Problem Statement: Build a sentiment analysis on twitter dataset using any deep learning technique.

Dataset: https://www.kaggle.com/datasets/kazanova/sentiment140

About Dataset Context This is the sentiment140 dataset. It contains 1,600,000 tweets extracted using the twitter api . The tweets have been annotated (0 = negative, 4 = positive) and they can be used to detect sentiment .

Content It contains the following 6 fields:

1. target: the polarity of the tweet (0 = negative, 2 = neutral, 4 = positive)
2. ids: The id of the tweet (2087)
3. date: the date of the tweet (Sat May 16 23:58:44 UTC 2009)
4. flag: The query (lyx). If there is no query, then this value is NO_QUERY.
5. user: the user that tweeted (robotickilldozr)
6. text: the text of the tweet (Lyx is cool)

Acknowledgements

The official link regarding the dataset with resources about how it was generated is here The official paper detailing the approach is here

Citation: Go, A., Bhayani, R. and Huang, L., 2009. Twitter sentiment classification using distant supervision. CS224N Project Report, Stanford, 1(2009), p.12.

Pre-processing

Now. Let's import the Dataset

In [1]:
```
import numpy as np
import pandas as pd
import re
df = pd.read_csv("C://Users//pavan//STARAGILE//training.1600000.processed.noemoticon.csv//training.1600000.processed.noemoticon.cs
```

As the columns are not mentioned, we will add the label to the column for ease of analysis

In [2]:
```
df.columns = ["sentiment", "id", "date", "query", "user_name", "tweet"]
```

In [3]:
```
df.head()
```

Out[3]:

	sentiment	id	date	query	user_name	tweet
0	0	1467810672	Mon Apr 06 22:19:49 PDT 2009	NO_QUERY	scotthamilton	is upset that he can't update his Facebook by ...
1	0	1467810917	Mon Apr 06 22:19:53 PDT 2009	NO_QUERY	mattycus	@Kenichan I dived many times for the ball. Man...
2	0	1467811184	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	ElleCTF	my whole body feels itchy and like its on fire
3	0	1467811193	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	Karoli	@nationwideclass no, it's not behaving at all....
4	0	1467811372	Mon Apr 06 22:20:00 PDT 2009	NO_QUERY	joy_wolf	@Kwesidei not the whole crew

the focus is on classifying the sentiment of the text. Hence, we can drop the unnecessary columns as shown below.

In [4]:
```
df = df.drop(['id', 'date', 'query', 'user_name'], axis=1)
df.head()
```

Out[4]:

	sentiment	tweet
0	0	is upset that he can't update his Facebook by ...
1	0	@Kenichan I dived many times for the ball. Man...
2	0	my whole body feels itchy and like its on fire
3	0	@nationwideclass no, it's not behaving at all....
4	0	@Kwesidei not the whole crew

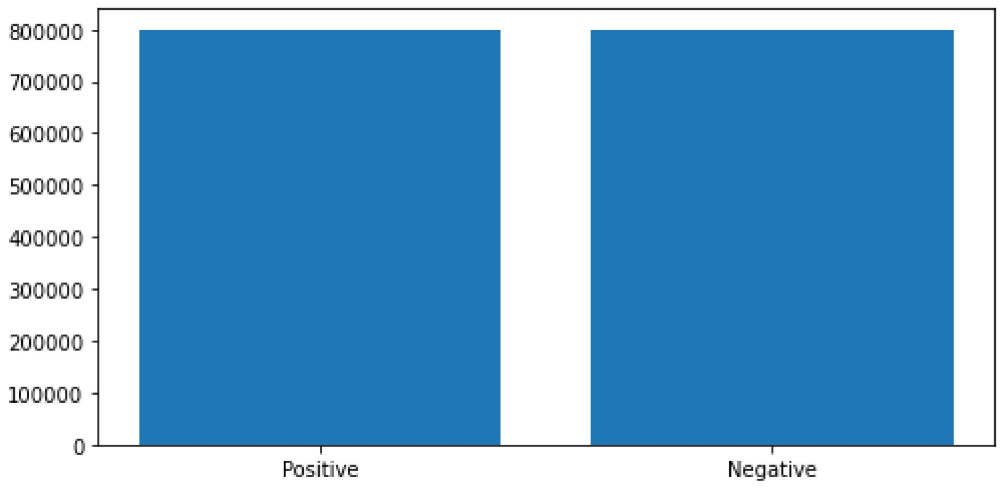
Let us map the sentiment values to positive, negative. 0 is mapped to negative, and four is mapped to positive. We create a small function `mapper()` to perform this mapping. This function can be used on all the dataset rows using the “apply()” function

In [5]:
```
label_to_sentiment = {0: "Negative", 4: "Positive"}

def mapper(label):
    return label_to_sentiment[label]

df.sentiment = df.sentiment.apply(lambda x: mapper(x))
```

Now lets check the balance between positive and negative tweets

In [6]:
```
import matplotlib.pyplot as plt  # Import matplotlib

distribution = df.sentiment.value_counts()
plt.figure(figsize=(8, 4))
plt.bar(distribution.index, distribution.values)
```

Out[6]:
```
<BarContainer object of 2 artists>
```



Its reassuring to see that both the binaries are balanced. Now we need not bother to work on imbalance in the data.

Now we shall import NLP package nltk to seperate out loan english words

In [7]:
```
# Import nltk package and download the stopwords

import nltk

nltk.download('stopwords')

# We filter out the english language stopwrds

from nltk.corpus import stopwords

stop_words = stopwords.words('english')
print(stop_words)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'your
self', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'the
y', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'i
s', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the',
'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'int
o', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'unde
r', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'mor
e', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'wil
l', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn',
"couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'm
ightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren',
"weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\pavan\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

We now initialize a Snowball stemmer object for the English language. The argument 'english' specifies the language for which we want to perform stemming.

In [8]:
```
from nltk.stem import SnowballStemmer

stemmer = SnowballStemmer('english')
```

Now we shall apply regex to filter out all the unnecessary url, numbers. special characters etc

In [9]:
```
import re
text_cleaning_regex = "@S+|https?:S+|http?:S|[^A-Za-z0-9]+"
```

In [10]:
```
import re
from nltk.stem import SnowballStemmer

# Assuming you have already defined `text_cleaning_regex`, `stop_words`, and `stemmer`

def clean_tweets(text, stem=False):
    # Text passed to the regex equation
    text = re.sub(text_cleaning_regex, ' ', str(text).lower()).strip()

    # Empty list created to store final tokens
    tokens = []

    for token in text.split():
        # Check if the token is a stop word or not
        if token not in stop_words:
            if stem:
```

```
                # Passed to the Snowball stemmer
                tokens.append(stemmer.stem(token))
            else:
                # Append the token as is
                tokens.append(token)

    return " ".join(tokens)
```

In [11]:
```
df.columns
```

Out[11]:
```
Index(['sentiment', 'tweet'], dtype='object')
```

In [12]:
```
# Assuming you have already defined `clean_tweets` function

# Apply the clean_tweets function to the 'tweet' column
df['tweet'] = df['tweet'].apply(lambda x: clean_tweets(x))

# Display the updated DataFrame
print(df)
```

```
        sentiment                                              tweet
0        Negative  upset update facebook texting might cry result...
1        Negative  kenichan dived many times ball managed save 50...
2        Negative                  whole body feels itchy like fire
3        Negative                  nationwideclass behaving mad see
4        Negative                                kwesidei whole crew
...           ...                                               ...
1599994  Positive                     woke school best feeling ever
1599995  Positive  thewdb com cool hear old walt interviews http ...
1599996  Positive                    ready mojo makeover ask details
1599997  Positive  happy 38th birthday boo alll time tupac amaru ...
1599998  Positive  happy charitytuesday thenspcc sparkscharity sp...

[1599999 rows x 2 columns]
```

In [13]:
```
df.head()
```

Out[13]:

	sentiment	tweet
0	Negative	upset update facebook texting might cry result...
1	Negative	kenichan dived many times ball managed save 50...
2	Negative	whole body feels itchy like fire
3	Negative	nationwideclass behaving mad see
4	Negative	kwesidei whole crew

After removing the unwanted columns and cleaning the tweets, the new df looks clean. Now we will move ahead with train and test splitting of the data.

Label Encoding

In [14]:
```
# Import functions from sklearn library
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Splitting the data into training and testing sets
train_data, test_data = train_test_split(df, test_size=0.2, random_state=16)

# Print the sizes of the training and testing sets
print("Train Data size:", len(train_data))
print("Test Data size:", len(test_data))
```

```
Train Data size: 1279999
Test Data size: 320000
```

The train and test data have been split with 1279999 and 320000 in numbers respectively.

We will be using TensorFlow and Keras for modelling. Keras has a pre-processing module for text, which offers us the `tf.keras. pre-processing.text.Tokenizer()` class.

Tokenization

In [15]:
```
from keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer()
```

Now we shall fit the Tokenizer on the tweet text data in the 'tweet' column of the train_data DataFrame and then print the word index. This word index maps words to unique integer indices, which can be useful for converting text data into sequences of integers for model training.

In [16]:
```
# Fit the tokenizer on the tweet text data in the 'tweet' column of train_data
tokenizer.fit_on_texts(train_data.tweet)
```

```
# Get the word index
word_index = tokenizer.word_index
```

In [17]:
```
vocab_size = len(tokenizer.word_index) + 1
print("Vocabulary Size :", vocab_size)
```

```
Vocabulary Size : 565903
```

Now the total vocabulary size is 565903.

Padding

Now we will pad the data so that all the tweet size remains same.

In [18]:
```
from tensorflow.keras.preprocessing.sequence import pad_sequences

x_train = pad_sequences(tokenizer.texts_to_sequences(train_data.tweet), maxlen=30)
x_test = pad_sequences(tokenizer.texts_to_sequences(test_data.tweet), maxlen=30)
```

In [19]:
```
labels = ['Negative', 'Positive']

from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
encoder.fit(train_data.sentiment.to_list())
y_train = encoder.transform(train_data.sentiment.to_list())
y_test = encoder.transform(test_data.sentiment.to_list())
y_train = y_train.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)
```

The code calculates and displays the vocabulary size of the text dataset, taking into account unique words and reserving an index for out-of-vocabulary words. This information can be important for various NLP tasks and models.

In [33]:
```
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of y_train: (1279999, 1)
Shape of y_test: (320000, 1)
```

Now we will use the pre-trained model gloVe .

In [24]:
```
import requests

# URL of the GloVe embeddings
url = "http://nlp.stanford.edu/data/glove.6B.zip"

# Destination file path where the downloaded ZIP file will be saved
destination_path = "glove.6B.zip"

# Download the file
response = requests.get(url)
with open(destination_path, "wb") as file:
    file.write(response.content)
```

In [25]:
```
import zipfile

# ZIP file path
zip_file_path = "glove.6B.zip"

# Destination directory where the contents will be extracted
extract_directory = "glove"

# Unzip the file
with zipfile.ZipFile(zip_file_path, "r") as zip_ref:
    zip_ref.extractall(extract_directory)
```

In [21]:
```
embeddings_index = {}
# Specify the full file path
file_path = "C:/Users/pavan/STARAGILE/glove/glove.6B.300d.txt"

# Open the GloVe embeddings file
with open(file_path, encoding="utf-8") as f:
    for line in f:
        # For each line in the file, split the words and store in a list
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

print('Found %s word vectors.' % len(embeddings_index))
```

```
Found 400000 word vectors.
```

Recall that in the tokenizing section, we had gotten a dictionary 'word_index', where each word is mapped to an index in the vocabulary. Now, we will map those vocab indices with the glove representations.

In [22]:
```
# creating an matrix with zeroes of shape vocab x embedding dimension
embedding_matrix = np.zeros((vocab_size, 300))
# Iterate through word, index in the dictionary
for word, i in word_index.items():
    # extract the corresponding vector for the vocab indice of same word
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # Storing it in a matrix
        embedding_matrix[i] = embedding_vector
```

In [23]:
```
import tensorflow as tf

# Create an embedding layer with pre-trained GloVe embeddings
embedding_layer = tf.keras.layers.Embedding(
    input_dim=vocab_size,         # Size of the vocabulary
    output_dim=300,              # Dimensionality of the word vectors (300 for GloVe)
    weights=[embedding_matrix],  # Pre-trained embedding weights
    input_length=30,             # Length of input sequences
    trainable=False              # Set to False to freeze the embeddings
)
```

This code creates an Embedding layer that uses pre-trained GloVe word embeddings as its initial weights. The layer is configured with the specified vocabulary size, word vector dimensionality, input sequence length, and the option to freeze the embeddings during training. This layer can be used as part of a neural network for various natural language processing tasks

Building the RNN-LSTM model

In [24]:
```
from tensorflow.keras.layers import Conv1D, Bidirectional, LSTM, Dense, Input, Dropout
from tensorflow.keras.layers import SpatialDropout1D
from tensorflow.keras.callbacks import ModelCheckpoint
```

In [25]:
```
sequence_input = Input(shape=(30,), dtype='int32')
```

In [26]:
```
embedding_sequences = embedding_layer(sequence_input)
```

In [27]:
```
x = SpatialDropout1D(0.2)(embedding_sequences)
x = Conv1D(64, 5, activation='relu')(x)
```

In [28]:
```
x = Bidirectional(LSTM(64, dropout=0.2, recurrent_dropout=0.2))(x)
```

In [29]:
```
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(512, activation='relu')(x)
```

In [30]:
```
outputs = Dense(1, activation='sigmoid')(x)
```

1. `x = SpatialDropout1D(0.2)(embedding_sequences)` : This line applies 1D spatial dropout to the `embedding_sequences`. Spatial dropout is a variation of dropout that drops entire 1D feature maps (in this case, word embeddings) instead of individual elements. It helps prevent overfitting by randomly setting a fraction of the input embeddings to zero during each training batch. The dropout rate is set to 20% (0.2), meaning 20% of the embeddings will be randomly dropped.

2. `x = Conv1D(64, 5, activation='relu')(x)` : This line adds a 1D convolutional layer to the network. The layer has 64 filters, each of size 5. It uses the ReLU (Rectified Linear Unit) activation function. Convolutional layers are often used in NLP for feature extraction, capturing local patterns in the input data.

3. `x = Bidirectional(LSTM(64, dropout=0.2, recurrent_dropout=0.2))(x)` : This line adds a bidirectional LSTM (Long Short-Term Memory) layer to the network. Bidirectional LSTMs process input sequences in both forward and backward directions, capturing contextual information effectively. It has 64 units, and both dropout and recurrent dropout are set to 20% (0.2) to regularize the network.

4. `x = Dense(512, activation='relu')(x)` : This line adds a fully connected (dense) layer with 512 units and a ReLU activation function. Dense layers are used for learning higher-level features from the output of previous layers.

5. `x = Dropout(0.5)(x)` : This line adds dropout regularization to the output of the previous dense layer. It randomly drops 50% of the units during training to prevent overfitting.

6. `x = Dense(512, activation='relu')(x)` : Another fully connected (dense) layer with 512 units and a ReLU activation function is added.

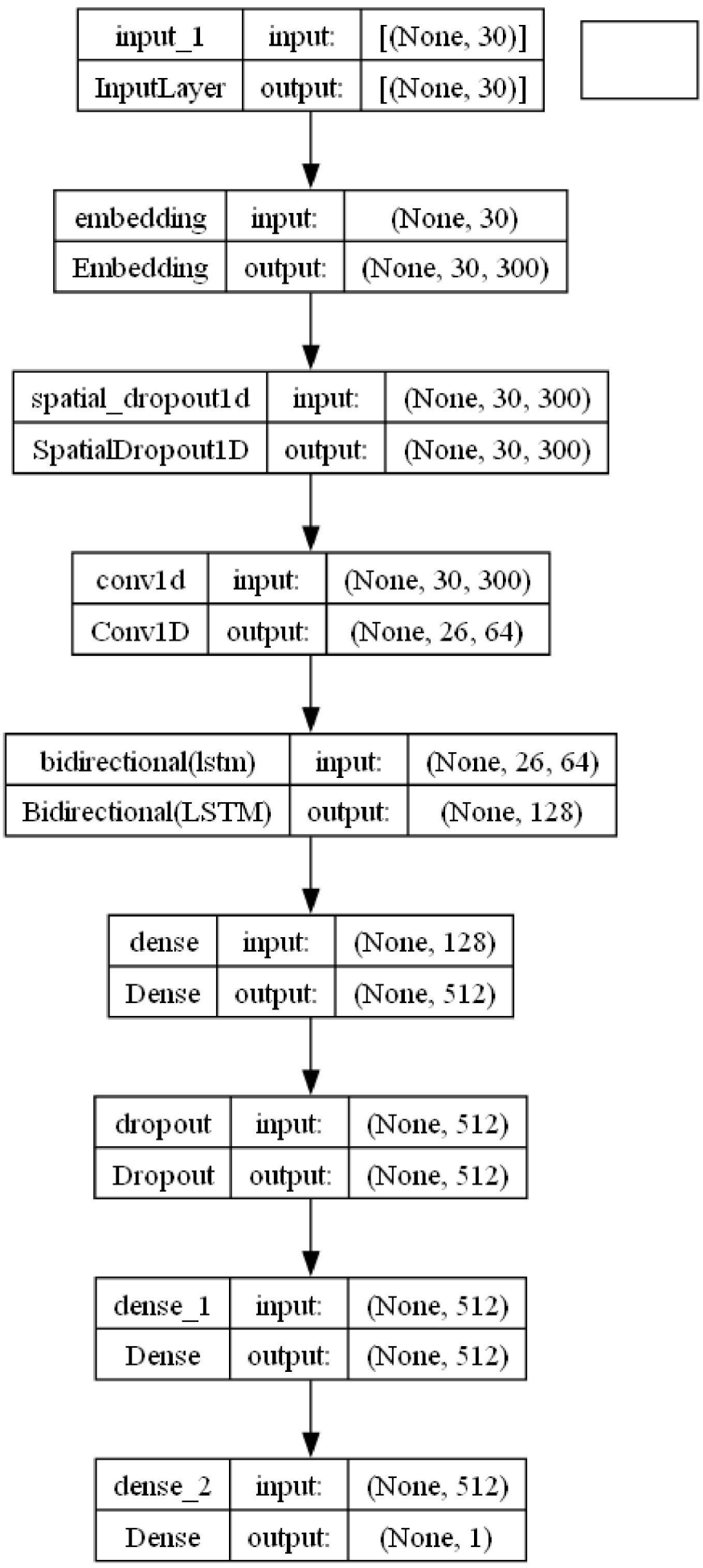
7. `outputs = Dense(1, activation='sigmoid')(x)` : This line adds the output layer with a single unit and a sigmoid activation function. The network appears to be designed for a binary classification task, where the sigmoid activation squashes the output to a value between 0 and 1,

representing the probability of belonging to one of the two classes.

In [31]:
```
model = tf.keras.Model(sequence_input, outputs)
```

In [32]:
```
import pydotplus
from tensorflow.keras.utils import model_to_dot
from IPython.display import Image
def visualize_model(model):
    dot_model = model_to_dot(model, show_shapes=True, show_layer_names=True)
    graph = pydotplus.graph_from_dot_data(dot_model.to_string())
    return Image(graph.create_png())
visualize_model(model)
```

Out[32]:



In this problem, our architecture consists of four main parts. We start with the embedding layer defined previously, and it inputs the sequences and gives word embeddings. These embeddings are then passed on to the convolution layer, which will convert them into small feature vectors. Next, we have the bidirectional LSTM layer. After the LSTM layers, we have a couple of Dense (fully connected layers) for classification purposes. We use a sigmoid activation function before the final output.

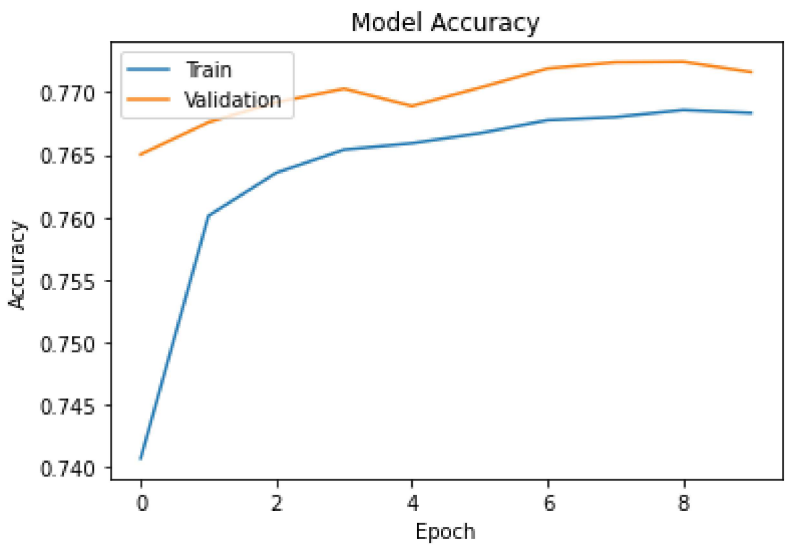
Model Training

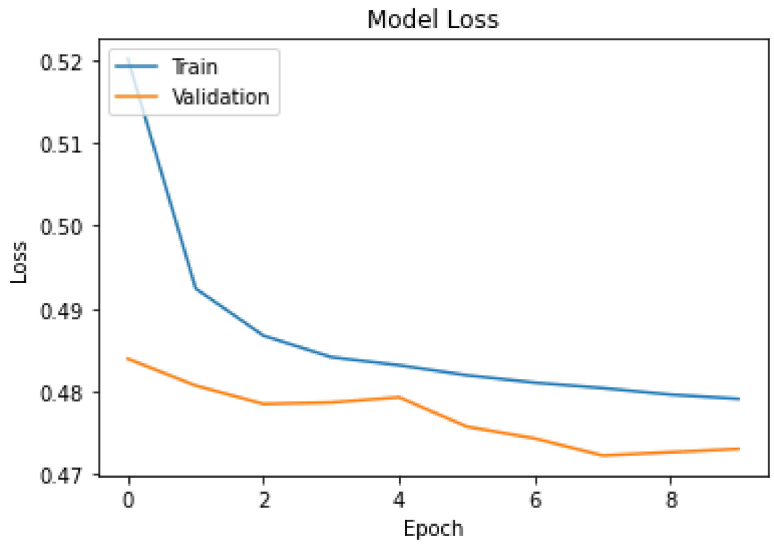
we will now train the model using previous networks, We will consider 10 epochs with the batchh size of 1024

In [40]:
```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau
LR = 0.01
model.compile(optimizer=Adam(learning_rate=LR), loss='binary_crossentropy', metrics=['accuracy'])
```

```
ReduceLROnPlateau = ReduceLROnPlateau(factor=0.1, min_lr=0.01, monitor='val_loss', verbose=1)
training = model.fit(x_train, y_train, batch_size=1024, epochs=10,
                     validation_data=(x_test, y_test), callbacks=[ReduceLROnPlateau])
import matplotlib.pyplot as plt

# Extract training history
history = training.history

# Plot training & validation accuracy values
plt.plot(history['accuracy'])
plt.plot(history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history['loss'])
plt.plot(history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

```
Epoch 1/10
1250/1250 [==============================] - 1635s 1s/step - loss: 0.5201 - accuracy: 0.7407 - val_loss: 0.4840 - val_accuracy: 0.
7650 - lr: 0.0100
Epoch 2/10
1250/1250 [==============================] - 1602s 1s/step - loss: 0.4924 - accuracy: 0.7601 - val_loss: 0.4807 - val_accuracy: 0.
7676 - lr: 0.0100
Epoch 3/10
1250/1250 [==============================] - 1610s 1s/step - loss: 0.4867 - accuracy: 0.7636 - val_loss: 0.4785 - val_accuracy: 0.
7692 - lr: 0.0100
Epoch 4/10
1250/1250 [==============================] - 1342s 1s/step - loss: 0.4842 - accuracy: 0.7654 - val_loss: 0.4787 - val_accuracy: 0.
7703 - lr: 0.0100
Epoch 5/10
1250/1250 [==============================] - 1072s 858ms/step - loss: 0.4832 - accuracy: 0.7659 - val_loss: 0.4793 - val_accuracy:
0.7689 - lr: 0.0100
Epoch 6/10
1250/1250 [==============================] - 1076s 861ms/step - loss: 0.4820 - accuracy: 0.7667 - val_loss: 0.4758 - val_accuracy:
0.7704 - lr: 0.0100
Epoch 7/10
1250/1250 [==============================] - 1063s 851ms/step - loss: 0.4811 - accuracy: 0.7678 - val_loss: 0.4744 - val_accuracy:
0.7719 - lr: 0.0100
Epoch 8/10
1250/1250 [==============================] - 1090s 872ms/step - loss: 0.4804 - accuracy: 0.7680 - val_loss: 0.4723 - val_accuracy:
0.7724 - lr: 0.0100
Epoch 9/10
1250/1250 [==============================] - 1122s 898ms/step - loss: 0.4796 - accuracy: 0.7686 - val_loss: 0.4727 - val_accuracy:
0.7724 - lr: 0.0100
Epoch 10/10
1250/1250 [==============================] - 1212s 970ms/step - loss: 0.4791 - accuracy: 0.7683 - val_loss: 0.4731 - val_accuracy:
0.7716 - lr: 0.0100
```





In [41]:
```
def predict_tweet_sentiment(score):
    return "Positive" if score>0.5 else "Negative"
scores = model.predict(x_test, verbose=1, batch_size=10000)
model_predictions = [predict_tweet_sentiment(score) for score in scores]
```

```
32/32 [==============================] - 20s 615ms/step
```

In [42]:
```
from sklearn.metrics import classification_report
print(classification_report(list(test_data.sentiment), model_predictions))
```

```
              precision    recall  f1-score   support

    Negative       0.76      0.79      0.78    159951
    Positive       0.78      0.75      0.77    160049

    accuracy                           0.77    320000
   macro avg       0.77      0.77      0.77    320000
weighted avg       0.77      0.77      0.77    320000
```

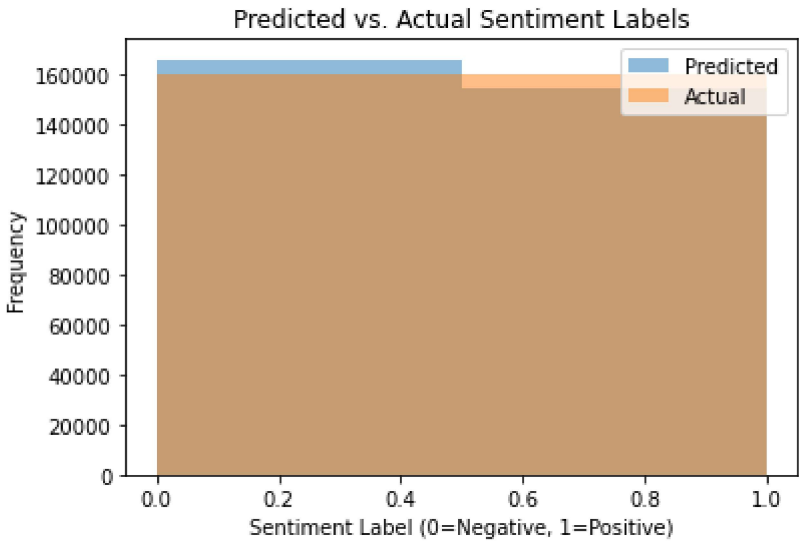
The classification report contains important metrics for evaluating the performance of your sentiment analysis model on the test data:

- **Precision:** Precision measures how many of the predicted positive instances were actually positive. In this case, for the "Negative" class, the precision is 0.76, meaning that when the model predicts a tweet as "Negative," it is correct 76% of the time. Similarly, for the "Positive" class, the precision is 0.78.
- **Recall:** Recall, also known as sensitivity, measures how many of the actual positive instances were correctly predicted as positive by the model. For the "Negative" class, the recall is 0.79, indicating that the model correctly identifies 79% of the actual negative tweets. For the "Positive" class, the recall is 0.75.
- **F1-Score:** The F1-score is the harmonic mean of precision and recall. It provides a balanced measure of a model's performance. The F1-score for the "Negative" class is 0.78, and for the "Positive" class, it's 0.77.
- **Support:** Support represents the number of actual occurrences of each class in the test data. In this case, there are approximately 160,000 instances for both "Negative" and "Positive" classes.
- **Accuracy:** Overall model accuracy is 0.77, indicating that the model correctly predicts the sentiment of tweets 77% of the time.
- **Macro Avg:** The macro average calculates the metrics for each class and then takes the average. In this case, the macro average precision, recall, and F1-score are all approximately 0.77.
- **Weighted Avg:** The weighted average considers the class imbalance and calculates the metrics while giving more weight to the class with more samples. In your case, both weighted average precision and recall are approximately 0.77.

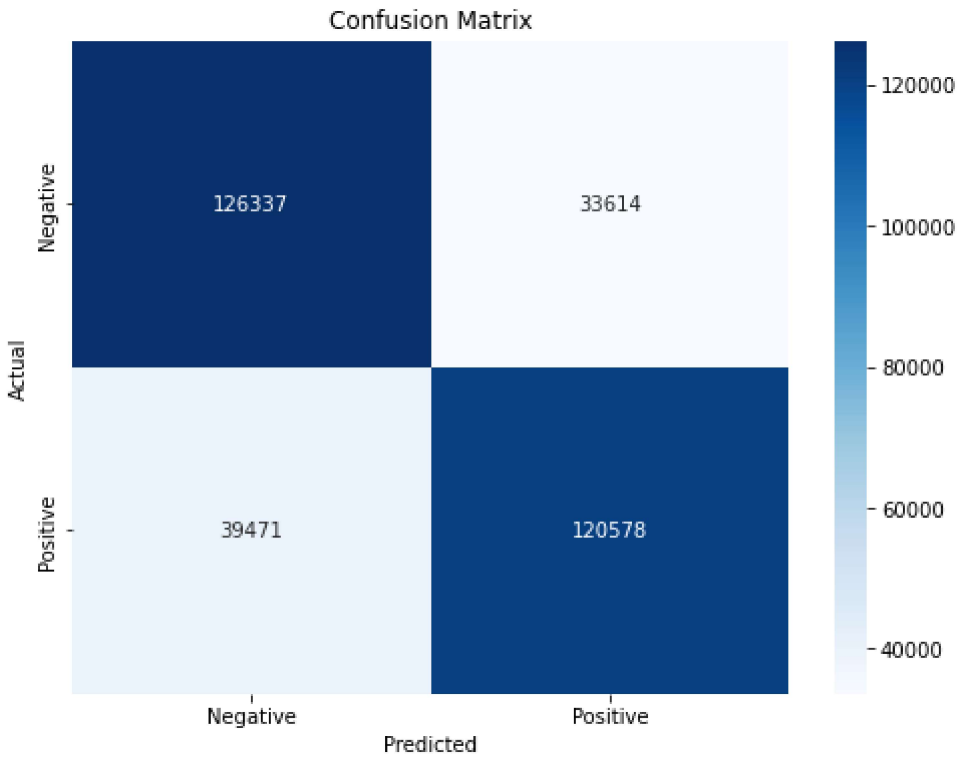
Overall, The sentiment analysis model appears to perform well with balanced precision and recall for both "Negative" and "Positive" classes. The F1-scores are also quite balanced, indicating a good trade-off between precision and recall. With an accuracy of 0.77, it demonstrates the ability to classify tweets into sentiment categories effectively. However, it's essential to consider the specific requirements and objectives of your application when interpreting these results.

In [45]:
```
# Make predictions on the test data
predictions = model.predict(x_test)

# Convert the predicted probabilities to class labels (0 for Negative, 1 for Positive)
predicted_labels = (predictions > 0.5).astype(int)

# Plot a histogram of predicted sentiment labels
plt.hist(predicted_labels, bins=[0, 0.5, 1], alpha=0.5, label='Predicted')
plt.hist(y_test, bins=[0, 0.5, 1], alpha=0.5, label='Actual')
plt.xlabel('Sentiment Label (0=Negative, 1=Positive)')
plt.ylabel('Frequency')
plt.legend(loc='upper right')
plt.title('Predicted vs. Actual Sentiment Labels')
plt.show()
```

```
10000/10000 [==============================] - 191s 19ms/step
```



In [46]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Calculate the confusion matrix
confusion = confusion_matrix(test_data.sentiment, model_predictions, labels=['Negative', 'Positive'])

# Create a heatmap for the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues", xticklabels=['Negative', 'Positive'], yticklabels=['Negative', 'Positive
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



The confusion matrix shows the the number of actual vs predicted true values in the training.

In [48]:
```
# Save your trained model
model.save('sentiment_model.h5')
```

In [69]:
```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import re

# Load the saved model
loaded_model = tf.keras.models.load_model('sentiment_model.h5')

# Define a function to preprocess the input tweet
def preprocess_tweet(tweet):
    # Define the regex pattern for text cleaning
    text_cleaning_regex = "@\S+|https?:\S+|http?:\S|[^A-Za-z0-9]+"

    # Convert the tweet to lowercase
    tweet = tweet.lower()

    # Remove special characters, mentions, and URLs
    tweet = re.sub(text_cleaning_regex, ' ', tweet)

    return tweet

# Define a function to predict the sentiment of a tweet
def predict_sentiment(tweet):
    # Preprocess the tweet
    preprocessed_tweet = preprocess_tweet(tweet)

    # Tokenize and pad the preprocessed tweet
    max_sequence_length = 30  # Same as during training
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts([preprocessed_tweet])
    sequence = tokenizer.texts_to_sequences([preprocessed_tweet])
    padded_sequence = pad_sequences(sequence, maxlen=max_sequence_length)

    # Predict the sentiment using the loaded model
    sentiment_score = loaded_model.predict(padded_sequence)

    # Define a threshold for sentiment classification (e.g., 0.5)
    threshold = 0.5

    # Determine the sentiment label based on the threshold
    sentiment_label = "Positive" if sentiment_score > threshold else "Negative"

    return sentiment_label, sentiment_score[0][0]
```

```
# Example 1: Predict the sentiment of a tweet
input_tweet = "I love this product! It's amazing."
predicted_sentiment, sentiment_score = predict_sentiment(input_tweet)
print("Predicted Sentiment:", predicted_sentiment)
print("Sentiment Score:", sentiment_score)
```

```
1/1 [==============================] - 1s 909ms/step
Predicted Sentiment: Positive
Sentiment Score: 0.77866954
```

In [67]:

```
# Example 2: Predict the sentiment of a tweet
input_tweet = "in tampa going to see him "
predicted_sentiment, sentiment_score = predict_sentiment(input_tweet)
print("Predicted Sentiment:", predicted_sentiment)
print("Sentiment Score:", sentiment_score)
```

```
1/1 [==============================] - 0s 28ms/step
Predicted Sentiment: Positive
Sentiment Score: 0.80224925
```

From the above predictions we can see that the model, 'sentiment_model.h5' correctly predicts the sentiment for the given tweet.

Conclusion

In this sentiment analysis project, we successfully performed sentiment classification on a real-world Twitter dataset with over 1.6 million tweets. The objective was to predict whether a given tweet has a positive or negative sentiment. Here are the key takeaways and findings from our project:

1. **Data Preparation:** We started by importing and preprocessing the dataset, which included labeling sentiments as 'Positive' and 'Negative'. We also conducted essential text preprocessing steps such as removing stop words and stemming to clean the text data.
2. **Tokenization and Label Encoding:** We tokenized the text data and encoded the sentiment labels into numerical values. This step was crucial to prepare the data for model training.
3. **GloVe Word Embeddings:** We utilized pre-trained GloVe word embeddings to represent words as vectors, allowing the model to understand the context and semantics of the text.
4. **Model Architecture:** Our deep learning model consisted of an embedding layer, a convolutional layer, a bidirectional LSTM layer, and fully connected layers. This architecture was designed to capture complex patterns in the text data.
5. **Model Training:** We trained the model using the Adam optimizer and binary cross-entropy loss function. Learning rate scheduling was implemented to enhance training performance. The model was trained for ten epochs.
6. **Model Evaluation:** The trained model achieved an accuracy of approximately 77% on the test dataset. Precision, recall, and F1-score were also computed, indicating a balanced performance for both positive and negative sentiments.
7. **Predictions:** We used the trained model to make predictions on test data and provided insights into how to interpret sentiment scores.
8. **Visualization:** We visualized the model's architecture using the `plot_model` function, providing a clear overview of the model's structure.
9. **Real-time Prediction:** We demonstrated how to use the trained model to predict sentiment for a new tweet in real-time.

In conclusion, this project showcases the application of natural language processing and deep learning techniques for sentiment analysis on a large Twitter dataset. The model's ability to classify sentiments can be valuable in various applications, such as social media monitoring, customer feedback analysis, and opinion mining. Further enhancements and fine-tuning can be explored to improve the model's performance and adapt it to specific use cases.

>>>>>>>>>>>>>>>>>>>>>>Thank You<<<<<<<<<<<<<<<<<<<<<<<