Report:
Part 1: Classical Machine Learning approaches:

Purpose: To assess the accuracy of sentiment analysis on the IMDB Reviews test dataset by employing various machine learning algorithms, including Naive Bayes, Random Forest, and KNN. For vectorizing the reviews, two techniques have been utilised: count vectorizer and Tfidf vectorizer. Different parameters of these vectorizers have been explored to observe trends in accuracy.

Process:

Step no 1: Pre processing

Performed preprocessing by given code with following detail,

```
# Load IMDb dataset
imdb_data = pd.read_csv("train.csv")

def preprocess_text(text):
    if text is None:
        return ""
    try:
        # Remove HTML tags and special characters
        text = re.sub('<[^<]+?>', ' ', text)
        text = re.sub(r'\W', ' ', text)
        text = re.sub(r'\s+', ' ', text)
        text = text.lower()
        # Tokenization
        tokens = word_tokenize(text)
        # Remove stopwords
        stop_words = set(stopwords.words('english'))
        tokens = [word for word in tokens if word not in stop_words]
        # Lemmatization
        lemmatizer = WordNetLemmatizer()
        tokens = [lemmatizer.lemmatize(word) for word in tokens]
        # POS Tagging
        tokens = pos_tag(tokens)
        return ' '.join([word for word, tag in tokens if tag.startswith('N')])  # Keep only nouns
    except Exception as e:
        print("Error:", e)
        print("Input Text:", text)
        return ""  # Return empty string in case of error


imdb_data['preprocessed_text'] = imdb_data['review'].apply(preprocess_text)
```


- Remove HTML tags and special characters.Eliminates HTML tags and non-alphanumeric characters, converting text to lowercase.

- Tokenization: Splits text into individual words or tokens.

- Remove stopwords: Filters out common English stopwords (e.g., "the", "is") to focus on meaningful words.
- Lemmatization: Converts words to their base or dictionary form for normalization.

- POS Tagging (Part-of-Speech Tagging): Identifies the part of speech for each word and retains only nouns, which typically carry significant semantic meaning.

The resulting preprocessed text contains only nouns, making it cleaner and more suitable for sentiment analysis.

Step no 2: Vectorizing the reviews:
In this part we have converted the reviews in the form of vectors by count vectorizer and Tfidf vectorizer, using their different parameters options of ngram_range and max features. We have performed 7 experiments with different features for both vectorizers.

```
# Vectorize the text data
vectorizer = TfidfVectorizer(max_features=5000)  # You can experiment with different parameters
X_train = vectorizer.fit_transform(imdb_data['preprocessed_text'])
y_train = imdb_data['sentiment']

X_test = vectorizer.fit_transform(imdb_data_test['preprocessed_text'])
y_test = imdb_data_test['sentiment']
```

```
# Initialize the CountVectorizer
vectorizer = CountVectorizer(max_features=5000,  # Limits the number of features
                             ngram_range=(1, 2),  # Considers unigrams and bigrams
                             stop_words='english',  # Removes English stopwords
                             max_df=0.8,  # Ignores terms with document frequency > 80%
                             min_df=5  # Ignores terms with document frequency < 5
                            )
```

Step no 3: Training the parameters by Machine Learning:
In this part we trained the model of three ML algorithms using train data with default settings, by using

```
# Train machine learning models
nb_classifier = MultinomialNB()
nb_classifier.fit(X_train, y_train)

rf_classifier = RandomForestClassifier()
rf_classifier.fit(X_train, y_train)

knn_classifier = KNeighborsClassifier()
knn_classifier.fit(X_train, y_train)

# Evaluate model performance
nb_pred = nb_classifier.predict(X_test)
rf_pred = rf_classifier.predict(X_test)
knn_pred = knn_classifier.predict(X_test)

nb_accuracy5 = accuracy_score(y_test, nb_pred)
rf_accuracy5 = accuracy_score(y_test, rf_pred)
knn_accuracy5 = accuracy_score(y_test, knn_pred)

print("Naive Bayes Accuracy:", nb_accuracy5)
print("Random Forest Accuracy:", rf_accuracy5)
print("k-NN Accuracy:", knn_accuracy5)
```

Results: Following are the results using above algorithmic steps:

	Settings	Naive Bayes Accuracy	Random Forest Accuracy	k-NN Accuracy
0	Tfidf vectorizer(max features=5000,ngram_range=(1,1))	0.550850	0.548250	0.497150
1	Tfidf vectorizer(max features=6000),ngram_range=(1,1)	0.532150	0.485200	0.496850
2	Tfidf vectorizer(max features=4000,,ngram_range=(1,1))	0.528900	0.500600	0.496500
3	Tfidf vectorizer(max features=5000,,ngram_range=(1,2))	0.531150	0.513850	0.508250
4	Count vectorizer(max features=5000,ngram_range=(1,1))	0.571650	0.523550	0.495850
5	Count vectorizer(max features=5000,ngram_range=(1, 2))	0.545850	0.530900	0.522450
6	Tfidf vectorizer(max features=5000, ngram_range=(1,3))	0.561350	0.524150	0.506050

Observations:

- CountVectorizer tends to achieve higher accuracies compared to TfidfVectorizer across most configurations.
- Naive Bayes generally outperforms Random Forest and k-NN for most configurations and vectorizers.
- The choice of max_features and ngram_range influences the performance of both vectorizers.
- Out of 7 Experiments 5 for Tfidf vectorizer and 2 for count vectorizer the best accuracy is 0.561350 for Tfidf vectorizer by Naive Bayes with 5000 maximum features and ngram_range(1,3) while the count vectorizer is showing best accuracy again with Naive Bayes at n_gram_range(1,1) and max_features=5000.

Challenges:

In this computational experiment KNN was continuously giving error that was handled by !pip install threadpoolctl==3.1.0, to overcome the dependencies conflict between sklearn and numpy.

Comparison:

Accuracy comparison of ML models has been conducted using "distilbert-base-uncased-finetuned-sst-2-english," which exhibits a notably high accuracy of 0.8891. This indicates poor performance of the ML model for sentiment classification on the IMDB reviews dataset.

```
<ipython-input-5-e34ed8bfc253>:8: FutureWarning: The error_bad_lines argument has been deprecated and will be removed in a

  test_df = pd.read_csv("test.csv", error_bad_lines=False)
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), s
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  28.0/28.0 [00:00<00:00, 877B/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 4.56MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 7.17MB/s]
config.json: 100%  483/483 [00:00<00:00, 7.47kB/s]
config.json: 100%  629/629 [00:00<00:00, 18.7kB/s]
model.safetensors: 100%  268M/268M [00:01<00:00, 178MB/s]
Test Accuracy: 0.8891
```

Part 2: Fine-tuning Pre-trained Language Models (PLMs):

b) Roberta:

Purpose: To assess the accuracy of sentiment analysis on the IMDB Reviews test dataset by fine tuned Roberta model using IMDB Reviews train dataset.and compare the accuracy with the "distilbert-base-uncased-finetuned-sst-2-english," on test data.

Process:

Step no 1: Loading model and tokenizer:
After importing the required libraries, we have loaded the Roberta-base model with its tokenizer by using following code.

```
# Load the pre-trained models and tokenizers
roberta_model =
AutoModelForSequenceClassification.from_pretrained("roberta-base")
roberta_tokenizer = AutoTokenizer.from_pretrained("roberta-base")
# print memory footprint
print("Memory footprint of roberta_model: ",
roberta_model.num_parameters() * 4 / 1024 / 1024, "MB")
```

Step no 2: Tokenize the train and test data:
Using the following code chunk we transformed the test and train reviews into tokenized form by using RoBERTa tokenizer. And converted the sentiments into numeric form.

```python
# Load the dataset from CSV files
dataset= load_dataset("csv", data_files={"train": "train.csv", "test": "test.csv"})

# Tokenize the text data
def tokenize_function(example):
    return roberta_tokenizer(example["review"], padding=True, truncation=True)

# Tokenize train and test datasets
train_dataset = dataset["train"].map(tokenize_function, batched=True)
test_dataset = dataset["test"].map(tokenize_function, batched=True)

# Optionally, convert labels to numerical format if necessary
# For example, if sentiment labels are strings ("positive", "negative"), convert them to integers (0, 1)
train_dataset = train_dataset.map(lambda examples: {"label": 1 if examples["sentiment"] == "positive" else 0})
test_dataset = test_dataset.map(lambda examples: {"label": 1 if examples["sentiment"] == "positive" else 0})
```

```
Generating train split: 30000/0 [00:00<00:00, 35766.56 examples/s]
Generating test split: 20000/0 [00:00<00:00, 35963.60 examples/s]
Map: 100% 30000/30000 [00:46<00:00, 670.35 examples/s]
Map: 100% 20000/20000 [00:20<00:00, 1045.33 examples/s]
Map: 100% 30000/30000 [00:03<00:00, 8612.31 examples/s]
Map: 100% 20000/20000 [00:01<00:00, 12442.37 examples/s]
```

Step no 3: Finetune RoBERTa model at train data:

For fine tuning the RoBERTa model we used the following values of hyperparameters.

Batch_size=8

Epoch=3

optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

For further details code snippet is attached,

```python
# Define the batch size and create DataLoader
BATCH_SIZE = 8
train_dataset = TensorDataset(input_ids, attention_masks, labels)
train_sampler = RandomSampler(train_dataset)
train_dataloader = DataLoader(train_dataset, sampler=train_sampler, batch_size=BATCH_SIZE)

# Specify GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Define the optimizer
optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

# Fine-tuning the model
EPOCHS = 3

for epoch in range(EPOCHS):
    model.train()

    # Initialize progress bar
    progress_bar = tqdm(train_dataloader, desc=f'Epoch {epoch + 1}/{EPOCHS}', leave=False, disable=False)

    for batch in progress_bar:
        batch = tuple(t.to(device) for t in batch)
        inputs = {'input_ids': batch[0],
                  'attention_mask': batch[1],
                  'labels': batch[2]}

        optimizer.zero_grad()

        outputs = model(**inputs)
        loss = outputs.loss
        loss.backward()

        optimizer.step()
```

After training for 4 hours we got the fine_tuned model of RoBERTa.

```
Some weights of RobertaForSequenceClassification were not initialized from the model checkpoint at
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:411: FutureWarning: This imple
  warnings.warn(
('/content/drive/My Drive/fine_tuned_roberta_model/tokenizer_config.json',
 '/content/drive/My Drive/fine_tuned_roberta_model/special_tokens_map.json',
 '/content/drive/My Drive/fine_tuned_roberta_model/vocab.json',
 '/content/drive/My Drive/fine_tuned_roberta_model/merges.txt',
 '/content/drive/My Drive/fine_tuned_roberta_model/added_tokens.json')
```

Step no 4: Checking Accuracy at test data:

Last step is to check the accuracy of the fine tuned RoBERTa model at test data, which came to be 0.9413.

```
model.eval()
predictions = []
true_labels = []

for batch in test_dataloader:
    batch = tuple(t.to(device) for t in batch)
    inputs = {'input_ids': batch[0], 'attention_mask': batch[1], 'labels': batch[2]}

    with torch.no_grad():
        outputs = model(**inputs)

    logits = outputs.logits
    predictions.extend(torch.argmax(logits, dim=1).cpu().tolist())
    true_labels.extend(inputs['labels'].cpu().tolist())

# Calculate accuracy
accuracy = accuracy_score(true_labels, predictions)
print(f"Test Accuracy: {accuracy}")
```

```
Test Accuracy: 0.9413
```

Results:

Test accuracy is reported to be 0.9413 by Finetuned RoBERTa model.

Comparison:

Accuracy comparison of Fine tuned Roberta model has been conducted using "distilbert-base-uncased-finetuned-sst-2-english," which exhibits comparatively low accuracy of 0.8891. This indicates good performance of the Finetuned Roberta model for sentiment classification on the IMDB reviews dataset.