Miguel Soniel and Jan Edgar Tupas CS171 BM1

M3 Summative Assessment

I. Data Collection

The project utilizes facial expression to recognize emotions. The data collected was facial features that provide information for emotion recognition. The features were extracted from raw image files. Seven classes of emotions were identified and 50 instances of datapoints were used for each class resulting in a dataset of 350 samples.

Libraries were imported to extract features from the raw images for data collection. The libraries used were os, numpy, pandas, tensorflow, VGG16 from keras, and preprocess_input from keras.

```
# Libraries needed for data collection
import os
import numpy as np
import pandas as pd
import tensorflow as tf
from keras.applications import VGG16
from keras.applications.vgg16 import preprocess_input
```

Google drive was used as a repository for accessing the raw image files and storing the resulting data files. The drive was mounted using the mount() method.

```
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

The paths of the raw image files grouped according to their specific emotion were specified. This is in preparation for accessing the images and extracting the features.

```
# Specify file paths of raw data
angry_path = '/content/drive/MyDrive/M3SA/Angry'
disgust_path = '/content/drive/MyDrive/M3SA/Disgust'
fear_path = '/content/drive/MyDrive/M3SA/Fear'
happy_path = '/content/drive/MyDrive/M3SA/Happy'
neutral_path = '/content/drive/MyDrive/M3SA/Neutral'
sad_path = '/content/drive/MyDrive/M3SA/Sad'
surprise_path = '/content/drive/MyDrive/M3SA/Surprise'
```

To implement image feature extraction, the VGG16 model from keras was instantiated. The top classification layer was removed to have the model serve as a model for extracting features from the images.

```
# Instantiate the VGG16 model without the top classification layer to be used for feature extraction
model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
```

A function is created and defined to extract the necessary features from the images. The function utilizes keras. preprocessing methods to preprocess the raw image data. The features are then extracted from the raw images using the VGG16 model and returned.

```python
# Define function to extract features from raw data images
def extract_features(img_path):
    img = tf.keras.preprocessing.image.load_img(img_path, target_size=(224, 224))
    x = tf.keras.preprocessing.image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    features = model.predict(x)
    return features.flatten()
```

Lists for filenames and features are initialized. The filenames of the images and the features extracted from the images will be stored in the lists for easy transfer to a dataframe.

```python
# Initialize lists to store image filenames and features
image_filenames = []
image_features = []
```

A loop is created to iterate through the raw image files grouped according to emotion. The loop iterates through the filenames of the images in the list and calls the extract_features function to apply the function on the image, returing the extracted features. The features are appended to the image_features list and the filename is appended to the image_filenames list. The same loop is implemented on all filename path lists for each emotion.

```python
# Iterate through the angry images in the angry folder and extract features
for filename in os.listdir(angry_path):
    if filename.endswith('.jpg'):
        image_path = os.path.join(angry_path, filename)
        features = extract_features(image_path)
        image_filenames.append(filename)
        image_features.append(features)
```

```python
# Iterate through the disgust images in the disgust folder and extract features
for filename in os.listdir(disgust_path):
    if filename.endswith('.jpg'):
        image_path = os.path.join(disgust_path, filename)
        features = extract_features(image_path)
        image_filenames.append(filename)
        image_features.append(features)
```

```python
# Iterate through the fear images in the fear folder and extract features
for filename in os.listdir(fear_path):
    if filename.endswith('.jpg'):
        image_path = os.path.join(fear_path, filename)
        features = extract_features(image_path)
        image_filenames.append(filename)
        image_features.append(features)
```

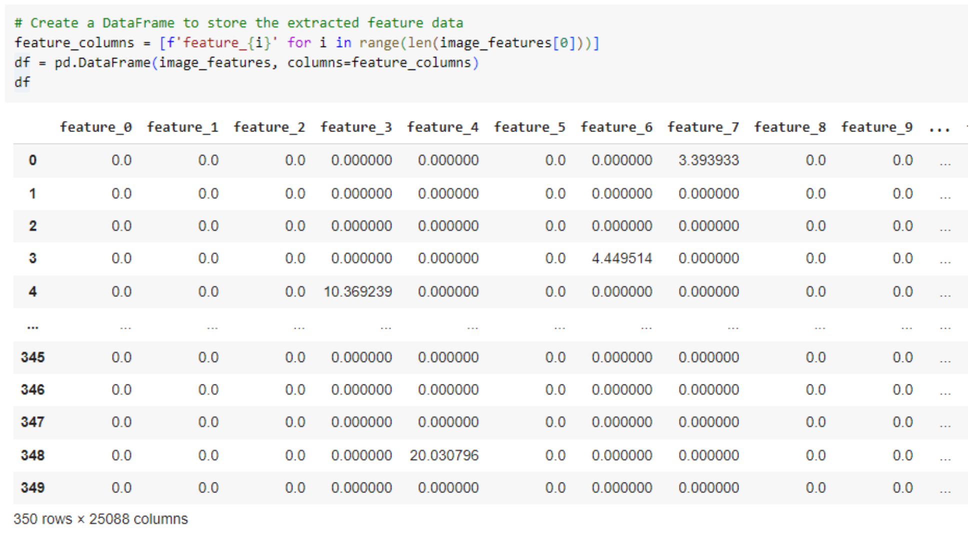
```
# Iterate through the happy images in the happy folder and extract features
for filename in os.listdir(happy_path):
    if filename.endswith('.jpg'):  # Assuming you have only JPG images
        image_path = os.path.join(happy_path, filename)
        features = extract_features(image_path)
        image_filenames.append(filename)
        image_features.append(features)
```

```
# Iterate through the neutral images in the neutral folder and extract features
for filename in os.listdir(neutral_path):
    if filename.endswith('.jpg'):  # Assuming you have only JPG images
        image_path = os.path.join(neutral_path, filename)
        features = extract_features(image_path)
        image_filenames.append(filename)
        image_features.append(features)
```

```
# Iterate through the sad images in the sad folder and extract features
for filename in os.listdir(sad_path):
    if filename.endswith('.jpg'):  # Assuming you have only JPG images
        image_path = os.path.join(sad_path, filename)
        features = extract_features(image_path)
        image_filenames.append(filename)
        image_features.append(features)
```

```
# Iterate through the surprise images in the surprise folder and extract features
for filename in os.listdir(surprise_path):
    if filename.endswith('.jpg'):  # Assuming you have only JPG images
        image_path = os.path.join(surprise_path, filename)
        features = extract_features(image_path)
        image_filenames.append(filename)
        image_features.append(features)
```

The features of each image have now been extracted. A new dataframe is created to store and show the extracted features. We can see that the dataframe has 350 rows indicating a sample of 350 images and 25088 columns indicating 25088 features extracted from the images

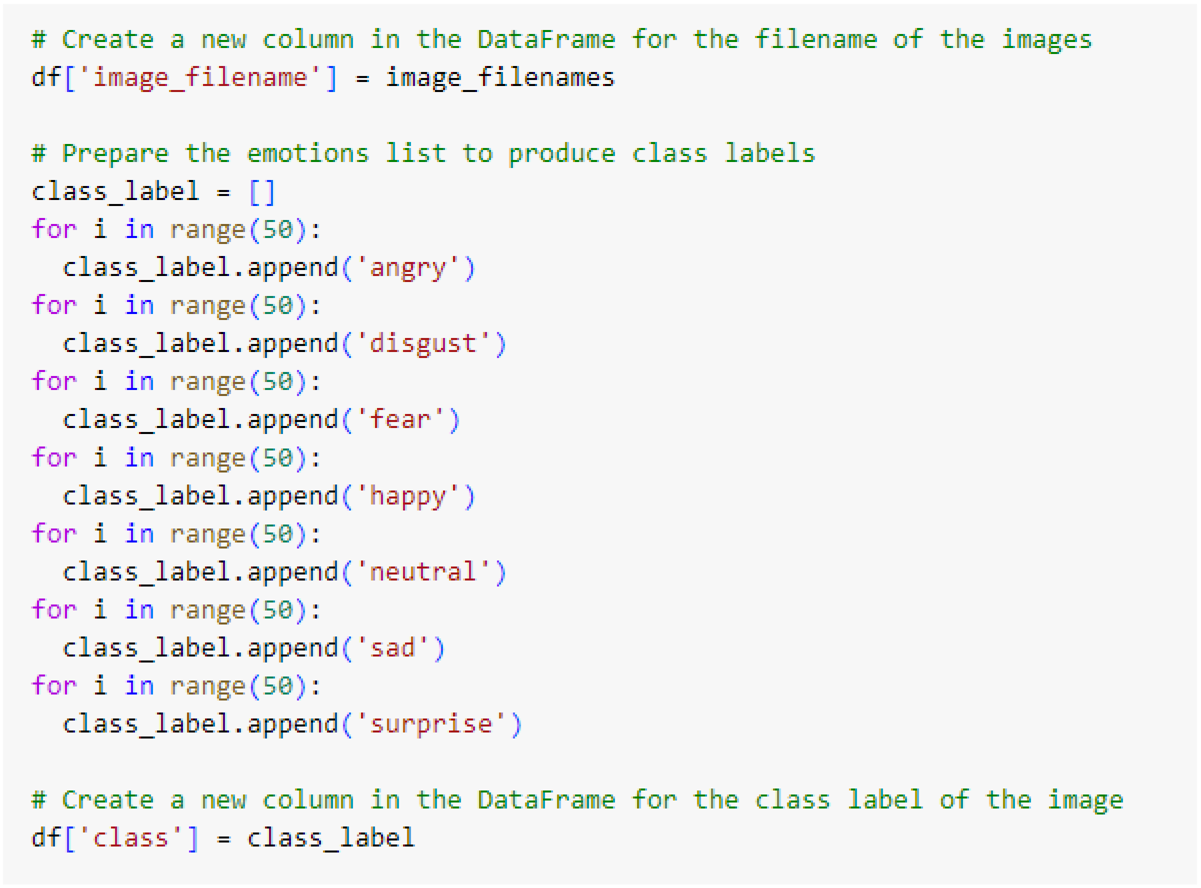


```
# Create a DataFrame to store the extracted feature data
feature_columns = [f'feature_{i}' for i in range(len(image_features[0]))]
df = pd.DataFrame(image_features, columns=feature_columns)
df
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	3.393933	0.0	0.0	...
1	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
2	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
3	0.0	0.0	0.0	0.000000	0.000000	0.0	4.449514	0.000000	0.0	0.0	...
4	0.0	0.0	0.0	10.369239	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
...	...	...	...	...	...	...	...	...	...	...	...
345	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
346	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
347	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
348	0.0	0.0	0.0	0.000000	20.030796	0.0	0.000000	0.000000	0.0	0.0	...
349	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...

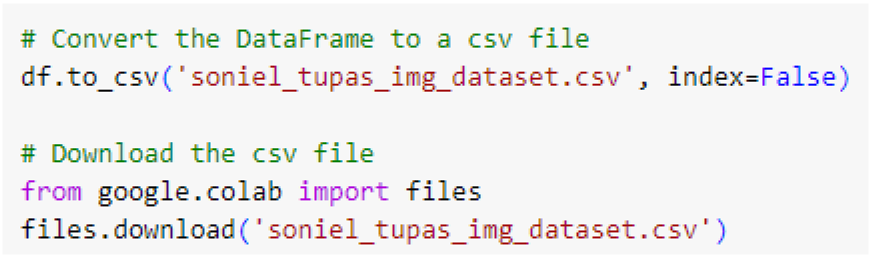
350 rows × 25088 columns

The image filenames are added to the dataframe under a new column image_filename. The emotion classifications are initialized in the list class_label and are transferred to the dataframe under the column class.



```
# Create a new column in the DataFrame for the filename of the images
df['image_filename'] = image_filenames

# Prepare the emotions list to produce class labels
class_label = []
for i in range(50):
  class_label.append('angry')
for i in range(50):
  class_label.append('disgust')
for i in range(50):
  class_label.append('fear')
for i in range(50):
  class_label.append('happy')
for i in range(50):
  class_label.append('neutral')
for i in range(50):
  class_label.append('sad')
for i in range(50):
  class_label.append('surprise')

# Create a new column in the DataFrame for the class label of the image
df['class'] = class_label
```

The dataframe is now complete with the features extracted from the raw images, the image filenames, and the corresponding class label which are the seven emotions. The dataframe is converted to a csv file and then downloaded to the local computer.



```
# Convert the DataFrame to a csv file
df.to_csv('soniel_tupas_img_dataset.csv', index=False)

# Download the csv file
from google.colab import files
files.download('soniel_tupas_img_dataset.csv')
```

II. Data Pre-Processing

Using the VGG16 model, the data was initially processed before feature extraction to ensure proper feature values. We can continue to process the data to attempt to improve its fit into the model translating to better model performance.

Libraries were imported to implement data pre-processing. The libraries used were pandas, numpy, sklearn's MinMaxScaler and sklearn's LabelEncoder

```python
# Libraries needed for data pre-processing
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import LabelEncoder
```

Again, google drive was used as a repository for accessing the raw image files and storing the resulting data files. The drive was mounted using the mount() method.

```python
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

The initial unprocessed dataset was retrieved from the google drive. It was read into the dataframe using the read_csv() method. The dataframe is shown.

```python
# Retrieve initial unprocessed dataset and read to the DataFrame
url = '/content/drive/MyDrive/M3SA/soniel_tupas_img_dataset.csv'
df = pd.read_csv(url)
df
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	3.393933	0.0	0.0	...
1	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
2	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
3	0.0	0.0	0.0	0.000000	0.000000	0.0	4.449514	0.000000	0.0	0.0	...
4	0.0	0.0	0.0	10.369239	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
...	...	...	...	...	...	...	...	...	...	...	...
345	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
346	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
347	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...
348	0.0	0.0	0.0	0.000000	20.030796	0.0	0.000000	0.000000	0.0	0.0	...
349	0.0	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...

350 rows × 25090 columns

The dataset specific class labels and the number of instances of each class are shown using the group.by() method and size() method. We can see that there are seven specific classes namely angry, disgust, fear, happy, neutral, sad and surprise. There are 50 instances for each classes inferring a balanced dataset.

```python
# Show dataset classes and number of instances for each class
df.groupby('class').size()
```

```
class
angry       50
disgust     50
fear        50
happy       50
neutral     50
sad         50
surprise    50
dtype: int64
```

To begin data pre-processing, the number of columns that have 0 as the value for all rows are shown. This is implemented using Boolean comparison and the all() and sum() method. It shows that there are 3688 columns or features whose rows are all 0 values.

```
# Show number of columns whose values for all rows are 0
num_zero_columns = (df == 0.0).all().sum()
print(num_zero_columns)
```

```
3688
```

We can remove the columns as they may contribute unnecessary influence on the dataset. The columns are removed using the loc and copy() method. The dataframe is shown where there are now 21402 columns or features in the dataset.

```
# Remove columns whose values for all rows are 0
df = df.loc[:, (df !=0.0).any(axis=0)].copy()
df
```

	feature_0	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	feature_10	...
0	0.0	0.0	0.000000	0.000000	0.0	0.000000	3.393933	0.0	0.0	0.000000	...
1	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
2	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
3	0.0	0.0	0.000000	0.000000	0.0	4.449514	0.000000	0.0	0.0	1.437616	...
4	0.0	0.0	10.369239	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
...	...	...	...	...	...	...	...	...	...	...	...
345	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
346	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
347	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
348	0.0	0.0	0.000000	20.030796	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
349	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...

350 rows × 21402 columns

We can run the same code to verify that there are no columns that have only 0 values in their rows. It shows that there are 0 columns.

```
# Verify that columns whose row values are 0 have been removed
num_zero_columns = (df == 0.0).all().sum()
print(num_zero_columns)
```

```
0
```

Because columns were removed, the columns names have to be adjusted. A list is created that contains the columns names. The new adjusted column names are implemented on the dataframe.

```
feature_columns = [f'feature_{i}' for i in range(21400)]
feature_columns.append('image_filename')
feature_columns.append('class')
df.columns = feature_columns
df
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.000000	0.000000	0.0	0.000000	3.393933	0.0	0.0	0.000000	...
1	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
2	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
3	0.0	0.0	0.000000	0.000000	0.0	4.449514	0.000000	0.0	0.0	1.437616	...
4	0.0	0.0	10.369239	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
...	...	...	...	...	...	...	...	...	...	...	...
345	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
346	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
347	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
348	0.0	0.0	0.000000	20.030796	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
349	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.000000	...

350 rows × 21402 columns

The feature values in the dataframe should be of the same datatype to ensure consistency and avoid problems during modelling. A loop is created that iterates through all feature value columns in the dataframe and identifies its datatype. The number of instances regarding the datatype is recorded. It is shown that all 21400 feature columns have a datatype of float64.

```
# Show and verify that the feature values have same datatype
floatType = 0
intType = 0
other = 0
for i in range(0, 21400):
  if(df[f'feature_{i}'].dtype) == 'float64':
    floatType += 1
  elif (df[f'feature_{i}'].dtype) == 'int64':
    intType += 1
  else:
    other +=1

print(floatType)
print(intType)
print(other)
```

```
21400
0
0
```

With the feature values in the dataframe, we can attempt to further lessen instances of bias and the influence of possible existing outliers by normalizing the feature values. A MinMaxScaler is used to scale the feature values within range of 0 to 1. The feature values are extracted and stored in X. The MinMaxScaler() is instantiated as scaler and is implemented on the feature values in X using the fit_transform() method. A new dataframe is created that now contains the normalized feature values, the image filename, and the corresponding emotion class label.

```
# Normalize feature values using MinMaxScaler to scale values between 0 and 1
X = df.iloc[:, 0:21400].copy()

scaler = MinMaxScaler()
normalized_features = scaler.fit_transform(X)

# Create new DataFrame for normalized values
feature_columns = [f'feature_{i}' for i in range(21400)]
df2 = pd.DataFrame(normalized_features, columns=feature_columns)
df2['image_filename'] = df['image_filename'].values
df2['class'] = df['class'].values
df2
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.0000	0.0	0.0	0.000000	0.134569	0.0	0.0	0.000000	...
1	0.0	0.0	0.0000	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
2	0.0	0.0	0.0000	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
3	0.0	0.0	0.0000	0.0	0.0	0.290357	0.000000	0.0	0.0	0.049624	...
4	0.0	0.0	0.2766	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
...	...	...	...	...	...	...	...	...	...	...	...
345	0.0	0.0	0.0000	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
346	0.0	0.0	0.0000	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
347	0.0	0.0	0.0000	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
348	0.0	0.0	0.0000	1.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	...
349	0.0	0.0	0.0000	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	...

350 rows × 21402 columns

All values to be used in the model have to be numerical as categorical values will result in errors. The values in the class column, which are the emotions, are categorical values and have to be converted to numerical values. One-hot encoding is implemented where LabelEncoder() is instantiated and used on the values of the class columns using the fit_transform() method. We can see that the classes are now converted to numerical value where angry is now 0, disgust is 1, fear is 2, happy is 3, neutral is 4, sad is 5, and surprised is 6.

```
# Perform one-hot encoding to convert categorical class values to numerical values
label_encoder = LabelEncoder()

df2['class'] = label_encoder.fit_transform(df2['class'])# Show dataset classes and number of instances for each class

df2.groupby('class').size()
```

```
class
0    50
1    50
2    50
3    50
4    50
5    50
6    50
dtype: int64
```

The dataset is randomized using the sample() method. This lessens the instances of bias when the dataset is applied in the model. The dataset is now pre-processed and ready to be used in the model.

```python
# Randomize the order of the dataset
df2 = df2.sample(frac=1, random_state=42)
df2 = df2.reset_index(drop=True)
df2
```

```
     feature_0  feature_1  feature_2  feature_3  feature_4  feature_5  feature_6  feature_7  feature_8  feature_9  ...
0          0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
1          0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
2          0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
3          0.0        0.0   0.066522        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
4          0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
...        ...        ...        ...        ...        ...        ...        ...        ...        ...        ...  ...
345        0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
346        0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
347        0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
348        0.0        0.0   0.000000        1.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
349        0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
350 rows × 21402 columns
```

The dataframe is converted into a csv file using the to_csv() method. The csv file is then downloaded using the download() method. The csv file containing the cleansed dataset can now be used for exploratory analysis and modelling.

```python
# Convert the DataFrame of the pre-processed data into csv file
df2.to_csv('soniel_tupas_cleansed_dataset.csv', index=False)

# Download the csv file
from google.colab import files
files.download('soniel_tupas_cleansed_dataset.csv')
```

III. Exploratory Data Analysis

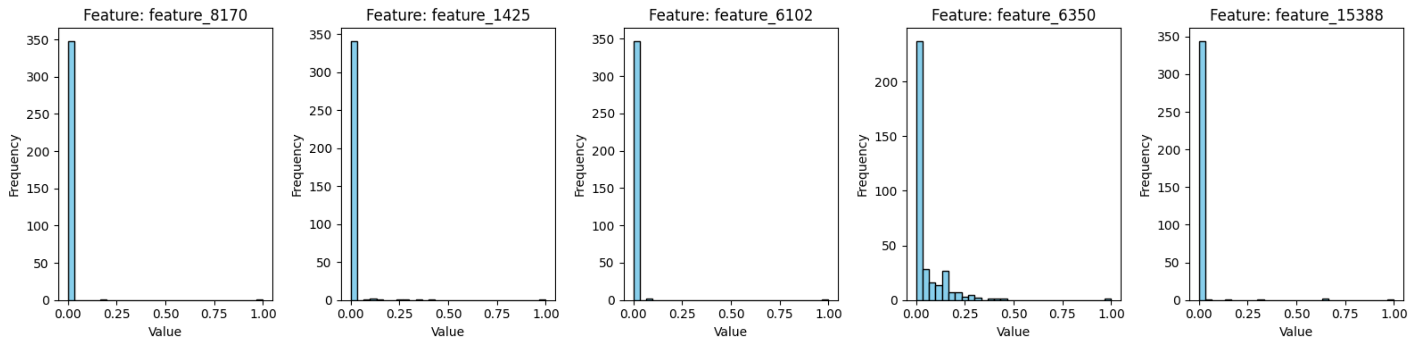
Visualizing the data may provide some insight to the given dataset. It may aid in identifying patterns in preparation for modelling. Libraries were imported to visualize the data. The libraries used were numpy, pandas, seaborn, matplotlib, and sklearn's PCA.

```python
# Libraries needed for Visualization
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
```

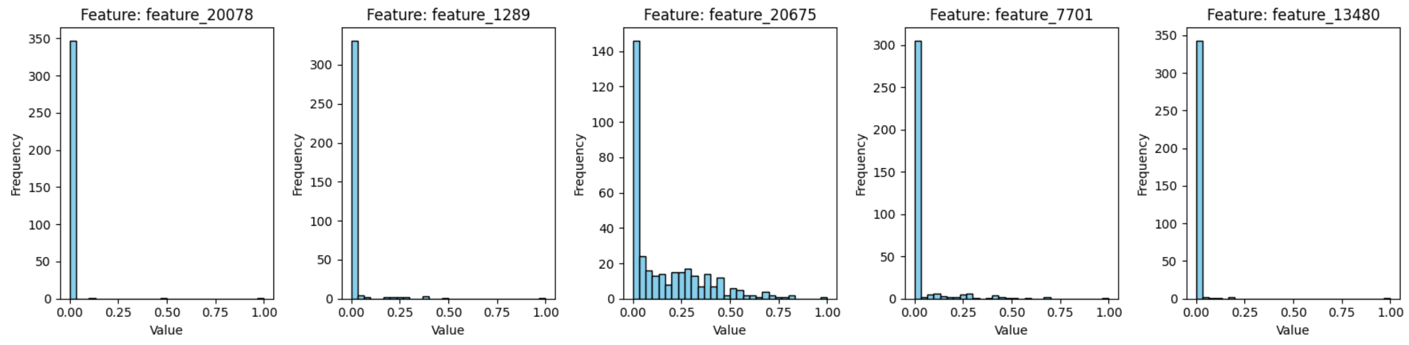
Histograms were used to visualize the frequency of the feature values in 5 feature samples. A loop was created that iterates through the selected feature sample and plots their data in the histogram. The code was run 4 times to show the histogram of 20 random features.

```
# Histogram
sample_size = 5
random_features = df.drop(columns=['class']).sample(n=sample_size, axis=1)

# Create a figure and axis for plotting
fig, ax = plt.subplots(nrows=1, ncols=sample_size, figsize=(16, 4))

# Loop through each sampled feature and plot the histogram
for i, feature_name in enumerate(random_features.columns):
    ax[i].hist(random_features[feature_name], bins=30, color='skyblue', edgecolor='black')
    ax[i].set_title(f'Feature: {feature_name}')
    ax[i].set_xlabel('Value')
    ax[i].set_ylabel('Frequency')

# Adjust the layout and display the plot
plt.tight_layout()
plt.show()
```

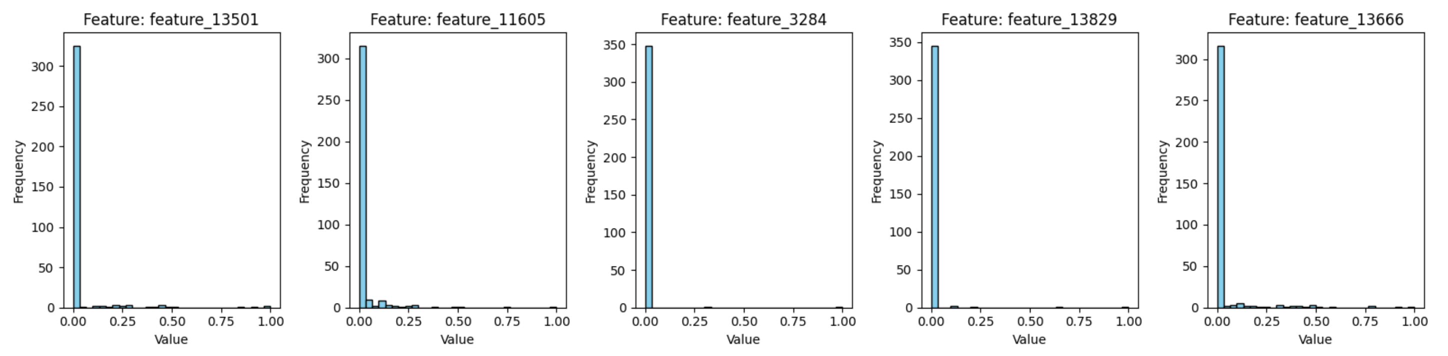
In the 5 feature samples, only feature 6350 manifests some distribution with respect to feature values. Feature 8170, 1425, 6102, and 15388 manifest distributions that are somewhat insignificant visually.



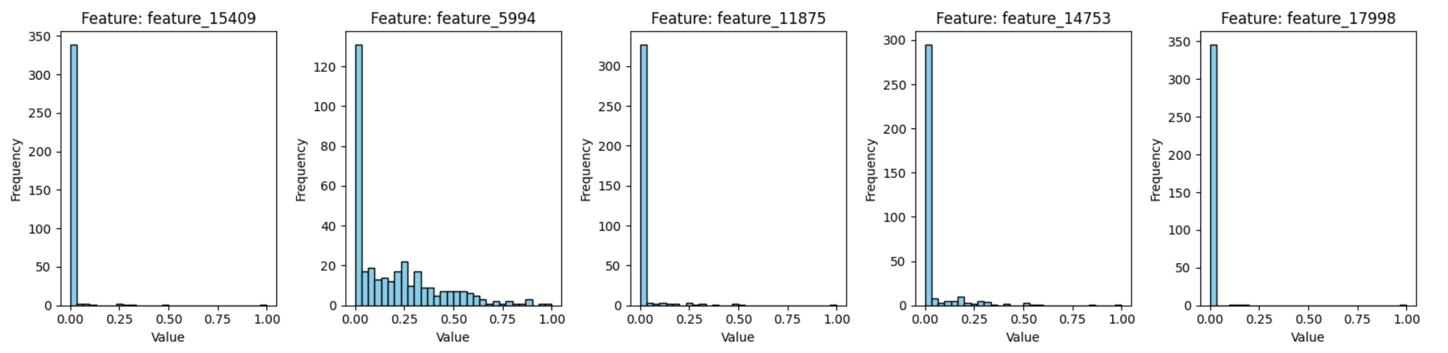
In the 5 feature samples, only feature 20675 manifests visible distribution with respect to feature values. Feature 20078, 1289, 7701, and 13480 manifest distributions that are somewhat insignificant visually.



In the 5 feature samples all features manifest distributions that are somewhat insignificant visually. High frequencies of values lie in between 0 and 0.25



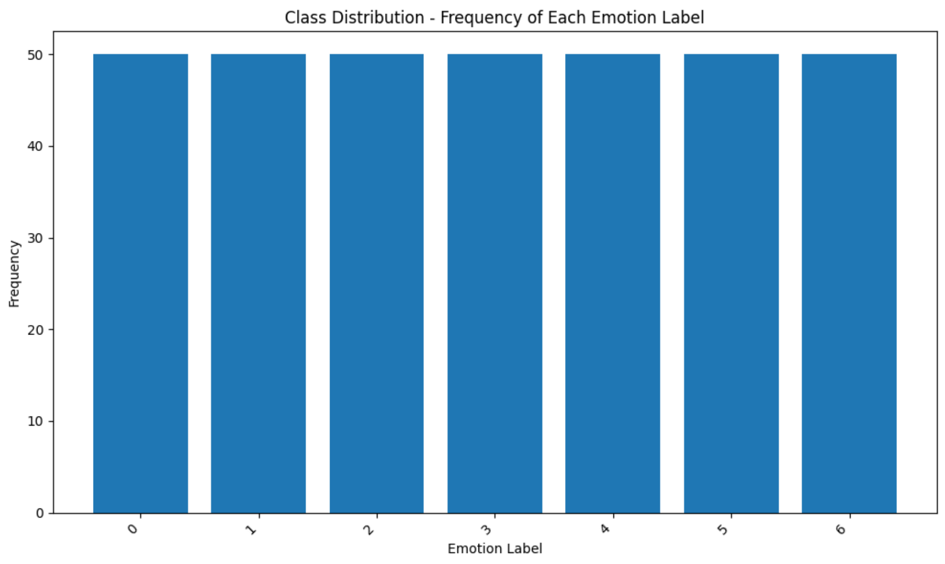
In the 5 feature samples, only feature 5994 manifests visible distribution with respect to feature values where high frequencies are in 0 to 0.25 and then decreases. Feature15409, 11875, 14753, and 17998 manifest distributions that are somewhat insignificant visually.



A bar plot is used the visualize the distribution of classes in the dataset. The class values are extracted along with the number of instances of each class and are plotted.

```python
import pandas as pd
import matplotlib.pyplot as plt

# Count the occurrences of each emotion label
emotion_counts = df['class'].value_counts()

# Sort the emotion labels by frequency in descending order
emotion_counts = emotion_counts.sort_values(ascending=False)

# Create a bar plot to visualize the class distribution
plt.figure(figsize=(10, 6))
plt.bar(emotion_counts.index, emotion_counts.values)
plt.xlabel('Emotion Label')
plt.ylabel('Frequency')
plt.title('Class Distribution - Frequency of Each Emotion Label')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

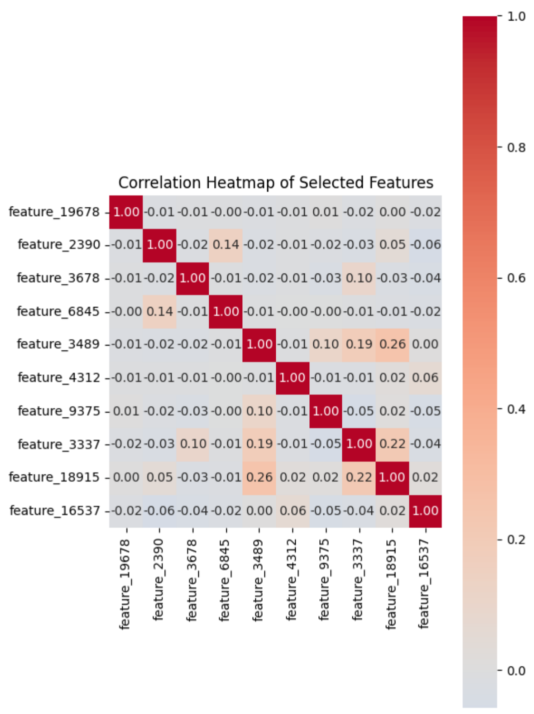
We can see that the classes are equally distributed in terms of their number of instances in the dataset. There are 50 instances for each class inferring that the dataset is balanced.



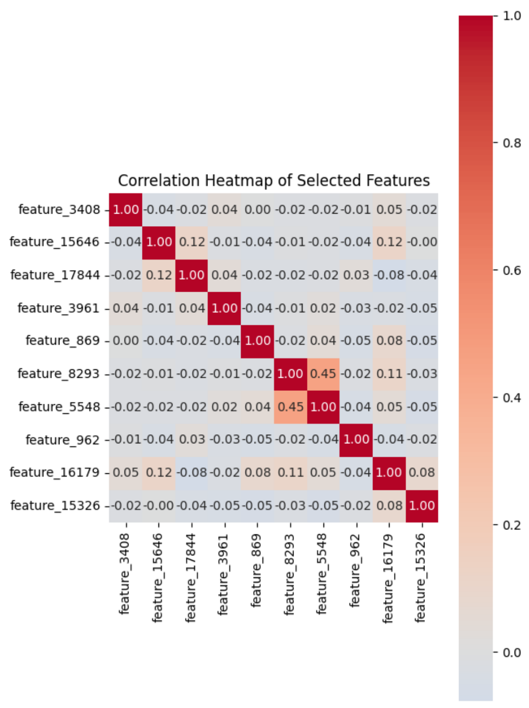
A correlation heatmap is used to visualize the correlation between the features. A correlation matrix is created from the selected sample of features and then used for visualization in the heatmap where warmer colors indicate positive correlations and coolers colors indicate negative correlations.

```
# Sample a subset of features for visualization
num_features_to_visualize = 10
selected_features = df.drop(columns=['class']).sample(n=num_features_to_visualize, axis=1)

# Calculate the correlation matrix
correlation_matrix = selected_features.corr()

# Create the heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, cmap='coolwarm', annot=True, fmt='.2f', center=0, square=True)
plt.title('Correlation Heatmap of Selected Features')
plt.tight_layout()
plt.show()
```

The heatmap for a selected sample of 10 features show little instances of positive correlations. There are correlations the feature values of 1 for each feature which is attributed to the normalization procedures implemented. There are more cool areas shown than warm areas, inferring more negative correlations over positive ones.



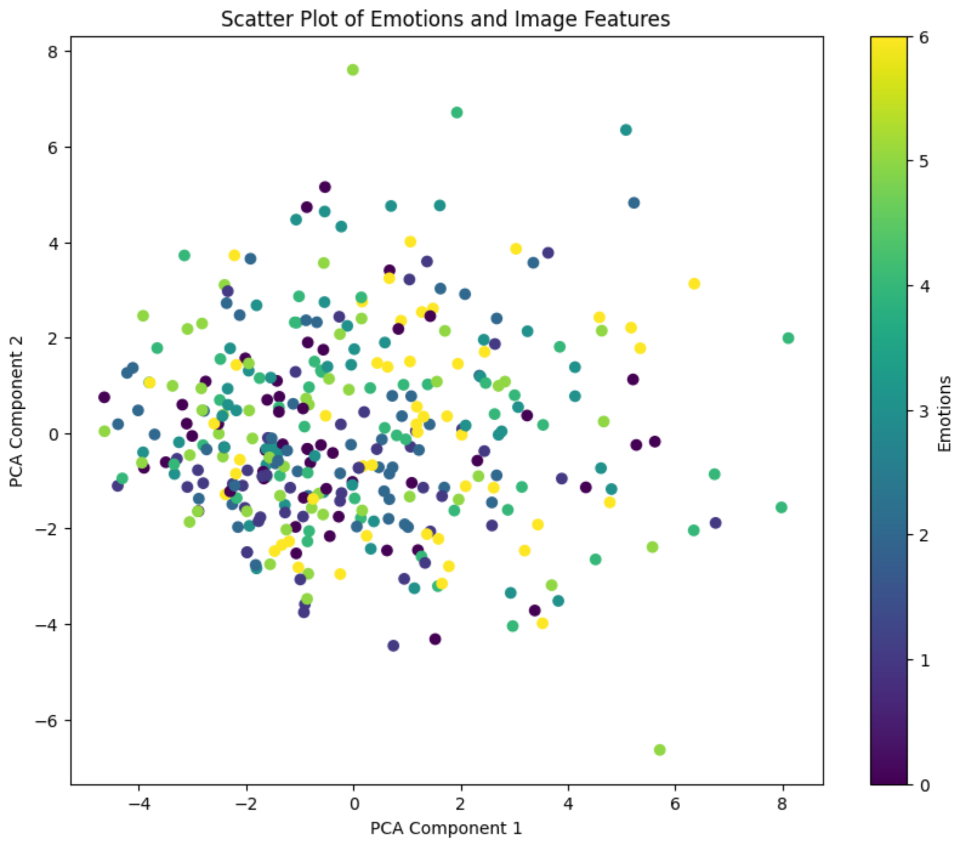
The heatmap for another selected sample of 10 features show similar results where small instances of positive correlations are detected. These correlations indicate that the feature values are consistently equal to 1 for each feature, which can be attributed to the applied normalization procedures. The visualization reveals a higher number of cool areas compared to warm areas, suggesting a prevalence of negative correlations over positive ones.



A scatterplot is used to visualize the class emotions based on two PCA components. Given that there are too many features and cannot fit in a scatterplot, dimensionality using PCA is used to represent 2 features. The PCA is acquired using the pca.fit_transform() method and is used for scatterplot visualization.

```
# Extract emotions and image features
emotions = df['class']
image_features = df.drop(columns=['class'])

# Perform dimensionality reduction using PCA
num_components = 2
pca = PCA(n_components=num_components)
reduced_features = pca.fit_transform(image_features)

# Create the scatter plot
plt.figure(figsize=(10, 8))
plt.scatter(reduced_features[:, 0], reduced_features[:, 1], c=emotions, cmap='viridis')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.title('Scatter Plot of Emotions and Image Features')
plt.colorbar(label='Emotions')
plt.show()
```

The scatterplot shows that the features of the emotions tend to overlap each other which signifies the lack of information provided by the features that differentiates one emotion to another. This means that the model may have difficulty in correctly classifying the datapoints based on the features. The model may inaccurately classify the datapoints because distinctions between the emotions based on the features are not clear.

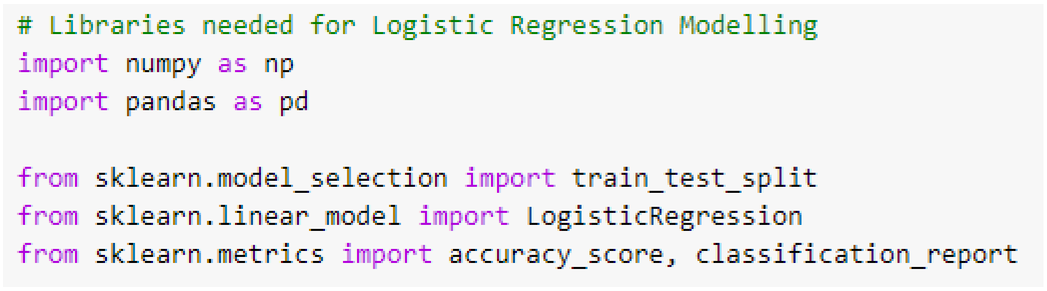


IV. Model Creation and Implementation

Seven models were created using Sklearn to implement classification using the generated dataset. The models used were Logistic Regression, Support Vector Machine, K-Nearest Neighbor, Gradient Boosting Machine, Random Forest, Decision Tree, and Naïve Bayes.

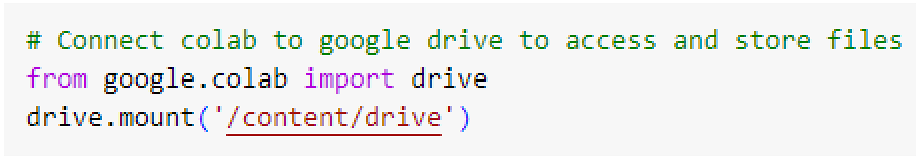
Logistic Regression

Libraries were imported to implement the logistic regression model. The libraries used were numpy, pandas, sklearn's train_test_split, sklearn's LogisticRegression, and sklearn's accuracy_score and classification_report



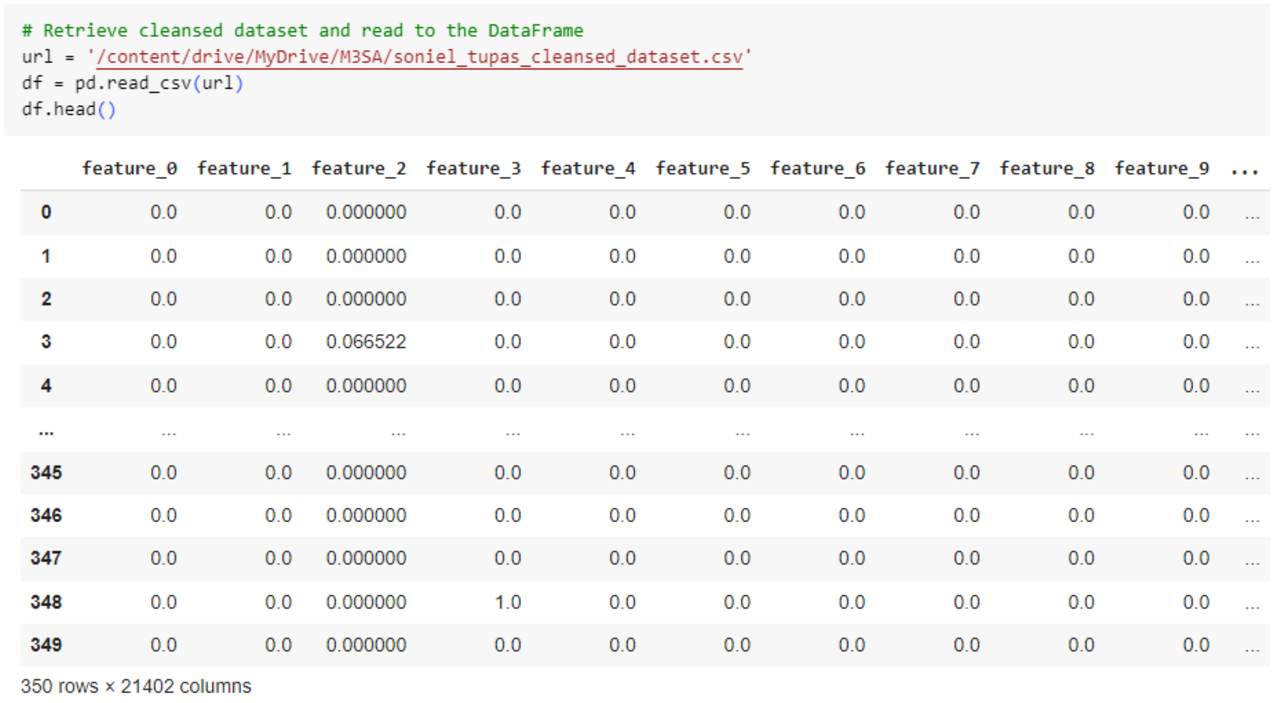
```
# Libraries needed for Logistic Regression Modelling
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
```

Google drive was used as a repository for accessing the raw image files and storing the resulting data files. The drive was mounted using the mount() method.



```
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

The dataset was retrieved from the google drive. It was read into the dataframe using the read_csv() method. The dataframe is shown.



```
# Retrieve cleansed dataset and read to the DataFrame
url = '/content/drive/MyDrive/M3SA/soniel_tupas_cleansed_dataset.csv'
df = pd.read_csv(url)
df.head()
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
1	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
2	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
3	0.0	0.0	0.066522	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
4	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
...	...	...	...	...	...	...	...	...	...	...	...
345	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
346	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
347	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
348	0.0	0.0	0.000000	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...
349	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

350 rows × 21402 columns

To prepare for modelling, the data was segmented. The feature values were stored in variable X. The class labels, which were the emotions, were stored in y.

```python
# Prepare the data for modelling
X = df.drop(columns=['image_filename','class'])
y = df['class']
```

The data was split into training and testing sets using the train_test_split() method. Stratified sampling was utilized as specified in the argument. The splitting was done on a 75-25 ratio and a random state of 42.

```python
# Split the data into training and testing sets using stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, random_state=42)
```

The Logistic Regression was instantiated as model with a max iteration of 1000. The model was then fit with the training set to train the model in predicting the class labels of the given feature values.

```python
# Initialize the Logistic Regression model and train the classifier
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
```

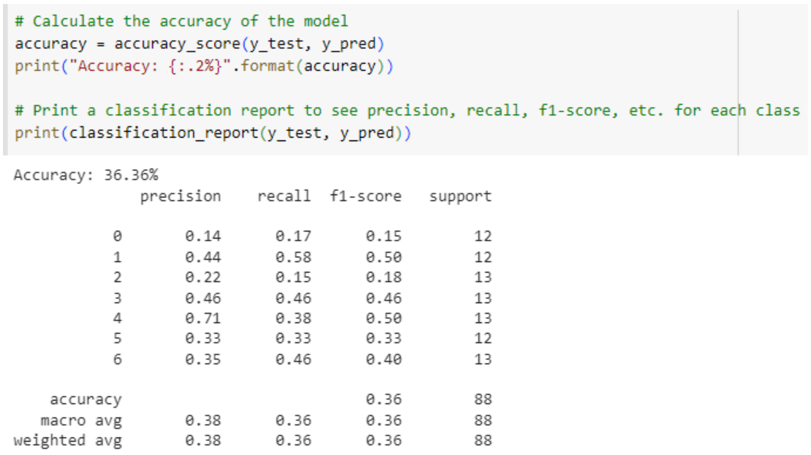
```
      LogisticRegression
LogisticRegression(max_iter=1000)
```

After training, predictions on the test set were made using the model. This was implemented using the predict() method and the results were stored in y_pred.

```python
# Use the trained model to make predictions on the test set
y_pred = model.predict(X_test)
```

Performance metrics were used to evaluate the performance of the model in correctly classifying the datapoints according to their corresponding emotion using the feature values. Accuracy represents the proportion of correctly predicted instances out of the total number of instances in the dataset. Precision measures the accuracy of the positive predictions made by the model for a class. Recall measures the ability of the model to identify all the positive samples correctly. F1 is the harmonic mean of precision and recall. It provides a balance between precision and recall and is often used as a single metric to evaluate the overall performance of a classifier for a specific class. Support is the number of samples in the dataset that belong to a particular class. It represents the frequency of occurrence of that class in the dataset.

It is shown that the accuracy of the model is 36.36%. This implies that that the model correctly predicted the target class for only 36.36% of the instances in the dataset which is relatively low and means that the model performed poorly. The precision, recall, f1-score, and support scores of the model for each class label is also shown. The highest precision score that the model received was 0.71 or 71% in class label 4 and the lowest score received was 0.14 or 14% in class label 0. The highest recall score that the model received was 0.58 or 58% in class label 2 and the lowest score received was 0.17 or 17% in class label 0. The highest f1-score that the model received was 0.50 or 50% in class label 1 and 4 and the lowest score received was 0.15 or 15% in class label 0. It can be observed that while the model performed poorly manifested in the low accuracy, precision, recall, and f1 scores, the manifested the poorest performance in classifying the proper datapoints in class label 0 which is the emotion angry. The macro average and weighted average precision, recall, and f1 scores of the model are 0.38 or 38%, 0.36 or 36% and 0.36 or 36% respectively. The scores received by the model which define the model's performance in emotion recognition through facial expression infer a poor performance with an accuracy of 36.36%, precision of 38%, recall of 36%, and f-1 score of 36%. The model struggles to identify the patterns in the dataset provided by the feature values extracted from the images. It is possible that the model requires more data for training to improve its performance. It is also possible that the data used does not provide sufficient information that the model can use to identify patterns and generate insights for classification. Over-all, adjustments to the dataset and the model have to be made to improve the performance of the model for emotion recognition through facial expression.

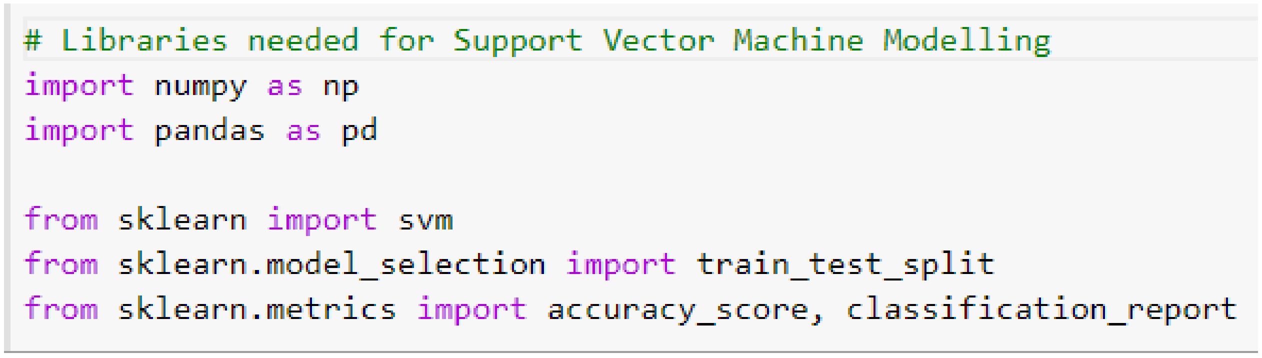


```
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2%}".format(accuracy))

# Print a classification report to see precision, recall, f1-score, etc. for each class
print(classification_report(y_test, y_pred))
```

```
Accuracy: 36.36%
              precision    recall  f1-score   support

           0       0.14      0.17      0.15        12
           1       0.44      0.58      0.50        12
           2       0.22      0.15      0.18        13
           3       0.46      0.46      0.46        13
           4       0.71      0.38      0.50        13
           5       0.33      0.33      0.33        12
           6       0.35      0.46      0.40        13

    accuracy                           0.36        88
   macro avg       0.38      0.36      0.36        88
weighted avg       0.38      0.36      0.36        88
```

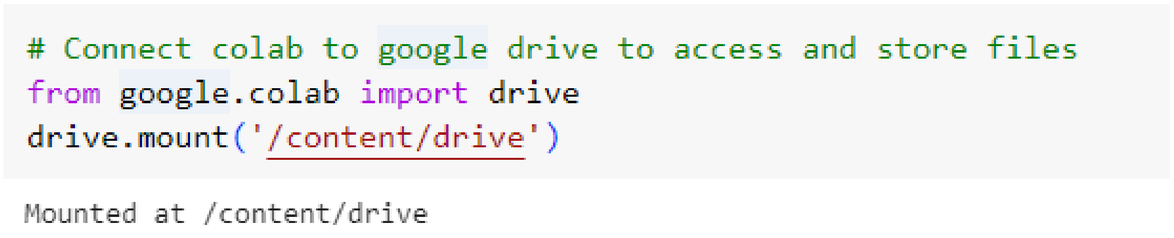
Support Vector Machine

Libraries that are needed to implement Support Vector Machine (SVM) Modelling were imported. The libraries used were numpy, pandas, and sklearn. Specifically, in sklearn, svm, train_test_split, accuracy_score and classification_report.



```
# Libraries needed for Support Vector Machine Modelling
import numpy as np
import pandas as pd

from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

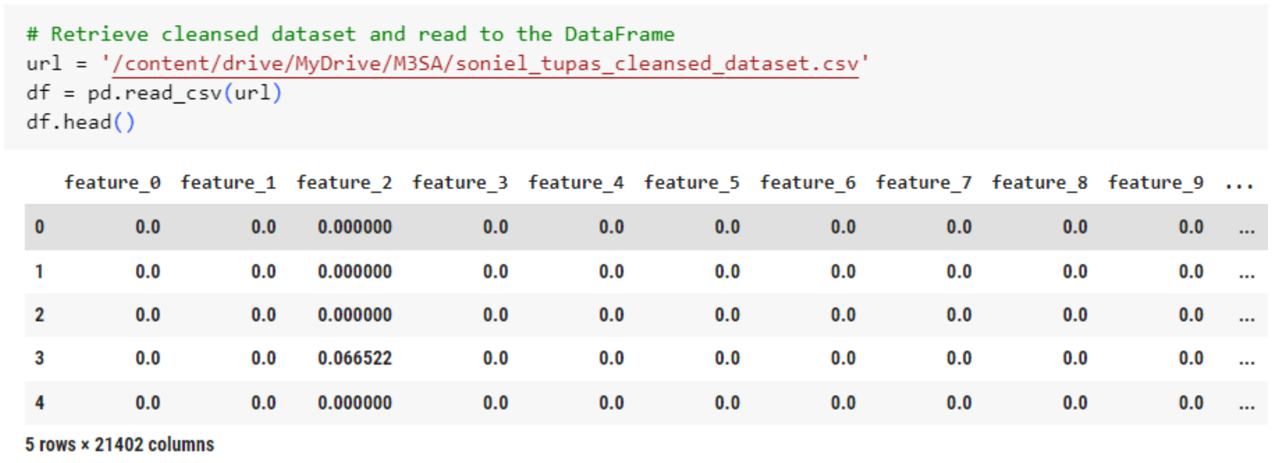
Google Drive was utilized as a repository for accessing the raw image files and where to store resulting data files. It was mounted using the mount() method.



```
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

The dataset was retrieved from the repository. It was read into the dataframe using the read_csv() method. The dataframe is shown.

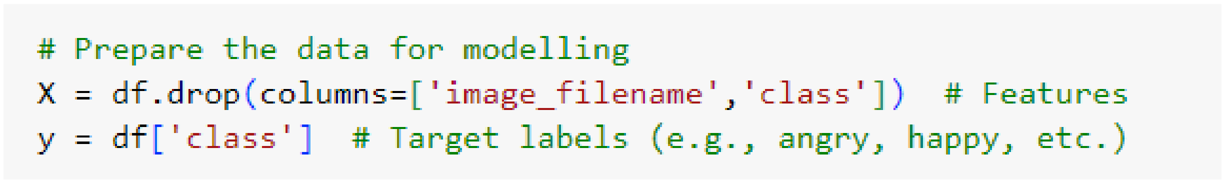


```
# Retrieve cleansed dataset and read to the DataFrame
url = '/content/drive/MyDrive/M3SA/soniel_tupas_cleansed_dataset.csv'
df = pd.read_csv(url)
df.head()
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
1	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
2	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
3	0.0	0.0	0.066522	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
4	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

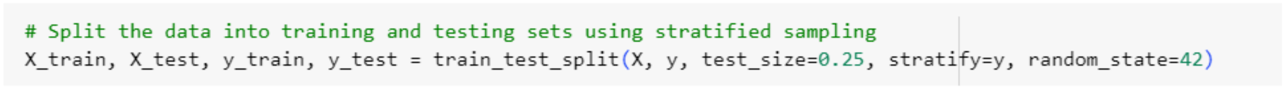
5 rows × 21402 columns

The data was segmented for the modelling preparation. The feature values were stored in the variable X. The emotions that were represented by class labels were stored in y.



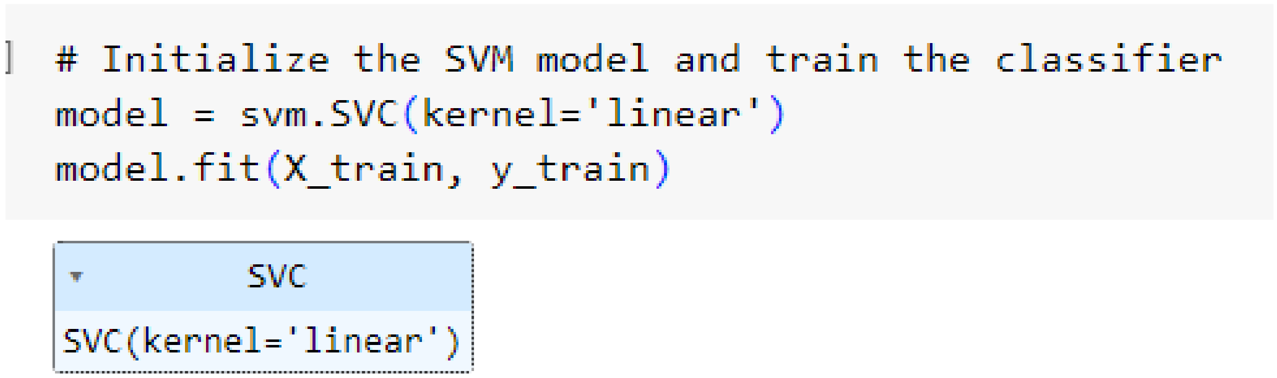
```
# Prepare the data for modelling
X = df.drop(columns=['image_filename','class'])  # Features
y = df['class']  # Target labels (e.g., angry, happy, etc.)
```

The data was split into training and testing sets using the train_test_split() method from sklearn. Stratified sampling was implemented. The splitting was done on a 75:25 ratio and a random state of 42.



```
# Split the data into training and testing sets using stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, random_state=42)
```

The SVM model was initialized using the svm() from sklearn where kernel is set as linear. The model was then fit with the training set to train the model in predicting the class labels of the given feature values.



```
# Initialize the SVM model and train the classifier
model = svm.SVC(kernel='linear')
model.fit(X_train, y_train)
```

```
SVC
SVC(kernel='linear')
```

After training, predictions in the test set were made using the model. This was implemented using the predict() method. The results were then stored in y_pred.

```
# Use the trained classifier to make predictions on the test set
y_pred = model.predict(X_test)
```

The accuracy of the model was calculated using the accuracy_score() method. After that, a classification report was shown to see precision, recall, f1-score and support for each class. This is used to measure the quality of predictions.

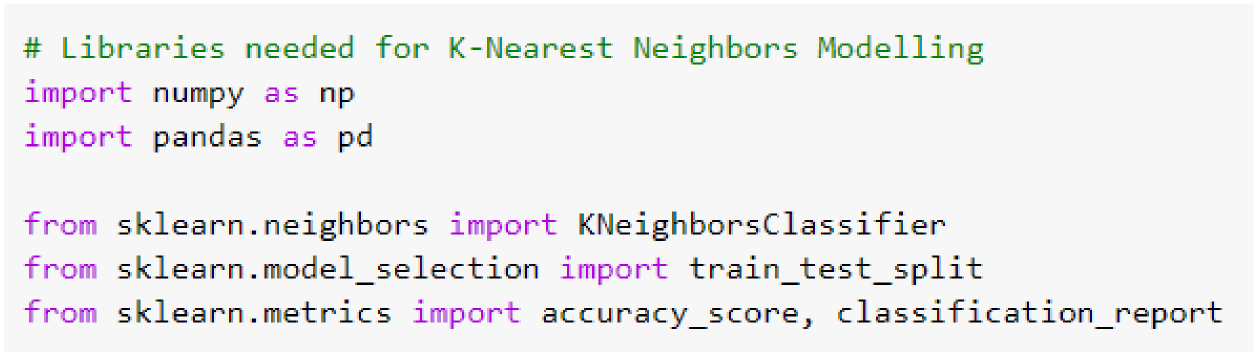
```
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Print a classification report to see precision, recall, f1-score, etc. for each class
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.3068181818181818
              precision    recall  f1-score   support

           0       0.12      0.17      0.14        12
           1       0.38      0.50      0.43        12
           2       0.25      0.23      0.24        13
           3       0.33      0.31      0.32        13
           4       0.67      0.31      0.42        13
           5       0.27      0.33      0.30        12
           6       0.36      0.31      0.33        13

    accuracy                           0.31        88
   macro avg       0.34      0.31      0.31        88
weighted avg       0.34      0.31      0.31        88
```

The information obtained from the classification report provides valuable insights into the predictive capabilities of the model. The accuracy of the model is found to be 30.68%, indicating that it correctly predicted the target class for only a relatively small portion of instances within the dataset. This performance can be considered poor. Examining the precision scores, it is evident that the model achieved its highest precision of 67% for label 4, while the lowest precision of 12% was observed for label 0. Similarly, the recall scores highlight the model's highest value of 50% for label 1, with the lowest being 17% for label 0. Additionally, the f1-scores, which combine precision and recall, indicate the highest score of 43% for label 1 and the lowest score of 14% for label 0. Furthermore, when looking at the support values, it is noticeable that labels 2, 3, 4, and 6 received the highest support of 13 instances each, whereas labels 0, 1, and 5 had the lowest support with 12 instances each. Although there are precision and recall scores above 50% for two labels, these results do not significantly alleviate the overall poor performance of the model, as evident from the low accuracy. This suggests that the model may benefit from more extensive training data to enhance its performance. Additionally, the current dataset may lack sufficient information to enable the model to identify meaningful patterns accurately. In light of these findings, adjustments to both the model and the dataset are necessary to improve the overall performance. Further data augmentation or collection and fine-tuning of the model's parameters might be valuable steps towards achieving more reliable predictions. Additionally, exploring alternative data sources or features that can provide more discriminatory information to the model may lead to better results.

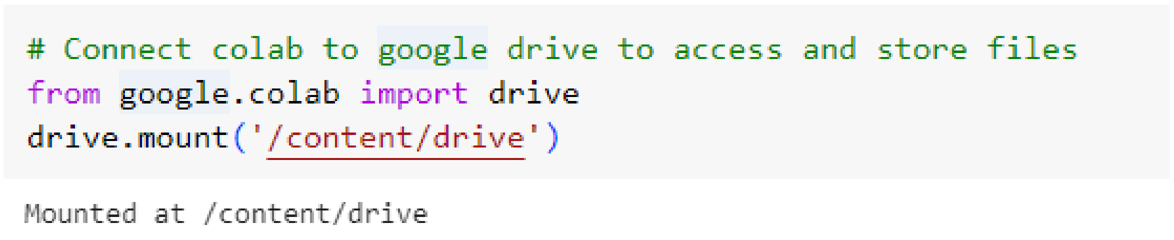
K-Nearest Neighbors

Libraries that are needed to implement K-Nearest Neighbors (KNN) Modelling were imported. The libraries used were numpy, pandas, and sklearn. Specifically, in sklearn, KNeighborsClassifier, train_test_split, accuracy_score and classification_report.



```python
# Libraries needed for K-Nearest Neighbors Modelling
import numpy as np
import pandas as pd

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

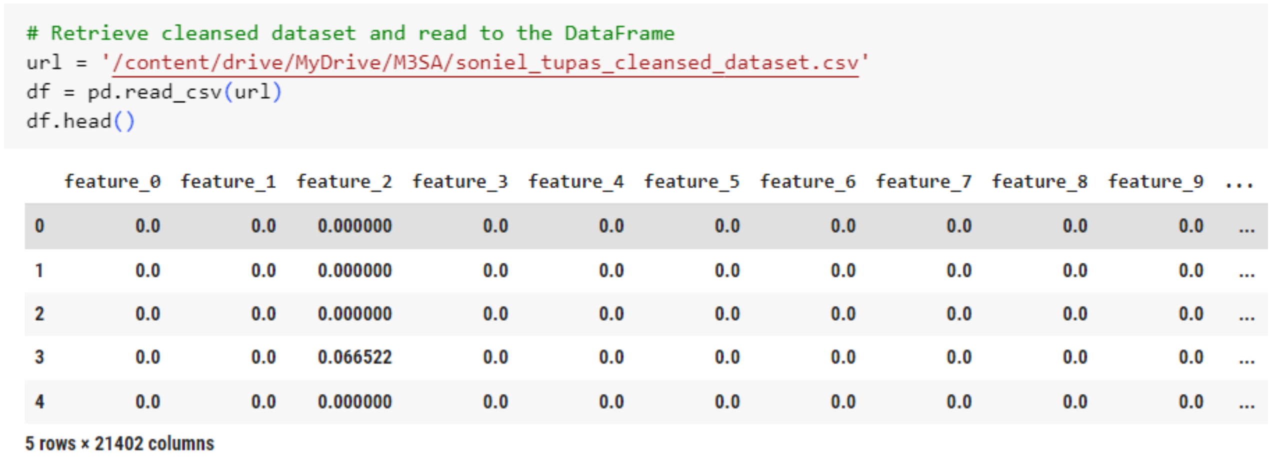
Google Drive was utilized as a repository for accessing the raw image files and where to store resulting data files. It was mounted using the mount() method.



```python
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

The dataset was retrieved from the repository. It was read into the dataframe using the read_csv() method. The dataframe is shown.

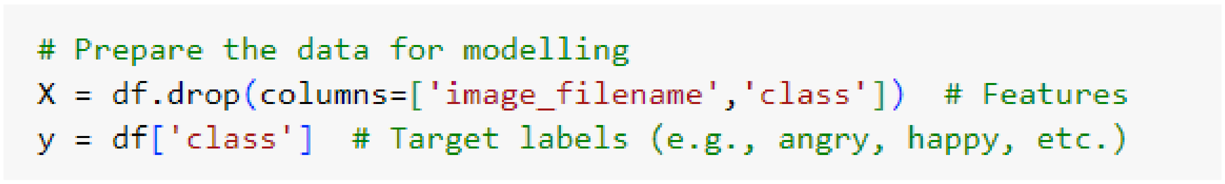


```python
# Retrieve cleansed dataset and read to the DataFrame
url = '/content/drive/MyDrive/M3SA/soniel_tupas_cleansed_dataset.csv'
df = pd.read_csv(url)
df.head()
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
1	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
2	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
3	0.0	0.0	0.066522	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
4	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

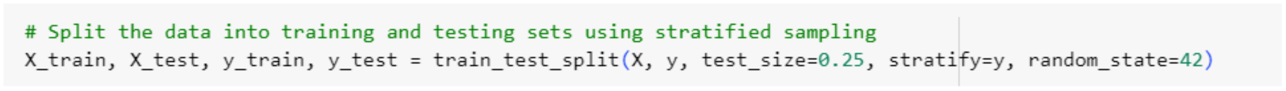
5 rows × 21402 columns

The data was segmented for the modelling preparation. The feature values were stored in the variable X. The emotions that were represented by class labels were stored in y.



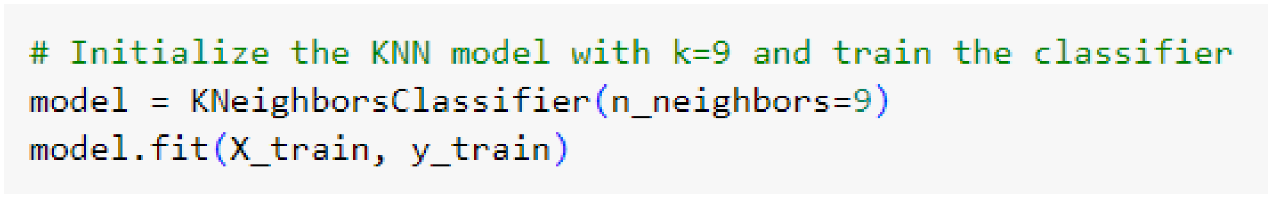
```python
# Prepare the data for modelling
X = df.drop(columns=['image_filename','class'])  # Features
y = df['class']  # Target labels (e.g., angry, happy, etc.)
```

The data was split into training and testing sets using the train_test_split() method from sklearn. Stratified sampling was implemented. The splitting was done on a 75:25 ratio and a random state of 42.

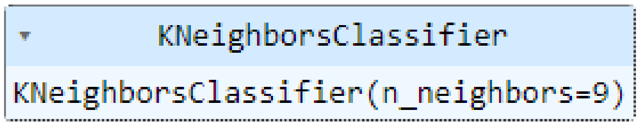


```python
# Split the data into training and testing sets using stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, random_state=42)
```

The KNN model was initialized using the KNeighborsClassifier() from sklearn where the value of n_neighbors is 9. The model was then fit with the training set to train the model in predicting the class labels of the given feature values.

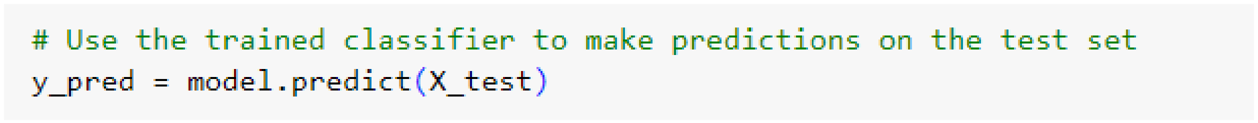


```python
# Initialize the KNN model with k=9 and train the classifier
model = KNeighborsClassifier(n_neighbors=9)
model.fit(X_train, y_train)
```



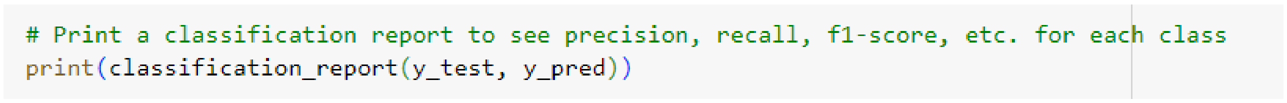
```
▾      KNeighborsClassifier
KNeighborsClassifier(n_neighbors=9)
```

After training, predictions in the test set were made using the model. This was implemented using the predict() method. The results were then stored in y_pred.

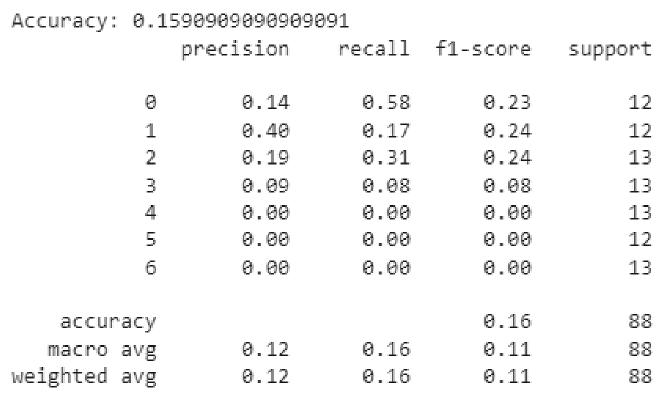


```python
# Use the trained classifier to make predictions on the test set
y_pred = model.predict(X_test)
```

The accuracy of the model was calculated using the accuracy_score() method. After that, a classification report was shown to see precision, recall, f1-score and support for each class. This is used to measure the quality of predictions.



```python
# Print a classification report to see precision, recall, f1-score, etc. for each class
print(classification_report(y_test, y_pred))
```

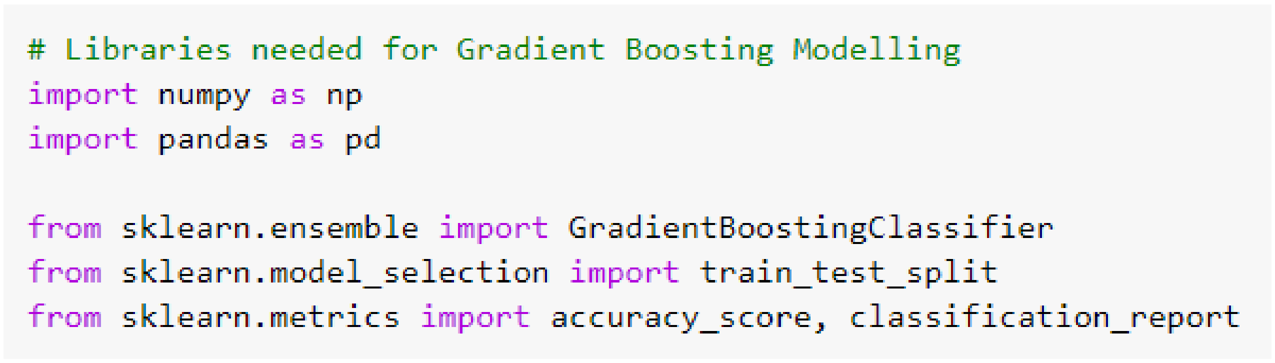


```
Accuracy: 0.1590909090909091
              precision    recall  f1-score   support

           0       0.14      0.58      0.23        12
           1       0.40      0.17      0.24        12
           2       0.19      0.31      0.24        13
           3       0.09      0.08      0.08        13
           4       0.00      0.00      0.00        13
           5       0.00      0.00      0.00        12
           6       0.00      0.00      0.00        13

    accuracy                           0.16        88
   macro avg       0.12      0.16      0.11        88
weighted avg       0.12      0.16      0.11        88
```

The data presented in the classification report serves as an indicator of the model's predictive efficacy. With an accuracy of 15.91%, the model correctly predicted the target class for only a small fraction of instances in the dataset, suggesting a subpar performance. Examining the precision scores, we find that the highest value attained by the model is 40% for label 1, while the lowest precision scores of 0% were observed for labels 4, 5, and 6. This indicates a considerable disparity in the model's ability to make accurate positive predictions for different classes. Furthermore, analyzing the recall scores, we observe that the model achieved the highest recall of 58% for label 0, while labels 4, 5, and 6 received a recall score of 0%. This signifies that the model failed to effectively capture instances belonging to these classes. The f1-scores, which encompass both precision and recall, demonstrate the model's highest performance of 24% for labels 1 and 2, while labels 4, 5, and 6 exhibited an f1-score of 0%, further accentuating the model's limitations in distinguishing these classes. Additionally, the support values reveal that the model had the highest number of instances (13) for labels 2, 3, 4, and 6, while the lowest support (12 instances) was observed for labels 0, 1, and 5. Taken together, these results indicate a consistent and poor performance across all evaluated aspects of the model. This can be attributed to various factors, such as the model's insufficient exposure to training data, limiting its ability to generalize effectively. Furthermore, the dataset employed may lack crucial information necessary for the model to discern meaningful patterns accurately. To address these shortcomings and enhance the overall performance, adjustments to both the model and the dataset are imperative. This may involve augmenting the training data to provide the model with a more comprehensive learning experience. Additionally, fine-tuning the model's parameters and architecture could lead to improved predictions. Moreover, exploring alternative data sources or features that can offer more discriminatory information may contribute to better outcomes. In light of these considerations, diligent efforts are required to enhance the model's capabilities and elevate its predictive prowess.

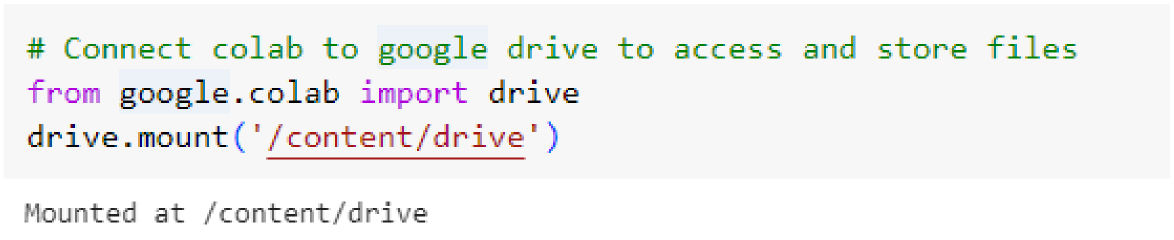
Gradient Boosting

Libraries that are needed to implement Gradient Boosting Modelling were imported. The libraries used were numpy, pandas, and sklearn. Specifically, in sklearn, GradientBoostingClassifier, train_test_split, accuracy_score and classification_report.



```
# Libraries needed for Gradient Boosting Modelling
import numpy as np
import pandas as pd

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

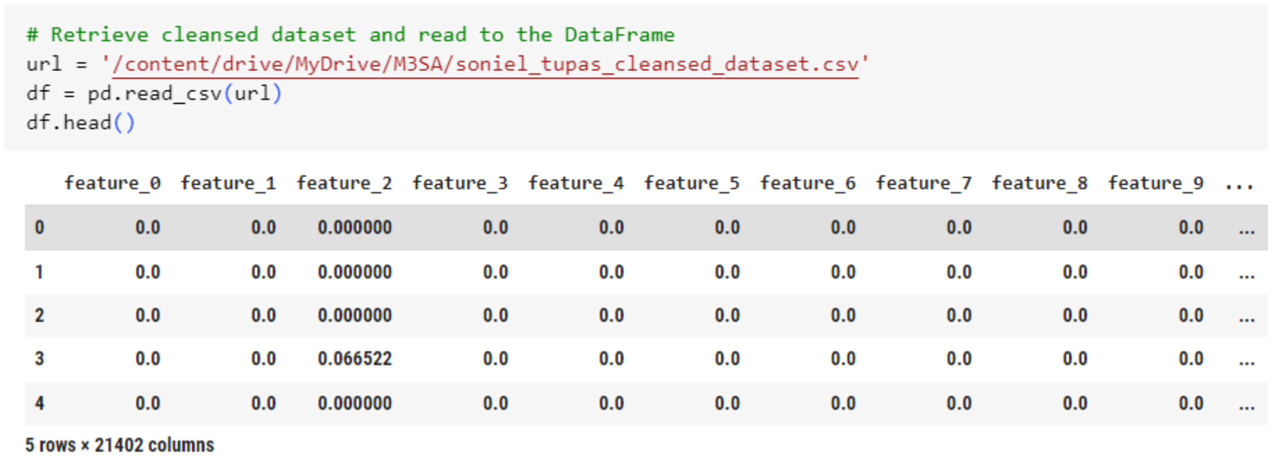
Google Drive was utilized as a repository for accessing the raw image files and where to store resulting data files. It was mounted using the mount() method.



```
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

The dataset was retrieved from the repository. It was read into the dataframe using the read_csv() method. The dataframe is shown.

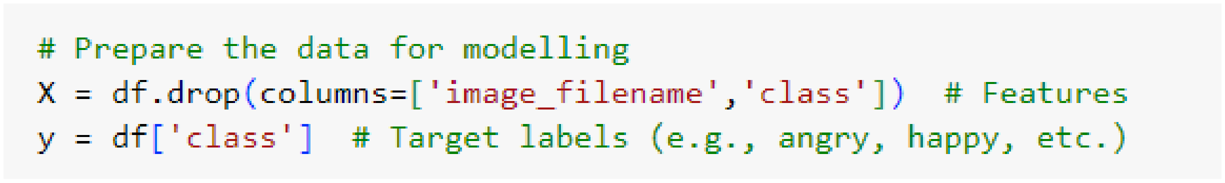


```
# Retrieve cleansed dataset and read to the DataFrame
url = '/content/drive/MyDrive/M3SA/soniel_tupas_cleansed_dataset.csv'
df = pd.read_csv(url)
df.head()
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
1	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
2	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
3	0.0	0.0	0.066522	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
4	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

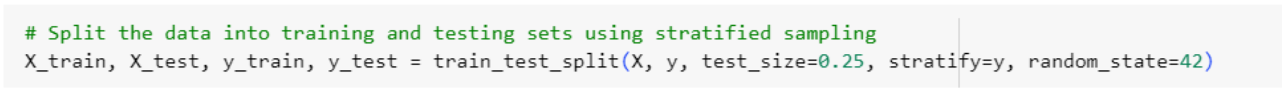
5 rows × 21402 columns

The data was segmented for the modelling preparation. The feature values were stored in the variable X. The emotions that were represented by class labels were stored in y.



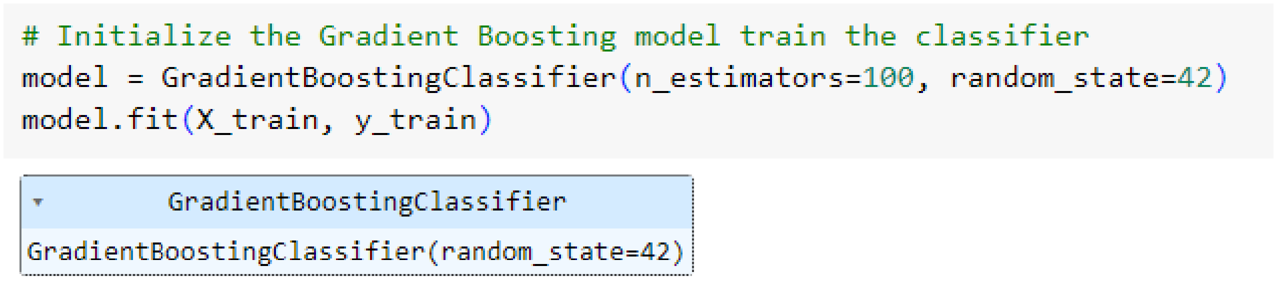
```
# Prepare the data for modelling
X = df.drop(columns=['image_filename','class'])  # Features
y = df['class']  # Target labels (e.g., angry, happy, etc.)
```

The data was split into training and testing sets using the train_test_split() method from sklearn. Stratified sampling was implemented. The splitting was done on a 75:25 ratio and a random state of 42.



```
# Split the data into training and testing sets using stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, random_state=42)
```

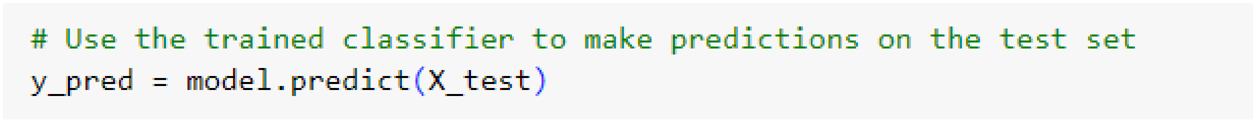
The Gradient Boosting model was initialized using the GradientBoostingClassifier() from sklearn where the value of n_estimators is 100. The model was then fit with the training set to train the model in predicting the class labels of the given feature values.



```
# Initialize the Gradient Boosting model train the classifier
model = GradientBoostingClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

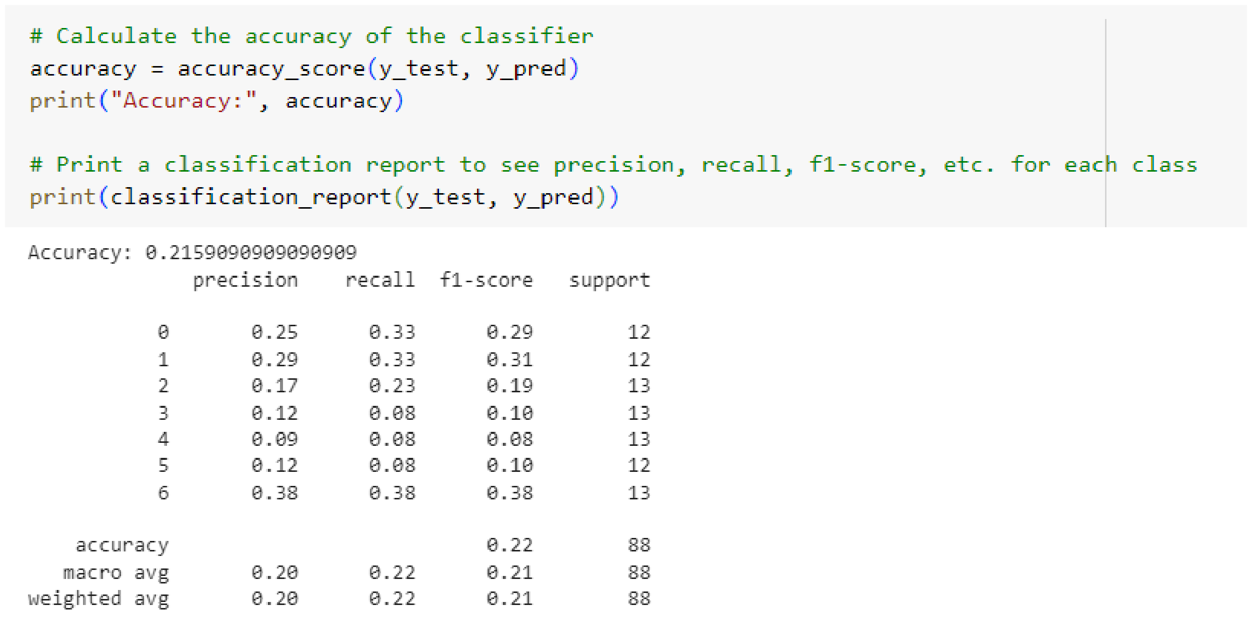
```
▾        GradientBoostingClassifier
GradientBoostingClassifier(random_state=42)
```

After training, predictions in the test set were made using the model. This was implemented using the predict() method. The results were then stored in y_pred.



```
# Use the trained classifier to make predictions on the test set
y_pred = model.predict(X_test)
```

The accuracy of the model was calculated using the accuracy_score() method. After that, a classification report was shown to see precision, recall, f1-score and support for each class. This is used to measure the quality of predictions.



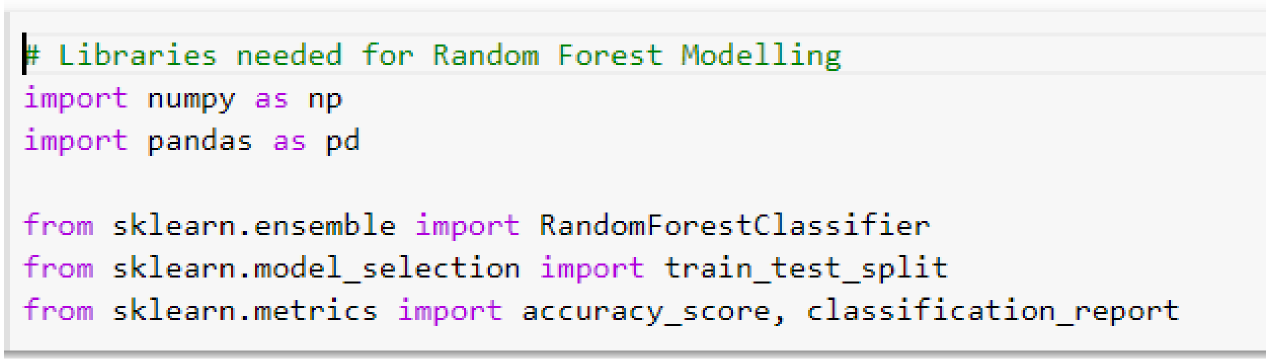
```
# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Print a classification report to see precision, recall, f1-score, etc. for each class
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.2159090909090909
              precision    recall  f1-score   support

           0       0.25      0.33      0.29        12
           1       0.29      0.33      0.31        12
           2       0.17      0.23      0.19        13
           3       0.12      0.08      0.10        13
           4       0.09      0.08      0.08        13
           5       0.12      0.08      0.10        12
           6       0.38      0.38      0.38        13

    accuracy                           0.22        88
   macro avg       0.20      0.22      0.21        88
weighted avg       0.20      0.22      0.21        88
```

With an accuracy of 25.59%, the model correctly predicted the target class for only a relatively small proportion, indicating a suboptimal performance. Analyzing the precision scores, the model achieved its highest value of 38% for label 6, while the lowest precision score of 9% was observed for label 4. This variation in precision suggests that the model's ability to make accurate positive predictions differs significantly across classes. In terms of recall scores, the model obtained the highest recall of 38% for label 6, with the lowest recall of 8% shared by labels 3, 4, and 5. This implies that the model had difficulty accurately identifying instances belonging to these specific classes. Furthermore, examining the f1-scores, which incorporate both precision and recall, the model achieved its highest performance of 38% for label 6, while label 4 displayed the lowest f1-score of 8%. This underscores the model's challenges in accurately classifying instances in label 4. Considering the support values, the model encountered the highest number of instances which was 13 for labels 2, 3, 4, and 6, while labels 0, 1, and 5 had the lowest support, with 12 instances each. Collectively, these results indicate a consistent and inadequate performance across all evaluated aspects of the model. This suggests that the model may require additional training data to enhance its predictive capability and improve generalization. Furthermore, the current dataset may lack crucial information necessary for the model to discern meaningful patterns accurately. To address these limitations and elevate overall performance, it is essential to make adjustments to both the model and the dataset. This could entail augmenting the training data to provide the model with a more comprehensive and diverse learning experience. Additionally, fine-tuning the model's parameters and architecture might lead to improved predictions. Exploring alternative data sources or features that can offer more discriminative information is also a potential avenue for enhancing performance. In conclusion, addressing the identified challenges through rigorous data and model improvements is crucial to bolster the model's effectiveness and predictive capacity in practical applications.

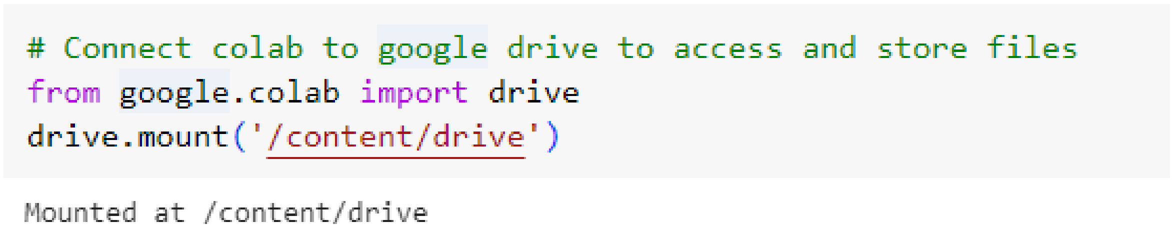
Random Forest

Libraries that are needed to implement Random Forest Modelling were imported. The libraries used were numpy, pandas, and sklearn. Specifically, in sklearn, RandomForestClassifier, train_test_split, accuracy_score and classification_report.



```
# Libraries needed for Random Forest Modelling
import numpy as np
import pandas as pd

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

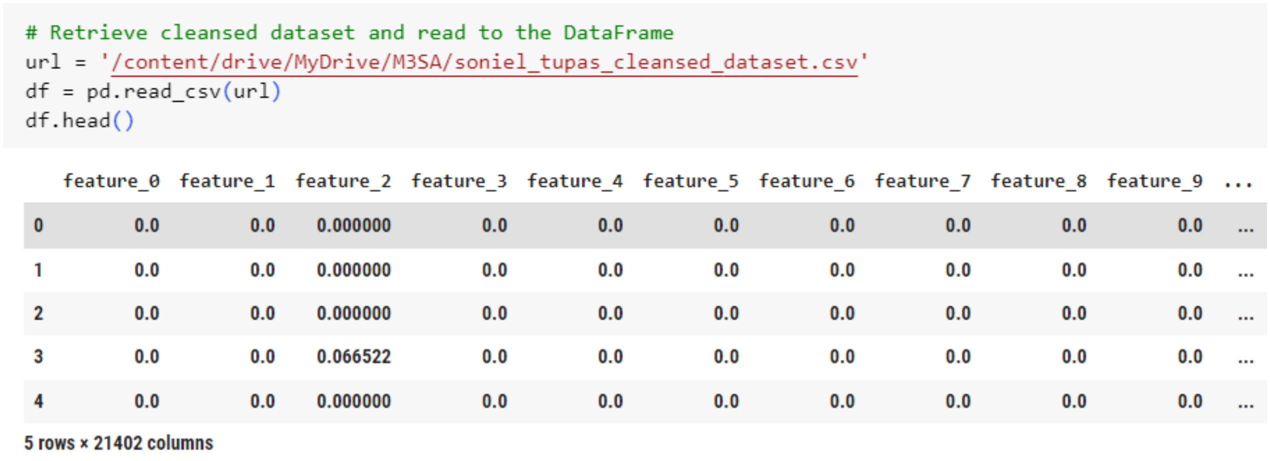
Google Drive was utilized as a repository for accessing the raw image files and where to store resulting data files. It was mounted using the mount() method.



```
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

The dataset was retrieved from the repository. It was read into the dataframe using the read_csv() method. The dataframe is shown.



```
# Retrieve cleansed dataset and read to the DataFrame
url = '/content/drive/MyDrive/M3SA/soniel_tupas_cleansed_dataset.csv'
df = pd.read_csv(url)
df.head()
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
1	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
2	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
3	0.0	0.0	0.066522	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
4	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

5 rows × 21402 columns

The data was segmented for the modelling preparation. The feature values were stored in the variable X. The emotions that were represented by class labels were stored in y.

```
# Prepare the data for modelling
X = df.drop(columns=['image_filename','class'])  # Features
y = df['class']  # Target labels (e.g., angry, happy, etc.)
```

The data was split into training and testing sets using the train_test_split() method from sklearn. Stratified sampling was implemented. The splitting was done on a 75:25 ratio and a random state of 42.

```
# Split the data into training and testing sets using stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, random_state=42)
```

The Random Forest model was initialized using the RandomForestClassifier() from sklearn. The model was then fit with the training set to train the model in predicting the class labels of the given feature values.

```
# Initialize the Random Forest model and train the classifier
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

```
▾        RandomForestClassifier
RandomForestClassifier(random_state=42)
```

After training, predictions in the test set were made using the model. This was implemented using the predict() method. The results were then stored in y_pred.

```
# Use the trained classifier to make predictions on the test set
y_pred = model.predict(X_test)
```

The accuracy of the model was calculated using the accuracy_score() method. After that, a classification report was shown to see precision, recall, f1-score and support for each class. This is used to measure the quality of predictions.

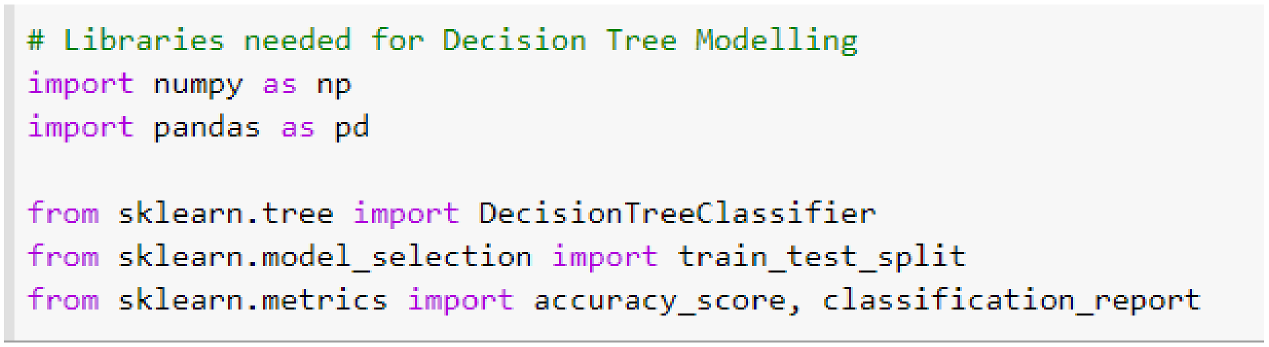
```
# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Print a classification report to see precision, recall, f1-score, etc. for each class
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.25
              precision    recall  f1-score   support

           0       0.25      0.33      0.29        12
           1       0.17      0.17      0.17        12
           2       0.00      0.00      0.00        13
           3       0.33      0.38      0.36        13
           4       0.50      0.31      0.38        13
           5       0.21      0.25      0.23        12
           6       0.27      0.31      0.29        13

    accuracy                           0.25        88
   macro avg       0.25      0.25      0.24        88
weighted avg       0.25      0.25      0.24        88
```

The classification report data provides a comprehensive evaluation of the model's predictive capabilities. With an accuracy of 25%, the model's correct predictions for the target class amounted to only 25% of the instances in the dataset, signifying a notably low performance. Delving into precision scores, the model achieved its highest precision of 50% for label 4, while label 2 received the lowest precision score of 0%. This considerable variation in precision indicates that the model's ability to make accurate positive predictions varies significantly across different classes. Regarding recall scores, the model obtained the highest recall of 38% for label 3, whereas label 2 exhibited the lowest recall score of 0%. This highlights the model's difficulty in accurately capturing instances belonging to label 2. Analyzing the f1-scores, which encompass both precision and recall, the model attained its highest performance of 38% for label 4, while label 2 displayed the lowest f1-score of 0%. This further emphasizes the model's limitations in effectively classifying instances in label 2. Examining the support values, the model encountered the highest number of instances (13) for labels 2, 3, 4, and 6, whereas labels 0, 1, and 5 had the lowest support, each with 12 instances. The consistent underperformance across all evaluated aspects of the model leads to the interpretation that the model's performance was unsatisfactory. To address this, several measures can be taken. Firstly, augmenting the training data might enhance the model's ability to generalize and make more accurate predictions. Secondly, fine-tuning the model's parameters and architecture could potentially yield improvements in performance. Lastly, exploring alternative data sources or feature engineering techniques may provide the model with more informative inputs, leading to better pattern recognition. In conclusion, it is evident from the analysis that significant improvements are necessary to enhance the model's overall performance. By employing the aforementioned adjustments and refining the model's approach, it is possible to achieve more reliable and effective predictions in real-world scenarios.

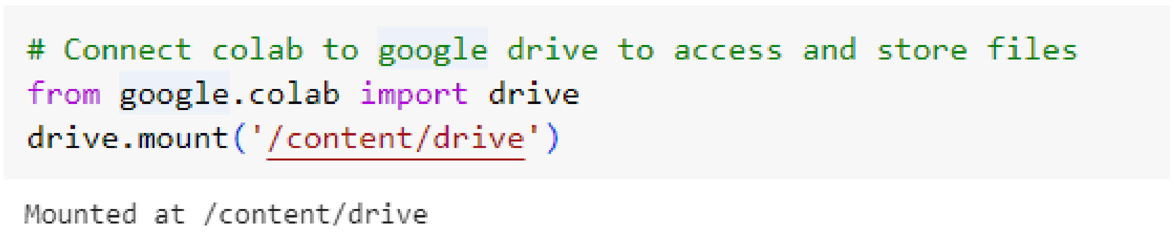
Decision Trees

Libraries that are needed to implement Naïve Bayes Modelling were imported. The libraries used were numpy, pandas, and sklearn. Specifically, in sklearn, DecisionTreeClassifier, train_test_split, accuracy_score and classification_report.



```
# Libraries needed for Decision Tree Modelling
import numpy as np
import pandas as pd

from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

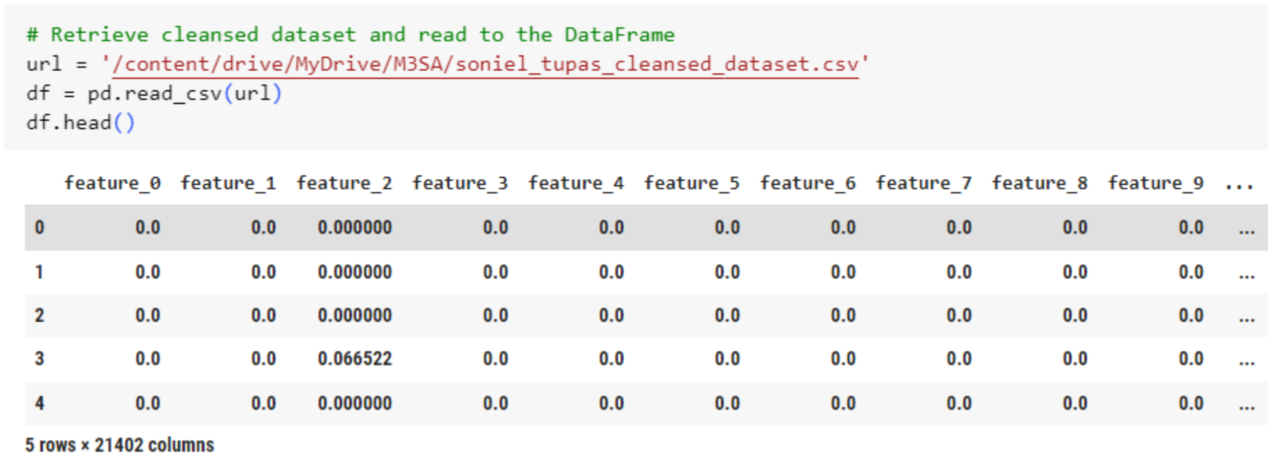
Google Drive was utilized as a repository for accessing the raw image files and where to store resulting data files. It was mounted using the mount() method.



```
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

The dataset was retrieved from the repository. It was read into the dataframe using the read_csv() method. The dataframe is shown.

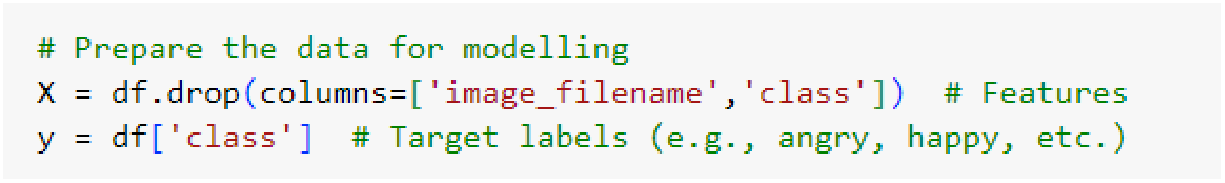


```
# Retrieve cleansed dataset and read to the DataFrame
url = '/content/drive/MyDrive/M3SA/soniel_tupas_cleansed_dataset.csv'
df = pd.read_csv(url)
df.head()
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	...
0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
1	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
2	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
3	0.0	0.0	0.066522	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
4	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

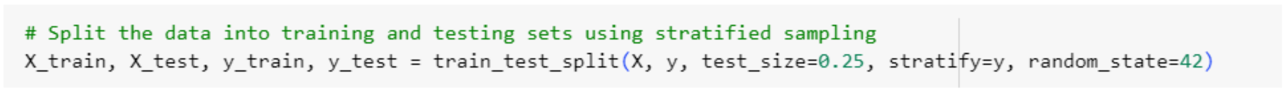
5 rows × 21402 columns

The data was segmented for the modelling preparation. The feature values were stored in the variable X. The emotions that were represented by class labels were stored in y.



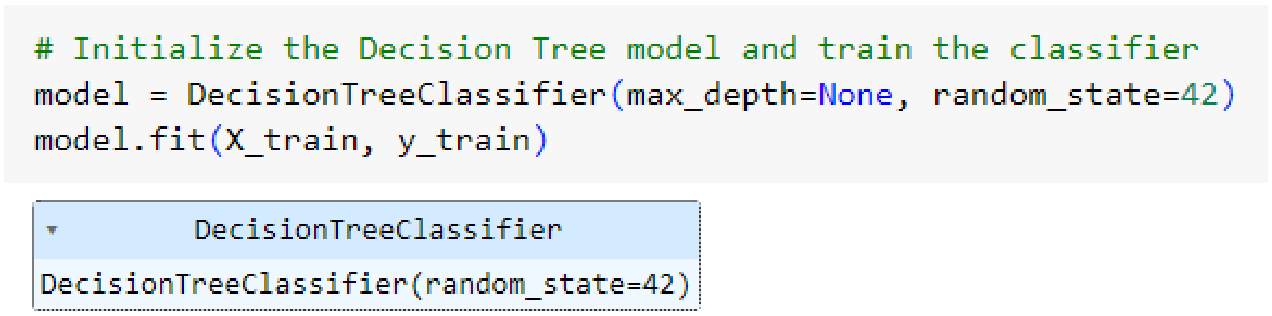
```
# Prepare the data for modelling
X = df.drop(columns=['image_filename','class'])  # Features
y = df['class']  # Target labels (e.g., angry, happy, etc.)
```

The data was split into training and testing sets using the train_test_split() method from sklearn. Stratified sampling was implemented. The splitting was done on a 75:25 ratio and a random state of 42.



```
# Split the data into training and testing sets using stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, random_state=42)
```

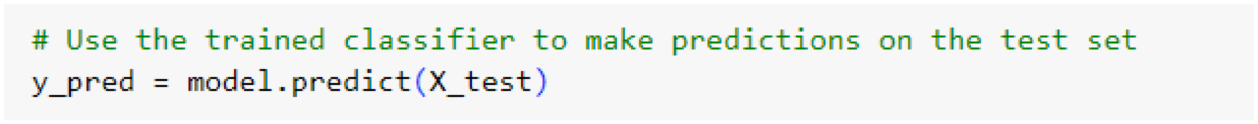
The Decision Tree model was initialized using the DecisionTreeClassifier() from sklearn. The model was then fit with the training set to train the model in predicting the class labels of the given feature values.



```
# Initialize the Decision Tree model and train the classifier
model = DecisionTreeClassifier(max_depth=None, random_state=42)
model.fit(X_train, y_train)
```

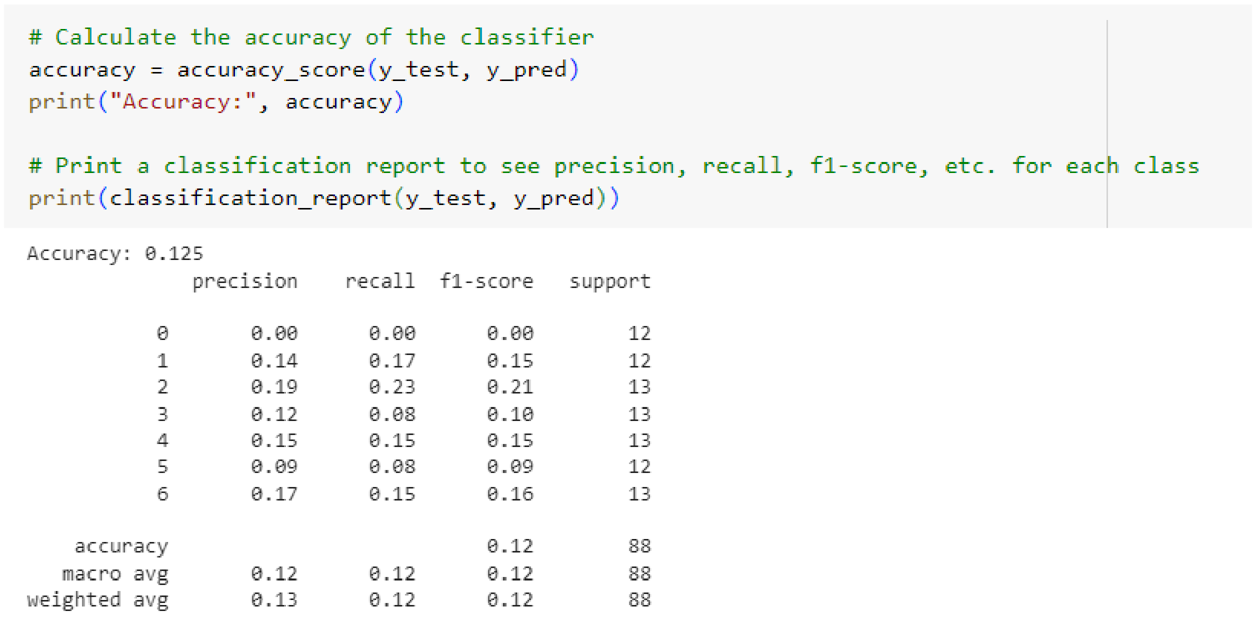
```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

After training, predictions in the test set were made using the model. This was implemented using the predict() method. The results were then stored in y_pred.



```
# Use the trained classifier to make predictions on the test set
y_pred = model.predict(X_test)
```

The accuracy of the model was calculated using the accuracy_score() method. After that, a classification report was shown to see precision, recall, f1-score and support for each class. This is used to measure the quality of predictions.



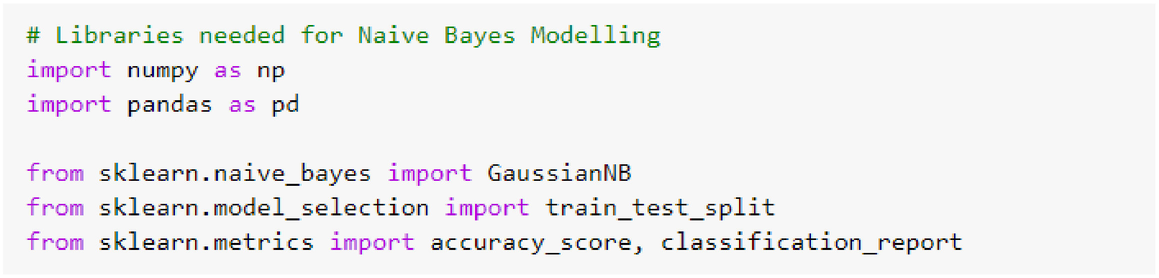
```python
# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Print a classification report to see precision, recall, f1-score, etc. for each class
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.125
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        12
           1       0.14      0.17      0.15        12
           2       0.19      0.23      0.21        13
           3       0.12      0.08      0.10        13
           4       0.15      0.15      0.15        13
           5       0.09      0.08      0.09        12
           6       0.17      0.15      0.16        13

    accuracy                           0.12        88
   macro avg       0.12      0.12      0.12        88
weighted avg       0.13      0.12      0.12        88
```

The classification report data serves as a comprehensive evaluation of the model's prediction quality. With an accuracy of 12.5%, the model correctly predicted the target class for only 12.5% of the instances in the dataset, indicating a significantly low level of performance. Analyzing precision scores, the model achieved its highest precision of 19% for label 2, while label 0 received the lowest precision score of 0%. This substantial variation in precision highlights the model's difficulty in making accurate positive predictions for certain classes. Regarding recall scores, the model attained the highest recall of 23% for label 2, while label 0 displayed the lowest recall score of 0%. This suggests that the model struggled to effectively capture instances belonging to label 0. Examining the f1-scores, which combine precision and recall, the model's highest performance was observed with an f1-score of 0.21% for label 1, while label 0 exhibited the lowest f1-score of 0%. This further emphasizes the model's limitations in correctly classifying instances in label 0. Furthermore, the support values indicate that the model encountered the highest number of instances (13) for labels 2, 3, 4, and 6, whereas labels 0, 1, and 5 had the lowest support, each with 12 instances. The consistent poor performance across all evaluated aspects of the model leads to the interpretation that the model's predictions were unsatisfactory. To address this, several measures can be taken. Firstly, increasing the amount of training data might improve the model's ability to generalize and make more accurate predictions. Secondly, fine-tuning the model's parameters and architecture could potentially lead to better performance. Lastly, exploring alternative data sources or conducting feature engineering may provide the model with more informative inputs, leading to better pattern recognition. In conclusion, the analysis clearly indicates the need for significant improvements to enhance the model's overall performance. By implementing the suggested adjustments and refining the model's approach, it is possible to achieve more reliable and effective predictions in real-world scenarios.

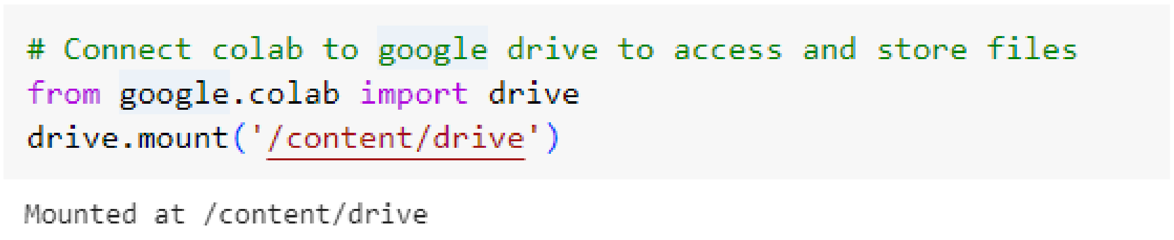
Naive Bayes

Libraries that are needed to implement Naïve Bayes Modelling were imported. The libraries used were numpy, pandas, and sklearn. Specifically, in sklearn, GaussianNB, train_test_split, accuracy_score and classification_report.



```
# Libraries needed for Naive Bayes Modelling
import numpy as np
import pandas as pd

from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

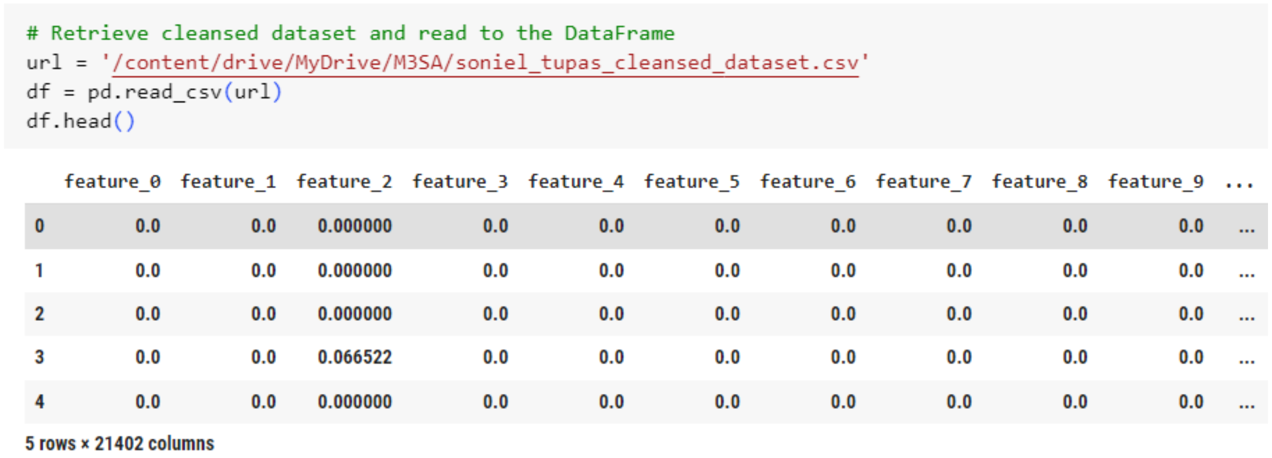
Google Drive was utilized as a repository for accessing the raw image files and where to store resulting data files. It was mounted using the mount() method.



```
# Connect colab to google drive to access and store files
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

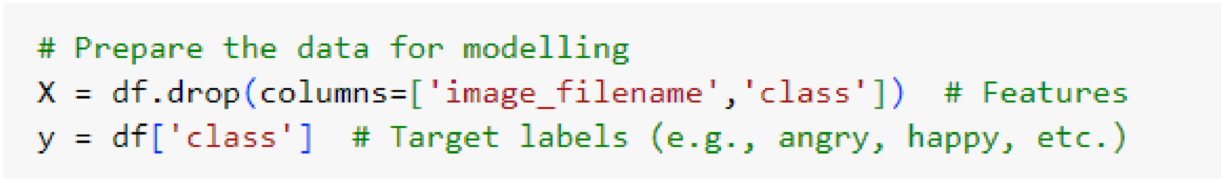
The dataset was retrieved from the repository. It was read into the dataframe using the read_csv() method. The dataframe is shown.



```
# Retrieve cleansed dataset and read to the DataFrame
url = '/content/drive/MyDrive/M3SA/soniel_tupas_cleansed_dataset.csv'
df = pd.read_csv(url)
df.head()
```

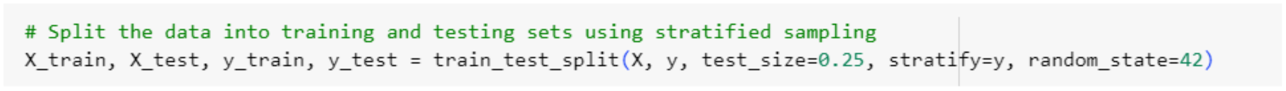
```
   feature_0  feature_1  feature_2  feature_3  feature_4  feature_5  feature_6  feature_7  feature_8  feature_9  ...
0        0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
1        0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
2        0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
3        0.0        0.0   0.066522        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
4        0.0        0.0   0.000000        0.0        0.0        0.0        0.0        0.0        0.0        0.0  ...
5 rows × 21402 columns
```

The data was segmented for the modelling preparation. The feature values were stored in the variable X. The emotions that were represented by class labels were stored in y.



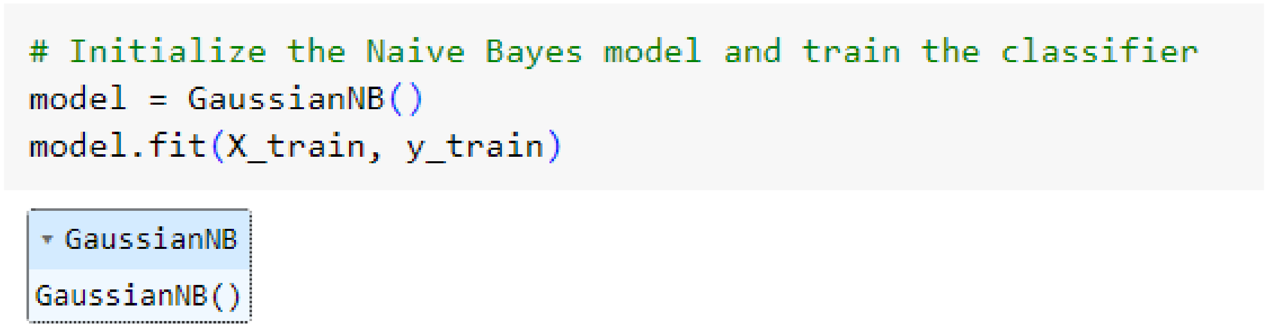
```
# Prepare the data for modelling
X = df.drop(columns=['image_filename','class'])  # Features
y = df['class']  # Target labels (e.g., angry, happy, etc.)
```

The data was split into training and testing sets using the train_test_split() method from sklearn. Stratified sampling was implemented also as what is required. The splitting was done on a 75:25 ratio and a random state of 42.



```
# Split the data into training and testing sets using stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, random_state=42)
```

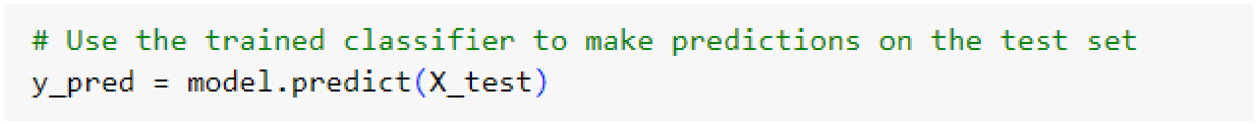
The Naïve Bayes model was initialized using the GaussianNB() from sklearn. The model was then fit with the training set to train the model in predicting the class labels of the given feature values.



```
# Initialize the Naive Bayes model and train the classifier
model = GaussianNB()
model.fit(X_train, y_train)
```

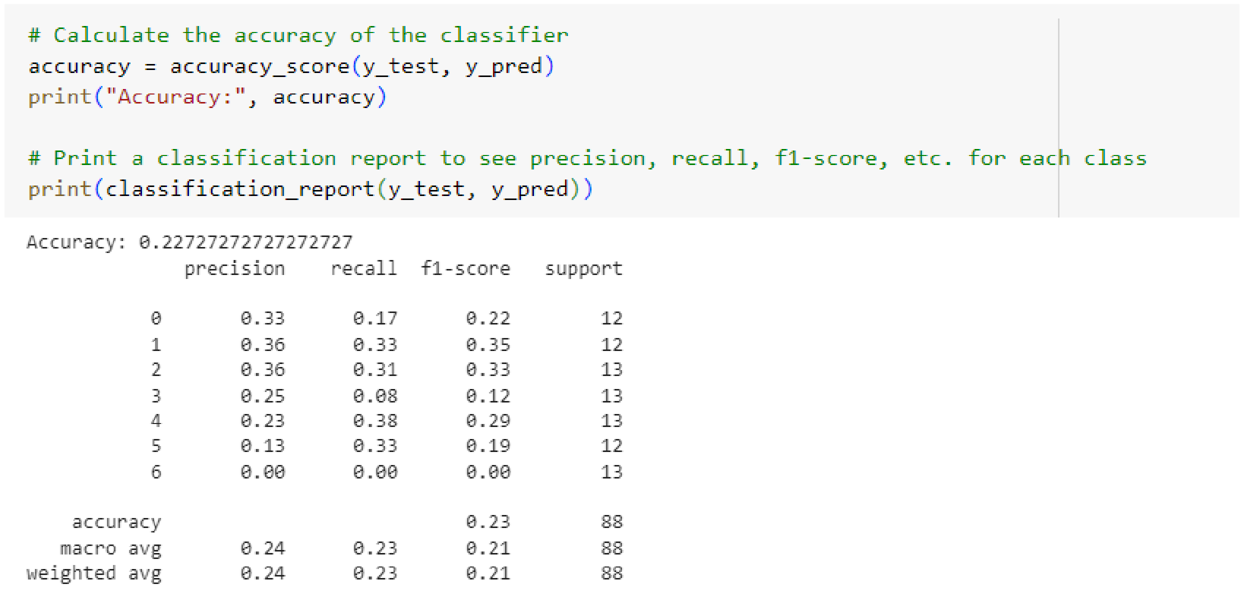
```
▾ GaussianNB
GaussianNB()
```

After training, predictions in the test set were made using the model. This was implemented using the predict() method. The results were then stored in y_pred.



```
# Use the trained classifier to make predictions on the test set
y_pred = model.predict(X_test)
```

The accuracy of the model was calculated using the accuracy_score() method. After that, a classification report was shown to see precision, recall, f1-score and support for each class. This is used to measure the quality of predictions.



```
# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Print a classification report to see precision, recall, f1-score, etc. for each class
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.22727272727272727
              precision    recall  f1-score   support

           0       0.33      0.17      0.22        12
           1       0.36      0.33      0.35        12
           2       0.36      0.31      0.33        13
           3       0.25      0.08      0.12        13
           4       0.23      0.38      0.29        13
           5       0.13      0.33      0.19        12
           6       0.00      0.00      0.00        13

    accuracy                           0.23        88
   macro avg       0.24      0.23      0.21        88
weighted avg       0.24      0.23      0.21        88
```

The data presented in the classification report provides insights into the model's predictive quality. With an accuracy of 23%, the model correctly predicted the target class for only 23% of the instances in the dataset, signifying a relatively low level of performance. Consequently, it can be inferred that the model's overall performance was unsatisfactory. Delving into precision scores, the model achieved its highest precision of 36% for labels 1 and 2, while label 6 received the lowest precision score of 0%. This variation in precision underscores the model's difficulty in making accurate positive predictions for label 6 and the varying degrees of accuracy for other classes. Moving on to recall scores, the model obtained the highest recall of 33% for labels 1 and 5, whereas label 6 exhibited the lowest recall score of 0%. This indicates the model's limitations in correctly capturing instances belonging to label 6 and the varying success rates for other classes. Analyzing the f1-scores, which combine precision and recall, the model attained its highest performance of 35% for label 1, while label 6 displayed the lowest f1-score of 0%. This further emphasizes the challenges faced by the model in effectively classifying instances in label 6. Furthermore, considering the support values, the model encountered the highest number of instances (13) for labels 2, 3, 4, and 6, while labels 0, 1, and 5 had the lowest support, each with 12 instances. The consistent underperformance across all evaluated aspects of the model leads to the interpretation that the model's predictions were inadequate. To address this, several measures can be taken. Firstly, augmenting the training data might provide the model with a more diverse and comprehensive learning experience, potentially leading to improved performance. Secondly, fine-tuning the model's parameters and architecture could help enhance its predictive capacity. Additionally, exploring alternative data sources or conducting feature engineering may provide the model with more informative inputs, thereby improving its ability to identify patterns accurately. In conclusion, it is evident from the analysis that substantial improvements are necessary to elevate the model's overall performance. By implementing the suggested adjustments and refining the model's approach, it is possible to achieve more reliable and effective predictions, thus enhancing the model's practical utility.

GOOGLE COLAB NOTEBOOK LINKS

Data Collection

https://colab.research.google.com/drive/1D1yHfZ6HjVqcjgWuj6f_qw01qLviMFbi?usp=sharing

Data Pre-processing

https://colab.research.google.com/drive/1U-sZb2Q4kxOl0tvHNarTRlfdWX7YtJCN?usp=sharing

Visualization

https://colab.research.google.com/drive/19dROt6DMOI0HzHPepDyu3lBcEeKq3pHD?usp=sharing

Logistic Regression

https://colab.research.google.com/drive/1QF4T2sWkrfdVdZsRiapOBbc4Qjs46ae-?usp=sharing

SVM

https://colab.research.google.com/drive/1M3B76eaxHe6FRS9zK1wktAxfgmQsMC2N?usp=sharing

KNN

https://colab.research.google.com/drive/1Ljy9TAKptbdHRcFttlYeAQbzB0iYlT5n?usp=sharing

Random Forest

https://colab.research.google.com/drive/1x0e19T5NYWrd7zD7V2oRf_sMf8_M-1Ql?usp=sharing

Gradient Boosting

https://colab.research.google.com/drive/1pmhRID6zbytzW0oD51fTKe6ylFDpAtSM?usp=sharing

Decision Tree

https://colab.research.google.com/drive/1l-fA1p7WPuuTMmg0OaxGkMj1uKFJTRzy?usp=sharing

Naïve Bayes

https://colab.research.google.com/drive/1JiJCEFfDEhMuK2gXSt8tU2ZlGQqOvtEj?usp=sharing