Sentiment Analysis for Marketing

PHASE -4

DATE	24 October 2023
TEAM ID	Proj-212173-Team-1
PROJECT NAME	Sentiment analysis for Marketing

Performing Sentiment Analysis On Airlines Dataset From Twitter Using Bert

BERT:

BERT, which stands for **Bidirectional Encoder Representations from Transformers**, is a highly influential natural language processing (NLP) model developed by Google in 2018. BERT revolutionized the field of NLP by introducing a new approach to pre-training models for understanding and generating human language.

Key characteristics and concepts of BERT :

1. Transformer Architecture: BERT is based on the Transformer architecture, which is a deep learning model architecture designed for sequential data. Transformers are known for their ability to handle long-range dependencies in data effectively.

2. Bidirectional Pre-training: Unlike previous NLP models, which were pre-trained in a unidirectional manner (either left-to-right or right-to-left), BERT is pre-trained in a bidirectional manner. It considers both the left and right context of a word when encoding it. This bidirectional approach allows BERT to capture richer contextual information and improve understanding of language.

3. Large-scale Pre-training: BERT is pre-trained on a massive amount of text data, typically containing billions of words. This pre-training allows it to learn general language understanding from a diverse range of text sources, making it highly adaptable to various downstream NLP tasks.

4. Two-Stage Training:

Pre-training: BERT's first stage is pre-training on a large corpus of text. It learns to predict missing words in sentences (masked language model) and determine whether two sentences follow each other (next sentence prediction). This process results in a powerful language model.

Fine-tuning: In the second stage, BERT can be fine-tuned on specific NLP tasks. By adding task-specific layers on top of the pre-trained model and training it on labeled data, BERT can be adapted for various tasks, such as text classification, named entity recognition, sentiment analysis, and more.

5. Contextual Embeddings: BERT produces contextual word embeddings, meaning that the representation of a word can vary based on the context in which it appears. This contextual understanding is a significant improvement over static word embeddings like Word2Vec or GloVe.

6. Multilingual BERT: BERT models exist for multiple languages, allowing it to be used for NLP tasks in different languages.

Steps to perform sentiment analysis using BERT:

1. Data Preparation:

Gather and preprocess your sentiment analysis dataset. This dataset should consist of text samples (e.g., customer reviews, social media posts) along with their associated sentiment labels (e.g., positive, negative, neutral).

2. Data Splitting:

Split your dataset into three subsets: training, validation, and testing. A common split might be 70% for training, 15% for validation, and 15% for testing. These subsets are used to train, validate, and evaluate your sentiment analysis model.

3. Tokenization:

Tokenize the text data using the BERT tokenizer. BERT requires specific tokenization, including adding [CLS] and [SEP] tokens and generating token IDs and attention masks.

4. Model Selection:

Choose a pre-trained BERT model. You can use a pre-trained model from libraries like Hugging Face's Transformers (e.g., 'bert-base-uncased') or fine-tune a BERT model specifically for your task.

5. Model Architecture:

Define the architecture of your sentiment analysis model. Typically, this involves adding a classification layer on top of the pre-trained BERT model to predict sentiment labels.

6. Fine-Tuning:

Fine-tune the pre-trained BERT model on your sentiment analysis dataset. This step adapts the model's parameters to your specific task. Train the model using the training dataset and validate it using the validation dataset. Monitor metrics like accuracy, loss, and others during training.

7. Hyperparameter Tuning:

Experiment with different hyperparameters such as batch size, learning rate, and dropout rate to optimize the model's performance on the validation dataset.

8. Evaluation:

After fine-tuning, evaluate your model on the test dataset to assess its generalization performance. Common evaluation metrics for sentiment analysis include accuracy, precision, recall, F1-score, and confusion matrices.

9. Inference:

Use the trained model to make sentiment predictions on new, unlabeled text data.

10. Post-processing:

Analyze the model's predictions and perform any post-processing steps if necessary. For example, you might convert probability scores into sentiment labels and apply a confidence threshold.

11. Visualization and Reporting:

Visualize and report the results of your sentiment analysis, including insights into sentiment trends and patterns.

12. Deployment (if applicable):

If sentiment analysis is part of a real-world application, deploy the model in a production environment where it can analyze sentiment in real-time.

13. Model Maintenance:

Monitor the performance of your sentiment analysis model in the production environment and retrain it periodically if needed to adapt to changing data patterns.

CODE AND OUTPUT:

```
[13] MODEL = 'bert-base-uncased'
     tokenizer = BertTokenizer.from_pretrained(MODEL, do_lower_case=True)
```

```
Downloading (…)okenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 664B/s]
Downloading (…)solve/main/vocab.txt: 100% 232k/232k [00:00<00:00, 819kB/s]
Downloading (…)/main/tokenizer.json: 100% 466k/466k [00:00<00:00, 12.5MB/s]
Downloading (…)lve/main/config.json: 100% 570/570 [00:00<00:00, 19.6kB/s]
```

EXPLANATION:

1. MODEL = 'bert-base-uncased': In this line, you define the variable `MODEL` and set it to the string `"bert-base-uncased"`. This is specifying the pre-trained BERT model you want to use. BERT is a popular NLP model developed by Google, and `"bert-base-uncased"` specifically refers to the "base" version of the BERT model that has been trained on a large corpus of text data in English. The "uncased" part indicates that the model has been trained on lowercase text and doesn't distinguish between uppercase and lowercase letters.

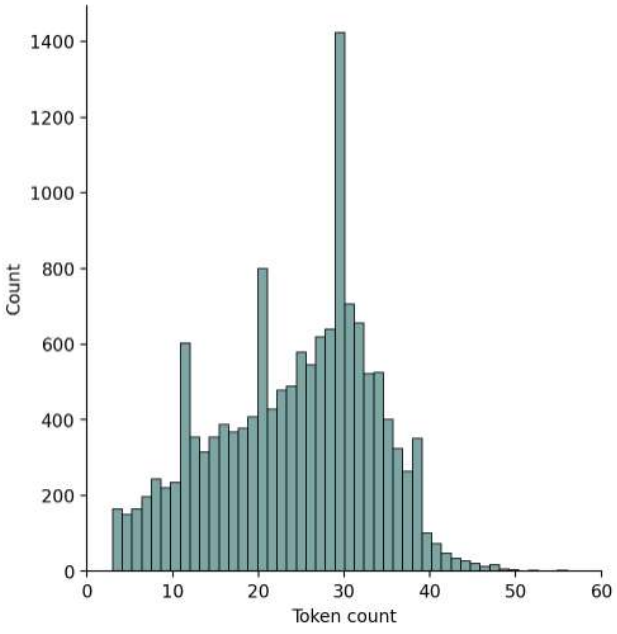
2. `tokenizer = BertTokenizer.from_pretrained(MODEL, do_lower_case=True)`: In this line, you are initializing a tokenizer for the BERT model. Tokenizers are used to split text into smaller units (tokens) that can be processed by the model. Here's what's happening in this line:

`BertTokenizer.from_pretrained(MODEL)`: This part of the code initializes a BERT tokenizer by loading the pre-trained tokenizer associated with the specified model, in this case, `"bert-base-uncased"`. The tokenizer is responsible for splitting text into subword tokens and converting them into numerical representations that can be fed into the BERT model.

`do_lower_case=True`: This argument tells the tokenizer to perform text tokenization while converting all text to lowercase. Since the BERT model you've chosen, `"bert-base-uncased"`, is trained on lowercase text, setting `do_lower_case` to `True` ensures that the input text is converted to lowercase during tokenization.

CODE AND OUTPUT:

```
[14] sns.displot(tokens)
     plt.xlim([0, 60]);
     plt.xlabel('Token count');
```



EXPLANATION:

**1. `tokens = [len(tokenizer.encode(text, max_length=512, truncation=True))** for text in df.text]`: In this line, you are iterating through each text sample in the DataFrame `df`. For each text sample (`text`), the code does the following:

`tokenizer.encode(text, max_length=512, truncation=True)`: This part of the code uses the BERT tokenizer (`tokenizer`) to tokenize the input text (`text`). The `max_length` parameter specifies the maximum length for the tokenized sequence. If the tokenized sequence exceeds this length, it will be truncated to fit within the limit. The `truncation=True` argument indicates that if the text is too long, it should be truncated to fit within the `max_length`.

`len(...)`: The `len()` function is then used to count the number of tokens in the tokenized sequence.

2. `sns.displot(tokens)`: This line of code uses Seaborn's `displot` function to create a histogram (distribution plot) of the token counts. The `tokens` list you computed in the previous step is used as the data for the histogram. This plot will show you the distribution of token counts across the text samples in your DataFrame.

3. `plt.xlim([0, 60])`: This line sets the x-axis (horizontal axis) limits of the plot. It restricts the range of token counts displayed on the x-axis to be between 0 and 60.

4. `plt.xlabel('Token count')`: This line labels the x-axis with the text "Token count," indicating that the x-axis represents the number of tokens.

CODE AND OUTPUT:

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

# Define a custom dataset, more info on how to build custom dataset can be
# found at https://pytorch.org/tutorials/beginner/data_loading_tutorial.html
class CustomDataset(Dataset):

    def __init__(
        self,
        tweets,
        labels,
        tokenizer,
        max_length
    ):
        self.tweets = tweets
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.tweets)

    def __getitem__(self, idx):
        tweet = self.tweets[idx]
        label = self.labels[idx]
```

```
    tokenize = self.tokenizer.encode_plus(
        tweet,
        add_special_tokens=True,
        max_length=self.max_length,
        return_token_type_ids=False,
        padding='max_length',
        return_attention_mask=True,
        return_tensors='pt'
    )
    return {
        'tweet': tweet,
        'input_ids': tokenize['input_ids'].flatten(),
        'attention_mask': tokenize['attention_mask'].flatten(),
        'targets': torch.tensor(label, dtype=torch.long)}
```

EXPLANATION:

1. `__init__` Method:

- **`self.tweets`:** This instance variable stores a list of text samples (tweets).

- **`self.labels`:** This instance variable stores corresponding labels or target values for each tweet.
- **`self.tokenizer`:** This is the BERT tokenizer, which is used to tokenize the text data.
- **`self.max_length`:** This variable specifies the maximum length to which the text will be tokenized. Texts that are longer than this value will be truncated, and shorter texts will be padded to match this length.

2. `__len__` Method:

This method returns the total number of examples in your dataset, which is the length of the `tweets` list. It is used by PyTorch's DataLoader to determine the number of batches to generate during training.

3. `__getitem__` Method:

This method is used to retrieve a specific example from the dataset given its index `idx`.

- **`tweet = self.tweets[idx]`** retrieves the text (tweet) at the specified index.
- **`label = self.labels[idx]`** retrieves the label or target associated with the tweet at the specified index.
- **`self.tokenizer.encode_plus(...)`:** This line tokenizes the text (tweet) using the BERT tokenizer. It performs the following tasks:
- **`add_special_tokens=True`:** It adds special tokens like [CLS] and [SEP] to the tokenized sequence, which are required for BERT model input.
- **`max_length=self.max_length`:** It truncates or pads the tokenized sequence to the specified `max_length`.
- **`return_token_type_ids=False`:** This argument indicates that token type IDs are not returned. Token type IDs are used in BERT for distinguishing different segments of text, but in many classification tasks, you don't need them.
- **`padding='max_length'`:** It pads the tokenized sequence to reach the `max_length`.
- **`return_attention_mask=True`:** It returns an attention mask, which is used to indicate which tokens are actual content (1) and which are padding (0).
- **`return_tensors='pt'`:** It returns PyTorch tensors.

The method returns a dictionary containing the following keys:

- **'tweet':** The original tweet text.
- **'input_ids':** The tokenized and padded input as a flattened tensor.
- **'attention_mask':** The attention mask as a flattened tensor.
- **'targets':** The label, converted to a PyTorch tensor with a data type of `torch.long`.

CODE AND OUTPUT:

```
MAX_LENGTH = 64
TEST_SIZE = 0.1
VALID_SIZE = 0.5
BATCH_SIZE = 16
NUM_WORKERS = 2

train_sampler, test_sampler = train_test_split(df, test_size=TEST_SIZE, random_state=RANDOM_STATE)
valid_sampler, test_sampler = train_test_split(test_sampler, test_size=VALID_SIZE, random_state=RANDOM_STATE)

train_set = CustomDataset(
    train_sampler['text'].to_numpy(),
    train_sampler['labels'].to_numpy(),
    tokenizer,
    MAX_LENGTH
)
test_set = CustomDataset(
    test_sampler['text'].to_numpy(),
    test_sampler['labels'].to_numpy(),
    tokenizer,
    MAX_LENGTH
)
valid_set = CustomDataset(
    valid_sampler['text'].to_numpy(),
    valid_sampler['labels'].to_numpy(),
    tokenizer,
    MAX_LENGTH
)
```

```
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, num_workers=NUM_WORKERS)
test_loader = DataLoader(test_set, batch_size=BATCH_SIZE, num_workers=NUM_WORKERS)
valid_loader = DataLoader(valid_set, batch_size=BATCH_SIZE, num_workers=NUM_WORKERS)
```

EXPLANATION:

1. `MAX_LENGTH = 64`: This constant defines the maximum length to which the text will be tokenized. Texts longer than this will be truncated, and shorter texts will be padded.

2. `TEST_SIZE = 0.1`: This constant specifies the size of the test set as a fraction of the total dataset. In this case, it's set to 10% of the data.

3. `VALID_SIZE = 0.5`: This constant specifies the size of the validation set as a fraction of the test set. It's set to 50% of the test set. Typically, the validation set is used to tune hyperparameters and monitor model performance during training.

4. `BATCH_SIZE = 16`: This constant defines the batch size, which is the number of data samples that will be processed together in each iteration during training. A batch size of 16 means that the model will see 16 data samples at a time.

5. `NUM_WORKERS = 2`: This constant specifies the number of worker processes to use for data loading. It can speed up data loading by parallelizing the process.

6. `train_sampler, test_sampler = train_test_split(df, test_size=TEST_SIZE, random_state=RANDOM_STATE)`:

This line uses the `train_test_split` function to split the original dataset `df` into training and testing sets. The `TEST_SIZE` constant controls the size of the testing set, and `RANDOM_STATE` is assumed to be defined elsewhere for reproducibility.

7. `valid_sampler, test_sampler = train_test_split(test_sampler, test_size=VALID_SIZE, random_state=RANDOM_STATE)`:

This line further splits the testing set into a validation set and a separate testing set. The `VALID_SIZE` constant controls the size of the validation set.

8. `train_set`, `test_set`, and **`valid_set`** are instances of the `CustomDataset` class. Each dataset is initialized with the appropriate text and labels from the respective split, and the BERT tokenizer is used to tokenize the text data with a maximum length of `MAX_LENGTH`.

9. `train_loader`, `test_loader`, and **`valid_loader`** are instances of PyTorch's `DataLoader` class. They are used to load data in batches for training, testing, and validation, respectively. The `batch_size` is set to `BATCH_SIZE`, and `num_workers` controls the number of worker processes used for data loading.

CODE AND OUTPUT:

```
0s
from torch import nn
class AirlineSentimentClassifier(nn.Module):

    def __init__(self, num_labels):
        super (AirlineSentimentClassifier, self).__init__()
        self.bert = BertModel.from_pretrained(MODEL)
        self.dropout = nn.Dropout(p=0.2)
        self.classifier = nn.Linear(self.bert.config.hidden_size, num_labels)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(
            input_ids=input_ids,
            attention_mask=attention_mask
        )
        pooled_output = outputs[1]
        pooled_output = self.dropout(pooled_output)
        out = self.classifier(pooled_output)
        return out
```

EXPLANATION:

1. `def __init__(self, num_labels)`: This is the constructor method for the `AirlineSentimentClassifier` class. It initializes the model's architecture.

`num_labels`: This parameter specifies the number of sentiment labels or classes that the model will predict.

INSIDE THE CONSTRUCTOR:

- **`super(AirlineSentimentClassifier, self).__init__()`:** This line calls the constructor of the parent class (`nn.Module`) to initialize the model.
- **`self.bert = BertModel.from_pretrained(MODEL)`:** This line initializes the BERT model for text encoding. The BERT model is loaded from the pre-trained checkpoint specified by the `MODEL` variable. This model will be used to extract contextual representations of the input text.
- **`self.dropout = nn.Dropout(p=0.2)`:** This line adds a dropout layer with a dropout probability of 0.2. Dropout is a regularization technique that helps prevent overfitting by randomly "dropping out" (setting to zero) a fraction of the neurons during training.
- **`self.classifier = nn.Linear(self.bert.config.hidden_size, num_labels)`:** This line defines a linear (fully connected) layer that takes the hidden state size of the BERT model as input and produces predictions for the sentiment labels. The number of output units in this layer is determined by `num_labels`.

2. `def forward(self, input_ids, attention_mask)`: This method defines the forward pass of the model, which describes how data flows through the network during inference or training.

- **`input_ids`:** These are the tokenized input IDs (indices) of the text data.

- **`attention_mask`:** This is the attention mask that indicates which tokens in the input are real (1) and which are padding (0).

INSIDE THE `FORWARD` METHOD:

- **`outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)`:** This line passes the `input_ids` and `attention_mask` to the BERT model. The BERT model processes the input text and returns a tuple of outputs.
- **`pooled_output = outputs[1]`:** This line extracts the "pooled" output from the BERT model, which is typically used as a representation of the entire input sequence.
- **`pooled_output = self.dropout(pooled_output)`:** The pooled output is passed through the dropout layer to apply regularization.
- **`out = self.classifier(pooled_output)`:** The pooled and dropout-processed output is then passed through the linear classifier to make predictions for the sentiment labels. The result, `out`, contains the model's predictions.

CODE AND OUTPUT:

```
[19] model = AirlineSentimentClassifier(len(labels_map))
     print(model)

     # Move tensors to GPU on CUDA enables devices
     if device:
         model.cuda()
```

```
Downloading model.safetensors: 100% 440M/440M [00:04<00:00, 33.2MB/s]
AirlineSentimentClassifier(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.2, inplace=False)
  (classifier): Linear(in_features=768, out_features=3, bias=True)
)
```

EXPLANATION:

1. `model = AirlineSentimentClassifier(len(labels_map))`: In this line, you create an instance of the `AirlineSentimentClassifier` model. `len(labels_map)` is used to determine the number of output units in the classifier, which is typically equal to the number of sentiment labels or classes. This line essentially initializes the model with random weights.

2. `print(model)`: This line prints the model to the console, which provides a textual representation of the model's architecture. You can inspect the structure of the model and its parameters by printing it.

3. `if device:`: This line checks whether a specific device (e.g., a GPU) is available for model training. The `device` variable should be set earlier in your code, and its value is typically determined based on the availability of GPU resources.

4. `model.cuda()`: If a GPU is available (as indicated by `device` being non-empty), this line moves the model to the GPU. In PyTorch, this is done by calling the `.cuda()` method on the model. This means that all the model's parameters and computations will be performed on the GPU, which can significantly accelerate training and inference for deep learning models, especially for large models like BERT.

CODE AND OUTPUT:

```
n_epochs = 10
learning_rate =2e-5

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimizer
optimizer = AdamW(model.parameters(), lr=learning_rate, correct_bias=False)

# Define scheduler
training_steps = len(train_loader)*n_epochs
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0,
    num_training_steps=training_steps
)
```

```
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:411: FutureWarning: This implementation of AdamW is de
  warnings.warn(
```

EXPLANATION:

1. `n_epochs = 10`: This variable specifies the number of training epochs. An epoch is a complete pass through the entire training dataset. In this case, the model will be trained for 10 epochs, meaning it will see and learn from the entire training dataset 10 times.

2. `learning_rate = 2e-5`: This variable sets the learning rate, which determines the step size during the optimization process. A learning rate of `2e-5` is a common choice for fine-tuning BERT-like models in NLP tasks.

3. `criterion = nn.CrossEntropyLoss()`: This line defines the loss function for training the model. `nn.CrossEntropyLoss` is commonly used for classification tasks. It computes the cross-entropy loss, which is a measure of the dissimilarity between the predicted class probabilities and the true class labels.

4. `optimizer = AdamW(model.parameters(), lr=learning_rate, correct_bias=False)`: This line initializes the optimizer for the model's parameters. In this case, it uses the AdamW optimizer. The parameters to the optimizer are as follows:

`model.parameters()`: It specifies the model's parameters that need to be updated during training.

`lr=learning_rate`: It sets the learning rate for the optimizer, as determined by the `learning_rate` variable.

`correct_bias=False`: This option indicates that the optimizer should not correct the bias terms, which is a common practice when using the AdamW optimizer with BERT-like models.

5. `training_steps = len(train_loader) * n_epochs`: This line calculates the total number of training steps that will be executed over all epochs. This is used to determine the schedule for the learning rate during training.

6. `scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=training_steps)`: This line sets up a learning rate scheduler. The scheduler is used to adjust the learning rate during training, typically starting with a warm-up phase.

- **`optimizer`:** This is the optimizer to which the scheduler will be applied.
- **`num_warmup_steps=0`:** It specifies the number of steps used for learning rate warm-up. In this case, there is no warm-up (0 steps).
- **`num_training_steps=training_steps`:** It specifies the total number of training steps, which is calculated based on the number of epochs and the size of the training data.

Training the model:

Training a BERT model typically involves fine-tuning a pre-trained BERT model on a specific NLP (Natural Language Processing) task, such as sentiment analysis, text classification, named entity recognition, or question answering.

- Implement a training loop that iterates over the training dataset in mini-batches.
- Compute the loss for each batch and perform back propagation to update the model's weights.
- Monitor training and validation loss during the training process.
-

CODE AND OUTPUT:

```
# Track changes in validation loss
valid_loss_min = np.Inf

for epoch in range(1, n_epochs+1):

    # Setting training and validation loss
    train_loss = []
    validation_loss = []
    tr_predictions = 0
    acc = 0
    val_predictions = 0

    ####################
    #  Train the model #
    ####################
    model = model.train()
    for data in train_loader:

        # Moving tensors to GPU on CUDA enabled devices
        if device:
            input_ids, attention_mask, targets = data["input_ids"].cuda(), data["attention_mask"].cuda(), data["targets
        # Clear the gradients of variables
        optimizer.zero_grad()
    #### Forward pass
    # Pass input through the model
    output = model(
        input_ids=input_ids,
        attention_mask=attention_mask
    )
    # Compute batch loss
    loss = criterion(output, targets)
    # Convert output probabilities to class probabilities
    _, pred = torch.max(output, 1)
    # Track correct predictions
    tr_predictions += torch.sum(pred == targets)

    #### Backward Pass
    # Compute gradients wrt to model parameters
    loss.backward()
    # To avoid exploding gradients, we clip the gradients of the model
    nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
    # Perform parameter update
    optimizer.step()
    # Update learning rate
    scheduler.step()
    # Update loss per mini batches
    train_loss.append(loss.item())

######################
# Validate the model #
######################
model.eval()
with torch.no_grad():
    for data in valid_loader:
```

```
# Moving tensors to GPU on CUDA enabled devices
if device:
    input_ids, attention_mask, targets = data["input_ids"].cuda(), data["attention_mask"].cuda(), data["tar

#### Forward pass
# Pass input through the model
output = model(
    input_ids=input_ids,
    attention_mask=attention_mask
)
# Compute batch loss
loss = criterion(output, targets)
# Convert output probabilities to class probabilities
_, pred = torch.max(output, 1)
# Update loss per mini batches
validation_loss.append(loss.item())
# Track correct predictions
val_predictions += torch.sum(pred == targets)
```

EXPLANATION:

1. `valid_loss_min = np.Inf` : The `valid_loss_min` variable is initialized with positive infinity (`np.Inf`). This variable is used to track the minimum validation loss observed during training.

2. The code enters a loop that iterates for each epoch within the range **`[1, n_epochs]`**, where `n_epochs` is the total number of training epochs.

3. `train_loss` and `validation_loss` lists are initialized to store the training and validation losses for each epoch.

4. `tr_predictions` and `val_predictions` variables are initialized to keep track of the number of correct predictions in the training and validation sets.

5. The model enters the training phase with **`model = model.train()`.** In this phase, the model's parameters are updated based on the training data.

6. The loop then iterates through the training data using the `train_loader`:

- Input data (input_ids, attention_mask) and target labels (targets) are loaded for each batch.
- If a GPU (CUDA) is available (`if device`), the input data and targets are moved to the GPU for processing.
- The gradients are cleared (`optimizer.zero_grad()`) to start fresh with each batch.
- A forward pass through the model is performed to obtain predictions and calculate the batch loss.
- The correct predictions are tracked and updated.
- Backpropagation is performed to compute gradients with respect to the model parameters.
- Gradient clipping is applied to prevent exploding gradients.
- Model parameters are updated based on computed gradients using the optimizer.
- The learning rate is updated according to the learning rate schedule defined by the `scheduler`.
- The batch loss is recorded in the `train_loss` list.

7. The model is evaluated on the validation **set (`model.eval()`),** and the same process is repeated as in the training loop, but without backpropagation (gradients are not updated).

8. `val_predictions` tracks correct predictions during validation.

CODE :

```
    # Compute accuracy
    train_accuracy = tr_predictions.double()/len(train_sampler)
    val_accuracy = val_predictions.double()/len(valid_sampler)

    # Print loss statistics
    print('Epoch: {}/{} \n\tTraining Loss: {:.6f} \n\tValidation Loss: {:.6f} \n\tTrain Accuracy: {:.6f} \n\tVal Accura

    # Save model if validation loss is decreased
    if val_accuracy > acc:
        print('Saving model...')
        torch.save(model.state_dict(), 'bert_base_fine_tuned.pt')
        acc = val_accuracy
```

OUTPUT:

```
Epoch: 1/10
        Training Loss: 0.478485
        Validation Loss: 0.426510
        Train Accuracy: 0.813221
        Val Accuracy: 0.848361
Saving model...
Epoch: 2/10
        Training Loss: 0.251598
        Validation Loss: 0.587404
        Train Accuracy: 0.912720
        Val Accuracy: 0.837432
Saving model...
Epoch: 3/10
        Training Loss: 0.147462
        Validation Loss: 0.694001
        Train Accuracy: 0.958333
        Val Accuracy: 0.848361
Saving model...
Epoch: 4/10
        Training Loss: 0.095958
        Validation Loss: 0.852052
        Train Accuracy: 0.976548
        Val Accuracy: 0.841530
Saving model...
Epoch: 5/10
        Training Loss: 0.062927
        Validation Loss: 0.967488
        Train Accuracy: 0.985504
        Val Accuracy: 0.842896
Saving model...
Epoch: 6/10
        Training Loss: 0.042360
        Validation Loss: 1.066000
        Train Accuracy: 0.990437
        Val Accuracy: 0.840164
Saving model...
Epoch: 7/10
        Training Loss: 0.032142
        Validation Loss: 1.132496
        Train Accuracy: 0.992410
        Val Accuracy: 0.833333
Saving model...
Epoch: 8/10
        Training Loss: 0.024429
        Validation Loss: 1.184951
        Train Accuracy: 0.993777
        Val Accuracy: 0.829235
Saving model...
Epoch: 9/10
        Training Loss: 0.018996
        Validation Loss: 1.230268
        Train Accuracy: 0.994991
        Val Accuracy: 0.831967
Saving model...
Epoch: 10/10
        Training Loss: 0.015075
        Validation Loss: 1.244014
        Train Accuracy: 0.995826
        Val Accuracy: 0.830601
Saving model...
```

EXPLANATION:

1**. `train_accuracy = tr_predictions.double() / len(train_sampler)` and `val_accuracy = val_predictions.double() / len(valid_sampler)`:** These lines calculate the training and validation accuracy. It appears that `tr_predictions` and `val_predictions` represent the number of correct predictions on the training and validation sets, and `len(train_sampler)` and `len(valid_sampler)` represent the total number of samples in the respective datasets. The division results in the ratio of correct predictions to the total number of samples, effectively giving the accuracy.

2. `print(...)`:

This line prints various training statistics, including:

- The current epoch.
- The training loss, which is calculated as the mean of all training losses within the epoch.
- The validation loss, which is calculated as the mean of all validation losses within the epoch.
- The training accuracy, which represents the ratio of correct predictions to the total number of training samples.
- The validation accuracy, which represents the ratio of correct predictions to the total number of validation samples.

3. **`if val_accuracy > acc:`**

- ✓ This conditional statement checks if the validation accuracy (`val_accuracy`) is greater than a previously defined accuracy threshold (`acc`).
- ✓ If the validation accuracy has improved (increased), the code enters the if block, indicating that the model has performed better on the validation set than in previous epochs.
- ✓ In this case, there is a model checkpointing mechanism that saves the model's state (weights and parameters) to a file named 'bert_base_fine_tuned.pt'. This is done to save the best-performing model so far.
- ✓ The `acc` variable is updated to the new validation accuracy, which acts as a threshold for future comparisons.

Testing the fine-tuned network:

After training, evaluate the model's performance on the test dataset to assess its generalization to unseen data.

Although the training accuracy reached over 99%, let's define some metrics and see how the model performs on unseen data.

CODE AND OUTPUT:

```
# Track test loss
test_loss = 0.0
class_predictions = list(0. for i in range(3))
class_total = list(0. for i in range(3))
predictions = []
labels = []

model.eval()
with torch.no_grad():
    for data in test_loader:

        # Moving tensors to GPU on CUDA enabled devices
        if device:
            input_ids, attention_mask, targets = data["input_ids"].cuda(), data["attention_mask"].cuda(), data["targets

        #### Forward pass
        # Pass input through the model
        output = model(
            input_ids=input_ids,
            attention_mask=attention_mask
        )
        # Compute batch loss
        loss = criterion(output, targets)
        # Update loss
        test_loss += loss.item()
        # convert output probabilities to predicted class
        _, pred = torch.max(output, 1)

        predictions.extend(pred)
        labels.extend(targets)

predictions = torch.stack(predictions) if not device else torch.stack(predictions).cpu()
labels = torch.stack(labels) if not device else torch.stack(labels).cpu()
```

EXPLANATION:

1. `test_loss = 0.0`: Initialize the variable `test_loss` to keep track of the cumulative test loss. This loss will be updated during the evaluation process.

2. `class_predictions` and `class_total`: These variables are initialized to keep track of class-wise predictions and total counts. The code appears to be set up for a multi-class classification problem with three classes.

3. `predictions` and **`labels`:** These lists are initialized to store the model's predictions and the true labels from the test dataset.

4. `model.eval()`: Puts the model in evaluation mode, which disables operations like dropout that are active during training but not during evaluation.

5. `with torch.no_grad()`: Inside this block, operations are performed without gradient tracking. This speeds up the evaluation process and saves memory.

6. `for data in test_loader`: This loop iterates over the test data in batches, which are loaded from the `test_loader`. Each batch contains input data (e.g., tokenized text, attention masks) and the corresponding target labels.

7. `if device`: This conditional checks whether a CUDA-enabled device (GPU) is available. If a GPU is available, it moves the input data and targets to the GPU using the `.cuda()` method.

8. Forward Pass:

`output = model(...)`: This line performs the forward pass of the model. It takes the input data, which includes tokenized text and attention masks, and computes predictions using the trained sentiment analysis model.

`loss = criterion(output, targets)`: Compute the loss between the model's predictions (`output`) and the true labels (`targets`) using the defined loss function (`criterion`). This loss is added to the `test_loss` variable, accumulating the loss for the entire test dataset.

9. Predictions and Labels:

`_, pred = torch.max(output, 1)`: This line extracts the predicted class labels from the model's output. It finds the class with the highest predicted probability as the model's prediction.

`predictions.extend(pred)` and **`labels.extend(targets)`:** Append the predictions and true labels to their respective lists.

10. `predictions` and `labels` are converted to PyTorch tensors after the loop. If the device is a GPU, the tensors are moved back to the CPU using `.cpu()`.

CODE AND OUTPUT:

```
print(classification_report(predictions, labels, target_names=['neutral', 'positive','negative']))
```

```
              precision    recall  f1-score   support

     neutral       0.68      0.71      0.69       146
    positive       0.79      0.81      0.80       118
    negative       0.92      0.90      0.91       468

    accuracy                           0.85       732
   macro avg       0.79      0.81      0.80       732
weighted avg       0.85      0.85      0.85       732
```

EXPLANATION:

1. Function Signature:

classification_report(y_true, y_pred, target_names=['class1', 'class2', ...])

- **`y_true`:** This is the true (actual) labels for the test data.
- **`y_pred`:** This is the predicted labels generated by your model.
- **`target_names`:** A list of labels for each class in your classification problem. In your case, the classes are 'neutral,' 'positive,' and 'negative.

2. Computation of Metrics:

For each class (e.g., 'neutral,' 'positive,' 'negative'), the `classification_report` function computes and reports several metrics, including:

- ✓ **Precision:** The proportion of true positive predictions among all instances predicted as that class.
- ✓ **Recall:** The proportion of true positive predictions among all instances that belong to that class.
- ✓ **F1-score:** The harmonic mean of precision and recall, which provides a balance between the two.

- ✓ **Support:** The number of occurrences of each class in the true labels.

3. Summary Report:

The `classification_report` function generates a summary report that includes these metrics for each class and an additional row labeled 'accuracy,' which shows the overall accuracy of the model across all classes.

4. Target Names:

The `target_names` parameter allows you to specify the names or labels for each class. In your case, 'neutral,' 'positive,' and 'negative' are the labels used to represent the classes.

5. Interpretation:

The generated report is useful for understanding how well your model performs for each sentiment class. For example, it will show if the model is better at predicting 'positive' sentiment compared to 'negative' sentiment.

You can compare precision and recall values to assess the trade-off between correctly identifying a class (precision) and not missing instances of that class (recall).

The F1-score combines both precision and recall into a single metric, providing a balanced evaluation.

The support values indicate the number of samples for each class in your test dataset.

Visualizations

Let's visualize our model performance through plotting a confusion matrix.

Metrics for positive tweets and neutral tweets are similar.

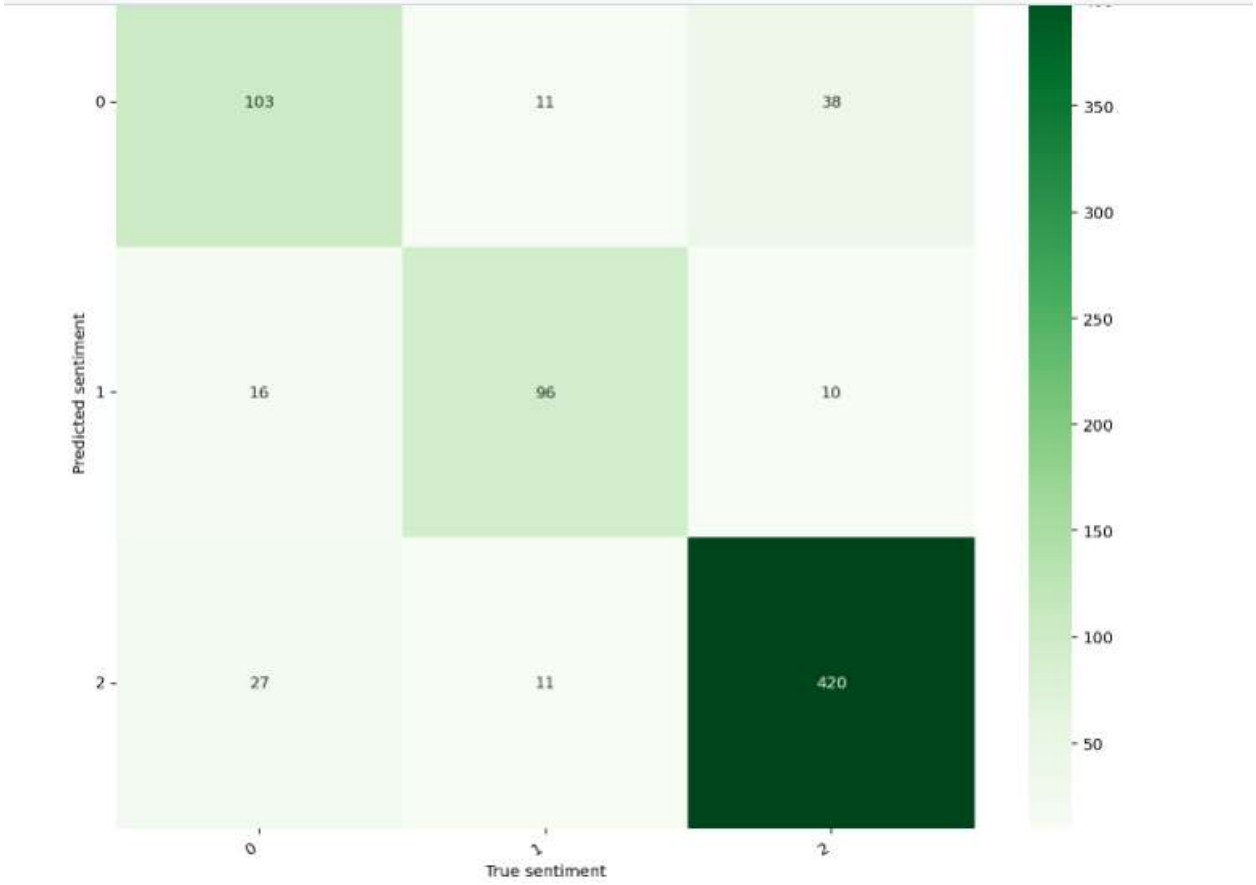
It is also seen that, the model performs well in the case of negative tweets.

We can conclude that it does an average job, also we cannot go entirely by accuracy when it comes to class imbalance.

CODE :

```
cm = confusion_matrix(labels, predictions)
heatmap = sns.heatmap(cm, annot=True, fmt='d', cmap='Greens')
heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right')
heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=30, ha='right')
plt.xlabel('True sentiment')
plt.ylabel('Predicted sentiment');
```

OUTPUT:



EXPLANATION:

1. `cm = confusion_matrix(labels, predictions)`: This line computes the confusion matrix by comparing the true labels (`labels`) and the predicted labels (`predictions`) from your sentiment analysis model.

The rows of the confusion matrix represent the true (actual) classes, while the columns represent the predicted classes.

Each cell in the matrix contains the count of instances that fall into a particular category, indicating how many samples were correctly classified (the diagonal) and how many were misclassified.

2. `heatmap = sns.heatmap(cm, annot=True, fmt='d', cmap='Greens')`: This line creates a heatmap visualization of the confusion matrix using the Seaborn library.

- **`cm`:** The confusion matrix computed earlier is provided as the input data for the heatmap.
- **`annot=True`:** This parameter is set to `True` to display the count values in each cell of the heatmap.
- **`fmt='d'`:** This parameter specifies that the counts should be displayed as integers.

- **`cmap='Greens'`:** The colormap used for coloring the cells of the heatmap. In this case, it uses green shades to represent the counts.

3. `heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right')`: This line sets the tick labels for the y-axis (the true sentiment labels). It specifies that the labels should not be rotated and should be aligned to the right.

4. `heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=30, ha='right')`: This line sets the tick labels for the x-axis (the predicted sentiment labels). It specifies that the labels should be rotated by 30 degrees and aligned to the right.

5. `plt.xlabel('True sentiment')` and **`plt.ylabel('Predicted sentiment')`:** These lines add labels to the x-axis and y-axis, indicating what the axis represents. In this case, the x-axis represents the predicted sentiment, and the y-axis represents the true sentiment.