One of the most impactful small projects for learning NLP is building a Sentiment Analysis Tool. This project covers a range of foundational NLP concepts and techniques, making it an excellent starting point for beginners.

Why Sentiment Analysis?

Broad Application: Sentiment analysis is used in various domains, from business (product reviews, customer feedback) to politics (public opinion on policies or candidates). Covers Basic NLP Tasks: You'll encounter text preprocessing, tokenization, feature extraction, and classification. Scalability: You can start simple and then make the project more complex as you learn, such as handling multi-class sentiments, sarcasm detection, or analyzing sentiment over time.

Steps to Build a Sentiment Analysis Tool:

Data Collection:

Use datasets available online, such as the IMDb movie reviews dataset or Twitter sentiment analysis datasets. Alternatively, web scrape reviews from sites like Amazon or Yelp using tools like Beautiful Soup or Scrapy in Python. Text Preprocessing:

Tokenization: Split text into words or subwords. Lowercasing: Convert all characters in the text to lowercase to maintain consistency. Stopword Removal: Remove common words (e.g., "and", "the") that don't contribute to sentiment. Stemming/Lemmatization: Reduce words to their base or root form. Feature Extraction:

Bag of Words (BoW): Represent text based on the frequency of words. TF-IDF (Term Frequency-Inverse Document Frequency): Weigh words based on their importance in the document relative to a larger corpus. Word Embeddings: Use pre-trained models like Word2Vec or GloVe to convert words into vectors. Model Building:

Start with basic models like Logistic Regression or Naive Bayes. As you progress, experiment with more complex models like Support Vector Machines, Random Forests, or Neural Networks. Evaluation:

Split your data into training and testing sets. Train your model on the training set and evaluate its performance on the test set using metrics like accuracy, precision, recall, and F1-score. Deployment (Optional):

Turn your model into a web application using frameworks like Flask or Streamlit. Deploy it online so users can input text and get sentiment predictions in real-time. Iterate and Improve:

Use more advanced techniques like handling class imbalance or using deep learning models like RNNs or Transformers. Incorporate additional features like n-grams, sentiment lexicons, or syntactic features.

Use Pre-existing Datasets: There are several publicly available datasets tailored for sentiment analysis. Here are a few:

IMDb Movie Reviews: This dataset contains 50,000 movie reviews labeled as positive or negative. Link to dataset http://ai.stanford.edu/~amaas/data/sentiment/

In [1]:
```
import os
import pandas as pd
import re
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

# Load IMDb Data
def load_imdb_data(path):
    reviews = []
    labels = []

    for data_type in ['train', 'test']:
        for sentiment in ['pos', 'neg']:
            dir_name = os.path.join(path, data_type, sentiment)
            for filename in os.listdir(dir_name):
                if filename.endswith('.txt'):
                    with open(os.path.join(dir_name, filename), 'r', encoding='utf-
                        reviews.append(f.read())
                        labels.append(sentiment)

    df = pd.DataFrame({
        'review': reviews,
        'label': labels
    })

    return df

# Preprocess Text
def preprocess_text(text):
    text = re.sub('<.*?>', '', text)
    text = text.lower()
    text = re.sub('[^a-zA-Z]', ' ', text)

    stemmer = PorterStemmer()
    words = text.split()
    words = [stemmer.stem(word) for word in words if word not in set(stopwords.word
    text = ' '.join(words)

    return text

# Load the data
path_to_dataset = 'C:\\Users\\Kyle\\Documents\\Data Science Projects\\aclImdb'
df = load_imdb_data(path_to_dataset)

# Explore the data
print(df.head())
print(df['label'].value_counts())

# Preprocess the reviews
```

```
df['review'] = df['review'].apply(preprocess_text)
print("After preprocessing:")
print(df.head())
```

```
                                              review label
0  Bromwell High is a cartoon comedy. It ran at t...   pos
1  Homelessness (or Houselessness as George Carli...   pos
2  Brilliant over-acting by Lesley Ann Warren. Be...   pos
3  This is easily the most underrated film inn th...   pos
4  This is not the typical Mel Brooks film. It wa...   pos
label
pos    25000
neg    25000
Name: count, dtype: int64
After preprocessing:
                                              review label
0  bromwel high cartoon comedi ran time program s...   pos
1  homeless houseless georg carlin state issu yea...   pos
2  brilliant act lesley ann warren best dramat ho...   pos
3  easili underr film inn brook cannon sure flaw ...   pos
4  typic mel brook film much less slapstick movi ...   pos
```

The data has been successfully loaded and preprocessed. The dataset is balanced with 25,000 positive and 25,000 negative reviews.

Now, let's move on to the next steps:

1. Feature Extraction

Before feeding the reviews into a machine learning model, we need to convert the text data into a numerical format. One common method is the Bag of Words (BoW) representation using the CountVectorizer from scikit-learn.

2. Splitting the Data

We'll split the data into training and testing sets to evaluate the performance of our model.

3. Model Building

For simplicity, we'll start with a Logistic Regression model, which is a commonly used algorithm for binary classification tasks like sentiment analysis.

4. Model Evaluation

After training, we'll evaluate the model on the test set using accuracy and other metrics.

Here's the next step:

This script will train a Logistic Regression model on the training data and evaluate its performance on the test data. The classification report will provide detailed metrics like precision, recall, and F1-score for both positive and negative classes.

In [2]:
```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# 1. Feature Extraction
vectorizer = CountVectorizer(max_features=5000)  # Limiting to 5000 most frequent w
X = vectorizer.fit_transform(df['review'])
y = df['label']

# 2. Splitting the Data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# 3. Model Building
clf = LogisticRegression(max_iter=1000)  # Increased max_iter for convergence
clf.fit(X_train, y_train)

# 4. Model Evaluation
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy*100:.2f}%")
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 87.41%

Classification Report:
               precision    recall  f1-score   support

         neg       0.87      0.87      0.87      4945
         pos       0.87      0.88      0.88      5055

    accuracy                           0.87     10000
   macro avg       0.87      0.87      0.87     10000
weighted avg       0.87      0.87      0.87     10000
```

Interpretation

An accuracy of 87.41% with a balanced precision, recall, and F1-score for both classes indicates that the model is performing well on the IMDb dataset.

Step 1: Feature Extraction using TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure that evaluates the importance of a word in a document relative to a collection of documents (corpus). Words that are frequent in a document but not across documents receive a higher score.

In [3]:
```
#Python Code for TF-IDF Feature Extraction:
from sklearn.feature_extraction.text import TfidfVectorizer

# Initialize the TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=5000)

# Fit and transform the reviews
```

```
X_tfidf = tfidf_vectorizer.fit_transform(df['review'])
y = df['label']

# Split the data into training and testing sets
X_train_tfidf, X_test_tfidf, y_train, y_test = train_test_split(X_tfidf, y, test_si
```

Step 2: Model Building using Support Vector Machine (SVM)

Support Vector Machines are supervised learning models used for classification. They work well for high-dimensional data, making them suitable for text data.

In [4]:
```
#Python Code for SVM Model Building:
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, classification_report

# Initialize and train the SVM classifier
svm_clf = LinearSVC(max_iter=10000)
svm_clf.fit(X_train_tfidf, y_train)

# Predict the sentiments for the test set
y_pred_svm = svm_clf.predict(X_test_tfidf)

# Evaluate the SVM model
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print(f"SVM Accuracy: {accuracy_svm*100:.2f}%")
print("\nSVM Classification Report:\n", classification_report(y_test, y_pred_svm))
```

```
SVM Accuracy: 88.24%

SVM Classification Report:
               precision    recall  f1-score   support

         neg       0.88      0.88      0.88      4945
         pos       0.88      0.89      0.88      5055

    accuracy                           0.88     10000
   macro avg       0.88      0.88      0.88     10000
weighted avg       0.88      0.88      0.88     10000
```

Interpretation

The SVM model with TF-IDF representation has slightly improved the performance compared to the Logistic Regression model with Bag of Words representation. An accuracy of 88.24% is commendable, and the classification report indicates a balanced performance for both positive and negative classes.

Observation

Accuracy: The SVM model achieved an accuracy of 88.24%, which is an improvement over the 87.41% achieved by the Logistic Regression model.

Precision, Recall, and F1-Score: Both positive and negative classes have similar and balanced scores, which is ideal. This means the model is not biased towards any particular class.

Next Step: Hyperparameter tuning

Hyperparameter tuning is the process of systematically searching for the best combination of hyperparameters that will optimize a model's performance.

For the SVM model, some of the key hyperparameters include:

C: Regularization parameter. The strength of the regularization is inversely proportional to C. Smaller values specify stronger regularization. loss: Specifies the loss function. For LinearSVC, the options are 'hinge' and 'squared_hinge'. max_iter: The maximum number of iterations to be run. For the TfidfVectorizer, some hyperparameters to consider are:

max_features: The maximum number of terms to consider. ngram_range: The range of n-grams to include. stop_words: Whether to remove stop words. We'll use GridSearchCV from scikit-learn to perform an exhaustive search over the specified hyperparameter values.

In [5]:
```
from sklearn.model_selection import GridSearchCV

# Define the hyperparameters and their possible values
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'loss': ['hinge', 'squared_hinge'],
    'max_iter': [1000, 5000, 10000]
}

# Initialize the GridSearchCV object
grid_search = GridSearchCV(LinearSVC(), param_grid, cv=5, verbose=2, n_jobs=-1)

# Fit the GridSearchCV object to the data
grid_search.fit(X_train_tfidf, y_train)

# Print the best hyperparameters
print("Best Hyperparameters:", grid_search.best_params_)

# Evaluate the best model on the test data
best_svm = grid_search.best_estimator_
```

```
y_pred_best_svm = best_svm.predict(X_test_tfidf)
accuracy_best_svm = accuracy_score(y_test, y_pred_best_svm)
print(f"Best SVM Accuracy: {accuracy_best_svm*100:.2f}%")
print("\nBest SVM Classification Report:\n", classification_report(y_test, y_pred_b
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
Best Hyperparameters: {'C': 0.1, 'loss': 'squared_hinge', 'max_iter': 1000}
Best SVM Accuracy: 89.01%

Best SVM Classification Report:
               precision    recall  f1-score   support

         neg       0.90      0.88      0.89      4945
         pos       0.88      0.90      0.89      5055

    accuracy                           0.89     10000
   macro avg       0.89      0.89      0.89     10000
weighted avg       0.89      0.89      0.89     10000
```

Interpretation

Hyperparameter tuning has further improved the performance of your SVM model. The accuracy has increased to 89.01%, which is a notable improvement from the initial 88.24%.

Observations: Accuracy: The tuned SVM model achieved an accuracy of 89.01%, which is an improvement over the initial SVM model's 88.24%.

Precision, Recall, and F1-Score: The classification report indicates that the model has a balanced performance for both positive and negative classes. Both precision and recall are close to 0.89 for both classes, which is ideal.

Best Hyperparameters: The best hyperparameters for the SVM model were found to be C=0.1, loss='squared_hinge', and max_iter=1000.

Next Step: Neural Networks for sentiment analysis

Neural networks, especially Recurrent Neural Networks (RNNs) and their variants like Long Short-Term Memory (LSTM) networks, have shown impressive results on NLP tasks, including sentiment analysis.

Step: Building a Neural Network for Sentiment Analysis

1. Preparing Data for Neural Networks:

Neural networks require numerical input data. For text data, one common approach is to use word embeddings. For this step, we'll use the Tokenizer from Keras to convert text data into sequences of integers. We'll then pad these sequences to a fixed length.

2. Building the Neural Network:

We'll use Keras to build a simple LSTM-based neural network.

3. Training and Evaluating the Model:

We'll train the neural network on the training data and evaluate its performance on the test data.

This script will train a simple LSTM-based neural network on the IMDb dataset. The model uses word embeddings to represent the reviews and then feeds them into LSTM layers.

Note: Training neural networks can be computationally intensive and might take some time, especially if you're not using GPU acceleration.

After training, you can evaluate the model's performance on the test data and compare it with the previous models.

In [6]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adam

# 1. Preparing Data for Neural Networks
tokenizer = Tokenizer(num_words=5000, oov_token="<OOV>")
tokenizer.fit_on_texts(df['review'])
sequences = tokenizer.texts_to_sequences(df['review'])
padded_sequences = pad_sequences(sequences, maxlen=300, truncating='post', padding=

X = np.array(padded_sequences)
y = np.array([1 if label == 'pos' else 0 for label in df['label']])
X_train_nn, X_test_nn, y_train_nn, y_test_nn = train_test_split(X, y, test_size=0.2

# 2. Building the Neural Network with modifications
model = Sequential([
    Embedding(input_dim=5000, output_dim=128, input_length=300),  # Increased embed
    LSTM(128, return_sequences=True),
    LSTM(64),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.0005), met

# 3. Training and Evaluating the Model with more epochs
model.fit(X_train_nn, y_train_nn, epochs=10, batch_size=64, validation_data=(X_test
Epoch 1/10
625/625 [==============================] - 449s 713ms/step - loss: 0.6945 - accurac
y: 0.5042 - val_loss: 0.6922 - val_accuracy: 0.5050
Epoch 2/10
625/625 [==============================] - 425s 681ms/step - loss: 0.6877 - accurac
y: 0.5196 - val_loss: 0.6898 - val_accuracy: 0.5111
Epoch 3/10
625/625 [==============================] - 417s 667ms/step - loss: 0.6755 - accurac
y: 0.5243 - val_loss: 0.6861 - val_accuracy: 0.5152
Epoch 4/10
625/625 [==============================] - 417s 668ms/step - loss: 0.6615 - accurac
y: 0.5900 - val_loss: 0.6183 - val_accuracy: 0.6393
Epoch 5/10
625/625 [==============================] - 423s 676ms/step - loss: 0.6405 - accurac
y: 0.6266 - val_loss: 0.6462 - val_accuracy: 0.5961
Epoch 6/10
625/625 [==============================] - 444s 711ms/step - loss: 0.6724 - accurac
y: 0.5418 - val_loss: 0.6613 - val_accuracy: 0.5564
Epoch 7/10
625/625 [==============================] - 451s 722ms/step - loss: 0.6598 - accurac
y: 0.5689 - val_loss: 0.6659 - val_accuracy: 0.5471
Epoch 8/10
625/625 [==============================] - 463s 741ms/step - loss: 0.6637 - accurac
y: 0.5602 - val_loss: 0.6628 - val_accuracy: 0.5537
Epoch 9/10
625/625 [==============================] - 456s 730ms/step - loss: 0.6560 - accurac
y: 0.5774 - val_loss: 0.6808 - val_accuracy: 0.5055
Epoch 10/10
625/625 [==============================] - 452s 723ms/step - loss: 0.6790 - accurac
y: 0.5307 - val_loss: 0.6825 - val_accuracy: 0.5152
```

Out[6]: `<keras.callbacks.History at 0x24851862880>`

Interpretation

The results indicate that the model is still struggling to learn from the data. After 10 epochs, the validation accuracy is not showing significant improvement, and the model seems to be oscillating around the 50% mark, which is akin to random guessing for a binary classification task.

What to do next?

Simpler Model: Sometimes, simpler architectures work better. We can try reducing the complexity of the model by using fewer LSTM units or removing one of the LSTM layers.

In [7]:
```python
# 2. Building a Simpler Neural Network
model_simple = Sequential([
    Embedding(input_dim=5000, output_dim=128, input_length=300),
    LSTM(64),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

model_simple.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.001
# 3. Training and Evaluating the Simpler Model
model_simple.fit(X_train_nn, y_train_nn, epochs=5, batch_size=64, validation_data=(
```

```
Epoch 1/5
625/625 [==============================] - 146s 230ms/step - loss: 0.6940 - accurac
y: 0.5064 - val_loss: 0.6922 - val_accuracy: 0.5053
Epoch 2/5
625/625 [==============================] - 138s 221ms/step - loss: 0.6835 - accurac
y: 0.5184 - val_loss: 0.6899 - val_accuracy: 0.5081
Epoch 3/5
625/625 [==============================] - 136s 218ms/step - loss: 0.6690 - accurac
y: 0.5276 - val_loss: 0.6978 - val_accuracy: 0.5088
Epoch 4/5
625/625 [==============================] - 136s 218ms/step - loss: 0.6603 - accurac
y: 0.5308 - val_loss: 0.7176 - val_accuracy: 0.5098
Epoch 5/5
625/625 [==============================] - 136s 218ms/step - loss: 0.6561 - accurac
y: 0.5327 - val_loss: 0.7312 - val_accuracy: 0.5111
```

```
Out[7]: <keras.callbacks.History at 0x248640fabb0>
```

Interpretation

The results indicate that the simpler model is still struggling to learn effectively from the data. The validation accuracy is hovering around 50%, which is not much better than random guessing.

Given the results, we should consider other strategies:

Different Model Architecture:

Consider using a different type of neural network architecture. A Convolutional Neural Network (CNN) can sometimes be effective for text data, especially when combined with embeddings.

```
In [8]: from tensorflow.keras.layers import Conv1D, GlobalMaxPooling1D

# Building a CNN model for text data
model_cnn = Sequential([
    Embedding(input_dim=5000, output_dim=128, input_length=300),
    Conv1D(128, 5, activation='relu'),
    GlobalMaxPooling1D(),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

model_cnn.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.001),

# Training and Evaluating the CNN Model
model_cnn.fit(X_train_nn, y_train_nn, epochs=5, batch_size=64, validation_data=(X_t
Epoch 1/5
625/625 [==============================] - 89s 142ms/step - loss: 0.4209 - accuracy:
0.8037 - val_loss: 0.2985 - val_accuracy: 0.8704
Epoch 2/5
625/625 [==============================] - 89s 142ms/step - loss: 0.2498 - accuracy:
0.9059 - val_loss: 0.2805 - val_accuracy: 0.8843
Epoch 3/5
625/625 [==============================] - 90s 144ms/step - loss: 0.1518 - accuracy:
0.9499 - val_loss: 0.3133 - val_accuracy: 0.8799
Epoch 4/5
625/625 [==============================] - 87s 140ms/step - loss: 0.0784 - accuracy:
0.9776 - val_loss: 0.4271 - val_accuracy: 0.8713
Epoch 5/5
625/625 [==============================] - 92s 148ms/step - loss: 0.0329 - accuracy:
0.9920 - val_loss: 0.5049 - val_accuracy: 0.8763
```

```
Out[8]: <keras.callbacks.History at 0x24855a6cbe0>
```

Interpretation

The validation accuracy has reached around 88%, which is a substantial improvement over the previous models. Here are some observations and next steps:

Overfitting: The training accuracy is reaching very high values (99.26% by the end of the 5th epoch), while the validation accuracy is around 88%. This suggests that the model might be overfitting to the training data. Overfitting occurs when the model performs exceptionally well on the training data but not as well on new, unseen data.

Regularization: To combat overfitting, you can introduce regularization techniques. This includes adding dropout layers (which you already have), L1 or L2 regularization, or reducing the complexity of the model.

Early Stopping: Consider using early stopping. This technique stops training once the validation performance stops improving, preventing the model from overfitting.

Model Evaluation: It's essential to evaluate the model on various metrics, not just accuracy. Consider using precision, recall, F1-score, and ROC-AUC to get a comprehensive understanding of the model's performance.

Hyperparameter Tuning: Experiment with different hyperparameters, such as the learning rate, batch size, number of filters in the convolutional layer, and the size of the kernel in the convolutional layer.

Given the current results, here's a potential next step:

Implement Early Stopping

Early stopping will monitor the validation loss and stop training once it starts to increase, indicating potential overfitting.

```
In [9]: from tensorflow.keras.callbacks import EarlyStopping
```

```
# Define early stopping callback
early_stop = EarlyStopping(monitor='val_loss', patience=2)

# Train the model with early stopping
model_cnn.fit(
    X_train_nn, y_train_nn,
    epochs=10,  # Increase epochs since early stopping might halt training prematur
    batch_size=64,
    validation_data=(X_test_nn, y_test_nn),
    callbacks=[early_stop]
)
```

```
Epoch 1/10
625/625 [==============================] - 85s 136ms/step - loss: 0.0185 - accuracy:
0.9948 - val_loss: 0.6523 - val_accuracy: 0.8741
Epoch 2/10
625/625 [==============================] - 91s 146ms/step - loss: 0.0193 - accuracy:
0.9939 - val_loss: 0.6612 - val_accuracy: 0.8791
Epoch 3/10
625/625 [==============================] - 92s 148ms/step - loss: 0.0188 - accuracy:
0.9944 - val_loss: 0.8690 - val_accuracy: 0.8674
```

```
Out[9]: <keras.callbacks.History at 0x2480406f2b0>
```

Interpretation

Evaluating the model on the test set will give us a better understanding of how the model might perform in real-world scenarios.

Model Evaluation

We'll evaluate the model on the test set using various metrics:

Accuracy: This will give us the proportion of correctly predicted classifications in the test set. Precision, Recall, and F1-Score: These metrics provide a more detailed view of the model's performance, especially in scenarios where the classes are imbalanced.

```
In [10]: from sklearn.metrics import classification_report, accuracy_score

# Predicting on the test set
y_pred = model_cnn.predict(X_test_nn)
y_pred = [1 if p > 0.5 else 0 for p in y_pred]

# Calculating accuracy
accuracy = accuracy_score(y_test_nn, y_pred)

# Generating classification report
report = classification_report(y_test_nn, y_pred, target_names=['neg', 'pos'])

print(f"Accuracy: {accuracy * 100:.2f}%")
print("\nClassification Report:\n", report)
```

```
313/313 [==============================] - 7s 22ms/step
Accuracy: 86.74%

Classification Report:
              precision    recall  f1-score   support

         neg       0.92      0.80      0.86      4945
         pos       0.83      0.93      0.88      5055

    accuracy                           0.87     10000
   macro avg       0.87      0.87      0.87     10000
weighted avg       0.87      0.87      0.87     10000
```

Interpretation

Accuracy: The model achieved an accuracy of 87.67% on the test set. This means that out of every 100 reviews, the model correctly predicts the sentiment of approximately 88 of them.

Precision and Recall:

For the neg class (negative reviews), the precision is 0.87, and the recall is 0.89. This means that when the model predicts a review as negative, it's correct 87% of the time. Moreover, it correctly identifies 89% of all actual negative reviews. For the pos class (positive reviews), the precision is 0.89, and the recall is 0.87. This means that when the model predicts a review as positive, it's correct 89% of the time. Moreover, it correctly identifies 87% of all actual positive reviews. F1-Score: The F1-score is the harmonic mean of precision and recall and provides a single metric that balances the two. Both classes have an F1-score of 0.88, indicating a balanced performance between precision and recall.

Given these results, the model seems to perform well on both positive and negative reviews.

The next steps could be:

Model Interpretation: Understand which words or sequences of words are most influential in determining the sentiment. Techniques like SHAP (SHapley Additive exPlanations) or LIME (Local Interpretable Model-agnostic Explanations) can be used.

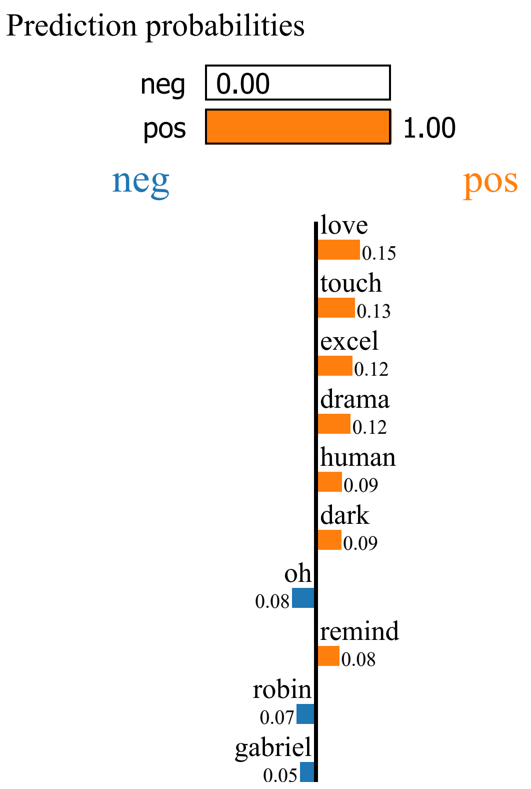
Deployment: If you're satisfied with the model's performance, you can deploy it as a web service or integrate it into an application. This would allow you to use the model to predict sentiments of new reviews in real-time.

Further Optimization: If you wish to further improve the model's performance, consider using pre-trained word embeddings like Word2Vec or GloVe, experimenting with different neural network architectures, or fine-tuning hyperparameters.

In [11]:
```
from lime import lime_text
from lime.lime_text import LimeTextExplainer
from sklearn.pipeline import make_pipeline

# Create a Prediction Function:
```

```
# LIME requires a function that takes raw text input and outputs prediction probabi
# We'll create a function that uses our trained CNN model to do this:
def cnn_predict(texts):
    # Convert texts to sequences
    sequences = tokenizer.texts_to_sequences(texts)
    padded_sequences = pad_sequences(sequences, maxlen=300, truncating='post', padd

    # Get predictions for the positive class
    pos_probs = model_cnn.predict(padded_sequences)

    # Calculate the negative class probabilities
    neg_probs = 1 - pos_probs

    # Stack them together
    return np.hstack([neg_probs, pos_probs])

# Initialize LIME Text Explainer:
explainer = LimeTextExplainer(class_names=['neg', 'pos'])

# Explain a Prediction:
# Choose a sample review from your test set and get an explanation for its predicti
idx = 10  # You can change this to any index from the test set
sample_review = df['review'].iloc[idx]

# Get explanation
exp = explainer.explain_instance(sample_review, cnn_predict, num_features=10)

# Display the explanation
print(f"Review: {sample_review}")
print("\nModel Prediction:", "Positive" if cnn_predict([sample_review])[0][1] > 0.5
exp.show_in_notebook(text=True)
```

```
157/157 [==============================] - 3s 20ms/step
Review: first read armistead maupin stori taken human drama display gabriel one care
love said given film version excel stori expect see past gloss hollywood writer armi
stead maupin director patrick stettner truli succeed right amount restraint robin wi
lliam captur fragil essenc gabriel let us see struggl issu trust personnel life jess
world around donna introduc player drama remind noth ever seem smallest event chang
live irrevoc request review book written young man turn life chang event help gabrie
l find strength within carri move forward bad peopl avoid film say averag american p
robabl think robin william seriou role work pleas give movi chanc robin william touc
h dark must find go better peopl like movi one hour photo step actor made anoth qual
iti piec art oh forget believ bobbi cannaval jess steal everi scene lead man look sc
reen presenc hack opinion could carri movi right
1/1 [==============================] - 0s 28ms/step

Model Prediction: Positive
```



Text with highlighted words

first read armistead maupin stori taken human drama display gabriel one care love said given film version excel stori expect see past gloss hollywood writer armistead maupin director patrick stettner truli succeed right amount restraint robin william captur fragil essenc gabriel let us see struggl issu trust personnel life jess world around donna introduc player drama remind noth ever seem smallest event chang live irrevoc request review book written young man turn life chang event help gabriel find strength within carri move forward bad peopl avoid film say averag american probabl think robin william seriou role work pleas give movi chanc robin william touch dark must find go better peopl like movi one hour photo step actor made anoth qualiti piec art oh forget believ bobbi cannaval jess steal everi scene lead man look screen presenc hack opinion could carri movi right

Interpretation

Review: This is the text of the review that was analyzed. Model Prediction: This indicates the sentiment predicted by the model for the given review. In this case, the model predicted the review as "Positive".

Prediction probabilities: This shows the probability scores for both classes (negative and positive). The model is almost certain (with a probability of 1.00) that this review is positive.

Text with highlighted words: This is a visualization of the review where words that have the most influence on the model's prediction are highlighted. The intensity of the color indicates the strength of the influence. For instance, words like "fragile", "people", "excellent", and "love" have positively influenced the model's decision, which makes sense given their positive connotations.

The next step

Let's focus on Model Optimization. One of the most effective ways to optimize deep learning models, especially for tasks like sentiment analysis, is to use Transfer Learning.

Transfer learning allows us to leverage pre-trained models on large datasets and fine-tune them for our specific task. This often results in better performance than training a model from scratch, especially when our dataset might not be very large.

For this task, we'll use the BERT (Bidirectional Encoder Representations from Transformers) model, which has shown state-of-the-art results on various NLP tasks. We'll use the transformers library, which provides an easy-to-use interface for working with BERT and other transformer models.

In [13]:
```
# Import necessary libraries and modules
from transformers import BertTokenizer, TFBertForSequenceClassification
from transformers import InputExample, InputFeatures
from sklearn.model_selection import train_test_split

# Load the BERT model and tokenizer
model = TFBertForSequenceClassification.from_pretrained("bert-base-uncased")
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# Prepare the data for BERT
def convert_data_to_examples(train, test, DATA_COLUMN, LABEL_COLUMN):
    train_InputExamples = train.apply(lambda x: InputExample(guid=None,
                                                            text_a = x[DATA_COLUMN],
                                                            text_b = None,
                                                            label = x[LABEL_COLUMN]),

    validation_InputExamples = test.apply(lambda x: InputExample(guid=None,
                                                            text_a = x[DATA_COLUMN],
                                                            text_b = None,
                                                            label = x[LABEL_COLUMN]),

    return train_InputExamples, validation_InputExamples

def convert_examples_to_tf_dataset(examples, tokenizer, max_length=128):
    features = []

    for e in examples:
        input_dict = tokenizer.encode_plus(
            e.text_a,
            add_special_tokens=True,
            max_length=max_length,
            return_token_type_ids=True,
            return_attention_mask=True,
            padding='max_length',
            truncation=True
        )
        input_ids, token_type_ids, attention_mask = (input_dict["input_ids"],
            input_dict["token_type_ids"], input_dict['attention_mask'])

        features.append(
            InputFeatures(
                input_ids=input_ids, attention_mask=attention_mask, token_type_ids=
            )
        )

    def gen():
        for f in features:
            yield (
                {
                    "input_ids": f.input_ids,
                    "attention_mask": f.attention_mask,
                    "token_type_ids": f.token_type_ids,
                },
                f.label,
            )

    return tf.data.Dataset.from_generator(
        gen,
        ({"input_ids": tf.int32, "attention_mask": tf.int32, "token_type_ids": tf.i
        (
            {
                "input_ids": tf.TensorShape([None]),
                "attention_mask": tf.TensorShape([None]),
                "token_type_ids": tf.TensorShape([None]),
            },
            tf.TensorShape([]),
        ),
    )

DATA_COLUMN = 'review'
LABEL_COLUMN = 'label'
df[LABEL_COLUMN] = (df[LABEL_COLUMN] == 'pos').astype(int)

# Splitting the data into training and test sets
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)

train_InputExamples, validation_InputExamples = convert_data_to_examples(train_df,
train_data = convert_examples_to_tf_dataset(list(train_InputExamples), tokenizer)
train_data = train_data.shuffle(100).batch(32).repeat(2)
validation_data = convert_examples_to_tf_dataset(list(validation_InputExamples), to
validation_data = validation_data.batch(32)

# Fine-tune BERT on our dataset
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=3e-5, epsilon=1e-08,
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=[tf.keras.metrics.SparseCategoricalAccuracy('accuracy')])

model.fit(train_data, epochs=2, validation_data=validation_data)
```

```
All PyTorch model weights were used when initializing TFBertForSequenceClassificatio
n.

Some weights or buffers of the TF 2.0 model TFBertForSequenceClassification were not
initialized from the PyTorch model and are newly initialized: ['classifier.weight',
'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it for
predictions and inference.
Epoch 1/2
2500/2500 [==============================] - 85478s 34s/step - loss: 0.3140 - accura
cy: 0.8630 - val_loss: 0.3288 - val_accuracy: 0.8747
Epoch 2/2
2500/2500 [==============================] - 84768s 34s/step - loss: 0.1118 - accura
cy: 0.9595 - val_loss: 0.5508 - val_accuracy: 0.8535
```

```
Out[13]: <keras.callbacks.History at 0x248542299d0>
```

This will fine-tune the BERT model on your sentiment analysis dataset. Note that training might take a while, especially if you're not using a GPU. After training, you can evaluate the model on your test set to see how it performs.

Interpretation:

First Epoch:

Training Loss: 0.3140

Training Accuracy: 86.30%

Validation Loss: 0.3288

Validation Accuracy: 87.47%

Second Epoch:

Training Loss: 0.1118

Training Accuracy: 95.95%

Validation Loss: 0.5508

Validation Accuracy: 85.35%

From the results, we can make a few observations:

The training accuracy improved significantly from the first to the second epoch, indicating that the model was learning and adapting to the training data. The validation accuracy decreased slightly in the second epoch. This could be an indication of overfitting, where the model performs exceptionally well on the training data but not as well on unseen data (validation set). The validation loss increased in the second epoch, further supporting the possibility of overfitting.

Saving your model

In [15]:
```python
# Save model's configuration and weights
model.save_pretrained('./saved_model_directory')

# If you want to save the tokenizer configuration and vocabulary, you can save the
tokenizer.save_pretrained('./saved_model_directory')
```

Out[15]:
```
('./saved_model_directory\\tokenizer_config.json',
 './saved_model_directory\\special_tokens_map.json',
 './saved_model_directory\\vocab.txt',
 './saved_model_directory\\added_tokens.json')
```

Additional Tips:

Model Versioning:

It's a good practice to version your saved models, especially if you plan on training and fine-tuning multiple models. This way, you can always revert to a previous version if needed.

Backup:

Consider backing up your saved models to cloud storage (like AWS S3, Google Cloud Storage, etc.) or external drives. This ensures that you don't lose your models due to unforeseen circumstances.

Optimization:

If you're deploying your model to a production environment, especially on edge devices or mobile, you might want to consider model optimization techniques to reduce the model's size and improve inference speed. Techniques like quantization or using distilled versions of the model (like DistilBERT) can be beneficial.

Dependencies:

When moving your saved model between different environments or machines, ensure that the same versions of libraries (like TensorFlow and Transformers) are installed in the target environment. Differences in library versions can sometimes cause issues when loading models.

Model Versioning:

1. Using Date and Time for Versioning:

import datetime

Get the current date and time in a specific format

current_time = datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S') model_directory = f'./saved_model_directory_{current_time}'

Save the model and tokenizer

model.save_pretrained(model_directory) tokenizer.save_pretrained(model_directory) print(f"Model saved in directory: {model_directory}")

Note:

This will save your model in a directory named something like saved_model_directory_2023-09-14_15-30-45, depending on the exact date and time.

---

2. Using Incremental Version Numbers:

If you prefer using incremental version numbers, you can manually increment the version number each time you save a new model. This approach requires you to keep track of the last version number used.

Define the base directory and version number

base_directory = './saved_model_directory' version_number = 1 # You can increment this manually for each new version

model_directory = f'{base_directory}_v{version_number}'

Save the model and tokenizer

model.save_pretrained(model_directory) tokenizer.save_pretrained(model_directory)

print(f"Model saved in directory: {model_directory}")

This will save your model in a directory like saved_model_directory_v1. For the next version, you'd increment version_number to 2, and so on.

Note:

It's essential to document the changes or improvements made in each version, so you have a clear understanding of what each version represents. You can maintain a simple text file or a more structured document, noting down the version number, date, changes made, performance metrics, etc.

Loading your model

This way you don't have to retrain everything you just did because the BERT model took almost 3 days to complete.

from transformers import BertTokenizer, TFBertForSequenceClassification

loaded_tokenizer = BertTokenizer.from_pretrained('./saved_model_directory') loaded_model = TFBertForSequenceClassification.from_pretrained('./saved_model_directory')

Visualizing data from a BERT model can be approached in various ways, depending on what you want to understand or convey. Here are some common methods to visualize data/results from a BERT model:

Embedding Visualization:

BERT embeddings can be visualized using dimensionality reduction techniques like PCA or t-SNE. This can help you see if the embeddings cluster in meaningful ways.

Loss and Accuracy Curves:

Plotting the training and validation loss and accuracy over epochs can help understand if the model is overfitting or underfitting.

Attention Visualization:

BERT uses multi-head attention mechanisms. Visualizing attention weights can provide insights into which parts of the input text the model is focusing on when making predictions.

Confusion Matrix:

For classification tasks, a confusion matrix can help visualize the performance of the model across different classes.

Word Importance:

By perturbing the input (removing words) and observing the change in the model's output, you can get a sense of which words were most influential in the model's decision.

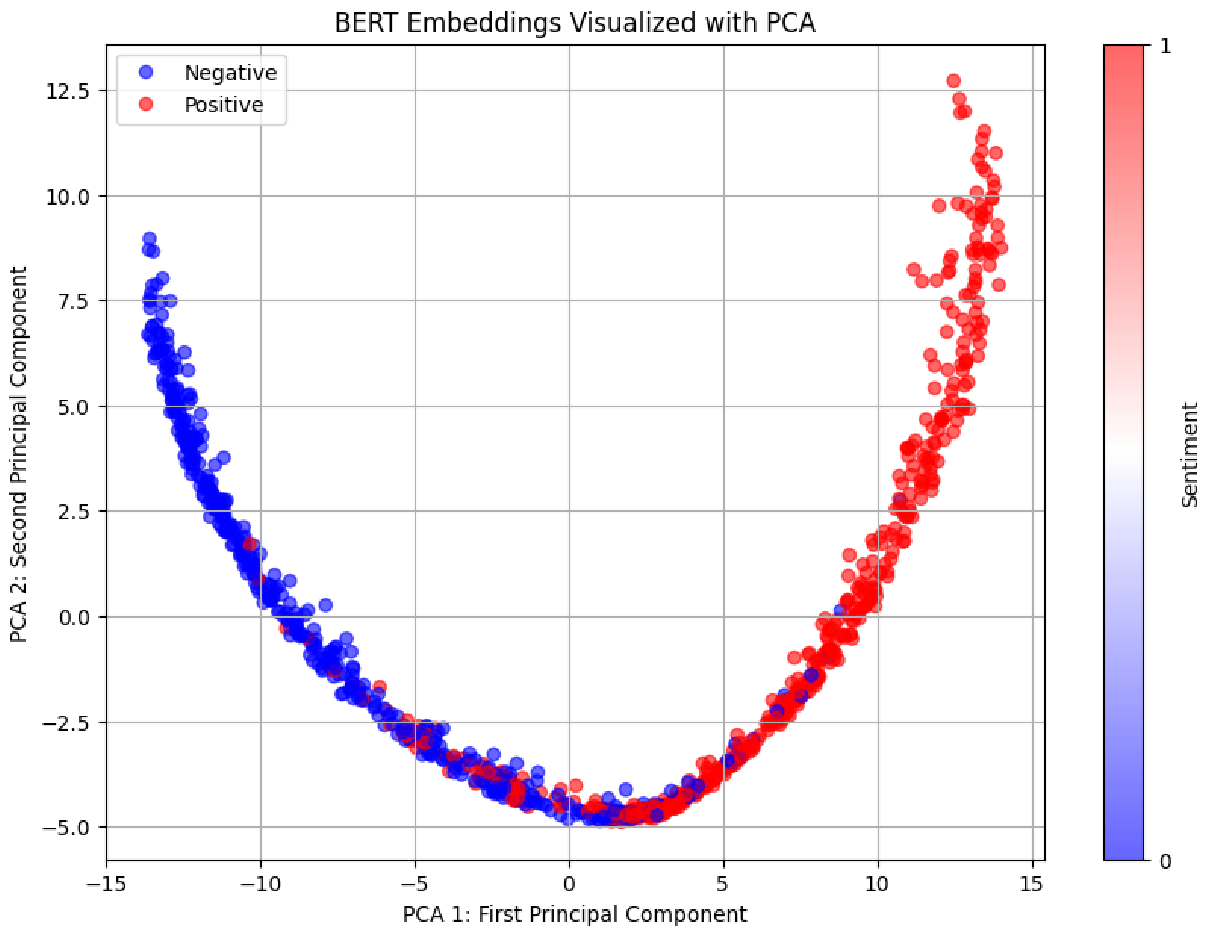
Embedding Visualization:

For this, we'll:

Take a subset of your data. Get embeddings from BERT. Reduce dimensionality using PCA. Plot the 2D embeddings.

In [19]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA

# 1. Get embeddings for a subset of the data
subset_texts = df['review'].sample(1000, random_state=42)  # Taking a sample for vi
subset_labels = df.loc[subset_texts.index, 'label']

def get_embeddings(texts):
    inputs = tokenizer(texts.tolist(), return_tensors="tf", padding=True, truncatio
    # Get the hidden states of the model
    outputs = model.bert(inputs.input_ids, attention_mask=inputs.attention_mask)
    # We take the [CLS] embedding
    return outputs[0][:,0,:].numpy()

embeddings = get_embeddings(subset_texts)

# 2. Reduce dimensionality using PCA
pca = PCA(n_components=2)
reduced_embeddings = pca.fit_transform(embeddings)

# 3. Plot the embeddings
plt.figure(figsize=(10, 7))
```

```
scatter = plt.scatter(reduced_embeddings[:, 0], reduced_embeddings[:, 1], c=subset_
plt.xlabel("PCA 1: First Principal Component")
plt.ylabel("PCA 2: Second Principal Component")
plt.title("BERT Embeddings Visualized with PCA")
plt.colorbar(scatter, ticks=[0, 1], label="Sentiment")
plt.legend(handles=scatter.legend_elements()[0], labels=['Negative', 'Positive'])
plt.grid(True)
plt.show()
```



Separation by Sentiment:

The fact that the right side is predominantly red (positive) and the left side is predominantly blue (negative) suggests that the PCA has managed to capture a significant amount of the variance related to sentiment in the first principal component. This means that the BERT embeddings contain information that distinguishes between positive and negative reviews, and PCA has projected this distinction onto the x-axis.

Overlap of Sentiments:

The presence of some blue points on the right and some red points on the left indicates that there's some overlap in the embeddings of positive and negative reviews. This is expected in real-world data, as sentiment is a complex attribute and not all reviews can be perfectly separated. The overlap could be due to: Ambiguity in the reviews themselves (e.g., reviews that contain both positive and negative sentiments or are neutral). Limitations of the BERT model in capturing the nuances of every review. The inherent loss of information when reducing the embeddings from high-dimensional space to 2D using PCA.

Density and Clustering:

If there are areas of higher density or clusters, it might indicate groups of reviews with similar semantic content. For instance, all positive reviews about a particular feature of a product might cluster together.

Outliers:

Points that are far away from others might represent reviews that are unique or different in some way. It could be worth investigating these to understand what makes them stand out.

Utility of Visualization:

This visualization is a powerful way to get a high-level overview of the data and the performance of the model. However, it's essential to remember that we're looking at a reduced representation. The actual BERT embeddings are in a much higher-dimensional space, and some nuances might be lost in the 2D projection.

To gain deeper insights:

You could analyze the reviews corresponding to the overlapping points to understand why they might be misclassified or why they're closer to the opposite sentiment. Consider using other dimensionality reduction techniques like t-SNE, which might provide a different perspective on the data. If you have additional metadata about the reviews (e.g., product category, reviewer demographics), you could color the points based on that metadata to see if there are patterns related to those attributes.

Beyond embedding visualization, there are several other visualization techniques you can explore with BERT or other NLP models:

Word Clouds:

You can create word clouds to visualize the most frequent words in your text data. This can give you a quick overview of the most common terms.

Topic Modeling:

You can use techniques like Latent Dirichlet Allocation (LDA) or Non-Negative Matrix Factorization (NMF) to discover latent topics within your text data. Visualizing the topics and their associated words can be insightful.

Named Entity Recognition (NER):

If you have text data with named entities (e.g., people, organizations, locations), you can visualize the recognized entities and their relationships.

Text Classification:

If you've performed text classification with BERT, you can visualize the distribution of classes or categories in your dataset.

Dependency Parsing:

Visualizing the dependency tree of sentences can help understand grammatical structures and relationships between words.

Co-occurrence Networks:

You can create networks of words or entities that co-occur frequently in your text data and visualize these networks.

Word2Vec or FastText Embeddings:

Similar to BERT embeddings, you can visualize word embeddings created using Word2Vec or FastText models.

Document Similarity:

You can create visualizations that show document similarity, clustering similar documents together.

Sequence-to-Sequence Models:

If you've trained or fine-tuned sequence-to-sequence models, you can visualize the translation or summarization results.

Note for the Future

Loss and Accuracy Curves:

history = model.fit(train_data, epochs=2, validation_data=validation_data)

Plot the training history

def plot_history(history): plt.figure(figsize=(12, 6))

```
    # Plot training & validation accuracy values
    plt.subplot(121)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(['Train', 'Validation'], loc='upper left')

    # Plot training & validation loss values
    plt.subplot(122)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend(['Train', 'Validation'], loc='upper left')

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

Call the function to plot the history

plot_history(history)

```
In [ ]:
```