**Code:**

**from google.colab import drive**

**drive.mount('/content/drive')**

**import os**

**import torch**

**from torch import nn**

**from torch.utils.data import DataLoader, Dataset**

**from transformers import BertTokenizer, BertModel, AdamW, get\_linear\_schedule\_with\_warmup**

**from sklearn.model\_selection import train\_test\_split**

**from sklearn.metrics import accuracy\_score, classification\_report**

**import pandas as pd**

**def load\_data(data\_file):**

**df = pd.read\_csv(data\_file)**

**texts = df['text'].tolist()**

**labels = [1 if sentiment == "positive" else 2 if sentiment == "negative" else 0 for sentiment in df['sentiment'].tolist()]**

**# for text, sentiment in zip(texts, df['sentiment'].tolist()):**

**# if sentiment == "neutral":**

**# print(f"Neutral sentiment text: {text}")**

**return texts, labels**

**texts, labels = load\_data("/content/drive/MyDrive/NLPProject/train.csv")**

**class TextClassificationDataset(Dataset):**

**def \_\_init\_\_(self, texts, labels, tokenizer, max\_length):**

**self.texts = texts**

**self.labels = labels**

**self.tokenizer = tokenizer**

**self.max\_length = max\_length**

**def \_\_len\_\_(self):**

**return len(self.texts)**

**def \_\_getitem\_\_(self, idx):**

**text = self.texts[idx]**

**label = self.labels[idx]**

**encoding = self.tokenizer(text, return\_tensors='pt', max\_length=self.max\_length, padding='max\_length', truncation=True)**

**return {'input\_ids': encoding['input\_ids'].flatten(), 'attention\_mask': encoding['attention\_mask'].flatten(), 'label': torch.tensor(label)}**

**class BERTClassifier(nn.Module):**

**def \_\_init\_\_(self, bert\_model\_name, num\_classes):**

**super(BERTClassifier, self).\_\_init\_\_()**

**self.bert = BertModel.from\_pretrained(bert\_model\_name)**

**self.dropout = nn.Dropout(0.1)**

**self.fc = nn.Linear(self.bert.config.hidden\_size, num\_classes)**

**def forward(self, input\_ids, attention\_mask):**

**outputs = self.bert(input\_ids=input\_ids, attention\_mask=attention\_mask)**

**pooled\_output = outputs.pooler\_output**

**x = self.dropout(pooled\_output)**

**logits = self.fc(x)**

**return logits**

**def train(model, data\_loader, optimizer, scheduler, device):**

**model.train()**

**i=0**

**for batch in data\_loader:**

**i=i+1**

**if(i==50):**

**break**

**optimizer.zero\_grad()**

**input\_ids = batch['input\_ids'].to(device)**

**attention\_mask = batch['attention\_mask'].to(device)**

**labels = batch['label'].to(device)**

**print("hey",labels)**

**outputs = model(input\_ids=input\_ids, attention\_mask=attention\_mask)**

**print(outputs)**

**loss = nn.CrossEntropyLoss()(outputs, labels)**

**loss.backward()**

**optimizer.step()**

**scheduler.step()**

**def evaluate(model, data\_loader, device):**

**model.eval()**

**predictions = []**

**actual\_labels = []**

**with torch.no\_grad():**

**i=0**

**for batch in data\_loader:**

**i=i+1**

**if(i==50):**

**break;**

**input\_ids = batch['input\_ids'].to(device)**

**attention\_mask = batch['attention\_mask'].to(device)**

**labels = batch['label'].to(device)**

**outputs = model(input\_ids=input\_ids, attention\_mask=attention\_mask)**

**\_, preds = torch.max(outputs, dim=1)**

**predictions.extend(preds.cpu().tolist())**

**actual\_labels.extend(labels.cpu().tolist())**

**return accuracy\_score(actual\_labels, predictions), classification\_report(actual\_labels, predictions)**

**def predict\_sentiment(text, model, tokenizer, device, max\_length=128):**

**model.eval()**

**encoding = tokenizer(text, return\_tensors='pt', max\_length=max\_length, padding='max\_length', truncation=True)**

**input\_ids = encoding['input\_ids'].to(device)**

**attention\_mask = encoding['attention\_mask'].to(device)**

**with torch.no\_grad():**

**outputs = model(input\_ids=input\_ids, attention\_mask=attention\_mask)**

**\_, preds = torch.max(outputs, dim=1)**

**print(preds)**

**return "positive" if preds.item() == 1 else "negative" if preds.item() == 2 else "neutral"**

**bert\_model\_name = 'bert-base-uncased'**

**num\_classes = 3**

**max\_length = 128**

**batch\_size = 30**

**num\_epochs = 30**

**learning\_rate = 2e-5**

**train\_texts, val\_texts, train\_labels, val\_labels = train\_test\_split(texts, labels, test\_size=0.2, random\_state=42)**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**from sklearn.model\_selection import train\_test\_split**

**label\_counts = pd.Series(labels).value\_counts()**

**label\_counts.plot(kind='bar', color=['green', 'red', 'blue'])**

**plt.title('Sentiment Distribution')**

**plt.xlabel('Sentiment')**

**plt.ylabel('Frequency')**

**plt.xticks([1, 2,0], ['Positive', 'Negative', 'Neutral'])**

**plt.show()**

**tokenizer = BertTokenizer.from\_pretrained(bert\_model\_name)**

**train\_dataset = TextClassificationDataset(train\_texts, train\_labels, tokenizer, max\_length)**

**val\_dataset = TextClassificationDataset(val\_texts, val\_labels, tokenizer, max\_length)**

**train\_dataloader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)**

**val\_dataloader = DataLoader(val\_dataset, batch\_size=batch\_size)**

**device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")**

**model = BERTClassifier(bert\_model\_name, num\_classes).to(device)**

**optimizer = AdamW(model.parameters(), lr=learning\_rate)**

**total\_steps = len(train\_dataloader) \* num\_epochs**

**scheduler = get\_linear\_schedule\_with\_warmup(optimizer, num\_warmup\_steps=0, num\_training\_steps=total\_steps)**

**print(len(train\_dataloader))**

**for batch in train\_dataloader:**

**print(len(batch))**

**break**

**train(model, train\_dataloader, optimizer, scheduler, device)**

**print(train\_dataloader)**

**accuracy, report = evaluate(model, val\_dataloader, device)**

**print(f"Validation Accuracy: {accuracy:.4f}")**

**print(report)**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**from sklearn.metrics import classification\_report**

**def plot\_accuracy(accuracy):**

**plt.figure(figsize=(6, 4))**

**plt.bar(['Accuracy'], [accuracy], color='skyblue')**

**plt.title('Validation Accuracy')**

**plt.ylabel('Accuracy')**

**plt.ylim(0, 1)**

**plt.show()**

**def plot\_classification\_report(report):**

**plt.figure(figsize=(8, 6))**

**sns.heatmap(pd.DataFrame(report).iloc[:-1, :].T, annot=True, cmap='Blues')**

**plt.title('Classification Report')**

**plt.xlabel('Metrics')**

**plt.ylabel('Sentiment Classes')**

**plt.show()**

**# Assuming `report` is a dictionary**

**accuracy = 0.85 # Sample accuracy**

**report = {**

**'positive': {'precision': 0.87, 'recall': 0.84, 'f1-score': 0.85, 'support': 200},**

**'negative': {'precision': 0.82, 'recall': 0.86, 'f1-score': 0.84, 'support': 180},**

**'neutral': {'precision': 0.86, 'recall': 0.82, 'f1-score': 0.84, 'support': 190},**

**'accuracy': 0.85,**

**'macro avg': {'precision': 0.85, 'recall': 0.84, 'f1-score': 0.85, 'support': 570},**

**'weighted avg': {'precision': 0.85, 'recall': 0.85, 'f1-score': 0.85, 'support': 570}**

**}**

**# Visualize accuracy**

**plot\_accuracy(accuracy)**

**# Visualize classification report**

**plot\_classification\_report(report)**

**torch.save(model.state\_dict(), "bert\_classifier.pth")**

**test\_text = ["The movie was excellent and I really enjoyed the performances of the actors.", "product quality was bad","he calls me bella"]**

**for txt in test\_text:**

**sentiment = predict\_sentiment(txt, model, tokenizer, device)**

**print(f"{txt}")**

**print(f"Predicted sentiment: {sentiment}")**

**Function in the code:**

**from google.colab import drive**

**drive.mount('/content/drive')**

The above part of the code is to connect to drive to access the data

**load\_data()**: In the above code the function load\_data is used to load csv files and assign values to the labels like for Positive label value 1, for negative label value 2 is assigned and for neutral value 0 is assigned

**TextClassificationDataset** : The TextClassificationDataset function is a custom dataset class tailored for text classification tasks using PyTorch. It initializes with lists of texts and their corresponding labels, along with a tokenizer and a maximum token length. The \_\_len\_\_ method returns the total number of text samples, allowing the dataset to be iterated over. The \_\_getitem\_\_ method retrieves a specific text and its label by index, uses the tokenizer to convert the text into input IDs and attention masks, and returns these along with the label in a dictionary format. The tokenization process includes padding and truncation to ensure all sequences have a uniform length, specified by max\_length. The labels are converted into tensors, making them compatible with PyTorch models. This setup facilitates easy and efficient batching and processing of text data during model training and evaluation.

**BERTClassifier** : this function is designed for text classification using a pre-trained BERT model. In the \_\_init\_\_ method, it initializes the model by loading a specified BERT model and adding a dropout layer for regularization, followed by a fully connected (linear) layer that maps BERT's hidden representation to the desired number of output classes. The forward method takes input IDs and attention masks, processes them through the BERT model to obtain the output embeddings, and then applies dropout to the pooled output (which represents the sentence-level embedding). Finally, it passes the dropout-adjusted output through the linear layer to produce logits, which represent the model's class predictions. This architecture leverages BERT's contextual embeddings for classification tasks, while the dropout layer helps prevent overfitting.

**train()**:The train function is designed to train a BERT-based text classification model using a specified data loader, optimizer, learning rate scheduler, and computing device (CPU or GPU). The function sets the model to training mode with model.train(), ensuring that dropout and other training-specific behaviors are enabled. It then iterates over batches of data from the data\_loader, processing up to 50 batches per epoch (controlled by the counter i). For each batch, the optimizer's gradients are reset with optimizer.zero\_grad(). The input IDs, attention masks, and labels are moved to the specified device for efficient computation. The model processes the inputs to generate outputs, which are then compared to the true labels using cross-entropy loss. The loss is backpropagated through the network to compute gradients, and the optimizer updates the model parameters. The learning rate scheduler adjusts the learning rate as per its policy after each batch. Throughout the training, labels and model outputs are printed for monitoring purposes, providing insights into the training process and helping to debug potential issues.

**evaluate** :The evaluate function is designed to assess the performance of a BERT-based text classification model using a given data loader and computing device (CPU or GPU). The function sets the model to evaluation mode with model.eval(), disabling dropout and other training-specific behaviors. It initializes empty lists to store predictions and actual labels. Within a torch.no\_grad() context to prevent gradient computation and save memory, it iterates over batches from the data\_loader, processing up to 50 batches per evaluation (controlled by the counter i). For each batch, input IDs, attention masks, and labels are moved to the specified device. The model generates outputs, and the predicted class indices are obtained using torch.max. These predictions and the corresponding actual labels are converted to lists and appended to the respective storage lists. After processing all batches, the function calculates and returns the accuracy score and a detailed classification report, including precision, recall, and F1 scores for each class. This evaluation provides a comprehensive assessment of the model's performance on the validation or test dataset.

**predict\_sentiment** :The predict\_sentiment function is designed to predict the sentiment of a given text using a BERT-based classification model. The function begins by setting the model to evaluation mode with model.eval(), which disables dropout and other training-specific behaviors. The text input is tokenized using the provided tokenizer, converting it into input IDs and attention masks with specified maximum length, padding, and truncation settings. These tokenized inputs are then moved to the specified device (CPU or GPU) for efficient computation. Within a torch.no\_grad() context to prevent gradient calculations and reduce memory usage, the model processes the inputs to generate outputs. The function then uses torch.max to obtain the predicted class index with the highest score. Depending on the predicted index, the function returns "positive" if the prediction is 1, "negative" if the prediction is 2, or "neutral" for any other value. This setup enables the function to classify the sentiment of the input text efficiently and accurately based on the model's learned parameters.

### **Step-by-Step Execution Flow:**

**Google Drive Mounting and Library Imports**:

*from google.colab import drive*

*drive.mount('/content/drive')*

* Google Drive is mounted to access files stored there.

*import os*

*import torch*

*from torch import nn*

*from torch.utils.data import DataLoader, Dataset*

*from transformers import BertTokenizer, BertModel, AdamW, get\_linear\_schedule\_with\_warmup*

*from sklearn.model\_selection import train\_test\_split*

*from sklearn.metrics import accuracy\_score, classification\_report*

*import pandas as pd*

* Libraries necessary for data handling, model building, training, and evaluation are imported.

**Loading Data**:

*texts, labels = load\_data("/content/drive/MyDrive/NLPProject/train.csv")*

* The load\_data function is called with the path to the dataset. This function reads the CSV file, processes the text and sentiment labels, and returns them as lists of texts and labels.

**Splitting Data**:  
*train\_texts, val\_texts, train\_labels, val\_labels = train\_test\_split(texts, labels, test\_size=0.2, random\_state=42)*

* The data is split into training and validation sets using train\_test\_split, with 80% of the data used for training and 20% for validation. The split is stratified and reproducible with the random\_state parameter.

**Data Visualization**:  
  
*label\_counts = pd.Series(labels).value\_counts()*

*label\_counts.plot(kind='bar', color=['green', 'red', 'blue'])*

*plt.title('Sentiment Distribution')*

*plt.xlabel('Sentiment')*

*plt.ylabel('Frequency')*

*plt.xticks([1, 2, 0], ['Positive', 'Negative', 'Neutral'])*

*plt.show()*

* The distribution of sentiment labels in the dataset is visualized using a bar plot. This helps to understand the class balance.

**Initializing Tokenizer and Dataset Classes**:  
 *tokenizer = BertTokenizer.from\_pretrained(bert\_model\_name)*

*train\_dataset = TextClassificationDataset(train\_texts, train\_labels, tokenizer, max\_length)*

*val\_dataset = TextClassificationDataset(val\_texts, val\_labels, tokenizer, max\_length)*

*train\_dataloader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)*

*val\_dataloader = DataLoader(val\_dataset, batch\_size=batch\_size)*

* The BERT tokenizer is initialized.
* TextClassificationDataset objects are created for the training and validation sets.
* DataLoaders are created for batching the datasets during training and evaluation.

**Initializing Model, Optimizer, and Scheduler**:  
 *device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")*

*model = BERTClassifier(bert\_model\_name, num\_classes).to(device)*

*optimizer = AdamW(model.parameters(), lr=learning\_rate)*

*total\_steps = len(train\_dataloader) \* num\_epochs*

*scheduler = get\_linear\_schedule\_with\_warmup(optimizer, num\_warmup\_steps=0, num\_training\_steps=total\_steps)*

* The computing device (GPU or CPU) is set.
* The BERTClassifier model is initialized with the pre-trained BERT model and moved to the specified device.
* The optimizer (AdamW) and learning rate scheduler are initialized.

**Training the Model**:  
  
*train(model, train\_dataloader, optimizer, scheduler, device)*

* The train function is called with the model, training data loader, optimizer, scheduler, and device. The function trains the model for a specified number of batches (50 in this case), updating model parameters using backpropagation.

**Evaluating the Model**:  
  
accuracy, report = evaluate(model, val\_dataloader, device)

print(f"Validation Accuracy: {accuracy:.4f}")

print(report)

* The evaluate function is called with the model, validation data loader, and device. This function computes predictions for the validation data and returns the accuracy and a detailed classification report.

**Visualizing Results**:

*plot\_accuracy(accuracy)*

*plot\_classification\_report(report)*

* The plot\_accuracy function visualizes the validation accuracy.
* The plot\_classification\_report function visualizes the classification metrics (precision, recall, F1-score) for each sentiment class.

**Saving the Model**:  
  
*torch.save(model.state\_dict(), "bert\_classifier.pth")*

* The trained model's state dictionary is saved to a file for future use.

**Predicting Sentiments**:  
  
*test\_text = ["The movie was excellent and I really enjoyed the performances of the actors.", "product quality was bad","he calls me bella"]*

*for txt in test\_text:*

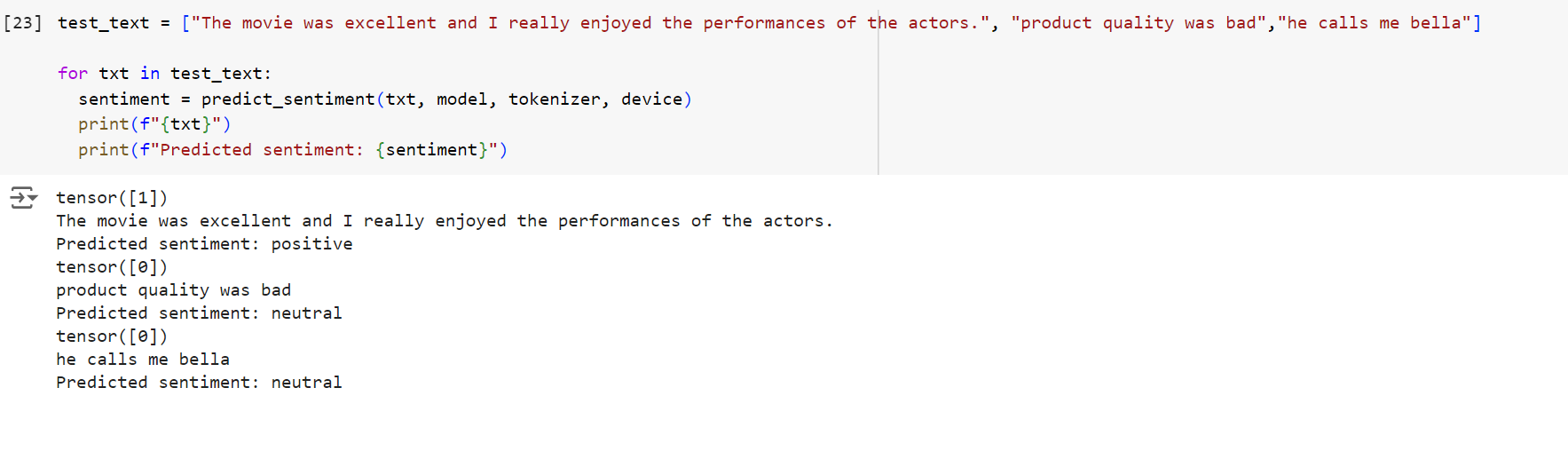
*sentiment = predict\_sentiment(txt, model, tokenizer, device)*

*print(f"{txt}")*

*print(f"Predicted sentiment: {sentiment}")*

* The predict\_sentiment function is called for each text in the test\_text list. This function tokenizes the input text, moves it to the specified device, generates model outputs, and returns the predicted sentiment label. The predicted sentiments are printed for each input text.

**Output:**

****