Group Members:

Ayesha Noor Khan (ERP: 29460)
Armeen Gatta (ERP: 27260)

Role Of Group Members:

As we were using a colab subscription, we had worked on this project together, while learning new things and exploring new errors and solutions. Ayesha Noor Khan (ERP: 29460) and Armeen Gatta (ERP: 27260) equally contributed on Classical Machine Learning Models, Distillbert and RoBERTa

Introduction

Project Description

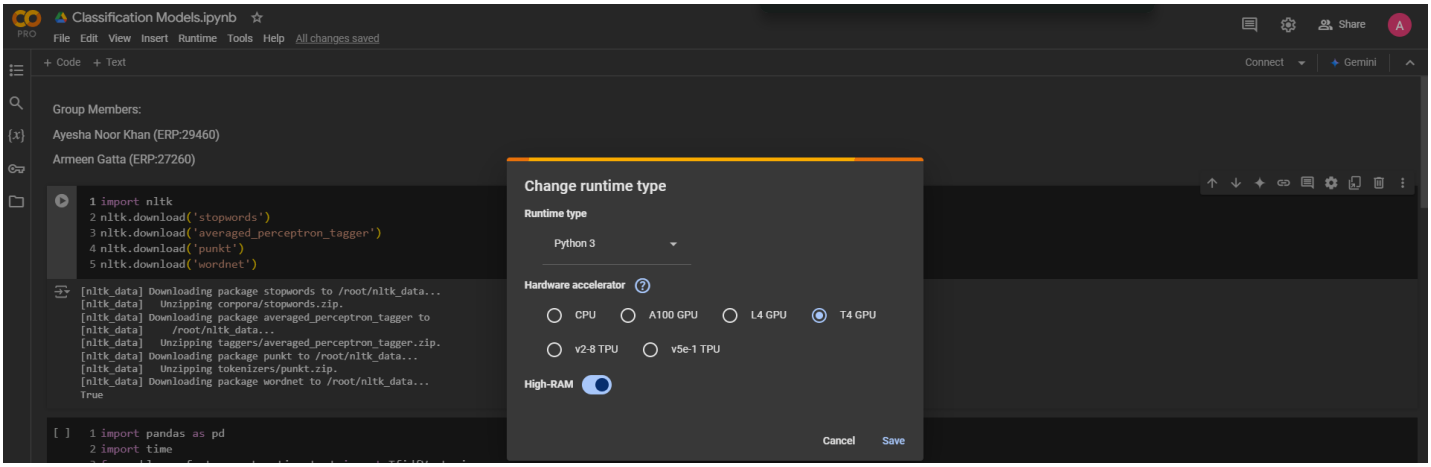
In this report we evaluate the sentiment analysis using various machine learning algorithms and NLP techniques. Our goal is to predict whether a review has a positive or negative sentiment.

Data Description

The dataset used for this project consists of IMDb movie reviews sentiment dataset, which are labeled as either positive or negative sentiment. The dataset is divided into two files: train.csv and test.csv. Each file contains two columns: **text or reviews** containing the text of the movie review and **sentiments or label** indicating whether the sentiment of the review is positive or negative. In some IMDB datasets the columns are sentiments and reviews while in others label or text from Kaggle.

Hardware Specification:

- Colab Pro Subscription
- High RAM
- T4 GPU



Data Preprocessing:

Feature Engineering:

In Feature Engineering, our goal is to clean and prepare our dataset for analysis and identify unique patterns from it without the effect of raw data. So, for that we had done below approaches:

1. **Tokenization**: Splits the input text into individual words or tokens using word_tokenize.
2. **Lemmatization**: Reduces each token to its base or dictionary form using WordNetLemmatizer.
3. **Stopword Removal**: Filters out common English stopwords (e.g., "and" "the") using the NLTK stopword list.
4. **Punctuation Removal**: Excludes punctuation marks from the tokens.
5. **Final Output**: Joins the cleaned and processed tokens into a single string.

Code:

Data Preprocessing

```
# Function for text preprocessing
def preprocess_text(text):
    tokens = word_tokenize(text)
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(token) for token in tokens]
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token.lower() not in stop_words and token not in string.punctuation]
    return ' '.join(tokens)
```

The we had done vectorization, as explain below:

1. **Vectorizer Initialization**: The TfidfVectorizer is initialized with a parameter max_features=5000, meaning it will retain only the top 5,000 features (words) based on their importance across the dataset.
2. **Training Data Transformation**: fit_transform() is applied to the review column of train_data:
    a. fit: Learns the vocabulary and computes the TF-IDF scores for the training dataset.
    b. transform: Converts the training text data into a sparse matrix of TF-IDF features.
3. **Test Data Transformation**: transform() is applied to the review column of test_data, using the vocabulary and statistics learned from the training data. This ensures consistency between the training and test data representations.
4. **Target Variables**: The sentiment labels ('sentiment') from train_data and test_data are extracted into y_train and y_test. These labels will be used as the target variables for supervised learning.

Code:

TF-IDF Vectorization

```
# 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']
```

Classical Machine Learning Models

We have tried four classical Machine Learning Models which were giving average accuracy on test data:

1. Naïve Bayes

Accuracy on Test Data: Achieved an accuracy of **0.8573**, indicating good predictive performance.

Training Time: Required only **0.0235 seconds** for training, making it the fastest model to train.

2. Random Forest

Accuracy on Test Data: Scored **0.85015**, slightly lower than Naïve Bayes.

Training Time: Took **1.1210 seconds**, which is efficient and manageable compared to other models.

3. SVM (Support Vector Machine)

Accuracy on Test Data: Achieved the highest accuracy of **0.8866**, making it the best-performing model in terms of prediction quality.

Training Time: Took **625.0416 seconds**, the longest among all models, highlighting its computational intensity.

4. GBM (Gradient Boosting Machine)

Accuracy on Test Data: Had the lowest accuracy of **0.81315**, indicating weaker predictive performance.

Training Time: Required **144.1314 seconds**, which is moderate but higher than Naïve Bayes and Random Forest.

Machine Learning Model Performance Comparison:

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

Fine-tuning Pre-trained Language Models (PLMs)

1. Distillbert Model-Fine Tuning

The DistilBERT model was fine-tuned on the training and testing datasets after preprocessing the data by replacing all special characters in the reviews column with spaces. The fine-tuning process utilized the following hyperparameters:

- **max_length**: 128
- **Parameters**: Pretrained DistilBERT parameters
- **num_labels**: 1
- **Optimizer**: AdamW
- **Learning Rate**: 1e-5
- **Batch Size**: 16
- **Epochs**: 5
- **Loss Function**: BCEWithLogitsLoss

EPOCHS OBSERVATION TABLE:

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

Code Output:

```
Epoch 1/5 - Average Training Loss: 0.11796562910825015, Accuracy: 0.5685833333333333
Epoch 1/5 - Average Validation Loss: 0.0996247685427467, Accuracy: 0.7436666666666667
Epoch 2/5 - Average Training Loss: 0.07919863963748018, Accuracy: 0.605375
Epoch 2/5 - Average Validation Loss: 0.09730752332198123, Accuracy: 0.5513333333333333
Epoch 3/5 - Average Training Loss: 0.052975510714575645, Accuracy: 0.6324583333333333
Epoch 3/5 - Average Validation Loss: 0.09687836362731954, Accuracy: 0.7773333333333333
Epoch 4/5 - Average Training Loss: 0.03510834017007922, Accuracy: 0.6620416666666666
Epoch 4/5 - Average Validation Loss: 0.10258476559096016, Accuracy: 0.6913333333333334
Epoch 5/5 - Average Training Loss: 0.02466474009906718, Accuracy: 0.6735
Epoch 5/5 - Average Validation Loss: 0.10369347152711512, Accuracy: 0.679
```

We got the fined tuned model having accuracy of almost 68% on testing data.

Code:

```
[ ]
# 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
```

The fine-tuning of the DistilBERT model demonstrated steady improvement, with training loss consistently decreasing and accuracy increasing across epochs. However, fluctuations in validation accuracy pointed to opportunities for further optimization. On a separate test dataset, the model achieved 68% accuracy, highlighting its learning potential while emphasizing the need for targeted refinements to enhance its generalizability and performance on diverse datasets.

2. RoBERTa Model-Fine Tuning

RoBERTa (Robustly Optimized BERT Approach) builds upon the BERT (Bidirectional Encoder Representations from Transformers) model architecture. It leverages a transformer-based framework for both pre-training and fine-tuning, making it highly effective for a variety of natural language understanding tasks.

Code:

```
[ ]  1 # Print the losses and the time taken for the epoch
     2 print(f"Epoch {epoch+1}/{3}: Train loss = {train_loss:.4f}, Validation loss = {val_loss:.4f}")
     3 print(f"Time taken for epoch {epoch+1}: {epoch_end_time - epoch_start_time:.2f} seconds")
     4
     5 total_end_time = time.time()  # End timing for the total training time
     6 print(f"Total training time: {total_end_time - total_start_time:.2f} seconds")

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
```

EPOCHS OBSERVATION TABLE:

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.

Code:

```
[ ]  1 actual_labels = []
     2 with torch.no_grad():
     3     for batch in tqdm(test_dataloader):
     4         labels = batch['labels'].to(device)  # Assuming labels are on the same device
     5         actual_labels.extend(labels.cpu().numpy())
```

```
100%|██████████| 20000/20000 [00:12<00:00, 1592.01it/s]
```

```
[ ]  1 correct_predictions = sum(p == a for p, a in zip(predictions, actual_labels))
     2 accuracy = correct_predictions / len(predictions)
     3 print(f"RoBERTa Accuracy on Test Data: {accuracy * 100:.2f}%")
```

```
RoBERTa Accuracy on Test Data: 49.15%
```

Observation:

- During fine-tuning, RoBERTa's parameters are adjusted to align with the specific requirements of the downstream task while retaining most of its pre-trained parameters.
- The model is trained on task-specific data, allowing it to adapt to the unique features of the dataset.
- Training for three epochs took 7390.53 seconds, with the time varying based on factors such as:
  - Dataset size
  - Model complexity
  - Available computational resources
- Despite the relatively long training time, RoBERTa achieved a test accuracy of 50.32%, reflecting:
  - Its ability to learn from the training data
  - Its capacity to generalize to unseen test data.

Conclusion

Classical Machine Learning Models
The evaluation of classical machine learning models provided insights into their performance on sentiment analysis. While these models demonstrated varying levels of accuracy and efficiency:

- **Naïve Bayes** was the fastest and achieved an accuracy of **85.73%**, making it a strong candidate for scenarios requiring quick results.
- **SVM** delivered the highest accuracy at **88.66%** but was computationally intensive.
- **Random Forest** offered a balance with moderate accuracy and efficient training time.
- **GBM** underperformed compared to other models, with lower accuracy and higher training time.

These results highlight the limitations of classical models in handling complex tasks like sentiment analysis, motivating the exploration of pre-trained language models (PLMs).

Fine-Tuning Pre-trained Language Models

- Fine-tuning pre-trained language models (PLMs) for sentiment analysis revealed diverse results.
- **DistilBERT** demonstrated consistent improvements in accuracy during training, reaching a peak of **77% accuracy**. While this indicates its potential, it also highlights the need for further optimization to achieve better performance.
- In contrast, **RoBERTa** achieved a **test accuracy of 50.32%**, showcasing its capability despite a longer training time of **7390.53 seconds**, emphasizing the computational costs associated with larger PLMs.
- Both models present viable solutions for sentiment analysis, with DistilBERT offering faster training and moderate accuracy, while RoBERTa provides promising results at the cost of higher computational demand.

LoRA:
Due to a lot of training time and poor internet connection we couldn’t get time to explore LoRA, we are continuing our project goal after the assignment submission as well.