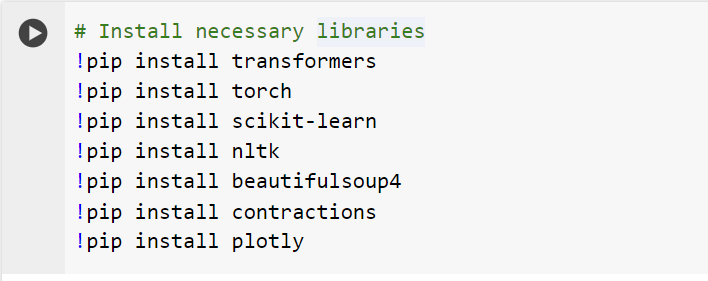
**Appendix N**

Google Colab Coding Guide

**Multiclass Text Classification using BERT, RoBERTa, and ALBERT**

**Library Installations**

****

**!pip install transformers:**

Installs the Hugging Face transformers library, which provides pre-trained models and tools for NLP tasks.

**!pip install torch:**

Installs PyTorch, a deep learning framework used for building and training neural networks.

**!pip install scikit-learn:**

Installs scikit-learn, a machine learning library for Python that provides simple and efficient tools for data mining and data analysis.

**!pip install nltk:**

Installs the Natural Language Toolkit (NLTK), a library for working with human language data.

**!pip install beautifulsoup4:**

Installs Beautiful Soup, a library for parsing HTML and XML documents.

**!pip install contractions:**

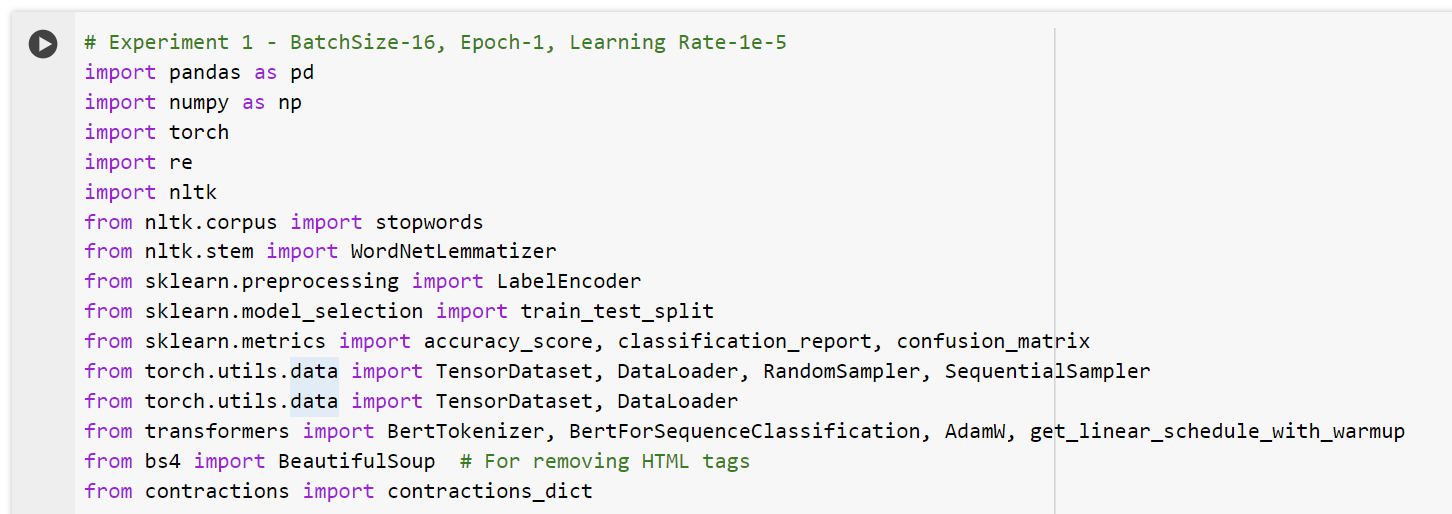
Installs the contractions library, which expands contractions in English text (e.g., "don't" to "do not").

**!pip install plotly:**

Installs Plotly, a graphing library that makes interactive, publication-quality graphs.

* Combining these libraries and imports allows for building a robust pipeline for multiclass text classification using BERT, including data preprocessing, model training, and evaluation.

**Library Imports**

****

**import pandas as pd:**

Imports the Pandas library, which is used for data manipulation and analysis.

**import numpy as np:**

Imports the NumPy library, which supports large multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

**import torch:**

Imports the PyTorch library for deep learning.

**import re:**

Imports the re module for regular expression operations.

**import nltk:**

Imports the NLTK library.

**from nltk.corpus import stopwords:**

Imports the list of stopwords from the NLTK corpus.

**from nltk.stem import WordNetLemmatizer:**

Imports the WordNetLemmatizer class from NLTK, which is used for lemmatizing words.

**from sklearn.preprocessing import LabelEncoder:**

Imports the LabelEncoder class from scikit-learn, which is used for encoding target labels with value between 0 and n\_classes-1.

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

Imports the train\_test\_split function from scikit-learn, which is used to split data into training and testing sets.

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

Imports evaluation metrics from scikit-learn: accuracy\_score, classification\_report, and confusion\_matrix.

**from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler:**

Imports classes and functions from PyTorch for handling data loading and sampling.

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

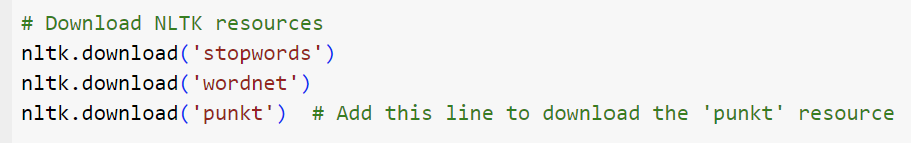
Imports the BERT tokenizer and model for sequence classification, as well as the AdamW optimizer and learning rate scheduler from the transformers library.

**from bs4 import BeautifulSoup:**

Imports the BeautifulSoup class from the Beautiful Soup library for parsing HTML and XML documents.

**from contractions import contractions\_dict:**

Imports the contractions dictionary from the contractions library for expanding contractions in text.



These lines of code are used to download specific resources from the NLTK (Natural Language Toolkit) library. Each resource serves a different purpose in text processing:

**Stopwords:**

Code: nltk.download('stopwords')

Description: Downloads the list of common stopwords in various languages. Stopwords are words that are commonly used in a language (such as "and", "the", "is") and are often removed during text preprocessing because they carry little meaningful information.

**WordNet:**

Code: nltk.download('wordnet')

Description: Downloads the WordNet database, which is a large lexical database of English. WordNet is used for lemmatization (reducing words to their base or root form), synonym detection, and other lexical tasks.

**Punkt:**

Code: nltk.download('punkt')

Description: Downloads the Punkt tokenizer models, which are used for tokenizing text into sentences and words. The Punkt tokenizer is a pre-trained, unsupervised machine-learning tokenizer that works well on a wide range of text.

**Purpose in the Multiclass Text Classification Task**

**Stopwords:** Useful for text preprocessing to remove common words that do not contribute significantly to the meaning of the text, helping to reduce noise in the data.

**WordNet:** Enables lemmatization, which reduces words to their base forms, making it easier to analyze and process text by reducing the number of unique words.

**Punkt:** Provides robust sentence and word tokenization, a crucial step in preparing text data for analysis and model training.



This block of code loads a CSV file into a Pandas DataFrame. Here's a detailed explanation:

**File Path**

Code: file\_path = "/content/drive/MyDrive/Dissertation\_UC/UAQTE\_Experience\_Multi\_Class\_TC\_Datasets.csv"

Description: Specifies the path to the CSV file you want to load. In this case, the file is stored in Google Drive under the specified directory.

**Loading Data**

Code: df = pd.read\_csv(file\_path)

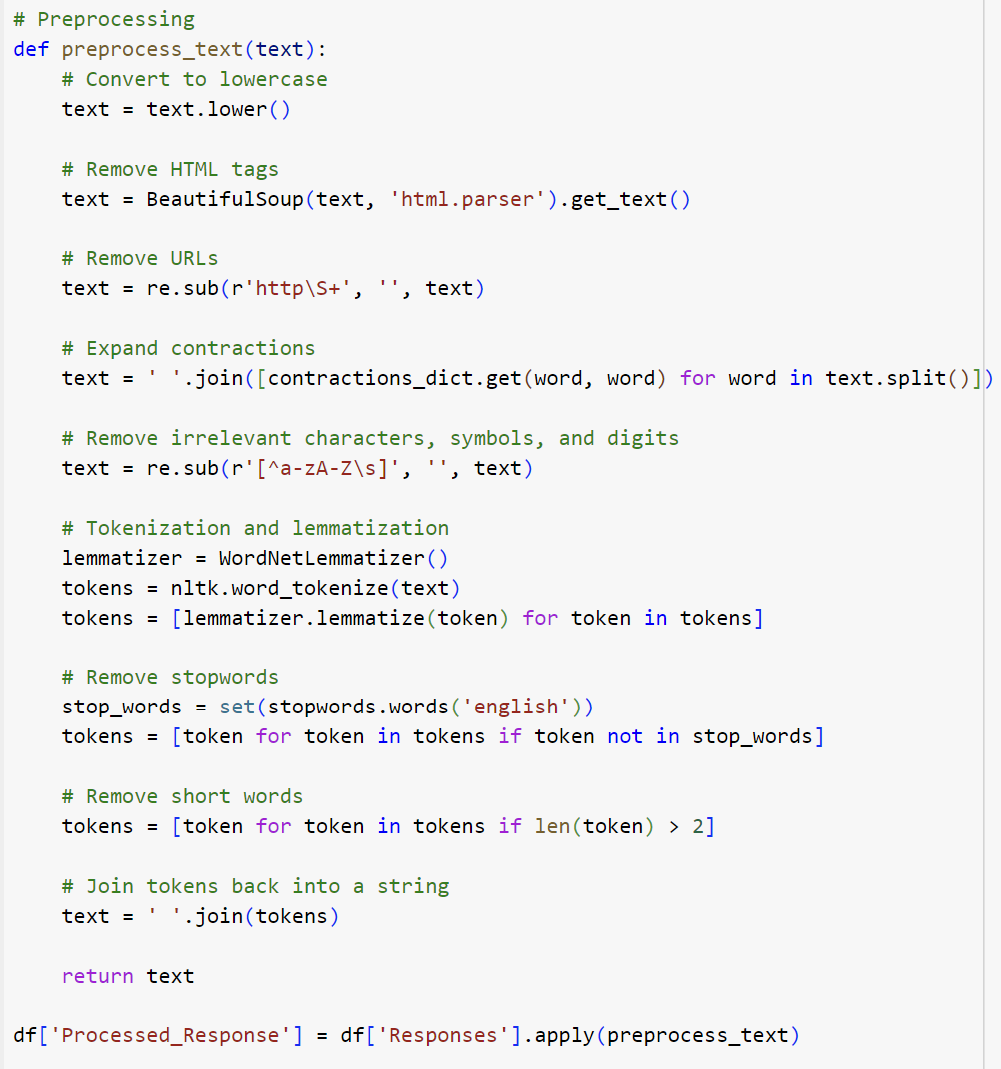
Description: Uses Pandas to read the CSV file at the specified path and load it into a DataFrame named df.

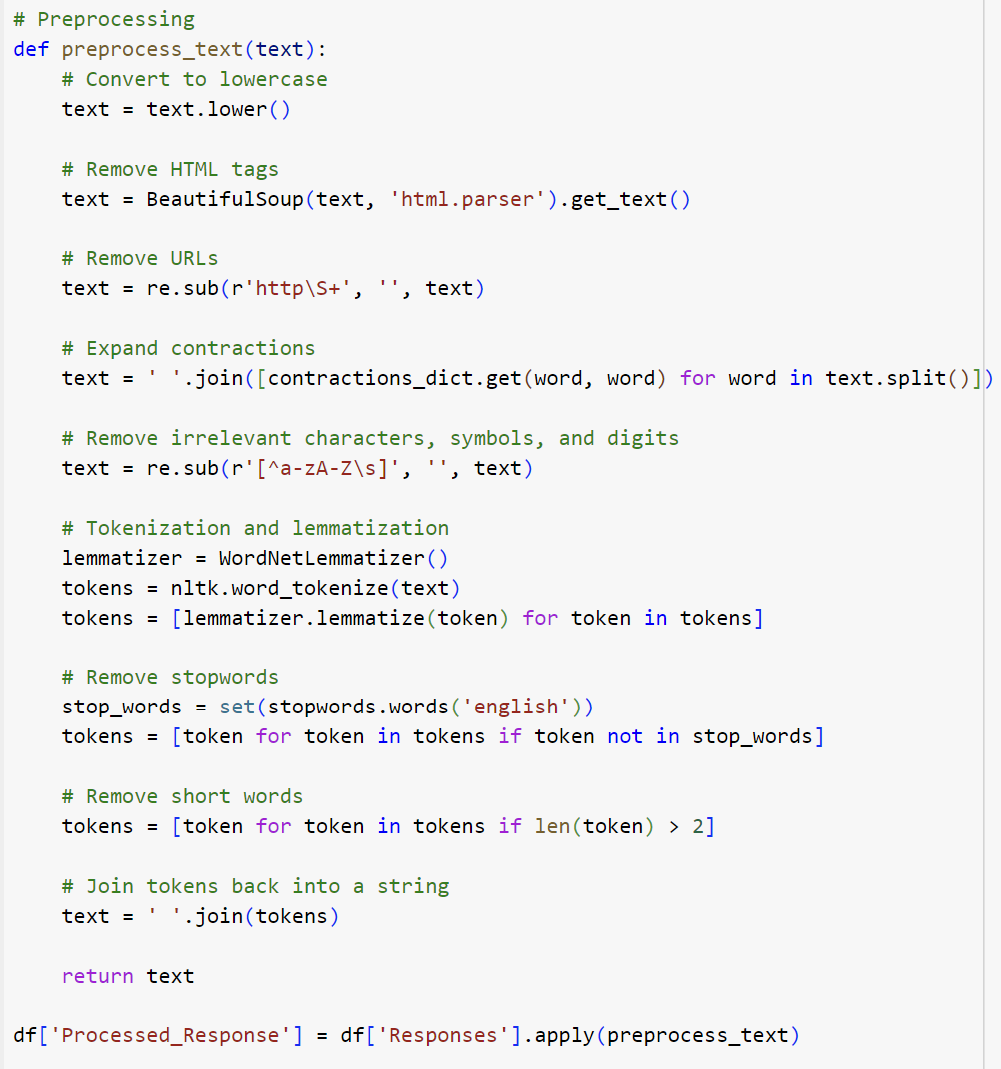
**Google Colab and Google Drive Integration:**

In Google Colab, you can access files stored in your Google Drive by mounting the drive. The file path provided assumes that the drive has already been mounted.

**Pandas DataFrame:**

Pandas is a powerful library for data manipulation and analysis in Python. The pd.read\_csv() function reads a CSV file into a DataFrame, which is a tabular data structure with labeled axes (rows and columns).





**Preprocessing Function**

Code: def preprocess\_text(text):

# Convert to lowercase

text = text.lower()

Description: Converts the entire text to lowercase to ensure uniformity and avoid case-sensitive discrepancies.

Code: text = BeautifulSoup(text, 'html.parser').get\_text()

Description: Uses Beautiful Soup to parse and remove any HTML tags from the text, leaving only the text content.

Code: text = re.sub(r'http\S+', '', text)

Description: Uses a regular expression to find and remove URLs from the text to clean it from irrelevant web links.

Code: text = ' '.join([contractions\_dict.get(word, word) for word in text.split()])

Description: Expands contractions (e.g., "don't" to "do not") using a dictionary of contractions (contractions\_dict) to make the text more readable and standardized.

Code: text = re.sub(r'[^a-zA-Z\s]', '', text)

Description: Uses a regular expression to remove all characters that are not letters or spaces, eliminating irrelevant characters, symbols, and digits.

Code: lemmatizer = WordNetLemmatizer()

tokens = nltk.word\_tokenize(text)

tokens = [lemmatizer.lemmatize(token) for token in tokens]

Description:

Tokenization: Splits the text into individual words (tokens) using NLTK's word\_tokenize.

Lemmatization: Reduces each word to its base or root form using NLTK's WordNetLemmatizer.

Code: stop\_words = set(stopwords.words('english'))

tokens = [token for token in tokens if token not in stop\_words]

Description: Removes common stopwords (e.g., "and", "the", "is") from the token list using NLTK's list of English stopwords, which are typically not useful for analysis.

Code: tokens = [token for token in tokens if len(token) > 2]

Description: Removes words that are shorter than three characters, as they are often not meaningful or informative.

Code: text = ' '.join(tokens)

Description: Joins the cleaned and processed tokens back into a single string of text.

Code: return text

Description: Returns the preprocessed text.

**Applying the Preprocessing Function**

Code: df['Processed\_Response'] = f['Responses'].apply(preprocess\_text)

Description: Applies the preprocess\_text function to each entry in the Responses column of the DataFrame df, creating a new column Processed\_Response with the cleaned and preprocessed text.

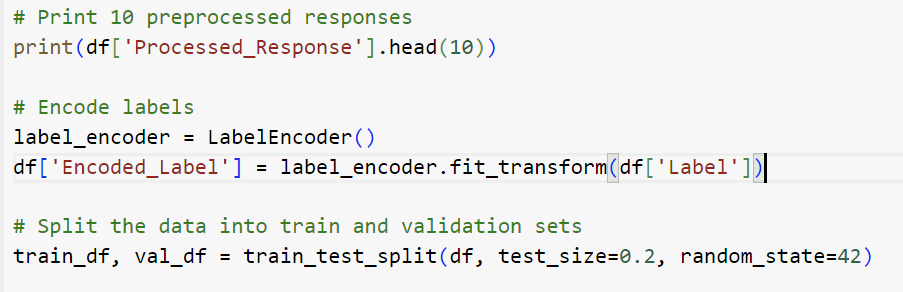
Code: df['Processed\_Response'] = f['Responses'].apply(preprocess\_text)

Description: Applies the preprocess\_text function to each entry in the Responses column of the DataFrame df, creating a new column Processed\_Response with the cleaned and preprocessed text.

The preprocessing function performs several text-cleaning steps:

1. Converts text to lowercase;
2. Removes HTML tags;
3. Removes URLs;
4. Expands contractions;
5. Removes irrelevant characters, symbols, and digits;
6. Tokenizes and lemmatizes text;
7. Removes stopwords;
8. Removes short words;
9. Joins tokens back into a single string.

* After defining the preprocessing function, it is applied to the Responses column in the DataFrame, and the cleaned text is stored in a new column Processed\_Response. This prepares the text data for further analysis or machine learning tasks, such as training a text classification model.



**Print Preprocessed Responses**

Code: print(df['Processed\_Response'].head(10))

Description: Prints the first 10 entries of the Processed\_Response column from the DataFrame df. This is useful for verifying the output of the text preprocessing to ensure that it has been cleaned and processed correctly.

**Encode Labels**

Code: label\_encoder = LabelEncoder()

df['Encoded\_Label'] = label\_encoder.fit\_transform(df['Label'])

Description:

* Label Encoder Initialization: Instantiates a LabelEncoder object from scikit-learn, which is used to convert categorical labels into a format that can be used in machine learning models (i.e., numerical format).
* Fit and Transform: Applies the fit\_transform method to the Label column of the DataFrame. This method first fits the encoder to the labels, determining the unique classes and assigning each one a unique integer, and then transforms the labels into these integers. The transformed labels are stored in a new column, Encoded\_Label, in the DataFrame.

**Split the Data into Train and Validation Sets**

Code: train\_df, val\_df = train\_test\_split(df, test\_size=0.2, random\_state=42)

Description:

* Train-Test Split Function: Uses the train\_test\_split function from scikit-learn to divide the DataFrame df into training and validation sets.
* Parameters:
  + test\_size=0.2: Specifies that 20% of the data should be set aside as the validation set.
  + random\_state=42: A seed value to ensure the split is reproducible, meaning the same random split will occur each time the code is run with this seed.
* Output: Creates two new DataFrames, train\_df for the training data and val\_df for the validation data. This split allows for the training of models on the training data and evaluation on the validation data to check for overfitting and to validate the model's performance on unseen data.

These steps cover critical aspects of a machine learning workflow:

* Verifying the output of preprocessing to ensure the data is clean and properly formatted.
* Encoding categorical labels into a numerical format necessary for fitting most types of machine learning models.
* Splitting the data into training and validation sets to enable the effective training and evaluation of models, helping to ensure that they generalize well to new, unseen data.

**Tokenization**

This section of the code deals with preparing the text data for training with a BERT model by converting the text into a format that the model can understand, including creating attention masks that are necessary for the model to distinguish between real data and padding.



Code: tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased', do\_lower\_case=True)

Description:

* BertTokenizer Initialization: Initializes the BertTokenizer from the Hugging Face transformers library. The tokenizer is loaded with a pre-trained BERT model (bert-base-uncased), which is designed to handle lowercased text.
* do\_lower\_case=True: Instructs the tokenizer to automatically convert all tokens to lowercase before tokenization, matching the casing used in the pre-training of the model.

Code: def tokenize\_text(df, max\_length=128):

input\_ids = []

attention\_masks = []

for text in df['Processed\_Response']:

encoded\_dict = tokenizer.encode\_plus(

text,

add\_special\_tokens=True,

max\_length=max\_length,

padding='max\_length',

return\_attention\_mask=True,

return\_tensors='pt',

truncation=True

)

input\_ids.append(encoded\_dict['input\_ids'])

attention\_masks.append(encoded\_dict['attention\_mask'])

input\_ids = torch.cat(input\_ids, dim=0)

attention\_masks = torch.cat(attention\_masks, dim=0)

return input\_ids, attention\_masks

Description:

* Function Definition: tokenize\_text takes a DataFrame and an optional max\_length parameter (default set to 128 tokens) to tokenize the text data.
* Input IDs and Attention Masks:
  + Input IDs: These are sequences of integers representing each token in the text. The BERT model uses these IDs to look up embeddings.
  + Attention Masks: These masks indicate to the model which tokens should be attended to and which should be ignored (e.g., padding tokens).
* Loop through Text Data: For each piece of text in the Processed\_Response column of the DataFrame:
  + Tokenization and Encoding: Uses tokenizer.encode\_plus() to perform the following:
    - add\_special\_tokens=True: Adds special tokens (e.g., [CLS], [SEP]) required by BERT.
    - max\_length: Ensures all sequences are padded or truncated to this length.
    - padding='max\_length': Pads shorter sequences to the max\_length.
    - return\_attention\_mask=True: Generates an attention mask for the sequence.
    - return\_tensors='pt': Returns PyTorch tensors.
    - truncation=True: Truncates longer sequences to the max\_length.
* Concatenate Tensors: After processing all texts, concatenates the list of tensors into a single tensor for input\_ids and attention\_masks.

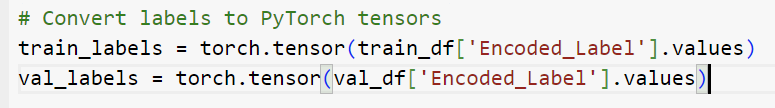
Code: train\_inputs, train\_masks = tokenize\_text(train\_df)

val\_inputs, val\_masks = tokenize\_text(val\_df)

Description: Tokenize Training and Validation Data: Applies the tokenize\_text function to the training and validation DataFrames, resulting in tensors of input IDs and attention masks for both sets. These tensors are then used for training and validating the BERT model.

* This code prepares the text data from the training and validation sets by tokenizing it according to the requirements of the BERT model, ensuring it is in the correct format for model ingestion, including the use of attention masks which are crucial for BERT's self-attention mechanism. This preprocessing is essential for leveraging BERT's capabilities in natural language understanding tasks.

**Convert Labels to PyTorch Tensors**

****

Code: train\_labels = torch.tensor(train\_df['Encoded\_Label'].values)

val\_labels = torch.tensor(val\_df['Encoded\_Label'].values)

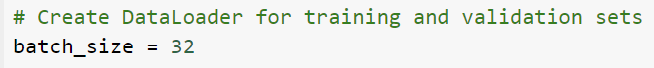
Description:

* Purpose: These lines convert the label arrays from the Pandas DataFrame columns into PyTorch tensors. PyTorch models require data in the form of tensors to perform computations during training and evaluation.
* Process:
  + train\_df['Encoded\_Label'].values: This extracts the values from the 'Encoded\_Label' column of the training DataFrame (train\_df) as a NumPy array.
  + torch.tensor(...): Converts the NumPy array into a PyTorch tensor.
  + This process is repeated for both the training and validation datasets, creating tensors named train\_labels and val\_labels, respectively.

Significance:

* Integration with PyTorch: By converting the labels into tensors, they can be easily integrated into the data loading and training pipeline of a PyTorch model. Tensors are the fundamental data structure in PyTorch, supporting GPU acceleration which is essential for training deep learning models efficiently.
* Data Type Consistency: Ensures that all data types used during model training, including input data (like token IDs and attention masks) and labels, are consistent and compatible with the operations defined in the model training loop.

**Batch Size Initialization**

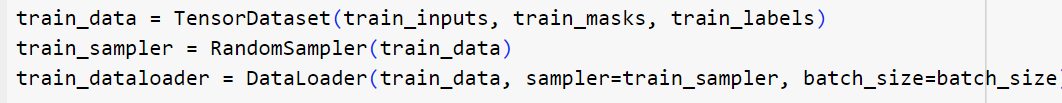


Code: batch\_size = 32

Description: Specifies the number of samples to be included in each batch during the training and validation process. Setting batch\_size to 32 means that each batch of data processed through the model, either during training or validation, will contain 32 examples.

The batch\_size is used when setting up DataLoader instances for both training and validation data sets in PyTorch. These DataLoader instances will then be used to iteratively provide batches of data to the model during the training loop.

**Create TensorDataset for Training Data**

****

Code: train\_sampler = RandomSampler(train\_data)

Description:

* Purpose: This line of code creates a TensorDataset object, which packages the tensors (train\_inputs, train\_masks, and train\_labels) into a single dataset. TensorDataset is a utility that provides a way to encapsulate tensors into a dataset upon which a DataLoader can operate.
* Components:
  + train\_inputs: Tensors containing the tokenized inputs.
  + train\_masks: Tensors containing the attention masks associated with the inputs.
  + train\_labels: Tensors containing the encoded labels for each input.

**Create Random Sampler for Training Data**

Code: train\_sampler = RandomSampler(train\_data)

Description:

* Purpose: Initializes a RandomSampler that will be used to shuffle the data every time data is loaded into the model during training. This shuffling is crucial for training deep learning models as it helps to reduce variance and ensure that the model does not learn the order of the data.
* Parameter:
  + train\_data: The dataset from which the sampler randomly picks elements.

**Create DataLoader for Training Data**

Code: train\_dataloader = DataLoader(train\_data, sampler=train\_sampler, batch\_size=batch\_size)

Description: Creates a DataLoader object that automatically batches the data from train\_data using the RandomSampler. The batch\_size parameter specifies how many samples per batch to load. This DataLoader facilitates the efficient loading of data batches during the training process, allowing for parallel processing and thus speeding up the training.

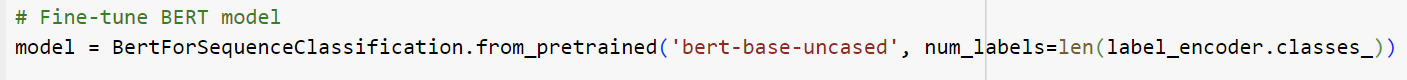
**Importance in the Training Process**

The DataLoader is critical in the training workflow as it:

* Manages Memory: Efficiently handles large datasets by loading small batches of data into memory, preventing memory overload on the GPU.
* Supports Multithreading: PyTorch DataLoader can load data in parallel using multiple workers (num\_workers argument), which significantly speeds up data preprocessing and the overall training cycle.
* Improves Training Dynamics: Using a RandomSampler shuffles the data, ensuring that the model does not learn anything from the sequence of the data, further helping to improve model generalization.

This setup is essential for training machine learning models efficiently, especially when dealing with large datasets that require sophisticated models like BERT for processing. It ensures that data is fed to the model in an optimized way, utilizing the computational resources effectively.

**Fine-tuning BERT Model**

****

Code: model = BertForSequenceClassification.from\_pretrained('bert-base-uncased', num\_labels=len(label\_encoder.classes\_))

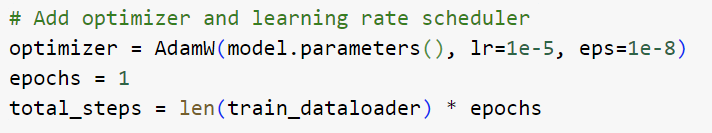
Description:

* BertForSequenceClassification:
  + Module: This is a class from the transformers library developed by Hugging Face. It provides a BERT model pre-configured for sequence classification tasks. This means it has a BERT model base and a top layer with a fully connected layer for classification.
* from\_pretrained:
  + Function: This function loads a pre-trained BERT model. The model weights are automatically downloaded if they are not locally available. It initializes the model with weights trained on a large corpus of text (like Wikipedia and BookCorpus).
  + 'bert-base-uncased': Specifies the particular variant of BERT to use. "Base" indicates a smaller version compared to the "Large" model, and "uncased" means it does not distinguish between upper and lower case, ideal for general text processing.
* num\_labels:
  + Parameter: This argument specifies the number of labels that the classification layer needs to predict. It dynamically adjusts the architecture of the output layer of the BERT model to match the number of labels in your dataset, allowing for multi-class classification.
  + len(label\_encoder.classes\_): Computes the number of unique labels in your dataset by examining the number of classes the label encoder has encoded. This ensures that the output layer of the model correctly corresponds to the number of classes you are predicting.

**Importance in the Training Process**

* Customization for Task: By specifying num\_labels, you ensure that the model's output layer is customized to predict the exact number of unique classes in your dataset, which is critical for effectively training a classifier.
* Adaptation of Pre-trained Model: Utilizing a pre-trained model like BERT allows you to leverage transfer learning, where the model has already learned a significant amount of relevant information from a large, diverse text corpus. This significantly reduces the time and data required to achieve high performance.
* Flexibility and Efficiency: The from\_pretrained method is efficient as it loads optimized model weights and only requires fine-tuning on a smaller dataset specific to the task at hand, which can often result in better performance compared to training a model from scratch.

**Optimizer Setup**

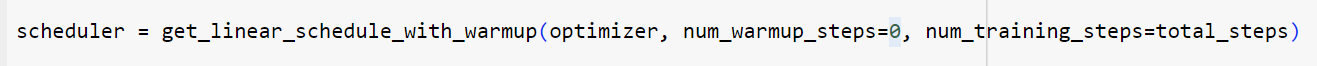
****

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

Description:

* AdamW:
  + Optimizer: A variant of the Adam optimizer that differs mainly in how it handles weight decay. It is designed specifically for training deep learning models.
  + model.parameters(): Specifies that the optimizer should update all the parameters of the BERT model.
* lr=1e-5: Sets the learning rate to 10–5. The learning rate controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing an appropriate learning rate is crucial for good training performance.
* eps=1e-8: Adds a very small number to the denominator in the optimizer's update step to prevent any division by zero. This helps in maintaining numerical stability.

**Learning Rate Scheduler Setup**

****

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

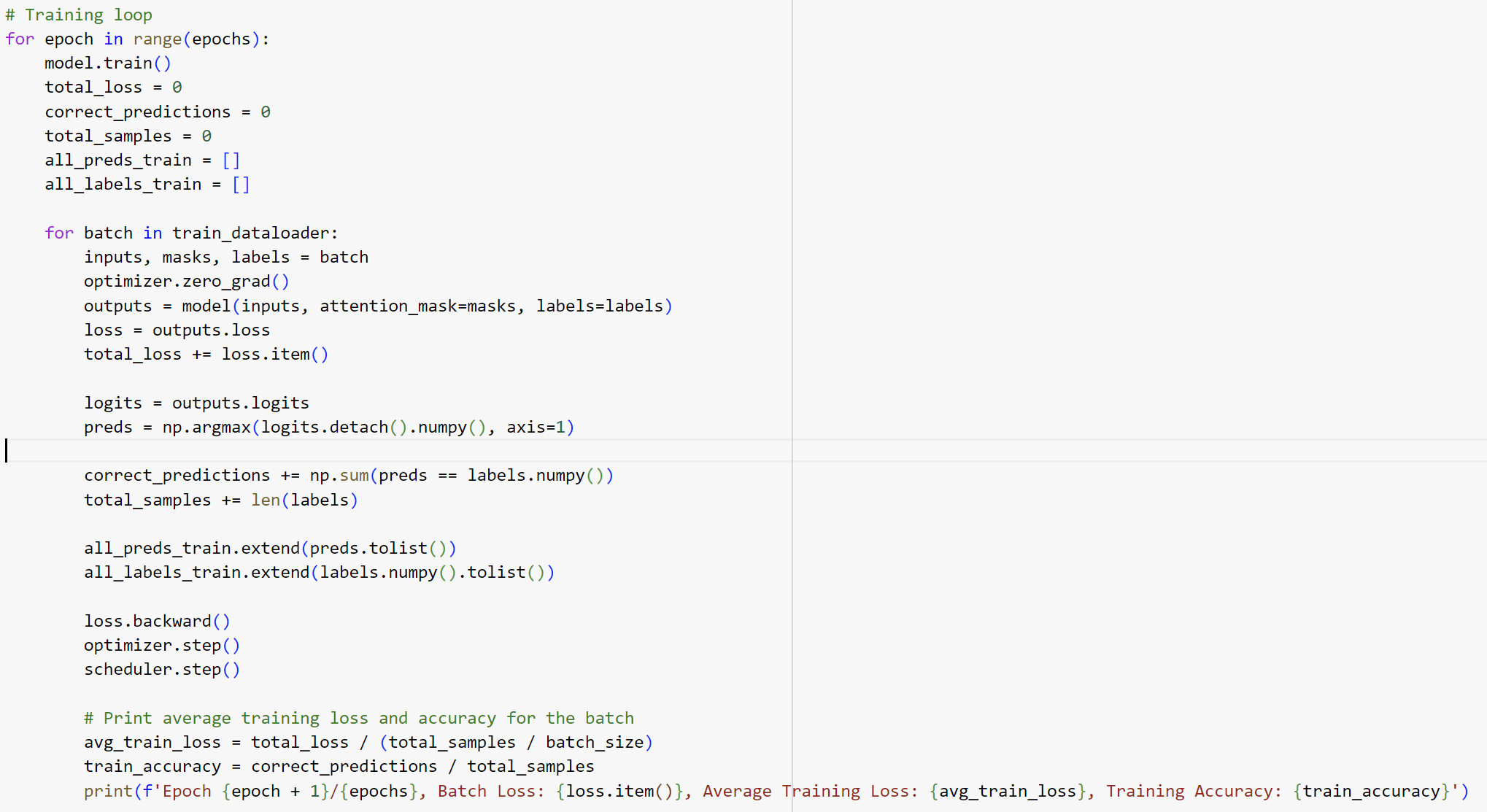
Description:

* get\_linear\_schedule\_with\_warmup:
  + Function: This function from the transformers library sets up a learning rate scheduler that initially increases the learning rate linearly for the given number of "warmup" steps, and then decreases it linearly after the warmup period until the end of training.
  + Parameters:
    - optimizer: Specifies the optimizer for which to schedule the learning rate, previously defined as AdamW(model.parameters(), lr=1e-5, eps=1e-8).
    - num\_warmup\_steps=0: Indicates that there are no warmup steps where the learning rate gradually increases from zero to the initial set learning rate (lr=1e-5). The learning rate starts from the initial set value and begins to decrease after reaching the peak immediately.
    - num\_training\_steps=total\_steps: The total number of training steps that the scheduler should consider. It defines how long the learning rate will be adapted, calculated as the product of the number of epochs and the length of train\_dataloader.

**Importance in the Training Process**

* Adaptive Learning Rate: By adjusting the learning rate throughout training, the scheduler can help the model train more effectively. A learning rate that is too high can cause the model to converge too quickly to a suboptimal solution, and a rate that is too low might slow down the training process or cause it to stall.
* Warmup Phase: Although set to zero in your configuration, a warmup phase can be crucial for stabilizing the model's parameters at the beginning of training. Gradually ramping up the learning rate can help prevent the model from diverging early in training due to high gradient updates.

**Training Loop Breakdown**

****

Description:

* Epoch Loop: Iterates over the specified number of epochs. An epoch represents a complete pass through the entire training dataset.
* Model State: Sets the model to training mode using model.train(), which is essential for layers like dropout and batch normalization to behave accordingly.
* Initialization:
  + total\_loss: Accumulates the loss for each batch within the epoch to compute the average loss later.
  + correct\_predictions: Counts the number of correctly predicted labels to calculate accuracy.
  + total\_samples: Tracks the total number of samples processed to help in calculating the average loss and accuracy.
  + all\_preds\_train, all\_labels\_train: Lists to store all predictions and labels from the training for further analysis or detailed accuracy assessment.
* Batch Processing:
  + Zero Gradient: Clears old gradients; otherwise, they would accumulate.
  + Model Forward Pass: Passes inputs and masks to the model, specifying labels to calculate the loss automatically.
  + Loss Calculation: Retrieves the loss from the model output.
  + Prediction and Accuracy Tracking:
    - Converts logits to probabilities and determines the class with the highest probability as the prediction.
    - Compares predictions to true labels to count correct predictions.
* Backpropagation and Optimization:
  + loss.backward(): Computes the gradient of the loss with respect to the model parameters.
  + optimizer.step(): Updates the model parameters based on the gradient.
  + scheduler.step(): Updates the learning rate according to the learning rate schedule.
* Logging:
  + Computes and prints the average training loss and accuracy after processing each batch, providing insights into the model's performance as training progresses.

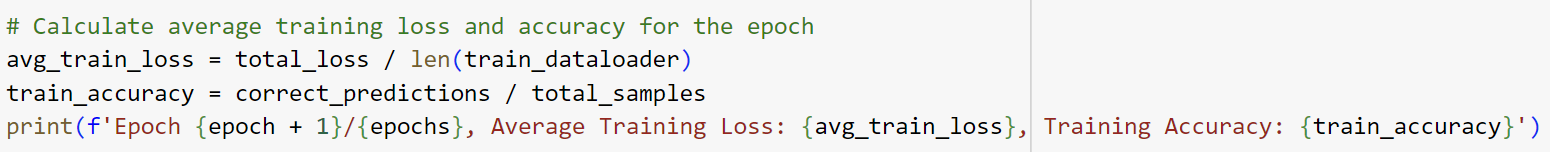
Importance and Impact

This training loop is crucial for fine-tuning the BERT model effectively. It integrates various components necessary for training deep learning models, such as dynamic adjustments to the learning rate, accurate tracking of performance metrics, and proper gradient management. Such a setup helps in:

* Optimizing Model Weights efficiently to minimize loss and improve accuracy.
* Monitoring Training Progress in real-time, allowing for adjustments based on immediate feedback from the loss and accuracy figures.
* Ensuring Model Generalizability by correctly implementing regularization strategies during the training phase.

By managing these elements effectively, the training loop ensures that the model learns to generalize well from the training data, avoiding issues like overfitting and underfitting, thus preparing it for robust performance on unseen data.

**Calculation of Average Training Loss and Accuracy**

****

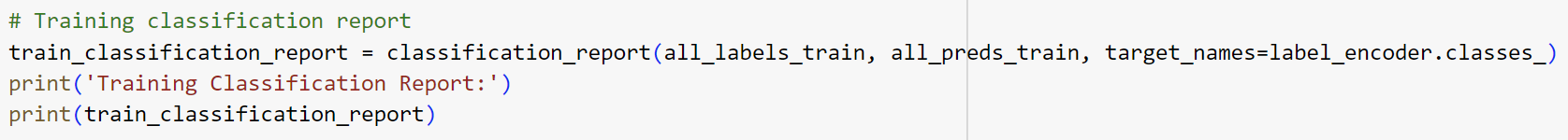
Description:

* Average Training Loss:
  + Calculation: avg\_train\_loss = total\_loss / len(train\_dataloader)
  + Explanation: This line computes the average training loss per batch by dividing the total accumulated loss (total\_loss) by the number of batches in the training DataLoader (len(train\_dataloader)). The total loss is summed over all batches processed during the epoch.
* Training Accuracy:
  + Calculation: train\_accuracy = correct\_predictions / total\_samples
  + Explanation: This line calculates the accuracy of the model for the epoch by dividing the total number of correct predictions (correct\_predictions) by the total number of samples processed (total\_samples). It reflects how well the model is predicting the correct labels across all batches.
* Logging:
  + Print Statement: Outputs the results for the epoch, including the average training loss and training accuracy. This feedback is crucial for monitoring the training process, understanding the model's performance, and diagnosing issues like overfitting or underfitting.

Importance in the Training Process

* Performance Monitoring: The end-of-epoch metrics provide a snapshot of the model’s performance after it has seen all training data once. These metrics are key indicators of the model's learning progress and are essential for validating the effectiveness of training interventions like adjustments in learning rate or model architecture.
* Diagnostic Tool: Regular feedback on loss and accuracy helps in diagnosing training issues. For example, if the loss is not decreasing or the accuracy is not improving as expected, it might indicate problems such as inadequate model capacity, poor optimization settings, or data quality issues.
* Model Tuning and Validation: By observing these metrics, you can make informed decisions about when to stop training, whether to adjust hyperparameters, or if it is necessary to implement techniques like early stopping to prevent overfitting.

**Training Classification Report Generation**



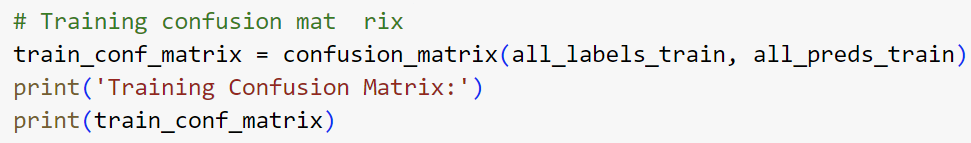
Description:

* Classification Report:
  + Function: classification\_report from scikit-learn.
  + Inputs:
    - all\_labels\_train: A list containing all the true labels collected over the epoch.
    - all\_preds\_train: A list containing all the predictions made by the model over the same epoch.
    - target\_names=label\_encoder.classes\_: An array of class names that corresponds to the labels, which improves the readability of the report by replacing numerical labels with actual class names.
  + Output: The function returns a text report showing the main classification metrics (precision, recall, and F1-score) for each class on a per-class basis, along with averages.
* Printing the Report:
  + The first print statement introduces the classification report, making the output more readable.
  + The second print statement outputs the detailed classification metrics for each class and averages.

Importance and Utility

* Detailed Performance Insight: The classification report provides a detailed look at the performance of the model for each class, which is crucial for multi-class classification tasks. Metrics like precision, recall, and F1-score are critical for understanding how well the model is performing, especially in scenarios where class imbalance might affect the overall accuracy.
* Model Tuning and Improvement:
  + Identify Weaknesses: By analyzing the classification report, you can identify which classes are well-predicted and which are not, guiding further model training and class-specific interventions.
  + Balance Training: If certain classes have significantly lower performance metrics, you might consider techniques like resampling, cost-sensitive learning, or gathering more data for those classes.
* Feedback Loop: Regularly generating these reports during training provides ongoing feedback, allowing for continuous monitoring and adjustment of the training process.

Training Confusion Matrix Generation



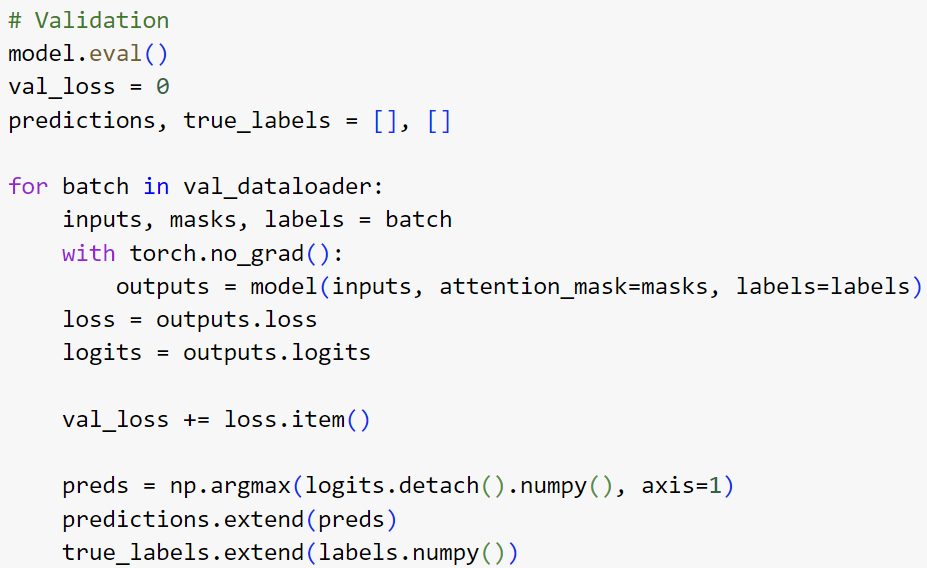
Description:

* Confusion Matrix:
  + Function: confusion\_matrix from scikit-learn.
  + Inputs:
    - all\_labels\_train: This is a list or array containing all the true label values collected during the epoch.
    - all\_preds\_train: This corresponds to all the predictions made by the model over the same epoch.
  + Output: The function returns a confusion matrix in the form of a 2D array where each row represents the instances in an actual class while each column represents the instances in a predicted class.
* Printing the Confusion Matrix:
  + The first print statement introduces the confusion matrix, making it clear what the following output represents.
  + The second print statement outputs the numerical matrix. Each cell in the matrix at position [i, j] indicates the number of samples from the true class i that were predicted as class j by the model.

Importance and Utility

* Diagnostic Tool: The confusion matrix is a powerful diagnostic tool to measure the accuracy of a classification model. It provides a visual and numerical way to identify where the model is confusing two classes, which is particularly useful for multi-class classification problems.
* Model Evaluation and Tuning:
  + Identify Misclassifications: It helps in identifying not just the errors quantitatively but also shows the types of errors the model is making, such as which specific classes are being confused.
  + Guide Model Improvements: Understanding these patterns can guide further model refinement, such as adding more training examples for frequently confused classes, applying class-weight adjustments, or tweaking the model architecture.
* Feedback Loop: Regular analysis of the confusion matrix during training provides insights into how changes to the model (like hyperparameters adjustments or training on additional data) are affecting overall performance and class-specific performance.

**Validation Process**



* Set Model to Evaluation Mode:

Code: model.eval()

Description: Switches the model to evaluation mode. This affects layers like dropout and batch normalization which behave differently during training vs. evaluation (e.g., dropout is disabled).

* Initialize Validation Loss:

Code: val\_loss = 0

Description: Initializes a variable to accumulate the total validation loss over all batches, which will be used to calculate the average loss later.

* Loop Over Validation Batches:
  + Iteration Over val\_dataloader: Iterates over each batch of data in the validation DataLoader.
  + Data Unpacking: Each batch contains inputs (input IDs), masks (attention masks), and labels.
* Model Inference Without Gradient Calculation:

Code: with torch.no\_grad():

Description: Ensures that the model’s forward pass is done without keeping track of gradients which saves memory and computations, appropriate for inference only.

* + Model Forward Pass: outputs = model(inputs, attention\_mask=masks, labels=labels)
    - Retrieves the model's output including the computed loss and logits (the raw, unnormalized scores output by the last layer of the model).
* Accumulate Validation Loss:

Code: val\_loss += loss.item()

Description: Adds the loss for the current batch to the total validation loss.

* Compute Predictions:

Code: preds = np.argmax(logits.detach().numpy(), axis=1)

Description: Converts logits to numpy arrays and uses np.argmax() to find the indices of the maximum logit value along the axis representing different classes. This translates logits to class predictions.

* Store Predictions and Labels:

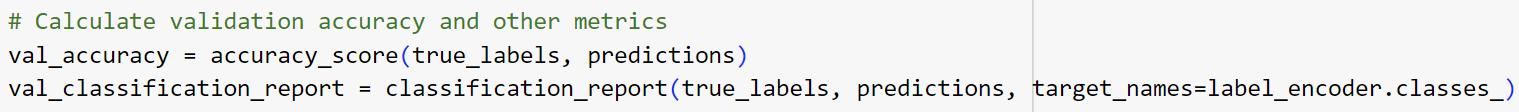
Code: predictions.extend(preds) and true\_labels.extend (labels.numpy())

Description: Appends the predictions and actual labels of the batch to the lists predictions and true\_labels, respectively. These lists accumulate predictions and labels from all batches in the validation dataset.

Importance and Utility

* Model Performance Evaluation: The validation phase is critical for assessing the model's performance on data it hasn't seen during training. This helps in identifying issues like overfitting.
* Metrics Calculation: After this loop, you can calculate metrics such as accuracy, precision, recall, F1-score, and others using predictions and true\_labels to understand the model's effectiveness across different classes.
* Loss Monitoring: Keeping track of the validation loss helps in comparing it with the training loss to check for overfitting.

**Validation Process**



* Calculate Validation Accuracy

Code: val\_accuracy = accuracy\_score(true\_labels, predictions)

Description:

* + Function: accuracy\_score from scikit-learn.
  + Inputs:
    - true\_labels: An array or list containing the true labels from the validation dataset.
    - predictions: An array or list containing the predicted labels produced by the model for the validation dataset.
  + Output: The function returns the accuracy score, which is the proportion of correct predictions (i.e., the number of correct predictions divided by the total number of predictions). This metric provides a quick and intuitive measure of how well the model is performing overall.
* Generate Validation Classification Report

Code: val\_classification\_report = classification\_report true\_labels, predictions, target\_names=label\_encoder.classes\_)

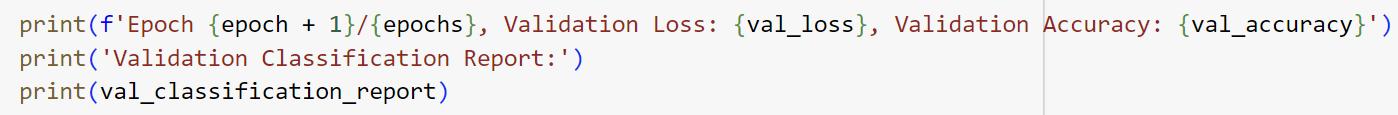
Description:

* + Function: classification\_report from scikit-learn.
  + Inputs:
    - true\_labels: The true labels against which the model's predictions are evaluated.
    - predictions: The labels predicted by the model.
    - target\_names=label\_encoder.classes\_: Provides a list of class names for which the report will generate metrics. These names replace numerical labels in the output, making the report easier to understand.
  + Output: A text summary of the main classification metrics per class, including precision, recall, and F1-score, along with overall averages. This report is valuable for seeing how the model performs across different classes, which helps in identifying any biases or weaknesses in how the model handles certain categories.

Importance and Utility

* Accuracy Measurement: Validation accuracy provides a straightforward metric to gauge overall performance. It is especially useful for communicating results in understandable terms and for comparisons with other models or benchmarks.
* Detailed Performance Analysis:
  + The classification report offers a more nuanced view of model performance than accuracy alone. For example, in imbalanced datasets, overall accuracy might be high while performance on minority classes might be poor.
  + Metrics like precision (proportion of positive identifications that were actually correct), recall (proportion of actual positives that were identified correctly), and F1-score (a harmonic mean of precision and recall) are crucial for assessing performance in such cases.
* Model Tuning and Improvement:
  + These metrics can guide further model tuning, such as adjusting class weights, changing the decision threshold, or adding more training data for underperforming classes.
  + Helps in validating the effectiveness of any changes made during the model development process, ensuring that improvements are based on solid evidence from the model's performance on the validation data.

**Output Validation Results**

****

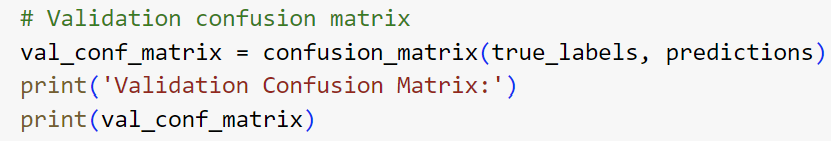
Description:

* First Print Statement:
  + Content: This line formats and prints a message that includes the current epoch number, the total number of epochs, the calculated validation loss, and the validation accuracy.
  + Purpose: Providing a concise summary of key performance metrics for the epoch helps track how the model's performance is progressing across epochs. It’s crucial for monitoring training dynamics and ensuring that the model is improving or maintaining performance over time without overfitting.
* Second and Third Print Statements:
  + Second Print Statement: Introduces the classification report, setting context for the following detailed performance metrics.
  + Third Print Statement: Outputs the classification\_report generated from scikit-learn, which details precision, recall, and F1-scores for each class along with averages.
  + Purpose: This detailed breakdown helps in understanding the model's strengths and weaknesses across different classes. It’s especially useful in applications where performance on specific classes is more critical than overall accuracy.

Importance and Utility

* Immediate Feedback: These print statements provide immediate feedback on the model's performance at the end of each validation run, which is essential for iterative model development and tuning.
* Performance Tracking: By printing these metrics at each epoch, you can track improvements or regressions in the model's performance due to changes in the model or training procedure. This tracking is vital for tuning hyperparameters and making informed decisions about training duration and techniques.
* Detailed Class-Level Insights: The classification report provides insights at the class level, which are crucial for handling class imbalance and focusing on classes that are critical for the specific application or use case. Understanding how the model performs in each class can guide targeted data collection or further tuning of the model to improve minority class performance.

**Generate and Print Validation Confusion Matrix**



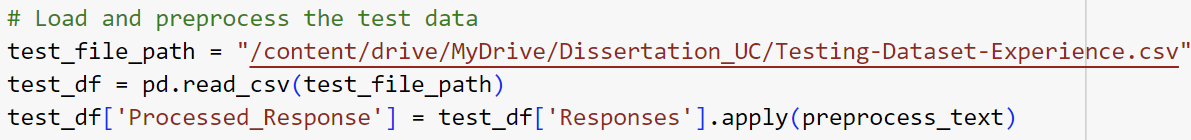
Description:

* Confusion Matrix Calculation:
  + Function: confusion\_matrix from scikit-learn.
  + Inputs:
    - true\_labels: This is an array or list containing all the true label values from the validation dataset.
    - predictions: This corresponds to all the predictions made by the model for the validation dataset.
  + Output: The function returns a confusion matrix in the form of a 2D array, where each row represents the instances in an actual class while each column represents the instances in a predicted class.
* Print the Confusion Matrix:
* The introductory print statement, "Validation Confusion Matrix:", sets the context for the output, making it clear to the viewer what is being displayed next.
* The confusion matrix itself is printed out next. The matrix shows how predictions are distributed across actual labels, providing detailed insight into the type and frequency of classification errors the model is making.

Importance and Utility

* Diagnostic Tool: The confusion matrix is invaluable for diagnosing classification performance because it visually breaks down the errors and successes of the model's predictions. It lets you see not just how many predictions were wrong, but how they were wrong:
  + Type I and Type II Errors: For binary classifications, it shows false positives and false negatives.
  + Misclassifications: For multiclass classifications, it shows which classes are frequently confused.
* Performance Evaluation: The matrix can help identify if the model is biased towards certain classes, or if there are classes that it consistently mislabels, guiding further data collection, model retraining, or adjustment in class weights.
* Improvement and Balancing: Especially useful in cases of class imbalance where traditional metrics like accuracy might be misleading. The confusion matrix provides a more nuanced view that can help in rebalancing or refocusing the model towards underperforming areas.

**Loading and Preprocessing Test Data**



Description:

* Loading Test Data:

Code: test\_df = pd.read\_csv(test\_file\_path)

Description: This line reads the CSV file specified by test\_file\_path into a DataFrame test\_df. The CSV file is assumed to be located in a Google Drive folder, as indicated by the path. This is a common approach when using Google Colab, which allows direct integration with Google Drive for file access.

* File Path Specification:

Code: test\_file\_path = "/content/drive/MyDrive/Dissertation\_UC/

Testing-Dataset-Experience.csv"

Description: Specifies the location of the test dataset on Google Drive. This path needs to be accessible from the environment where the code is running, typically set up after mounting Google Drive in Google Colab.

* Preprocessing Test Data:

Code: test\_df['Processed\_Response'] = test\_df['Responses'].apply

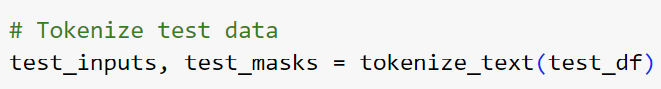
(preprocess\_text)

Description: Applies the preprocess\_text function to each entry in the Responses column of the test DataFrame. This function is assumed to perform various text preprocessing steps such as lowercasing, removing HTML tags, URLs, and non-alphanumeric characters, expanding contractions, and possibly removing stopwords and performing tokenization and lemmatization. The processed text is then stored in a new column Processed\_Response.

Importance and Utility

* Consistency in Data Preparation: It's crucial that the same preprocessing steps applied to the training and validation datasets are also applied to the test dataset to ensure consistency in how data is fed into the model. This prevents issues related to data mismatch which could adversely affect the model's performance.
* Evaluation Readiness: Preprocessing the test data prepares it for feeding into the model for final evaluations. This step ensures that the data is in the correct format and condition for the model to process effectively.
* Automated Pipeline: By automating the loading and preprocessing steps, you reduce the risk of manual errors and increase the efficiency of the testing process, making it easier to replicate and scale as needed.

**Tokenization of Test Data**

****

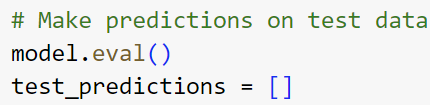
Description:

* Function Call: tokenize\_text(test\_df)
  + Function: The tokenize\_text function is assumed to be designed to handle the tokenization process tailored for a BERT model, converting raw text into a format that includes input IDs and attention masks.
  + Input: test\_df is passed to the function, which should contain at least one column (Processed\_Response) with preprocessed text ready for tokenization.
* Processing:
  + Text Conversion: Each text entry is processed to convert into a sequence of tokens. This typically involves breaking down the text into words or subwords, which are then mapped to token IDs defined in BERT's vocabulary.
  + Attention Masks: Alongside token IDs, attention masks are generated to indicate to the model which parts of the input are actual data and which are padding. This is crucial because BERT and similar models are sensitive to sequence length and require uniform input sizes.
* Output:
* test\_inputs: Contains the token IDs for the test data. These are numeric representations of the text where each integer corresponds to a token in the BERT vocabulary.
* test\_masks: Comprises the attention masks for the test data, which help the model differentiate between meaningful tokens and padding.

Importance and Utility

* Preparation for Model Inference: Tokenization is a crucial step to format the text data into a structure that can be processed by BERT. Since BERT expects specific input formats (token IDs and attention masks), this step ensures that the test data adheres to these requirements.
* Ensures Model Compatibility: Proper tokenization and attention mask generation are essential for maintaining compatibility with the model's trained parameters. Incorrect tokenization or mask generation could lead to suboptimal or incorrect model predictions.
* Efficiency in Processing: Properly formatted inputs (padded and masked according to the model's requirements) optimize the computational efficiency during the model inference stage, ensuring that resources are used effectively, especially when processing large amounts of test data.

**Making Predictions**

****

Model Evaluation Mode

Code: model.eval()

Description:

* + Purpose: Sets the BERT model to evaluation mode. This is crucial as it tells the model to disable certain layers and behaviors that are only appropriate during training, such as dropout layers or batch normalization, which behave differently during training vs. evaluation.
  + Effect: Ensures that the model's behavior is consistent and predictable when making predictions, which is essential for obtaining reliable and reproducible results during inference.
* Initialize List for Predictions

Code: test\_predictions = []

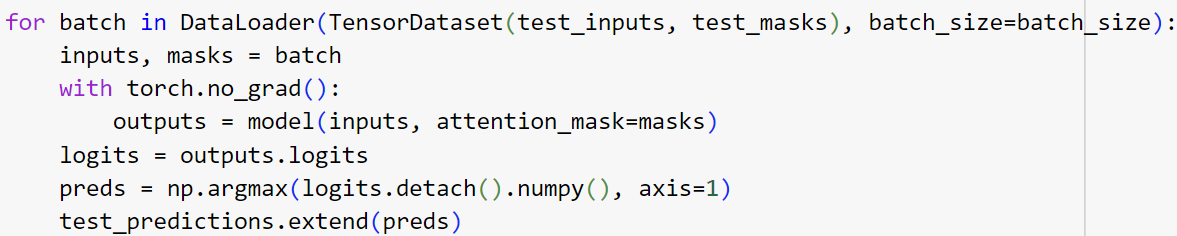
Description:

* + Purpose: Initializes an empty list named test\_predictions to store the outputs from the model when processing the test dataset. This list will accumulate the predicted labels or scores for each batch of data processed by the model.
  + Usage: As you process each batch of test data, you'll append the model's predictions to this list. After all data has been processed, test\_predictions will contain the complete set of predictions which can then be used for further analysis or metrics calculation.

Significance and Further Steps

* Consistency and Reliability: Setting the model to evaluation mode ensures that the model behaves consistently, providing reliable predictions that are not influenced by training-specific operations like dropout.
* Accumulation of Results: Initializing an empty list for storing predictions is a standard practice in machine learning workflows. It allows for efficient collection of results batch-by-batch, which is particularly useful when dealing with large datasets that must be processed in multiple smaller batches due to memory constraints.

**Prediction Loop for Test Data**

****

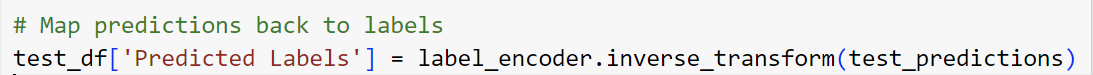
Description:

* DataLoader Setup:
  + Purpose: Wraps the test inputs and masks into a TensorDataset and creates a DataLoader for it. This allows you to process the test data in manageable batches specified by batch\_size.
  + Usage: Efficient handling of data during inference, especially useful when the dataset is too large to fit into memory all at once.
* Batch Processing:
  + Variables inputs, masks: Extracts the input IDs and attention masks from each batch.
  + Context Manager torch.no\_grad(): Ensures that no gradients are computed during this forward pass, which reduces memory consumption and speeds up computation.
* Model Inference:
  + Forward Pass: outputs = model(inputs, attention\_mask=masks) sends the inputs and masks through the model, calculating the raw logits (outputs from the final layer before activation function).
  + Logits Handling: logits = outputs.logits extracts the logits from the model's output.
* Prediction Calculation:
  + Convert Logits to Predictions: preds = np.argmax(logits.detach().numpy(), axis=1) applies the argmax function to determine the class with the highest score for each example in the batch.
    - detach() is used to remove logits from the computation graph (to ensure they don't take up memory for gradients),
    - numpy() converts the tensor to a NumPy array for easy manipulation with NumPy functions.
  + Extend Prediction List: test\_predictions.extend(preds) appends the predictions from the current batch to the test\_predictions list, accumulating the predictions across all batches.

Importance and Utility

* Efficient Memory Usage: Using torch.no\_grad() ensures that the memory normally used for gradients during training is not used, making inference more memory-efficient.
* Batch Processing: By processing the data in batches, you manage GPU memory more effectively and ensure that the model can handle large datasets.
* Accurate Model Evaluation: The method used to compute predictions (argmax) directly translates model output scores to class labels, allowing for a straightforward evaluation of model performance against true labels (if available).

**Mapping Predictions Back to Labels**



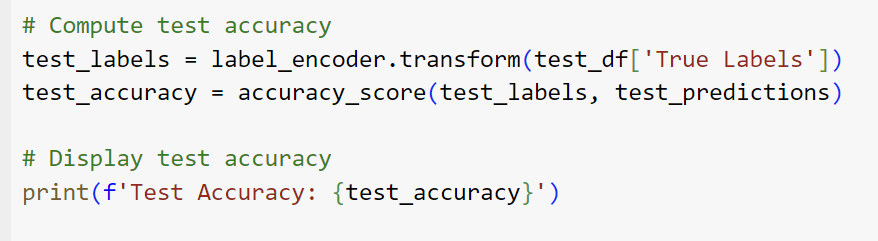
Description:

* Function Used: label\_encoder.inverse\_transform()
  + Purpose: Converts the numerical predictions (which were transformed to numeric form for model processing) back into their original, categorical label form using the LabelEncoder that was fitted on the training data.
  + Input: test\_predictions, which is a list of numerical indices representing the predicted class labels by the model.
  + Output: An array of original categorical labels corresponding to these indices.
* Assignment to DataFrame:
  + DataFrame Column: test\_df['Predicted Labels']
  + Operation: Assign the array of original categorical labels back into the test DataFrame under Predicted Labels. This allows for easy viewing and comparison of the predicted labels alongside any existing data in test\_df.

Importance and Utility

* Interpretability: Converting numerical predictions back to their original labels enhances the interpretability of the results. Users can understand what each prediction means in the context of the problem domain.
* Integration: Storing these labels in the test DataFrame integrates the predictions with the rest of the test data, facilitating further analysis, such as performance evaluation and error analysis, and preparing reports or presentations communicating the model's outcomes.
* Validation and Reporting: With the predicted labels readily available in a human-readable form, it becomes straightforward to validate the model's performance against actual outcomes where available or to compile detailed reports on the model's predictions for stakeholders.

**Compute and Display Test Accuracy**

****

* Compute Test Accuracy

Code: test\_labels = label\_encoder.transform(test\_df['True

Labels'])

test\_accuracy = accuracy\_score(test\_labels, test\_predictions)

Description:

* Transform True Labels:
  + Function Used: label\_encoder.transform()
  + Purpose: Converts the categorical labels from the True Labels column in test\_df into numerical form using the same LabelEncoder that was used for the original label encoding. This ensures that the true labels and the predicted labels are in the same format, allowing for direct comparison.
  + Input: test\_df['True Labels'] - a column from the DataFrame containing the true categorical labels for the test data.
  + Output: test\_labels - an array of numerical indices representing the true class labels.
* Calculate Accuracy:
  + Function Used: accuracy\_score() from sklearn.metrics
  + Purpose: Computes the accuracy of the predictions made by the model by comparing the predicted labels (test\_predictions) with the true labels (test\_labels).
  + Inputs:
    - test\_labels: The true labels transformed into numerical indices.
    - test\_predictions: The list of predictions made by the model, also in numerical form.
* Output: test\_accuracy - a float representing the proportion of correct predictions out of the total number of predictions made.
* Display Test Accuracy

Code: print(f'Test Accuracy: {test\_accuracy}')

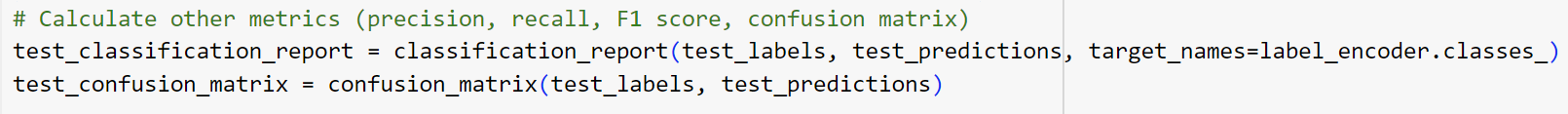
Description:

* + Print Statement: Outputs the calculated accuracy to the console or standard output.
  + Purpose: Provides a quick, readable summary of how well the model performed on the test dataset, indicating the percentage of test instances that were correctly classified.

Importance and Utility

* Model Evaluation: Accuracy is a straightforward and commonly used metric for evaluating classification models. It provides an immediate sense of how often the model is correct across all classes.
* Model Validation: High accuracy on the test set, especially if it is close to the training and validation accuracies, indicates that the model generalizes well to new, unseen data.
* Decision Making: The reported test accuracy can be crucial for deciding whether the model is ready for deployment or if further improvements are necessary.

**Calculate Precision, Recall, and F1 Score**

****

Code: test\_classification\_report = classification\_report

(test\_labels, test\_predictions, target\_names=

label\_encoder.classes\_)

Description:

* + Function Used: classification\_report from sklearn.metrics
    - Purpose: Generates a report that shows the precision, recall, and F1 score for each class, as well as macro-average, weighted average, and (optionally) micro-average across classes.
    - Inputs:
      * test\_labels: The true labels for the test data, transformed into numerical indices that represent each class.
      * test\_predictions: The predicted labels for the test data, provided by the model.
      * target\_names=label\_encoder.classes\_: A list of class names that correspond to the indices in the labels, enhancing the readability of the report by replacing numerical class indices with descriptive class names.
  + Output: test\_classification\_report - a string that tabulates the precision, recall, and F1 score for each class, along with averages across classes, providing a comprehensive overview of the model's performance.
* Generate Confusion Matrix

Code: test\_confusion\_matrix = confusion\_matrix(test\_labels,

test\_predictions)

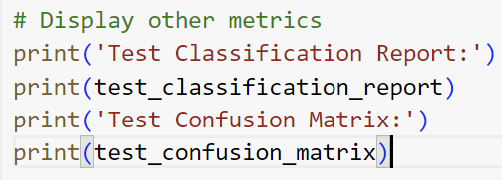
Description:

* + Function Used: confusion\_matrix from sklearn.metrics
    - Purpose: Creates a confusion matrix that illustrates the performance of the model by showing the actual versus predicted classifications.
    - Inputs:
      * test\_labels: True labels for the test data.
      * test\_predictions: Predictions made by the model.
  + Output: test\_confusion\_matrix - a 2D array where each row represents the instances in an actual class and each column represents the instances in a predicted class. The diagonal elements represent correct classifications, while off-diagonal elements are misclassifications.

Importance and Utility

* Detailed Performance Insight: Precision, recall, and F1 score are critical for assessing the quality of predictions, particularly in imbalanced datasets where accuracy might not reflect the true performance.
  + Precision: The ratio of correctly predicted positive observations to the total predicted positives. High precision relates to a low false positive rate.
  + Recall (Sensitivity): The ratio of correctly predicted positive observations to all observations in actual class - it shows the ability of the model to find all the positive samples.
  + F1 Score: The weighted average of Precision and Recall. This score takes both false positives and false negatives into account. It is particularly useful when the classes are imbalanced.
* Confusion Matrix: Offers a visualization of the model's performance with regard to each class, helping identify classes that are hard to predict and those that are easily confused with others, guiding potential improvements in model training or data preprocessing.

**Display Test Metrics**

****

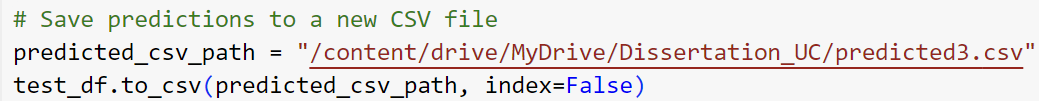
Description**:**

* + First Print Statement (Test Classification Report:):
    - * Purpose: Introduces and labels the upcoming output as the classification report. This helps in understanding that the following printed information relates to precision, recall, and F1 scores for each class.
      * Output: The test\_classification\_report, which is generated by classification\_report from sklearn.metrics, provides detailed metrics for each class including precision, recall, and the F1 score, along with overall averages (macro, weighted, and possibly micro).
  + Second Print Statement (test\_classification\_report):
    - * Content: Displays the full text of the classification report that details the performance of the model across all classes.
  + Third Print Statement (Test Confusion Matrix:):
    - * Purpose: Signals that the next output is the confusion matrix, setting the context for interpreting the array format data that quantifies the model’s classification performance.
  + Fourth Print Statement (test\_confusion\_matrix):
    - * Content: Outputs the confusion matrix itself, which is a table where each row represents the instances of an actual class, while each column represents the instances of a predicted class. Correct predictions are located in the diagonal of the matrix, whereas off-diagonal entries indicate misclassifications.

Importance and Utility

* Transparency and Interpretation:
  + Displaying these metrics enhances transparency about the model's performance and assists in quickly identifying areas of strength and weakness.
  + Stakeholders can understand the model's effectiveness and limitations, including team members not directly involved in the modeling process.
* Decision Making:
  + Helps in making informed decisions regarding the model's deployment. For instance, if certain classes show significantly lower performance, it might indicate a need for additional training data or a review of the model’s ability to generalize.
  + These metrics are crucial for regulatory and compliance purposes in industries where model accuracy and fairness are closely monitored.
* Error Analysis:
  + The confusion matrix is particularly useful for identifying specific types of errors (e.g., consistently confusing two particular classes), which can guide further data collection, feature engineering, or tweaking of the model architecture.

**Saving Predictions to a CSV File**



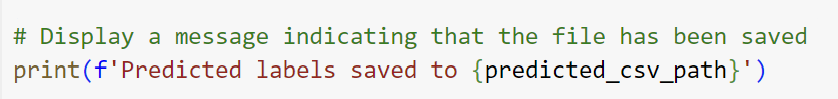
Description:

* File Path Specification:
  + Variable: predicted\_csv\_path
  + Content: Specifies the path where the new CSV file containing the predictions will be saved. The path points to a location in Google Drive, which is common when using Google Colab for easy access and storage integration.
* DataFrame to CSV:
  + Function: to\_csv()
  + Purpose: Writes the DataFrame test\_df, which now includes the Predicted Labels column along with any other data it originally contained, to a CSV file.
* Parameters:
  + predicted\_csv\_path: The file path where the CSV file will be saved.
  + index=False: This parameter ensures that the DataFrame's index (row numbers) is not saved in the CSV file, which helps keep the data clean and relevant, especially when the index is not meaningful.

Importance and Utility

* Documentation and Record-Keeping:
  + Saving the predictions to a CSV file provides a permanent, accessible record of the model's outputs. This is essential for traceability and accountability in model deployment and performance evaluation.
* Further Analysis:
  + With predictions saved in a CSV format, it's easy to use tools like Excel, Python, R, or other data analysis software to analyze the predictions further. Analysts or business users can examine the results, compare them with actual outcomes if those are available, or use the data for presentations and reports.
* Sharing and Collaboration:
  + Saving the file to a commonly accessible location like Google Drive facilitates easy sharing with team members, stakeholders, or external parties. It supports collaborative review and decision-making processes regarding the model’s deployment or further development.

**Display Save Confirmation Message**

****

Description:

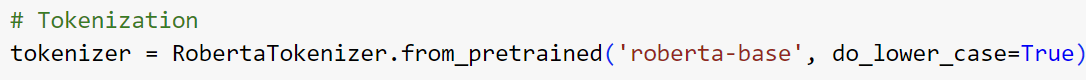
* Purpose: To inform the user that the operation of saving the predicted labels to a CSV file has been completed successfully.
* Output: The message dynamically includes the path to the file (predicted\_csv\_path), making it clear exactly where the file has been saved. This can be particularly helpful in environments where multiple output files are generated, or when paths might change based on variables or conditions.

Importance and Utility

* User Feedback: Providing immediate and clear feedback in the form of a print statement helps in monitoring the status of scripts, especially those that might run for long periods or involve multiple steps. It confirms that a specific crucial step—saving predictions—has been completed without errors.
* Debugging Aid: In case there are issues with subsequent parts of the script or if the file is not found where expected, this confirmation message can serve as a checkpoint to verify that at least the file saving was intended to be successful, aiding in troubleshooting.
* Workflow Clarity: For users or colleagues who might use or review the script later, such print statements clarify what the script is expected to do at each stage, making the code easier to understand and maintain.

**Multiclass Text Classification Implementing RoBERTa**

* Tokenization with RoBERTa



Code: tokenizer = RobertaTokenizer.from\_pretrained('roberta-

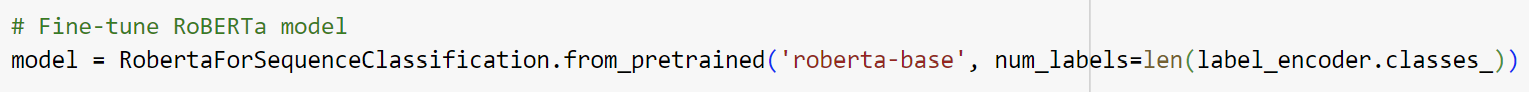
base', do\_lower\_case=True)

Description:

* + RobertaTokenizer: This is a class from the Hugging Face transformers library. It is designed to tokenize text into tokens that RoBERTa can understand. The tokenizer breaks down the text into words or subwords, which are then mapped to token IDs defined in RoBERTa's vocabulary.
  + from\_pretrained('roberta-base', do\_lower\_case=True): This function loads the tokenizer with a pre-trained vocabulary from the 'roberta-base' model. The do\_lower\_case=True parameter indicates that the text should be converted to lowercase before tokenization, which is typical for 'uncased' models that do not distinguish between uppercase and lowercase letters.

Importance:

* Preparation for Model Input: Tokenization is the first step in preparing raw text for processing by RoBERTa. It ensures that the input text is correctly formatted and aligned with the model’s training data.
* Efficiency and Accuracy: Using a pre-trained tokenizer helps maintain consistency and leverages prior knowledge embedded in the tokenizer, improving the model's efficiency and accuracy.
* Fine-tuning RoBERTa for Sequence Classification



Code: model = RobertaForSequenceClassification.from\_pretrained

('roberta-base', num\_labels=len(label\_encoder.classes\_))

Description:

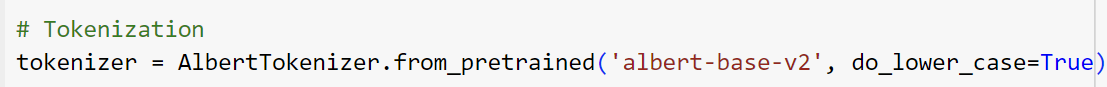
* + RobertaForSequenceClassification: This is a class from the Hugging Face transformers library, designed specifically for the task of sequence classification. It includes a RoBERTa model followed by a classification layer.
  + from\_pretrained('roberta-base', num\_labels=len(label\_encoder.classes\_)): Loads the RoBERTa model with weights pre-trained on the 'roberta-base' setup. The num\_labels=len(label\_encoder.classes\_) argument configures the model’s output layer to predict a fixed number of classes as determined by label\_encoder.classes\_, which represents the different classes in your dataset.

Importance:

* Model Customization: Fine-tuning allows the RoBERTa model to adapt to the specific characteristics of your dataset beyond the generic pre-training. By setting num\_labels, the model's final layer is customized to output predictions across the number of unique classes in your dataset.
* Leveraging Pre-training: Using a model pre-trained on a large corpus (like the one RoBERTa was trained on) and then fine-tuning it on a specific task can significantly improve performance, especially when the available labeled data for the task is limited.

**Multiclass Text Classification Implementing ALBERT**

* Tokenization with ALBERT



Code: tokenizer = AlbertTokenizer.from\_pretrained('albert-base-

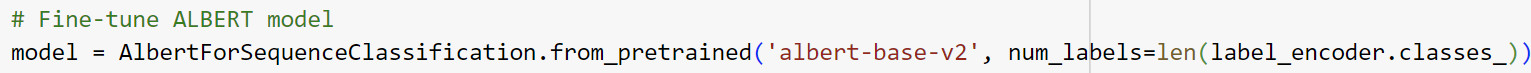
v2', do\_lower\_case=True)

Description:

* AlbertTokenizer: This is a tokenizer class from the Hugging Face transformers library designed for the ALBERT model. The tokenizer is responsible for converting text into a format suitable for the model, including splitting the text into tokens and converting these tokens into their corresponding IDs from ALBERT’s vocabulary.
* from\_pretrained('albert-base-v2', do\_lower\_case=True): This method loads the tokenizer with a pre-trained vocabulary from the 'albert-base-v2' model. The do\_lower\_case=True option ensures that the text is converted to lowercase before tokenization, which is crucial for consistent processing as the model is case-insensitive.

Importance of Tokenization:

* Preprocessing: Proper tokenization ensures that the model receives input in the correct format, which is crucial for achieving high performance.
* Compatibility: Using the tokenizer that matches the model architecture ensures compatibility and maximizes the model's efficiency by leveraging pre-built optimizations.
* Fine-tuning ALBERT for Sequence Classification



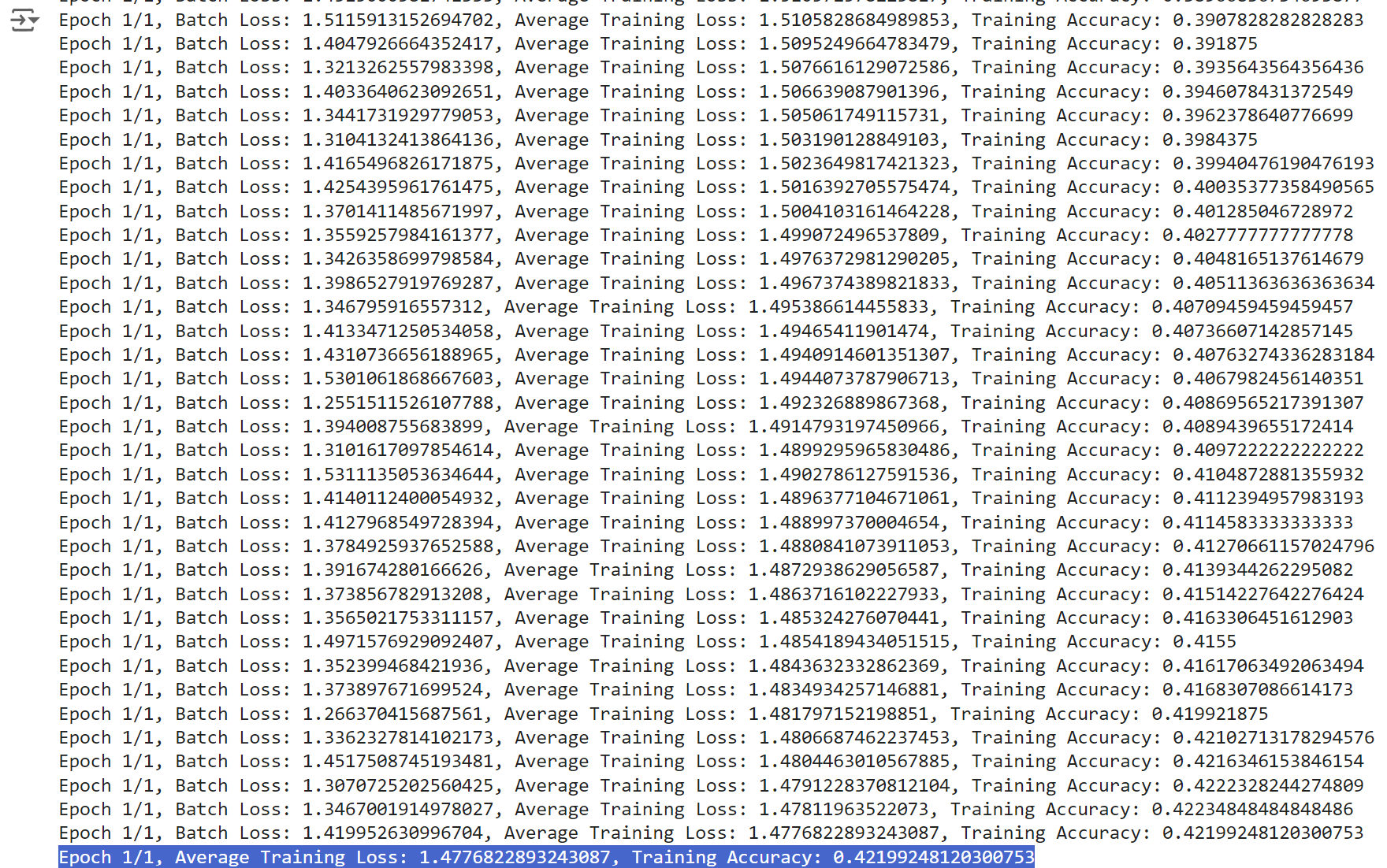
Code: model = AlbertForSequenceClassification.from\_pretrained

('albert-base-v2', num\_labels=len(label\_encoder.classes\_))

Description:

* AlbertForSequenceClassification: This is a model class tailored specifically for sequence classification tasks. It combines the ALBERT architecture with a classification layer on top, making it suitable for tasks where the goal is to predict which of several categories an input belongs to.
* from\_pretrained('albert-base-v2', num\_labels=len(label\_encoder.classes\_)): Initializes the model with weights from a pre-trained 'albert-base-v2' and configures the output layer to handle the correct number of classes as determined by label\_encoder.classes\_. This number represents the unique labels in your dataset.

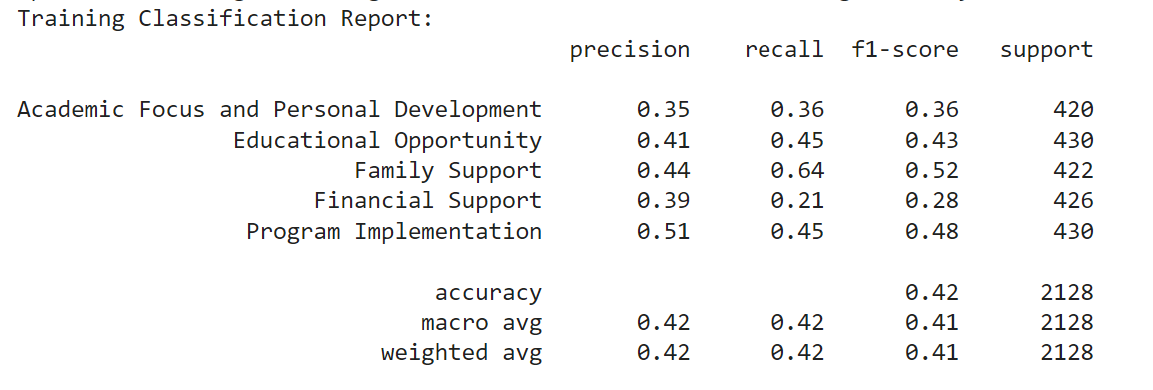
Sample Output for BERT Multiclass Text Classification



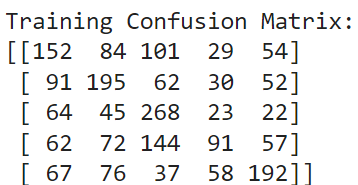
Components of the Output

* Epoch and Batch Information: "Epoch 1/1" indicates that these reports are from the first and only epoch planned in this training session. Each batch represents a subset of the training data used to update the model's weights.
* Batch Loss: This is the loss computed for each batch, which measures how well the model's predictions matched the actual outcomes in that batch. Loss functions (like cross-entropy loss for classification tasks) quantify the difference between the expected outcomes and the predictions. A lower loss indicates better performance.
* Average Training Loss: This is the cumulative average of the batch losses up to the current batch. It provides a more general view of how well the model is learning over time. As more batches are processed, this average should ideally decrease, indicating improved model performance.
* Training Accuracy: This metric reflects the percentage of predictions that were correct in each batch. It's a direct indicator of how effectively the model is performing its classification task. Higher accuracy means the model is making more correct predictions.
* Average Training Loss:
  + Value: 1.4776822893243087
  + Interpretation: This is the average loss computed across all batches of data during the epoch. The loss function (not specified here but often something like cross-entropy for classification tasks) measures the disparity between the model's predictions and the actual targets. The reported value indicates the model's average error rate in prediction per example across the training dataset.
  + Context: A lower average loss is desirable as it indicates better model performance. The specific value's interpretability, however, depends on the context of the problem and the loss function used. It is hard to conclusively rate the performance without knowing the baseline or threshold loss values for "good" performance specific to this task. However, seeing the loss decrease over successive epochs would typically be a positive sign.
* Training Accuracy:
  + Value: 42.199248120300753%
  + Interpretation: This percentage represents the model's accuracy over the entire epoch. It is the proportion of predictions the model got right out of all the predictions made.
  + Context: Accuracy is an intuitive measure of performance, especially for classification problems. An accuracy of approximately 42.2% suggests that the model correctly predicts the outcome for about 42.2% of the cases in the training set. This level of accuracy may indicate that the model is struggling to learn effective patterns from the data, which might be due to several factors such as insufficient model complexity, inadequate training (needing more epochs), or issues with the dataset itself (like imbalances or insufficient feature representation).

Training Classification Report

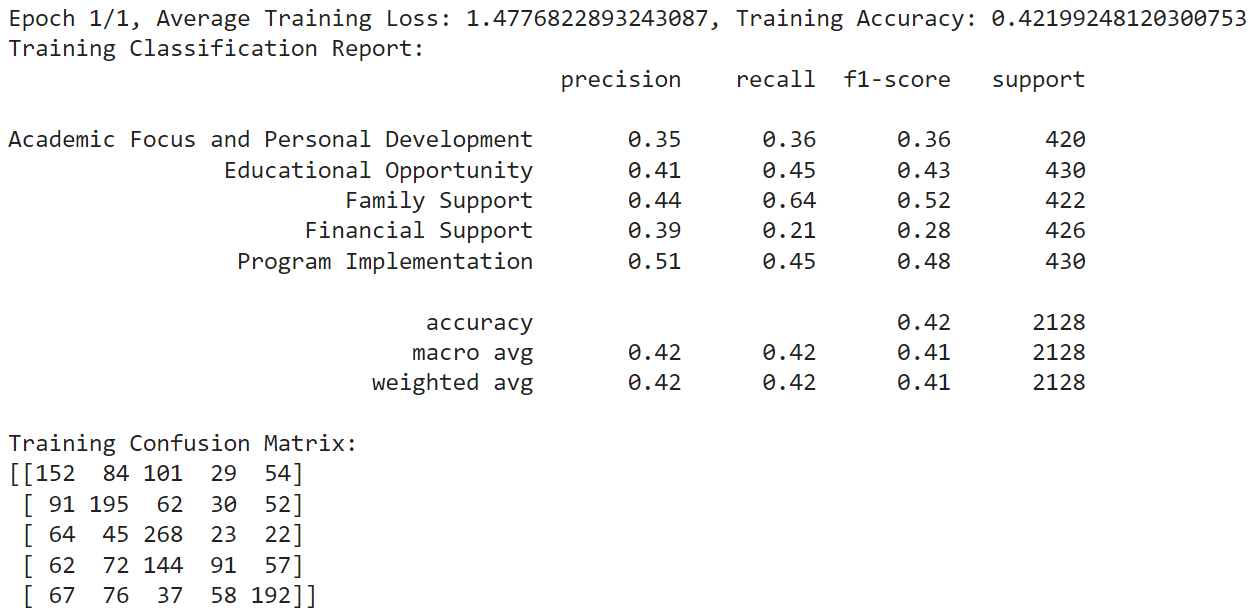


The classification, validation, and test reports provided a detailed evaluation of a model's performance. Each class's precision, recall, and F1-scores are reported, along with overall accuracy and average metrics.

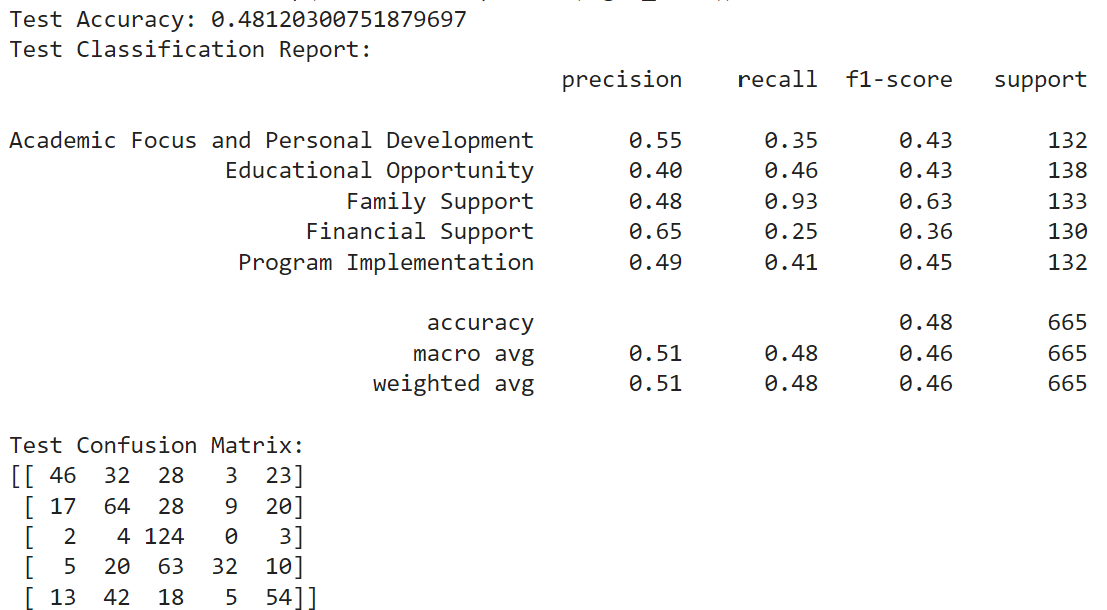


The confusion matrix offers valuable insight into how the model's predictions are distributed across different actual classes. This type of matrix is a powerful tool for understanding both the strengths and weaknesses of a classifier, especially in how it handles multi-class problems.

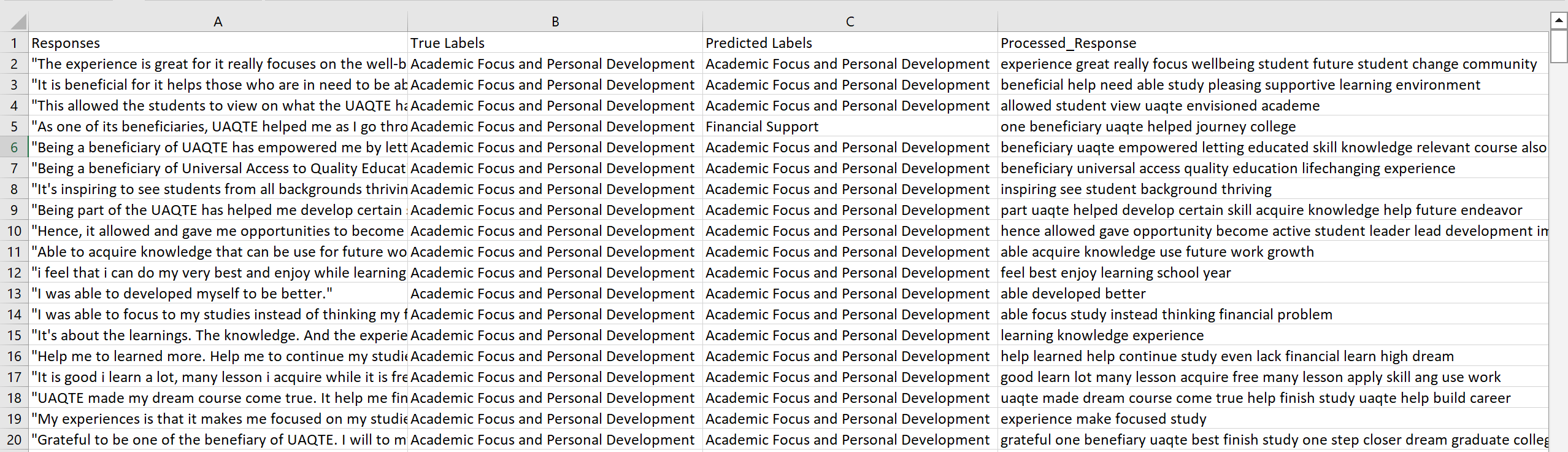
Validation Classification Report and Confusion Matrix



Test Classification Report and Confusion Matrix

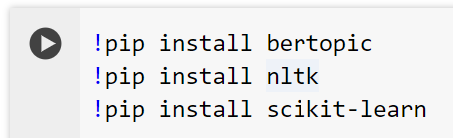


Predicted labels in CSV format



Topic Modeling using BERTopic

**Installation of Libraries**

****

!pip install bertopic

* Purpose: BERTopic is an advanced library for topic modeling using state-of-the-art language models based on transformers (like BERT and variants). It leverages these models to create dense clusters of topics from documents, enhancing the ability to discover more coherent and meaningful topics compared to traditional topic modeling techniques such as LDA.
* Functionality: It uses UMAP for dimensionality reduction, HDBSCAN for clustering, and c-TF-IDF to construct topic representations.

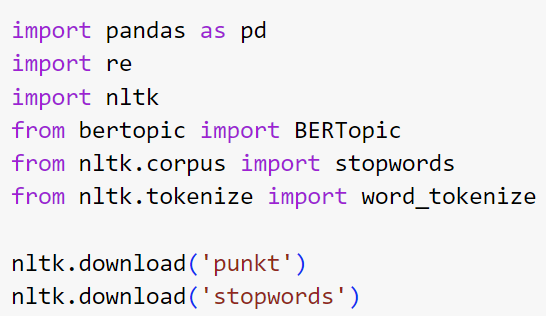
!pip install NLTK (Natural Language Toolkit)

* Purpose: NLTK is a powerful library for working with human language data (text), providing easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.
* Functionality: In the context of BERTopic, NLTK can be used for preprocessing tasks like tokenization, stop-word removal, and other linguistic analyses that might be necessary before feeding data into the topic modeling pipeline.

!pip install scikit-learn

* Purpose: scikit-learn is one of the most popular machine learning libraries for Python. It features various classification, regression, clustering algorithms, and tools for model fitting, data preprocessing, model selection, and evaluation.
* Functionality: In topic modeling, scikit-learn can be utilized for features like vectorization of text data, performing dimensionality reduction, and various other machine learning tasks that support the topic modeling process.

**Importing Libraries and Functions**



* pandas: A powerful data manipulation and analysis library for Python. It is typically used for reading, writing, and transforming data, particularly useful for handling structured data in tabular form.
* re: Python's built-in library for regular expression operations, which allows for searching, matching, or splitting text based on patterns.
* nltk: Stands for Natural Language Toolkit, a suite of libraries and programs for symbolic and statistical natural language processing (NLP) for English.
* BERTopic: A library for extracting topics from collections of text documents using transformers and advanced clustering algorithms.
* stopwords: A module in NLTK containing lists of stopwords that are commonly omitted in text processing tasks (e.g., "and", "the", etc.).
* word\_tokenize: A function from NLTK used to split text into words, effectively turning a body of text into a list of tokens.

Downloading NLTK Resources

* punkt: This is a pre-trained model that helps the word\_tokenize function to tokenize words and sentences. Tokenizers divide strings into lists of substrings. For example, tokenizers can divide a text into words or sentences. The 'punkt' package is a pre-trained model that understands languages' structure and helps split text into a list of sentences.
* stopwords: This resource provides a list of 'stopwords' for several languages. Stopwords are generally the most common words in a language and are often removed during text preprocessing to focus on meaningful words. This is crucial for NLP tasks, including topic modeling, where common words add little value to understanding the content's themes.
* By setting up your environment with these libraries and resources, you equip yourself with a robust toolkit for performing sophisticated NLP tasks, including but not limited to topic modeling, sentiment analysis, and text classification.

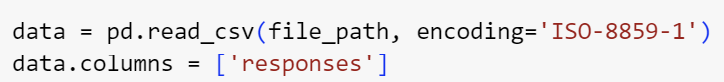
**Loading Data**



Description:

* Variable Assignment: The file\_path variable is assigned a string that specifies the location of a CSV file. This string is the path to the file within the Google Drive directory structure.
* File Path Details:
  + /content/drive/: This is the root directory for Google Drive in a Google Colab environment, assuming that the Google Drive has been mounted successfully to the Colab notebook.
  + MyDrive/ refers to the user's main Google Drive folder.
  + Dissertation\_UC/: Likely a specific folder created by the user for organizing files related to a dissertation or a similar academic project.
  + STUDENT\_SURVEY\_EXIT.csv: The name of the CSV file containing data, presumably survey data from students.

Reading the CSV File with Specific Encoding



Code: pd.read\_csv(file\_path, encoding='ISO-8859-1')

Description:

* Functionality: This line of code loads a CSV file into a pandas DataFrame. The encoding='ISO-8859-1' parameter is particularly important because it specifies the character encoding used in the file. ISO-8859-1, also known as Latin-1, covers Western European languages, which might be necessary if the data includes special characters not supported by the default UTF-8 encoding.
* Purpose: Ensuring the correct encoding is used when reading a file prevents errors related to character decoding, especially with data that may have been created or saved in non-standard formats or from different geographical regions.

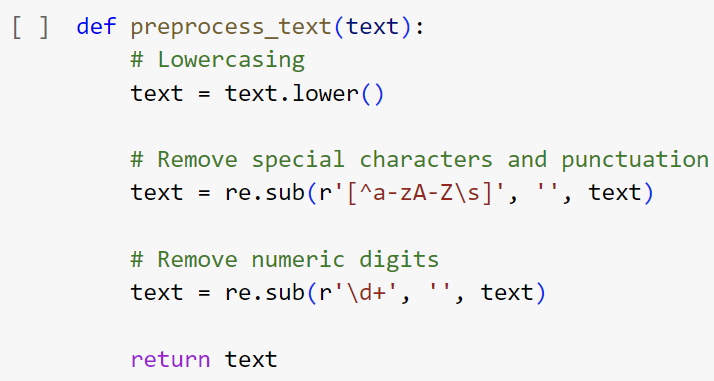
Setting DataFrame Column Names

Code: data.columns = ['responses']

Description:

* Functionality: This command explicitly sets the column names of the DataFrame. If the CSV file was loaded without a header row, or if you want to rename the columns to something more descriptive or convenient for your analysis, this step is crucial.
* Purpose: Naming columns appropriately can greatly simplify subsequent data manipulation and analysis steps by providing clear and meaningful identifiers for each data column.

**Preprocessing steps**



* Lowercasing

Code: text = text.lower()

* + Purpose: Converts all characters in the text to lowercase. This is a crucial step in text preprocessing because it helps in standardizing the text data. For example, words like "Apple", "apple", and "APPLE" are considered the same word after this transformation.
  + Effect: Reduces complexity for downstream processing by treating the same words uniformly, regardless of how they appear in the text (e.g., beginning of a sentence vs. middle).
* Remove Special Characters and Punctuation

Code: text = re.sub(r'[^a-zA-Z\s]', '', text)

* + Regular Expression Explained:
    - [^a-zA-Z\s]: Matches any character that is not a letter (a-z or A-Z) or a whitespace character (\s).
    - The caret (^) inside the square brackets denotes a negation, so the pattern matches any character not listed.
  + Purpose: Cleans the text by removing punctuation and special characters that might not be useful in many text analysis contexts. This includes characters like !, @, #, &, and punctuation such as commas, periods, and question marks.
  + Effect: Simplifies the text to only contain words and spaces, making it more uniform and often improving the performance of natural language processing tasks by focusing only on meaningful words.
* Remove Numeric Digits

Code: text = re.sub(r'\d+', '', text)

* + Regular Expression Explained:
    - \d+: Matches one or more digits. The + quantifier means "one or more" of the preceding element (\d for digits).
  + Purpose: Eliminates numbers from the text, which is useful if numeric data does not provide meaningful information for the analysis or model. For instance, in sentiment analysis or topic modeling, numbers might not contribute to understanding the sentiment or the topics.
  + Effect: Further cleans up the text to focus strictly on words, removing potential noise introduced by numbers.

**Text Normalization**



Description:

* DataFrame Operation: This line of code operates on the pandas DataFrame named data.
* Column Access and Creation: data['responses'] accesses the 'responses' column of the DataFrame, which presumably contains raw textual data collected from surveys or similar sources. data['processed\_responses'] creates a new column in the DataFrame or modifies it if it already exists.
* Apply Function: The apply(preprocess\_text) function applies the preprocess\_text function to each element in the 'responses' column. This function is designed to preprocess text by making all characters lowercase and removing special characters, punctuation, and numeric digits.
* Assignment: The results of the apply operation, which are the cleaned text strings, are stored in the new 'processed\_responses' column of the DataFrame.
* Scikit-learn: Silhouette Score

Code: from sklearn.metrics import silhouette\_score

Description:

* Silhouette Score: This metric is used to evaluate the quality of clusters created by an algorithm. It measures how similar an object is to its own cluster (cohesion) compared to other clusters (separation).
* Calculation: The silhouette score is calculated using each sample's mean intra-cluster distance and the mean nearest-cluster distance. The score ranges from -1 to +1, where a high value indicates that the clusters are well-separated and dense.
* Usage: Often used in clustering problems such as K-means or hierarchical clustering to determine the optimal number of clusters by comparing the silhouette scores for different numbers of clusters.
* Gensim: Coherence Model

Code: from gensim.models.coherencemodel import CoherenceModel

Description:

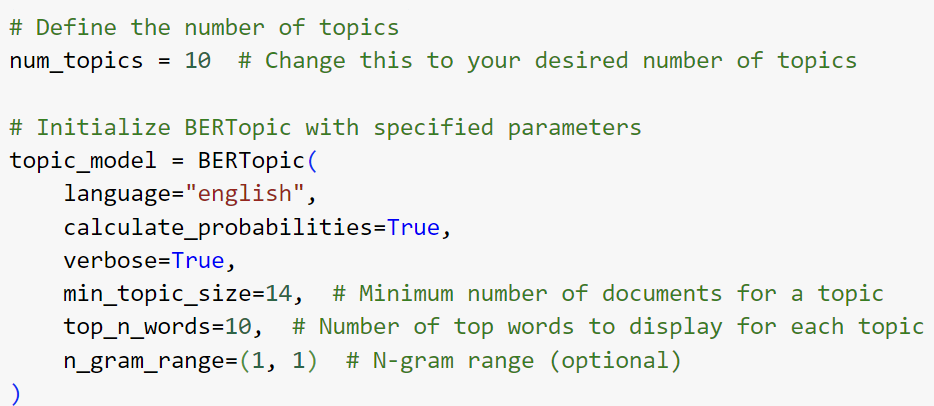
* Coherence Model: Used to evaluate the coherence of topics produced by topic models like Latent Dirichlet Allocation (LDA) or other models that generate topics from documents. Topic coherence measures how semantically similar the high scoring words in a topic are, which can help in assessing the human interpretability of these topics.
* Usage: After training a topic model, you can use the Coherence Model to calculate metrics like 'c\_v', 'u\_mass', 'c\_uci', and 'c\_npmi', which help in understanding how coherent the topics are. A higher coherence score generally means that the topic is more interpretable and semantically meaningful.
* Gensim: Dictionary

Code: from gensim.corpora import Dictionary

Description:

* Dictionary: This class is used to create a mapping between words and their integer ids. It is used to prepare text data before passing it to topic models in Gensim.
* Functionality: The Dictionary can be used to convert a collection of worded documents to a bag-of-words format; the subsequent corpus is a list of (word\_id, word\_frequency) tuples per document.
* Usage: Essential for preprocessing text for Gensim's topic modeling, as it provides a way to encode words as unique identifiers and to filter out extreme cases, such as very rare words or words that are too common.

**BERTopic Initialization Code**



* Define the Number of Topics

Code: num\_topics = 10

Description: This line sets a variable num\_topics that specifies the desired number of topics the model should try to find. While BERTopic is primarily an HDBSCAN-based model that does not require the number of topics to be set beforehand, you might use this variable to control post-processing steps or for informing other aspects of your analysis where a fixed number of topics is useful.

* BERTopic Model Initialization

Code: topic\_model = BERTopic(

language="english",

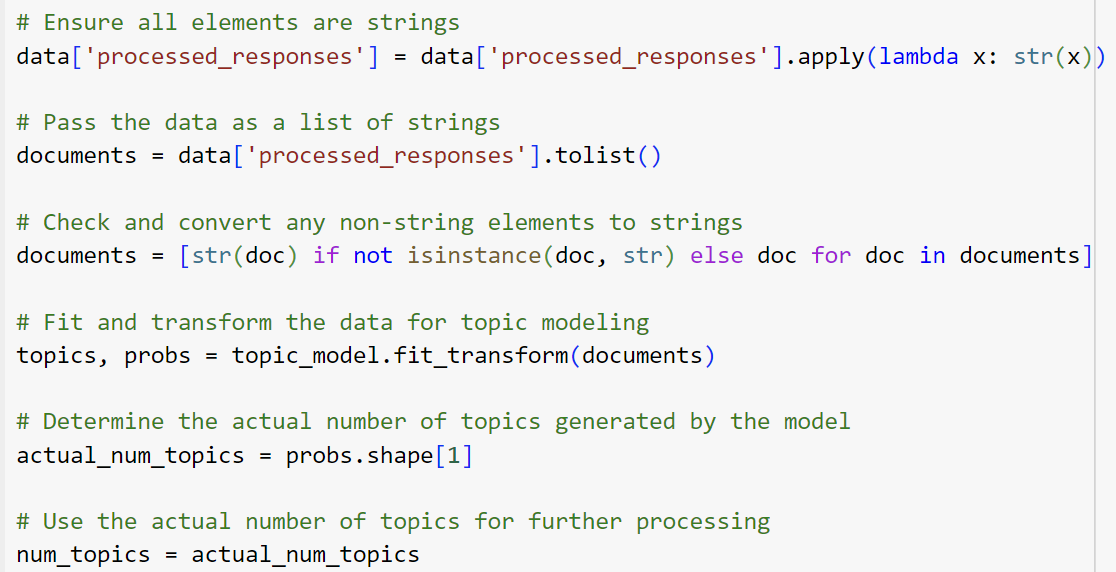
calculate\_probabilities=True,

verbose=True, min\_topic\_size=14, top\_n\_words=10,

n\_gram\_range=(1, 1))

Description:

* Parameters Explained:
* language: Specifies the model language, which influences the stopwords and other language-specific processing. "English" is set here, tailoring the model for English text data.
* calculate\_probabilities: Enables the calculation of probabilities for each topic. This is important for assessing the likelihood that a document belongs to each topic, which can be useful for detailed topic analysis and understanding document distributions.
* verbose: When set to True, the model outputs detailed logs of the processing stages, which can help in debugging or understanding the model's internal workings.
* min\_topic\_size: Sets the minimum number of documents that a cluster needs to be considered a topic. Here, it is set to 14, which means topics with fewer than 14 documents will not be formed. This parameter helps in filtering out noise and ensuring that topics are robust and substantiated by enough documents.
* top\_n\_words: Specifies the number of words to display for each topic when querying the topic descriptions. Setting this to 10 means each topic will be summarized by its 10 most significant words, aiding interpretability.
* n\_gram\_range: Defines the range of n-gram sizes for the text. (1, 1) means only unigrams (single words) are considered. Adjusting this to (1, 2) or higher can allow the model to also consider bigrams or larger n-grams, potentially capturing more nuanced topic descriptors like multi-word expressions.



* Ensure All Elements are Strings

Code: data['processed\_responses'] = data['processed\_responses'] .apply(lambda x: str(x))

Purpose: Converts all entries in the processed\_responses column of the DataFrame to strings. This step is crucial because the text processing functions and BERTopic require string input to function correctly. It prevents potential type-related errors during processing.

Method: The apply (lambda x: str(x)) method is used to apply the string conversion function to each element in the column.

* Convert DataFrame Column to List

Code: documents = data[‘processed\_responses’].tolist()

Purpose: Transforms the processed\_responses column into a list of strings. Topic modeling algorithms typically require a list of documents (where each document is a string) as input.

Method: The tolist() method converts the pandas Series into a standard Python list, making it compatible for use with BERTopic.

* Check and Convert Non-String Elements to Strings

Code: documents = [str(doc) if not isinstance(doc, str) else doc for doc in documents]

Purpose: Ensures that each element in the documents list is a string. This step is somewhat redundant given the initial conversion but acts as a failsafe to catch any elements that might not have been properly converted or were altered in subsequent operations.

Method: A list comprehension checks the type of each document; if any document is not a string, it is converted to a string.

* Fit and Transform the Data for Topic Modeling

Code: topics, probs = topic\_model.fit\_transform(documents)

Purpose: Feed the list of documents into the BERTopic model to fit the model and transform the text data into topics. This method returns two outputs: topics, which assigns a topic to each document, and probs, which provides the probability distribution of topics for each document.

Method: The fit\_transform method is a common pattern in machine learning and NLP for applying a model to data, combining the fitting (training) and transformation (assignment of labels or features) steps into one.

* Determine the Actual Number of Topics Generated by the Model

Code: actual\_num\_topics = probs.shape[1]

Purpose: Retrieves the actual number of topics that the model has determined based on the data. BERTopic uses a clustering approach that may result in different topics than initially expected or desired, depending on the data's inherent clustering.

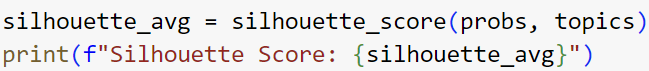
Method: The shape [1] accesses the second dimension of the probs array, which corresponds to the number of topics.

* Update the Number of Topics for Further Processing

Code: num\_topics = actual\_num\_topics

Purpose: Updates the num\_topics variable to reflect the actual number of topics found by BERTopic. This allows subsequent steps in the analysis to align with the model's output, ensuring consistency and relevance in further processing and analysis.

* Silhouette Score Calculation



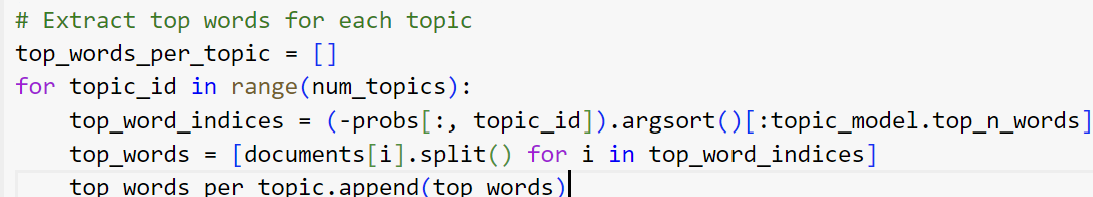
Code: silhouette\_avg = silhouette\_score(probs, topics)

Function Used: silhouette\_score from the sklearn.metrics module.

Inputs:

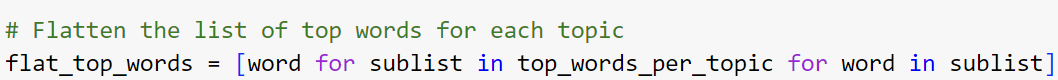
* + probs: This should be an array where each row corresponds to a document, and each column corresponds to the probability of that document belonging to a particular topic (cluster). These are the probabilities that documents belong to the topics as determined by BERTopic.
  + topics: This is typically an array or a list where each element is the cluster label (topic) assigned to each document. In the context of BERTopic, this would be the result from the fit\_transform method, which outputs the most likely topic for each document.
* Purpose: The Silhouette Score is calculated using both the distance between each point (document) and the points in its own cluster (cohesion) and the distance to points in the nearest cluster other than its own (separation). The score is a measure of how similar an object is to its own cluster compared to other clusters, with a higher score indicating better-defined clusters.

**Extracting Top Words for Each Topic**



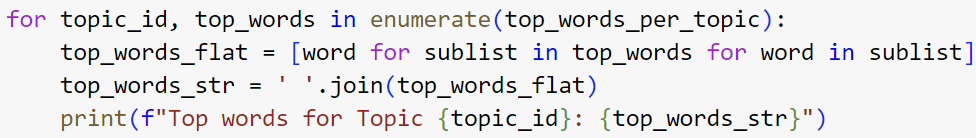
* Initialization: A list top\_words\_per\_topic is initialized to store the top words for each topic.
* Loop Over Each Topic: Iterates over each topic based on the num\_topics variable.
* Extract Top Word Indices:
  + (-probs[:, topic\_id]).argsort()[:topic\_model.top\_n\_words]: This line performs a few operations:
    - probs[:, topic\_id]: Selects the probability distribution for the topic\_id from the probs matrix, which contains the probabilities of each document belonging to each topic.
    - (-probs[:, topic\_id]): Negates the probabilities to use argsort(), which sorts elements in ascending order; by negating, the highest probabilities come first.
    - .argsort()[:topic\_model.top\_n\_words]: Sorts the indices of the documents based on their topic probabilities in descending order and takes the indices of the top words as specified by top\_n\_words.
* Extract Words:
  + [documents[i].split() for i in top\_word\_indices]: For each index in top\_word\_indices, the corresponding document is retrieved from documents, split into words, and the words are collected.
* Store Top Words: The list of top words for each topic is appended to top\_words\_per\_topic.

**Flatten the List of Top Words**



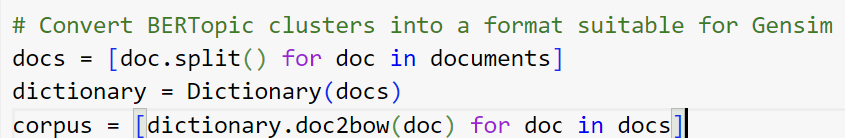
* Purpose: Creates a single list containing all words from all topics, flattening the nested list structure. This operation might be used for further analysis or visualization, though it is not used directly in the subsequent printing step.

**Print the Top Words for Each Topic**



* Enumerate and Iterate: Loops through each topic's top words, using enumerate to keep track of the topic\_id.
* Flatten the Words List: Flattens the nested list of words for each topic to make it suitable for displaying.
* Create String of Top Words: Joins the flattened list of words into a single string.
* Print Each Topic's Top Words: Displays a formatted string that lists the top words for each topic, aiding in the interpretation of what each topic is likely about.

**Converting Text Data to Gensim Format**



* Tokenize Documents

Code: docs = [doc.split() for doc in documents]

Purpose: Splits each document into a list of words. This is essential because Gensim's models and functions require text data to be tokenized into lists of strings.

* Create a Gensim Dictionary

Code: dictionary = Dictionary(docs)

Purpose: Creates a Gensim Dictionary object that maps each unique word to a unique integer ID. This object is crucial for converting text data into a sparse vector format that Gensim can work with.

* Convert Documents to Bag-of-Words (BoW) Format

Code: corpus = [dictionary.doc2bow(doc) for doc in docs]

Purpose: Transforms the list of words (docs) into a list of vectors where each vector represents a document. Each vector is a collection of tuples.

**Computing Coherence Scores**

* Initialize Coherence Model

Code: coherence\_model = CoherenceModel(

texts=docs, # Use the list of documents

corpus=corpus,

dictionary=dictionary,

topics=flat\_top\_words, # Provide flattened BERTopic top words

coherence='c\_v' # You can choose other coherence measures as well

)

* Parameters Explained
  + texts: The tokenized documents, which are used for certain types of coherence measures that require segmented texts.
  + corpus: The BoW corpus of the documents.
  + dictionary: The Gensim dictionary mapping of IDs to words.
  + topics: The list of words representing the top words for each topic. It's essential that this matches the expected format (a list of lists where each inner list contains the top words for a topic).
* coherence: Specifies the type of coherence measure to calculate. "c\_v" is commonly used and considered a good measure of topic coherence, assessing the degree of semantic similarity between high scoring words within the topic.
* Compute and Print Coherence Score

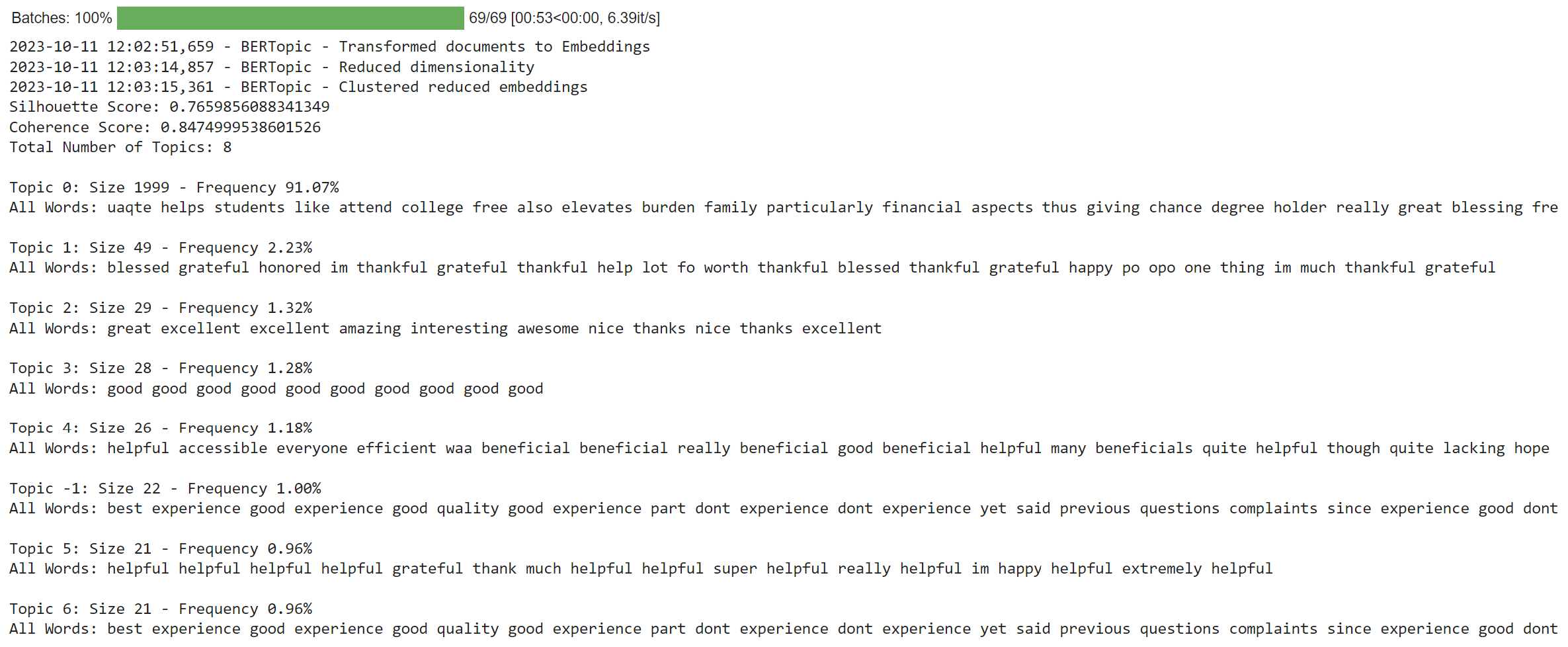
Code: coherence\_score = coherence\_model.get\_coherence()

print(f"Coherence Score: {coherence\_score}")

Purpose: Computes the coherence score for the set topics using the specified measure. The coherence score helps in evaluating how meaningful the topics are, with higher scores generally indicating more coherent and interpretable topics.

Output: Prints the coherence score, providing an easy-to-interpret metric that assesses the quality of the topics derived from the BERTopic model.

**Sample Output**

****

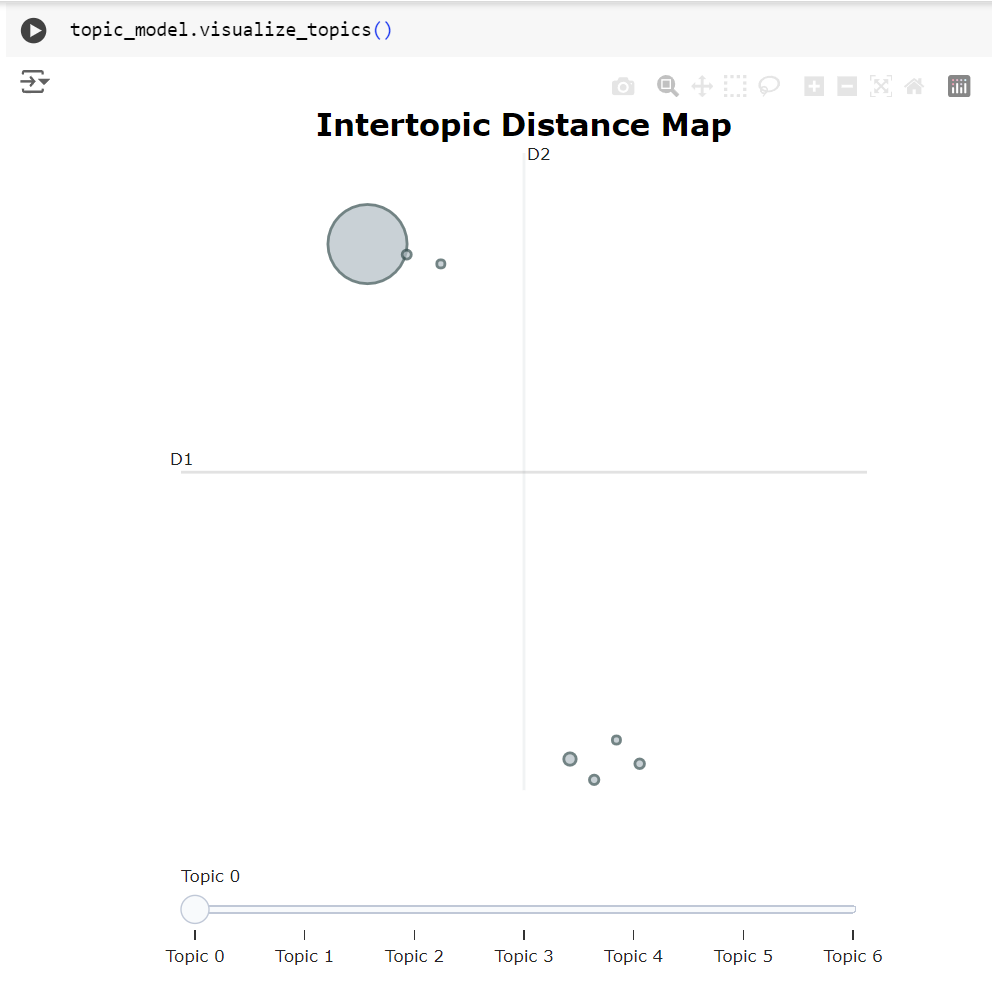
The output provided a comprehensive summary of a topic modeling session conducted using BERTopic, which includes logs of processing steps, evaluation scores, and detailed descriptions of the topics identified.

**Process Log and Metrics**

* Batches Processed:
  + Detail: 69/69 [00:53<00:00, 6.39it/s] shows that all 69 batches were processed in 53 seconds, with an average speed of 6.39 iterations per second.
  + Interpretation: Indicates efficient processing and completion of the batch operations without interruption.
* BERTopic Processing Steps:
  + Transformed documents to Embeddings: This log entry indicates that the raw text documents were converted into embeddings, likely using a transformer-based model like BERT. Embeddings are vector representations that capture the semantic properties of the text.
  + Reduced dimensionality: Suggests the application of a dimensionality reduction technique (likely UMAP) to the embeddings, which helps in clustering by simplifying the data while retaining essential information.
  + Clustered reduced embeddings: Indicates that clustering (likely using HDBSCAN) was applied to the dimensionality-reduced data to identify distinct groups or topics based on document similarity.
* Evaluation Scores:
  + Silhouette Score: 0.7659856088341349: A high silhouette score close to 1, which suggests that the clusters (topics) are well-defined and clearly distinguishable from each other.
  + Coherence Score: 0.8474999538601526: Also a high coherence score, indicating that the words within each topic frequently co-occur in a meaningful way, making the topics semantically coherent and interpretable.

**Topic Summary**

* Total Number of Topics: 8: Indicates the model identified 8 distinct topics (note: Topic -1 is typically used in BERTopic to represent noise or outlier documents that do not fit well into any other topic).
* Detailed Topic Descriptions:
* Topics 0 through 6: Each topic is summarized by the size (number of documents it contains), its frequency (percentage of total documents), and a list of characteristic words or phrases.
* Topic 0: The largest and most dominant topic, with a broad range of words suggesting a focus on financial aspects, educational experiences, and gratitude.
* Topic 1: Characterized by expressions of gratitude, this topic seems to reflect positive personal sentiments.
* Topic 2: Contains adjectives like "great," "excellent," and "amazing," likely reflecting positive evaluations.
* Topic 3 and others: Smaller topics with more focused or less diverse word sets reflect specific user experience sentiments or aspects.
* Topic -1: Often represents documents that did o fit well into other topics, showing a mix of general and disjointed terms related to experiences and expectations.

****

The method topic\_model.visualize\_topics() from BERTopic is used to generate a visualization of the topics discovered during the topic modeling process. This visualization can be extremely helpful for quickly understanding the relationships between different topics and the overall structure of the data.

* Overview of visualize\_topics()
* Purpose: This function creates a two-dimensional visualization of the topics that have been identified by BERTopic. It uses dimensionality reduction techniques (usually UMAP or t-SNE) to project the high-dimensional topic vectors into a two-dimensional space where similar topics are placed closer together.
* Output: The output is usually an interactive plot, often implemented using libraries like Plotly or Matplotlib in a Jupyter Notebook or a Python script environment. This allows users to hover over points (topics) to see additional details or to click and zoom for a closer look.
* Interpreting the Visualization
* Clusters of Topics: Each point or node in the visualization represents a topic. Topics that are similar to each other based on their content (common significant words and themes) are typically closer together in the space, forming clusters. These clusters can indicate overarching themes in the dataset.
* Distance Between Topics: The spatial distance between any two topics in the visualization can be interpreted as a measure of their dissimilarity. Closer topics are more similar, while topics that are further apart are less related. This can help in understanding how distinct certain topics are within the dataset.
* Interactive Elements
* Hovering: Moving your cursor over a topic often displays a tooltip with information such as the topic number and the top words that characterize the topic. This feature makes it easier to explore the content of each topic without needing to refer back to other outputs or lists.
* Zooming and Panning: These features allow for detailed examination of areas where topics might be densely packed together, helping to discern subtle differences or similarities between closely located topics.