Sentiment Analysis

1. Introduction

Sentiment analysis, a subfield of natural language processing (NLP), involves analyzing textual data to determine the sentiment expressed by the writer. This project aims to classify user sentiments as positive, negative, or neutral based on their textual inputs. The analysis was conducted using various machine learning models, including Naive Bayes, Support Vector Machines (SVM), Long Short-Term Memory networks (LSTM), and Gated Recurrent Units (GRU). Additionally, the project utilized LIME (Local Interpretable Model-agnostic Explanations) for model interpretation.

Objective:

The primary objective of this project is to build and evaluate machine learning models that can classify user sentiments as positive, negative, or neutral based on textual data. Specifically, we seek to achieve the following goals:

- ➢ Develop predictive models using text data to classify sentiments.
- ➢ Evaluate the performance of different machine learning algorithms for sentiment classification.
- ➢ Provide insights into the trends and patterns observed in the textual data.

Statement:

The problem at hand is to develop predictive models capable of accurately classifying user sentiments in textual data. This task involves dealing with the inherent complexity of natural language and the challenges of text preprocessing and feature extraction.

2. Importing Libraries

Libraries such as pandas, NumPy, matplotlib, and seaborn are imported for data manipulation, visualization, and exploratory data analysis. Additionally, nltk is used for natural language processing tasks, including tokenization, removing stopwords, and lemmatization. warnings is used to suppress unnecessary warnings during execution. TensorFlow, Keras for model.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import nltk
nltk.download('punkt')
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import string
nltk.download('stopwords')
nltk.download('wordnet')
import warnings
warnings.filterwarnings("ignore")
```

3. Data Collection and Preprocessing

Data Collection

- The dataset used in this project was sourced from a CSV file, train.csv.
- The data contains text data along with sentiment labels.
- The data was imported using the pandas library.

```
df = pd.read_csv('train.csv', encoding='ISO-8859-1')
```

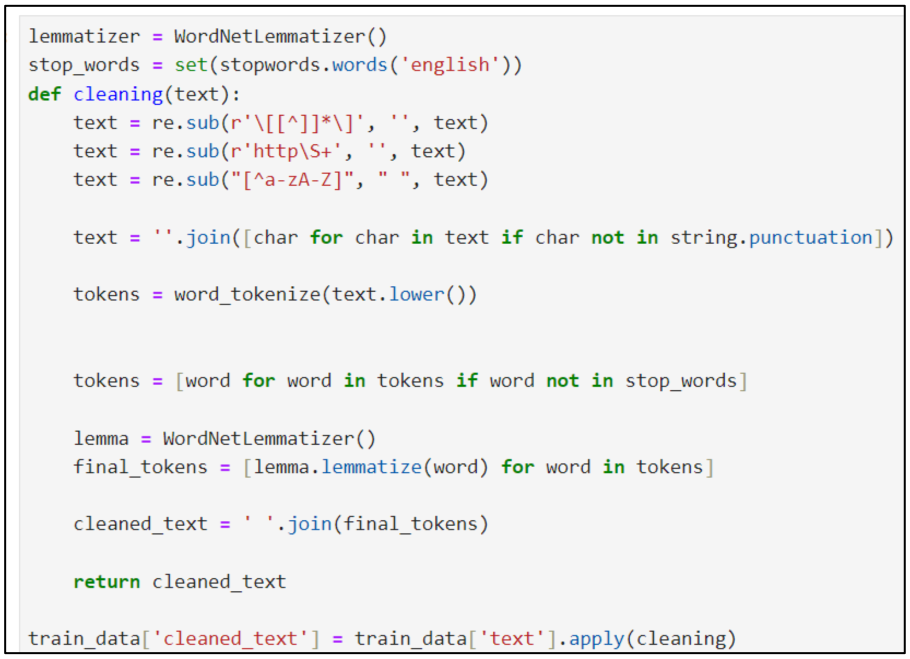
Data Preprocessing

Initial Data Inspection

- Inspected the dataset to understand its structure and contents.
- Checked for duplicates and missing values.
- Analyzed basic statistics of the data.
- Cleaned the dataset by removing rows with missing values.
- Renamed columns to eliminate spaces, replacing them with underscores.

Text Cleaning

- Preprocessed text data to make it suitable for analysis.
- Removed unwanted characters, such as punctuation, links and special symbols.
- Tokenized the text to split it into individual words.
- Removed stopwords to eliminate common but uninformative words.
- Lemmatized the words to reduce them to their base or root form.



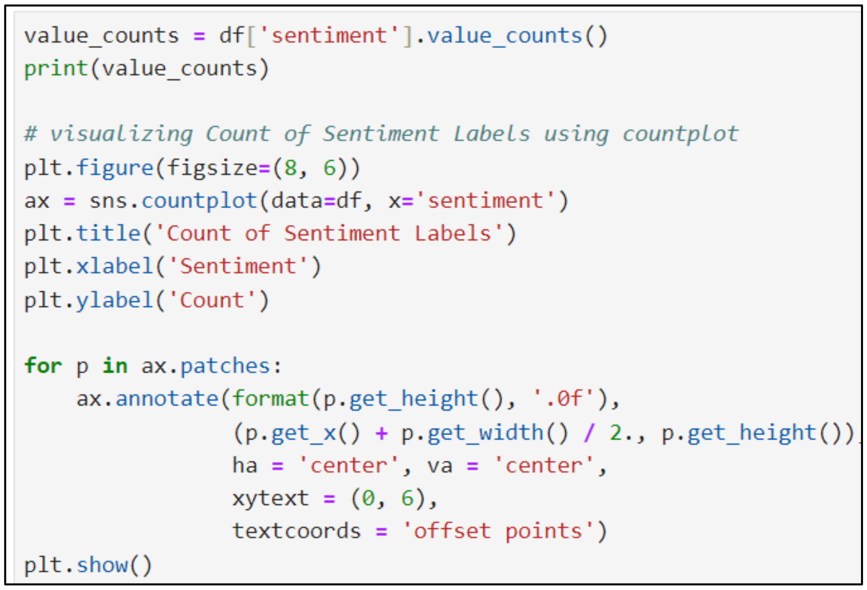
```python
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
def cleaning(text):
    text = re.sub(r'\[[^]]*\]', '', text)
    text = re.sub(r'http\S+', '', text)
    text = re.sub("[^a-zA-Z]", " ", text)

    text = ''.join([char for char in text if char not in string.punctuation])

    tokens = word_tokenize(text.lower())


    tokens = [word for word in tokens if word not in stop_words]

    lemma = WordNetLemmatizer()
    final_tokens = [lemma.lemmatize(word) for word in tokens]

    cleaned_text = ' '.join(final_tokens)

    return cleaned_text

train_data['cleaned_text'] = train_data['text'].apply(cleaning)
```

4. Exploratory Data Analysis (EDA)

4.1 Count of Sentiment Labels:

The distribution of sentiment labels (positive, negative, neutral) was visualized using count plots. This helped in understanding the balance of the dataset.

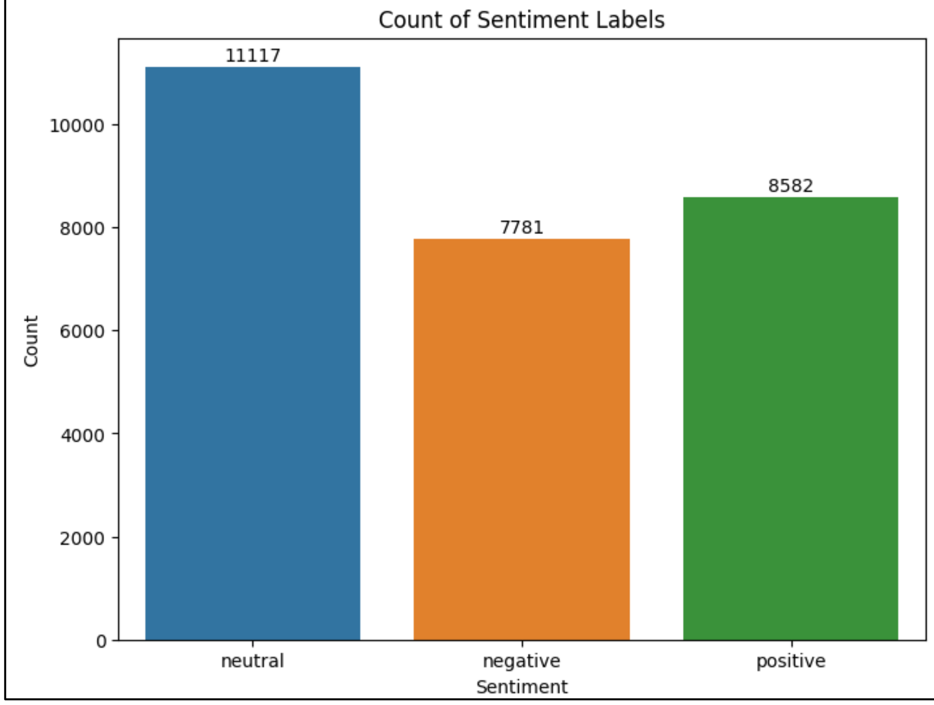
Code:



```python
value_counts = df['sentiment'].value_counts()
print(value_counts)

# visualizing Count of Sentiment Labels using countplot
plt.figure(figsize=(8, 6))
ax = sns.countplot(data=df, x='sentiment')
plt.title('Count of Sentiment Labels')
plt.xlabel('Sentiment')
plt.ylabel('Count')

for p in ax.patches:
    ax.annotate(format(p.get_height(), '.0f'),
                (p.get_x() + p.get_width() / 2., p.get_height())
                ha = 'center', va = 'center',
                xytext = (0, 6),
                textcoords = 'offset points')
plt.show()
```

Output:

```
sentiment
neutral     11117
positive     8582
negative     7781
Name: count, dtype: int64
```



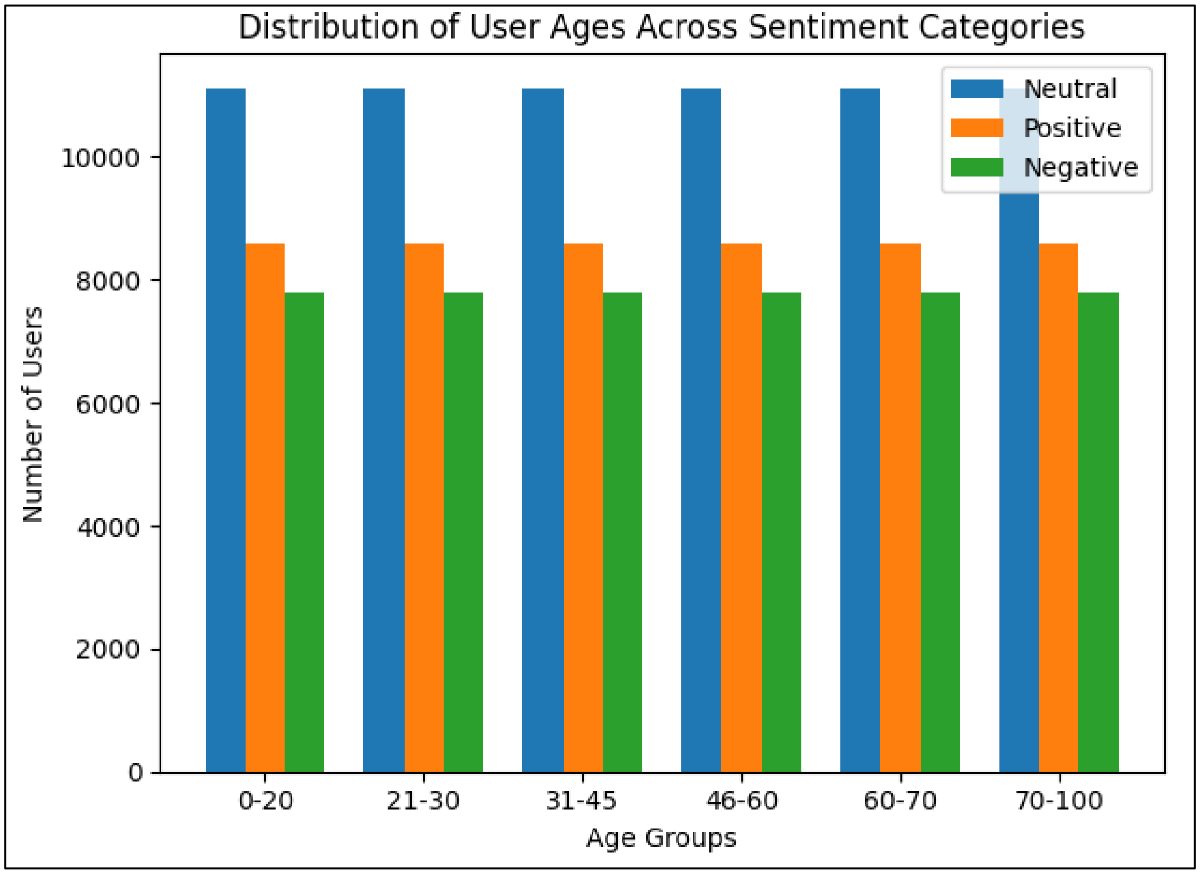
4.2 Distribution of User Ages Across Sentiment Categories:

The dataset contained user ages, and their distribution across different sentiment categories was analyzed using bar plots. The code extracts unique age groups and assigns counts for each sentiment category.

Code:

```
age_groups = df['Age_of_User'].unique()
neutral_count = value_counts['neutral']
positive_count = value_counts['positive']
negative_count = value_counts['negative']

bar_width = 0.25
index = range(len(age_groups))

plt.bar(index, [neutral_count] * len(age_groups), bar_width, label='Neutral')
plt.bar([i + bar_width for i in index], [positive_count] * len(age_groups), bar_width, label='Positive')
plt.bar([i + 2 * bar_width for i in index], [negative_count] * len(age_groups), bar_width, label='Negative')

plt.xlabel('Age Groups')
plt.ylabel('Number of Users')
plt.title('Distribution of User Ages Across Sentiment Categories')
plt.xticks([i + bar_width for i in index], age_groups)
plt.legend()
plt.tight_layout()
plt.show()
```

Output:



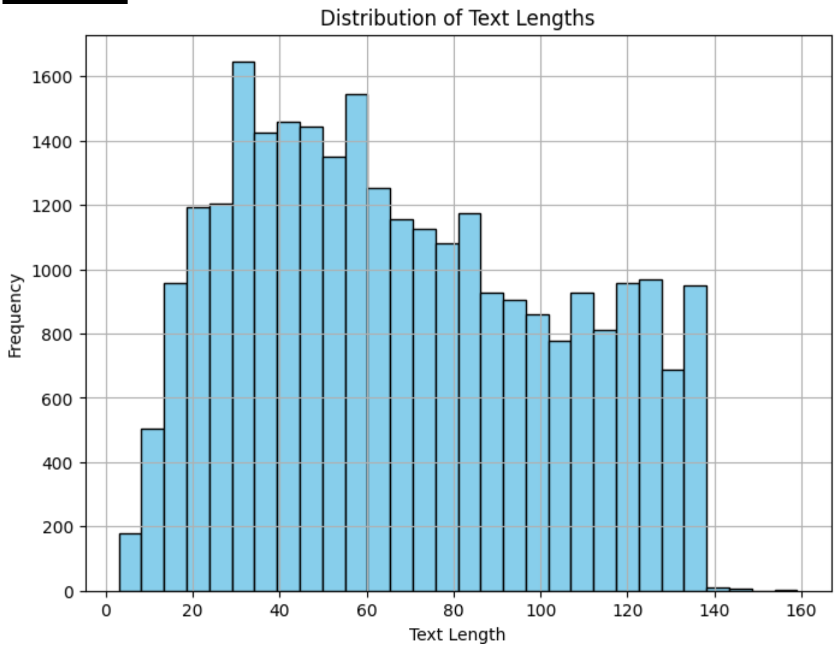
4.3 Distribution of Text Lengths:

The lengths of text entries were analyzed to understand the variation in the amount of text provided by users

Code:

```
text_lengths = [len(text) for text in df['text']]

plt.figure(figsize=(8, 6))
plt.hist(text_lengths, bins=30, color='skyblue', edgecolor='black')
plt.xlabel('Text Length')
plt.ylabel('Frequency')
plt.title('Distribution of Text Lengths')
plt.grid(True)
plt.show()
```

Output:



4.4 Most Common Words in Selected Text:

The frequency of words in the text data was analyzed to identify the most common words, providing insights into the language used by the users.

Code:

```
from nltk.probability import FreqDist

selected_text = " ".join(df['selected_text'].astype(str))
tokens = word_tokenize(selected_text)
tokens = [word for word in tokens if word.isalnum()]
tokens = [word.lower() for word in tokens]
freq_dist = FreqDist(tokens)

print("Most common words in selected text:")
print(freq_dist.most_common(10))
```

Output:

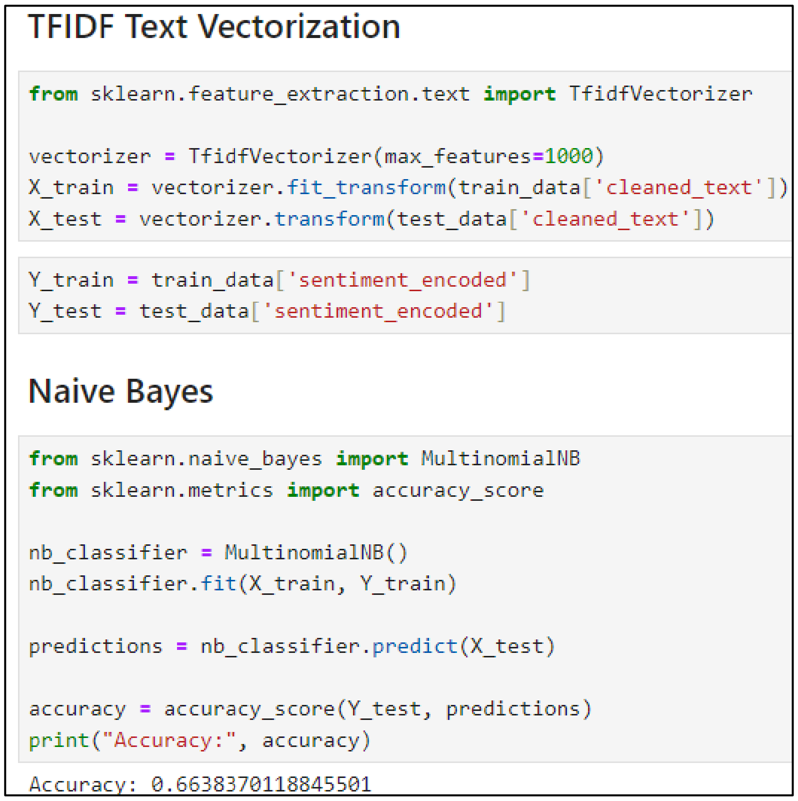
```
Most common words in selected text:
[('i', 8997), ('to', 5303), ('the', 4594), ('a', 3541), ('you', 2870), ('it', 2836), ('my', 2793), ('and', 2356), ('is', 2117),
('s', 2002)]
```

5. Machine Learning Models

5.1 Naive Bayes

A Naive Bayes classifier was used as the baseline model for sentiment analysis. This probabilistic model is simple yet effective for text classification tasks. The text data was vectorized using TF-IDF (Term Frequency-Inverse Document Frequency) before being fed into the model. The classifier achieved reasonable accuracy, making it a good starting point for comparison with more complex models.

Code:



TFIDF Text Vectorization

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(max_features=1000)
X_train = vectorizer.fit_transform(train_data['cleaned_text'])
X_test = vectorizer.transform(test_data['cleaned_text'])
```

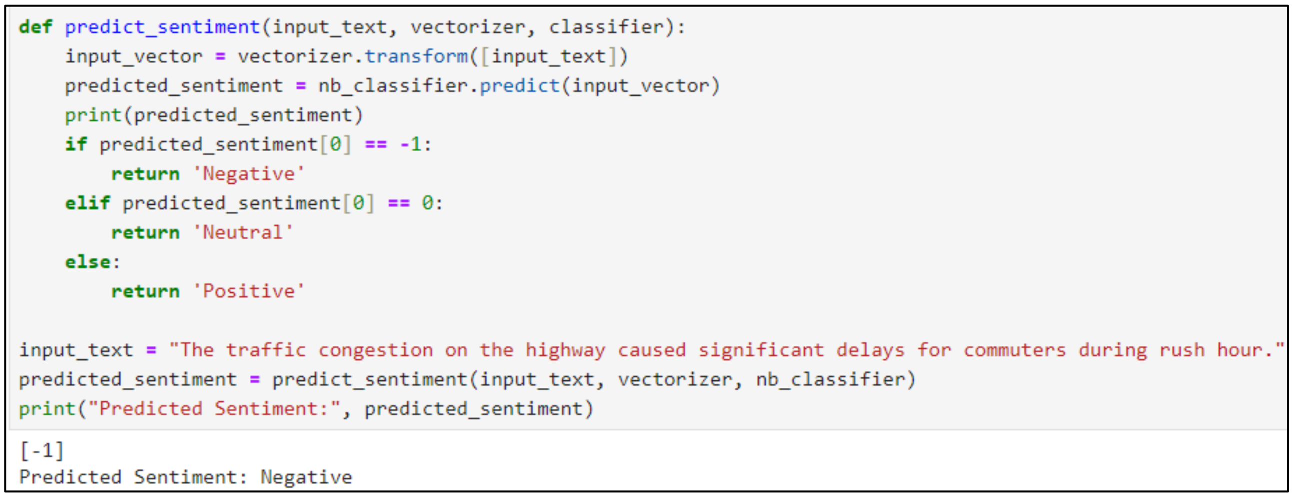
```
Y_train = train_data['sentiment_encoded']
Y_test = test_data['sentiment_encoded']
```

Naive Bayes

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

nb_classifier = MultinomialNB()
nb_classifier.fit(X_train, Y_train)

predictions = nb_classifier.predict(X_test)

accuracy = accuracy_score(Y_test, predictions)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.6638370118845501
```

Predictions on custom data:



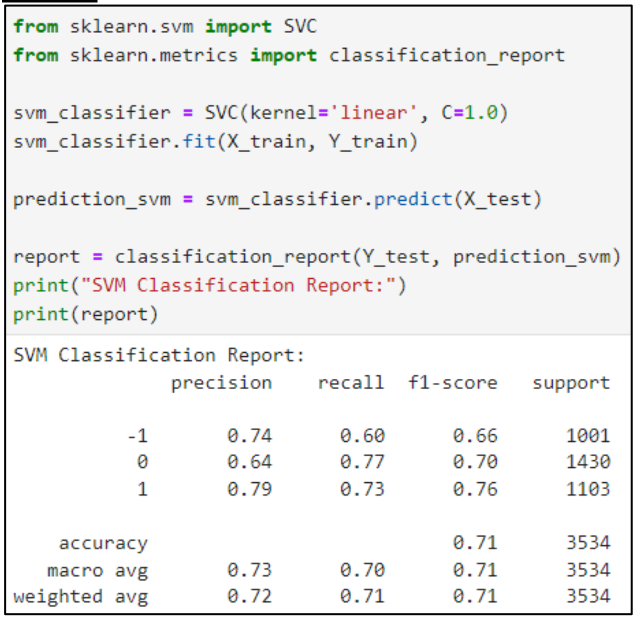
```
def predict_sentiment(input_text, vectorizer, classifier):
    input_vector = vectorizer.transform([input_text])
    predicted_sentiment = nb_classifier.predict(input_vector)
    print(predicted_sentiment)
    if predicted_sentiment[0] == -1:
        return 'Negative'
    elif predicted_sentiment[0] == 0:
        return 'Neutral'
    else:
        return 'Positive'

input_text = "The traffic congestion on the highway caused significant delays for commuters during rush hour."
predicted_sentiment = predict_sentiment(input_text, vectorizer, nb_classifier)
print("Predicted Sentiment:", predicted_sentiment)
```

```
[-1]
Predicted Sentiment: Negative
```

5.2 Support Vector Machine (SVM)

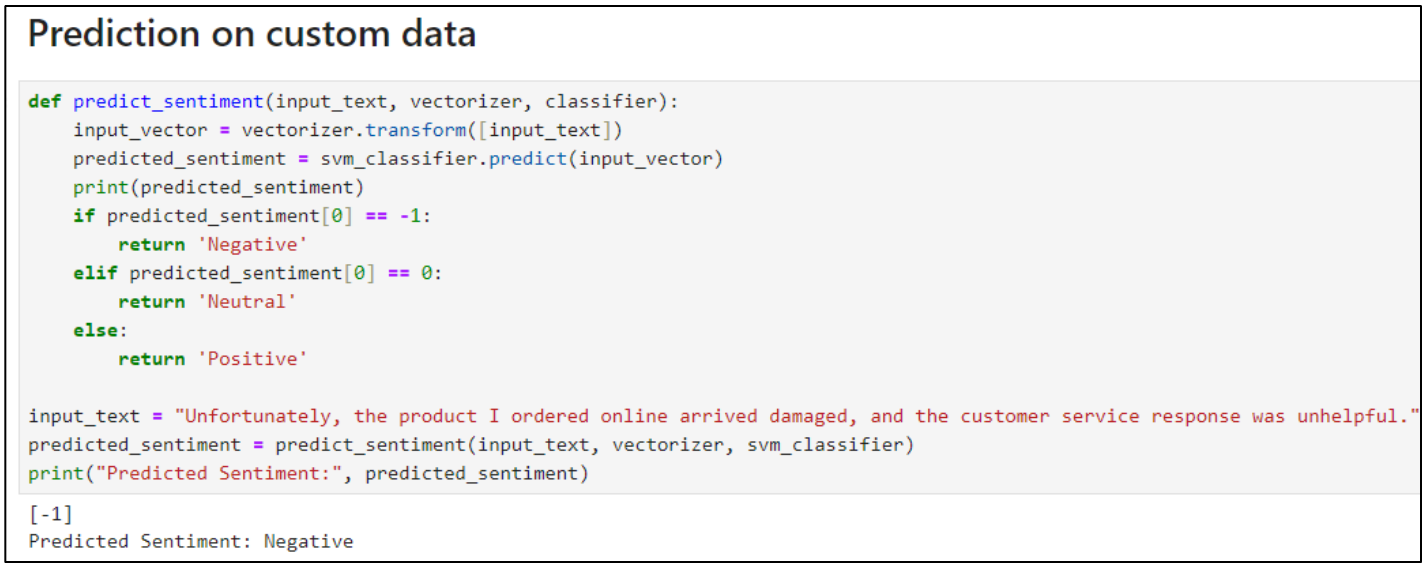
An SVM with a linear kernel was used to classify sentiments. SVMs are powerful classifiers that work well with high-dimensional data, such as text data. The model's performance was evaluated using a classification report, which provided metrics such as precision, recall, and F1-score.

Code:



```
from sklearn.svm import SVC
from sklearn.metrics import classification_report

svm_classifier = SVC(kernel='linear', C=1.0)
svm_classifier.fit(X_train, Y_train)

prediction_svm = svm_classifier.predict(X_test)

report = classification_report(Y_test, prediction_svm)
print("SVM Classification Report:")
print(report)
```

```
SVM Classification Report:
              precision    recall  f1-score   support

          -1       0.74      0.60      0.66      1001
           0       0.64      0.77      0.70      1430
           1       0.79      0.73      0.76      1103

    accuracy                           0.71      3534
   macro avg       0.73      0.70      0.71      3534
weighted avg       0.72      0.71      0.71      3534
```



Prediction on custom data

```
def predict_sentiment(input_text, vectorizer, classifier):
    input_vector = vectorizer.transform([input_text])
    predicted_sentiment = svm_classifier.predict(input_vector)
    print(predicted_sentiment)
    if predicted_sentiment[0] == -1:
        return 'Negative'
    elif predicted_sentiment[0] == 0:
        return 'Neutral'
    else:
        return 'Positive'

input_text = "Unfortunately, the product I ordered online arrived damaged, and the customer service response was unhelpful."
predicted_sentiment = predict_sentiment(input_text, vectorizer, svm_classifier)
print("Predicted Sentiment:", predicted_sentiment)
```

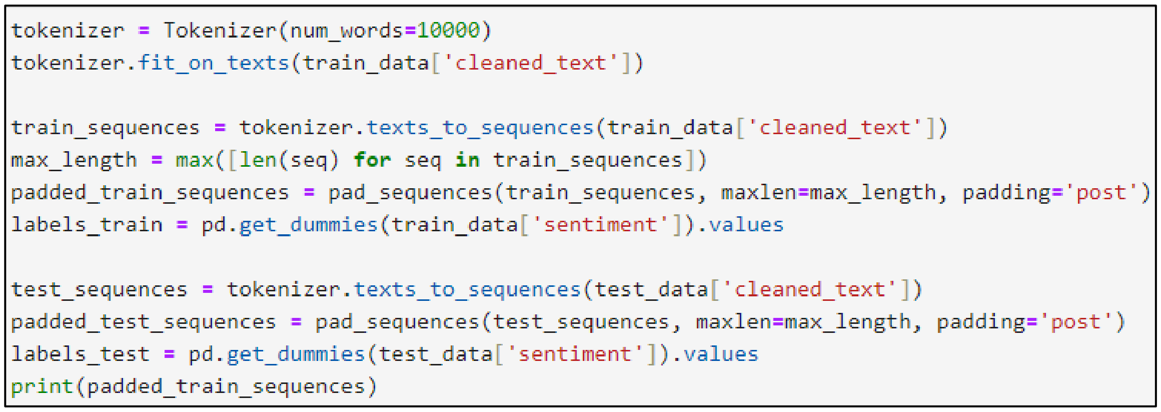
```
[-1]
Predicted Sentiment: Negative
```

5.3 Long Short-Term Memory (LSTM)

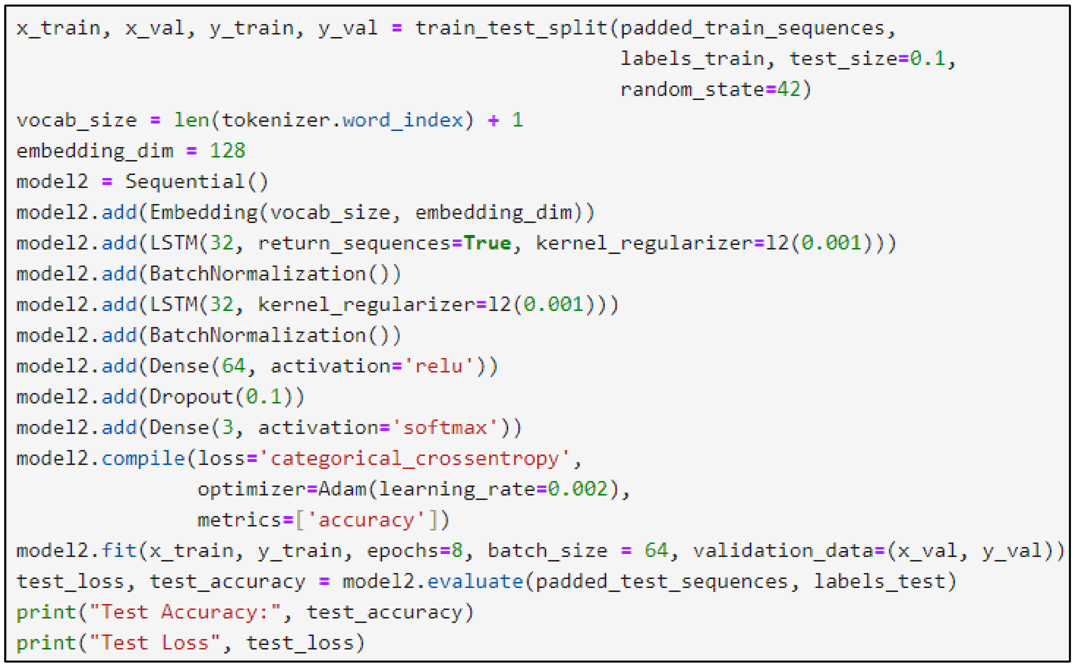
LSTM networks, a type of recurrent neural network (RNN), were employed to capture long-term dependencies in the text data. The text sequences were tokenized and padded to ensure uniform input lengths. The LSTM model was trained and validated, showing improved performance over traditional models like Naive Bayes and SVM.

- Utilized Tokenizer from Keras to tokenize text data.
- Padded sequences to ensure uniform input length for the LSTM model.

Code:

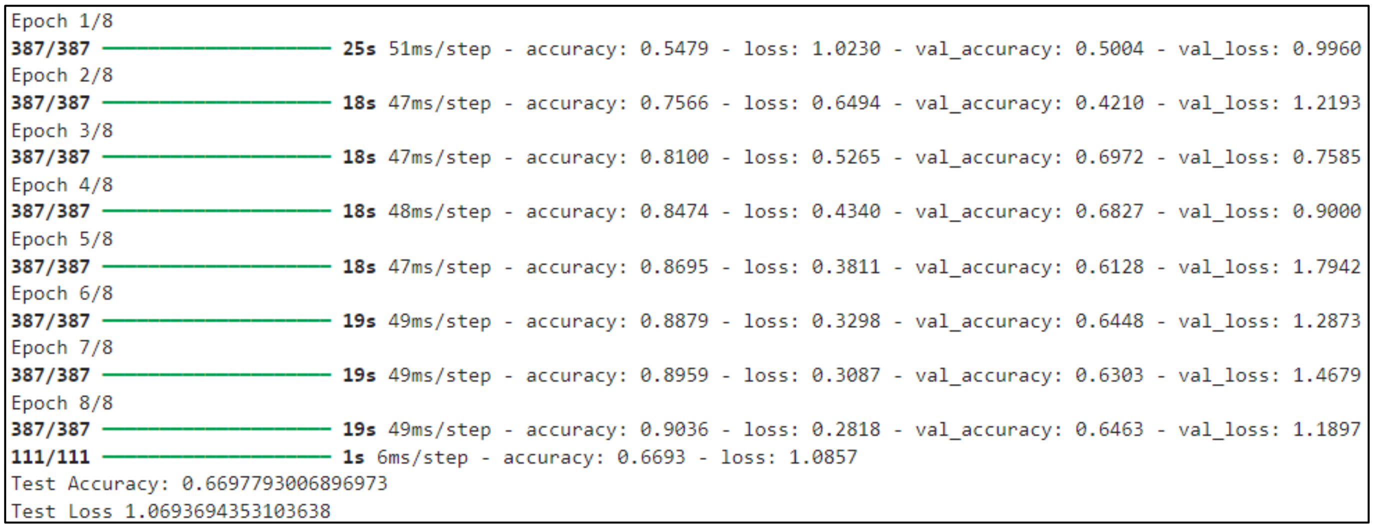


```python
tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(train_data['cleaned_text'])

train_sequences = tokenizer.texts_to_sequences(train_data['cleaned_text'])
max_length = max([len(seq) for seq in train_sequences])
padded_train_sequences = pad_sequences(train_sequences, maxlen=max_length, padding='post')
labels_train = pd.get_dummies(train_data['sentiment']).values

test_sequences = tokenizer.texts_to_sequences(test_data['cleaned_text'])
padded_test_sequences = pad_sequences(test_sequences, maxlen=max_length, padding='post')
labels_test = pd.get_dummies(test_data['sentiment']).values
print(padded_train_sequences)
```



```python
x_train, x_val, y_train, y_val = train_test_split(padded_train_sequences,
                                                  labels_train, test_size=0.1,
                                                  random_state=42)
vocab_size = len(tokenizer.word_index) + 1
embedding_dim = 128
model2 = Sequential()
model2.add(Embedding(vocab_size, embedding_dim))
model2.add(LSTM(32, return_sequences=True, kernel_regularizer=l2(0.001)))
model2.add(BatchNormalization())
model2.add(LSTM(32, kernel_regularizer=l2(0.001)))
model2.add(BatchNormalization())
model2.add(Dense(64, activation='relu'))
model2.add(Dropout(0.1))
model2.add(Dense(3, activation='softmax'))
model2.compile(loss='categorical_crossentropy',
               optimizer=Adam(learning_rate=0.002),
               metrics=['accuracy'])
model2.fit(x_train, y_train, epochs=8, batch_size = 64, validation_data=(x_val, y_val))
test_loss, test_accuracy = model2.evaluate(padded_test_sequences, labels_test)
print("Test Accuracy:", test_accuracy)
print("Test Loss", test_loss)
```

Epochs:



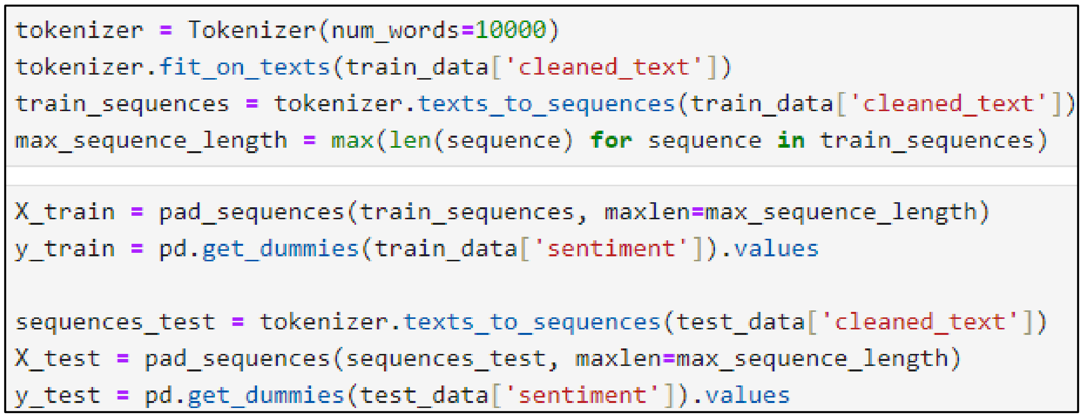
```
Epoch 1/8
387/387 ━━━━━━━━━━━━━━━━━━━━ 25s 51ms/step - accuracy: 0.5479 - loss: 1.0230 - val_accuracy: 0.5004 - val_loss: 0.9960
Epoch 2/8
387/387 ━━━━━━━━━━━━━━━━━━━━ 18s 47ms/step - accuracy: 0.7566 - loss: 0.6494 - val_accuracy: 0.4210 - val_loss: 1.2193
Epoch 3/8
387/387 ━━━━━━━━━━━━━━━━━━━━ 18s 47ms/step - accuracy: 0.8100 - loss: 0.5265 - val_accuracy: 0.6972 - val_loss: 0.7585
Epoch 4/8
387/387 ━━━━━━━━━━━━━━━━━━━━ 18s 48ms/step - accuracy: 0.8474 - loss: 0.4340 - val_accuracy: 0.6827 - val_loss: 0.9000
Epoch 5/8
387/387 ━━━━━━━━━━━━━━━━━━━━ 18s 47ms/step - accuracy: 0.8695 - loss: 0.3811 - val_accuracy: 0.6128 - val_loss: 1.7942
Epoch 6/8
387/387 ━━━━━━━━━━━━━━━━━━━━ 19s 49ms/step - accuracy: 0.8879 - loss: 0.3298 - val_accuracy: 0.6448 - val_loss: 1.2873
Epoch 7/8
387/387 ━━━━━━━━━━━━━━━━━━━━ 19s 49ms/step - accuracy: 0.8959 - loss: 0.3087 - val_accuracy: 0.6303 - val_loss: 1.4679
Epoch 8/8
387/387 ━━━━━━━━━━━━━━━━━━━━ 19s 49ms/step - accuracy: 0.9036 - loss: 0.2818 - val_accuracy: 0.6463 - val_loss: 1.1897
111/111 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.6693 - loss: 1.0857
Test Accuracy: 0.6697793006896973
Test Loss 1.0693694353103638
```

5.4 Gated Recurrent Unit (GRU)

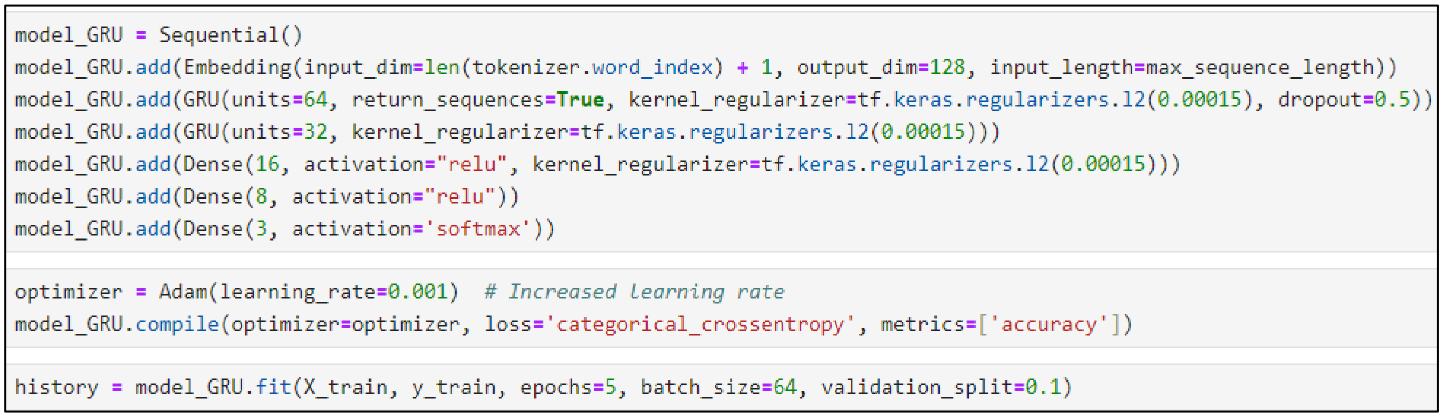
GRU, another type of RNN, was used to compare its performance with LSTM. The GRU model was configured with embedding layers and dropout regularization to prevent overfitting. The training process included monitoring validation accuracy, and the model's performance was evaluated on the test data.

- Utilized Tokenizer from Keras to tokenize text data.
- Padded sequences to ensure uniform input length for the LSTM model.

<u>Code:</u>

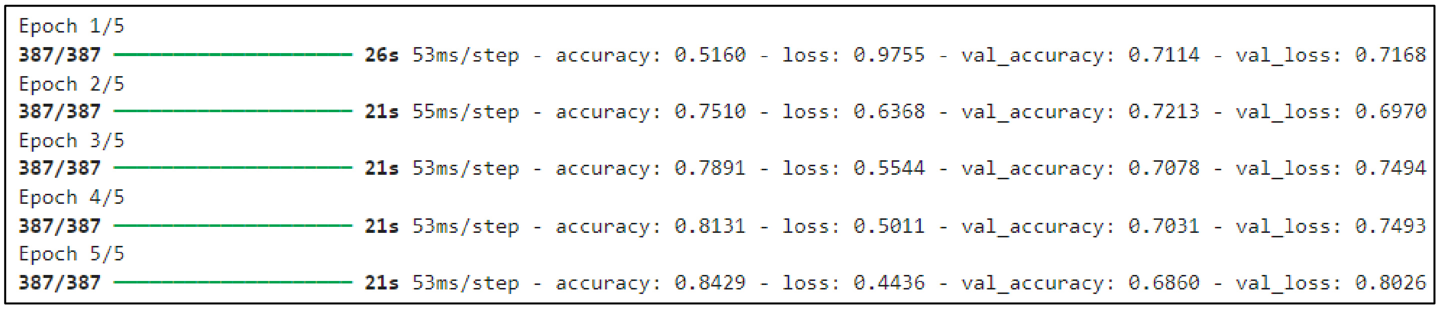


```python
tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(train_data['cleaned_text'])
train_sequences = tokenizer.texts_to_sequences(train_data['cleaned_text'])
max_sequence_length = max(len(sequence) for sequence in train_sequences)

X_train = pad_sequences(train_sequences, maxlen=max_sequence_length)
y_train = pd.get_dummies(train_data['sentiment']).values

sequences_test = tokenizer.texts_to_sequences(test_data['cleaned_text'])
X_test = pad_sequences(sequences_test, maxlen=max_sequence_length)
y_test = pd.get_dummies(test_data['sentiment']).values
```

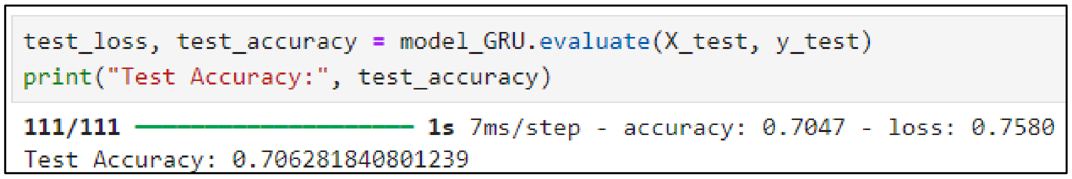


```python
model_GRU = Sequential()
model_GRU.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=128, input_length=max_sequence_length))
model_GRU.add(GRU(units=64, return_sequences=True, kernel_regularizer=tf.keras.regularizers.l2(0.00015), dropout=0.5))
model_GRU.add(GRU(units=32, kernel_regularizer=tf.keras.regularizers.l2(0.00015)))
model_GRU.add(Dense(16, activation="relu", kernel_regularizer=tf.keras.regularizers.l2(0.00015)))
model_GRU.add(Dense(8, activation="relu"))
model_GRU.add(Dense(3, activation='softmax'))

optimizer = Adam(learning_rate=0.001)  # Increased learning rate
model_GRU.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

history = model_GRU.fit(X_train, y_train, epochs=5, batch_size=64, validation_split=0.1)
```

<u>Epochs</u>:

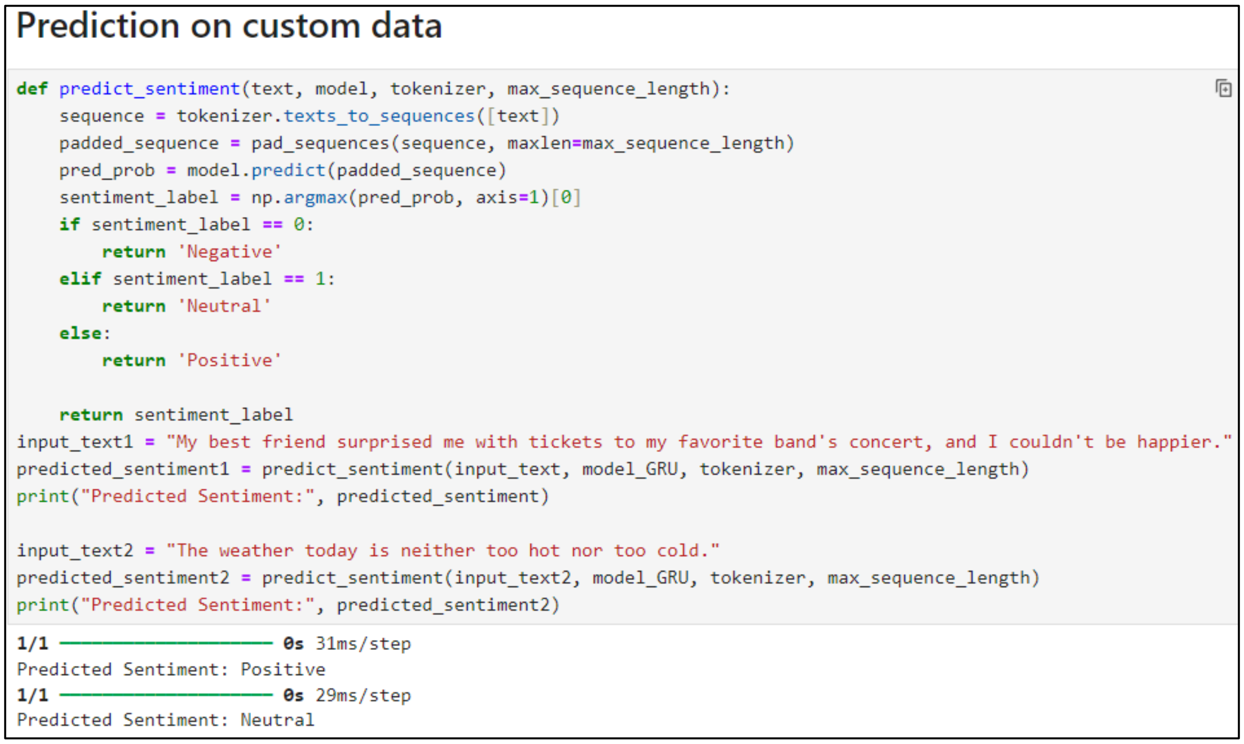


```
Epoch 1/5
387/387 ━━━━━━━━━━━━━━━━━━━━ 26s 53ms/step - accuracy: 0.5160 - loss: 0.9755 - val_accuracy: 0.7114 - val_loss: 0.7168
Epoch 2/5
387/387 ━━━━━━━━━━━━━━━━━━━━ 21s 55ms/step - accuracy: 0.7510 - loss: 0.6368 - val_accuracy: 0.7213 - val_loss: 0.6970
Epoch 3/5
387/387 ━━━━━━━━━━━━━━━━━━━━ 21s 53ms/step - accuracy: 0.7891 - loss: 0.5544 - val_accuracy: 0.7078 - val_loss: 0.7494
Epoch 4/5
387/387 ━━━━━━━━━━━━━━━━━━━━ 21s 53ms/step - accuracy: 0.8131 - loss: 0.5011 - val_accuracy: 0.7031 - val_loss: 0.7493
Epoch 5/5
387/387 ━━━━━━━━━━━━━━━━━━━━ 21s 53ms/step - accuracy: 0.8429 - loss: 0.4436 - val_accuracy: 0.6860 - val_loss: 0.8026
```



```python
test_loss, test_accuracy = model_GRU.evaluate(X_test, y_test)
print("Test Accuracy:", test_accuracy)
```

```
111/111 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.7047 - loss: 0.7580
Test Accuracy: 0.706281840801239
```



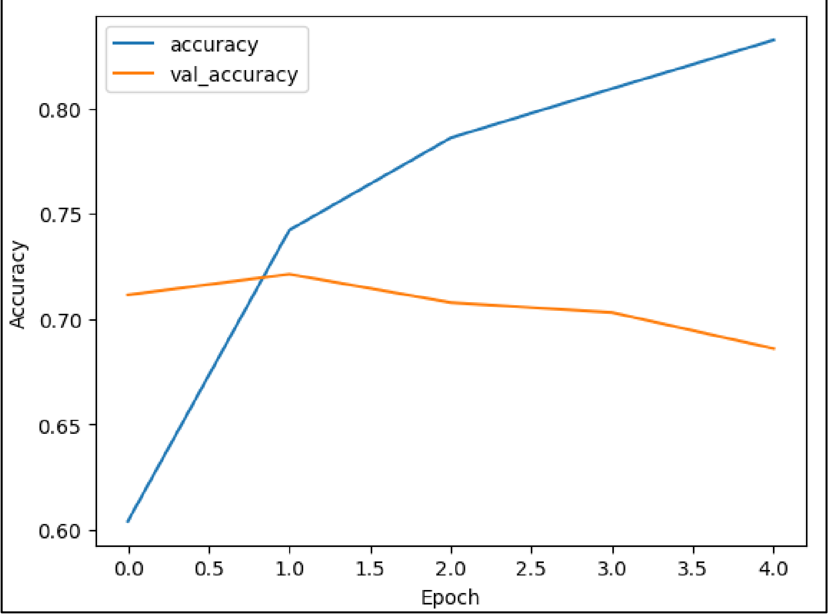
Prediction on custom data

```python
def predict_sentiment(text, model, tokenizer, max_sequence_length):
    sequence = tokenizer.texts_to_sequences([text])
    padded_sequence = pad_sequences(sequence, maxlen=max_sequence_length)
    pred_prob = model.predict(padded_sequence)
    sentiment_label = np.argmax(pred_prob, axis=1)[0]
    if sentiment_label == 0:
        return 'Negative'
    elif sentiment_label == 1:
        return 'Neutral'
    else:
        return 'Positive'

    return sentiment_label
input_text1 = "My best friend surprised me with tickets to my favorite band's concert, and I couldn't be happier."
predicted_sentiment1 = predict_sentiment(input_text, model_GRU, tokenizer, max_sequence_length)
print("Predicted Sentiment:", predicted_sentiment)

input_text2 = "The weather today is neither too hot nor too cold."
predicted_sentiment2 = predict_sentiment(input_text2, model_GRU, tokenizer, max_sequence_length)
print("Predicted Sentiment:", predicted_sentiment2)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
Predicted Sentiment: Positive
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
Predicted Sentiment: Neutral
```

6. Analysis of Model Performance of GRU model

```python
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



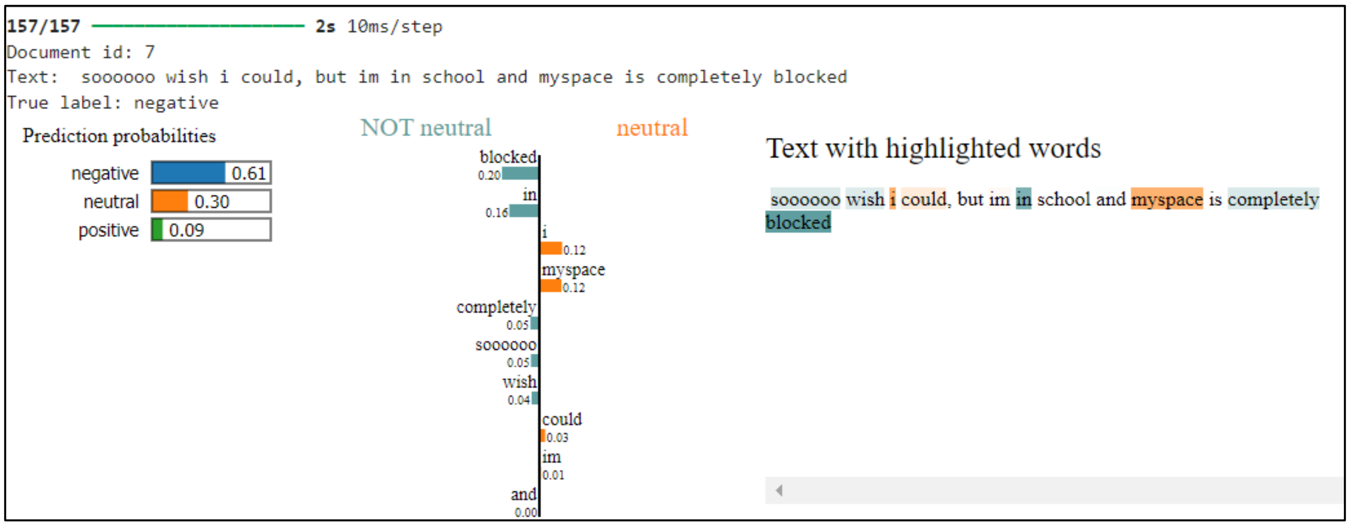
7. Model Interpretation (LIME)

LIME was used to interpret the predictions of the GRU model. LIME provides local explanations for model predictions by perturbing the input data and observing the changes in the output. This helps in understanding which features (words) contributed most to the sentiment prediction for a given text. The explanations provided by LIME were visualized and analyzed to ensure the model's predictions were interpretable and aligned with human intuition.

Code:

```python
from lime.lime_text import LimeTextExplainer
class_names = ['negative', 'neutral', 'positive']
def predict_prob(texts):
    sequences = tokenizer.texts_to_sequences(texts)
    padded = pad_sequences(sequences, maxlen=max_sequence_length)
    return model_GRU.predict(padded)
explainer = LimeTextExplainer(class_names=class_names)
idx = 7
sample_text = test_data['text'].iloc[idx]
exp = explainer.explain_instance(sample_text, predict_prob, num_features=10)
print(f"Document id: {idx}")
print(f"Text: {sample_text}")
print(f"True label: {test_data['sentiment'].iloc[idx]}")
exp.show_in_notebook(text=True)
```

Output:



```
157/157 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step
Document id: 7
Text:  soooooo wish i could, but im in school and myspace is completely blocked
True label: negative
```

8. Conclusion

This project demonstrated the application of various machine learning models for sentiment analysis. The preprocessing steps ensured that the text data was clean and standardized, which is crucial for training effective models. EDA provided valuable insights into the dataset, while the implementation of different models showcased their strengths and weaknesses in sentiment classification. LIME helped in interpreting the model predictions, making the results more transparent and trustworthy. Overall, the project highlighted the importance of combining data preprocessing, robust modeling, and interpretability techniques to build reliable sentiment analysis systems.