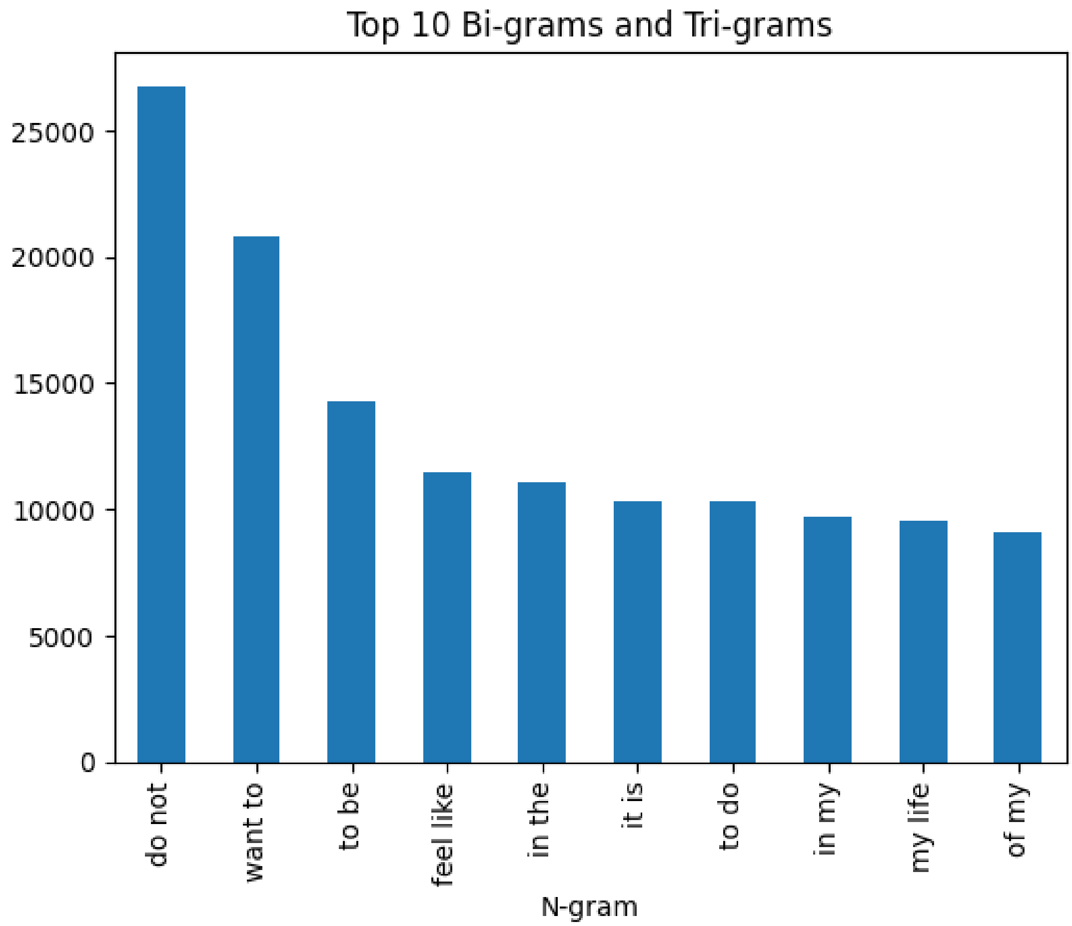
We want to conduct bi-grams and tri-grams analysis for these reasons: Contextual Insights: Bi-grams and tri-grams capture phrases and context that single words (unigrams) might miss. This is particularly important in mental health, where phrases like “feeling down” or “very anxious” provide more insight than individual words.

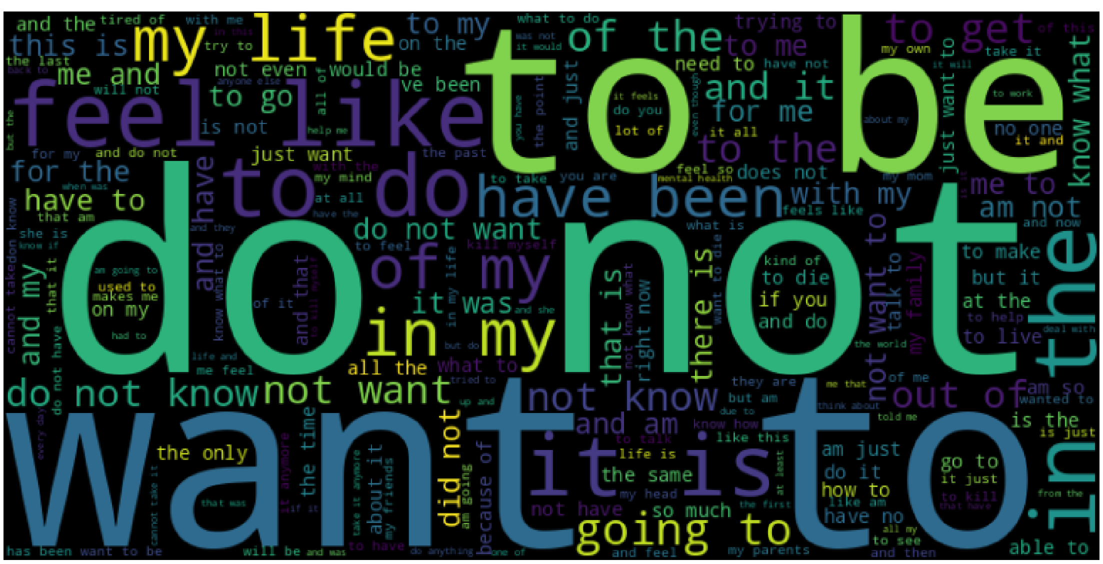
Improved Sentiment Detection: Phrases often convey sentiment more accurately than single words. For example, “not happy” is a negative sentiment that might be missed if only individual words are considered.

Identifying Common Themes: Visualizing bi-grams and tri-grams helps identify common themes and expressions in the dataset. This can reveal patterns in how people express their mental health experiences.

[21]:
```
# import tokenizer
from sklearn.feature_extraction.text import CountVectorizer
```

```python
# Tokenization and N-gram generation
# Create a CountVectorizer object with ngram_range set to (2, 3) to generate␣
↪bi-grams and tri-grams
vectorizer = CountVectorizer(ngram_range=(2, 3))

# Fit and transform the 'statement' column of the DataFrame to generate the␣
↪n-grams
X = vectorizer.fit_transform(df1['statement'])

# Frequency distribution
# Sum the occurrences of each n-gram across all documents
sum_words = X.sum(axis=0)

# Create a list of tuples where each tuple contains an n-gram and its␣
↪corresponding frequency
words_freq = [(word, sum_words[0, idx]) for word, idx in vectorizer.vocabulary_.
↪items()]

# Sort the list of tuples by frequency in descending order
words_freq = sorted(words_freq, key=lambda x: x[1], reverse=True)

# DataFrame for visualization
# Convert the list of tuples into a DataFrame for easier visualization
df_freq = pd.DataFrame(words_freq, columns=['N-gram', 'Frequency'])

# Bar plot
# Plot the top 10 most frequent n-grams as a bar plot
df_freq.head(10).plot(kind='bar', x='N-gram', y='Frequency', legend=False)
plt.title('Top 10 Bi-grams and Tri-grams')
plt.show()

# Word cloud
# Generate a word cloud from the n-gram frequencies
wordcloud = WordCloud(width=800, height=400).
↪generate_from_frequencies(dict(words_freq))

# Display the word cloud
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```


Top 10 Bi-grams and Tri-grams
25000
20000
15000
10000
5000
0
do not
want to
to be
feel like
in the
it is
to do
in my
my life
of my
N-gram


do not
want to
to be
feel like
my life
to do
have been
it is
in the
in my
of my
of the
to get
not want
do not know
not know
going to
do not want
and it
to the
with my
for me
to me
this is
me and
to go
for the
have to
out of
am not
to make
but it
at the
to live
is the
am just
go to
the same
there is
that is
it was
all the
the only
did not
able to

This visualizes the most common bi-grams and tri-grams in our dataset, providing insights into common phrases and patterns, which is particularly useful for sentiment analysis in mental health. It helps identify key expressions and themes that might indicate different emotional states or communication gaps. The top ten are do not, want to, to be, feel like, in the, it is, to do, in my, my life, and of my.

Negative Sentiments: Phrases like “do not” and “feel like” might indicate negative sentiments or expressions of reluctance and emotional states. These bi-grams can help identify statements where individuals are expressing dissatisfaction or discomfort.

Desires and Intentions: Bi-grams such as “want to” and “to do” suggest expressions of desires, intentions, or plans. Analyzing these can reveal what individuals are striving for or what actions they are considering, which can be linked to their mental state.

Self-Reflection: Phrases like “in my,” “my life,” and “of my” indicate self-reflection and personal experiences. These bi-grams can help identify statements where individuals are discussing their personal lives and feelings, which are critical for understanding their mental health.

General Context: Bi-grams like “to be,” “in the,” and “it is” provide general context and can be part of various expressions. While they might not directly indicate sentiment, they help in understanding the structure and flow of the text.

2.0.2 Week 4 Code- Preprocessing the data

df1 is the dataframe that does not have any of the missing values. filtered_df is the dataframe with the outliers removed and no missing values. We will keep the outliers since sentiment analysis often has extreme reviews (e.g., very short or long ones) that can hold strong emotions, valuable for classification especially when using non-transformer models. For transformer models, we may remove outliers by shortening the word length input or use filtered_df to reduce computational time without sacrificing performance. For now, we will use df1 to preprocess the data. Transformer models tend to have different preprocessing techniques anyways.

Warning: After installing `imbalanced-learn`, please **restart the kernel** for the changes to take effect.

You can do this in Jupyter Notebook by clicking:
Kernel → Restart Kernel

[22]:
```
# install packages
!pip install imbalanced-learn
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: imbalanced-learn in /home/jupyter-
geean/.local/lib/python3.12/site-packages (0.13.0)
Requirement already satisfied: numpy<3,>=1.24.3 in
/opt/tljh/user/lib/python3.12/site-packages (from imbalanced-learn) (1.26.4)
Requirement already satisfied: scipy<2,>=1.10.1 in
/opt/tljh/user/lib/python3.12/site-packages (from imbalanced-learn) (1.13.1)
Requirement already satisfied: scikit-learn<2,>=1.3.2 in
/opt/tljh/user/lib/python3.12/site-packages (from imbalanced-learn) (1.6.1)
Requirement already satisfied: sklearn-compat<1,>=0.1 in /home/jupyter-
```

```
geean/.local/lib/python3.12/site-packages (from imbalanced-learn) (0.1.3)
Requirement already satisfied: joblib<2,>=1.1.1 in
/opt/tljh/user/lib/python3.12/site-packages (from imbalanced-learn) (1.4.2)
Requirement already satisfied: threadpoolctl<4,>=2.0.0 in
/opt/tljh/user/lib/python3.12/site-packages (from imbalanced-learn) (3.5.0)
```

[23]:

```
# import packages
import seaborn as sns

import re
import random
from imblearn.over_sampling import RandomOverSampler # used for oversampling
from scipy.sparse import hstack, csr_matrix  # To combine sparse matrices

import nltk # nlp package
nltk.download('punkt') # install punkt tokenizer
from nltk.tokenize import word_tokenize # tokenize the text
from nltk.stem import PorterStemmer # stem the text

from sklearn.feature_extraction.text import TfidfVectorizer # used for tf-idf
from sklearn.model_selection import GridSearchCV # used to find the best␣
 ↪parameters
from sklearn.preprocessing import LabelEncoder # used to encode categorical␣
 ↪variables
```

```
[nltk_data] Downloading package punkt to /home/jupyter-
[nltk_data]     geean/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

[24]:

```
# Remove duplicates based on 'statement'
df_unique = df1.drop_duplicates(subset=['statement'])

df_unique.head()
```

[24]:

	statement	status	statement_len
0	oh my gosh	Anxiety	3
1	trouble sleeping, confused mind, restless hear…	Anxiety	10
2	All wrong, back off dear, forward doubt. Stay …	Anxiety	14
3	I've shifted my focus to something else but I'…	Anxiety	11
4	I'm restless and restless, it's been a month n…	Anxiety	14

2.1 Why Add Sentence & Character Length in NLP?

Enhancing NLP models like **Naïve Bayes** and **XGBoost** with numerical metadata (e.g., sentence length, character count) improves performance by capturing structural insights.

2.1.1 Benefits:

- **Structural Insights**

- **Sentence Length**: Differentiates concise vs. verbose texts (e.g., tweets vs. articles).
- **Character Length**: Indicates complexity, verbosity, or spam tendencies.

- **Better Model Interpretability**
  - Helps tree-based models (e.g., XGBoost) make effective splits.
  - Useful for readability assessment, spam detection, and authorship identification.
- **Performance Boost**
  - Combines well with TF-IDF, embeddings, and n-grams.
  - Provides independent signals, improving classification accuracy.

2.1.2 Key Use Cases:

- **Sentiment Analysis**: Short reviews are often more direct (positive/negative).

[25]:
```python
# Calculate the number of characters and sentences
# install nltk downloader punkt_tab for sentences
nltk.download('punkt_tab')

df_unique['num_of_characters'] = df_unique['statement'].str.len()
df_unique['num_of_sentences'] = df_unique['statement'].apply(lambda x: len(nltk.
    ↪sent_tokenize(x)))

# Generate descriptive statistics
description = df_unique[['num_of_characters', 'num_of_sentences']].describe()

# Display the descriptive statistics
print(description)
```

```
[nltk_data] Downloading package punkt_tab to /home/jupyter-
[nltk_data]     geean/nltk_data…
[nltk_data]   Package punkt_tab is already up-to-date!

       num_of_characters  num_of_sentences
count       51073.000000      51073.000000
mean          575.375051          6.249251
std           847.661079         10.762749
min             2.000000          1.000000
25%            79.000000          1.000000
50%           313.000000          3.000000
75%           745.000000          8.000000
max         32759.000000       1260.000000
```

2.2 Text Preprocessing

2.2.1 Convert to lowercase for uniformity

[26]:
```python
# convert to lowercase
#rename columns
df_unique.rename(columns={'statement': 'original_statement'}, inplace=True)

# create a new cleaned statement column called statement
df_unique['statement']=df_unique['original_statement'].str.lower()
# see the first ten rows
df_unique.head()
```

[26]:
```
                                  original_statement   status  statement_len  \
0                                         oh my gosh  Anxiety              3
1  trouble sleeping, confused mind, restless hear…  Anxiety             10
2  All wrong, back off dear, forward doubt. Stay …  Anxiety             14
3  I've shifted my focus to something else but I'…  Anxiety             11
4  I'm restless and restless, it's been a month n…  Anxiety             14

   num_of_characters  num_of_sentences  \
0                 10                 1
1                 64                 2
2                 78                 2
3                 61                 1
4                 72                 2

                                           statement
0                                         oh my gosh
1  trouble sleeping, confused mind, restless hear…
2  all wrong, back off dear, forward doubt. stay …
3  i've shifted my focus to something else but i'…
4  i'm restless and restless, it's been a month n…
```

2.2.2 Remove URLS and other text, punctuation, and special text

To remove specific patterns such as URLs or other unwanted text (like `[View Poll](https://www.reddit.com/poll/...)`) from a column in a pandas DataFrame, we can use regular expressions with the re module or pandas built-in string methods. This will help us get cleaner relevant text.

[27]:
```python
def remove_patterns(text):
    # Remove URLs
    text = re.sub(r'http[s]?://\S+', '', text)
    # Remove markdown-style links
    text = re.sub(r'\[.*?\]\(.*?\)', '', text)
    # Remove handles (that start with '@')
    text = re.sub(r'@\w+', '', text)
```

```
    # Remove punctuation and other special characters
    text = re.sub(r'[^\w\s]', '', text)
    return text.strip()

# Apply the function to the 'statement' column
df_unique['statement'] = df_unique['statement'].apply(remove_patterns)
# see the first ten rows
df_unique.head()
```

[27]:

```
                                    original_statement   status  statement_len  \
0                                           oh my gosh  Anxiety              3
1  trouble sleeping, confused mind, restless hear…  Anxiety             10
2  All wrong, back off dear, forward doubt. Stay …  Anxiety             14
3  I've shifted my focus to something else but I'…  Anxiety             11
4  I'm restless and restless, it's been a month n…  Anxiety             14

   num_of_characters  num_of_sentences  \
0                 10                 1
1                 64                 2
2                 78                 2
3                 61                 1
4                 72                 2

                                           statement
0                                         oh my gosh
1  trouble sleeping confused mind restless heart …
2  all wrong back off dear forward doubt stay in …
3  ive shifted my focus to something else but im …
4  im restless and restless its been a month now …
```

2.2.3 Tokenization

This is when the statements are split into words/tokens. Tokenization is **essential in NLP** as it breaks text into smaller units (tokens), making it **processable by models** like Naïve Bayes, XGBoost, and Transformers.

2.2.4 Key Benefits

- **Structures Raw Text** → Converts unstructured text into a usable format.
- **Boosts Model Performance** → Enables better text classification, sentiment analysis, etc.
- **Handles Language Variability**
  - **Word-based**: `"I love NLP"` → `["I", "love", "NLP"]`
  - **Subword-based (BPE, WordPiece)**: `"unhappiness"` → `["un", "happiness"]`

– **Character-based**: Useful for languages without spaces (e.g., Chinese).

- **Improves Efficiency** → Reduces complexity for machine learning models.

[28]:
```python
# Apply word_tokenize to each element in the 'statement' column
df_unique['tokens'] = df_unique['statement'].apply(word_tokenize)
# see the first ten rows
df_unique.head()
```

[28]:
```
                          original_statement   status  statement_len  \
0                                 oh my gosh  Anxiety              3
1  trouble sleeping, confused mind, restless hear…  Anxiety             10
2  All wrong, back off dear, forward doubt. Stay …  Anxiety             14
3  I've shifted my focus to something else but I'…  Anxiety             11
4  I'm restless and restless, it's been a month n…  Anxiety             14

   num_of_characters  num_of_sentences  \
0                 10                 1
1                 64                 2
2                 78                 2
3                 61                 1
4                 72                 2

                                      statement  \
0                                    oh my gosh
1  trouble sleeping confused mind restless heart …
2  all wrong back off dear forward doubt stay in …
3  ive shifted my focus to something else but im …
4  im restless and restless its been a month now …

                                         tokens
0                                [oh, my, gosh]
1  [trouble, sleeping, confused, mind, restless, …
2  [all, wrong, back, off, dear, forward, doubt, …
3  [ive, shifted, my, focus, to, something, else,…
4  [im, restless, and, restless, its, been, a, mo…
```

2.3 What is Stemming & Why is it Important in NLP?

Stemming reduces words to their root form by removing prefixes/suffixes, helping **normalize text** and **reduce dimensionality** in NLP.

2.3.1 How It Works

- "running" → "run"
- "happily" → "happi"
- "flies" → "fli"

2.3.2 Why It Matters

- **Reduces Vocabulary Size** → Groups similar words.
- **Boosts Search & NLP Models** → "run" and "running" treated alike.
- **Speeds Up Processing** → Fewer unique tokens.

2.3.3 Limitation

- Can produce **incorrect roots** ("better" → "bet").
- **Lemmatization** (more accurate) uses a dictionary.

2.3.4 Takeaway

Stemming **simplifies text, reduces redundancy, and improves efficiency** in NLP.

[29]:
```python
# Initialize the stemmer
stemmer = PorterStemmer()

# Function to stem tokens and convert them to strings
def stem_tokens(tokens):
    return ' '.join(stemmer.stem(str(token)) for token in tokens)

# Apply the function to the 'tokens' column
df_unique['tokens_stemmed'] = df_unique['tokens'].apply(stem_tokens)

# print the first ten rows
df_unique.head()
```

[29]:
```
                                 original_statement   status  statement_len  \
0                                        oh my gosh  Anxiety              3
1  trouble sleeping, confused mind, restless hear…  Anxiety             10
2  All wrong, back off dear, forward doubt. Stay …  Anxiety             14
3  I've shifted my focus to something else but I'…  Anxiety             11
4  I'm restless and restless, it's been a month n…  Anxiety             14

   num_of_characters  num_of_sentences  \
0                 10                 1
1                 64                 2
2                 78                 2
3                 61                 1
4                 72                 2

                                          statement  \
0                                        oh my gosh
1  trouble sleeping confused mind restless heart …
2  all wrong back off dear forward doubt stay in …
```

```
3  ive shifted my focus to something else but im …
4  im restless and restless its been a month now …

                                              tokens  \
0                                       [oh, my, gosh]
1  [trouble, sleeping, confused, mind, restless, …
2  [all, wrong, back, off, dear, forward, doubt, …
3  [ive, shifted, my, focus, to, something, else,…
4  [im, restless, and, restless, its, been, a, mo…

                                      tokens_stemmed
0                                         oh my gosh
1  troubl sleep confus mind restless heart all ou…
2  all wrong back off dear forward doubt stay in …
3  ive shift my focu to someth els but im still w…
4  im restless and restless it been a month now b…
```

2.4 Not Removing Stop Words

Stop words are **frequent words** that may not carry significant meaning in NLP tasks.

2.4.1 General Stop Words

- **Articles** → *a, an, the*
- **Prepositions** → *in, on, at, by, with*
- **Pronouns** → *I, you, he, she, it, they*
- **Conjunctions** → *and, but, or, so*
- **Auxiliary Verbs** → *is, are, was, were, have, do, does*

For **mental health sentiment analysis**, it's best to **keep stop words** because:

2.4.2 Context Matters

- Words like **"not," "never," "very"** can flip sentiment.
  - *"not okay"* *"okay"*

2.4.3 Emotional Expressions

- Stop words are essential for capturing **feelings and emotions**.
  - *"I feel so lost"* carries more meaning than *"feel lost"*.

We can see from the word clouds that there are not many stop words that are in bold so we will not remove them.

[30]:
```
# Now we create our dataset for train-validation-test adding the numerical␣
↪features
```

[31]:
```
X = df_unique[['tokens_stemmed', 'num_of_characters', 'num_of_sentences']]
y = df_unique['status']
```

2.5 Why Use Label Encoding for Categorical Variables?

Label encoding converts **categorical target variables (Y)** into numerical format for machine learning models.

2.5.1 Why is it Necessary?

- **ML Models Require Numeric Input** → Algorithms like **XGBoost, Naïve Bayes, SVM** can't process text labels.
- **Standardizes Target Variable** → Maps categories to integers (*e.g., "positive" → 2, "neutral" → 1, "negative" → 0*).
- **Compatible with Many Models** → Needed for both **classification** and **regression tasks**.

[32]:
```
# label encode our categorical variables for y
lbl_enc = LabelEncoder()
y = lbl_enc.fit_transform(y.values)
```

2.5.2 Train-Validation-Test Split

Most common splits are 80-20 so we will use this split here. We will also create a validation set that is 10% and the test set is 10%. The final splits will be 80-10-10.

I did these splits, since this is a common split in machine learning and data science but also because with 80% of the data as a training dataset, a large portion ensures that the model has enough data to learn from, which helps in capturing the underlying patterns and relationships in the data. With a Validation Set of 10%, it can tune hyperparameters and make decisions about the model architecture and helps prevent overfitting by providing a checkpoint to evaluate the model's performance on unseen data during the training process. With a 10% Test Set, we can evaluate the model's performance after it has been trained and validated and 10% is a large enough size given that we have approximately 50,000 data points. The 80-10-10 split is a balanced approach that ensures the model has sufficient data for training while also providing enough data for validation and testing to ensure robust performance

[33]:
```
from sklearn.model_selection import train_test_split

# Split the data into training (80%) and temporary (20%) sets
train_x, temp_x, train_y, temp_y = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Split the temporary set into validation (50%) and test (50%) sets
val_x, test_x, val_y, test_y = train_test_split(temp_x, temp_y, test_size=0.5,␣
 ↪random_state=42)

# Print the sizes of the splits
```

```
print(f"Training set size: {len(train_x)}")
print(f"Validation set size: {len(val_x)}")
print(f"Test set size: {len(test_x)}")
```

```
Training set size: 40858
Validation set size: 5107
Test set size: 5108
```

2.5.3 Week 5 Code- Feature Engineering, Data Augmentation and Reducing Dimensionality

We created the number of sentences and the character length as new features from the "statement" column earlier.

Since we only had "statement" column to begin with, we did not need to reduce dimensionality from our original dataset. However, we decided to remove statement_length from the data since these are very similar to number of characters and number of sentences.

Enhancing NLP models like Naïve Bayes and XGBoost with numerical metadata significantly improves performance by capturing valuable structural insights. Sentence length, for example, differentiates concise texts like tweets from verbose articles. Character count can indicate text complexity, verbosity, or even spam tendencies. These structural features not only offer better model interpretability, aiding tree-based models like XGBoost in making effective splits, but also boost overall performance. This metadata combines well with traditional NLP features like TF-IDF, embeddings, and n-grams, providing independent signals that enhance classification accuracy. Applications include readability assessment, spam detection, and even authorship identification, demonstrating the broad utility of incorporating numerical metadata into NLP workflows.

[34]:
```
# look at training set to see the new features that we built earlier such as␣
↪num_of_characters and num_of_sentences
train_x.head()
```

[34]:
```
                                          tokens_stemmed  num_of_characters  \
1488   my children were given a no limit theme whi ar…                131
21192  i feel complet lost with thing too mani overwh…               5371
23638  hi i am an incom senior in highschool and my m…                544
35989  never been to war but i get terribl nightmar t…                101
44721  justagirl 9 that s great about your licens wis…                 78

       num_of_sentences
1488                  1
21192                41
23638                 6
35989                 2
44721                 1
```

2.5.4 Convert Text to Features using tf-idf to reduce dimensionality

Now, we will transform tokens (words) into numerical values that represent the importance of words in a document relative to a collection of documents. This helps highlight unique words in a document while downplaying common ones, making it easier for machine learning models to identify relevant patterns and make better predictions.

TF-IDF plays a crucial role in sentiment analysis by weighting the importance of words within a document relative to the entire collection of documents (corpus). While it doesn't strictly *reduce* dimensionality like PCA by eliminating features (words), it effectively manages complexity by assigning weights that reflect a word's relevance. Term Frequency (TF) measures how often a word appears in a specific document, while Inverse Document Frequency (IDF) quantifies how rare that word is across the corpus. Common words like "the" or "a" appear frequently in almost all documents, resulting in low IDF scores. Conversely, words that appear frequently in some documents but rarely elsewhere have high IDF scores. The TF-IDF score, the product of TF and IDF, reflects the overall importance of a word in a particular document within the larger context of the corpus.

This weighting scheme is key to sentiment analysis because it downplays the influence of common, often uninformative words that contribute little to sentiment. These words, while frequent, are essentially noise. At the same time, TF-IDF highlights the words that are most discriminative of sentiment, those that appear frequently in documents expressing a particular sentiment but rarely elsewhere. By emphasizing these key terms, TF-IDF effectively reduces the impact of less relevant dimensions (words), allowing sentiment analysis models to focus on the most informative features. This leads to improved performance by making the model more robust to the curse of dimensionality, even though the actual number of features isn't reduced. In short, TF-IDF acts as a feature weighting mechanism, prioritizing the signal (sentiment indicators) over the noise (common words).

[53]:
```
# 1. Initialize TF-IDF Vectorizer and fit/transform on the 'tokens_stemmed'␣
↪column
vectorizer = TfidfVectorizer(ngram_range=(1, 2), max_features=50000)
X_train_tfidf = vectorizer.fit_transform(train_x['tokens_stemmed'])
X_val_tfidf = vectorizer.transform(val_x['tokens_stemmed'])
test_x_tfidf = vectorizer.transform(test_x['tokens_stemmed'])

# 3. Numerical Feature Extraction
X_train_num = csr_matrix(train_x[['num_of_characters', 'num_of_sentences']].
↪values)
X_val_num = csr_matrix(val_x[['num_of_characters', 'num_of_sentences']].values)
test_x_num = csr_matrix(test_x[['num_of_characters', 'num_of_sentences']].
↪values)

# 4. Combine Features
X_train_combined = hstack([X_train_tfidf, X_train_num])
X_val_combined = hstack([X_val_tfidf, X_val_num])
test_x_combined = hstack([test_x_tfidf, test_x_num])

# Print number of feature words
print('Number of feature words:', len(vectorizer.get_feature_names_out()))
```

```
Number of feature words: 50000
```

[36]:
```
X_train_combined.shape
```

[36]: (40858, 50002)

2.6 Why Oversampling for an Imbalanced Dataset?

In an **imbalanced dataset**, the model may favor the majority class, leading to **biased predictions**. **Oversampling** helps balance the dataset by increasing minority class samples.

2.6.1 Why is Oversampling Needed?

- **Prevents Majority Class Bias** → Ensures the model learns patterns from both classes.
- **Improves Model Performance** → Leads to better recall, F1-score, and generalization.
- **Enhances Minority Class Representation** → Avoids underestimating rare but important cases.

2.6.2 Why Does Random Over-Sampling Work Best?

Maintains Original Data Distribution → Simply duplicates minority class samples, avoiding synthetic noise (SMOTE).
Preserves Minority Class Variability → Unlike SMOTE, which may create unrealistic synthetic samples.
Avoids Data Loss → Unlike Under-Sampling, which removes majority class samples and risks losing valuable information.

[37]:
```
# Apply Random Over-Sampling on the vectorized data
ros = RandomOverSampler(random_state=101)
X_train_resampled, y_train_resampled = ros.
 ↪fit_resample(X_train_combined,train_y)
```

[38]:
```
# see the new dataset
X_train_resampled.shape
```

[38]: (89215, 50002)

2.6.3 Week 6 Code

2.6.4 Logistic Regression for NLP Sentiment Analysis in Mental Health

Why Logistic Regression is Good for NLP Sentiment Analysis Logistic regression is a popular choice for sentiment analysis in NLP, especially in the context of mental health, due to several reasons:

- **Simplicity and Interpretability**: Logistic regression is easy to implement and interpret. The coefficients can provide insights into the importance of different features (words or phrases) in predicting sentiment.
- **Efficiency**: It is computationally efficient and can handle large datasets, making it suitable for real-time applications.

- **Performance**: Logistic regression often performs well on text classification tasks, providing a strong baseline for more complex models.
- **Regularization**: It supports regularization techniques (L1 and L2) to prevent overfitting, which is crucial when dealing with high-dimensional text data.

Multiclass Classification Using Logistic Regression One-vs-Rest (OvR) Strategy The one-vs-rest (OvR) strategy is used for multi-class classification problems. Here's how it works:

Binary Classifiers: For a classification problem with N classes, the OvR strategy involves training N separate binary classifiers. Each classifier is responsible for distinguishing one class from all the others.

Training: Each binary classifier is trained to predict whether a given instance belongs to its specific class (positive class) or to any of the other classes (negative class). Prediction: When making predictions, each classifier outputs a probability or score indicating how likely an instance belongs to its class. The final prediction is the class with the highest probability or score among all classifiers.

2.7 Model Comparisons: Logistic Regression with Regularization

Here's a comparison of three logistic regression models with varying regularization:

Model 1: Logistic Regression with L1 Regularization

- `solver='liblinear'`: Uses the 'liblinear' solver, suitable for small datasets.
- `penalty='l1'`: Applies L1 regularization (Lasso), which can help with feature selection by shrinking some coefficients to zero.
- `C=10`: Inverse of regularization strength. A higher value of C means less regularization.
- `random_state=101`: Ensures reproducibility by setting a seed for the random number generator.

Model 2: Logistic Regression with L2 Regularization

- `solver='liblinear'`: Uses the 'liblinear' solver, suitable for small datasets.
- `penalty='l2'`: Applies L2 regularization (Ridge), which helps prevent overfitting by shrinking the coefficients but not to zero.
- `C=10`: Inverse of regularization strength. A higher value of C means less regularization.
- `random_state=101`: Ensures reproducibility by setting a seed for the random number generator.

Model 3: Logistic Regression with L1 Regularization and Stronger Regularization

- `solver='liblinear'`: Uses the 'liblinear' solver, suitable for small datasets.
- `penalty='l1'`: Applies L1 regularization (Lasso), which can help with feature selection by shrinking some coefficients to zero.
- `C=5`: Inverse of regularization strength. A lower value of C means stronger regularization compared to Model 1.
- `random_state=101`: Ensures reproducibility by setting a seed for the random number generator.

Summary of Differences:

The key difference between these models is the type and strength of regularization. Models 1 and 3 use L1 regularization, while Model 2 uses L2. Model 3 applies stronger L1 regularization than Model 1 due to the lower value of `C`. The choice of L1 vs. L2 and the optimal value of `C` depends on the specific dataset and the goals of the modeling (e.g., feature selection, preventing overfitting).

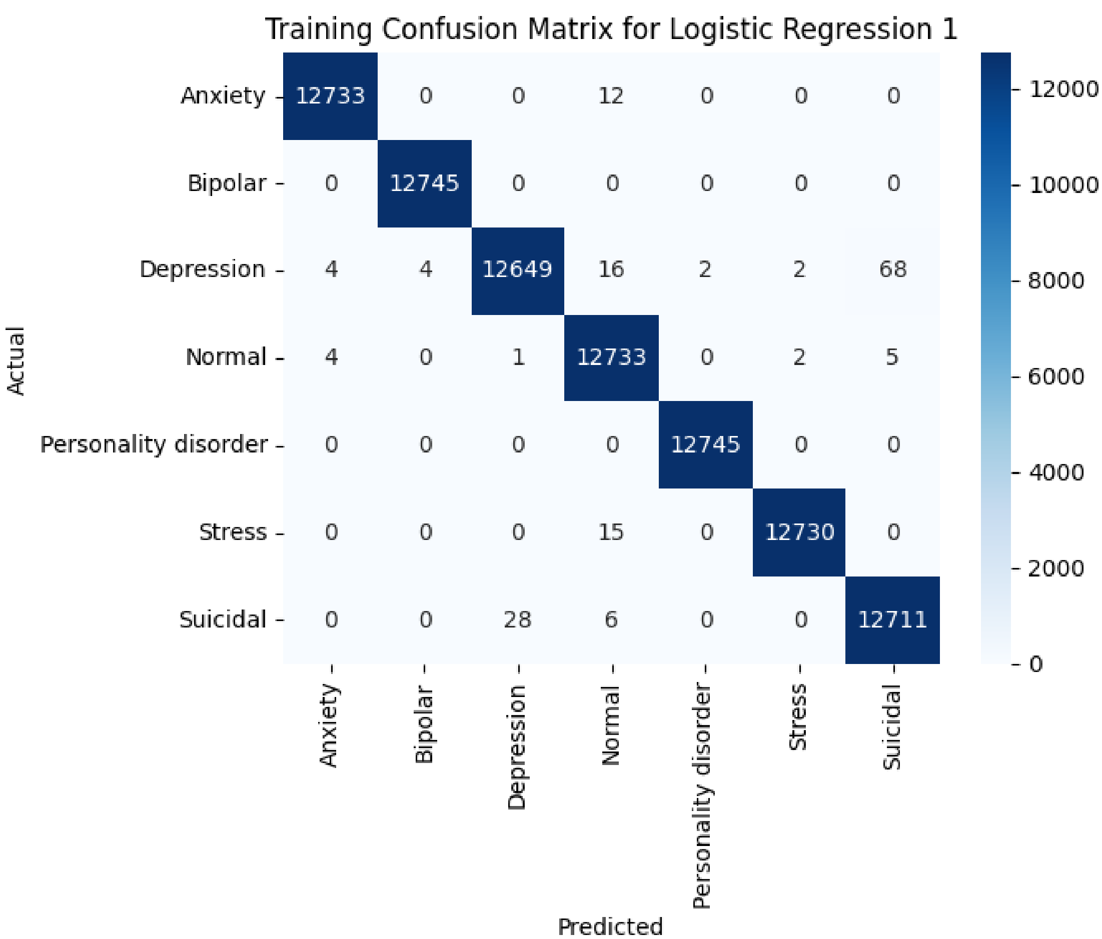
[ ]:
```
# Define a dictionary of classifiers with their specific parameters.
# import packages needed
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix
classifiers = {
    'Logistic Regression 1': LogisticRegression(solver='liblinear',␣
↪penalty='l1', C=10, random_state=101),
    'Logistic Regression 2': LogisticRegression(solver='liblinear',␣
↪penalty='l2', C=10, random_state=101),
    'Logistic Regression 3': LogisticRegression(solver='liblinear',␣
↪penalty='l1', C=5, random_state=101)
}
```

Training the Model and Calculating the Metrics

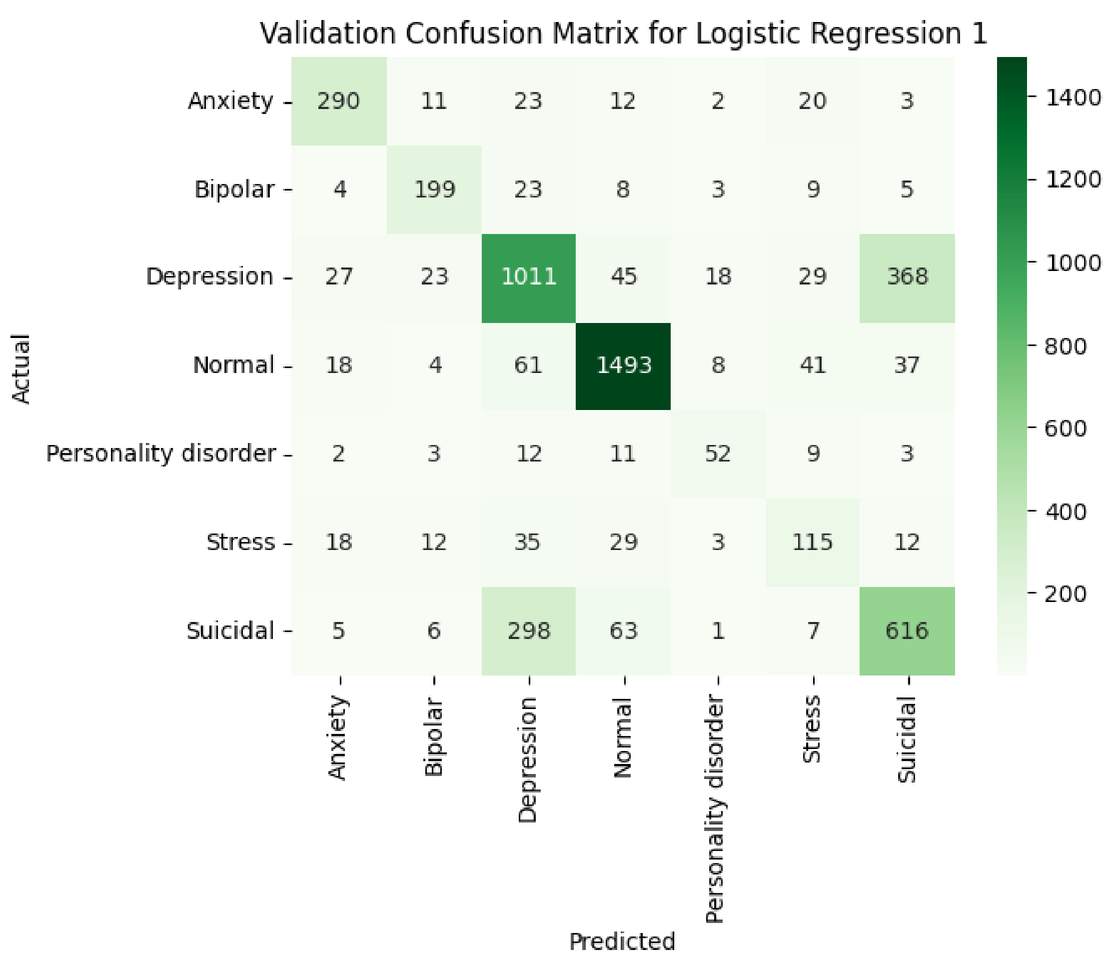
[ ]:
```
# create an empty list to store accuracy scores
accuracy_scores = []
training_accuracy_scores = []

# create for loop to train, predict, and evaluate each model as well as the␣
↪training, validation accuracy, confusion matrix and classification report
for name, clf in classifiers.items():
    clf.fit(X_train_resampled, y_train_resampled)

    # Calculate and store training accuracy
    y_train_pred = clf.predict(X_train_resampled)
    training_accuracy = accuracy_score(y_train_resampled, y_train_pred)
    training_accuracy_scores.append(training_accuracy)

    # Calculate and store validation accuracy
    y_pred = clf.predict(X_val_combined)
    accuracy = accuracy_score(val_y, y_pred)
    accuracy_scores.append(accuracy)

    print("\n")
    print(f"For {name}:")
    print(f"  Training Accuracy: {training_accuracy}")
    print(f"  Validation Accuracy: {accuracy}")

    # Compute and display training confusion matrix and classification report
    print("\nTraining Set Metrics:")
```

```
    conf_matrix_train = confusion_matrix(y_train_resampled, y_train_pred)
    print(classification_report(y_train_resampled, y_train_pred,␣
↪target_names=lbl_enc.classes_))

    sns.heatmap(conf_matrix_train, annot=True, fmt='d', cmap='Blues',␣
↪xticklabels=lbl_enc.classes_, yticklabels=lbl_enc.classes_)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Training Confusion Matrix for {name}')
    plt.show()

    # Compute and display validation confusion matrix and classification report
    print("\nValidation Set Metrics:")
    conf_matrix_val = confusion_matrix(val_y, y_pred)
    print(classification_report(val_y, y_pred, target_names=lbl_enc.classes_))

    sns.heatmap(conf_matrix_val, annot=True, fmt='d', cmap='Greens',␣
↪xticklabels=lbl_enc.classes_, yticklabels=lbl_enc.classes_)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Validation Confusion Matrix for {name}')
    plt.show()

# Print a summary of accuracy scores at the end
print("\nSummary of Accuracy Scores:")
for i, (name, _) in enumerate(classifiers.items()):
    print(f"{name}:")
    print(f"  Training Accuracy: {training_accuracy_scores[i]}")
    print(f"  Validation Accuracy: {accuracy_scores[i]}")

# (Optional) You can further analyze or compare accuracy scores here for␣
↪overfitting For example:
for i, (name, _) in enumerate(classifiers.items()):
    print(f"{name}:")
    print(f"  Training Accuracy: {training_accuracy_scores[i]}")
    print(f"  Validation Accuracy: {accuracy_scores[i]}")
    difference = training_accuracy_scores[i] - accuracy_scores[i]
    print(f"  Difference: {difference}")  # This would display the difference␣
↪between the training and validation accuracy
    if difference > 0.1: # difference is greater than 0.1
        print("    Possible Overfitting")
    elif difference < 0.02: # difference is less than 0.02
        print("    Possible Underfitting")
    else:
        print("    Model is likely a good fit")
```

```
For Logistic Regression 1:
  Training Accuracy: 0.9981056997141736
  Validation Accuracy: 0.7393773252398669

Training Set Metrics:
                      precision    recall  f1-score   support

            Anxiety       1.00      1.00      1.00     12745
            Bipolar       1.00      1.00      1.00     12745
         Depression       1.00      0.99      1.00     12745
             Normal       1.00      1.00      1.00     12745
Personality disorder       1.00      1.00      1.00     12745
             Stress       1.00      1.00      1.00     12745
           Suicidal       0.99      1.00      1.00     12745

           accuracy                           1.00     89215
          macro avg       1.00      1.00      1.00     89215
       weighted avg       1.00      1.00      1.00     89215
```



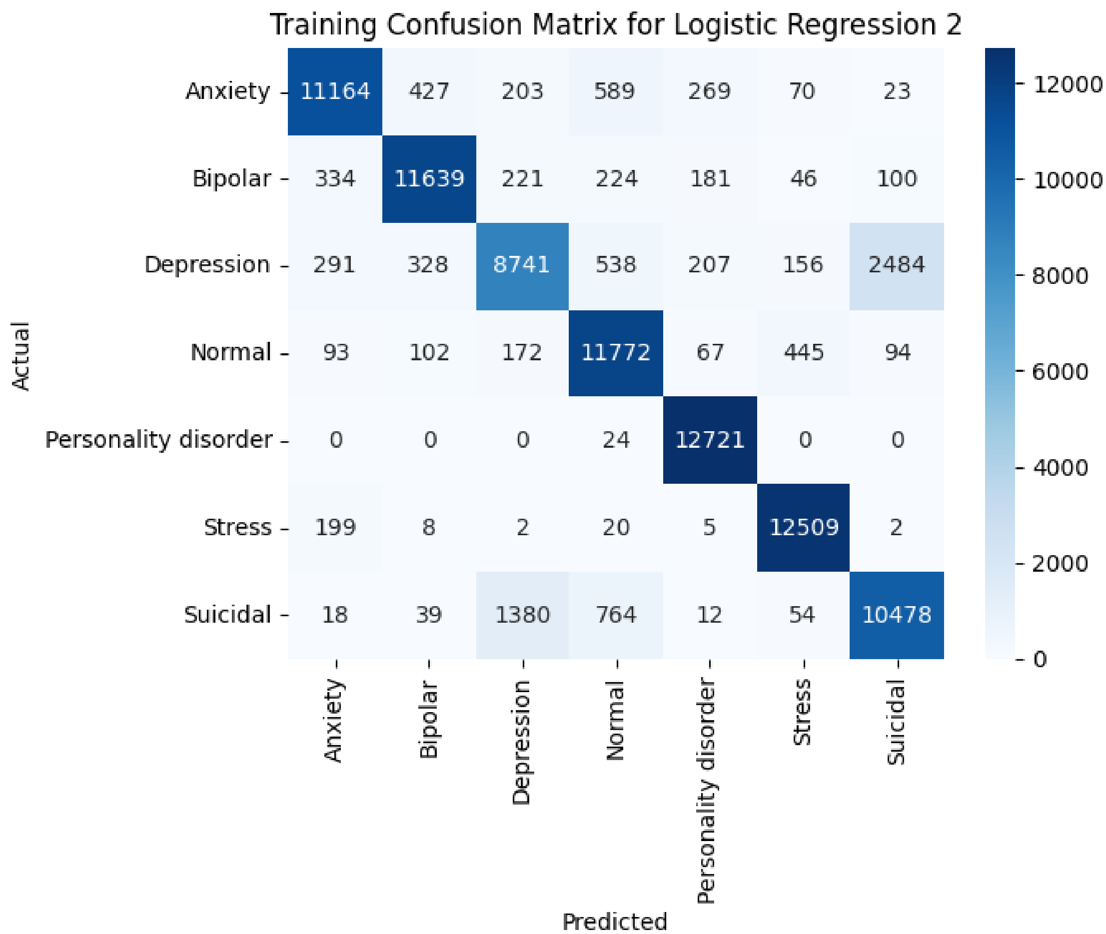
```
Validation Set Metrics:
                      precision    recall  f1-score   support

             Anxiety       0.80      0.80      0.80       361
             Bipolar       0.77      0.79      0.78       251
          Depression       0.69      0.66      0.68      1521
              Normal       0.90      0.90      0.90      1662
Personality disorder       0.60      0.57      0.58        92
              Stress       0.50      0.51      0.51       224
            Suicidal       0.59      0.62      0.60       996

            accuracy                           0.74      5107
           macro avg       0.69      0.69      0.69      5107
        weighted avg       0.74      0.74      0.74      5107
```



```
For Logistic Regression 2:
  Training Accuracy: 0.8857703301014404
  Validation Accuracy: 0.7575876248286665
```

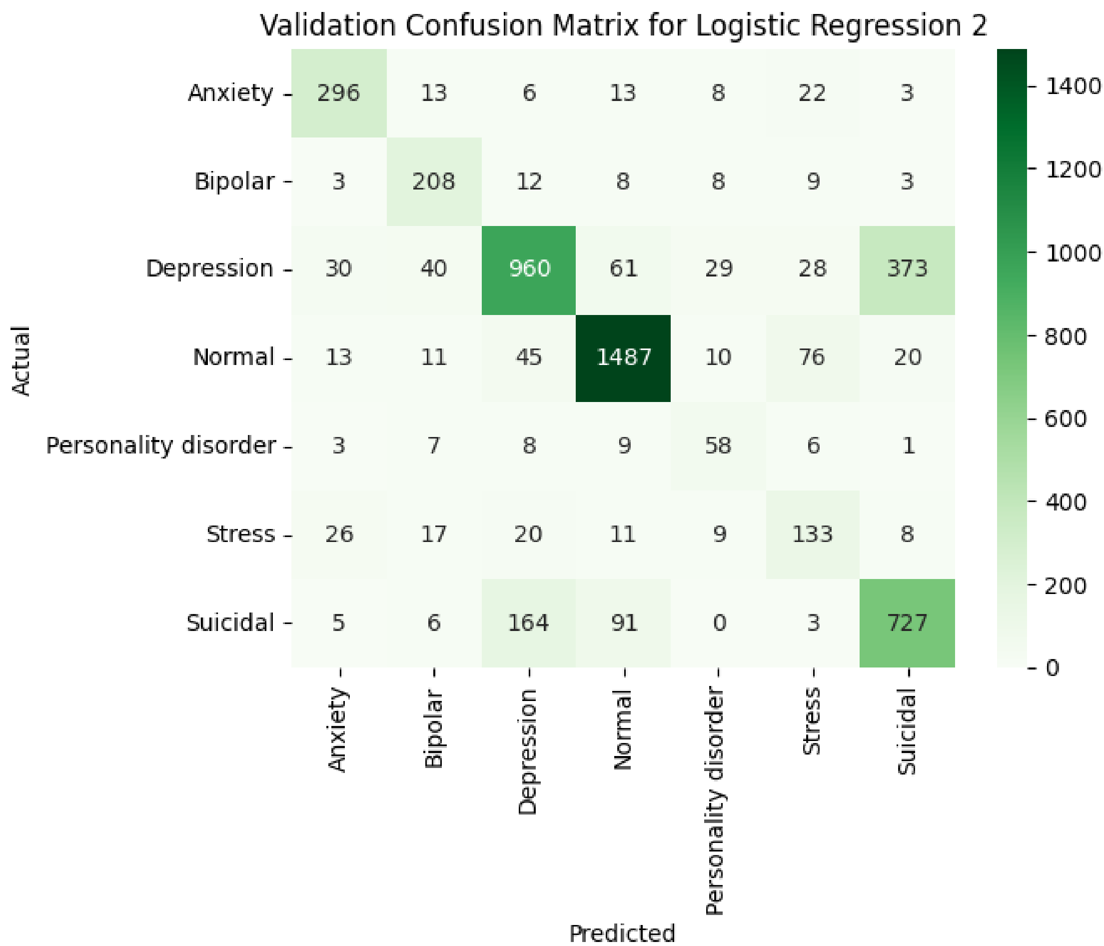
Training Set Metrics:

	precision	recall	f1-score	support
Anxiety	0.92	0.88	0.90	12745
Bipolar	0.93	0.91	0.92	12745
Depression	0.82	0.69	0.75	12745
Normal	0.85	0.92	0.88	12745
Personality disorder	0.94	1.00	0.97	12745
Stress	0.94	0.98	0.96	12745
Suicidal	0.79	0.82	0.81	12745
accuracy			0.89	89215
macro avg	0.88	0.89	0.88	89215
weighted avg	0.88	0.89	0.88	89215



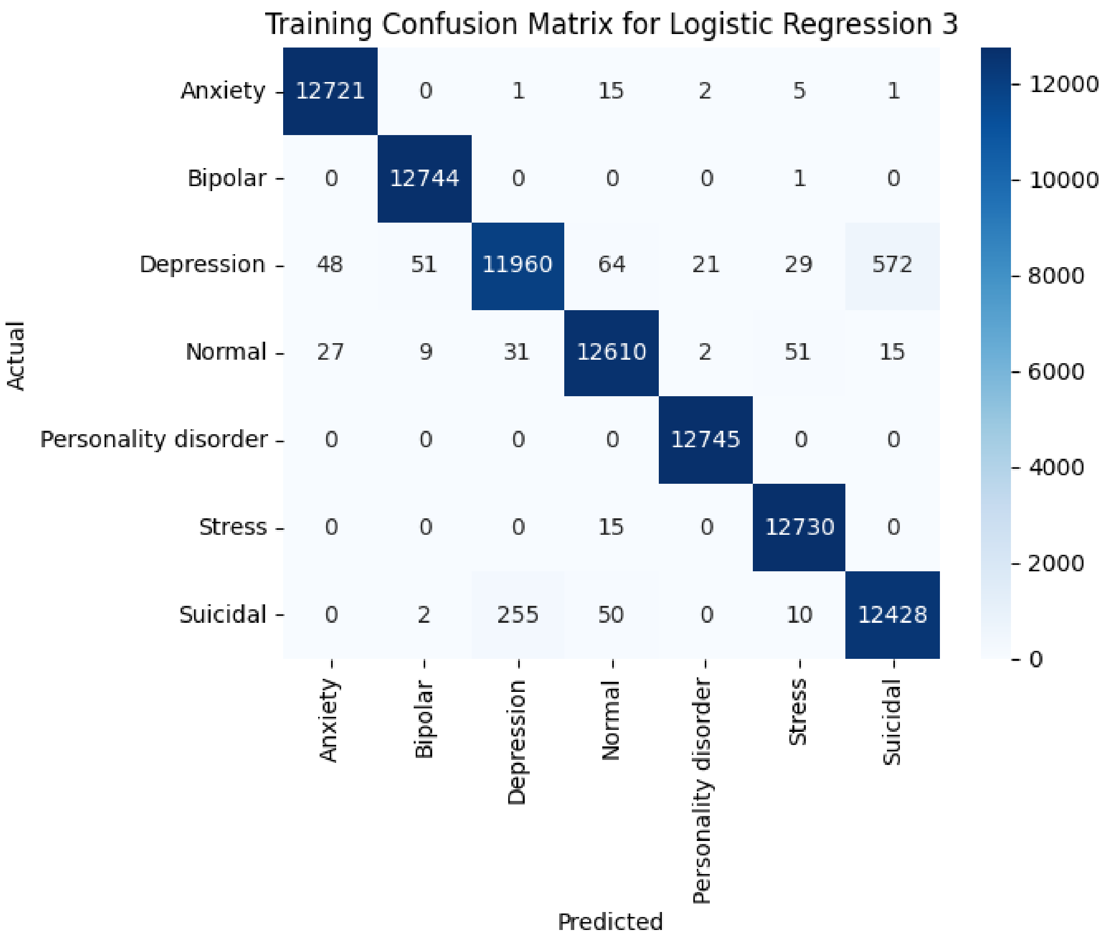
Validation Set Metrics:

	precision	recall	f1-score	support
Anxiety	0.79	0.82	0.80	361
Bipolar	0.69	0.83	0.75	251
Depression	0.79	0.63	0.70	1521
Normal	0.89	0.89	0.89	1662
Personality disorder	0.48	0.63	0.54	92
Stress	0.48	0.59	0.53	224
Suicidal	0.64	0.73	0.68	996
accuracy			0.76	5107
macro avg	0.68	0.73	0.70	5107
weighted avg	0.77	0.76	0.76	5107




```
For Logistic Regression 3:
  Training Accuracy: 0.9856862635207084
  Validation Accuracy: 0.7513217152927355

Training Set Metrics:
                      precision    recall  f1-score   support

             Anxiety       0.99      1.00      1.00     12745
             Bipolar       1.00      1.00      1.00     12745
          Depression       0.98      0.94      0.96     12745
              Normal       0.99      0.99      0.99     12745
Personality disorder       1.00      1.00      1.00     12745
              Stress       0.99      1.00      1.00     12745
            Suicidal       0.95      0.98      0.96     12745

            accuracy                           0.99     89215
```

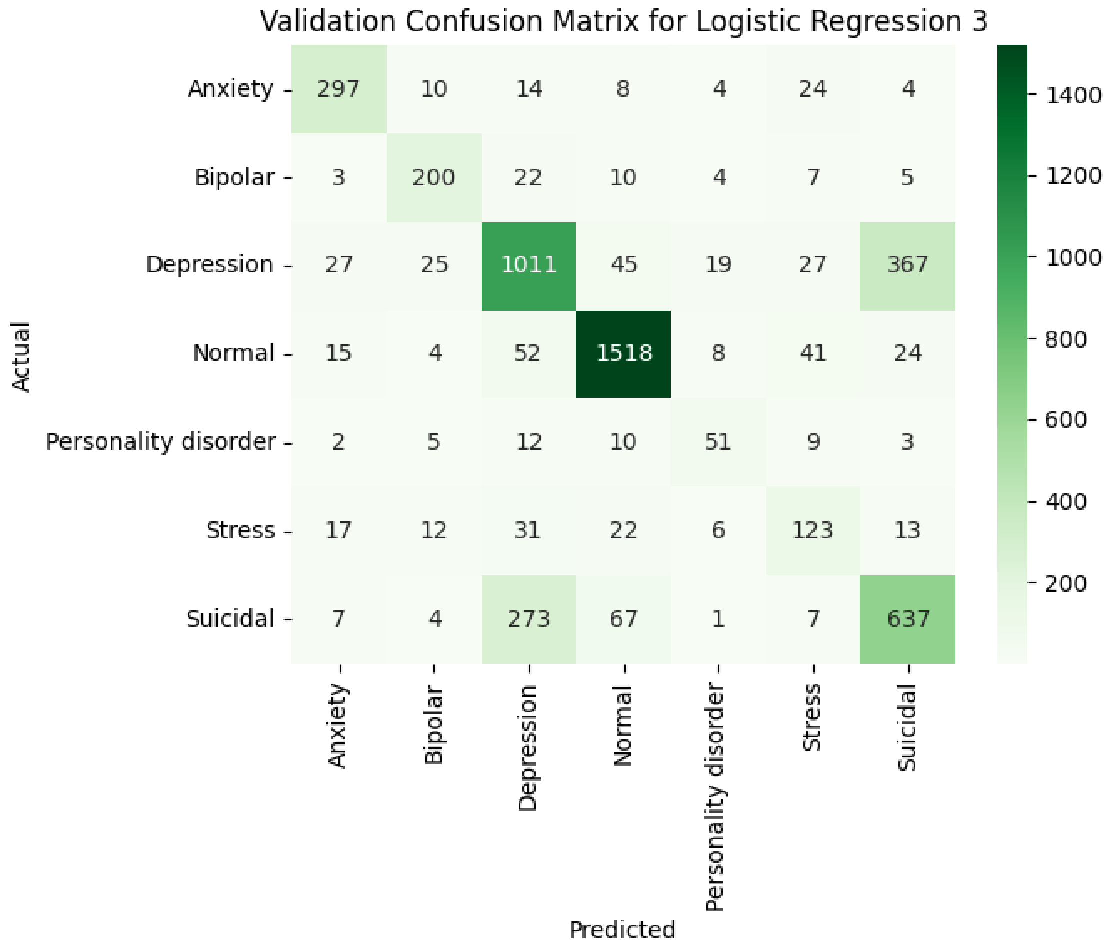
	precision	recall	f1-score	support
macro avg	0.99	0.99	0.99	89215
weighted avg	0.99	0.99	0.99	89215



Validation Set Metrics:

	precision	recall	f1-score	support
Anxiety	0.81	0.82	0.81	361
Bipolar	0.77	0.80	0.78	251
Depression	0.71	0.66	0.69	1521
Normal	0.90	0.91	0.91	1662
Personality disorder	0.55	0.55	0.55	92
Stress	0.52	0.55	0.53	224
Suicidal	0.60	0.64	0.62	996
accuracy			0.75	5107
macro avg	0.69	0.71	0.70	5107

```
weighted avg       0.75      0.75      0.75      5107
```




```
Summary of Accuracy Scores:
Logistic Regression 1:
  Training Accuracy: 0.9981056997141736
  Validation Accuracy: 0.7393773252398669
Logistic Regression 2:
  Training Accuracy: 0.8857703301014404
  Validation Accuracy: 0.7575876248286665
Logistic Regression 3:
  Training Accuracy: 0.9856862635207084
  Validation Accuracy: 0.7513217152927355
Logistic Regression 1:
  Training Accuracy: 0.9981056997141736
  Validation Accuracy: 0.7393773252398669
  Difference: 0.25872837447430674
```

```
    Possible Overfitting
Logistic Regression 2:
  Training Accuracy: 0.8857703301014404
  Validation Accuracy: 0.7575876248286665
  Difference: 0.12818270527277387
    Possible Overfitting
Logistic Regression 3:
  Training Accuracy: 0.9856862635207084
  Validation Accuracy: 0.7513217152927355
  Difference: 0.23436454822797292
    Possible Overfitting
```

2.8 Model Comparison

Here's a comparison of three models based on their training and validation accuracies:

Model	Training Accuracy	Validation Accuracy
Model 1	0.99	0.74
Model 2	0.89	0.757
Model 3	0.99	0.751

Best Model: Model 2

Reason: Model 2 has the highest validation accuracy (0.757) and shows less sign of overfitting compared to Model 1 and Model 3, which have a large gap between training and validation accuracy. A smaller difference between training and validation accuracy suggests better generalization to unseen data. Furthermore, Model 2 had the highest recall scores for more of the classifiers (5 highest recall values).

2.8.1 Week 7 Code

2.9 Bernoulli Naive Bayes for Sentiment Analysis for Mental Health

Bernoulli Naive Bayes is a popular choice for sentiment analysis, especially in domains like mental health, due to several key advantages:

2.9.1 1. Focus on Presence of Words

- **Binary Features**: Bernoulli Naive Bayes works with binary features – whether a word is present or absent in a text. This is well-suited for sentiment analysis where the presence of certain words (e.g., *"sad," "anxious," "hopeless"*) can strongly indicate mental health concerns.

- **Simplicity**: It doesn't consider the frequency of words, which can be less important than their mere presence in identifying sentiment related to mental health.

2.9.2 2. Efficiency with High-Dimensional Data

- **Text Data**: Mental health texts often involve diverse vocabularies and high dimensionality. Bernoulli Naive Bayes handles this efficiently, making it suitable for large datasets or situations requiring quick analysis.

2.9.3 3. Interpretability and Explainability

- **Transparency**: Understanding why a model classifies a text as expressing a particular sentiment is crucial in mental health. Bernoulli Naive Bayes offers some level of interpretability, allowing you to see which words contribute most to the classification. This can be valuable for researchers and clinicians.

2.9.4 4. Sensitivity to Subtle Cues

- **Detecting Subtleties**: In mental health contexts, subtle language cues can be significant. While the "naive" assumption of word independence has limitations, it can sometimes be beneficial. By treating words independently, the model might pick up on subtle signals that would be missed if it focused heavily on word combinations or context.

2.9.5 5. Suitability for Imbalanced Data

- **Prevalence of Challenges**: Mental health datasets often have imbalanced classes, with fewer instances of certain conditions. Bernoulli Naive Bayes can sometimes perform well with imbalanced data, making it relevant for mental health applications where some conditions are less prevalent.

2.10 Second Set of Classifiers for Naive Bayes

Each classifier applies **Bernoulli Naive Bayes** with a different `alpha` value. The `alpha` parameter controls **Laplace smoothing**, which prevents the model from assigning zero probability to words that were not seen in the training data. It plays a crucial role in improving generalization and handling unseen words in text classification.

2.10.1 Effect of Different `alpha` Values:

- **= 0.1**: Low smoothing, making the model more sensitive to rare words.
- **= 1.0**: Default smoothing, providing a balance between sensitivity and generalization.
- **= 10.0**: Higher smoothing, reducing the impact of rare words by distributing probability more evenly.

2.10.2 Importance of `binarize=0.0`

The `binarize` parameter ensures that the model works with **binary word features** (presence or absence of a word) rather than word frequency. This is particularly useful for **sentiment analysis in mental health** because:

1. **Focus on Important Words**: Instead of considering how many times a word appears, the model only cares **whether** it appears, which can improve classification when key words (e.g.,

"anxious," "hopeless," "sad") strongly indicate sentiment.

2. **Reduces Noise**: In mental health text data, some words may appear frequently but are not necessarily relevant (e.g., filler words). Binarization helps eliminate this noise.

3. **Works Well with BernoulliNB**: The Bernoulli Naive Bayes model assumes binary features by design, making `binarize=0.0` a natural choice.

[41]:
```python
# create second set of classifiers for naive bayes
from sklearn.naive_bayes import BernoulliNB
classifiers2 = {
    'Bernoulli NB (alpha=0.1)': BernoulliNB(alpha=0.1, binarize=0.0),
    'Bernoulli NB (alpha=1.0)': BernoulliNB(alpha=1.0, binarize=0.0),
    'Bernoulli NB (alpha=10.0)': BernoulliNB(alpha=10.0, binarize=0.0),
}
```

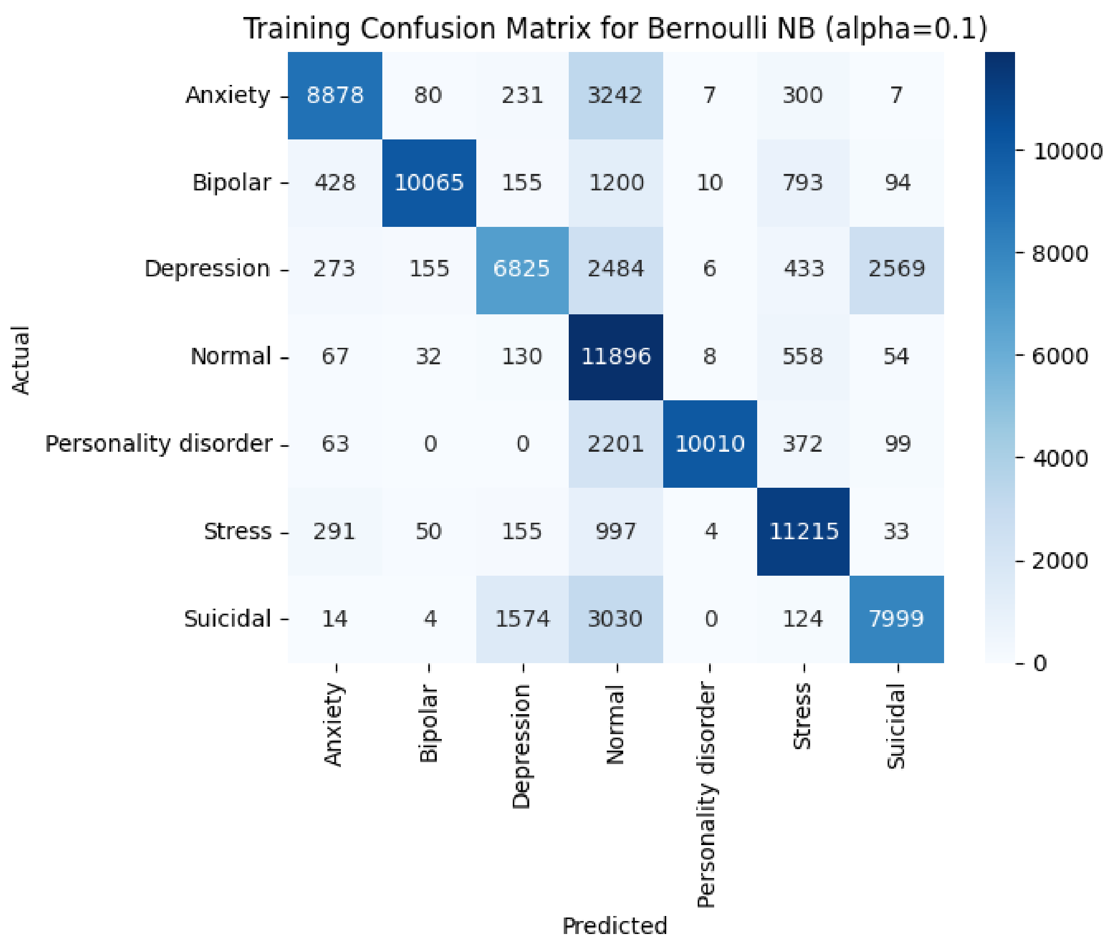
[42]:
```python
# Now run the same training code from week 6 for week 7
# create an empty list to store accuracy scores
accuracy_scores2 = []
training_accuracy_scores2 = []

# create for loop to train, predict, and evaluate each model as well as the␣
↪training, validation accuracy, confusion matrix and classification report
for name, clf in classifiers2.items():
    clf.fit(X_train_resampled, y_train_resampled)

    # Calculate and store training accuracy
    y_train_pred = clf.predict(X_train_resampled)
    training_accuracy = accuracy_score(y_train_resampled, y_train_pred)
    training_accuracy_scores2.append(training_accuracy)

    # Calculate and store validation accuracy
    y_pred = clf.predict(X_val_combined)
    accuracy = accuracy_score(val_y, y_pred)
    accuracy_scores2.append(accuracy)

    print("\n")
    print(f"For {name}:")
    print(f"  Training Accuracy: {training_accuracy}")
    print(f"  Validation Accuracy: {accuracy}")

    # Compute and display training confusion matrix and classification report
    print("\nTraining Set Metrics:")
    conf_matrix_train = confusion_matrix(y_train_resampled, y_train_pred)
    print(classification_report(y_train_resampled, y_train_pred,␣
↪target_names=lbl_enc.classes_))
```

```
    sns.heatmap(conf_matrix_train, annot=True, fmt='d', cmap='Blues',␣
↪xticklabels=lbl_enc.classes_, yticklabels=lbl_enc.classes_)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Training Confusion Matrix for {name}')
    plt.show()

    # Compute and display validation confusion matrix and classification report
    print("\nValidation Set Metrics:")
    conf_matrix_val = confusion_matrix(val_y, y_pred)
    print(classification_report(val_y, y_pred, target_names=lbl_enc.classes_))

    sns.heatmap(conf_matrix_val, annot=True, fmt='d', cmap='Greens',␣
↪xticklabels=lbl_enc.classes_, yticklabels=lbl_enc.classes_)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Validation Confusion Matrix for {name}')
    plt.show()

# Print a summary of accuracy scores at the end
print("\nSummary of Accuracy Scores:")
for i, (name, _) in enumerate(classifiers2.items()):
    print(f"{name}:")
    print(f"  Training Accuracy: {training_accuracy_scores2[i]}")
    print(f"  Validation Accuracy: {accuracy_scores2[i]}")

# (Optional) You can further analyze or compare accuracy scores here for␣
↪overfitting For example:
for i, (name, _) in enumerate(classifiers2.items()):
    print(f"{name}:")
    print(f"  Training Accuracy: {training_accuracy_scores2[i]}")
    print(f"  Validation Accuracy: {accuracy_scores2[i]}")
    difference = training_accuracy_scores2[i] - accuracy_scores2[i]
    print(f"  Difference: {difference}")  # This would display the difference␣
↪between the training and validation accuracy
    if difference > 0.1: # difference is greater than 0.1
        print("    Possible Overfitting")
    elif difference < 0.02: # difference is less than 0.02
        print("    Possible Underfitting")
    else:
        print("    Model is likely a good fit")
```

```
For Bernoulli NB (alpha=0.1):
  Training Accuracy: 0.749739393599731
  Validation Accuracy: 0.6389269629919718
```

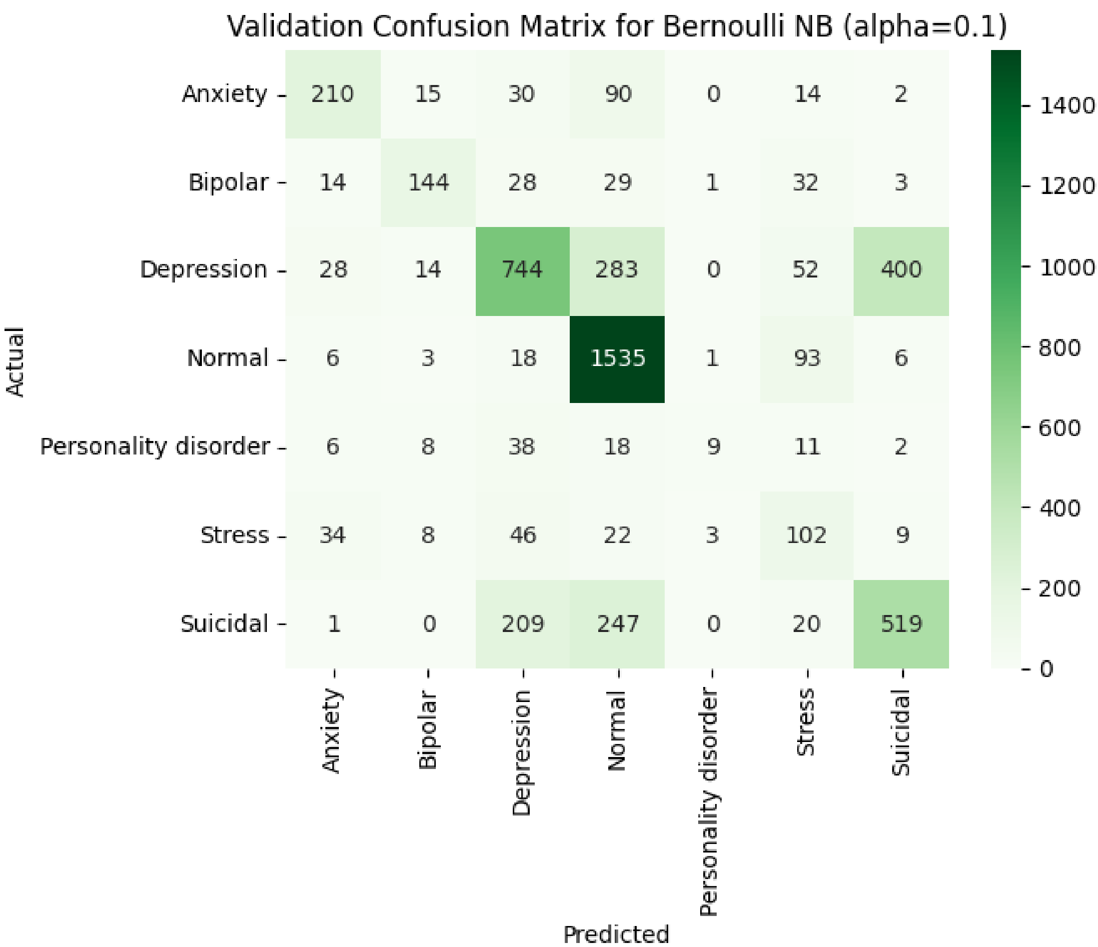
Training Set Metrics:

	precision	recall	f1-score	support
Anxiety	0.89	0.70	0.78	12745
Bipolar	0.97	0.79	0.87	12745
Depression	0.75	0.54	0.63	12745
Normal	0.47	0.93	0.63	12745
Personality disorder	1.00	0.79	0.88	12745
Stress	0.81	0.88	0.85	12745
Suicidal	0.74	0.63	0.68	12745
accuracy			0.75	89215
macro avg	0.80	0.75	0.76	89215
weighted avg	0.80	0.75	0.76	89215

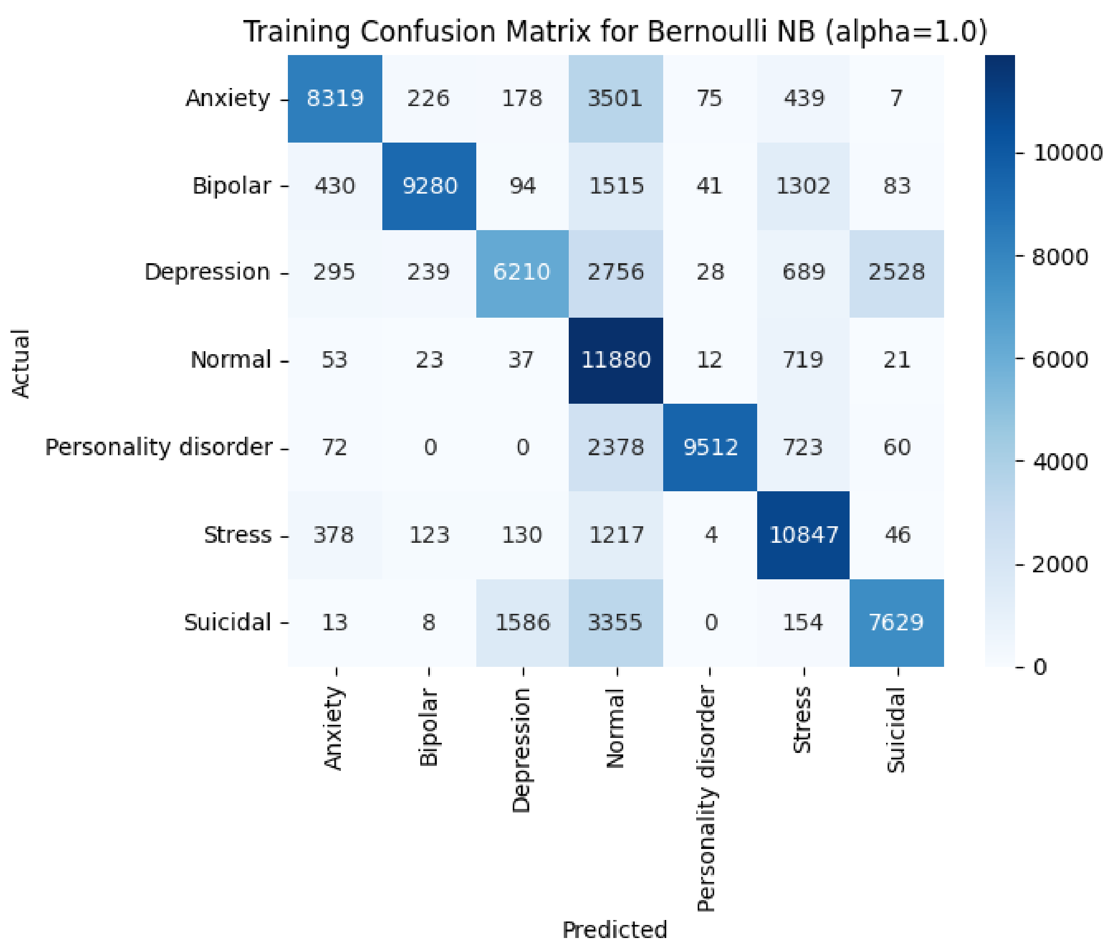


Validation Set Metrics:

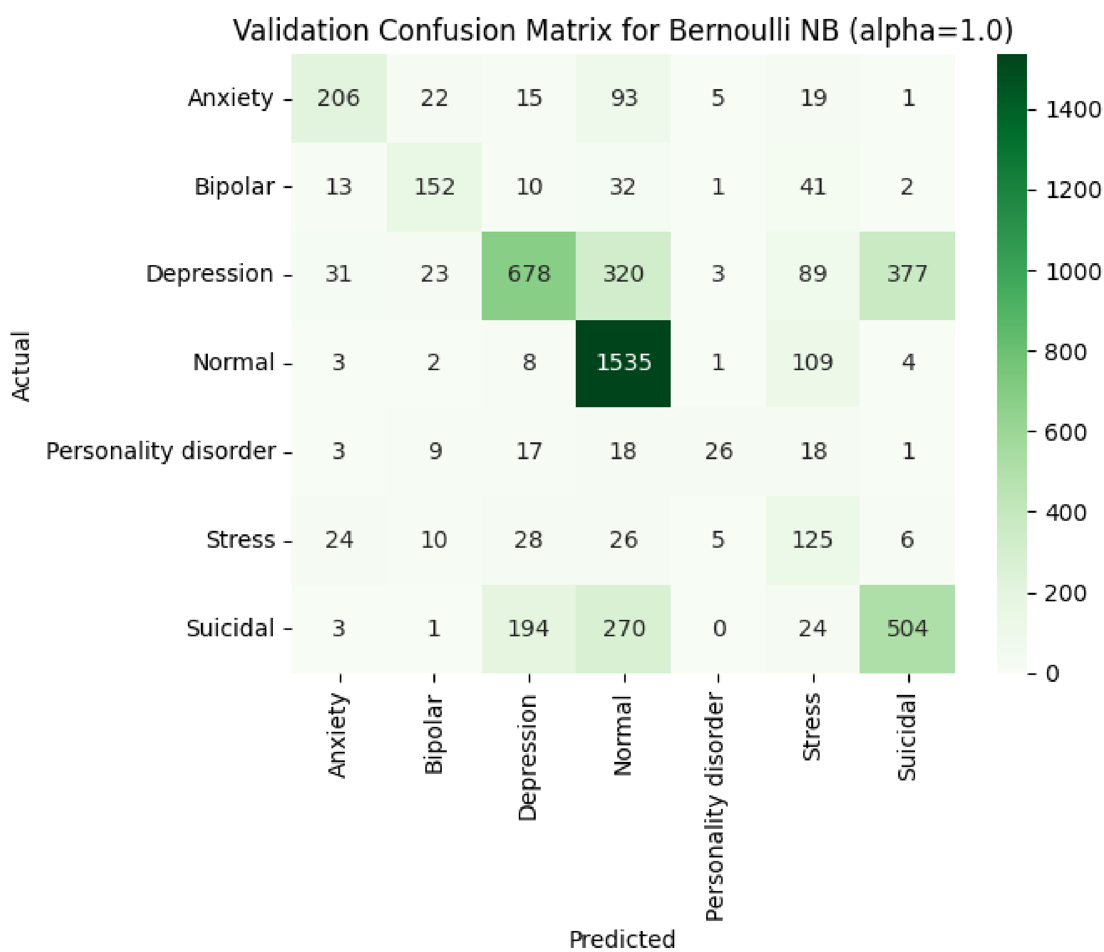
	precision	recall	f1-score	support
Anxiety	0.70	0.58	0.64	361
Bipolar	0.75	0.57	0.65	251
Depression	0.67	0.49	0.56	1521
Normal	0.69	0.92	0.79	1662
Personality disorder	0.64	0.10	0.17	92
Stress	0.31	0.46	0.37	224
Suicidal	0.55	0.52	0.54	996
accuracy			0.64	5107
macro avg	0.62	0.52	0.53	5107
weighted avg	0.64	0.64	0.63	5107



```
For Bernoulli NB (alpha=1.0):
  Training Accuracy: 0.7137476881690299
  Validation Accuracy: 0.6316820050910515

Training Set Metrics:
                      precision    recall  f1-score   support

             Anxiety       0.87      0.65      0.75     12745
             Bipolar       0.94      0.73      0.82     12745
          Depression       0.75      0.49      0.59     12745
              Normal       0.45      0.93      0.60     12745
Personality disorder       0.98      0.75      0.85     12745
              Stress       0.73      0.85      0.79     12745
            Suicidal       0.74      0.60      0.66     12745

            accuracy                           0.71     89215
           macro avg       0.78      0.71      0.72     89215
        weighted avg       0.78      0.71      0.72     89215
```



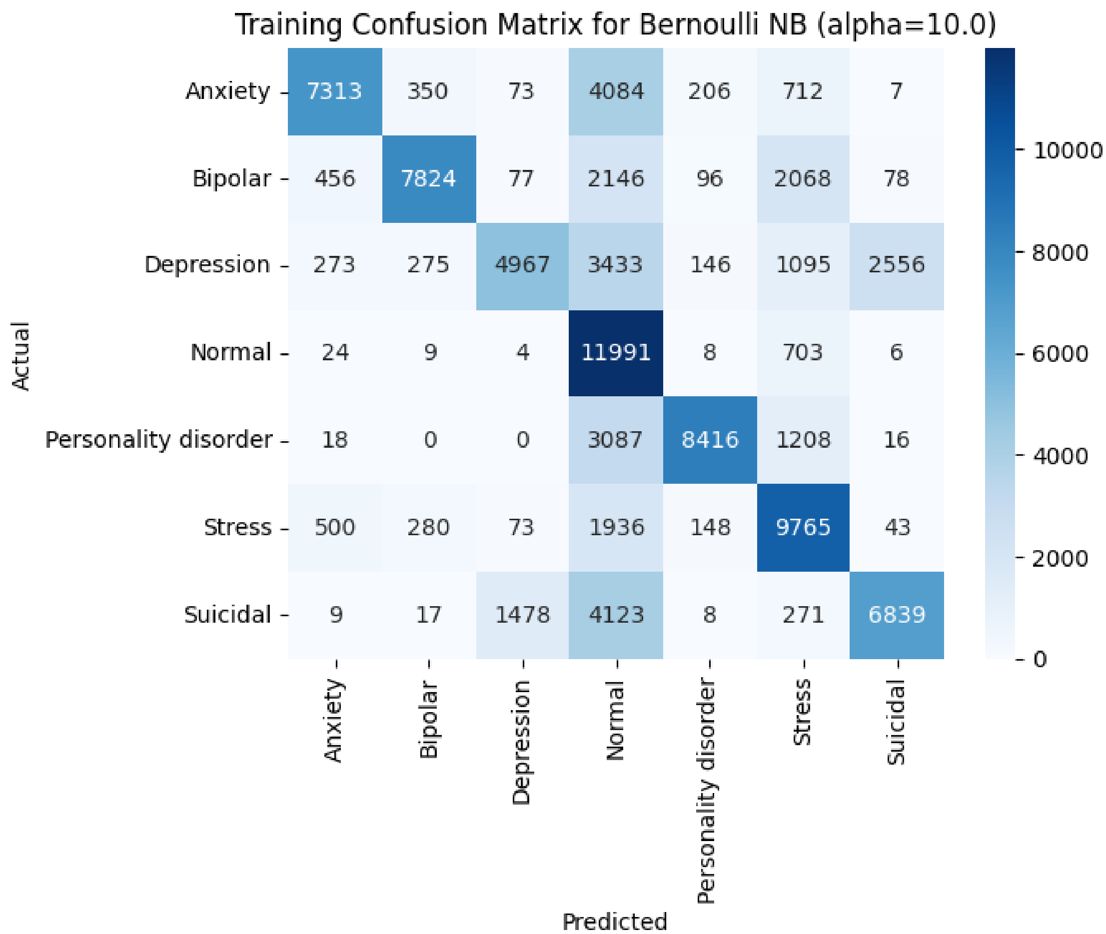
```
Validation Set Metrics:
                       precision    recall  f1-score   support

              Anxiety       0.73      0.57      0.64       361
              Bipolar       0.69      0.61      0.65       251
           Depression       0.71      0.45      0.55      1521
               Normal       0.67      0.92      0.78      1662
 Personality disorder       0.63      0.28      0.39        92
               Stress       0.29      0.56      0.39       224
             Suicidal       0.56      0.51      0.53       996

             accuracy                           0.63      5107
            macro avg       0.61      0.56      0.56      5107
         weighted avg       0.65      0.63      0.62      5107
```



For Bernoulli NB (alpha=10.0):
  Training Accuracy: 0.6401950344672981
  Validation Accuracy: 0.6052476992363423

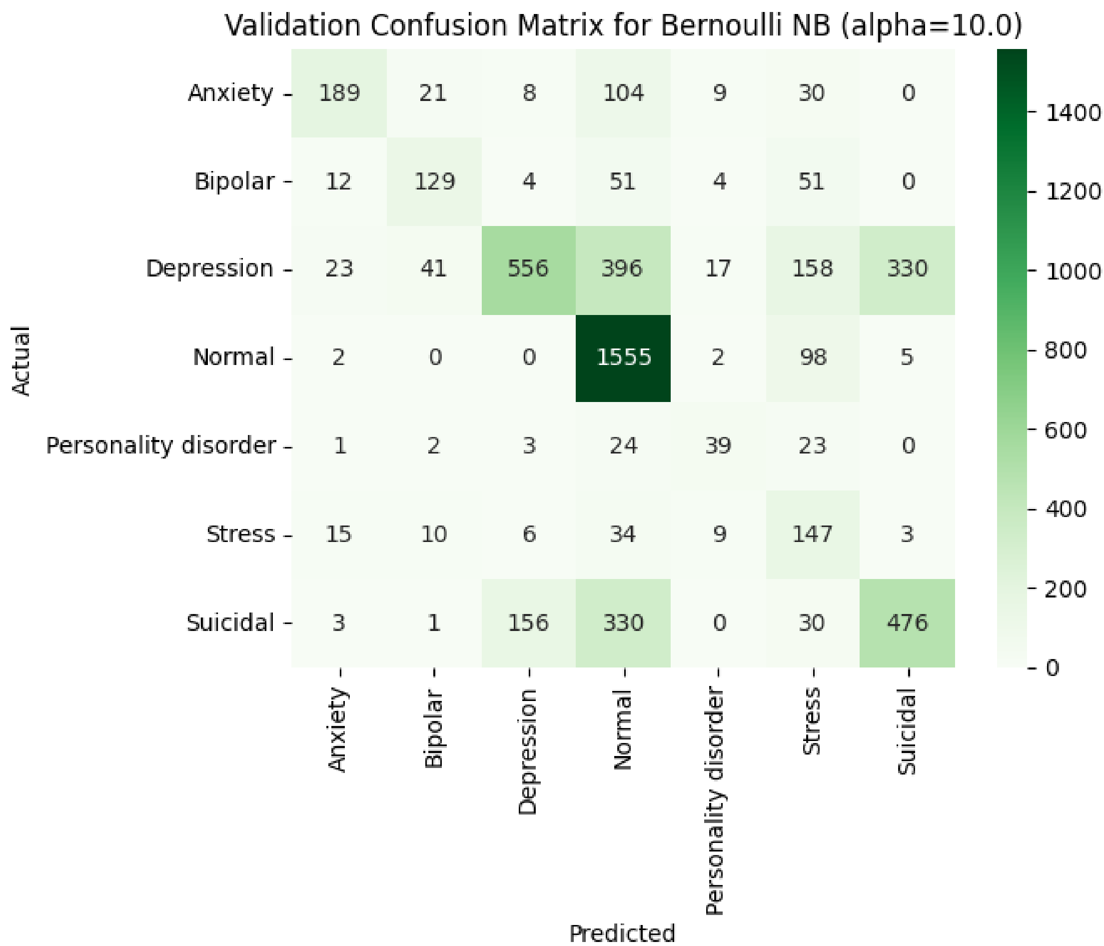
Training Set Metrics:

	precision	recall	f1-score	support
Anxiety	0.85	0.57	0.69	12745
Bipolar	0.89	0.61	0.73	12745
Depression	0.74	0.39	0.51	12745
Normal	0.39	0.94	0.55	12745
Personality disorder	0.93	0.66	0.77	12745
Stress	0.62	0.77	0.68	12745
Suicidal	0.72	0.54	0.61	12745
accuracy			0.64	89215
macro avg	0.73	0.64	0.65	89215
weighted avg	0.73	0.64	0.65	89215



Validation Set Metrics:

	precision	recall	f1-score	support
Anxiety	0.77	0.52	0.62	361
Bipolar	0.63	0.51	0.57	251
Depression	0.76	0.37	0.49	1521
Normal	0.62	0.94	0.75	1662
Personality disorder	0.49	0.42	0.45	92
Stress	0.27	0.66	0.39	224
Suicidal	0.58	0.48	0.53	996
accuracy			0.61	5107
macro avg	0.59	0.56	0.54	5107
weighted avg	0.65	0.61	0.59	5107




```
Summary of Accuracy Scores:
Bernoulli NB (alpha=0.1):
  Training Accuracy: 0.749739393599731
  Validation Accuracy: 0.6389269629919718
Bernoulli NB (alpha=1.0):
  Training Accuracy: 0.7137476881690299
  Validation Accuracy: 0.6316820050910515
Bernoulli NB (alpha=10.0):
  Training Accuracy: 0.6401950344672981
  Validation Accuracy: 0.6052476992363423
Bernoulli NB (alpha=0.1):
  Training Accuracy: 0.749739393599731
  Validation Accuracy: 0.6389269629919718
  Difference: 0.11081243060775914
    Possible Overfitting
Bernoulli NB (alpha=1.0):
  Training Accuracy: 0.7137476881690299
```

```
  Validation Accuracy: 0.6316820050910515
  Difference: 0.08206568307797835
    Model is likely a good fit
Bernoulli NB (alpha=10.0):
  Training Accuracy: 0.6401950344672981
  Validation Accuracy: 0.6052476992363423
  Difference: 0.034947335230955834
    Model is likely a good fit
```

2.11 Model Comparison

Here's a comparison of three models based on their training and validation accuracies:

Model	Training Accuracy	Validation Accuracy
Model 1	0.75	0.6389
Model 2	0.71	0.6317
Model 3	0.99	0.6052

Best Model: Model 2

Reason: Model 1 has the highest validation accuracy (0.6389). Furthermore, Model 2 had the highest recall scores for more of the classifiers (4 highest recall values) and has less overfitting making it the better model with accuracy close to Model 1.

2.12 Week 8 Code

3 Why Use XGBoost for Sentiment Analysis in Mental Health?

3.1 1. Handles Imbalanced Data Well

Mental health sentiment datasets often have class imbalances (e.g., more neutral or non-anxious posts than highly anxious ones). XGBoost provides:
- `scale_pos_weight` to balance classes
- Custom loss functions to focus on minority class performance

3.2 2. Captures Complex Relationships

Mental health sentiment is influenced by nuanced language patterns, and XGBoost can:
- Identify **non-linear** interactions between words and context
- Work well with **engineered features** (e.g., TF-IDF scores, sentiment polarity, linguistic cues)

3.3 3. Works Well with Sparse Data

- Sentiment features (TF-IDF, word embeddings) are typically **high-dimensional and sparse**.
- XGBoost efficiently handles sparse matrices with the `gpu_hist` tree method, making it fast even on large text datasets.

3.4 4. Robust to Noisy Data

- Mental health text data often contains **spelling errors, slang, and abbreviations**.

- XGBoost's regularization (`reg_alpha`, `reg_lambda`) helps prevent overfitting on noisy text features.

3.5 5. Fast and Scalable

- XGBoost is optimized for **parallel computation** and can leverage GPUs for faster training.

- It **scales well** to large datasets, making it ideal for **social media sentiment analysis**.

3.6 6. Feature Importance & Interpretability

- Unlike deep learning models, XGBoost provides **feature importance scores**, helping explain **which words or features impact mental health sentiment predictions**.

- This interpretability is crucial for mental health professionals who need actionable insights.

3.7 7. Effective with Ensemble Methods

- Can be combined with **logistic regression, LSTMs, or BERT-based models** for improved sentiment classification.

- Hybrid approaches like **XGBoost + Embeddings** can yield strong performance on mental health text data.

3.7.1 Conclusion

XGBoost is a **powerful, efficient, and interpretable** model for **sentiment analysis in mental health**, especially when working with **engineered text features** like TF-IDF or word embeddings. It effectively handles imbalanced, sparse, and noisy data while providing insights into key predictive features.

3.8 Explanation of the Three XGBoost Models

Here's a breakdown of the three XGBoost classifiers defined in the `classifiers3` dictionary, highlighting the key differences in their hyperparameters and their intended effects:

1. XGB_Conservative

- **`learning_rate=0.05`**: This is a very low learning rate. It means the model takes small steps towards minimizing the loss function. This makes the learning process slower but can lead to more accurate models, especially if the data is noisy.
- **`max_depth=3`**: This limits the maximum depth of each tree in the ensemble. Shallow trees are less prone to overfitting but might not capture complex relationships in the data.
- **`n_estimators=200`**: This sets the number of boosting rounds (trees) to 200. While this is less than a more agressive model, it is still a substantial number of trees.

- **`subsample=0.6`**: This means that 60% of the training data is randomly sampled for each tree. This helps reduce overfitting and speeds up training.
- **`colsample_bytree=0.6`**: This means that 60% of the columns (features) are randomly sampled for each tree. This also helps reduce overfitting and increases diversity among the trees.
- **`reg_alpha=2` and `reg_lambda=2`**: These are L1 and L2 regularization terms, respectively. High values increase regularization, which adds a penalty for complex models, further reducing overfitting.
- **`tree_method='hist'`**: This uses the histogram-based algorithm for building trees, which is generally faster than the exact greedy algorithm, especially for large datasets.
- **Intended Effect**: This configuration is designed to be conservative, aiming for robustness and reduced overfitting. The low learning rate and strong regularization should produce a model that generalizes well, but it might take longer to train and might not capture very fine-grained patterns.

2. XGB_Faster

- **`learning_rate=0.1`**: This is a moderate learning rate, twice that of the "Conservative" model. It allows the model to learn faster.
- **`max_depth=3`**: Same as the "Conservative" model, limiting tree depth.
- **`n_estimators=200`**: Same number of trees as the "Conservative" model.
- **`subsample=0.7` and `colsample_bytree=0.7`**: These values are slightly higher than the "Conservative" model, meaning that more data and features are used for each tree. This can lead to slightly better performance but also slightly increased risk of overfitting.
- **`reg_alpha=1` and `reg_lambda=1`**: These regularization values are lower than the "Conservative" model, reducing the strength of regularization.
- **`tree_method='hist'`**: Same as the "Conservative" model.
- **Intended Effect**: This model aims for a balance between speed and performance. The increased learning rate and reduced regularization should make it faster than the "Conservative" model while maintaining reasonable accuracy.

3. XGB_Fastest

- **`learning_rate=0.2`**: This is a relatively high learning rate, allowing the model to converge even faster.
- **`max_depth=3`**: Same as the other models.
- **`n_estimators=200`**: Same number of trees as the other models.
- **`subsample=0.7` and `colsample_bytree=0.7`**: Same as the "Faster" model.
- **`reg_alpha=1` and `reg_lambda=1`**: Same as the "Faster" model.
- **`tree_method='hist'`**: Same as the other models.
- **Intended Effect**: This model is designed for speed. The high learning rate should result in the fastest training time among the three models. However, it might be more prone to overfitting and might not achieve the same level of accuracy as the other models, especially if the data is complex or noisy.

Summary of Differences

Parameter	XGB_Conservative	XGB_Faster	XGB_Fastest
`learning_rate`	0.05	0.1	0.2
`subsample`	0.6	0.7	0.7
`colsample_bytree`	0.6	0.7	0.7

Parameter	XGB_Conservative	XGB_Faster	XGB_Fastest
`reg_alpha`	2	1	1
`reg_lambda`	2	1	1

In essence, the models trade off between speed and potential accuracy:

- "Conservative" prioritizes accuracy and robustness, potentially at the cost of training time.
- "Faster" strikes a balance between speed and accuracy.
- "Fastest" prioritizes speed, potentially sacrificing some accuracy.

[43]:
```
# Free up memory
import gc
gc.collect()
```

[43]: 39544

[44]:
```
from xgboost import XGBClassifier

# Define three versions of the XGBClassifier with different hyperparameters
classifiers3 = {
    'XGB_Conservative': XGBClassifier(
        learning_rate=0.05,
        max_depth=3,  # Reduced from 6
        n_estimators=200,  # Reduced from 800
        subsample=0.6,
        colsample_bytree=0.6,
        reg_alpha=2,
        reg_lambda=2,
        random_state=101,
        tree_method='hist',
    ),
    'XGB_Faster': XGBClassifier(
        learning_rate=0.1,
        max_depth=3,
        n_estimators=200,
        subsample=0.7,
        colsample_bytree=0.7,
        reg_alpha=1,
        reg_lambda=1,
        random_state=101,
        tree_method='hist',
    ),
    'XGB_Fastest': XGBClassifier(
        learning_rate=0.2,
        max_depth=3,
        n_estimators=200,
        subsample=0.7,
```

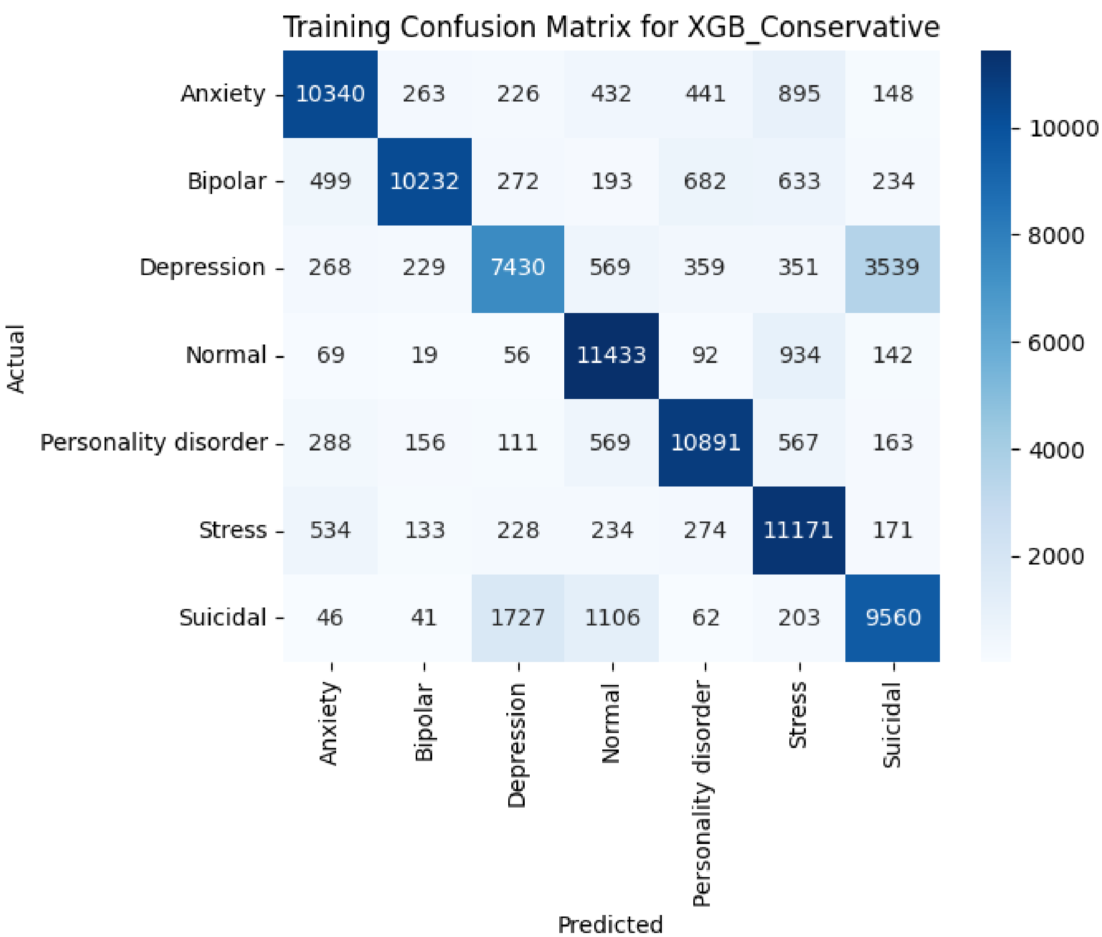
```
        colsample_bytree=0.7,
        reg_alpha=1,
        reg_lambda=1,
        random_state=101,
        tree_method='hist',
    )
}
```

[45]:
```
# Now run the same training code from week 6 for week 8
# create an empty list to store accuracy scores
accuracy_scores3 = []
training_accuracy_scores3 = []

# create for loop to train, predict, and evaluate each model as well as the␣
↪training, validation accuracy, confusion matrix and classification report
for name, clf in classifiers3.items():
    clf.fit(X_train_resampled, y_train_resampled)

    # Calculate and store training accuracy
    y_train_pred = clf.predict(X_train_resampled)
    training_accuracy = accuracy_score(y_train_resampled, y_train_pred)
    training_accuracy_scores3.append(training_accuracy)

    # Calculate and store validation accuracy
    y_pred = clf.predict(X_val_combined)
    accuracy = accuracy_score(val_y, y_pred)
    accuracy_scores3.append(accuracy)

    print("\n")
    print(f"For {name}:")
    print(f"  Training Accuracy: {training_accuracy}")
    print(f"  Validation Accuracy: {accuracy}")

    # Compute and display training confusion matrix and classification report
    print("\nTraining Set Metrics:")
    conf_matrix_train = confusion_matrix(y_train_resampled, y_train_pred)
    print(classification_report(y_train_resampled, y_train_pred,␣
↪target_names=lbl_enc.classes_))

    sns.heatmap(conf_matrix_train, annot=True, fmt='d', cmap='Blues',␣
↪xticklabels=lbl_enc.classes_, yticklabels=lbl_enc.classes_)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Training Confusion Matrix for {name}')
    plt.show()

    # Compute and display validation confusion matrix and classification report
```

```
    print("\nValidation Set Metrics:")
    conf_matrix_val = confusion_matrix(val_y, y_pred)
    print(classification_report(val_y, y_pred, target_names=lbl_enc.classes_))

    sns.heatmap(conf_matrix_val, annot=True, fmt='d', cmap='Greens',␣
 ↪xticklabels=lbl_enc.classes_, yticklabels=lbl_enc.classes_)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'Validation Confusion Matrix for {name}')
    plt.show()

# Print a summary of accuracy scores at the end
print("\nSummary of Accuracy Scores:")
for i, (name, _) in enumerate(classifiers3.items()):
    print(f"{name}:")
    print(f"  Training Accuracy: {training_accuracy_scores3[i]}")
    print(f"  Validation Accuracy: {accuracy_scores3[i]}")

# (Optional) You can further analyze or compare accuracy scores here for␣
 ↪overfitting For example:
for i, (name, _) in enumerate(classifiers3.items()):
    print(f"{name}:")
    print(f"  Training Accuracy: {training_accuracy_scores3[i]}")
    print(f"  Validation Accuracy: {accuracy_scores3[i]}")
    difference = training_accuracy_scores3[i] - accuracy_scores3[i]
    print(f"  Difference: {difference}")  # This would display the difference␣
 ↪between the training and validation accuracy
    if difference > 0.1: # difference is greater than 0.1
        print("    Possible Overfitting")
    elif difference < 0.02: # difference is less than 0.02
        print("    Possible Underfitting")
    else:
        print("    Model is likely a good fit")
```

```
For XGB_Conservative:
  Training Accuracy: 0.7964692036092585
  Validation Accuracy: 0.7409438026238496

Training Set Metrics:
              precision    recall  f1-score   support

     Anxiety       0.86      0.81      0.83     12745
     Bipolar       0.92      0.80      0.86     12745
  Depression       0.74      0.58      0.65     12745
      Normal       0.79      0.90      0.84     12745
```

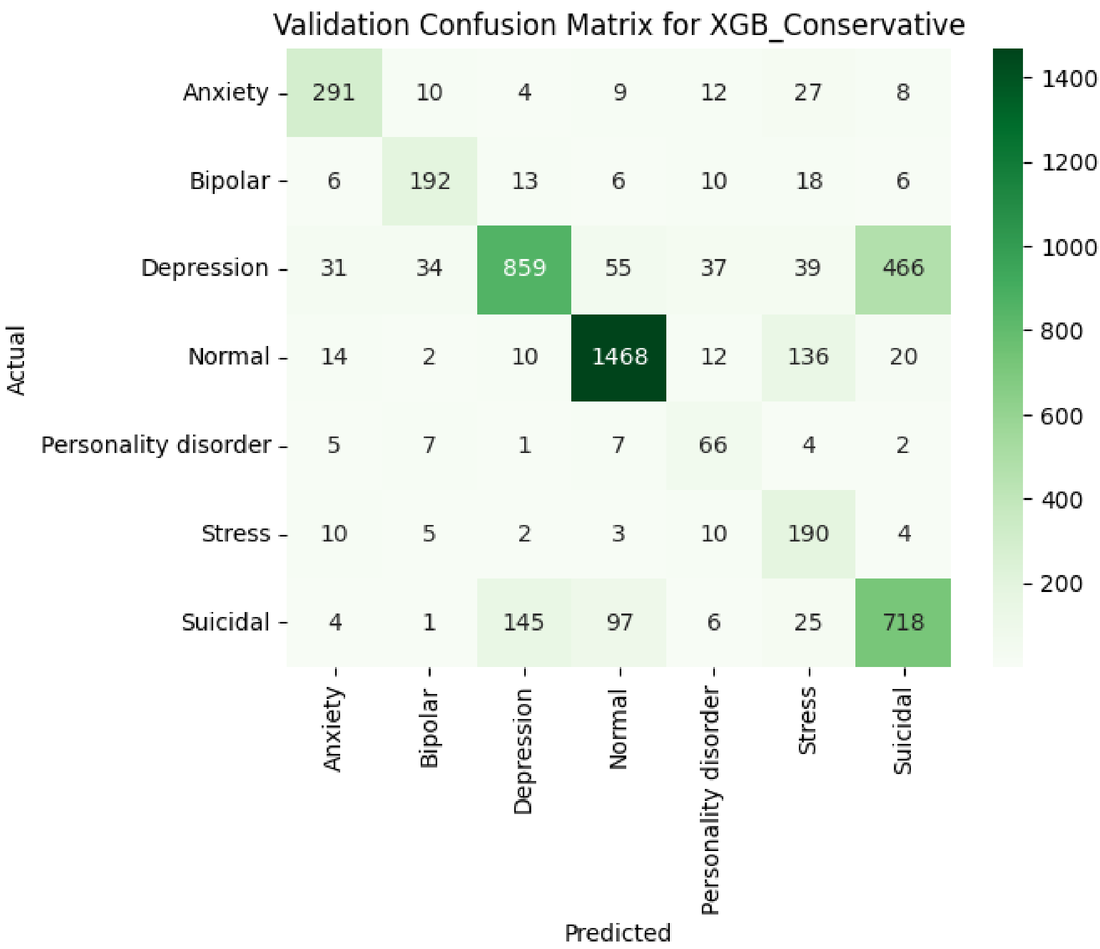
	precision	recall	f1-score	support
Personality disorder	0.85	0.85	0.85	12745
Stress	0.76	0.88	0.81	12745
Suicidal	0.68	0.75	0.72	12745
accuracy			0.80	89215
macro avg	0.80	0.80	0.79	89215
weighted avg	0.80	0.80	0.79	89215



Validation Set Metrics:

	precision	recall	f1-score	support
Anxiety	0.81	0.81	0.81	361
Bipolar	0.76	0.76	0.76	251
Depression	0.83	0.56	0.67	1521
Normal	0.89	0.88	0.89	1662
Personality disorder	0.43	0.72	0.54	92

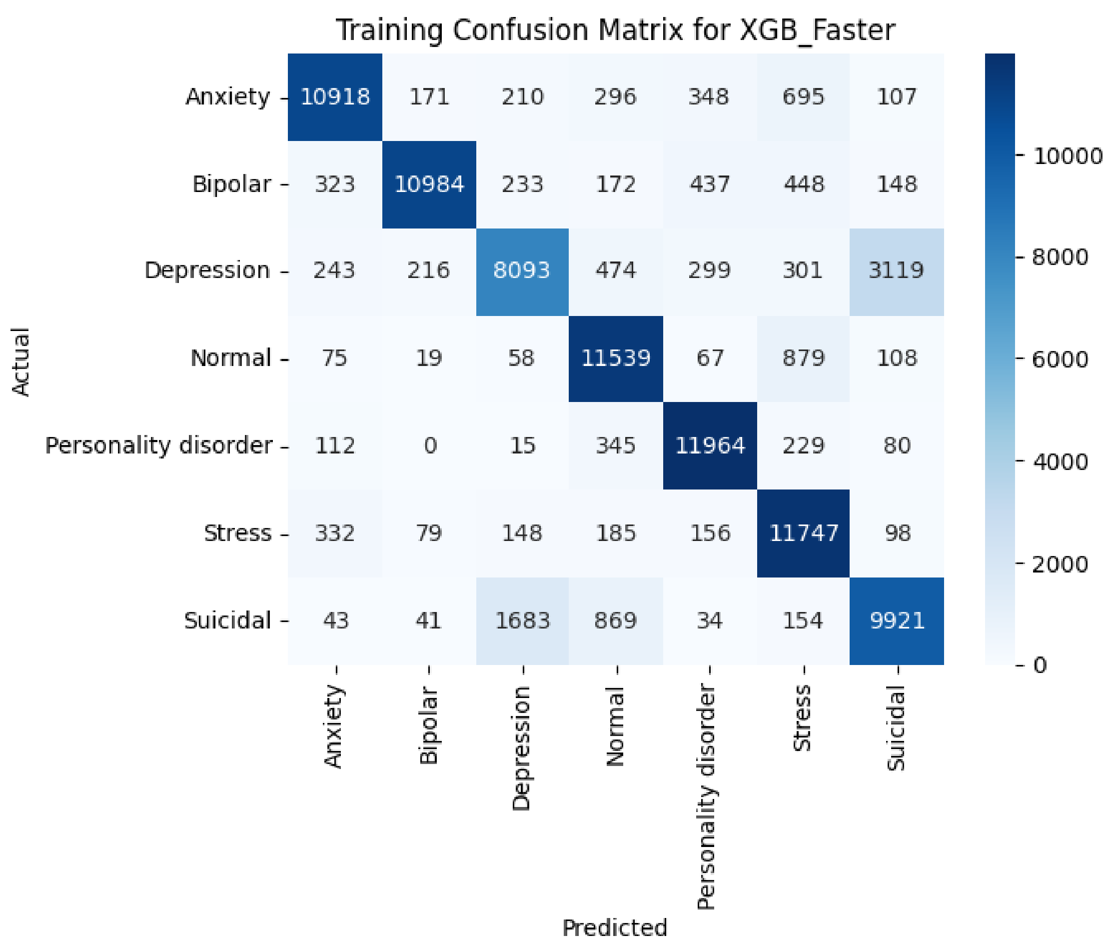
	precision	recall	f1-score	support
Stress	0.43	0.85	0.57	224
Suicidal	0.59	0.72	0.65	996
accuracy			0.74	5107
macro avg	0.68	0.76	0.70	5107
weighted avg	0.77	0.74	0.74	5107




```
For XGB_Faster:
  Training Accuracy: 0.8425264809729306
  Validation Accuracy: 0.7603289602506363

Training Set Metrics:
```

	precision	recall	f1-score	support
Anxiety	0.91	0.86	0.88	12745

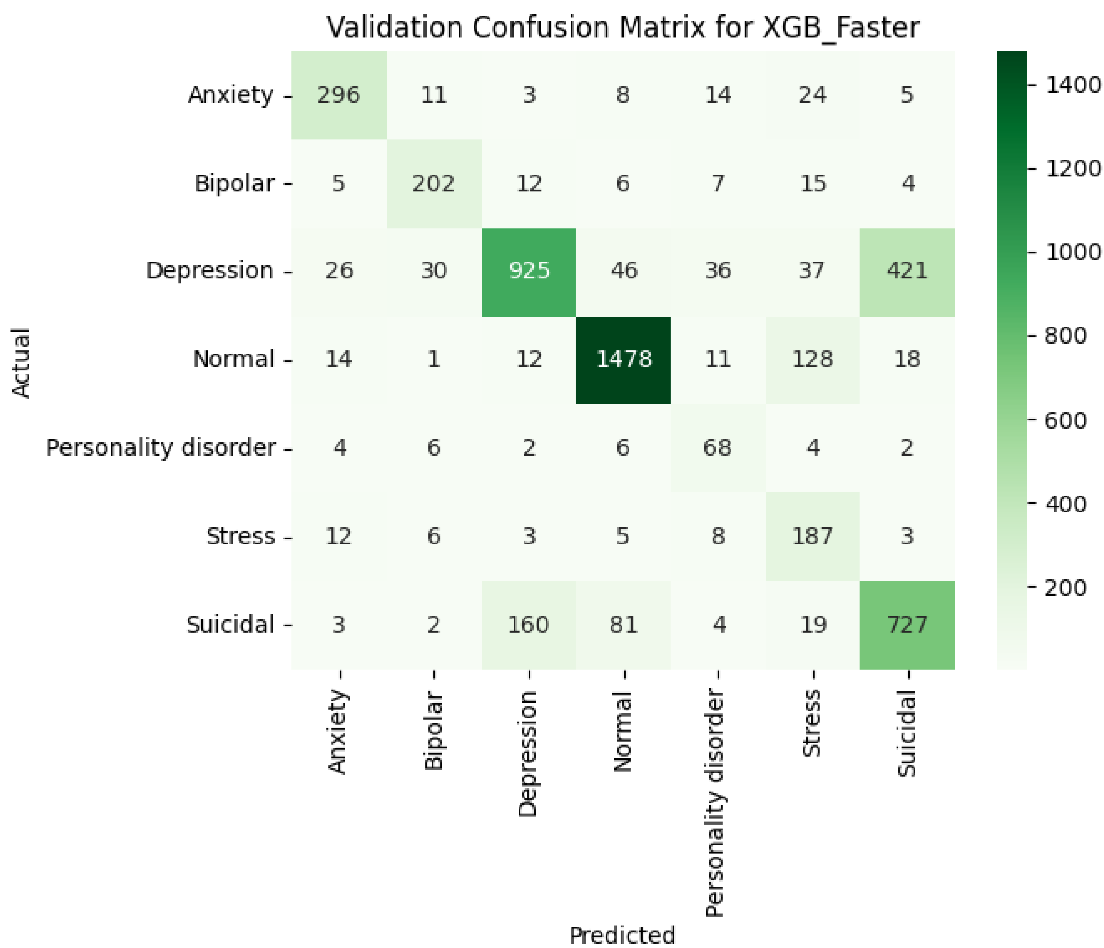
Bipolar	0.95	0.86	0.91	12745
Depression	0.78	0.63	0.70	12745
Normal	0.83	0.91	0.87	12745
Personality disorder	0.90	0.94	0.92	12745
Stress	0.81	0.92	0.86	12745
Suicidal	0.73	0.78	0.75	12745
accuracy			0.84	89215
macro avg	0.84	0.84	0.84	89215
weighted avg	0.84	0.84	0.84	89215



Validation Set Metrics:

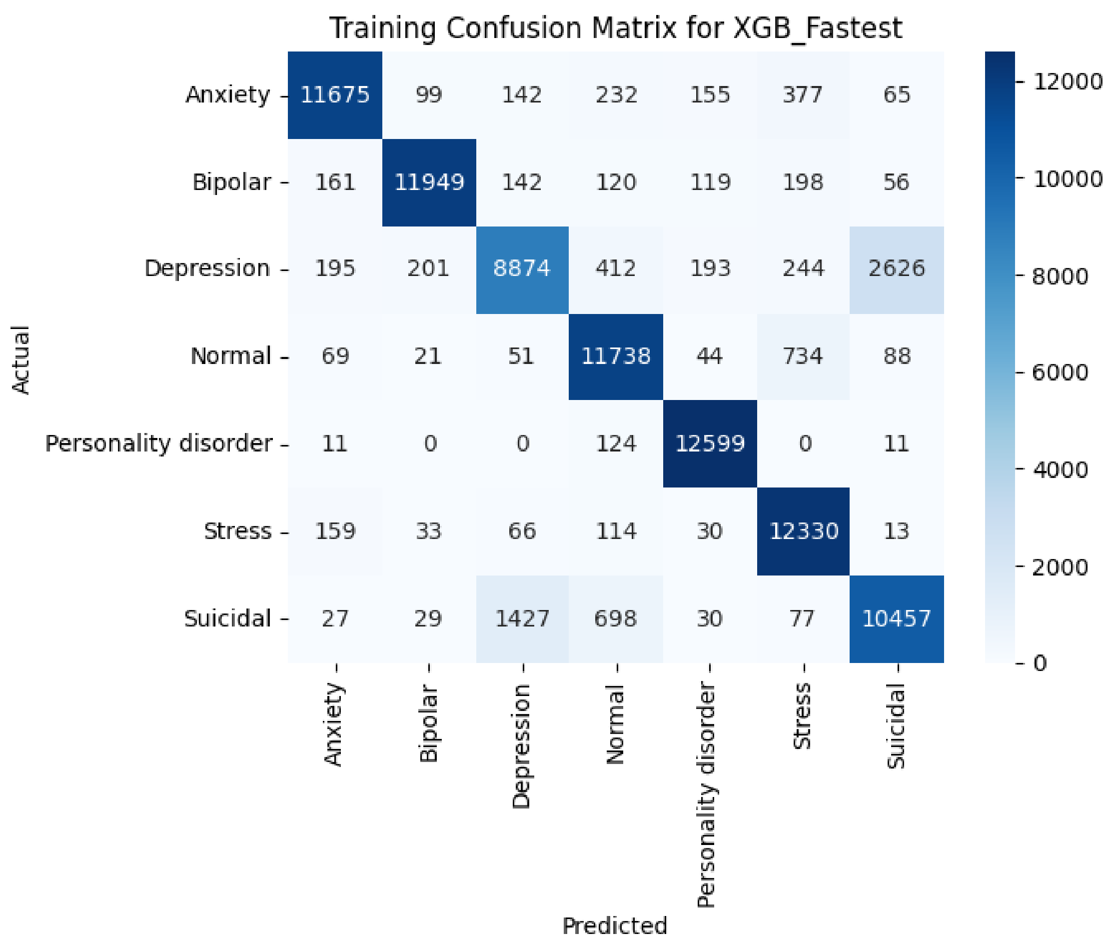
	precision	recall	f1-score	support
Anxiety	0.82	0.82	0.82	361
Bipolar	0.78	0.80	0.79	251

```
       Depression       0.83      0.61      0.70      1521
           Normal       0.91      0.89      0.90      1662
Personality disorder    0.46      0.74      0.57        92
           Stress       0.45      0.83      0.59       224
         Suicidal       0.62      0.73      0.67       996

         accuracy                           0.76      5107
        macro avg       0.70      0.78      0.72      5107
     weighted avg       0.79      0.76      0.76      5107
```




```
For XGB_Fastest:
  Training Accuracy: 0.8924732388051336
  Validation Accuracy: 0.7736440180144899

Training Set Metrics:
```

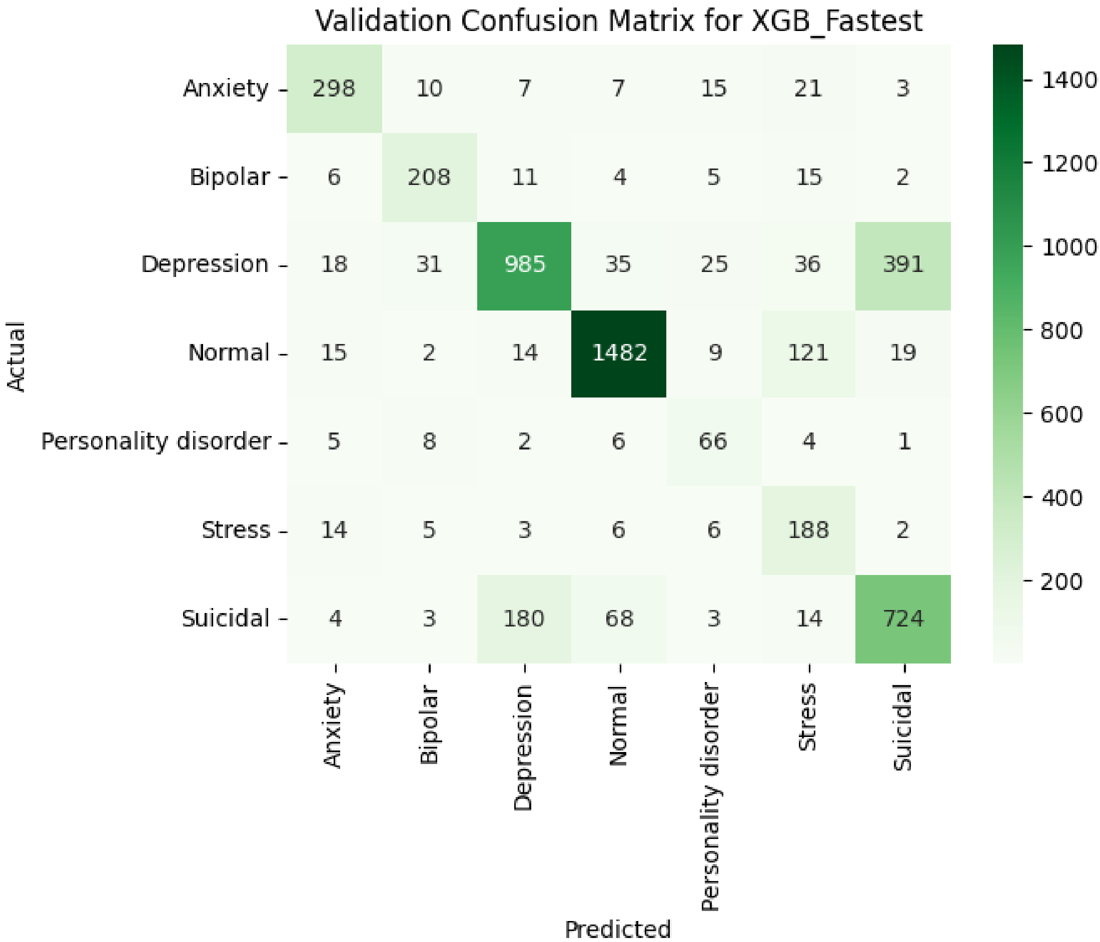
	precision	recall	f1-score	support
Anxiety	0.95	0.92	0.93	12745
Bipolar	0.97	0.94	0.95	12745
Depression	0.83	0.70	0.76	12745
Normal	0.87	0.92	0.90	12745
Personality disorder	0.96	0.99	0.97	12745
Stress	0.88	0.97	0.92	12745
Suicidal	0.79	0.82	0.80	12745
accuracy			0.89	89215
macro avg	0.89	0.89	0.89	89215
weighted avg	0.89	0.89	0.89	89215



Validation Set Metrics:

	precision	recall	f1-score	support

Anxiety	0.83	0.83	0.83	361
Bipolar	0.78	0.83	0.80	251
Depression	0.82	0.65	0.72	1521
Normal	0.92	0.89	0.91	1662
Personality disorder	0.51	0.72	0.60	92
Stress	0.47	0.84	0.60	224
Suicidal	0.63	0.73	0.68	996
accuracy			0.77	5107
macro avg	0.71	0.78	0.73	5107
weighted avg	0.79	0.77	0.78	5107




```
Summary of Accuracy Scores:
XGB_Conservative:
  Training Accuracy: 0.7964692036092585
```

```
  Validation Accuracy: 0.7409438026238496
XGB_Faster:
  Training Accuracy: 0.8425264809729306
  Validation Accuracy: 0.7603289602506363
XGB_Fastest:
  Training Accuracy: 0.8924732388051336
  Validation Accuracy: 0.7736440180144899
XGB_Conservative:
  Training Accuracy: 0.7964692036092585
  Validation Accuracy: 0.7409438026238496
  Difference: 0.055525400985408924
    Model is likely a good fit
XGB_Faster:
  Training Accuracy: 0.8425264809729306
  Validation Accuracy: 0.7603289602506363
  Difference: 0.08219752072229425
    Model is likely a good fit
XGB_Fastest:
  Training Accuracy: 0.8924732388051336
  Validation Accuracy: 0.7736440180144899
  Difference: 0.11882922079064373
    Possible Overfitting
```

3.9 Model Comparison

Here's a comparison of three models based on their training and validation accuracies:

Model	Training Accuracy	Validation Accuracy
Model 1	0.80	0.74
Model 2	0.84	0.76
Model 3	0.89	0.77

Best Model: Model 3

Reason: Model 3 has the highest validation accuracy (0.77). Furthermore, Model 3 had the highest average recall scores.

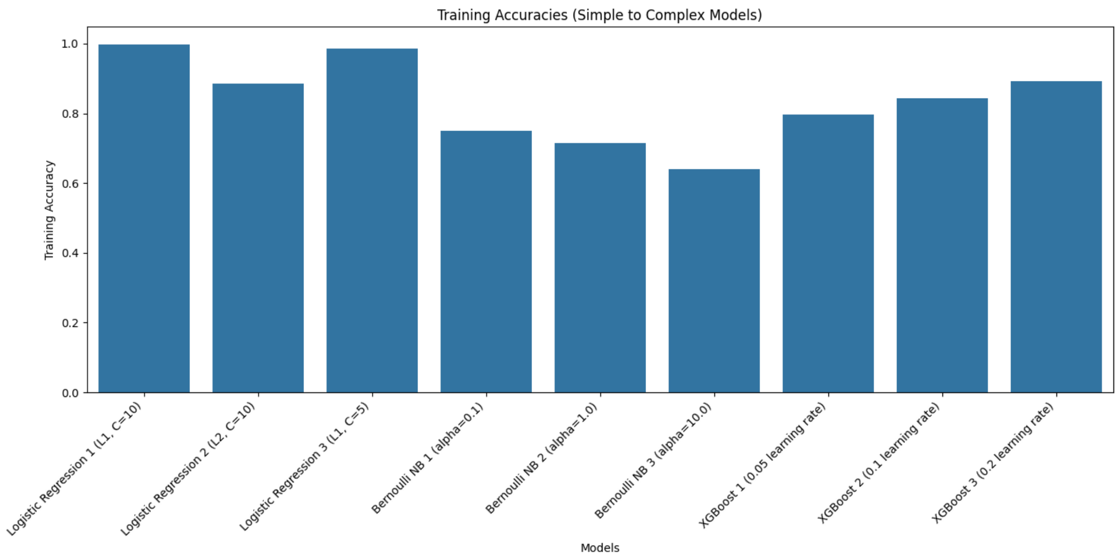
3.9.1 Week 9 Code

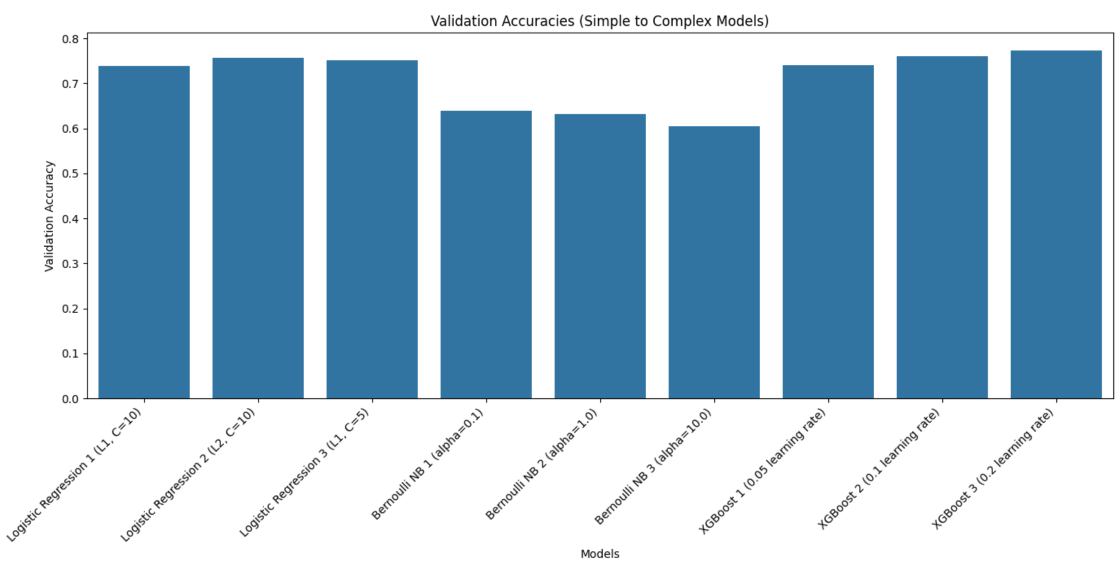
Evaluation of the validation set is shown in previous code. Now we will create plots that show all models and their training accuracies, validation accuracies, training weighted average recall, and validation weighted average recall.

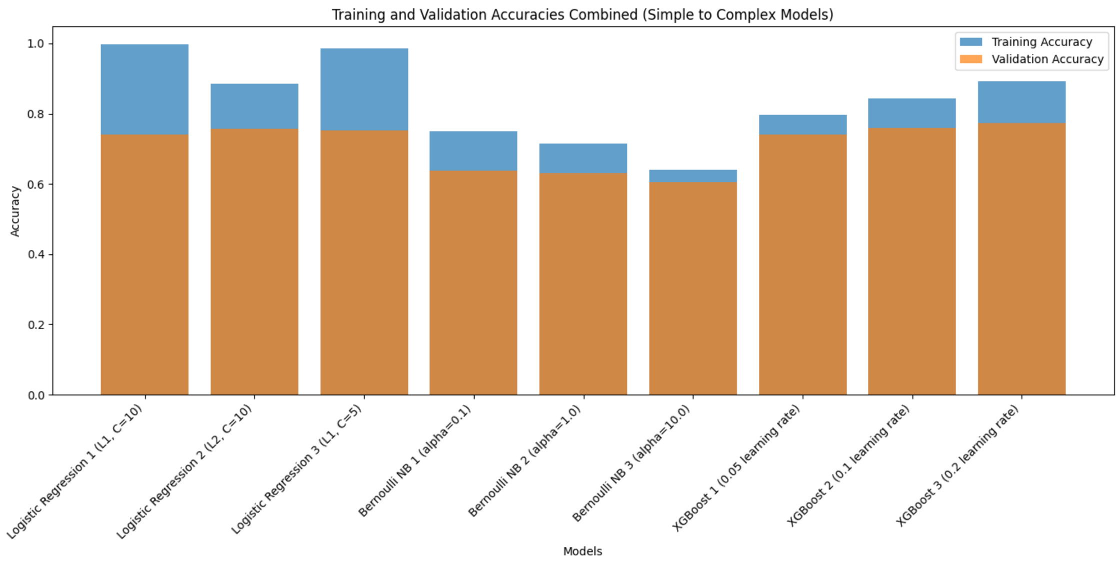
[48]:
```
# Comparing Models on Validation and Training Accuracy plots

# Model names for all 9
model_names = ["Logistic Regression 1 (L1, C=10)", "Logistic Regression 2 (L2,␣
↪C=10)","Logistic Regression 3 (L1, C=5)",
```

```
                "Bernoulli NB 1 (alpha=0.1)", "Bernoulli NB 2 (alpha=1.0)",␣
 ↪"Bernoulli NB 3 (alpha=10.0)",
                "XGBoost 1 (0.05 learning rate)", "XGBoost 2 (0.1 learning␣
 ↪rate)", "XGBoost 3 (0.2 learning rate)"]


# Combine the accuracy scores and model names
training_accuracies_all = training_accuracy_scores + training_accuracy_scores2␣
 ↪+ training_accuracy_scores3
validation_accuracies_all = accuracy_scores + accuracy_scores2 +␣
 ↪accuracy_scores3

# Create a DataFrame for easy sorting and plotting
data = pd.DataFrame({
    'Model': model_names,
    'Training Accuracy': training_accuracies_all,
    'Validation Accuracy': validation_accuracies_all
})

# Plot training accuracies
plt.figure(figsize=(14, 7))
sns.barplot(x='Model', y='Training Accuracy', data=data)
plt.title("Training Accuracies (Simple to Complex Models)")
plt.xlabel("Models")
plt.ylabel("Training Accuracy")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()

# Plot validation accuracies (sorted)
plt.figure(figsize=(14, 7))
sns.barplot(x='Model', y='Validation Accuracy', data=data)
plt.title("Validation Accuracies (Simple to Complex Models)")
plt.xlabel("Models")
plt.ylabel("Validation Accuracy")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()

#Combined Bar plot
plt.figure(figsize=(14,7))

plt.bar(data['Model'], data['Training Accuracy'], label = 'Training Accuracy',␣
 ↪alpha = 0.7)
plt.bar(data['Model'], data['Validation Accuracy'], label = 'Validation␣
 ↪Accuracy', alpha = 0.7)
```

```
plt.xticks(rotation=45, ha='right')
plt.title('Training and Validation Accuracies Combined (Simple to Complex␣
↪Models)')
plt.ylabel('Accuracy')
plt.xlabel('Models')
plt.legend()
plt.tight_layout()
plt.show()
```

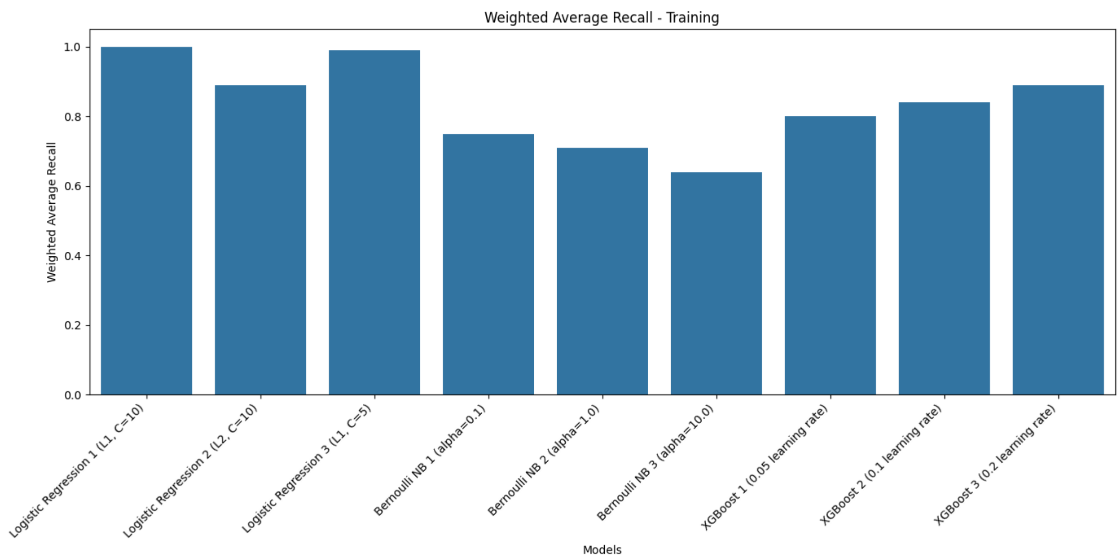


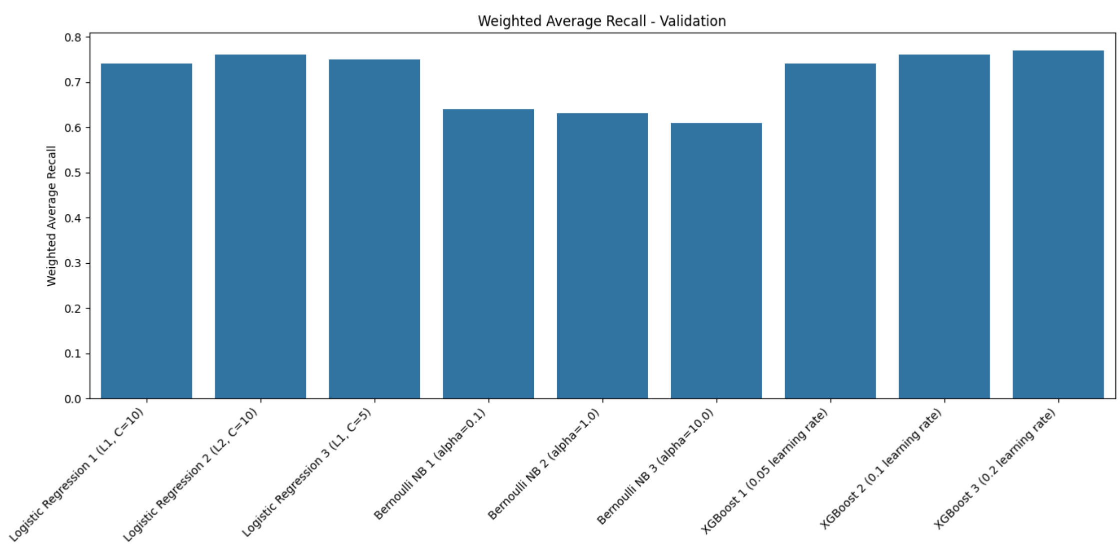




[49]:
```
# Training and Validation Weighted Average Charts

#  weight_avg_recalls_all for training and validation pulled from␣
↪classification report in the same order as the model names above.
weight_avg_recalls_all_train = [1.00, 0.89, 0.99, 0.75, 0.71, 0.64, 0.80, 0.84,␣
↪0.89]
weight_avg_recalls_all_val = [0.74, 0.76, 0.75, 0.64, 0.63, 0.61, 0.74, 0.76, 0.
↪77]

# Create DataFrames for easy plotting
data_train = pd.DataFrame({
    'Model': model_names,
    'Training Recall': weight_avg_recalls_all_train
})

data_val = pd.DataFrame({
    'Model': model_names,
    'Validation Recall': weight_avg_recalls_all_val
})

# Bar plot for Training Recall
plt.figure(figsize=(14, 7))
sns.barplot(x='Model', y='Training Recall', data=data_train)
plt.xlabel("Models")
plt.ylabel("Weighted Average Recall")
plt.xticks(rotation=45, ha="right")
plt.title("Weighted Average Recall - Training")
plt.tight_layout()
```

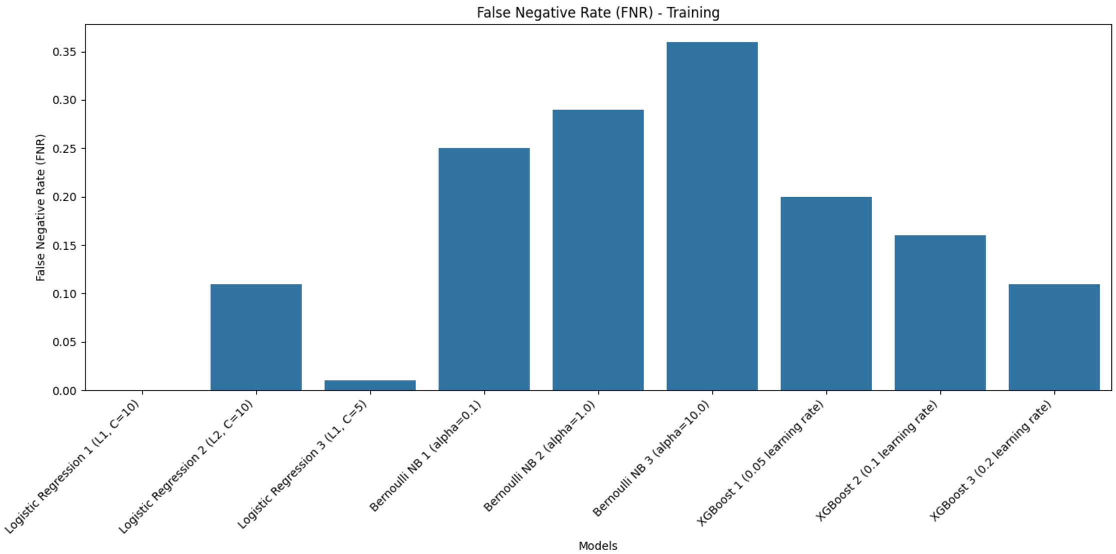
```
plt.show()

# Bar plot for Validation Recall
plt.figure(figsize=(14, 7))
sns.barplot(x='Model', y='Validation Recall', data=data_val)
plt.xlabel("Models")
plt.ylabel("Weighted Average Recall")
plt.xticks(rotation=45, ha="right")
plt.title("Weighted Average Recall - Validation")
plt.tight_layout()
plt.show()
```

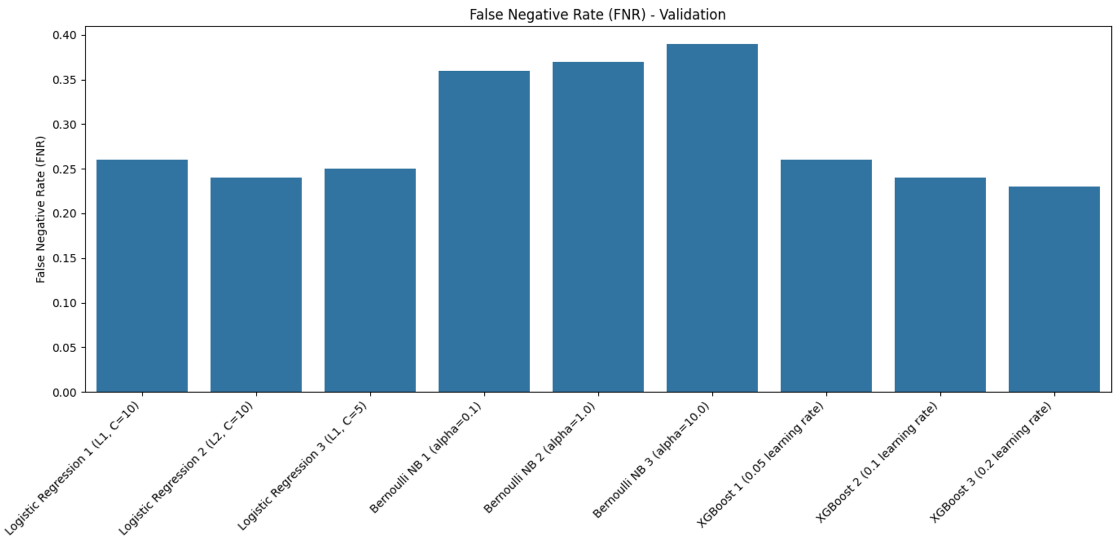




[2]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Training and Validation Weighted Average Charts

# weight_avg_recalls_all for training and validation pulled from classification␣
↪report in the same order as the model names above.
weight_avg_recalls_all_train = [1.00, 0.89, 0.99, 0.75, 0.71, 0.64, 0.80, 0.84,␣
↪0.89]
weight_avg_recalls_all_val = [0.74, 0.76, 0.75, 0.64, 0.63, 0.61, 0.74, 0.76, 0.
↪77]

# Model names for all 9
model_names = ["Logistic Regression 1 (L1, C=10)", "Logistic Regression 2 (L2,␣
↪C=10)","Logistic Regression 3 (L1, C=5)",
               "Bernoulli NB 1 (alpha=0.1)", "Bernoulli NB 2 (alpha=1.0)",␣
↪"Bernoulli NB 3 (alpha=10.0)",
               "XGBoost 1 (0.05 learning rate)", "XGBoost 2 (0.1 learning␣
↪rate)", "XGBoost 3 (0.2 learning rate)"]

# Calculate False Negative Rate (FNR)
false_negative_all_train = [1 - recall for recall in␣
↪weight_avg_recalls_all_train]
weight_avg_recalls_all_val = [1 - recall for recall in␣
↪weight_avg_recalls_all_val]

# Create DataFrames for easy plotting
data_train = pd.DataFrame({
    'Model': model_names,
    'Training FNR': false_negative_all_train
})

data_val = pd.DataFrame({
    'Model': model_names,
    'Validation FNR': weight_avg_recalls_all_val
})

# Bar plot for Training FNR
plt.figure(figsize=(14, 7))
sns.barplot(x='Model', y='Training FNR', data=data_train)
plt.xlabel("Models")
plt.ylabel("False Negative Rate (FNR)")
plt.xticks(rotation=45, ha="right")
plt.title("False Negative Rate (FNR) - Training")
```

```
plt.tight_layout()
plt.show()

# Bar plot for Validation FNR
plt.figure(figsize=(14, 7))
sns.barplot(x='Model', y='Validation FNR', data=data_val)
plt.xlabel("Models")
plt.ylabel("False Negative Rate (FNR)")
plt.xticks(rotation=45, ha="right")
plt.title("False Negative Rate (FNR) - Validation")
plt.tight_layout()
plt.show()
```





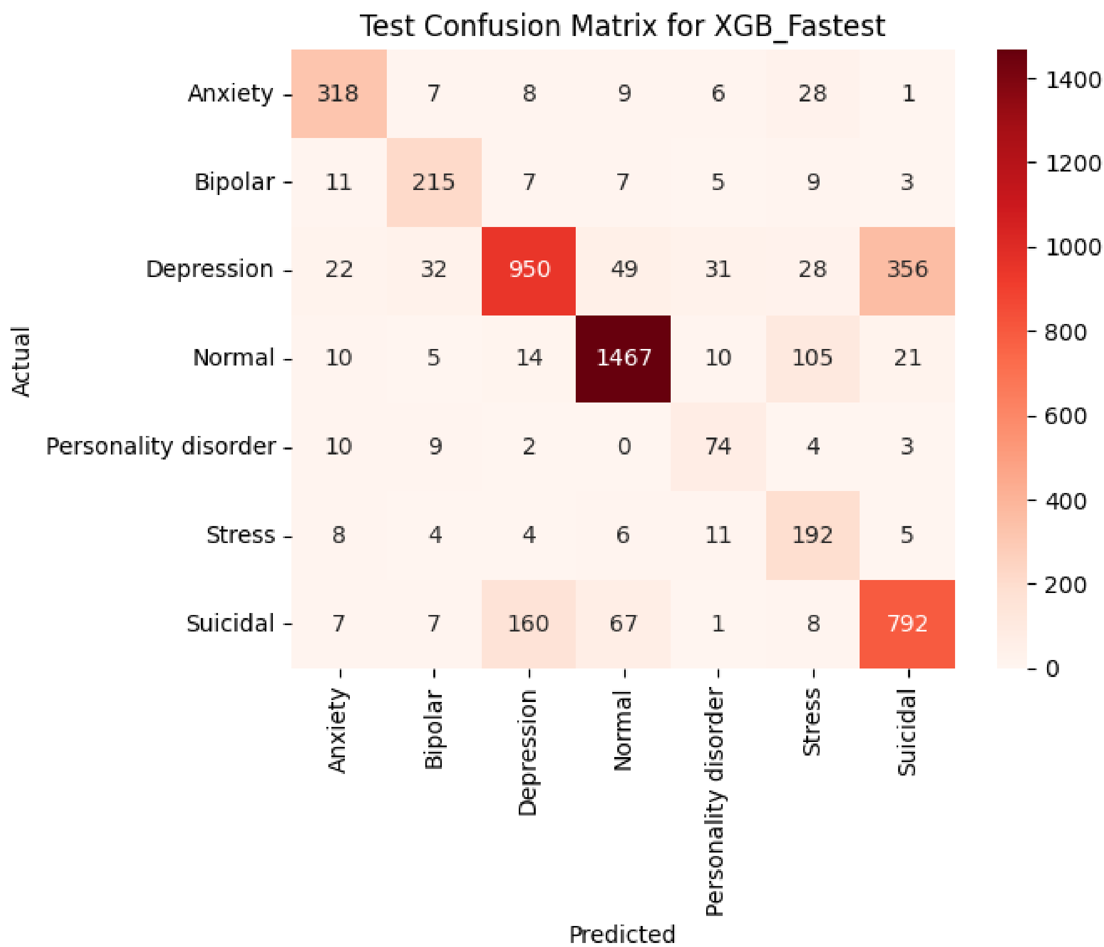
3.9.2 Best Model

The best model is XGBoost 3 with 0.2 learning rate because it has the highest validation accuracy and highest validation average recall. Furthermore, it has a lower difference between training and validation accuracy, which means there is a lower chance of overfitting.

[54]:
```
# Now evaulate the test set using the best model XGBoost Fastest

# Get the last model from classifiers3
last_model_name, last_model = list(classifiers3.items())[-1]

# Predict on the test set using the last model (using test_x_combined)
y_test_pred = last_model.predict(test_x_combined)

# Calculate and print test accuracy
test_accuracy = accuracy_score(test_y, y_test_pred)
print("\n")
print(f"For {last_model_name}:")
print(f"  Test Accuracy: {test_accuracy}")

# Compute and display test confusion matrix and classification report
print("\nTest Set Metrics:")
conf_matrix_test = confusion_matrix(test_y, y_test_pred)
print(classification_report(test_y, y_test_pred, target_names=lbl_enc.classes_))

sns.heatmap(conf_matrix_test, annot=True, fmt='d', cmap='Reds',␣
 ↪xticklabels=lbl_enc.classes_, yticklabels=lbl_enc.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title(f'Test Confusion Matrix for {last_model_name}')
plt.show()

# Optional: Compare Training, Validation, and Test scores.
print(f"{last_model_name}:")
print(f"  Training Accuracy: {training_accuracy_scores3[-1]}")
print(f"  Validation Accuracy: {accuracy_scores3[-1]}")
print(f"  Test Accuracy: {test_accuracy}")
```

```
For XGB_Fastest:
  Test Accuracy: 0.7846515270164448

Test Set Metrics:
                          precision    recall  f1-score   support
```

Anxiety	0.82	0.84	0.83	377
Bipolar	0.77	0.84	0.80	257
Depression	0.83	0.65	0.73	1468
Normal	0.91	0.90	0.91	1632
Personality disorder	0.54	0.73	0.62	102
Stress	0.51	0.83	0.64	230
Suicidal	0.67	0.76	0.71	1042
accuracy			0.78	5108
macro avg	0.72	0.79	0.75	5108
weighted avg	0.80	0.78	0.79	5108




```
XGB_Fastest:
  Training Accuracy: 0.8924732388051336
  Validation Accuracy: 0.7736440180144899
  Test Accuracy: 0.7846515270164448
```