SENTIMENT ANALYSIS PROJECT

Table of Contents

1. Introduction

This project aims to perform sentiment analysis on a dataset containing text. We will preprocess the data, implement various machine learning models, and evaluate their performance.

2. Data Exploration

<u>Dataset Description</u>

The dataset contains the following columns:
- `textID`
- `text`
- `selected_text`
- `sentiment`
- `Time of Tweet`
- `Age of User`
- `Country`
- `Population -2020`
- `Land Area (Km²)`
- `Density (P/Km²)`

<u>Key Variables</u>

- text: The content of the tweet.
- sentiment: The sentiment label of the tweet (e.g., positive, neutral, negative).

3. Data Preprocessing

Steps

1. **Lowercasing:** Convert all characters in the text to lowercase to ensure uniformity.
2. **Remove Special Characters and Digits:** Eliminate non-alphabetic characters and digits to clean the text.
3. **Tokenization:** Split the text into individual words or tokens.

Code Implementation

```
def clean(text):
    text = re.sub(r'[^a-zA-Z\s]', '', str(text))
    # Convert text to lowercase
    text = text.lower()
    # Remove special characters and digits
    text = re.sub(r'\W', ' ', text)
    text = re.sub(r'\d', ' ', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text
```

4. Text Vectorization

Using TF-IDF

TF-IDF Vectorization transforms **text data** into **numerical features** by calculating the Term Frequency-Inverse Document Frequency, which reflects the importance of a word in a document relative to its presence across the entire corpus.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Example text data
corpus = df['text'].tolist()
# Initialize the TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=5000)
# Fit and transform the text data to TF-IDF vectors
tfidf_vectors = tfidf_vectorizer.fit_transform(corpus)

# Convert sentiment labels to one-hot encoding
one_hot_labels = pd.get_dummies(df['sentiment']).values

# Split the data into training and testing sets
```

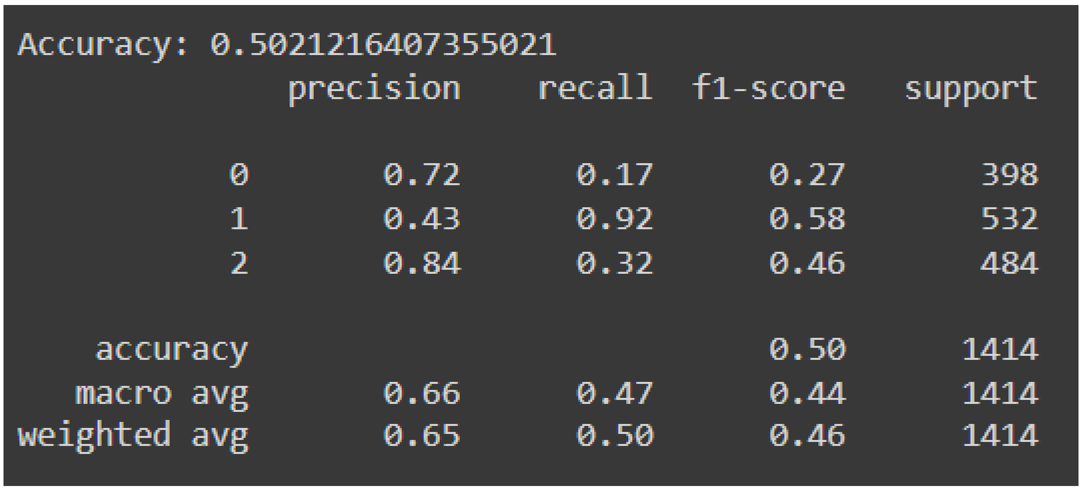
```
X_train, X_test, Y_train, Y_test = train_test_split(tfidf_vectors,
one_hot_labels, test_size=0.4)

# Print the shape of the training data
print(X_train.shape)
print(Y_train.shape)
```

One-hot encoding is used to convert categorical labels into a binary format that can be provided to machine learning algorithms to do a better job in prediction.

5. Model Development

Naive Bayes:
Naive Bayes is a probabilistic classifier based on Bayes' theorem, assuming strong independence between features. It's well-suited for text classification tasks due to its efficiency with high-dimensional and sparse data, like word counts in NLP applications and sentiment analysis.

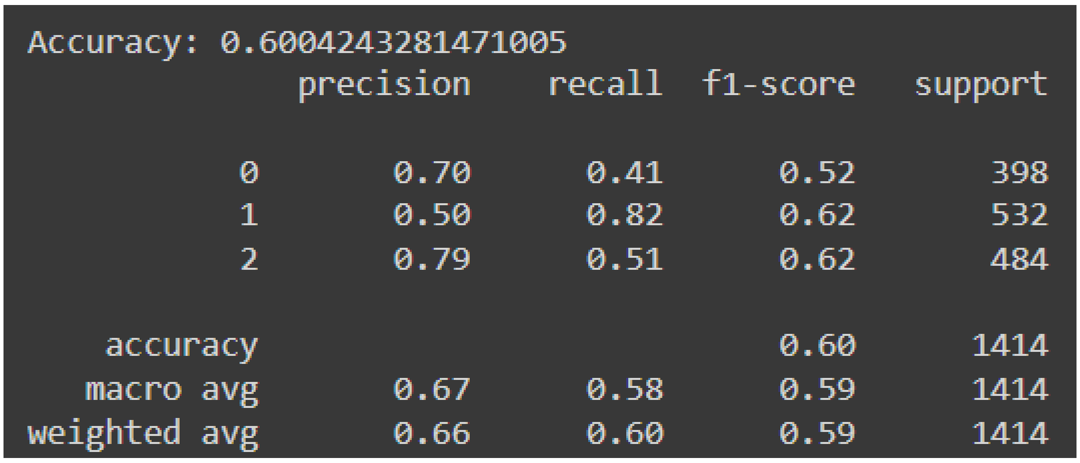
```
#train the Naive Bayes model
nb_model = MultinomialNB()
nb_model.fit(X_train, Y_train.argmax(axis=1))

# Make predictions
Y_pred = nb_model.predict(X_test)

# Evaluate the model
print(f"Accuracy: {accuracy_score(Y_test.argmax(axis=1), Y_pred)}")
print(classification_report(Y_test.argmax(axis=1), Y_pred))
```



```
Accuracy: 0.5021216407355021
              precision    recall  f1-score   support

           0       0.72      0.17      0.27       398
           1       0.43      0.92      0.58       532
           2       0.84      0.32      0.46       484

    accuracy                           0.50      1414
   macro avg       0.66      0.47      0.44      1414
weighted avg       0.65      0.50      0.46      1414
```

Support Vector Machines (SVM)
Support Vector Machines (SVM) are supervised learning models used for classification tasks. SVMs classify text documents (such as reviews or tweets) into positive, negative, or neutral sentiments based on the features extracted from the text (TF-IDF scores).

```
# Initialize and train the SVM model
svm_model = SVC(kernel='linear')
svm_model.fit(X_train, Y_train.argmax(axis=1))

# Make predictions
Y_pred = svm_model.predict(X_test)

# Evaluate the model
print(f"Accuracy: {accuracy_score(Y_test.argmax(axis=1), Y_pred)}")
print(classification_report(Y_test.argmax(axis=1), Y_pred))
```

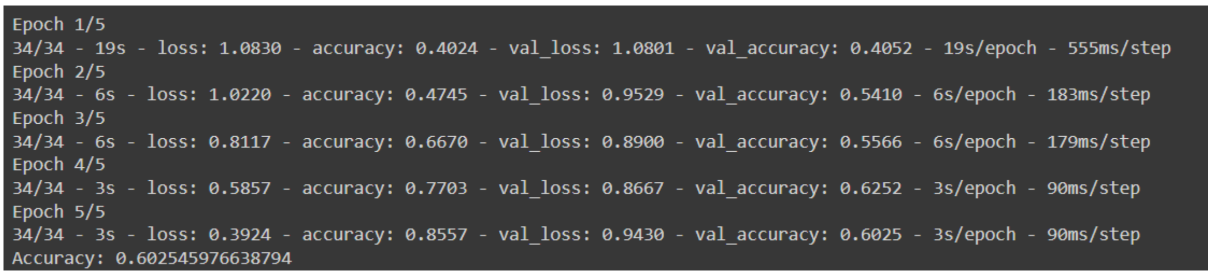


```
Accuracy: 0.6004243281471005
              precision    recall  f1-score   support

           0       0.70      0.41      0.52       398
           1       0.50      0.82      0.62       532
           2       0.79      0.51      0.62       484

    accuracy                           0.60      1414
   macro avg       0.67      0.58      0.59      1414
weighted avg       0.66      0.60      0.59      1414
```

LSTM Network
This LSTM-based sentiment analysis model tokenizes text, pads sequences, and encodes sentiments as one-hot vectors. It incorporates an embedding layer for learning word representations, an LSTM layer for sequential analysis with dropout, and a dense layer with softmax activation for classification. Trained with categorical cross-entropy loss and Adam optimizer, the model learns to predict sentiment labels from text data, evaluating performance using accuracy on test data.

```
texts = df['text'].tolist()

# Tokenize the text data
tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
word_index = tokenizer.word_index
```

```python
# Pad the sequences
max_seq_length = max(len(seq) for seq in sequences)
data = pad_sequences(sequences, maxlen=max_seq_length)

# Convert sentiment labels to one-hot encoding
one_hot_labels = pd.get_dummies(df['sentiment']).values

# Split the data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(data,
one_hot_labels, test_size=0.4, random_state=42)

# Initialize the LSTM model
lstm_model = Sequential()
lstm_model.add(Embedding(input_dim=len(word_index) + 1, output_dim=100,
input_length=max_seq_length))
lstm_model.add(SpatialDropout1D(0.2))
lstm_model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
lstm_model.add(Dense(one_hot_labels.shape[1], activation='softmax'))

# Compile the model
lstm_model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Train the model
lstm_model.fit(X_train, Y_train, epochs=5, batch_size=64,
validation_data=(X_test, Y_test), verbose=2)

# Evaluate the model
loss, accuracy = lstm_model.evaluate(X_test, Y_test, verbose=0)
print(f"Accuracy: {accuracy}")
```



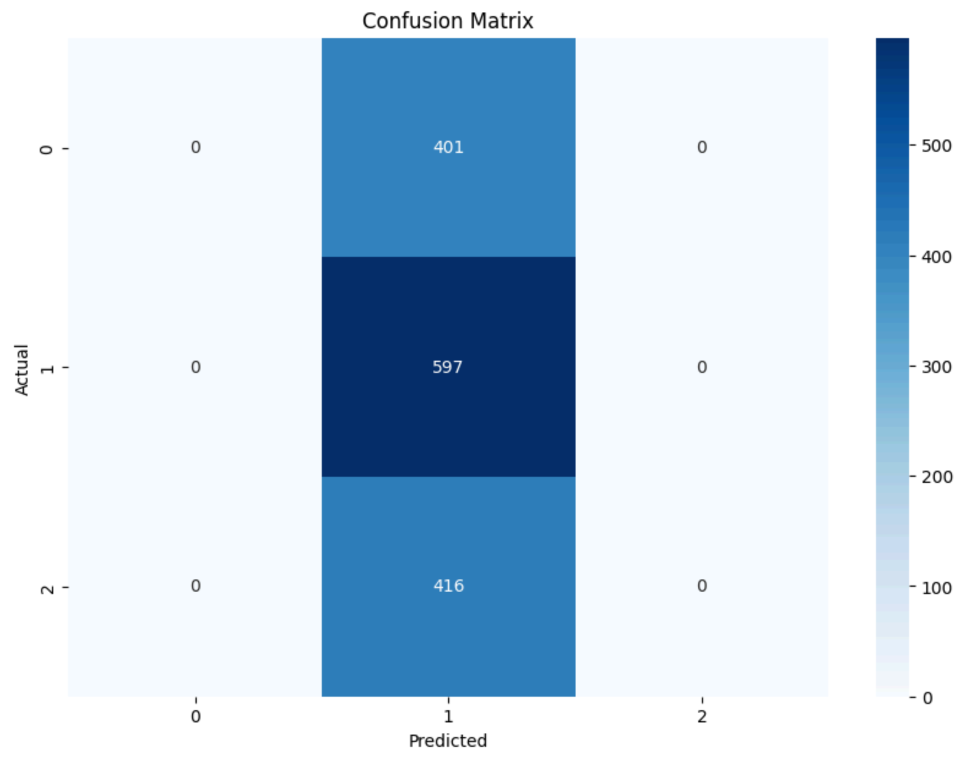
```
Epoch 1/5
34/34 - 19s - loss: 1.0830 - accuracy: 0.4024 - val_loss: 1.0801 - val_accuracy: 0.4052 - 19s/epoch - 555ms/step
Epoch 2/5
34/34 - 6s - loss: 1.0220 - accuracy: 0.4745 - val_loss: 0.9529 - val_accuracy: 0.5410 - 6s/epoch - 183ms/step
Epoch 3/5
34/34 - 6s - loss: 0.8117 - accuracy: 0.6670 - val_loss: 0.8900 - val_accuracy: 0.5566 - 6s/epoch - 179ms/step
Epoch 4/5
34/34 - 3s - loss: 0.5857 - accuracy: 0.7703 - val_loss: 0.8667 - val_accuracy: 0.6252 - 3s/epoch - 90ms/step
Epoch 5/5
34/34 - 3s - loss: 0.3924 - accuracy: 0.8557 - val_loss: 0.9430 - val_accuracy: 0.6025 - 3s/epoch - 90ms/step
Accuracy: 0.602545976638794
```

6. Model Evaluation

Confusion Matrix

1. It summarizes the classification results, detailing the counts of true positives, true negatives, false positives, and false negatives across sentiment categories (e.g., positive, negative).

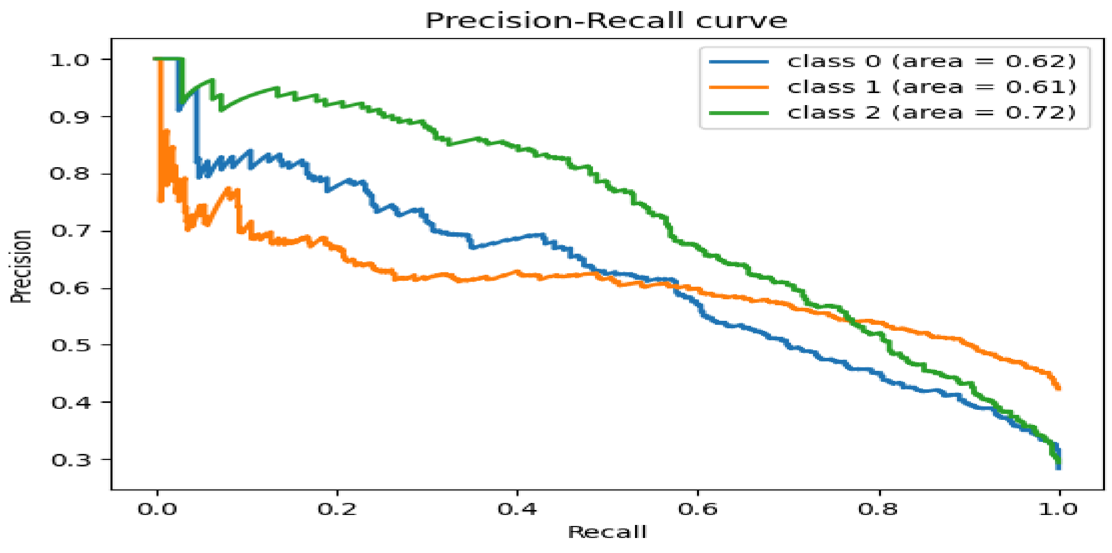
```
y_pred = best_svm_model.predict(X_test)
y_pred_proba = best_svm_model.predict_proba(X_test)
Y_test_single_label = Y_test.argmax(axis=1)
# Compute confusion matrix
conf_matrix = confusion_matrix(Y_test_single_label, y_pred)
#plot
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=best_svm_model.classes_,
yticklabels=best_svm_model.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



Precision-Recall Curve

1. It illustrates how well a classifier can correctly identify positive sentiment instances (precision) while also capturing all positive sentiment instances (recall).
2. Class 2 with AUC-PR of 0.72 suggests the model achieves relatively higher precision and recall for predicting its sentiment compared to the sentiments of class 0 and class 1.

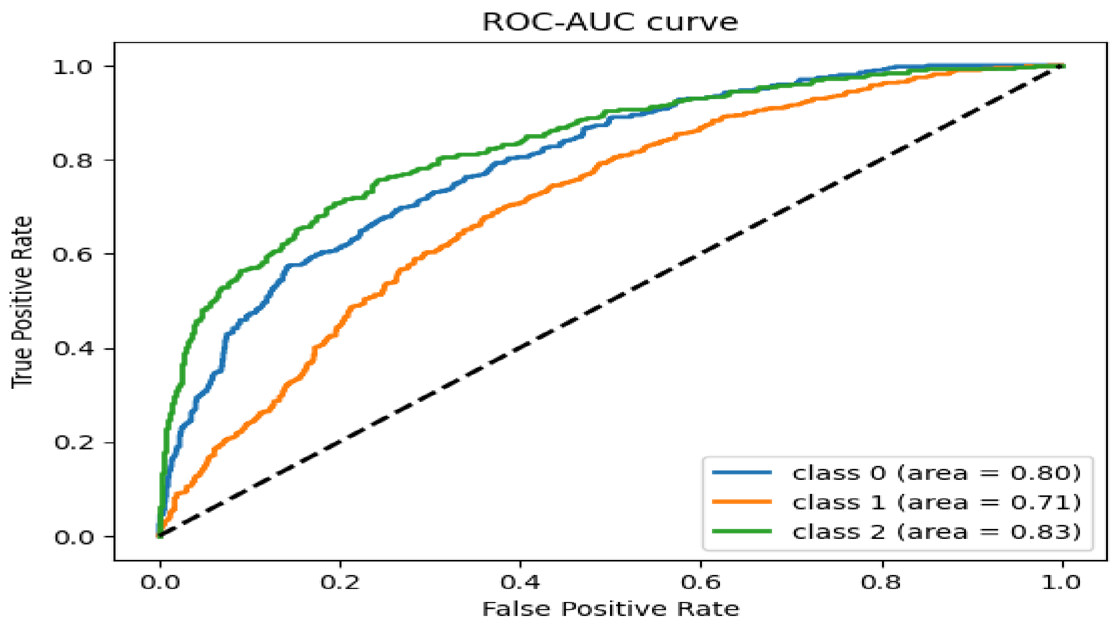
```
precision = dict()
recall = dict()
average_precision = dict()
n_classes = len(np.unique(Y_test_single_label))

for i in range(n_classes):
    precision[i], recall[i], _ =
precision_recall_curve(Y_test_single_label == i, y_pred_proba[:, i])
    average_precision[i] =
average_precision_score(Y_test_single_label== i, y_pred_proba[:, i])

# Plot Precision-Recall curve
plt.figure()
for i in range(n_classes):
    plt.plot(recall[i], precision[i], lw=2, label='class {0} (area =
{1:0.2f})'.format(i, average_precision[i]))

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.legend(loc="best")
plt.title("Precision-Recall curve")
plt.show()
```



ROC-AUC Curve

1. ROC curves are often used for binary sentiment classification tasks (e.g., positive vs. negative sentiments).
2. For multi-class sentiment analysis, ROC curves can be computed for each sentiment class against the rest (e.g., positive vs. rest), but often precision-recall curves are preferred due to their suitability for imbalanced datasets and multi-class scenarios.

```
fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(Y_test_single_label == i,
y_pred_proba[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot ROC curve
plt.figure()
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], lw=2, label='class {0} (area =
{1:0.2f})'.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend(loc="best")
plt.title("ROC-AUC curve")
plt.show()
```



7. Conclusion

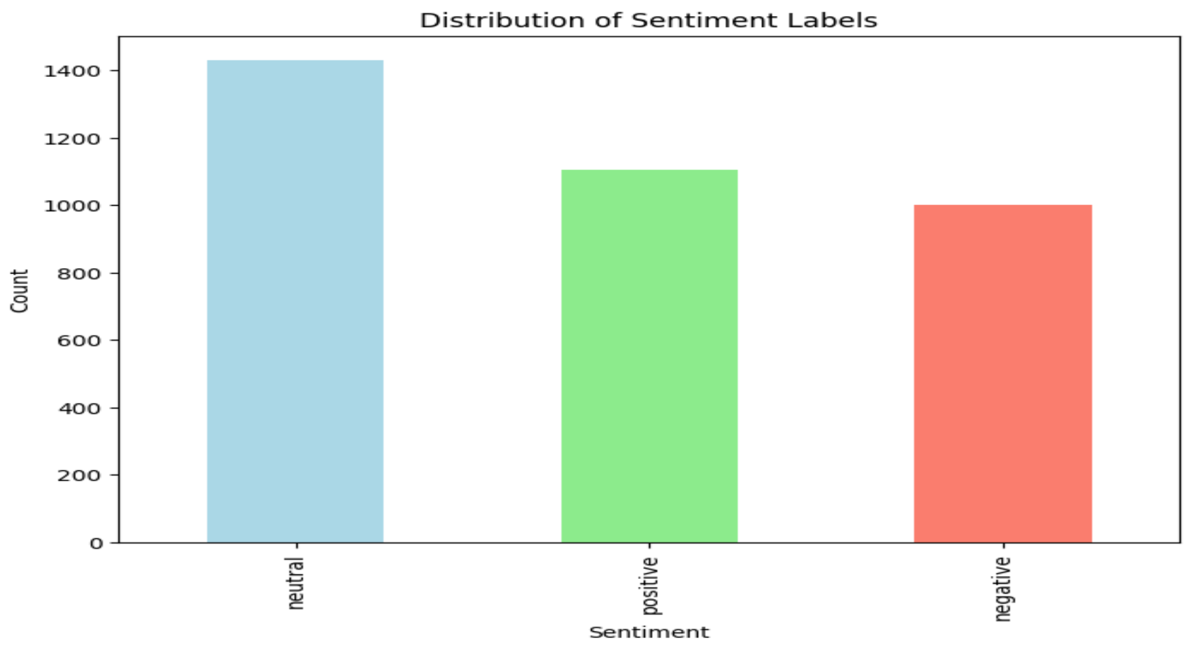
After evaluating all models, we can conclude that SVM performs the best with an accuracy of **0.6004243281471005**

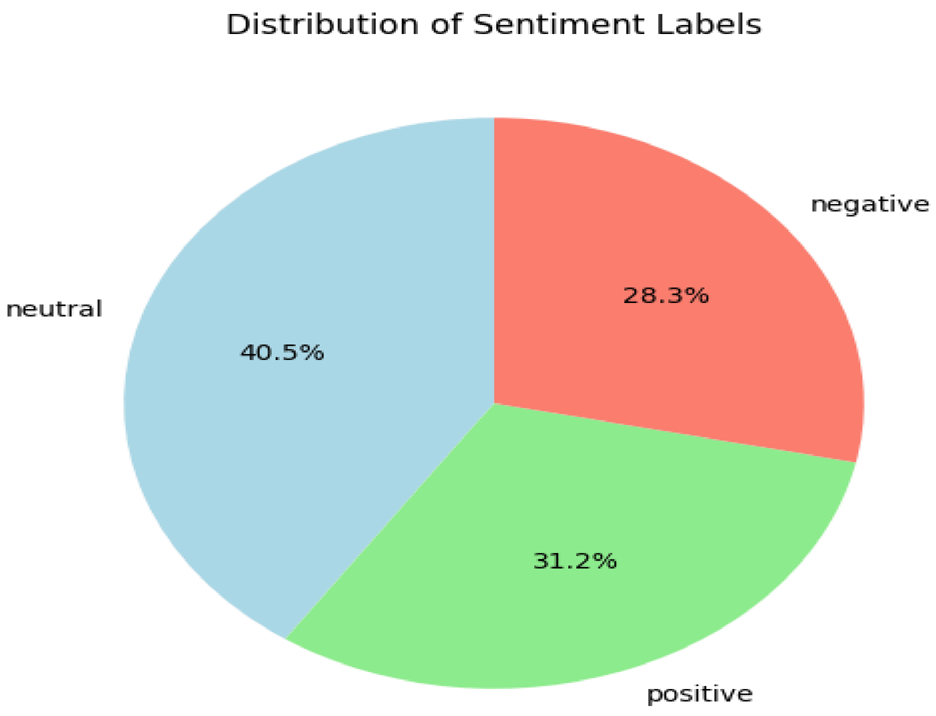
Further hyperparameter tuning is done to improve the performance.

```
param_grid = {
    'C': [0.1, 1, 10],
    # 'kernel': ['linear', 'rbf'],
    'gamma': ['scale', 'auto']
}
grid_search = GridSearchCV(SVC(), param_grid, cv=5, scoring='accuracy',
verbose=1)

# Perform Grid Search
grid_search.fit(X_train, Y_train.argmax(axis=1))

# Print the best parameters and best score
print(f"Best parameters: {grid_search.best_params_}")
print(f"Best accuracy: {grid_search.best_score_}")

# Evaluate the best model on the test set
best_svm_model = grid_search.best_estimator_
Y_pred_best_svm = best_svm_model.predict(X_test)
```

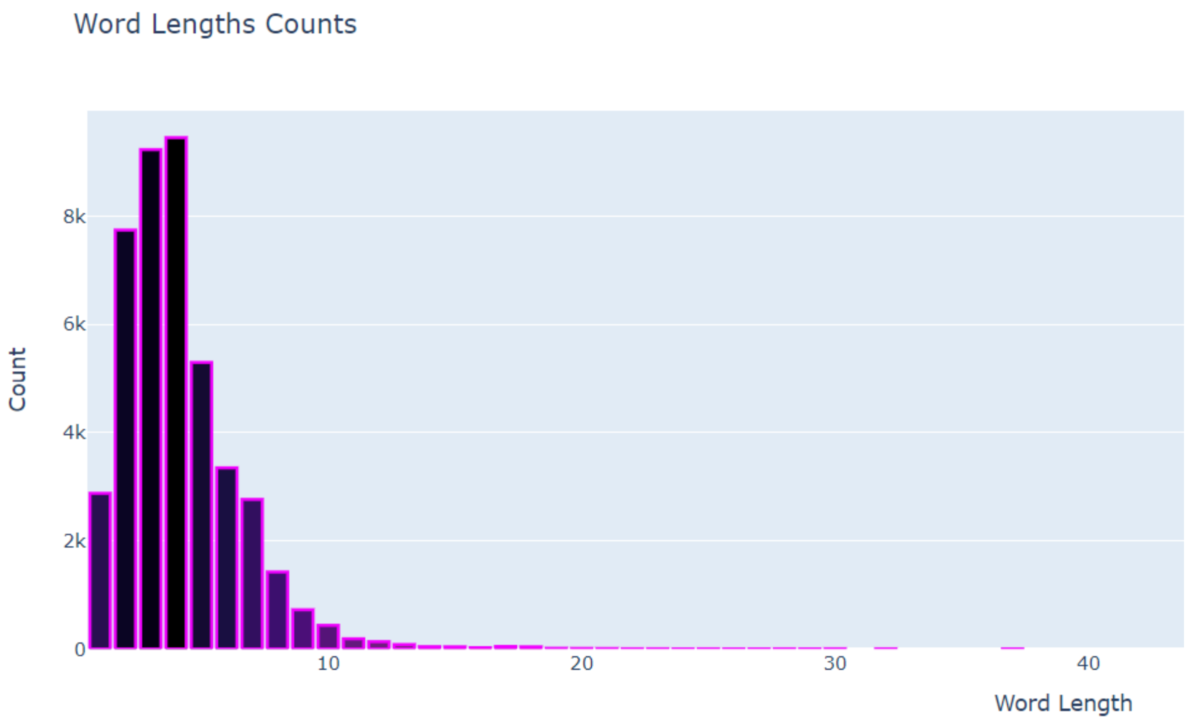
VISUALISATIONS

```
plt.figure(figsize=(8, 6))
sentiment_counts.plot.bar(color=['lightblue', 'lightgreen', 'salmon'])
plt.title('Distribution of Sentiment Labels')
plt.xlabel('Sentiment')
plt.ylabel('Count')
plt.show()
```



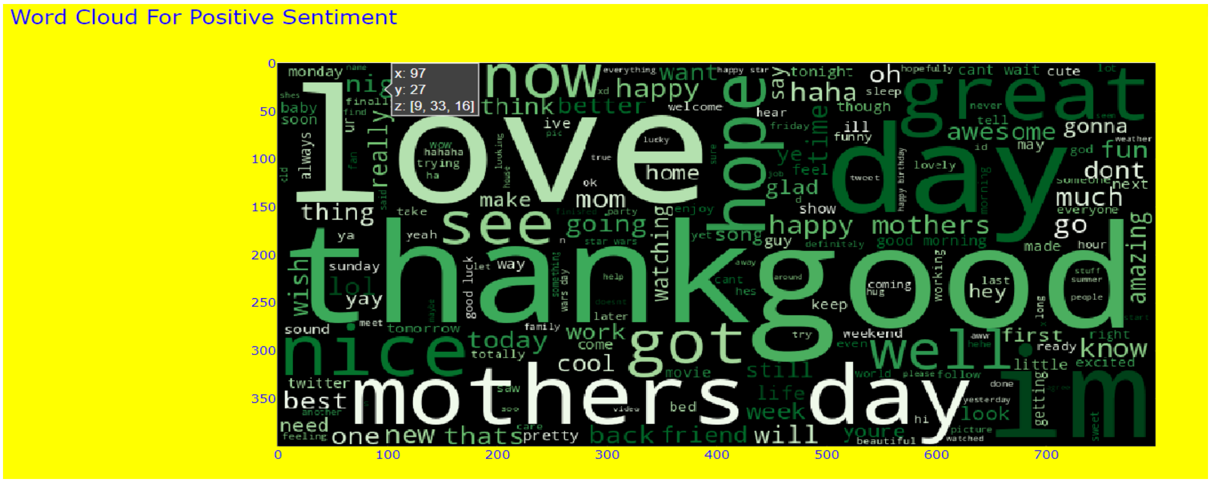


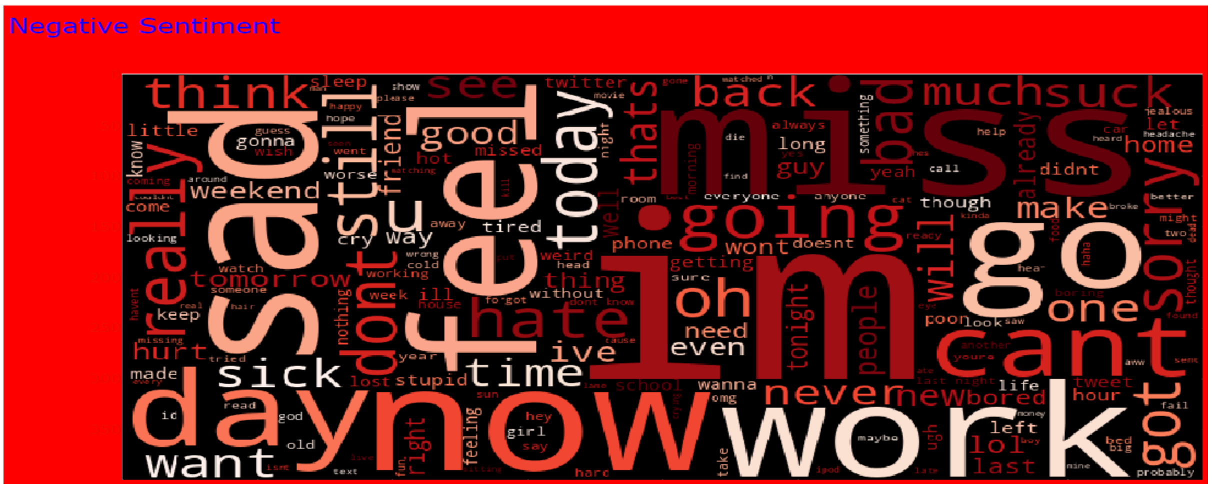
```
bar_trace = go.Bar(
    x=[length for length, count in sorted_word_lengths],
    y=[count for length, count in sorted_word_lengths],
    marker=dict(
        color=[colorscale[i] for i in range(len(sorted_word_lengths))],
        line=dict(
            color=palette[0],
            width=2
        )
    ),
    hovertemplate='Word Length: %{x}<br>Count: %{y}<extra></extra>'
)
```

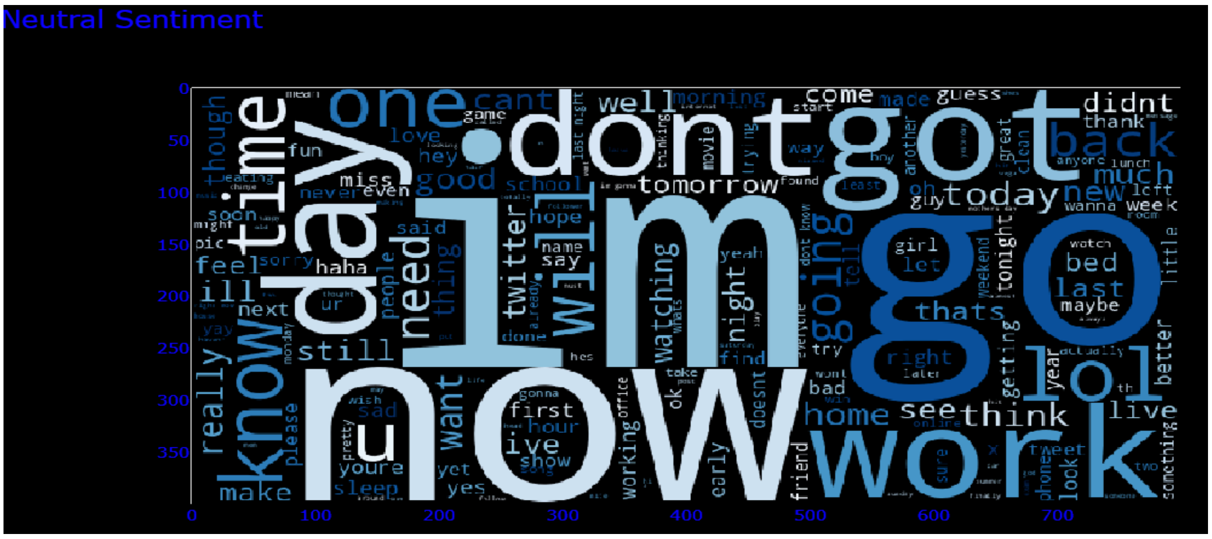


WORD CLOUDS

```
# Word Cloud for Positive Sentiment
positive_text = ' '.join(df[df['sentiment'] == 'positive']['text'])
wordcloud = WordCloud(background_color='black', width = 800, height =
400, max_words=200, colormap='Greens').generate(positive_text)
fig = go.Figure(go.Image(z = np.dstack((wordcloud.to_array(),
wordcloud.to_array(), wordcloud.to_array()))))
fig.update_layout(
    title = 'Word Cloud For Positive Sentiment',
    plot_bgcolor = 'blue',
    paper_bgcolor = 'yellow',
    font_color = palette[2],
    title_font_color = palette[2],
    title_font_size = 20,
    margin = dict(t = 80, l = 50, r = 50, b = 50)
)
fig.show()
```


Word Cloud For Positive Sentiment
x: 97
y: 27
z: [9, 33, 16]


Negative Sentiment


Neutral Sentiment