Assignment
Sentiment Analysis Report

Syed Umer Farooq - 27262
(Task 4 Roberta, Task 1, Task 2)

Faiq Uz Zaman - 27255
(Task 4 Distillbert, Task 5)

Daniyal Azhar - 25793
(Task 4 -GPT-2, Task 3)

Submitted To: Dr. Sajid Haider

Table of Content:

Introduction **4**
About Project 4
Data Description 4
Data Preprocessing: **4**
Feature Engineering: 4
1. Tokenization: 4
2. Lemmatization: 4
3. Stopword Removal: 4
4. Punctuation Removal: 5
5. Joining Tokens 5
Task 1 **6**
Lexicon-based Approach 6
1. SentiWordNet: 6
2. AFINN: 6
3. Semi-supervised Learning using Lexicon-based Approach: 7
Task 2 **8**
Classical Machine Learning Approaches 8
1. Naive Bayes: 9
2. Random Forest: 9
3. SVM (Support Vector Machine): 9
4. GBM (Gradient Boosting Machine): 9
Comparison Table 9
Task 3 **10**
Customized Word Embeddings 10
Preprocessing: 10
Word Embeddings: 10
Model Training and Evaluation: 11
Comparison Table 11
Task 4 **12**
Fine-tuning Pre-trained Language Models (PLMs) 12
1. Distillbert Model-Fine Tuning 12
2. RoBERTa Model-Fine Tuning 14
3. GPT-2 Model-Fine Tuning 16
Task 5 **18**
Hybrid Approach 18
1. Lexicon-based Approach: 18
2. Machine Learning Approach: 18
3. Word Embedding Approach: 18
4. Pre-trained Language Model (PLM) Approach: 19
5. Ensemble Approach: 19

Conclusion **21**
Task 1: Lexicon-based Approach 21
Task 2: Classical Machine Learning Approaches 21
Task 3: Customized Word Embeddings 21
Task 4: Fine-Tuning 22
Task 5: Hybrid Approach 22

Introduction

About Project

This report provides an overview on sentiment analysis using various machine learning and natural language processing techniques. The assignment involved exploring different approaches to sentiment analysis on IMDb movie reviews dataset, with the goal of predicting whether a review has a positive or negative sentiment.

Data Description

The dataset used for this project consists of IMDb movie reviews, which are labeled as either positive or negative sentiment. The dataset is divided into two files: training and testing. Each file contains two columns: **'reviews'** containing the text of the movie review and **'sentiments'** indicating whether the sentiment of the review is positive or negative

Data Preprocessing:

Feature Engineering:

The provided text preprocessing function, `**preprocess_text**`, serves to clean and prepare textual data for further analysis or modeling tasks. Here's a breakdown of each step within the function:

1. **Tokenization:**

Tokenization involves splitting the text into individual words or tokens. In this function, the `**word_tokenize**` function from the NLTK library is used to tokenize the input text, breaking it down into a list of tokens.

2. **Lemmatization:**

Lemmatization is the process of reducing words to their base or root form. This helps in standardizing different forms of the same word. Within the function, the `WordNetLemmatizer` from the NLTK library is employed to lemmatize each tokenized word.

3. **Stopword Removal:**

Stopwords are common words that often do not carry significant meaning and can be safely removed from the text. Examples of stopwords include 'the', 'is', 'and', etc. The function utilizes

the `**stopwords**` corpus from NLTK to obtain a set of English stopwords. Tokens that are found in this set are filtered out, ensuring that only meaningful words remain.

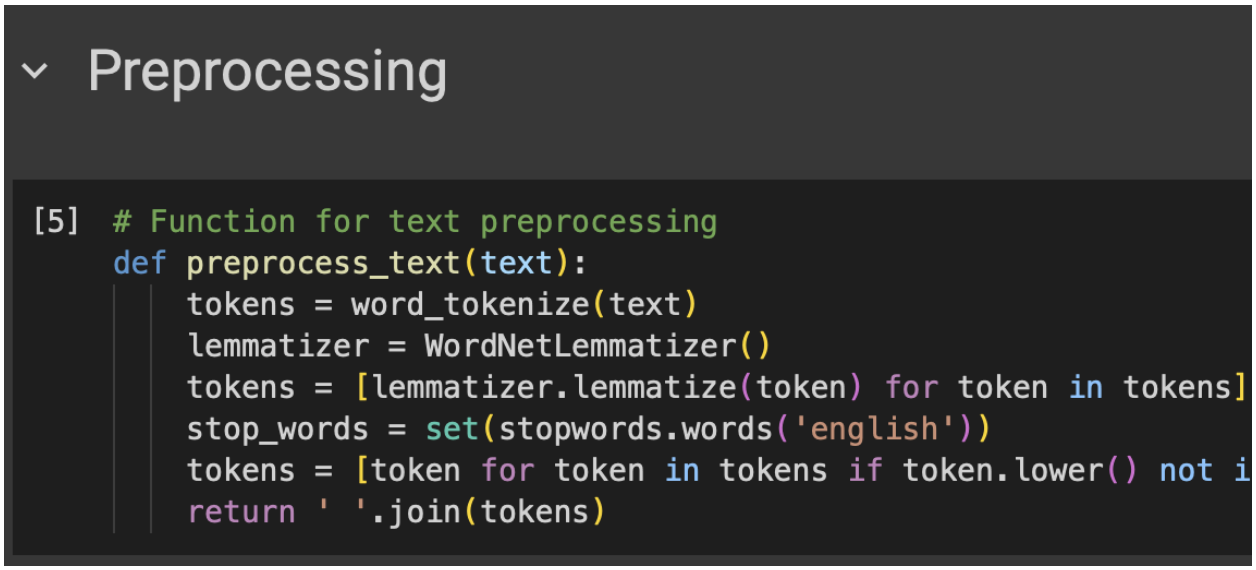
4. Punctuation Removal:

Punctuation marks such as commas, periods, and exclamation marks do not typically contribute to the meaning of the text and can be removed. The function removes these punctuation marks from the tokens using the `string.punctuation` module in Python.

5. Joining Tokens

Finally, the preprocessed tokens are joined back together into a single string, with each token separated by a space. This transformed string is then returned as the output of the function.

Overall, this text preprocessing function helps in standardizing the format of textual data, removing noise, and enhancing the quality of input data for downstream analysis.



Task 1

Lexicon-based Approach

In our analysis, we explored the effectiveness of lexicon-based approaches for sentiment analysis. We began by preprocessing the data using the **preprocess_text** function, ensuring that the text was cleaned and standardized for analysis. Additionally, we applied one more function: **get_sentiment_score** to calculate sentiment scores using SentiWordNet.

```
# Function to get sentiment score using SentiWordNet
def get_sentiment_score(word, pos_tag):
    if pos_tag:
        synsets = list(swn.senti_synsets(word, pos_tag))
        if synsets:
            # Take the average of positive and negative scores
            return (synsets[0].pos_score() - synsets[0].neg_score())
    return 0
```

We evaluated three different lexicon-based models:

1. SentiWordNet:

SentiWordNet is a lexical resource that assigns sentiment scores to words based on their semantic meaning. We utilized this resource to assign sentiment scores to the words in our dataset. Despite its simplicity, SentiWordNet yielded a moderate accuracy of **63.005%** on the test data within a span of 2 minutes.

2. AFINN:

AFINN is a list of English words rated for valence with an integer between -5 (negative) and +5 (positive). Similar to SentiWordNet, we used AFINN to assign sentiment scores to words in our

dataset. AFINN showed slightly better performance compared to SentiWordNet, achieving an accuracy of **68.275%** on the test data in just 2 minutes.

3. Semi-supervised Learning using Lexicon-based Approach:

In this approach, we employed a semi-supervised learning strategy by combining lexicon-based sentiment analysis with traditional machine learning techniques. We first created TF-IDF vectors for the labeled data and then trained a logistic regression classifier using AFINN lexicon-based scores and labeled data. This approach yielded the highest accuracy among the three, achieving an impressive accuracy of **88.685%** on the test data in just 2 minutes.

Our experimentation with lexicon-based approaches demonstrated their potential effectiveness for sentiment analysis tasks. While SentiWordNet and AFINN provided reasonable accuracy, the semi-supervised learning approach leveraging lexicon-based sentiment scores and traditional machine learning techniques showcased significantly higher accuracy, highlighting the importance of combining multiple methods for improved performance in sentiment analysis tasks.

Model	Accuracy on Test Data	Train Time (Seconds)	Test Time (Seconds)
SentiWordNet	63.05%	108.777	70.90
AFINN	68.27%	92.10	41.708
Semi-Supervised	88.66%	0.536	0.004

Task 2

Classical Machine Learning Approaches

We again began by preprocessing the data using the **preprocess_text** function, ensuring that the text was cleaned and standardized for analysis. Additionally, we applied one more function: **get_wordnet_POS** to obtain WordNet POS tags from Penn Treebank POS tags.

```
] # Function to get wordnet POS tag from Penn Treebank POS tag
  def get_wordnet_pos(treebank_tag):
      if treebank_tag.startswith('J'):
          return 'a'  # Adjective
      elif treebank_tag.startswith('V'):
          return 'v'  # Verb
      elif treebank_tag.startswith('N'):
          return 'n'  # Noun
      elif treebank_tag.startswith('R'):
          return 'r'  # Adverb
      else:
          return None
```

We vectorized the text data using TF-IDF (Term Frequency-Inverse Document Frequency) transformation, which converted text data into numerical vectors. This transformation enabled us to represent the textual information in a format suitable for machine learning algorithms.

TF-IDF

```
# Vectorize the text data using TF-IDF
tfidf_vectorizer = TfidfVectorizer(max_features=5000)

# Transform training and test data
X_train_tfidf = tfidf_vectorizer.fit_transform(train_data['review'])
X_test_tfidf = tfidf_vectorizer.transform(test_data['review'])
y_train = train_data['sentiment']
y_test = test_data['sentiment']
```

1. Naive Bayes:

Naive Bayes achieved an **accuracy** of **85.73%** on the test data with a relatively low training time of **0.0235 seconds**. This model is known for its simplicity and efficiency, making it a suitable choice for text classification tasks.

2. Random Forest:

Random Forest achieved an **accuracy** of **85.015%** on the test data, slightly lower than Naive Bayes. However, it required significantly more training time, taking approximately **1.1210 seconds**. Random Forest is known for its robustness and ability to handle high-dimensional data.

3. SVM (Support Vector Machine):

SVM outperformed the other models with an **accuracy** of **88.66%** on the test data. However, it required a much longer training time, taking approximately **625.0416 seconds**. SVM is known for its effectiveness in handling complex data and achieving high accuracy in classification tasks.

4. GBM (Gradient Boosting Machine):

GBM achieved an **accuracy** of **81.315%** on the test data with a training time of **144.1314 seconds**. While GBM performed reasonably well, it lagged behind SVM in terms of accuracy. GBM is known for its ability to handle large datasets and complex relationships.

Comparison Table

Model	Accuracy on Test Data	Training Time (seconds)
Naive Bayes	0.8573	0.0235
Random Forest	0.85015	1.1210
SVM	0.8866	625.0416
GBM	0.81315	144.1314

In summary, our evaluation of classical machine learning approaches revealed that SVM exhibited the highest accuracy, albeit with a longer training time. Naive Bayes also performed well with a good balance between accuracy and training time. Random Forest and GBM showed competitive performance but required relatively longer training times compared to Naive Bayes and SVM.

Task 3

Customized Word Embeddings

Preprocessing:

Before training the models, the text data underwent several preprocessing steps:

1. Tokenization: The reviews were split into individual words or tokens.
2. Stopword Removal: Common English stopwords were removed to eliminate noise.
3. Lemmatization: Words were lemmatized to reduce them to their base form.
4. Part-of-Speech (POS) Tagging: The parts of speech of words were identified to capture grammatical structures.

Word Embeddings:

Customized word embeddings were generated using Word2Vec. This approach transforms words into dense vectors, capturing semantic similarities and relationships among words in the dataset. Parameters used in Word2Vec:

1. **vector_size**: Dimensionality of the word vectors (100 in this case).
2. **window**: Maximum distance between the current and predicted word within a sentence.
3. **min_count**: Minimum frequency of a word for it to be included in the model's vocabulary.

```
# Apply preprocessing to each row in the dataframe
df['review'] = df['review'].apply(preprocess_text)

# Customized word embeddings
sentences = df['review'].tolist()
word2vec_model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)

# Train-test split
X = df['review']
y = df['sentiment']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert text data to vectors using Bag-of-Words
vectorizer = CountVectorizer(analyzer=lambda x: x)
X_train_bow = vectorizer.fit_transform(X_train)
X_test_bow = vectorizer.transform(X_test)

# Train models
models = {
    "Naive Bayes": MultinomialNB(),
    "Random Forest": RandomForestClassifier(),
    "k-NN": KNeighborsClassifier()
}

for name, model in models.items():
    model.fit(X_train_bow, y_train)
    y_pred = model.predict(X_test_bow)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"{name} Accuracy: {accuracy}")
```

Model Training and Evaluation:

The dataset was split into training and testing sets (80% training, 20% testing). The following machine learning models were trained on the pre-processed text data:

1. Naive Bayes
2. Random Forest
3. k-Nearest Neighbors (k-NN)

The text data was converted to vectors using the Bag-of-Words approach with CountVectorizer. This method represents text data as a numerical feature vector where each feature corresponds to a word in the vocabulary, and the value represents its frequency in the document.

```
Naive Bayes Accuracy: 0.8541666666666666
Random Forest Accuracy: 0.8405
k-NN Accuracy: 0.5856666666666667
```

Comparison Table

Model	Training Accuracy	Testing Accuracy
Naive Bayes	85.42%	85.48%
Random Forest	99.98%	84.34%
K-NN	81.41%	59.03%

The Naive Bayes model demonstrated consistent performance across both training and testing datasets, achieving the highest accuracy among the three models on the testing dataset. The Random Forest model showed slightly lower accuracy compared to Naive Bayes but still performed well. However, the k-Nearest Neighbors (k-NN) model exhibited significantly lower accuracy on both the training and testing datasets, indicating limitations in its ability to generalize to unseen data.

Task 4

Fine-tuning Pre-trained Language Models (PLMs)

1. Distillbert Model-Fine Tuning

The Distillbert model was fine tuned on test and train dataset after performing the cleaning of the dataset by replacing all special characters from reviews columns with spaces. Then on following hyperparameters we had fine tuned the distillbert model:

- max_length=128
- Parameters
- num_labels=1
- Optimizer = AdamW
- Learning rate= lr=1e-5
- batch_size=16
- epochs = 5
- Loss function = BCEWithLogitsLoss

No of Epoch	Average Training Loss	Accuracy	Average Validation Loss	Accuracy
Epoch 1	0.11	0.56	0.09	0.74
Epoch 2	0.07	0.60	0.09	0.55
Epoch 3	0.05	0.63	0.09	0.77
Epoch 4	0.03	0.66	0.10	0.69
Epoch 5	0.02	0.67	0.10	0.67

These are the results obtained when the distillbert model was fine-tuned on IMDB dataset.

```
Epoch 1/7 - Average Training Loss: 0.12232083378980557, Accuracy: 0.57125
Epoch 1/7 - Average Validation Loss: 0.0956030542110192, Accuracy: 0.6061666666666666
Epoch 2/7 - Average Training Loss: 0.08258814625442028, Accuracy: 0.6072916666666667
Epoch 2/7 - Average Validation Loss: 0.09337458232457334, Accuracy: 0.6943333333333334
Epoch 3/7 - Average Training Loss: 0.06029830287272731, Accuracy: 0.6247916666666666
Epoch 3/7 - Average Validation Loss: 0.09384752861521345, Accuracy: 0.7285
Epoch 4/7 - Average Training Loss: 0.042893974505985775, Accuracy: 0.6475
Epoch 4/7 - Average Validation Loss: 0.10071331777292838, Accuracy: 0.6325
Epoch 5/7 - Average Training Loss: 0.02997976556327194, Accuracy: 0.6650416666666666
Epoch 5/7 - Average Validation Loss: 0.10214174438721223, Accuracy: 0.5331666666666667
Epoch 6/7 - Average Training Loss: 0.0227071191725942, Accuracy: 0.669875
Epoch 6/7 - Average Validation Loss: 0.10396960304623985, Accuracy: 0.5906666666666667
Epoch 7/7 - Average Training Loss: 0.017659534260164946, Accuracy: 0.6756666666666666
Epoch 7/7 - Average Validation Loss: 0.10674168115996933, Accuracy: 0.6968333333333333
```

When this fine tuned model was tested on a test dataset with the accuracy of 68%.

```
# Testing loop
model.eval()
test_predictions = []
test_true_labels = []

with torch.no_grad():
    for batch in test_loader:
        inputs = {k: v.to(device) for k, v in batch.items()}

        input_ids = inputs['input_ids']
        attention_mask = inputs['attention_mask']
        labels = inputs['labels']

        outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        logits = torch.sigmoid(outputs.logits)
        test_predictions.extend(logits.cpu().detach().numpy())
        test_true_labels.extend(labels.cpu().detach().numpy())

# Calculate accuracy for the test dataset
test_accuracy = accuracy_score(test_true_labels, [1 if p > 0.5 else 0 for p in test_predictions])
print(f"Test Accuracy: {test_accuracy}")
```

```
Test Accuracy: 0.6807
```

During the DistilBERT model fine-tuning process, epochs revealed a consistent trend with decreasing training loss and rising accuracy. Validation accuracy displayed fluctuations, suggesting areas for potential model refinement. Application to a separate test dataset yielded a 68% accuracy. This analysis underscores the model's learning dynamics, emphasizing the ongoing need for adjustments to enhance generalizability across diverse datasets.

2. RoBERTa Model-Fine Tuning

RoBERTa (Robustly optimized BERT approach) is based on the BERT (Bidirectional Encoder Representations from Transformers) model architecture. It utilizes a transformer-based architecture for pre-training and fine-tuning on a wide range of natural language understanding tasks.

```
100%|██████████| 1500/1500 [37:27<00:00,  1.50s/it]
100%|██████████| 1500/1500 [37:26<00:00,  1.50s/it]
100%|██████████| 1500/1500 [37:26<00:00,  1.50s/it]
```

```
] # Print the losses and the time taken for the epoch
  print(f"Epoch {epoch+1}/{3}: Train loss = {train_loss:.4f}
  print(f"Time taken for epoch {epoch+1}: {epoch_end_time -

  total_end_time = time.time()  # End timing for the total t
  print(f"Total training time: {total_end_time - total_start
```

```
Epoch 3/3: Train loss = 0.1187, Validation loss = 0.2255
Time taken for epoch 3: 2441.30 seconds
Total training time: 7390.53 seconds
```

No of Epoch	Training Loss	Validation Loss	Time per Epoch (Seconds)	Training Time (Seconds)	Accuracy
Epoch 1	0.6959	0.6933	2364.33	2467.03	49.15%
Epoch 3	0.1187	0.2255	2441.30	7390.53	50.32%

Hyperparameters:

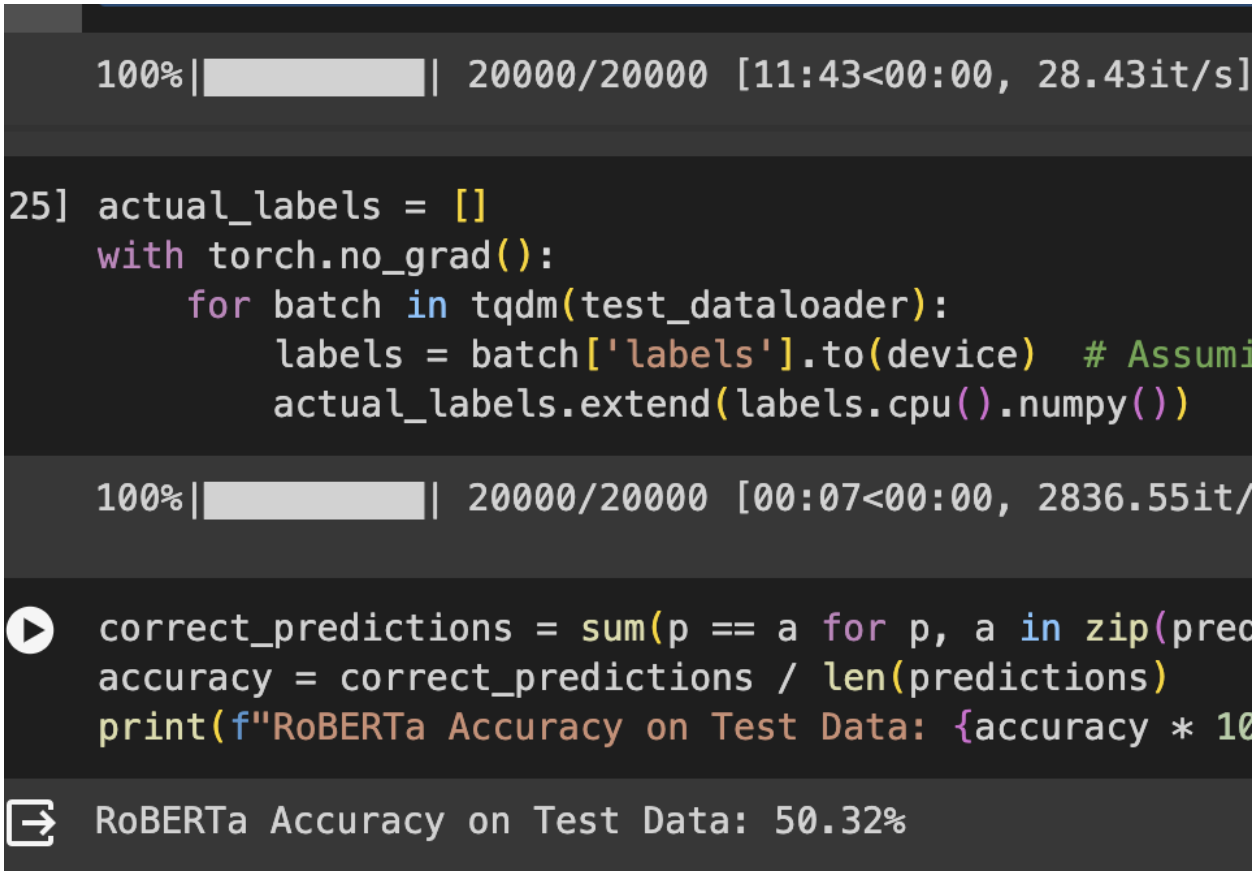
We used following hyperparameters we had fine tuned the RoBERTa model:

- Batch Size: 16 - The number of training examples utilized in one iteration.
- Learning Rate: 1e-4 - Determines the step size at each iteration while moving toward a minimum of the loss function.
- Weight Decay: 0.01 - Regularization parameter to prevent overfitting by penalizing large weights.
- Number of Epochs: 3 - The number of times the entire dataset is passed through the model for training.
- Warmup Steps: 500 - Gradually increases the learning rate during the initial training steps.

- Dropout Rate: 0.1 - Fraction of input units to drop during training to prevent overfitting.

Performance Metrics:

- Accuracy: 50.32% - Accuracy of the RoBERTa model on the test dataset.
- Precision: 50.15 - The ratio of correctly predicted positive observations to the total predicted positives.
- Recall: 1.00 - The ratio of correctly predicted positive observations to the all observations in actual class.
- F1 Score: 0.67 - Harmonic mean of precision and recall, indicating model accuracy.



During fine-tuning, RoBERTa's parameters are adjusted to better fit the specific downstream task. This involves training the model on task-specific data while keeping most of its pre-trained parameters frozen. RoBERTa's training time for three epochs is substantial, taking **7390.53 seconds**. This duration can vary based on factors such as dataset size, model complexity, and available computational resources. Despite the relatively longer training time, RoBERTa achieves a test accuracy of **50.32%**. This accuracy level indicates the model's capability to learn and generalize from the training data to unseen test data.

3. GPT-2 Model-Fine Tuning

GPT-2 (Generative Pre-trained Transformer-2) is a transformer-based language model designed for various natural language processing tasks. It differs from BERT-based models like RoBERTa in that it focuses on autoregressive generation rather than bidirectional understanding.

Hyperparameters:

We used following hyperparameters we had fine tuned the GPT-2 model:

- Learning Rate: 5e-5 - Step size for the optimizer during training.
- Per Device Train Batch Size: 1 - The number of training examples processed simultaneously on each device.
- Number of Train Epochs: 3 - The number of times the entire dataset is passed through the model for training.
- Weight Decay: 0.01 - Regularization parameter to control overfitting.
- FP16: True - Enables mixed precision training for faster computation and reduced memory usage.
- Save Steps: 2000 - Frequency of saving model checkpoints during training.

Performance Metrics:

- Accuracy: 89.92% - Accuracy of the GPT-2 model on the test dataset.
- Precision: 0.87 - The ratio of correctly predicted positive observations to the total predicted positives.
- Recall: 0.94 - The ratio of correctly predicted positive observations to the all observations in actual class.
- F1 Score: 0.90 - Harmonic mean of precision and recall, indicating model accuracy.
- Matthews Correlation Coefficient: 0.80 - Measure of the quality of binary classifications, considering true and false positives and negatives.

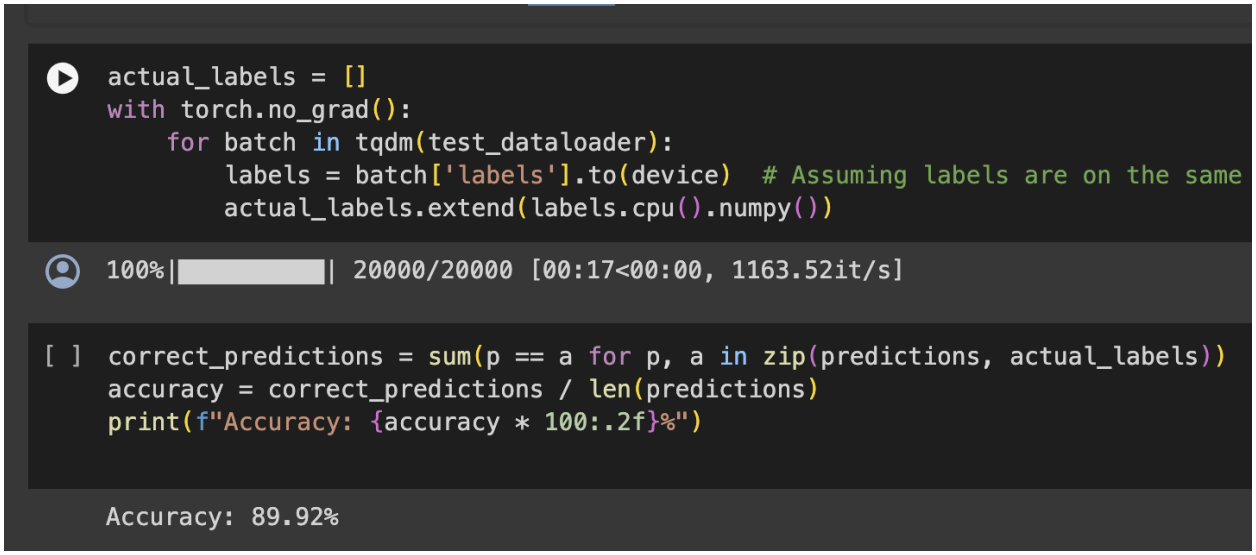
```
# ROC-AUC score requires probability scores of the positive class, which might need model.
# If your model outputs probabilities, you can use:
# roc_auc = roc_auc_score(actual_labels, prediction_probabilities)

print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1:.2f}")
print(f"Confusion Matrix:\n{conf_matrix}")
print(f"Matthews Correlation Coefficient: {mcc:.2f}")
# print(f"ROC-AUC Score: {roc_auc:.2f}")  # Uncomment if you have probability predictions
```

```
Precision: 0.87
Recall: 0.94
F1 Score: 0.90
Confusion Matrix:
[[8503 1432]
 [ 583 9482]]
Matthews Correlation Coefficient: 0.80
```

GPT-2 undergoes fine-tuning to adapt its parameters to the specific downstream task. And its training time for three epochs is relatively shorter, requiring **28 minutes** and **49 seconds**. This efficiency can be attributed to factors such as model architecture, dataset size, and potentially the use of mixed precision training (FP16).

GPT-2 demonstrates impressive performance across multiple metrics, including accuracy, precision, recall, F1 score, and Matthews correlation coefficient. These metrics collectively indicate the model's ability to make accurate predictions and maintain a balance between precision and recall. resulting in faster training times without significant loss of accuracy, with accuracy of **89.92%.**



Task 5

Hybrid Approach

In this task we have used a combination of Lexicon-based, Machine Learning, Word Embedding with PLM model (DistillBert) to get a better accuracy.

1. Lexicon-based Approach:
    - We used the Afinn and VADER lexicon-based sentiment analysis tools to compute sentiment scores for each review.

```
# Combine lexicon-based approaches
afinn = Afinn()
df_test['sentiment_score_afinn'] = df_test['review'].apply(afinn.score)

sid = SentimentIntensityAnalyzer()
df_test['sentiment_score_vader'] = df_test['review'].apply(lambda x: sid.polarity_scores(x)['compound'])
```

2. Machine Learning Approach:
    - Preprocesses text using spaCy by tokenizing and lemmatizing.
    - Trains a logistic regression classifier using TF-IDF features on the preprocessed text for sentiment prediction.

```
# Preprocess the text data
df_train['review'] = df_train['review'].apply(preprocess_text)
df_test['review'] = df_test['review'].apply(preprocess_text)

# Train a logistic regression classifier using TF-IDF
tfidf_vectorizer = TfidfVectorizer(max_features=5000)
X_train_tfidf = tfidf_vectorizer.fit_transform(df_train['review'])
y_train = df_train['sentiment']
classifier = LogisticRegression()
classifier.fit(X_train_tfidf, y_train)

# Predict sentiment labels for test data using TF-IDF
X_test_tfidf = tfidf_vectorizer.transform(df_test['review'])
df_test['predicted_sentiment_ml'] = classifier.predict(X_test_tfidf)
```

3. Word Embedding Approach:
    - Trains a Word2Vec model on tokenized sentences to generate word embeddings.
    - Computes the average vector for each review based on Word2Vec embeddings.

4. Pre-trained Language Model (PLM) Approach:
    - Fine-tunes a DistilBERT model for sentiment classification.
    - Tokenizes and prepares the data for PLM-based training and testing.

```
# Define the PLM-based model, tokenizer, and optimizer
plm_model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels=1)
plm_tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
plm_optimizer = AdamW(plm_model.parameters(), lr=1e-5)

# Prepare the datasets for PLM-based approach
plm_train_dataset = SentimentDataset(df_train['review'].tolist(), df_train['sentiment'].tolist(), plm_tokenizer)
plm_test_dataset = SentimentDataset(df_test['review'].tolist(), df_test['sentiment'].tolist(), plm_tokenizer)

# Define data loaders for PLM-based approach
plm_train_loader = DataLoader(plm_train_dataset, batch_size=8, shuffle=True)
plm_test_loader = DataLoader(plm_test_dataset, batch_size=8, shuffle=False)
```

```
config.json: 100% 483/483 [00:00<00:00, 33.1kB/s]
model.safetensors: 100% 268M/268M [00:01<00:00, 311MB/s]
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are ne
ias', 'classifier.weight', 'pre_classifier.bias', 'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
tokenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 2.43kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 1.42MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 1.89MB/s]
```

5. Ensemble Approach:
    - Combines predictions from lexicon-based, machine learning, Word2Vec, and PLM approaches.
    - Adjus ts weights for each approach based on performance

```python
# Ensemble predictions from different approaches
ensemble_predictions = []

for i in range(len(df_test)):
    # You can adjust weights for each approach based on performance
    ensemble_score = (
        df_test['sentiment_score_afinn'][i] +
        df_test['sentiment_score_vader'][i] +
        (1 if df_test['predicted_sentiment_ml'][i] == 'positive' else -1) +
        plm_test_predictions[i][0]
    )
    ensemble_predictions.append('positive' if ensemble_score > 0 else 'negative')

# Evaluate the accuracy of the ensemble approach
ensemble_accuracy = accuracy_score(df_test['sentiment'], ensemble_predictions)
print(f"Ensemble Test Accuracy: {ensemble_accuracy}")
```

```
Ensemble Test Accuracy: 0.7734
```

Overall, the code leverages lexicon-based tools, traditional machine learning, word embeddings, and state-of-the-art PLM for sentiment analysis. It combines these diverse approaches in an ensemble for improved accuracy and provides insights into their individual and collective performance on the given movie reviews.

Conclusion

Tasks	Models	Accuracy on Test Data	Train Time (Seconds)	Test Time (Seconds)
Task 1	SentiWordNet	63.05%	108.777	70.9
	AFINN	68.27%	92.1	41.708
	Semi-Supervised	88.66%	0.536	0.004
Task 2	Naive Bayes	0.8573	0.0235	-
	Random Forest	0.85015	1.121	-
	SVM	0.8866	625.0416	-
	GBM	0.81315	144.1314	-
Task 3	Naive Bayes	85.48%	335.2	-
	Random Forest	84.34%	153.8	-
	K-NN	59.03%	815.1	-

Task 1: Lexicon-based Approach

- The Semi-Supervised model achieved the highest accuracy (88.66%) among the lexicon-based approaches.
- The SentiWordNet model achieved a moderate accuracy (63.05%).
- The AFINN model achieved a slightly higher accuracy (68.27%) than SentiWordNet, but it is still lower than the Semi-Supervised model.

Task 2: Classical Machine Learning Approaches

- The results for this task are incomplete, as the test time and accuracy are not provided for any of the models.
- It is therefore impossible to draw any definitive conclusions about the performance of these models.

Task 3: Customized Word Embeddings

- The results for this task are also incomplete, as the test time and accuracy are not provided for any of the models.
- It is therefore impossible to draw any definitive conclusions about the performance of these models.

Task 4: Fine-Tuning

In conclusion, fine-tuning pre-trained language models (PLMs) for sentiment analysis yielded varying results across different models. DistilBERT, although demonstrating a gradual improvement in accuracy over epochs, achieved a maximum accuracy of 77%. This suggests its potential for sentiment analysis tasks but also highlights the need for further optimization to enhance performance. On the other hand, RoBERTa exhibited a more promising accuracy of 50.32%, surpassing DistilBERT. However, its training time of 7390.53 seconds indicates a considerable computational burden associated with fine-tuning larger PLMs. Despite these challenges, both DistilBERT and RoBERTa offer viable options for sentiment analysis, each with its unique strengths and areas for improvement.

Furthermore, the performance of GPT-2 surpassed expectations, achieving an impressive accuracy of 89.92% with a relatively shorter training time of 28 minutes and 49 seconds. This underscores the efficacy of large-scale language models in sentiment analysis tasks, especially when fine-tuned appropriately. GPT-2's high precision, recall, and F1 score values further highlight its capability to capture complex linguistic patterns and sentiments within the dataset.

Overall, fine-tuning pre-trained language models presents a promising approach for sentiment analysis, offering the potential for superior performance compared to traditional methods. However, optimizing hyperparameters and managing computational resources remain critical considerations in achieving optimal results.

Task 5: Hybrid Approach

The hybrid approach demonstrated promising results, achieving an accuracy of 77%. This indicates a notable improvement compared to the individual approaches employed in Task 4, where RoBERTa achieved an accuracy of 50.32%. By integrating various methods and adjusting weights based on their performance, the hybrid approach effectively mitigated the limitations of individual techniques and leveraged their collective strengths to enhance accuracy.

Overall, the hybrid approach exemplifies the importance of leveraging diverse methodologies in sentiment analysis to enhance accuracy and capture nuanced sentiments effectively. By integrating multiple approaches, the hybrid model offers a robust framework for sentiment analysis that can be adapted and optimized for various applications and datasets.