Sentiment Analysis using BERT

(zainalh2@illinois.edu)

1. Introduction

This document illustrates the use of the BERT pre-trained model to classify the sentiment of Twitter tweets.

The target audience of the document:

1. The audience has ***no experience in BERT pre-training Model***.
2. The audience has the curiosity on how to use and fine-tune the BERT model for simple training tasks such as text classification (categorization).
3. The audience has some basic understanding of Data Mining and Machine Learning.
    - Understand the concept of Text Mining.
    - Understand the concept of Training and Validation datasets
    - Understand the concept of F1-Score evaluation
4. The audience has a basic knowledge of Python 3, Porch usage, and Jupyter Notebook.
5. And of course, the audience who is looking forward to spending fun time with Python code!

The main sources of inspiration for this document are the following Internet sources:

1. Cousera course (link)
2. An article about BERT fine-tuning (link)

This document combines BERT implementation and concepts from both sources into a single document. I hope this document will be useful for everyone new to the BERT world!

2. What is BERT and Text Sentiment Analysis?

2.1. What is BERT?

BERT is ***Bidirectional Encoder Representations from Transformer*** is a Natural Processing Language (NLP) pre-training model developed by Google (link).

This means BERT provides a base model that is already trained using a large number of a text corpus. Luckily Google folks had already done this for us, and they shared the model with the world for us to use for free! Hundreds of GPU hours needed to train the original base BERT model.

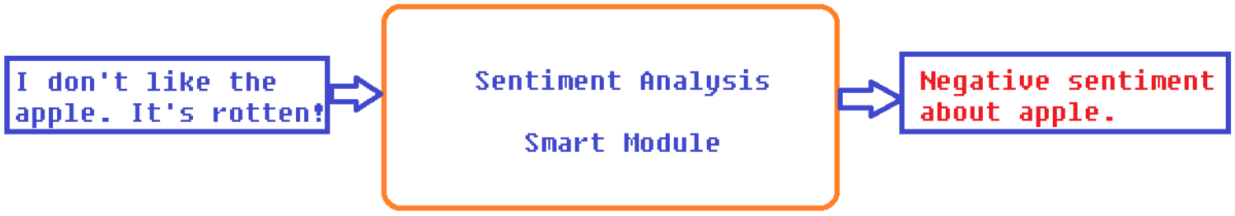
Useful Internet sources to understand BERT:

1. BERT for dummies (link)
2. The Illustrated BERT (link)
3. The BERT original paper (link).

2.2. What is Text Sentiment analysis?

Text Sentiment Analysis is a study in Text Mining to systematically identify, extract, quantify, and study subjective information in unstructured text data (link).

Imagine we have a "smart" module in which we can feed text data as an input. The "smart" module, implemented as a software code, will be able to output the sentiment found in the text data, as illustrated by the diagram below:



In the illustration above, we pass a text data **"*I don't like the apple. It's rotten!*"**. We as a human can easily identify that the opinion holder expressed his/her negative sentiment about an apple. To us, the reason is so obvious that the apple is rotten. But for a computer to understand the sentiment, it requires a complex computation model. We can use BERT to model the computation of the "smart" module to detect sentiment found in text data.

3. Case Study

This document uses a case study implemented in the Software code section to illustrate BERT capability to classify the sentiment found in text data.

The dataset is labeled Twitter tweet. Each label expresses the sentiment of the tweet. In software code, we will split the dataset into training and validation datasets. The training dataset will be used to train the base BERT model. Using the trained BERT model, we will use the validation dataset to evaluate the accuracy of the trained BERT model.

3.1. Dataset

The case study uses a dataset that we can freely download from the Internet.

3.1.1. Dataset Source

Dataset is taken from the source:

1. Wang, Bo; Tsakalidis, Adam; Liakata, Maria; Zubiaga, Arkaitz; Procter, Rob; Jensen, Eric (2016): SMILE Twitter Emotion dataset. figshare. Dataset. (link)

3.1.2. Text structure in the dataset

Before we use the dataset using BERT, we need to analyze and describe the data format found in the dataset.

Dataset has 3085 lines (tweets). Each line/record consists of 3 columns:

- Column-1: The unique ID of the line/tweet
- Column-2: The tweet message
- Column-3: The sentiment label of the message (sad, happy, etc.)

Consider the sample line/record from the dataset below:

611537640857411584,"@britishmuseum @SenderosP The Rosetta Stone ;)",happy

The value of each column from the sample line/record will be:

- Column-1 (The Unique ID): ***611537640857411584***
- Column-2 (The tweet message): ***@britishmuseum @SenderosP The Rosetta Stone ;)***
- Column-3 (The sentiment label): ***happy***

3.2. Software Code

Software code is implemented using Google Colab Jupyter Notebook. The notebook copy is also available in my Github account (link).

Read the software code comment carefully to follow along with the detailed explanation of each step in the software code.

3.2.1. Google Colab Environment

This document uses the Google Colab environment to implement the software code. The introduction of this environment is available here (link). The first step is to make sure we utilize GPU capability freely available in this environment:

```
[1]  # Check if the GPU is available in the Colab environment
     # To activate GPU, in this Colab Notebook:
     # Go to Edit -> Notebook Settings
     # And make sure select GPU as Hardware Accelerator
     import tensorflow as tf

     device_name = tf.test.gpu_device_name()
     if device_name == '/device:GPU:0':
         print('Found GPU at: {}'.format(device_name))
     else:
         raise SystemError('GPU device not found')
```

```
Found GPU at: /device:GPU:0
```

```
[2]  # Since we are going to use PyTorch and GPU
     # We need to tell PyTorch to use GPU explicitly
     # (PyTorch uses CPU by default)
     import torch

     # If GPU is available.
     if torch.cuda.is_available():
         # PyTorch to use the GPU
         device = torch.device("cuda")
         print('There are %d GPU(s) available.' % torch.cuda.device_count())
         print('We will use the GPU:', torch.cuda.get_device_name(0))
     # If GPU is not available. Use the CPU.
     else:
         print('No GPU available, using the CPU instead.')
         device = torch.device("cpu")
```

```
There are 1 GPU(s) available.
We will use the GPU: Tesla T4
```

3.2.2. *Analyze the dataset using pandas DataFrame*

```
[3]  # Load data from dataset file to pandas DataFrame
     import pandas as pd  # https://pandas.pydata.org/
     # Pandas DataFrame columns: [id, text, category]. The column "id" is the index.
     # In this environment, the dataset file is available here: sample_data/smile-annotations-final.csv.
     # Adjust the file location according to your system environment to run the software code
     df = pd.read_csv('sample_data/smile-annotations-final.csv', names=['id', 'text', 'category'])
     df.set_index('id', inplace=True)
```

```
[4]  # Take a look at first 5 records
     df.head()
```

id	text	category
611857364396965889	@aandraous @britishmuseum @AndrewsAntonio Merc...	nocode
614484565059596288	Dorian Gray with Rainbow Scarf #LoveWins (from...	happy
614746522043973632	@SelectShowcase @Tate_StIves ... Replace with ...	happy
614877582664835073	@Sofabsports thank you for following me back. ...	happy
611932373039644672	@britishmuseum @TudorHistory What a beautiful ...	happy

```
[5]  # Number of record of each category
     df.category.value_counts()
```

```
nocode                1572
happy                 1137
not-relevant           214
angry                   57
surprise                35
sad                     32
happy|surprise          11
happy|sad                9
disgust|angry            7
disgust                  6
sad|disgust              2
sad|angry                2
sad|disgust|angry        1
Name: category, dtype: int64
```

```
[6]  # Filter out multiple label (the ones with | character) to the sake of simplicty.
     df = df[~df.category.str.contains('\|')]
     # Filter out 'nocode' category. 'nocode' does not represent particular sentiment.
     df = df[df.category != 'nocode']
     # Filter out 'disgust' since it has only 6 records. We need more record to train the model.
     df = df[df.category != 'disgust']
     # Record counting of each category after data filter.
     df.category.value_counts()
```

```
happy           1137
not-relevant     214
angry             57
surprise          35
sad               32
Name: category, dtype: int64
```

```
[7]  # Assign label to each category. Label value is integer.
     # Label will be used by the model to classify the text sentiment.
     possible_labels = df.category.unique()
     label_dict = {}
     for index, possible_label in enumerate(possible_labels):
         label_dict[possible_label] = index
     df['label'] = df.category.replace(label_dict)
     # Review the first 5 records after the new label column added.
     df.head()
```

id	text	category	label
614484565059596288	Dorian Gray with Rainbow Scarf #LoveWins (from...	happy	0
614746522043973632	@SelectShowcase @Tate_StIves ... Replace with ...	happy	0
614877582664835073	@Sofabsports thank you for following me back. ...	happy	0
611932373039644672	@britishmuseum @TudorHistory What a beautiful ...	happy	0
611570404268883969	@NationalGallery @ThePoldarkian I have always ...	happy	0

3.2.3. Split the dataset into a training dataset and validation dataset

Dataset is split with the following distribution:

- 80% as Training dataset: This is used to train the model.
- 20% as Validation dataset: This is used to evaluate the trained model.

```
[8]  # We will use sklearn library to split the dataset into training and validation dataset
     from sklearn.model_selection import train_test_split # https://scikit-learn.org/

     # Here we split the dataset. The 80% VS 20% data split is considered a fair split.
     # However we need to have a bigger data size for training compared to the testing dataset.
     # Thus the split for our case:
     #   1) 80% as the training dataset
     #   2) 20% as validation dataset
     # To choose which one goes to the training or validation dataset, it will be done randomly.
     # traing_test_split function will use random_state to randomly choose the dataset.
     # random_state set to 42. Popular integer random seeds are 0 and 42.
     X_train, X_val, y_train, y_val = train_test_split(df.index.values,
                                                       df.label.values,
                                                       test_size=0.20,
                                                       random_state=42,
                                                       stratify=df.label.values)
```

```
[9]  # Adding a new column called "data_type".
     # this is to label each text record either
     # it is "train"-ing dataset
     # or "val"-idation dataset
     df['data_type'] = ['not_set']*df.shape[0]
     df.loc[X_train, 'data_type'] = 'train'
     df.loc[X_val, 'data_type'] = 'val'

     # Verify the data distribution by category and data_type.
     # Here we should have an 80% vs 20% distribution for each category & data_type.
     df.groupby(['category', 'label', 'data_type']).count()
```

			text
category	**label**	**data_type**	
angry	**2**	**train**	45
		val	12
happy	**0**	**train**	910
		val	227
not-relevant	**1**	**train**	171
		val	43
sad	**3**	**train**	26
		val	6
surprise	**4**	**train**	28
		val	7

3.2.4. Tokenizing and Encoding

Tokenization in BERT is another interesting topic to explore. BERT uses the WordPiece tokenization strategy.

Internet sources to explore this topic further:

- Original paper (link)
- An article about BERT Token Embedding (link)

```
[10] # We will use the Huggingface library to instantiate transformers.
     # Reference about Huggingface transformers: https://github.com/huggingface/transformers
     # By default Google Colab does not have transformers installed.
     # The command below will install transformers
     !pip install transformers
```

[11]
```
from transformers import BertTokenizer # https://huggingface.co/transformers/model_doc/bert.html
from torch.utils.data import TensorDataset # https://pytorch.org/
import torch

# We create our BERT tokenizer.
# This will create WordPiece tokenizer which is required to format the input data
# input a format recognized by BERT model (more details below).
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)
```

```
Downloading: 100% [232k/232k [00:00<00:00, 3.15MB/s]
```

[12]
```
# Before we move on to the next, we should have one check done.
# This check will help us decide on how to encode our text data (details below).
df.text.str.len().max() # Longest Tweet message character length
```

```
149
```

[13]
```
# This part is all about preparing our data!
# We encode our data into a format that the BERT model can understand.
# It will add special tokens to the text data: [CLS] and [SEP]
# It will add a special [PAD] token after [SEP] in each text record.
# Why? Because BERT accepts fixed-length data with a maximum 512 token.
# In our case, max_length is set to 152 considering the maximum length of our text data is 149
# return_tensors='pt' means we are preparing our encoded data for PyTorch
encoded_data_train = tokenizer.batch_encode_plus(
    df[df.data_type=='train'].text.values,
    add_special_tokens=True,
    return_attention_mask=True,
    max_length = 152,
    padding='max_length',
    return_tensors='pt'
)

encoded_data_val = tokenizer.batch_encode_plus(
    df[df.data_type=='val'].text.values,
    add_special_tokens=True,
    return_attention_mask=True,
    max_length = 152,
    padding='max_length',
    return_tensors='pt'
)
```

```
# BERT id representation for each token
# input_ids_*

# Attention mask to differiantiate between token data and padding [PAD] token
# mask = 1 for token data
# mask = 0 for padding [PAD] token
# attention_masks_*

# create tensor for the category label
# labels_*

input_ids_train = encoded_data_train['input_ids']
attention_masks_train = encoded_data_train['attention_mask']
labels_train = torch.tensor(df[df.data_type=='train'].label.values)

input_ids_val = encoded_data_val['input_ids']
attention_masks_val = encoded_data_val['attention_mask']
labels_val = torch.tensor(df[df.data_type=='val'].label.values)
```

[14]
```
# Let's take a look at our text data in a BERT encoded representation!
# This is how the BERT model "sees" our text data.
encoded_data_train
```

```
{'input_ids': tensor([[  101, 16092,  3897,  ...,     0,     0,     0],
        [  101,  1030, 10682,  ...,     0,     0,     0],
        [  101,  1030,  2329,  ...,     0,     0,     0],
        ...,
        [  101,  1523,  1030,  ...,     0,     0,     0],
        [  101,  1030,  3680,  ...,     0,     0,     0],
        [  101,  1030,  2120,  ...,     0,     0,     0]]), 'token_type_ids': tensor([[0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        ...,
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0],
        [0, 0, 0,  ..., 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        ...,
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]])}
```

[15]
```
# Create the Tensor dataset. We will pass this tensor datasets to data loader.
# For more details about data loader, carry on to the next sections.
dataset_train = TensorDataset(input_ids_train, attention_masks_train, labels_train)
dataset_val = TensorDataset(input_ids_val, attention_masks_val, labels_val)
```

3.2.5. Setting Pre-Trained BERT Model

```
[16] # Huggingface BERT Models available: https://huggingface.co/models
     from transformers import BertForSequenceClassification

     # Here, we will use a pre-trained BERT model BertForSequenceClassification
     # BertForSequenceClassification is a BERT Base model with different top layers
     # and output types designed to accommodate specific NLP tasks such as classification tasks.
     # More detail about BertForSequenceClassification: https://huggingface.co/transformers/model_doc/bert.html#bertforsequenceclassification
     model = BertForSequenceClassification.from_pretrained("bert-base-uncased",
                                                           num_labels=len(label_dict),
                                                           output_attentions=False,
                                                           output_hidden_states=False)

     # Device is set to GPU
     model.to(device)
```

3.2.6. Creating Data Loaders

Data loader is responsible to pass a batch of data (TensorDataset) to the model to process.

```
[17] from torch.utils.data import DataLoader, RandomSampler, SequentialSampler

     # Number of text records pass to the model at the same batch
     # batch_size=32 is recommended in the BERT original paper
     batch_size = 32

     # Create data loader for training TensorDataset
     dataloader_train = DataLoader(dataset_train, sampler=RandomSampler(dataset_train), batch_size=batch_size)
     # Create data loader for validation TensorDataset
     dataloader_validation = DataLoader(dataset_val, sampler=SequentialSampler(dataset_val), batch_size=batch_size)
```

3.2.7. Setting Up Optimizer and Scheduler

```
[18] from transformers import AdamW, get_linear_schedule_with_warmup

     # Optimizer in a nutshell is the algorithm to speed up the model learning process (training cost reduction at higher speed).
     # For more detail about optimizer: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

     # For fine-tuning, it is recommended choosing from the following values:
     #   1. Learning rate (Adam): 5e-5, 3e-5, 2e-5
     #   2. The epsilon parameter eps = 1e-8 is a very small number to prevent any division by zero in the implementation.
     # This is the recommendation from BERT's original paper.

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

```
[19] # Recommended to have 2, 3 or 4 EPOCHS for fine-tuning BERT on a specific NLP task.
     # This is the recommendation from BERT original paper.
     epochs = 4

     # Total number of training steps is [number of batches] x [number of epochs].
     # Create the learning rate scheduler. Reference: https://huggingface.co/transformers/main_classes/optimizer_schedules.html
     scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=len(dataloader_train)*epochs)
```

3.2.8. Defining Performance Metrics

```
[20] import numpy as np
     from sklearn.metrics import f1_score

     # Function to calculate F1 score: https://en.wikipedia.org/wiki/F-score
     # F1 score is harmonic mean of the precision and recall
     def calculate_f1_score(preds, labels):
         preds_flat = np.argmax(preds, axis=1).flatten()
         labels_flat = labels.flatten()
         return f1_score(labels_flat, preds_flat, average='weighted')

     # Function to calculate accuracy per category.
     def accuracy_per_category(preds, labels):
         label_dict_inverse = {v: k for k, v in label_dict.items()}

         preds_flat = np.argmax(preds, axis=1).flatten()
         labels_flat = labels.flatten()

         for label in np.unique(labels_flat):
             y_preds = preds_flat[labels_flat==label]
             y_true = labels_flat[labels_flat==label]
             print(f'Class: {label_dict_inverse[label]}')
             print(f'Accuracy: {len(y_preds[y_preds==label])}/{len(y_true)}\n')
```

3.2.9. Creating the Training Loop

This training and validation code is based on the run_glue.py script here (link)

[21]
```python
# Helper function to evaluate the validation result from the trained model
def evaluate(dataloader_val):
    # To set the model into a training mode
    model.eval()
    loss_val_total = 0
    predictions, true_vals = [], []

    for batch in dataloader_val:
        batch = tuple(b.to(device) for b in batch)
        inputs = {'input_ids':      batch[0],
                  'attention_mask': batch[1],
                  'labels':         batch[2],
                 }

        with torch.no_grad():
            # evaluate the validation dataset
            outputs = model(**inputs)

        loss = outputs[0]
        logits = outputs[1]
        loss_val_total += loss.item()

        logits = logits.detach().cpu().numpy()
        label_ids = inputs['labels'].cpu().numpy()
        predictions.append(logits)
        true_vals.append(label_ids)

    loss_val_avg = loss_val_total/len(dataloader_val)

    predictions = np.concatenate(predictions, axis=0)
    true_vals = np.concatenate(true_vals, axis=0)

    return loss_val_avg, predictions, true_vals
```

[22]
```
#########################################################
# Here we will traing the model using training dataset #
#########################################################

from tqdm.notebook import tqdm # https://github.com/tqdm/tqdm
import random

# The random seed used to initialise the weights
# and select the order of the training data.
# Set the seed value all over the place to make this reproducible.
seed_val = 17
random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)

# loop over the full dataset for a number of epochs times.
for epoch in tqdm(range(epochs)):

    # To set the model into a training mode.
    model.train()

    # Measure the total training loss for each epoch.
    loss_train_total = 0
    # Progressbar to show the progress of the current epoch.
    progress_bar = tqdm(dataloader_train, desc='Epoch {:1d}'.format(epoch), leave=False, disable=False)
```

```
# Process each batch in the current epoch.
for batch in progress_bar:

    # Always clear any previously calculated gradients before performing a backward pass.
    # (source: https://stackoverflow.com/questions/48001598/why-do-we-need-to-call-zero-grad-in-pytorch)
    model.zero_grad()

    # Unpack current training batch.
    # batch contains three pytorch tensors:
    #   [0]: input ids
    #   [1]: attention masks
    #   [2]: labels
    batch = tuple(b.to(device) for b in batch)

    inputs = {'input_ids':      batch[0],
              'attention_mask': batch[1],
              'labels':         batch[2],
             }

    # This is the actual learning.
    outputs = model(**inputs)

    # Current training loss.
    loss = outputs[0]
    # Current total training loss.
    loss_train_total = loss_train_total + loss.item()

    # Perform a backward pass to calculate the gradients.
    loss.backward()
```

```
        # Clip the norm of the gradients to 1.0.
        # This is to help prevent the "exploding gradients" problem.
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # Update parameters and take a step using the computed gradient.
        # The optimizer dictates the "update rule"--how the parameters are
        # modified based on their gradients, the learning rate, etc.
        optimizer.step()

        # Update the learning rate.
        scheduler.step()


        progress_bar.set_postfix({'training_loss': '{:.3f}'.format(loss.item()/len(batch))})

    # Save the trained BERT model for the current epoch iteration
    torch.save(model.state_dict(), f'finetuned_BERT_epoch_{epoch}.model')

    loss_train_avg = loss_train_total/len(dataloader_train)
    val_loss, predictions, true_vals = evaluate(dataloader_validation)
    val_f1 = calculate_f1_score(predictions, true_vals)

    # Report the summary of epoch iteration
    tqdm.write(f'\nEpoch {epoch}')
    tqdm.write(f'Training loss: {loss_train_avg}')
    tqdm.write(f'Validation loss: {val_loss}')
    tqdm.write(f'F1 Score (Weighted): {val_f1}')
```


```
Epoch 0
Training loss: 1.0198001265525818
Validation loss: 0.7589993864297867
F1 Score (Weighted): 0.669251250081174

Epoch 1
Training loss: 0.7479538643682325
Validation loss: 0.6324919879436492
F1 Score (Weighted): 0.7580030616732636

Epoch 2
Training loss: 0.5936409484695744
Validation loss: 0.48897247537970545
F1 Score (Weighted): 0.7788180018210169

Epoch 3
Training loss: 0.47576660844119817
Validation loss: 0.47983667999505997
F1 Score (Weighted): 0.8029831536363403
```

3.2.10. Evaluate the trained model

```
[23] ######################################################################
     # Here we will evaluate the trained-model using validation dataset #
     ######################################################################

     # Validate all the trained BERT model (for each EPOCH)
     for _, epoch in enumerate(range(epochs)):
       tqdm.write(f'EPOCH: {epoch}')
       model.load_state_dict(torch.load('finetuned_BERT_epoch_{0}.model'.format(epoch), map_location=torch.device('cpu')))
       _, predictions, true_vals = evaluate(dataloader_validation)
       accuracy_per_category(predictions, true_vals)
       tqdm.write(f'######################################################################')
```

```
EPOCH: 0
Class: happy
Accuracy: 227/227

Class: not-relevant
Accuracy: 0/43

Class: angry
Accuracy: 0/12

Class: sad
Accuracy: 0/6

Class: surprise
Accuracy: 0/7

################################################################################
EPOCH: 1
Class: happy
Accuracy: 226/227

Class: not-relevant
Accuracy: 14/43

Class: angry
Accuracy: 0/12

Class: sad
Accuracy: 0/6

Class: surprise
Accuracy: 0/7

################################################################################
EPOCH: 2
Class: happy
Accuracy: 219/227

Class: not-relevant
Accuracy: 22/43

Class: angry
Accuracy: 0/12

Class: sad
Accuracy: 0/6

Class: surprise
Accuracy: 0/7
```

```
################################################################################
EPOCH: 3
Class: happy
Accuracy: 223/227

Class: not-relevant
Accuracy: 21/43

Class: angry
Accuracy: 3/12

Class: sad
Accuracy: 0/6

Class: surprise
Accuracy: 0/7

################################################################################
```

4. Summary

This document demonstrates that we can create a trained model using a pre-trained BERT model with minimal effort and training time using the PyTorch interface.

In our case study, we use the model to classify Twitter tweet's sentiment. The table below summarizes the accuracy of the trained BERT model to categorize the text data based on the provided training & evaluation dataset:

EPOCH	Happy	Non-Relevant	Angry	Sad	Surprise
EPOCH 1	227/227	0/43	0/12	0/6	0/7
EPCOH 2	226/227	14/43	0/12	0/6	0/7
EPOCH 3	219/227	22/43	0/12	0/6	0/7
EPOCH 4	223/227	21/43	3/12	0/6	0/7

With 4- EPOCH iterations, the model can classify the "happy", "non-relevant", and "Angry" sentiments with good accuracy (less accuracy for "Non-Relevant" and "Angry"). However, it fails to categorize the sentiment of "Sad" and "Surprise".

Let's review the number of training and evaluation records for each sentiment categories:

Category	Number of Training Records	Number of Evaluation Records
Happy	910	227
Non-Relevant	171	43
Angry	45	12
Sad	26	6
Surprise	28	7

Apart from BERT model fine-tuning process, looking at the provided number of training records to train the model, we can ***conclude*** that the model can learn with higher accuracy if the provided training dataset has more records.