PHASE 5

SENTIMENTAL ANALYSIS FOR MARKETING

MEMBERS

- SOUNDHAR BALAJI.B
- RADHIKA.M
- ROHANSHAJ.K.R
- VIGNESH.V
- NELSON JOSEPH M

Part I INTRODUCTION

In the course of our project, we embarked on a journey to dissect the complex landscape of public sentiment towards airlines on Twitter. By employing BERT NLP, a cutting-edge natural language processing model, we were able to achieve a nuanced understanding of how travelers perceive various airlines. This project holds paramount significance in the era of real-time communication and instant feedback. With Twitter as a platform for people to share their experiences and opinions, our sentiment analysis not only gauged positive, negative, or neutral reactions but also unearthed valuable insights into specific aspects of airline service that trigger particular emotions. The power of BERT NLP, with its contextual understanding, allowed us to achieve a level of accuracy and depth in sentiment analysis that traditional methods often struggle to attain.

The inclusion of visual outputs in our project takes the analysis to the next level by making the findings more accessible and engaging. The graphs, charts, and word clouds not only provide a quick overview of sentiment distribution but also highlight frequently mentioned terms, offering a qualitative layer to our quantitative analysis. This visual storytelling aids stakeholders, including airlines themselves, in comprehending the key trends and sentiments at a glance. Beyond mere sentiment labeling, we were able to reveal patterns over time, helping airlines to identify areas of improvement or positive practices they should continue. By bridging the gap between data science and data communication, our project serves as a comprehensive tool for decision-makers and analysts in the airline industry.

Looking ahead, this project offers a multitude of possibilities. It serves as a strong foundation for continued research and analysis, as airlines strive to better understand and respond to customer feedback. Real-time sentiment tracking could be an exciting next step, enabling airlines to react swiftly to emerging trends or issues. Moreover, the project's innovative approach to sentiment analysis using BERT NLP opens doors to explore other domains where nuanced understanding of public opinion is crucial. As data-driven decision-making becomes increasingly pivotal in today's business landscape, our sentiment analysis project showcases the value of leveraging advanced NLP and data visualization techniques to gain a deeper understanding of public sentiment and, in turn, improve customer experiences in a variety of industries.

I

In the realm of marketing, sentiment analysis plays a pivotal role in understanding how customers perceive products, brands, and services. It enables marketers to extract valuable insights, uncover trends, and make

II

In the realm of marketing, sentiment analysis plays a pivotal role in understanding how customers perceive products, brands, and services. It enables marketers to extract valuable insights, uncover trends, and make data-driven decisions that can impact marketing strategies, product development, and customer satisfaction.

III

In the realm of marketing, sentiment analysis plays a pivotal role in understanding how customers perceive products, brands, and services. It enables marketers to extract valuable insights, uncover trends, and make

PROBLEMSTATEMENT

Documentation

Clearly outline the problem statement, design thinking process, and the phases of development.
Describe the dataset used, data preprocessing steps, and sentiment analysis techniques.
Document any innovative techniques or approaches used during the development. Submission

Compile all the code files, including the data preprocessing, and sentiment analysis techniques.
Provide a well-structured README file that explains how to run the code and any dependencies.
Share the submission on platforms like GitHub or personal portfolio for others to access and review.

DATASETLINK

https://www.kaggle.com/datasets/crowdflower/twitter-airlinesentiment

NOTEBOOK LINK

click here

APPROACH USED

"In conclusion, we have successfully conducted sentiment analysis on Twitter airline data using BERT NLP, a state-of-the-art natural language processing model. Our project aimed to gauge public sentiment towards various airlines by analyzing tweets. Here are the key takeaways:

1. Data Collection: We collected a comprehensive dataset of tweets related to different airlines, representing a wide spectrum of opinions and experiences.

2. Data Preprocessing: We preprocessed the data, which involved tasks like cleaning, tokenization, and removing irrelevant information. This step was crucial in ensuring the quality of our analysis.

3. Sentiment Analysis: We leveraged BERT, a powerful NLP model, to perform sentiment analysis on the Twitter data. This allowed us to classify tweets into positive, negative, or neutral sentiments, providing valuable insights into public perception.

4. Visual Outputs: To enhance the understanding of the sentiment analysis results, we generated visual outputs. These visuals included graphs, charts, and word clouds to illustrate sentiment distribution, frequently mentioned terms, and trends over time.

5. Key Findings: Our analysis revealed important insights into how different airlines are perceived on Twitter. We were able to identify common pain points, positive experiences, and trends in sentiment.

6. Innovative Approach: By employing BERT NLP, we utilized cutting-edge technology to improve the accuracy of sentiment analysis, making our project stand out in terms of innovation and effectiveness.

7. Future Directions: This project can serve as a foundation for further research, allowing airlines to better understand customer sentiment and make data-driven improvements. Future work could involve real-time sentiment tracking or additional feature engineering.

“In summary, our sentiment analysis of Twitter airline data using BERT NLP, coupled with visual outputs, provided valuable insights into customer sentiment towards different airlines. The project's use of advanced NLP techniques and visualizations enhances its applicability in the field of customer feedback analysis and public opinion monitoring."

Installing necessary libraries

In [ ]:
```
pip install emoji
```

In [ ]:
```
!pip install transformers
```

In [ ]:
```
pip install torch
```

In [ ]:
```
!pip install transformers
```

In [ ]:
```
pip install nltk
```

In [ ]:
```
pip install pydot
```

In [ ]:
```
pip install graphviz
```

In [ ]:
```
pip install tensorflow
```

In [ ]:
```
pip install tensorflow==2.14.0
```

The below code imports essential libraries like NumPy, Pandas, and TensorFlow for data manipulation and deep learning.The code includes tools for text processing, such as regular expressions and stemming using NLTK. It prepares for text tokenization and padding for neural network input using TensorFlow and Keras. It imports PyTorch for deep learning capabilities and data loading utilities like data loaders.

In [ ]:
```
import numpy as np
import pandas as pd

import re
import emoji
import re

from nltk.stem import PorterStemmer
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.model_selection import train_test_split
pd.set_option('display.max_colwidth',200)
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
import matplotlib.pyplot as plt
import tensorflow as tf
import torch
# importing nn module
import torch.nn as nn
#library for progress bar
from tqdm import notebook
from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler


#library for computing class weights
from sklearn.utils.class_weight import compute_class_weight

from sklearn.metrics import classification_report


import time
import datetime
```

Checking if GPU is available.

In [ ]:
```
# Checking if GPU is available.
if torch.cuda.is_available():
  device=torch.device('cuda')
```

In [ ]:
```
print(device)
torch.cuda.get_device_name(0)
# Current GPU is Tesla T4
```

```
cuda
```

Out[85]:
```
'Tesla T4'
```

###READING DATASHEET

In [ ]:
```
data = pd.read_csv('Tweets.csv')
```

In [ ]:

In [ ]:
```
data.head()
```

Out[87]:

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	negativereason_confidence	airline	airline_sentiment_gold	name
0	570306133677760513	neutral	1.0000	NaN	NaN	Virgin America	NaN	cairdin
1	570301130888122368	positive	0.3486	NaN	0.0000	Virgin America	NaN	jnardino
2	570301083672813571	neutral	0.6837	NaN	NaN	Virgin America	NaN	yvonnalynn
3	570301031407624196	negative	1.0000	Bad Flight	0.7033	Virgin America	NaN	jnardino
4	570300817074462722	negative	1.0000	Can't Tell	1.0000	Virgin America	NaN	jnardino

```
data
```

Out[88]:

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	negativereason_confidence	airline	airline_sentiment_gold	
0	570306133677760513	neutral	1.0000	NaN	NaN	Virgin America	NaN NaN	
							NaN	y
1	570301130888122368	positive	0.3486	NaN	0.0000	Virgin America		
							NaN	
2	570301083672813571	neutral	0.6837	NaN	NaN	Virgin America		
							NaN	
							...	
3	570301031407624196	negative	1.0000	Bad Flight	0.7033	Virgin America		
							NaN	I
4	570300817074462722	negative	1.0000	Can't Tell	1.0000	Virgin America		
...	...	...	...	...	...	...		
							NaN	Ha
13504	569846356409339906	positive	1.0000	NaN	NaN	American		
							NaN	steph
13505	569846302663688192	negative	1.0000	Customer Service Issue	0.6834	American		
							NaN	SFald
13506	569846045892608001	negative	1.0000	Customer Service Issue	0.6414	American		
							NaN	Ha
13507	569846023553720321	negative	1.0000	Customer Service Issue	0.6681	American		
13508	569845438494457856	negative	1.0000	Cancelled Flight	1.0000	American		

In [ ]:

13509 rows × 15 columns

#PREPROCESSING

This code filters the DataFrame to keep only the rows where the "airline_sentiment_confidence" is 0.6 or higher, and it reindexes the DataFrame for consistency.

In [ ]:
```
confidence_threshold = 0.6

data = data.drop(data.query("airline_sentiment_confidence < @confidence_threshold").index, axis=0).reset_index(drop=True)
```

tweets_df will contain two columns: "text" and "airline_sentiment," with the text data from the "text" column and the corresponding sentiment labels from the

In [ ]:
```
tweets_df = pd.concat([data['text'], data['airline_sentiment']], axis=1)
tweets_df
```

Out[91]:

	text	airline_sentiment
0	@VirginAmerica What @dhepburn said.	neutral
1	@VirginAmerica I didn't today... Must mean I need to take another trip!	neutral
2	@VirginAmerica it's really aggressive to blast obnoxious "entertainment" in your guests' faces & they have little recourse	negative
3	@VirginAmerica and it's a really big bad thing about it	negative
4	@VirginAmerica seriously would pay $30 a flight for seats that didn't have this playing.\nit's really the only bad thing about flying VA	negative
...	...	...
13278	@AmericanAir thank you for doing the best you could to get me rebooked. Agent on phone & addtl resolution on DM was very much appreciated.	positive
13279	@AmericanAir no email no phone call no nothing. You've screwed with my flight and my family/Friends flights. You Cancelled Flighted reservations for	negative
13280	@AmericanAir If you care, could you have someone call me to explain what is going on.	negative
13281	Hey @AmericanAir why automated call me and then hang up at 4:45 am!?! And why can't I reschedule Cancelled Flighted flights via web!?! Come on!!!	negative
13282	@AmericanAir from a service rep but that hasn't happened	negative

13283 rows × 2 columns

tweets_df.isna().sum().sum() returns the total count of missing values in the entire tweets_df DataFrame. This can be useful for data quality assessment and data preprocessing. If the result is 0, it means there are no missing values in the DataFrame. If the result is greater than 0, it indicates the number of missing values in the DataFrame.

In [ ]:
```
tweets_df.isna().sum().sum()
```

Out[92]:
```
0
```

The result of this code will provide you with a count of each unique sentiment category present in the "airline_sentiment" column, which is useful for understanding the distribution of sentiment labels in your dataset. Typically, it's used for initial exploratory data analysis to get insights into the class distribution of a categorical variable.

In [ ]:
```
tweets_df['airline_sentiment'].value_counts()
```

Out[93]:
```
negative    8238
neutral     2851
positive    2194
Name: airline_sentiment, dtype: int64
```

The "airline_sentiment" column is transformed into a numeric representation where 'negative' is represented as 0, 'neutral' as 1, and 'positive' as 2. This numeric representation can be useful when working with machine learning models that require numerical inputs for sentiment analysis, as it encodes the sentiment labels in the desired order.

In [ ]:
```
sentiment_ordering = ['negative', 'neutral', 'positive']

tweets_df['airline_sentiment'] = tweets_df['airline_sentiment'].apply(lambda x: sentiment_ordering.index(x))
```

In [ ]:
```
tweets_df
```

Out[95]:

	text	airline_sentiment
0	@VirginAmerica What @dhepburn said.	1
1	@VirginAmerica I didn't today... Must mean I need to take another trip!	1
2	@VirginAmerica it's really aggressive to blast obnoxious "entertainment" in your guests' faces & they have little recourse	0
3	@VirginAmerica and it's a really big bad thing about it	0
4	@VirginAmerica seriously would pay $30 a flight for seats that didn't have this playing.\nit's really the only bad thing about flying VA	0
...	...	...
13278	@AmericanAir thank you for doing the best you could to get me rebooked. Agent on phone & addtl resolution on DM was very much appreciated.	2
13279	@AmericanAir no email no phone call no nothing. You've screwed with my flight and my family/Friends flights. You Cancelled Flighted reservations for	0
13280	@AmericanAir If you care, could you have someone call me to explain what is going on.	0
13281	Hey @AmericanAir why automated call me and then hang up at 4:45 am!?! And why can't I reschedule Cancelled Flighted flights via web!?! Come on!!!	0
13282	@AmericanAir from a service rep but that hasn't happened	0

13283 rows × 2 columns

In [ ]:
```
emoji.demojize('@AmericanAir right on cue with the delays 👌')
```

Out[96]:
```
'@AmericanAir right on cue with the delays:OK_hand:'
```

##TEXT CLEANING

this function performs a series of text cleaning and preprocessing steps, including lowercasing, removing mentions, hashtags, URLs, punctuation, and

In [ ]:

```
ps = PorterStemmer()

def process_tweet(tweet):
    new_tweet = tweet.lower()
    new_tweet = re.sub(r'@\w+', '', new_tweet) # Remove @s
    new_tweet = re.sub(r'#', '', new_tweet) # Remove hashtags
    new_tweet = re.sub(r':', ' ', emoji.demojize(new_tweet)) # Turn emojis into words
    new_tweet = re.sub(r'http\S+', '',new_tweet) # Remove URLs
    new_tweet = re.sub(r'\$\S+', 'dollar', new_tweet) # Change dollar amounts to dollar
    new_tweet = re.sub(r'[^a-z0-9\s]', '', new_tweet) # Remove punctuation
    new_tweet = re.sub(r'[0-9]+', 'number', new_tweet) # Change number values to number
    new_tweet = new_tweet.split(" ")
    new_tweet = list(map(lambda x: ps.stem(x), new_tweet)) # Stemming the words
    new_tweet = list(map(lambda x: x.strip(), new_tweet)) # Stripping whitespace from the words
    if '' in new_tweet:
        new_tweet.remove('')
    return new_tweet
```

The result is that you have tweets as a Series of preprocessed tweets (each element is a list of words), and labels as a NumPy array containing sentiment labels for each tweet. This data is typically used for training and evaluating machine learning models for sentiment analysis.

In [ ]:

```
tweets = tweets_df['text'].apply(process_tweet)
labels = np.array(tweets_df['airline_sentiment'])
```

In [ ]:

```
tweets
```

Out[99]:

```
0
[what, , said]
1                                                                                    [i, didnt, today, must, mean, i,
need, to, take, anoth, trip]
2                                            [it, realli, aggress, to, blast, obnoxi, entertain, in, your, guest, face, am
p, they, have, littl, recours]
3                                                                                                   [and, it, a, reall
i, big, bad, thing, about, it]
4                   [serious, would, pay, dollar, a, flight, for, seat, that, didnt, have, thi, playing\nit, realli, the, onl
i, bad, thing, about, fli, va]
                                                                            ...
13278             [thank, you, for, do, the, best, you, could, to, get, me, rebook, agent, on, phone, amp, addtl, resolut, on,
dm, wa, veri, much, appreci]
13279                [no, email, no, phone, call, no, noth, youv, screw, with, my, flight, and, my, familyfriend, flight, you,
cancel, flight, reserv, for]
13280                                                                 [if, you, care, could, you, have, someon, call, me, t
o, explain, what, is, go, on]
13281     [hey, whi, autom, call, me, and, then, hang, up, at, number, number, am, and, whi, cant, i, reschedul, cancel, fligh
t, flight, via, web, come, on]
13282                                                                                              [from, a, servic, re
p, but, that, hasnt, happen, ]
Name: text, Length: 13283, dtype: object
```

Vocab length will be the number of unique words in your dataset. Max sequence length will be the length (in terms of the number of words) of the longest tweet in your dataset. These values are important for tasks like text tokenization, padding sequences for model input, and determining the vocabulary size for embedding layers in neural networks.

In [ ]:

```
# Get size of vocabulary
vocabulary = set()

for tweet in tweets:
    for word in tweet:
        if word not in vocabulary:
            vocabulary.add(word)

vocab_length = len(vocabulary)

# Get max length of a sequence
max_seq_length = 0

for tweet in tweets:
    if len(tweet) > max_seq_length:
        max_seq_length = len(tweet)

# Print results
print("Vocab length:", vocab_length)
print("Max sequence length:", max_seq_length)
```

```
Vocab length: 10759
Max sequence length: 90
```

model_inputs is a 2D NumPy array where each row represents a tweet, and the integer values in each row correspond to the preprocessed and tokenized words. All sequences have been padded to have the same length to prepare them for use as inputs to a machine learning model, such as a neural network for sentiment analysis.

In [ ]:

```
tokenizer = Tokenizer(num_words=vocab_length)
tokenizer.fit_on_texts(tweets)
sequences = tokenizer.texts_to_sequences(tweets)


word_index = tokenizer.word_index

model_inputs = pad_sequences(sequences, maxlen=max_seq_length, padding='post')
```

In [ ]:

```
model_inputs
```

Out[102]:

```
array([[  49,    2,  209, ...,    0,    0,    0],
       [   5,  190,   99, ...,    0,    0,    0],
       [  15,  142, 2740, ...,    0,    0,    0],
       ...,
       [  69,    8,  234, ...,    0,    0,    0],
       [ 490,   70,  851, ...,    0,    0,    0],
       [  30,    7,   40, ...,    0,    0,    0]], dtype=int32)
```

In [ ]:

```
model_inputs.shape
```

Out[103]:

```
(13283, 90)
```

By splitting your data into training and testing sets, you can train your machine learning model on one portion of the data and evaluate its performance on another independent portion. This helps you assess how well your model generalizes to new, unseen data.

In [ ]:

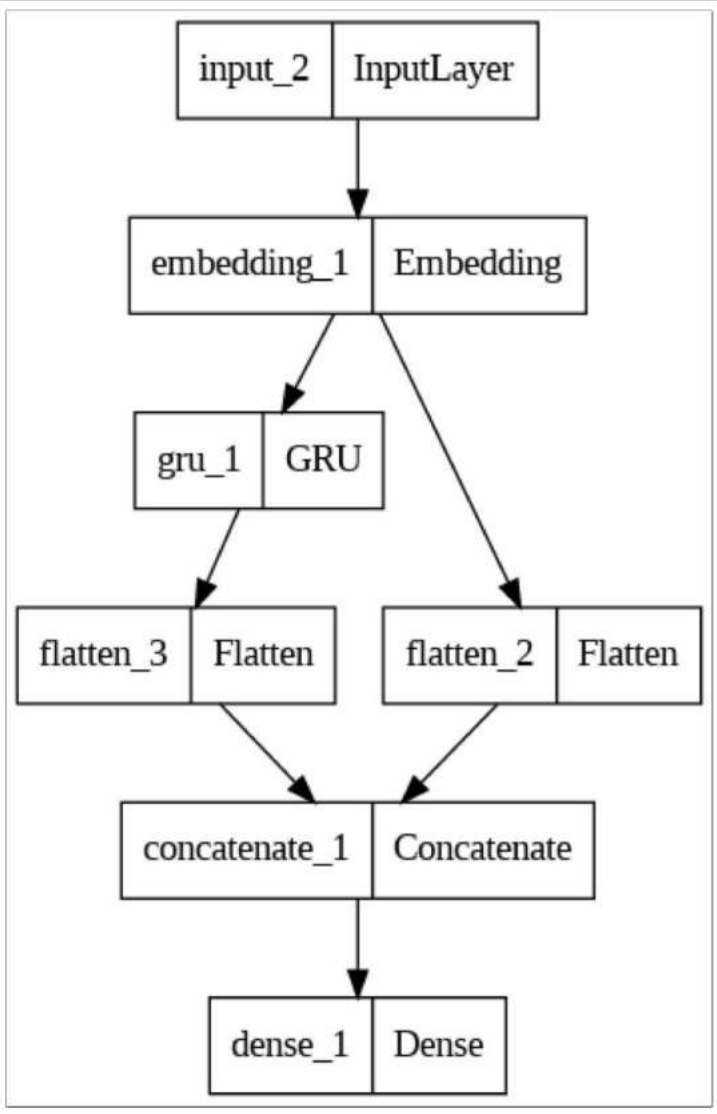
```
X_train, X_test, y_train, y_test = train_test_split(model_inputs, labels, train_size=0.7, random_state=22)
```

This model combines two branches: one that flattens the embeddings and another that processes the embeddings using a GRU layer. The concatenated output is then used for sentiment classification. This architecture allows the model to capture different aspects of the input data and make sentiment predictions.

In [ ]:

```
embedding_dim = 32


inputs = tf.keras.Input(shape=(max_seq_length,))

embedding = tf.keras.layers.Embedding(
    input_dim=vocab_length,
    output_dim=embedding_dim,
    input_length=max_seq_length
)(inputs)


# Model A (just a Flatten layer)
flatten = tf.keras.layers.Flatten()(embedding)

# Model B (GRU with a Flatten layer)
gru = tf.keras.layers.GRU(units=embedding_dim)(embedding)
gru_flatten = tf.keras.layers.Flatten()(gru)

# Both A and B are fed into the output
concat = tf.keras.layers.concatenate([flatten, gru_flatten])

outputs = tf.keras.layers.Dense(3, activation='softmax')(concat)


model = tf.keras.Model(inputs, outputs)

tf.keras.utils.plot_model(model)
```

Out[105]:



The training process updates the model's weights using backpropagation and optimization. The training history, including loss and accuracy values for each epoch, is stored in the history variable, which you can use for visualization and analysis. This code demonstrates how to train a deep learning model for sentiment analysis with early stopping and learning rate reduction strategies.

In [ ]:

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)


batch_size = 32
epochs = 100

history = model.fit(
    X_train,
    y_train,
    validation_split=0.2,
    batch_size=batch_size,
    epochs=epochs,
    callbacks=[
        tf.keras.callbacks.EarlyStopping(
            monitor='val_loss',
            patience=3,
            restore_best_weights=True,
            verbose=1
        ),
        tf.keras.callbacks.ReduceLROnPlateau()
    ]
)
```

```
Epoch 1/100
233/233 [==============================] - 26s 96ms/step - loss: 0.8076 - accuracy: 0.6530 - val_loss: 0.7008 - val_accuracy:
0.7054 - lr: 0.0010
Epoch 2/100
233/233 [==============================] - 6s 27ms/step - loss: 0.5502 - accuracy: 0.7829 - val_loss: 0.5706 - val_accuracy:
0.7667 - lr: 0.0010
Epoch 3/100
233/233 [==============================] - 7s 29ms/step - loss: 0.3929 - accuracy: 0.8630 - val_loss: 0.5230 - val_accuracy:
0.7925 - lr: 0.0010
Epoch 4/100
233/233 [==============================] - 4s 18ms/step - loss: 0.2910 - accuracy: 0.9062 - val_loss: 0.5137 - val_accuracy:
0.7957 - lr: 0.0010
Epoch 5/100
233/233 [==============================] - 6s 25ms/step - loss: 0.2163 - accuracy: 0.9360 - val_loss: 0.5246 - val_accuracy:
0.7898 - lr: 0.0010
Epoch 6/100
233/233 [==============================] - 3s 13ms/step - loss: 0.1614 - accuracy: 0.9579 - val_loss: 0.5376 - val_accuracy:
0.7941 - lr: 0.0010
Epoch 7/100
233/233 [==============================] - ETA: 0s - loss: 0.1209 - accuracy: 0.9715Restoring model weights from the end of th
e best epoch: 4.
233/233 [==============================] - 2s 11ms/step - loss: 0.1209 - accuracy: 0.9715 - val_loss: 0.5599 - val_accuracy:
0.7914 - lr: 0.0010
Epoch 7: early stopping
```

if you run model.evaluate(X_test, y_test) and the model has been trained to perform sentiment analysis, you'll receive metrics such as the test loss and test accuracy, which will give you insights into how well the model generalizes to new, unseen data.

In [ ]:

```
model.evaluate(X_test, y_test)
```

```
125/125 [==============================] - 1s 5ms/step - loss: 0.4996 - accuracy: 0.8018
```

Out[107]:

```
[0.49962925910949707, 0.8017565608024597]
```

IMPORT BERT AS NLP

After running this code, the bert variable will contain the pre-trained BERT model, and you can use it for various natural language processing tasks, including text classification, question-answering, and more. You can fine-tune this model on your specific NLP tasks or use it as a feature extractor to obtain contextual embeddings for text data.

In [ ]:

```
from transformers.models.bert.modeling_bert import BertModel
# Import BERT pretrained module
from transformers import BertModel

#Download uncased bert base model
bert=BertModel.from_pretrained('bert-base-uncased')
```

In [ ]:

```
# Print BERT arcitecture
print(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()
  )
)
```

After running this code, the tokenizer variable will be an instance of the BERT tokenizer, and you can use it to tokenize text data, converting it into the format required for input to the BERT model. This tokenizer takes care of tokenizing, padding, and other text preprocessing tasks, making it easy to work with BERT for various NLP tasks.

In [ ]:

```
from transformers.models.bert.tokenization_bert_fast import BertTokenizerFast
# importing BERT tokenizer
tokenizer=BertTokenizerFast.from_pretrained('bert-base-uncased',do_lower_case=True)
```

The sentence_id will be a list of integers representing the token IDs in the encoded sequence, including the special tokens. This integer sequence is suitable for feeding into a BERT-based model for various NLP tasks like text classification, named entity recognition, and more.

In [ ]:

```
text='Jim Henson was a puppeteer'
sentence_id=tokenizer.encode(text,
                             # add special character tokens
                             add_special_tokens=True,
                             # Specifying maximum length for any input sequences
                             max_length=10,
                             # if exceeeding 10, then it will be truncated, if <10, then it will be padded.
                             truncation=True,
                             # add pad tokens to the right side of the sequence
                             pad_to_max_length='right'
                             )
print("Integer Sequence:{}".format(sentence_id))
```

```
Integer Sequence:[101, 3958, 27227, 2001, 1037, 13997, 11510, 102, 0, 0]

/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:2606: FutureWarning: The `pad_to_max_length` a
rgument is deprecated and will be removed in a future version, use `padding=True` or `padding='longest'` to pad to the longest
sequence in the batch, or use `padding='max_length'` to pad to a max length. In this case, you can give a specific length with
`max_length` (e.g. `max_length=45`) or leave max_length to None to pad to the maximal input size of the model (e.g. 512 for Be
rt).
  warnings.warn(
```

The output of this code will be the original text tokens from your input text, demonstrating the bidirectional nature of the tokenizer. This is useful for verifying that the encoding and decoding processes work correctly and for inspecting the tokenization of your text.

In [ ]:

```
# converting integers back to text
print("Tokenizer Text: ",tokenizer.convert_ids_to_tokens(sentence_id))
```

```
Tokenizer Text:  ['[CLS]', 'jim', 'henson', 'was', 'a', 'puppet', '##eer', '[SEP]', '[PAD]', '[PAD]']
```

The output of this code will be the original text string that you initially provided for tokenization. This step verifies that the encoding and decoding processes are consistent and that you can recover the original text from the encoded sequence. It's a crucial step when working with tokenization in NLP tasks.

In [ ]:
```
decoded=tokenizer.decode(sentence_id)
print('Decoded String:{}'.format(decoded))
```

```
Decoded String:[CLS] jim henson was a puppeteer [SEP] [PAD] [PAD]
```

print(att_mask): This line prints the generated attention mask as a list of 1s and 0s. The att_mask list is important in NLP tasks, as it tells the model which parts of the input sequence should be given attention during processing. Padding tokens are typically ignored (masked) to ensure that the model focuses on the actual content of the input.

In [ ]:
```
att_mask=[int(tok>0) for tok in sentence_id]
print(att_mask)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 0, 0]
```

By creating tensors and reshaping them, you prepare the input in the required format for BERT models, which typically expect a batch dimension for processing multiple input sequences simultaneously.

In [ ]:
```
# convert lists to tensors
# torch.tensor creates a tensor of given data
sent_id=torch.tensor(sentence_id)
attn_mask=torch.tensor(att_mask)
print('Shape of sentence_id before reshaping is: {}'.format(sent_id.shape))
print('Shape of sentence_id before reshaping is: {}'.format(attn_mask.shape))
print('\n')
# reshaping tensor in form of batch,text length
sent_id=sent_id.unsqueeze(0)
attn_mask=attn_mask.unsqueeze(0)
print('Shape of sentence_id after reshaping is: {}'.format(sent_id.shape))
print('Shape of sentence_id after reshaping is: {}'.format(attn_mask.shape))
print('\n')
# reshaped tensor
print(sent_id)
```

```
Shape of sentence_id before reshaping is: torch.Size([10])
Shape of sentence_id before reshaping is: torch.Size([10])


Shape of sentence_id after reshaping is: torch.Size([1, 10])
Shape of sentence_id after reshaping is: torch.Size([1, 10])


tensor([[  101,  3958, 27227,  2001,  1037, 13997, 11510,   102,     0,     0]])
```

The outputs variable will store the model's output, which typically includes information such as hidden states, pooled representations, etc., depending on the specific BERT model architecture used.

By calling bert(sent_id, attention_mask=attn_mask), you're using the BERT model to process your input text and obtain contextualized representations of the tokens. These representations can be used for various downstream NLP tasks, such as text classification, entity recognition, or question answering.

In [ ]:
```
# passing integer sequence and attention mask tensor to BERT model
outputs=bert(sent_id,attention_mask=attn_mask)
```

The all_hidden_states tensor contains the contextualized representations of the tokens after they have been processed by the BERT model. These representations are valuable for various downstream NLP tasks that require understanding the context and relationships between words in a text.

In [ ]:
```
# Unpacking the output of BERT model

# all_hidden_states is a collection of all the output vectors/ hidden states (of encoder) at each timestamps or position of the
all_hidden_states=outputs[0]

print(all_hidden_states.shape)
print(all_hidden_states)
```

```
torch.Size([1, 10, 768])
tensor([[[-0.2531,  0.2038, -0.3862,  ..., -0.3034,  0.6197,  0.2373],
         [-0.2323, -0.0044, -0.5479,  ...,  0.0765,  0.8122, -0.4710],
         [ 0.2590,  0.7140, -0.5438,  ..., -0.3774,  0.9987,  0.5400],
         ...,
         [ 0.7873,  0.3299, -0.0351,  ...,  0.2932, -0.5141,  0.0308],
         [-0.5547, -0.3669, -0.1106,  ...,  0.2593,  0.5321, -0.3871],
         [-0.5461, -0.2414, -0.2111,  ...,  0.3100,  0.5863, -0.3467]]],
       grad_fn=<NativeLayerNormBackward0>)
```

The cls_hidden_state tensor contains the encoded representation of the entire input sequence, as summarized by the [CLS] token. This representation can be used for various tasks such as text classification, where you need to make predictions based on the overall content of the input sequence.

In [ ]:

```
# this output contains output vector against the CLS token only (at the first position of BERT model)
# this output vector encodes the entire input sequence

cls_hidden_state=outputs[1]

print(cls_hidden_state.shape)
print(cls_hidden_state)
```

```
torch.Size([1, 768])
tensor([[-0.8767, -0.4109, -0.1220,  0.4494,  0.1945, -0.2698,  0.8316,  0.3127,
          0.1178, -1.0000, -0.1561,  0.6677,  0.9891, -0.3451,  0.8812, -0.6753,
         -0.3079, -0.5580,  0.4380, -0.4588,  0.5831,  0.9956,  0.4467,  0.2863,
          0.3924,  0.6864, -0.7513,  0.9043,  0.9436,  0.8207, -0.6493,  0.3524,
         -0.9919, -0.2295, -0.0742, -0.9936,  0.3698, -0.7558,  0.0792, -0.2218,
         -0.8637,  0.4711,  0.9997, -0.4368,  0.0404, -0.3498, -1.0000,  0.2663,
         -0.8711,  0.0508,  0.0505, -0.1634,  0.1716,  0.4363,  0.4330, -0.0333,
         -0.0416,  0.2206, -0.2568, -0.6122, -0.5916,  0.2569, -0.2622, -0.9041,
          0.3221, -0.2394, -0.2634, -0.3454, -0.0723,  0.0081,  0.8297,  0.2279,
          0.1614, -0.6555, -0.2062,  0.3280, -0.4016,  1.0000, -0.0952, -0.9874,
         -0.0400,  0.0717,  0.3675,  0.3373, -0.3710, -1.0000,  0.4479, -0.1722,
         -0.9917,  0.2677,  0.4844, -0.2207, -0.3207,  0.3715, -0.2171, -0.2522,
         -0.3071, -0.3161, -0.1988, -0.0860, -0.0114, -0.1982, -0.1799, -0.3221,
          0.1751, -0.4442, -0.1570, -0.0434, -0.0893,  0.5717,  0.3112, -0.2900,
          0.3305, -0.9430,  0.6061, -0.2984, -0.9873, -0.3956, -0.9926,  0.7857,
         -0.1692, -0.2719,  0.9505,  0.5628,  0.2904, -0.1693,  0.1619, -1.0000,
         -0.1697, -0.1534,  0.2513, -0.2857, -0.9846, -0.9638,  0.5565,  0.9200,
          0.1805,  0.9995, -0.2122,  0.9391,  0.3246, -0.3937, -0.1248, -0.5209,
          0.0519,  0.1141, -0.6463,  0.3529, -0.0322, -0.3837, -0.3796, -0.2830,
          0.1280, -0.9191, -0.4201,  0.9145,  0.0713, -0.2455,  0.5212, -0.2642,
         -0.3675,  0.8082,  0.2577,  0.2755, -0.0157,  0.3675, -0.3107,  0.4502,
         -0.8224,  0.2841,  0.4360, -0.3193,  0.2164, -0.9851, -0.4444,  0.5759,
          0.9878,  0.7531,  0.3384,  0.2003, -0.2602,  0.4695, -0.9561,  0.9855,
         -0.1712,  0.2295,  0.1220, -0.1386, -0.8436, -0.3783,  0.8371, -0.3204,
         -0.8457, -0.0473, -0.4219, -0.3593, -0.2187,  0.5282, -0.3149, -0.4375,
         -0.0440,  0.9242,  0.9296,  0.7735, -0.3733,  0.3945, -0.9049, -0.2898,
          0.2695,  0.2910,  0.1695,  0.9932, -0.3069, -0.1611, -0.8349, -0.9827,
          0.1299, -0.8555, -0.0531, -0.6830,  0.3926,  0.2873, -0.1899,  0.2598,
         -0.9201, -0.7455,  0.3943, -0.3955,  0.4015, -0.2341,  0.7593,  0.3421,
         -0.6143,  0.5170,  0.8987,  0.1072, -0.6858,  0.6481, -0.2454,  0.8712,
         -0.5958,  0.9936,  0.3404,  0.4972, -0.9452, -0.2347, -0.8748, -0.0154,
         -0.1293, -0.5265,  0.4235,  0.4206,  0.3663,  0.7488, -0.4650,  0.9900,
         -0.8695, -0.9701, -0.5203, -0.0900, -0.9914,  0.0978,  0.2844, -0.0424,
         -0.4649, -0.4546, -0.9620,  0.8035,  0.2177,  0.9705, -0.0793, -0.7985,
         -0.3436, -0.9537, -0.0035, -0.0945,  0.4291,  0.0391, -0.9602,  0.4497,
          0.5135,  0.4913,  0.0608,  0.9948,  1.0000,  0.9810,  0.8865,  0.7961,
         -0.9894, -0.5122,  1.0000, -0.8521, -1.0000, -0.9412, -0.6633,  0.3110,
         -1.0000, -0.1468, -0.1235, -0.9465, -0.0891,  0.9796,  0.9700, -1.0000,
          0.9324,  0.9259, -0.4503,  0.4591, -0.1785,  0.9819,  0.2285,  0.4423,
         -0.2615,  0.4124, -0.5252, -0.8534,  0.0365, -0.0670,  0.8944,  0.1913,
         -0.4782, -0.9402,  0.2293, -0.1581, -0.2440, -0.9604, -0.1924, -0.0555,
          0.5484,  0.1915,  0.2038, -0.7367,  0.2698, -0.7307,  0.3715,  0.5640,
         -0.9386, -0.5717,  0.3818, -0.2775,  0.1536, -0.9608,  0.9702, -0.3502,
          0.1524,  1.0000,  0.3876, -0.9001,  0.2547,  0.1857,  0.0832,  1.0000,
          0.3811, -0.9852, -0.4053,  0.2576, -0.3923, -0.4125,  0.9994, -0.1463,
         -0.0428,  0.2818,  0.9899, -0.9923,  0.8351, -0.8563, -0.9634,  0.9617,
          0.9268, -0.4225, -0.7369,  0.1318,  0.1107,  0.2294, -0.8914,  0.6082,
          0.4665, -0.0720,  0.8555, -0.7973, -0.3478,  0.4201, -0.1762,  0.0761,
          0.2823,  0.4571, -0.1350,  0.1190, -0.3509, -0.4039, -0.9556,  0.0262,
          1.0000, -0.2164,  0.0569, -0.2296, -0.1003, -0.1827,  0.4036,  0.4715,
         -0.3293, -0.8471, -0.0518, -0.8453, -0.9935,  0.6732,  0.2284, -0.1968,
          0.9998,  0.5194,  0.2326,  0.1718,  0.7497, -0.0192,  0.4518, -0.0327,
          0.9765, -0.3259,  0.3491,  0.7471, -0.3186, -0.3019, -0.5725,  0.0563,
         -0.9206,  0.0572, -0.9589,  0.9565,  0.3109,  0.3348,  0.1635, -0.0619,
          1.0000, -0.6020,  0.5309, -0.3723,  0.6636, -0.9851, -0.6789, -0.4312,
         -0.1435, -0.0827, -0.2497,  0.1323, -0.9786, -0.0474, -0.0304, -0.9444,
         -0.9927,  0.2508,  0.6172,  0.1679, -0.7980, -0.6078, -0.4906,  0.4646,
         -0.1934, -0.9396,  0.5453, -0.3000,  0.4329, -0.3340,  0.4408, -0.2058,
          0.8344,  0.1265, -0.0307, -0.2098, -0.8340,  0.7114, -0.7410,  0.0518,
         -0.1481,  1.0000, -0.3100,  0.1461,  0.7011,  0.6334, -0.2857,  0.1618,
          0.0966,  0.2955, -0.0981, -0.1832, -0.6208, -0.3013,  0.4337,  0.0283,
         -0.2959,  0.7579,  0.4711,  0.3666, -0.0531,  0.0914,  0.9969, -0.2267,
         -0.1165, -0.5533, -0.1262, -0.3575, -0.2124,  1.0000,  0.3679,  0.0604,
         -0.9936, -0.2000, -0.9208,  0.9999,  0.8511, -0.8783,  0.5650,  0.2405,
         -0.2859,  0.6935, -0.2598, -0.2655,  0.2893,  0.2862,  0.9774, -0.4575,
         -0.9764, -0.5964,  0.3966, -0.9575,  0.9939, -0.5326, -0.2349, -0.4376,
         -0.0250,  0.2574,  0.0274, -0.9762, -0.1582,  0.1821,  0.9811,  0.3014,
         -0.3820, -0.9007, -0.1151,  0.3936, -0.0680, -0.9449,  0.9809, -0.9313,
          0.2600,  1.0000,  0.3860, -0.5243,  0.2401, -0.4410,  0.3253, -0.1413,
          0.5428, -0.9466, -0.2817, -0.3262,  0.4330, -0.2120, -0.2457,  0.7247,
          0.2134, -0.3430, -0.6305, -0.1214,  0.4871,  0.7498, -0.2957, -0.1829,
          0.1699, -0.1391, -0.9264, -0.4167, -0.2995, -0.9991,  0.6411, -1.0000,
         -0.1510, -0.5473, -0.2219,  0.8075,  0.3862, -0.1392, -0.7206, -0.0710,
          0.6995,  0.6656, -0.2889,  0.2902, -0.6951,  0.1622, -0.1298,  0.3182,
          0.1694,  0.6526, -0.2735,  1.0000,  0.1370, -0.3043, -0.9189,  0.3041,
         -0.2604,  1.0000, -0.7969, -0.9715,  0.2110, -0.5773, -0.7218,  0.2477,
         -0.0304, -0.7015, -0.6577,  0.9111,  0.8219, -0.3693,  0.4537, -0.3062,
         -0.3671,  0.0856,  0.1595,  0.9903,  0.2790,  0.8213, -0.2885, -0.0724,
          0.9636,  0.2213,  0.6892,  0.2070,  1.0000,  0.3249, -0.8999,  0.2644,
         -0.9700, -0.2610, -0.9228,  0.4016,  0.1170,  0.8570, -0.3587,  0.9672,
          0.0667,  0.1108, -0.1840,  0.4711,  0.3127, -0.9391, -0.9892, -0.9908,
          0.3962, -0.5013, -0.0640,  0.3811,  0.1530,  0.4712,  0.3781, -1.0000,
          0.9466,  0.3529,  0.2077,  0.9735,  0.2019,  0.4726,  0.4248, -0.9892,
         -0.9203, -0.3418, -0.2910,  0.6572,  0.5584,  0.8190,  0.4319, -0.4171,
         -0.4697,  0.4653, -0.8583, -0.9940,  0.4802,  0.0740, -0.8986,  0.9559,
         -0.4745, -0.1616,  0.4457,  0.1412,  0.8933,  0.8280,  0.4313,  0.2437,
          0.6787,  0.9043,  0.8940,  0.9903, -0.2561,  0.6986, -0.0055,  0.3281,
          0.6809, -0.9586,  0.1583,  0.0033, -0.2711,  0.3025, -0.1928, -0.9207,
          0.5260, -0.2139,  0.5709, -0.2302,  0.1593, -0.4779, -0.1577, -0.7036,
         -0.5208,  0.4676,  0.2335,  0.9372,  0.4775, -0.1995, -0.5655, -0.2336,
          0.0798, -0.9315,  0.8288, -0.0946,  0.5294,  0.0223, -0.0744,  0.7821,
          0.1236, -0.3705, -0.3959, -0.7528,  0.8145, -0.3204, -0.4786, -0.5135,
```

```
         0.7306,  0.3208,  0.9981, -0.3959, -0.3492, -0.1118, -0.2872,  0.3596,
        -0.1345, -1.0000,  0.2896,  0.2262,  0.1702, -0.3530,  0.1111, -0.0755,
        -0.9565, -0.2658,  0.2530, -0.0490, -0.5834, -0.4616,  0.3937,  0.2329,
         0.5620,  0.8138, -0.0288,  0.5621,  0.3811,  0.0852, -0.6049,  0.8452]],
      grad_fn=<TanhBackward0>)
```

In [ ]:
```
data.shape
```

Out[120]: (13283, 15)

In [ ]:
```
df=tweets_df
```

The class_counts list can be used for various purposes, such as data analysis, visualization, or further processing in your code. It provides a convenient way to access and work with the counts of each sentiment category in the DataFrame.

In [ ]:
```
# Sabing value counts to a list
class_counts=df['airline_sentiment'].value_counts().to_list()
```

###TEXT CLEANING IN PREPROCESSING

The preprocess function can be used to clean and preprocess text data before further analysis, such as natural language processing tasks like text classification, sentiment analysis, or text generation. It's a common practice to prepare text data by removing noise and unnecessary information.

In [ ]:
```
def preprocess(text):
  # converting text tolower case
  text=text.lower()
  # remove user mentions
  text=re.sub(r'@[A-Za-z0-9]+','',text)
  # remove hashtags if needed keep for now
  #text=re.sub(r'#[A-Za-z0-9]+','',text)

  # remove links
  text=re.sub(r'http\S+','',text)

  # Split tokens so that extra spaces which were added due to above substitution are removed
  tokens=text.split()

  # join tokens by space
  return ' '.join(tokens)
```

This code efficiently applies your text preprocessing function to each text entry in the 'text' column and stores the cleaned text in a new column for further analysis or text processing tasks.

In [ ]:
```
# using apply function to apply this preprocess function on each row of the text column
data['cleaned_text']=data['text'].apply(preprocess)
```

The result of this code will be a table-like display showing the first few rows of the DataFrame data, with columns for sentiment, the original text, and the cleaned (preprocessed) text. This is helpful for visually inspecting the data and verifying the preprocessing steps you applied to the text.

In [ ]:
```
data.head()[['airline_sentiment','text','cleaned_text']]
```

Out[126]:

	airline_sentiment	text	cleaned_text
0	neutral	@VirginAmerica What @dhepburn said.	what said.
1	neutral	@VirginAmerica I didn't today... Must mean I need to take another trip!	i didn't today... must mean i need to take another trip!
2	negative	@VirginAmerica it's really aggressive to blast obnoxious "entertainment" in your guests' faces & they have little recourse	it's really aggressive to blast obnoxious "entertainment" in your guests' faces & they have little recourse
3	negative	@VirginAmerica and it's a really big bad thing about it	and it's a really big bad thing about it
4	negative	@VirginAmerica seriously would pay $30 a flight for seats that didn't have this playing.\nit's really the only bad thing about flying VA	seriously would pay $30 a flight for seats that didn't have this playing. it's really the only bad thing about flying va

After executing this code, you have two variables, text and labels, that contain the preprocessed text data and their corresponding sentiment labels. These variables are typically used as input to machine learning or deep learning models for sentiment analysis or other natural language processing tasks.

In [ ]:
```
# Saving cleaned text and labels to variables
text=data['cleaned_text'].values
labels=data['airline_sentiment'].values
```

In [ ]:
```

```

###PREPARING INPUT AND OUTPUT

The LabelEncoder is useful when you have categorical labels in your dataset, and you need to convert them into a format suitable for machine learning algorithms that require numerical inputs. You can use the fit and transform methods of the LabelEncoder to perform this encoding.

In [ ]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
```

After running this code, the labels variable will be a NumPy array containing numerical representations for the sentiment labels. For example, 'positive' might be represented as 0, 'negative' as 1, and 'neutral' as 2. These numerical labels are suitable for use in machine learning models, as they replace the original text labels with numeric values that can be processed by algorithms.

In [ ]:
```
# Using label encoder, convert textual labels (positive, negative, neutral) into numners
le=LabelEncoder()

#fit and transform target strings to a number
labels=le.fit_transform(labels)
```

In [ ]:
```
le.classes_
```

Out[131]:
```
array(['negative', 'neutral', 'positive'], dtype=object)
```

In [ ]:
```
labels
```

Out[132]:
```
array([1, 1, 0, ..., 0, 0, 0])
```

In [ ]:
```
len(labels)
```

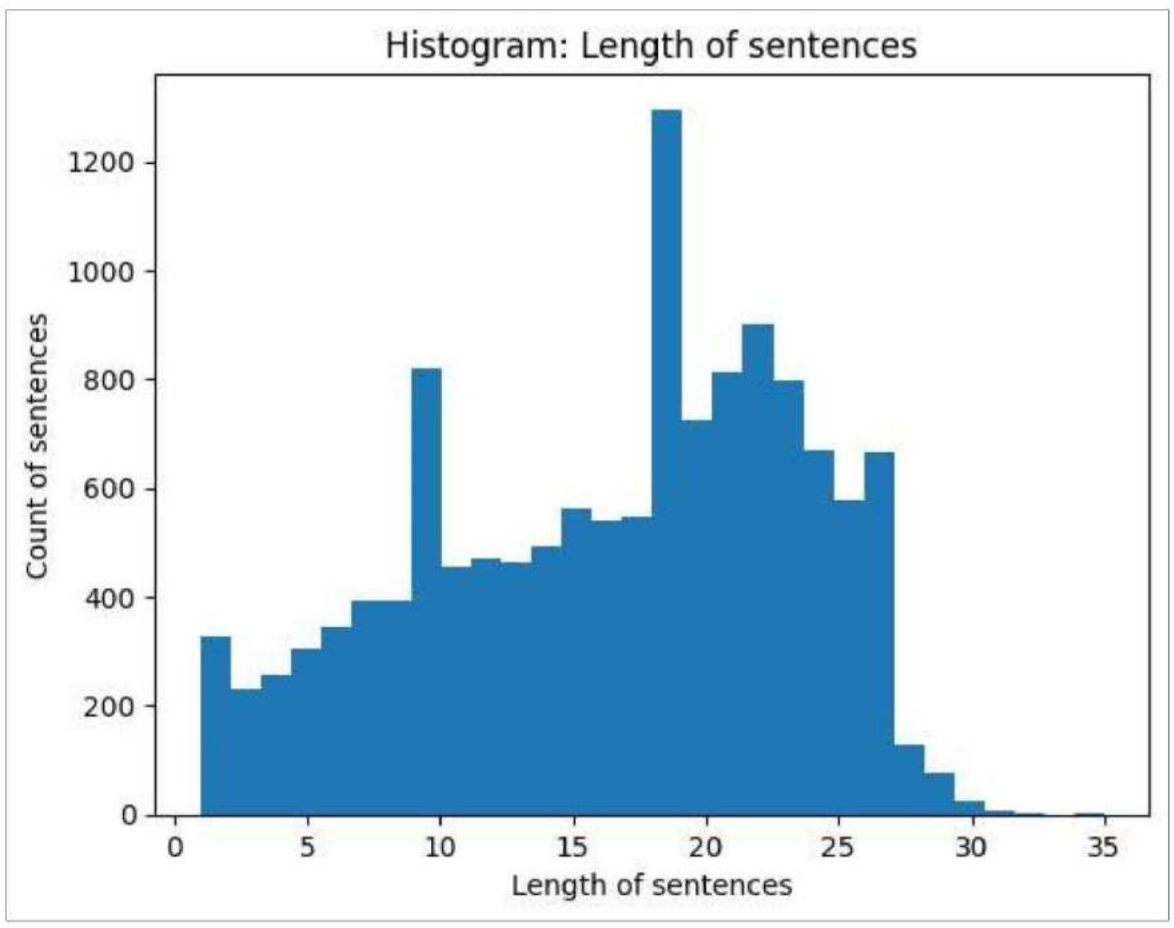
Out[133]:
```
13283
```

#VISUALIZATION

The resulting plot is a histogram that shows the distribution of sentence lengths in your text data. You can visualize how many sentences fall into different length ranges, which can be useful for understanding the data's characteristics and deciding how to preprocess it further for various NLP tasks.

In [ ]:
```
num=[len(i.split()) for i in text]
plt.hist(num,bins=30)
plt.title('Histogram: Length of sentences')
plt.xlabel('Length of sentences')
plt.ylabel('Count of sentences')
```

Out[135]:
```
Text(0, 0.5, 'Count of sentences')
```



Tuning hyperparameters like max_len is an important part of optimizing the performance of machine learning and deep learning models for specific tasks. You can experiment with different values for max_len to see how it affects your model's performance on your NLP task.

In [ ]:
```
max_len=28 # This is a hyper parameter which can be tuned
```

By the end of this code, the sent_id list will contain integer sequences for each preprocessed tweet, suitable for input to BERT-based models or other NLP tasks.

In [ ]:

```
# Create an empty list to save integer sequence
sent_id=[]

# iterate over each tweet and encode it using bert tokenizer
for i in notebook.tqdm(range(len(text))):
  encoded_sent=tokenizer.encode(text[i],
                                add_special_tokens=True,
                                max_length= max_len,
                                truncation=True,
                                pad_to_max_length='right'
                                )

  # save integer sequence to a list
  sent_id.append(encoded_sent)
```

```
  0%|          | 0/13283 [00:00<?, ?it/s]
```

```
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:2606: FutureWarning: The `pad_to_max_length` a
rgument is deprecated and will be removed in a future version, use `padding=True` or `padding='longest'` to pad to the longest
sequence in the batch, or use `padding='max_length'` to pad to a max length. In this case, you can give a specific length with
`max_length` (e.g. `max_length=45`) or leave max_length to None to pad to the maximal input size of the model (e.g. 512 for Be
rt).
  warnings.warn(
```

In [ ]:
```
print(text[0])
```

```
what said.
```

In [ ]:
```
print(sent_id[0])
```

```
[101, 2054, 2056, 1012, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

In [ ]:
```
len(sent_id)
```

Out[140]:
```
13283
```

By the end of this code, the attention_mask list will contain attention masks for each integer sequence in the sent_id list. These attention masks specify which tokens in each sequence should receive attention (1) and which tokens should be masked out (0) during processing in an NLP model, such as a BERT-based model.

In [ ]:
```
attention_mask=[]

for sent in sent_id:
  attn_mask=[int(token_id>0) for token_id in sent]
  attention_mask.append(attn_mask)
```

In [ ]:
```
len(attention_mask)
```

Out[142]:
```
13283
```

###Training and Validation Data

These splits are typically used in the training and evaluation of machine learning or deep learning models, such as BERT-based models, for NLP tasks.

In [ ]:
```
# Splitting input data
train_inputs,validation_inputs, train_labels,validation_labels=train_test_split(sent_id,labels,random_state=2018, test_size=0.1
# Splitting masks
train_mask,validation_mask,_,_= train_test_split(attention_mask,labels,random_state=2018,test_size=0.1,stratify=labels)
```

By converting the data and masks into Torch tensors, you make them compatible with PyTorch and can use them as inputs to a BERT-based model for training and evaluation. This is a common practice in deep learning with PyTorch.

In [ ]:
```
# Converting all inputs and labels into torch tensors which is the required datatype for the BERT model

train_inputs=torch.tensor(train_inputs)
train_labels=torch.tensor(train_labels)
train_mask=torch.tensor(train_mask)

validation_inputs=torch.tensor(validation_inputs)
validation_labels=torch.tensor(validation_labels)
validation_mask=torch.tensor(validation_mask)
```

In [ ]:
```
validation_inputs
```

Out[146]:
```
tensor([[ 101, 2061, 2008,  ..., 2138, 1997,  102],
        [ 101, 2821, 1012,  ...,    0,    0,    0],
        [ 101, 2339, 2052,  ...,    0,    0,    0],
        ...,
        [ 101, 4283, 2005,  ...,    0,    0,    0],
        [ 101, 7632, 2045,  ..., 1018, 2847,  102],
```

By setting up these data loaders, you can efficiently iterate through your training and validation data in batches, which is important for training and evaluating deep learning models like BERT-based models. It helps improve training speed and memory efficiency.

In [ ]:
```
# batch size
batch_size=64

# Creating Tensor Dataset for training data
train_data=TensorDataset(train_inputs,train_mask,train_labels)

# Defining a random sampler during training
train_sampler=RandomSampler(train_data)

# Creating iterator using DataLoader. This iterator supports batching, customized data loading order
train_dataloader=DataLoader(train_data,sampler=train_sampler,batch_size=batch_size )

# Creating tensor dataset for validation data
validation_data=TensorDataset(validation_inputs,validation_mask,validation_labels)

# Defining a sequential sampler during validation, bcz there is no need to shuffle the data. We just need to validate
validation_sampler=SequentialSampler(validation_data)

# Create an iterator over validation dataset
validation_dataloader=DataLoader(validation_data,sampler=validation_sampler,batch_size=batch_size)
```

By using the iterator and **next**(), you can efficiently load and process batches of data during training. This is a common practice when training deep learning models with PyTorch, as it allows you to work with manageable batches of data while training the model.

In [ ]:
```
# Create an iterator object
iterator=iter(train_dataloader)

# Loads batch data
sent_id,mask,target=iterator.__next__()
```

In [ ]:
```
sent_id.shape
```

Out[149]:
```
torch.Size([64, 28])
```

In [ ]:
```
outputs=bert(sent_id,attention_mask=mask)
```

By examining the shapes of these tensors, you can understand the structure of the BERT model's output and use it for downstream tasks like text classification or feature extraction. The hidden states contain rich contextual information about the input text, which is valuable for various natural language processing tasks.

In [ ]:
```
hidden_states=outputs[0]
CLS_hidden_state=outputs[1]

print("Shape of Hidden States:",hidden_states.shape)
print("Shape of CLS Hidden State:",CLS_hidden_state.shape)
```

```
Shape of Hidden States: torch.Size([64, 28, 768])
Shape of CLS Hidden State: torch.Size([64, 768])
```

In [ ]:
```
type(hidden_states)
```

Out[152]:
```
torch.Tensor
```

#Fine-Tuning BERT

Freezing parameters is a common technique when you want to fine-tune a pre-trained model for a specific task. By keeping the pre-trained weights fixed, you can prevent them from being changed and focus on training only the additional layers or components added for the specific task. This can be beneficial when you have limited labeled data and you want to leverage the knowledge learned by the pre-trained model.

In [ ]:
```
# turn off the gradient of all parameters

for param in bert.parameters():
  param.requires_grad=False
```

This custom classifier is designed to work with BERT embeddings, perform classification, and return class probabilities. You can use this classifier to train and evaluate a text classification model using BERT embeddings.

In [ ]:

```
class classifier(nn.Module):

    #define the layers and wrappers used by model
    def __init__(self, bert):

      #constructor
      super(classifier, self).__init__()

      #bert model
      self.bert = bert

      # dense layer 1
      self.fc1 = nn.Linear(768,512)

      #dense layer 2 (Output layer)
      self.fc2 = nn.Linear(512,3)

      #dropout layer
      self.dropout = nn.Dropout(0.1)

      #relu activation function
      self.relu =  nn.ReLU()

      #softmax activation function
      self.softmax = nn.LogSoftmax(dim=1)

    #define the forward pass
    def forward(self, sent_id, mask):

      #pass the inputs to the model
      all_hidden_states, cls_hidden_state = self.bert(sent_id, attention_mask=mask, return_dict=False)

      #pass CLS hidden state to dense layer
      x = self.fc1(cls_hidden_state)

      #Apply ReLU activation function
      x = self.relu(x)

      #Apply Dropout
      x = self.dropout(x)

      #pass input to the output layer
      x = self.fc2(x)

      #apply softmax activation
      x = self.softmax(x)

      return x
```

By moving the model to the GPU, you can take advantage of GPU acceleration when training and making predictions, which is particularly beneficial for deep learning models with large neural architectures like BERT.

In [ ]:

```
# create the model
model=classifier(bert)

# push the model to GPU, if available
model=model.to(device)
```

This architecture represents a typical neural network setup for text classification, where BERT embeddings are used as features, and dense layers are employed for classification. The model takes input data in the form of integer sequences and attention masks, processes it through the layers, and produces class probabilities as output.

You can further train, fine-tune, or use this model for text classification tasks based on your specific requirements.

In [ ]:

```
# model arcitecture
model
```

Out[157]:
```
classifier(
  (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()
    )
  )
  (fc1): Linear(in_features=768, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=3, bias=True)
  (dropout): Dropout(p=0.1, inplace=False)
  (relu): ReLU()
  (softmax): LogSoftmax(dim=1)
)
```

In [ ]:
```
type(sent_id)
```

Out[158]:
```
torch.Tensor
```

By moving these tensors to the GPU, you can take full advantage of the GPU's parallel processing capabilities when training and making predictions with your deep learning model. This is a common practice for improving the training speed of deep learning models, especially for models with a significant number of parameters like BERT-based models.

In [ ]:
```
# push the tensors to GPU
sent_id=sent_id.to(device)
mask=mask.to(device)
target=target.to(device)
```

The outputs variable typically contains the model's predictions or class probabilities for the input data. These predictions can then be used to compute a loss and perform backpropagation to update the model's parameters during training.

In [ ]:
```
# pass inputs to the model
outputs=model(sent_id,mask)
```

In [ ]:
```
outputs=outputs.to(device)
```

In [ ]:

```
print(outputs)
```

```
tensor([[-1.2232, -1.0100, -1.0744],
        [-1.1630, -1.1001, -1.0368],
        [-1.2991, -0.9757, -1.0490],
        [-1.1139, -1.0148, -1.1737],
        [-1.2009, -1.0234, -1.0797],
        [-1.1697, -1.0318, -1.0991],
        [-1.0872, -1.0524, -1.1592],
        [-1.2265, -0.9934, -1.0895],
        [-1.1732, -1.0778, -1.0490],
        [-1.1801, -1.0813, -1.0397],
        [-1.1764, -1.0488, -1.0752],
        [-1.1602, -1.0256, -1.1148],
        [-1.2054, -1.0620, -1.0366],
        [-1.1465, -1.0620, -1.0892],
        [-1.2220, -1.0083, -1.0773],
        [-1.2599, -0.9961, -1.0584],
        [-1.2138, -1.0896, -1.0035],
        [-1.1712, -1.1177, -1.0134],
        [-1.2372, -1.0056, -1.0672],
        [-1.1331, -1.1503, -1.0178],
        [-1.2231, -1.0425, -1.0409],
        [-1.1484, -1.1201, -1.0311],
        [-1.2305, -0.9841, -1.0963],
        [-1.2644, -0.9668, -1.0868],
        [-1.1275, -1.0897, -1.0793],
        [-1.1852, -1.0842, -1.0324],
        [-1.1931, -1.0517, -1.0574],
        [-1.1857, -1.0545, -1.0611],
        [-1.2097, -1.0274, -1.0677],
        [-1.1815, -1.0492, -1.0701],
        [-1.1104, -1.0503, -1.1371],
        [-1.1696, -1.0490, -1.0811],
        [-1.2009, -1.0319, -1.0708],
        [-1.1560, -1.0016, -1.1458],
        [-1.2123, -1.0040, -1.0905],
        [-1.2090, -1.0525, -1.0428],
        [-1.2269, -1.0298, -1.0506],
        [-1.1871, -1.0303, -1.0846],
        [-1.1873, -1.0200, -1.0955],
        [-1.1152, -1.0546, -1.1275],
        [-1.1781, -1.0283, -1.0950],
        [-1.1709, -1.0207, -1.1099],
        [-1.1761, -1.0342, -1.0906],
        [-1.1532, -1.0311, -1.1154],
        [-1.0658, -1.0251, -1.2147],
        [-1.0293, -1.0305, -1.2521],
        [-1.2047, -1.0635, -1.0356],
        [-1.2065, -1.0597, -1.0379],
        [-1.1798, -1.0853, -1.0360],
        [-1.1391, -1.0284, -1.1323],
        [-1.2485, -1.0418, -1.0210],
        [-1.2462, -1.0245, -1.0401],
        [-1.1619, -1.0094, -1.1312],
        [-1.1140, -1.0380, -1.1470],
        [-1.1645, -1.0683, -1.0662],
        [-1.2286, -0.9953, -1.0856],
        [-1.1400, -1.0602, -1.0972],
        [-1.1500, -1.0331, -1.1164],
        [-1.1303, -1.0630, -1.1038],
        [-1.2427, -1.0062, -1.0619],
        [-1.1400, -1.0412, -1.1173],
        [-1.1931, -1.0781, -1.0315],
        [-1.1844, -1.0609, -1.0558],
        [-1.1058, -1.0606, -1.1307]], device='cuda:0',
       grad_fn=<LogSoftmaxBackward0>)
```

Knowing the number of trainable parameters is important because it can help you understand the model's complexity and the computational resources required for training and inference. It's especially crucial for optimizing model performance and resource utilization.

In [ ]:

```
# no. of trainable parameters
def count_parameters(model):
  return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 395,267 trainable parameters
```

The Adam optimizer is a popular choice for training deep learning models. It adapts the learning rate for each parameter based on their past gradients, which can lead to faster convergence and better training performance.

Once you have set up the optimizer, you can use it during the training process to update the model's parameters using the gradients computed during backpropagation.

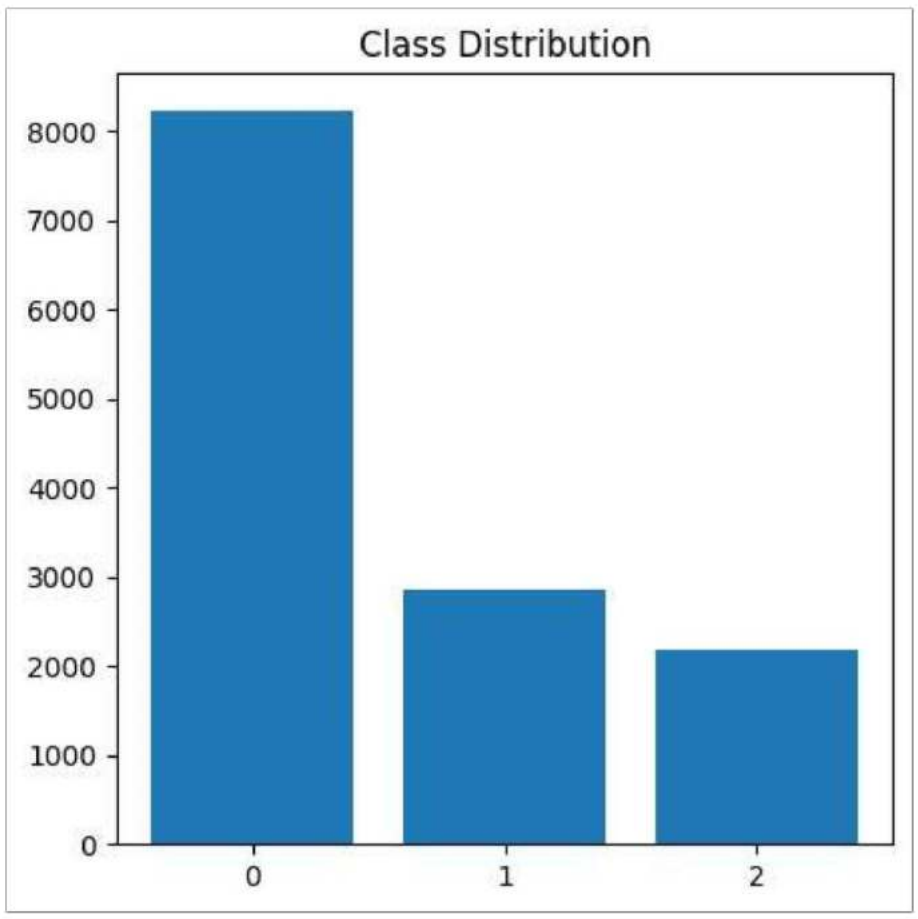
In [ ]:

```
# Adam optmizer
optimizer=torch.optim.Adam(model.parameters(),lr=0.0005)
```

This bar chart is a common way to visualize the distribution of classes in a dataset, especially in a classification task. It helps you understand the balance or imbalance between classes and assess the distribution of data across different categories.

In [ ]:
```
# Understnding class distribution

keys=['0','1','2']

# set figure size
plt.figure(figsize=(5,5))

# plot bar chart
plt.bar(keys,class_counts)

# set title
plt.title('Class Distribution')
```

Out[165]: Text(0.5, 1.0, 'Class Distribution')



The class weights are typically used during model training to adjust the loss function, giving more weight to underrepresented classes and less weight to overrepresented classes, thus ensuring fair and accurate learning.

In [ ]:
```
# library for array processing
import numpy as np

# computing the class weights
class_weights=compute_class_weight(class_weight='balanced',classes=np.unique(labels),y=labels)
print("Class Weights:",class_weights)
```

```
Class Weights: [0.53746864 1.55302233 2.01807961]
```

Using class weights in the loss function is a common practice in scenarios where there is a class imbalance in the dataset, and it helps the model effectively learn from all classes, regardless of their frequency.

In [ ]:
```
# Converting a list of class weights into a tensor
weights=torch.tensor(class_weights, dtype=torch.float)

# transferring weights to GPU
weights=weights.to(device)

# define the loss function
cross_entropy=nn.NLLLoss(weight=weights)
```

The computed loss is a critical metric during the training process, and it serves as a guide for adjusting the model's parameters to improve its performance. Reducing the loss typically leads to better model accuracy and effectiveness in the classification task.

In [ ]:
```
# Computing the loss
print(target)
#print(outputs)
loss=cross_entropy(outputs,target)
print('Loss: ',loss)
```

```
tensor([0, 1, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 0, 1, 0, 2, 0, 0,
        0, 0, 0, 1, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 2, 2, 1, 0, 0, 1, 2, 1, 0, 1,
        0, 2, 1, 2, 0, 2, 2, 0, 2, 2, 0, 0, 0, 1, 0, 2], device='cuda:0')
Loss:  tensor(1.1130, device='cuda:0', grad_fn=<NllLossBackward0>)
```

In [ ]:

```
# Function for computing time in hh:mm:ss

def format_time(elapsed):

  elapsed_rounded=int(round(elapsed))

  # format intp hh:mm:ss
  return str(datetime.timedelta(seconds=elapsed_rounded))
```

This function is essential for training your model and tracking the training progress. It calculates the loss, performs backpropagation, and updates the model's parameters for each batch, ensuring that the model learns from the training data. Additionally, it collects the model's predictions, which can be useful for further analysis and evaluation.

In [ ]:

```
# Defining a training function for the model:

def train():
  print('\n Training')

  # set the model on training phase- Dropout layers are activated
  model.train()
  # recording current time
  t0=time.time()
  # initialize the loss and accuracy to 0
  total_loss,total_accuracy=0,0

  # Create an empty list to save the model prediction
  total_preds=[]

  # for every batch
  for step, batch in enumerate(train_dataloader):
    #Progress update after every 40 batches
    if step % 40==0 and not step==0:
      elapsed=format_time(time.time()-t0)         # Calculate elapsed time in minutes
      print(' Batch{:>5,} of {:>5,}. Elapsed: {:}.'.format(step,len(train_dataloader),elapsed)) # Print progress
    batch=tuple(t.to(device) for t in batch)      # push the batch to GPU

    # batch is a part of all the records in train_dataloader. It contains 3 pytorch tensors:
    # [0]: input ids
    # [1]: attention masks
    # [2]: labels

    sent_id,mask,labels=batch

    #Pytorch doesn't automatically clear previously calculated gradients, hence before performing a backward pass, clear previou
    model.zero_grad()

    # Perform a forward pass. This returns the model predictions
    preds=model(sent_id,mask)

    # Compute the loss between actual and predicted values
    loss=cross_entropy(preds,labels)

    #Accumulate training loss over all the batches, so that we can calculate the average loss at the end
    # loss is a tensor containing a single value.
    # .itme() method just returns the Python value from the tensor

    total_loss=total_loss+loss.item()

    # Perform backward pass to calculate the gradients
    loss.backward()
    # During backward pass, information about parameter changes flows backwards, from the output to the hidden layers to the inp

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

    # The model predictions are stored on GPU, so push it to CPU
    preds=preds.detach().cpu().numpy()

    # Accumulate model predicitons of each batch
    total_preds.append(preds)



    # Compute the training loss of an epoch
  avg_loss=total_loss/len(train_dataloader)

  # The prediction are in the form of (no. of batches, size of batch, no. of classes)
  # So we need to resahpe the predictions in the form of number of samples x number of classes

  total_preds=np.concatenate(total_preds, axis=0)

  return avg_loss,total_preds
```

#EVALUATION

In [ ]:
```python
# define a function for evaluating the model

def evaluate():
  print("'n Evaluating....")

  # set the model on validation phase. Here dropout layers are deactivated
  model.eval()

  # record the current time
  t0=time.time()

  # initialize loss and accuracy to 0
  total_loss, total_accuracy=0,0

  # Create an empty list to save model predicitons
  total_preds=[]

  # for each batch

  for step, batch in enumerate(validation_dataloader):
    if step%40==0 and not step ==0:
      elapsed=format_time(time.time()-t0)
      print('  Batch {:>5,}  of  {:>5,}.    Elapsed: {:}.'.format(step, len(validation_dataloader), elapsed))

    batch=tuple(t.to(device) for t in batch)
    sent_id,mask,labels=batch

    #deactivate autograd
    with torch.no_grad():

      preds=model(sent_id,mask)
      loss=cross_entropy(preds,labels)
      total_loss=total_loss+loss.item()
      preds=preds.detach().cpu().numpy()
      total_preds.append(preds)

  avg_loss=total_loss/len(validation_dataloader)

  total_preds=np.concatenate(total_preds,axis=0)

  return avg_loss,total_preds
```

This function is crucial for evaluating your model's performance on data it hasn't seen during training. It calculates the loss and collects the model's predictions, which can be used to assess how well the model generalizes to unseen data.

In [ ]:

```
#define a function for evaluating the model
def evaluate():

  print("\nEvaluating.....")

  #set the model on training phase - Dropout layers are deactivated
  model.eval()

  #record the current time
  t0 = time.time()

  #initialize the loss and accuracy to 0
  total_loss, total_accuracy = 0, 0

  #Create a empty list to save the model predictions
  total_preds = []

  #for each batch
  for step,batch in enumerate(validation_dataloader):

    # Progress update every 40 batches.
    if step % 40 == 0 and not step == 0:

      # Calculate elapsed time in minutes.
      elapsed = format_time(time.time() - t0)

      # Report progress.
      print('  Batch {:>5,}  of  {:>5,}.    Elapsed: {:}.'.format(step, len(validation_dataloader), elapsed))

  #push the batch to gpu
  batch = tuple(t.to(device) for t in batch)

  #unpack the batch into separate variables
  # `batch` contains three pytorch tensors:
  #   [0]: input ids
  #   [1]: attention masks
  #   [2]: labels
  sent_id, mask, labels = batch

  #deactivates autograd
  with torch.no_grad():

    # Perform a forward pass. This returns the model predictions
    preds = model(sent_id, mask)

    #compute the validation loss between actual and predicted values
    loss = cross_entropy(preds,labels)

    # Accumulate the validation loss over all of the batches so that we can
    # calculate the average loss at the end. `loss` is a Tensor containing a
    # single value; the `.item()` function just returns the Python value
    # from the tensor.
    total_loss = total_loss + loss.item()

    #The model predictions are stored on GPU. So, push it to CPU
    preds=preds.detach().cpu().numpy()

    #Accumulate the model predictions of each batch
    total_preds.append(preds)

  #compute the validation loss of a epoch
  avg_loss = total_loss / len(validation_dataloader)

  #The predictions are in the form of (no. of batches, size of batch, no. of classes).
  #So, reshaping the predictions in form of (number of samples, no. of classes)
  total_preds  = np.concatenate(total_preds, axis=0)

  return avg_loss, total_preds
```

#TRAIN MODEL

This code demonstrates how to train a model for a specified number of epochs while tracking and saving the best model based on the validation loss. The training and validation losses are recorded for each epoch, providing insights into how the model's performance changes during training.

In [ ]:
```
# Assign the initial loss to infinite
best_valid_loss=float('inf')

# Create an empty list to store training and validation loss of each epoch
train_losses=[]
valid_losses=[]

epochs=5

#for each epoch repeat call the train() method
for epoch in range(epochs):
  print('\n ............epoch {:} / {:} .......'.format(epoch + 1, epochs))

  #train model
  train_loss,_ =train()

  #evaluate model
  valid_loss,_=evaluate()

  # save the best model
  if valid_loss<best_valid_loss:
    best_valid_loss=valid_loss
    torch.save(model.state_dict(),'Saved_weights.pt')

  # Accumulate training and validaion loss
  train_losses.append(train_loss)
  valid_losses.append(valid_loss)

  print(f'\nTraining Loss: {train_loss:.3f}')
  print(f'Validation Loss: {valid_loss:.3f}')

print("")
print("Training complete!")
```

```
 ............epoch 1 / 5 .......

 Training
 Batch   40 of   187. Elapsed: 0:00:04.
 Batch   80 of   187. Elapsed: 0:00:08.
 Batch  120 of   187. Elapsed: 0:00:12.
 Batch  160 of   187. Elapsed: 0:00:16.

Evaluating.....

Training Loss: 0.990
Validation Loss: 0.910

 ............epoch 2 / 5 .......

 Training
 Batch   40 of   187. Elapsed: 0:00:04.
 Batch   80 of   187. Elapsed: 0:00:09.
 Batch  120 of   187. Elapsed: 0:00:13.
 Batch  160 of   187. Elapsed: 0:00:17.

Evaluating.....

Training Loss: 0.824
Validation Loss: 0.775

 ............epoch 3 / 5 .......

 Training
 Batch   40 of   187. Elapsed: 0:00:04.
 Batch   80 of   187. Elapsed: 0:00:09.
 Batch  120 of   187. Elapsed: 0:00:14.
 Batch  160 of   187. Elapsed: 0:00:18.

Evaluating.....

Training Loss: 0.774
Validation Loss: 0.722

 ............epoch 4 / 5 .......

 Training
 Batch   40 of   187. Elapsed: 0:00:04.
 Batch   80 of   187. Elapsed: 0:00:09.
 Batch  120 of   187. Elapsed: 0:00:14.
 Batch  160 of   187. Elapsed: 0:00:18.

Evaluating.....

Training Loss: 0.746
Validation Loss: 0.771

 ............epoch 5 / 5 .......

 Training
 Batch   40 of   187. Elapsed: 0:00:04.
 Batch   80 of   187. Elapsed: 0:00:09.
 Batch  120 of   187. Elapsed: 0:00:13.
 Batch  160 of   187. Elapsed: 0:00:18.

Evaluating.....

Training Loss: 0.744
Validation Loss: 0.698

Training complete!
```

#EVALUATE MODEL

After running this code, your model will be loaded with the weights that were determined to be the best during training. This is useful when you want to use the trained model for making predictions on new data or for further evaluation.

In [ ]:
```
  # load weights of best model
path='Saved_weights.pt'
model.load_state_dict(torch.load(path))
```

Out[177]:
```
<All keys matched successfully>
```

The validation loss indicates how well the model is performing on data that it hasn't seen during training. A lower validation loss typically indicates better model performance. The preds variable contains the model's predicted values for the validation data, which can be used for further analysis or evaluation tasks.

In [ ]:
```
# get the model prediction on the validation data
valid_loss, preds=evaluate()
# this returns 2 elements- Validation loss and prediction
print(valid_loss)
```

```
Evaluating.....
0.698055747009459
```

Now, the y_pred variable contains the class labels predicted by the model, and the y_true variable contains the actual class labels. You can use these two arrays to evaluate the model's classification performance, for example, by calculating accuracy, precision, recall, or any other relevant metrics.

In [ ]:
```
# Converting the log(probabilities) into class & then choosing index of maximum value as class
y_pred=np.argmax(preds,axis=1)

# actual labels
y_true=validation_labels
```

In [ ]:
```
print(classification_report(y_true,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.88      0.77      0.82       824
           1       0.51      0.55      0.53       285
           2       0.57      0.80      0.66       220

    accuracy                           0.72      1329
   macro avg       0.65      0.70      0.67      1329
weighted avg       0.75      0.72      0.73      1329
```

THANK YOU