In this notebook, You will do amazon review classification with BERT.[Download data from [this (https://www.kaggle.com/snap/amazon-fine-food-reviews/data)](https://www.kaggle.com/snap/amazon-fine-food-reviews/data) link]

It contains 5 parts as below. Detailed instrctions are given in the each cell. please read every comment we have written.

1. Preprocessing
2. Creating a BERT model from the Tensorflow HUB.
3. Tokenization
4. getting the pretrained embedding Vector for a given review from the BERT.
5. Using the embedding data apply NN and classify the reviews.
6. Creating a Data pipeline for BERT Model.

## instructions:

1. Don't change any Grader Functions. Don't manipulate any Grader functions. If you manipulate any, it will be considered as plagiarised.

2. Please read the instructions on the code cells and markdown cells. We wil l explain what to write.

3. please return outputs in the same format what we asked. Eg. Don't return List if we are asking for a numpy array.

4. Please read the external links that we are given so that you will learn t he concept behind the code that you are writing.

5. We are giving instructions at each section if necessary, please follow th em.

## Every Grader function has to return True.

In [ ]:

```
!pip install sentencepiece
```

```
Collecting sentencepiece
  Downloading https://files.pythonhosted.org/packages/f5/99/e0808cb947ba10
f575839c43e8fafc9cc44e4a7a2c8f79c60db48220a577/sentencepiece-0.1.95-cp37-c
p37m-manylinux2014_x86_64.whl (1.2MB)
     |████████████████████████████████| 1.2MB 13.4MB/s
Installing collected packages: sentencepiece
Successfully installed sentencepiece-0.1.95
```

In [ ]:

```
#all imports
import numpy as np
import pandas as pd
import tensorflow as tf
import tensorflow_hub as hub
import math
from tensorflow.keras.models import Model

%matplotlib inline
import warnings
warnings.filterwarnings("ignore")


import nltk
import string
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn.metrics import roc_curve, auc
from nltk.stem.porter import PorterStemmer

import re
# Tutorial about Python regular expressions: https://pymotw.com/2/re/
import string
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.stem.wordnet import WordNetLemmatizer

from gensim.models import Word2Vec
from gensim.models import KeyedVectors
import pickle

from tqdm import tqdm
import os
from collections import Counter
from bs4 import BeautifulSoup

from sklearn.metrics import roc_auc_score, accuracy_score
```

In [ ]:

```
tf.test.gpu_device_name()
```

Out[ ]:

```
'/device:GPU:0'
```

## Grader function 1

In [ ]:

```
def grader_tf_version():
    assert((tf.__version__)>'2')
    return True
grader_tf_version()
```

Out[ ]:

```
True
```

# Part-1: Preprocessing

In [ ]:

```
!gdown --id "1GsD8JlAc_0yJ-1151LNr6rLw83RRUPgt"
!gdown --id "1QwjqTsqTX2vdy7fTmeXjxP3dq8IAVLpo"
```

```
Downloading...
From: https://drive.google.com/uc?id=1GsD8JlAc_0yJ-1151LNr6rLw83RRUPgt
To: /content/Reviews.csv
301MB [00:03, 76.6MB/s]
Downloading...
From: https://drive.google.com/uc?id=1QwjqTsqTX2vdy7fTmeXjxP3dq8IAVLpo
To: /content/test.csv
100% 62.1k/62.1k [00:00<00:00, 24.2MB/s]
```

In [ ]:

```
#Read the dataset - Amazon fine food reviews
reviews = pd.read_csv(r"Reviews.csv")
#check the info of the dataset
reviews.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 568454 entries, 0 to 568453
Data columns (total 10 columns):
 #   Column                  Non-Null Count   Dtype
---  ------                  --------------   -----
 0   Id                      568454 non-null  int64
 1   ProductId               568454 non-null  object
 2   UserId                  568454 non-null  object
 3   ProfileName             568438 non-null  object
 4   HelpfulnessNumerator    568454 non-null  int64
 5   HelpfulnessDenominator  568454 non-null  int64
 6   Score                   568454 non-null  int64
 7   Time                    568454 non-null  int64
 8   Summary                 568427 non-null  object
 9   Text                    568454 non-null  object
dtypes: int64(5), object(5)
memory usage: 43.4+ MB
```

In [ ]:

```
#get only 2 columns - Text, Score
#drop the NAN values1
reviews = reviews[["Text","Score"]]
reviews = reviews.dropna()
```

In [ ]:

```
#if score> 3, set score = 1
#if score<=2, set score = 0
#if score == 3, remove the rows.
def scoreFilter(score):
  if score > 3:
    return int(1)
  elif score <= 2:
    return int(0)

reviews = reviews[reviews.Score != 3]
reviews["Score"] = reviews["Score"].apply(lambda score:scoreFilter(score))
```

## Grader function 2

In [ ]:

```
def grader_reviews():
    temp_shape = (reviews.shape == (525814, 2)) and (reviews.Score.value_counts()[1]==4
43777)
    assert(temp_shape == True)
    return True
grader_reviews()
```

Out[ ]:

```
True
```

In [ ]:

```
def get_wordlen(x):
    return len(x.split())
reviews['len'] = reviews.Text.apply(get_wordlen)
reviews = reviews[reviews.len<50]
reviews = reviews.sample(n=100000, random_state=30)
```

In [ ]:

```
#remove HTML from the Text column and save in the Text column only
reviews["Text"] = reviews["Text"].apply(lambda x: BeautifulSoup(x, "lxml").text)
```

In [ ]:

```
#print head 5
reviews.head(5)
```

Out[ ]:

| | Text | Score | len |
|---|---|---|---|
| **64117** | The tea was of great quality and it tasted lik... | 1 | 30 |
| **418112** | My cat loves this. The pellets are nice and s... | 1 | 31 |
| **357829** | Great product. Does not completely get rid of ... | 1 | 41 |
| **175872** | This gum is my favorite! I would advise every... | 1 | 27 |
| **178716** | I also found out about this product because of... | 1 | 22 |

In [ ]:

```
#split the data into train and test data(20%) with Stratify sampling, random state 33,
from sklearn.model_selection import train_test_split
X = reviews.drop(columns=["Score"]).values
y = reviews["Score"].values
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=33, test_size=0.
2, stratify=y)
```

In [ ]:

```
print(Counter(y_train))
print(Counter(y_test))
```

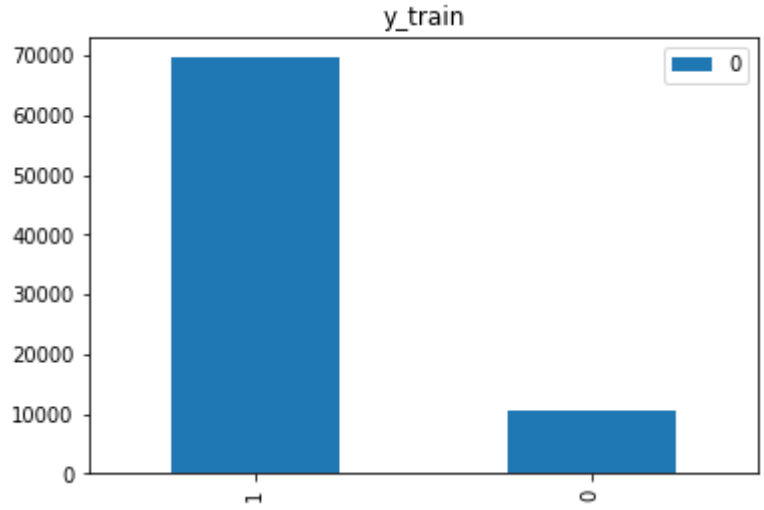
```
Counter({1: 69603, 0: 10397})
Counter({1: 17401, 0: 2599})
```

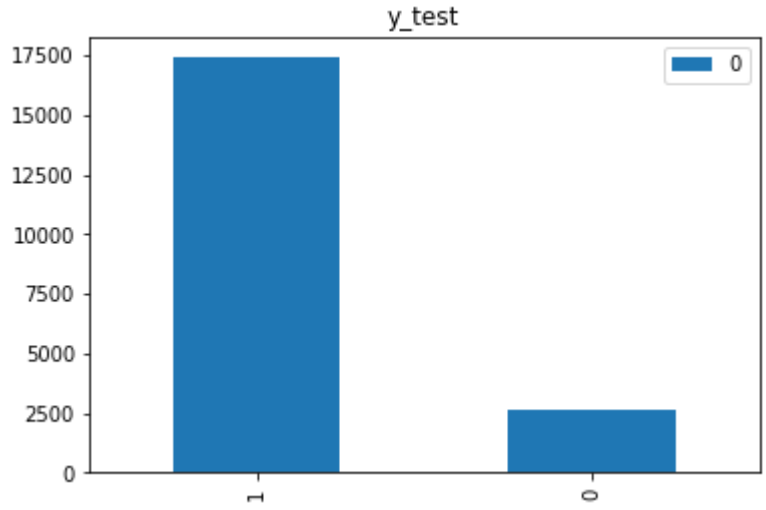
In [ ]:

```
#plot bar graphs of y_train and y_test

df = pd.DataFrame.from_dict(Counter(y_train), orient='index')
df.plot(kind='bar')
plt.title("y_train")
plt.show()

df = pd.DataFrame.from_dict(Counter(y_test), orient='index')
df.plot(kind='bar')
plt.title("y_test")
plt.show()
```




In [ ]:

```
#saving to disk. if we need, we can load preprocessed data directly.
reviews.to_csv('preprocessed.csv', index=False)
```

# Part-2: Creating BERT Model

If you want to know more about BERT, You can watch live sessions on Transformers and BERt. we will strongly recommend you to read Transformers (https://jalammar.github.io/illustrated-transformer/), BERT Paper (https://arxiv.org/abs/1810.04805) and, This blog (https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/).

For this assignment, we are using BERT uncased Base model (https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/1). It uses L=12 hidden layers (i.e., Transformer blocks), a hidden size of H=768, and A=12 attention heads. </pre>

In [ ]:

```
## Loading the Pretrained Model from tensorflow HUB
tf.keras.backend.clear_session()

# maximum length of a seq in the data we have, for now i am making it as 55. You can ch
ange this
max_seq_length = 55

#BERT takes 3 inputs

#this is input words. Sequence of words represented as integers
input_word_ids = tf.keras.layers.Input(shape=(max_seq_length,), dtype=tf.int32, name="i
nput_word_ids")

#mask vector if you are padding anything
input_mask = tf.keras.layers.Input(shape=(max_seq_length,), dtype=tf.int32, name="input
_mask")

#segment vectors. If you are giving only one sentence for the classification, total seg
vector is 0.
#If you are giving two sentenced with [sep] token separated, first seq segment vectors
 are zeros and
#second seq segment vector are 1's
segment_ids = tf.keras.layers.Input(shape=(max_seq_length,), dtype=tf.int32, name="segm
ent_ids")

#bert layer
bert_layer = hub.KerasLayer("https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-
12/1", trainable=False)
pooled_output, sequence_output = bert_layer([input_word_ids, input_mask, segment_ids])

#Bert model
#We are using only pooled output not sequence out.
#If you want to know about those, please read https://www.kaggle.com/questions-and-answ
ers/86510
bert_model = Model(inputs=[input_word_ids, input_mask, segment_ids], outputs=pooled_out
put)
```

In [ ]:

```
bert_model.summary()
```

```
Model: "model"
__________________________________________________________________________
_____________________
Layer (type)                    Output Shape         Param #     Connected
to
==========================================================================
=======================
input_word_ids (InputLayer)     [(None, 55)]         0
__________________________________________________________________________
_____________________
input_mask (InputLayer)         [(None, 55)]         0
__________________________________________________________________________
_____________________
segment_ids (InputLayer)        [(None, 55)]         0
__________________________________________________________________________
_____________________
keras_layer (KerasLayer)        [(None, 768), (None, 109482241   input_wor
d_ids[0][0]
                                                                 input_mas
k[0][0]
                                                                 segment_i
ds[0][0]
==========================================================================
=======================
Total params: 109,482,241
Trainable params: 0
Non-trainable params: 109,482,241
__________________________________________________________________________
_____________________
```

In [ ]:

```
bert_model.output
```

Out[ ]:

```
<KerasTensor: shape=(None, 768) dtype=float32 (created by layer 'keras_lay
er')>
```

# Part-3: Tokenization

In [ ]:

```
#getting Vocab file
vocab_file = bert_layer.resolved_object.vocab_file.asset_path.numpy()
do_lower_case = bert_layer.resolved_object.do_lower_case.numpy()
```

In [ ]:

```
vocab_file
```

Out[ ]:

```
b'/tmp/tfhub_modules/03d6fb3ce1605ad9e5e9ed5346b2fb9623ef4d3d/assets/voca
b.txt'
```

In [ ]:

```
#import tokenization - We have given tokenization.py file
import tokenization
```

In [ ]:

```
# Create tokenizer " Instantiate FullTokenizer"
# name must be "tokenizer"
# the FullTokenizer takes two parameters 1. vocab_file and 2. do_lower_case
# we have created these in the above cell ex: FullTokenizer(vocab_file, do_lower_case )
# please check the "tokenization.py" file the complete implementation
tokenizer = tokenization.FullTokenizer(vocab_file,do_lower_case)
```

## Grader function 3

In [ ]:

```
#it has to give no error
def grader_tokenize(tokenizer):
    out = False
    try:
        out=('[CLS]' in tokenizer.vocab) and ('[SEP]' in tokenizer.vocab)
    except:
        out = False
    assert(out==True)
    return out
grader_tokenize(tokenizer)
```

Out[ ]:

```
True
```

In [ ]:

```
len(X_train)
for i in range(80000):
  if len(X_train[i][0].split(" ")) > 55:
    print(i)
    break
```

```
20421
```

In [ ]:

```
# Create train and test tokens (X_train_tokens, X_test_tokens) from (X_train, X_test) u
sing Tokenizer and
def create_tokens(text):
  tokens = tokenizer.tokenize(text)
  segment = [0]*55
  # mask = [0]*10
    # maximum number of tokens is 55(We already given this to BERT layer above) so shap
e is (None, 55)
    # if it is less than 55, add '[PAD]' token else truncate the tokens length.(similar
to padding)
  if len(tokens) == max_seq_length-2:
    tokens = tokens[:max_seq_length-2]
    mask = [1]*max_seq_length

  if len(tokens) > max_seq_length-2:    # TRUNCATE
    tokens = tokens[:max_seq_length-2]
    mask = [1]*max_seq_length

  if len(tokens) < max_seq_length-2:  # PAD
    pad_length = ((max_seq_length-2)-len(tokens))
    mask = [1]*(len(tokens)+2) + [0]*pad_length # (len+2)*0 + 1*rem [len+2 because toke
ns+cls+seq]
    tokens = tokens + ['[PAD]']*pad_length

  tokens = ['[CLS]', *tokens ,'[SEP]']

  # positional encoding
  tokens = np.array(list(map(lambda x : tokenizer.vocab[x],tokens)))
  mask = mask
  segment = segment
  return tokens,mask,segment

def create_token_dataset(data):
  token_list, msk_list, segment_list = [], [], []
  for row in data:
    tkn, msk, seg = create_tokens(row[0])
    token_list.append(tkn)
    msk_list.append(msk)
    segment_list.append(seg)

  return np.array(token_list), np.array(msk_list), np.array(segment_list)


X_train_tokens, X_train_mask, X_train_segment = create_token_dataset(X_train)
X_test_tokens, X_test_mask, X_test_segment = create_token_dataset(X_test)

# Based on padding, create the mask for Train and Test ( 1 for real token, 0 for '[PA
D]'),
# it will also same shape as input tokens (None, 55) save those in X_train_mask, X_test
_mask

# Create a segment input for train and test. We are using only one sentence so all zero
s. This shape will also (None, 55)

# type of all the above arrays should be numpy arrays

# after execution of this cell, you have to get
# X_train_tokens, X_train_mask, X_train_segment
# X_test_tokens, X_test_mask, X_test_segment
```

In [ ]:

```
import pickle
```

In [ ]:

```
##save all your results to disk so that, no need to run all again.
pickle.dump((X_train, X_train_tokens, X_train_mask, X_train_segment, y_train),open('train_data.pkl','wb'))
pickle.dump((X_test, X_test_tokens, X_test_mask, X_test_segment, y_test),open('test_data.pkl','wb'))
```

In [ ]:

```
#you can load from disk
#X_train, X_train_tokens, X_train_mask, X_train_segment, y_train = pickle.load(open("train_data.pkl", 'rb'))
#X_test, X_test_tokens, X_test_mask, X_test_segment, y_test = pickle.load(open("test_data.pkl", 'rb'))
```

## Grader function 4

In [ ]:

```
def grader_alltokens_train():
    out = False

    if type(X_train_tokens) == np.ndarray:

        temp_shapes = (X_train_tokens.shape[1]==max_seq_length) and (X_train_mask.shape[1]==max_seq_length) and \
        (X_train_segment.shape[1]==max_seq_length)

        segment_temp = not np.any(X_train_segment)

        mask_temp = np.sum(X_train_mask==0) == np.sum(X_train_tokens==0)

        no_cls = np.sum(X_train_tokens==tokenizer.vocab['[CLS]'])==X_train_tokens.shape[0]

        no_sep = np.sum(X_train_tokens==tokenizer.vocab['[SEP]'])==X_train_tokens.shape[0]

        out = temp_shapes and segment_temp and mask_temp and no_cls and no_sep

    else:
        print('Type of all above token arrays should be numpy array not list')
        out = False
    assert(out==True)
    return out

grader_alltokens_train()
```

Out[ ]:

```
True
```

## Grader function 5

In [ ]:

```
def grader_alltokens_test():
    out = False
    if type(X_test_tokens) == np.ndarray:

        temp_shapes = (X_test_tokens.shape[1]==max_seq_length) and (X_test_mask.shape[1
]==max_seq_length) and \
        (X_test_segment.shape[1]==max_seq_length)

        segment_temp = not np.any(X_test_segment)

        mask_temp = np.sum(X_test_mask==0) == np.sum(X_test_tokens==0)

        no_cls = np.sum(X_test_tokens==tokenizer.vocab['[CLS]'])==X_test_tokens.shape[0
]


        no_sep = np.sum(X_test_tokens==tokenizer.vocab['[SEP]'])==X_test_tokens.shape[0
]


        out = temp_shapes and segment_temp and mask_temp and no_cls and no_sep

    else:
        print('Type of all above token arrays should be numpy array not list')
        out = False
    assert(out==True)
    return out
grader_alltokens_test()
```

Out[ ]:

```
True
```

# Part-4: Getting Embeddings from BERT Model

We already created the BERT model in the part-2 and input data in the part-3. We will utlize those two and will get the embeddings for each sentence in the Train and test data.</pre>

In [ ]:

```
bert_model.input
```

Out[ ]:

```
[<KerasTensor: shape=(None, 55) dtype=int32 (created by layer 'input_word_
ids')>,
 <KerasTensor: shape=(None, 55) dtype=int32 (created by layer 'input_mas
k')>,
 <KerasTensor: shape=(None, 55) dtype=int32 (created by layer 'segment_id
s')>]
```

In [ ]:

```
bert_model.output
```

Out[ ]:

```
<KerasTensor: shape=(None, 768) dtype=float32 (created by layer 'keras_lay
er')>
```

In [ ]:

```
# get the train output, BERT model will give one output so save in
# X_train_pooled_output
X_train_pooled_output=bert_model.predict([X_train_tokens,X_train_mask,X_train_segment])
```

In [ ]:

```
# get the test output, BERT model will give one output so save in
# X_test_pooled_output
X_test_pooled_output=bert_model.predict([X_test_tokens,X_test_mask,X_test_segment])
```

In [ ]:

```
##save all your results to disk so that, no need to run all again.
# pickle.dump((X_train_pooled_output, X_test_pooled_output),open('final_output.pkl','w
b'))
```

In [ ]:

```
# !gdown --id "15lru1ykPTGWuHqBI9goV127RYnl5lp_k"
# X_train_pooled_output, X_test_pooled_output= pickle.load(open('final_output.pkl', 'r
b'))
```

```
Downloading...
From: https://drive.google.com/uc?id=15lru1ykPTGWuHqBI9goV127RYnl5lp_k
To: /content/final_output.pkl
307MB [00:01, 263MB/s]
```

In [ ]:

```
X_train_pooled_output.shape
```

Out[ ]:

```
(80000, 768)
```

## Grader function 6

In [ ]:

```
#now we have X_train_pooled_output, y_train
#X_test_pooled_ouput, y_test

#please use this grader to evaluate
def greader_output():
    assert(X_train_pooled_output.shape[1]==768)
    assert(len(y_train)==len(X_train_pooled_output))
    assert(X_test_pooled_output.shape[1]==768)
    assert(len(y_test)==len(X_test_pooled_output))
    assert(len(y_train.shape)==1)
    assert(len(X_train_pooled_output.shape)==2)
    assert(len(y_test.shape)==1)
    assert(len(X_test_pooled_output.shape)==2)
    return True
greader_output()
```

Out[ ]:

```
True
```

# Part-5: Training a NN with 768 features

Create a NN and train the NN. 1. **You have to use AUC as metric.**

1. You can use any architecture you want.
2. You have to use tensorboard to log all your metrics and Losses. You have to send those logs.
3. Print the loss and metric at every epoch.
4. You have to submit without overfitting and underfitting. </pre>

In [ ]:

```
##imports
from tensorflow.keras.layers import Input, Dense, Activation, Dropout, Flatten
from tensorflow.keras.models import Model
import datetime

%load_ext tensorboard

!rm -rf ./logs
log_dir="logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1
, write_graph=True)
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

In [ ]:

```
# ##create an NN and
input = Input(shape=(768))
dense1 = Dense(512,activation='relu')(input)
dense3 = Dense(128,activation='relu')(dense1)
drop2 = Dropout(0.25)(dense3)
dense4 = Dense(64,activation='relu')(drop2)
dense5 = Dense(32,activation='relu')(dense4)
output = Dense(1,activation='sigmoid')(dense5)
```

In [ ]:

```
model = Model(input,output)
```

In [ ]:

```
model.compile(optimizer=tf.keras.optimizers.Adam(lr=0.001),loss="binary_crossentropy",m
etrics=["accuracy"])
```

In [ ]:

```
class Metrics_Callback(tf.keras.callbacks.Callback):
  def __init__(self,x_val,y_val):
    self.x_val = x_val
    self.y_val = y_val

  def on_train_begin(self, logs={}):
    self.history = {"auc_score":[]}

  def on_epoch_end(self, epoch, logs={}):
    y_pred = self.model.predict(self.x_val)
    auc_score = roc_auc_score(self.y_val,y_pred)
    self.history["auc_score"].append(auc_score)
    print(' val_auc_score: ',auc_score)

auc_callback = Metrics_Callback(X_test_pooled_output,y_test)
```

In [ ]:

```
model.fit(X_train_pooled_output,y_train,
          epochs=50,
          batch_size=64,
          validation_data=(X_test_pooled_output,y_test),
          callbacks=[auc_callback])
```

```
Epoch 1/50
1250/1250 [==============================] - 4s 3ms/step - loss: 0.3306 -
accuracy: 0.8772 - val_loss: 0.2607 - val_accuracy: 0.8936
 val_auc_score:  0.9021875282406164
Epoch 2/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2482 -
accuracy: 0.8995 - val_loss: 0.2280 - val_accuracy: 0.9086
 val_auc_score:  0.9142916872516139
Epoch 3/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2358 -
accuracy: 0.9029 - val_loss: 0.2436 - val_accuracy: 0.9037
 val_auc_score:  0.9018239366951155
Epoch 4/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2269 -
accuracy: 0.9077 - val_loss: 0.2340 - val_accuracy: 0.9123
 val_auc_score:  0.9200900033629481
Epoch 5/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2201 -
accuracy: 0.9115 - val_loss: 0.2144 - val_accuracy: 0.9137
 val_auc_score:  0.9246574570075412
Epoch 6/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2199 -
accuracy: 0.9119 - val_loss: 0.2249 - val_accuracy: 0.9146
 val_auc_score:  0.9243068007285055
Epoch 7/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2151 -
accuracy: 0.9127 - val_loss: 0.2262 - val_accuracy: 0.9083
 val_auc_score:  0.9255489400942161
Epoch 8/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2187 -
accuracy: 0.9110 - val_loss: 0.2071 - val_accuracy: 0.9180
 val_auc_score:  0.9310637859216496
Epoch 9/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2108 -
accuracy: 0.9144 - val_loss: 0.2046 - val_accuracy: 0.9176
 val_auc_score:  0.9324471186959288
Epoch 10/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2140 -
accuracy: 0.9140 - val_loss: 0.2185 - val_accuracy: 0.9114
 val_auc_score:  0.9297069428041654
Epoch 11/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2115 -
accuracy: 0.9133 - val_loss: 0.2523 - val_accuracy: 0.8970
 val_auc_score:  0.9313628227484417
Epoch 12/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2034 -
accuracy: 0.9157 - val_loss: 0.2095 - val_accuracy: 0.9182
 val_auc_score:  0.9266902286046326
Epoch 13/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2047 -
accuracy: 0.9175 - val_loss: 0.2149 - val_accuracy: 0.9111
 val_auc_score:  0.929448149028598
Epoch 14/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2044 -
accuracy: 0.9183 - val_loss: 0.2060 - val_accuracy: 0.9168
 val_auc_score:  0.9313873444758087
Epoch 15/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2045 -
accuracy: 0.9164 - val_loss: 0.2037 - val_accuracy: 0.9232
 val_auc_score:  0.9351206392701557
Epoch 16/50
```

```
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2035 -
accuracy: 0.9166 - val_loss: 0.2124 - val_accuracy: 0.9141
 val_auc_score:  0.933495494403463
Epoch 17/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2035 -
accuracy: 0.9167 - val_loss: 0.2022 - val_accuracy: 0.9186
 val_auc_score:  0.9361055437257446
Epoch 18/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.2030 -
accuracy: 0.9169 - val_loss: 0.1994 - val_accuracy: 0.9194
 val_auc_score:  0.9361527298088838
Epoch 19/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1998 -
accuracy: 0.9186 - val_loss: 0.1985 - val_accuracy: 0.9208
 val_auc_score:  0.9358966115328757
Epoch 20/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1979 -
accuracy: 0.9197 - val_loss: 0.2026 - val_accuracy: 0.9173
 val_auc_score:  0.9368504381815987
Epoch 21/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1953 -
accuracy: 0.9192 - val_loss: 0.2063 - val_accuracy: 0.9208
 val_auc_score:  0.9352905777153131
Epoch 22/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1976 -
accuracy: 0.9194 - val_loss: 0.1969 - val_accuracy: 0.9236
 val_auc_score:  0.935445336569995
Epoch 23/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1946 -
accuracy: 0.9218 - val_loss: 0.2184 - val_accuracy: 0.9143
 val_auc_score:  0.9366285596664814
Epoch 24/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1973 -
accuracy: 0.9197 - val_loss: 0.2240 - val_accuracy: 0.9093
 val_auc_score:  0.9321717854685394
Epoch 25/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1951 -
accuracy: 0.9201 - val_loss: 0.2357 - val_accuracy: 0.9032
 val_auc_score:  0.9232044396311003
Epoch 26/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1955 -
accuracy: 0.9198 - val_loss: 0.1957 - val_accuracy: 0.9222
 val_auc_score:  0.9386123873108884
Epoch 27/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1930 -
accuracy: 0.9221 - val_loss: 0.1914 - val_accuracy: 0.9229
 val_auc_score:  0.9407367781842154
Epoch 28/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1954 -
accuracy: 0.9200 - val_loss: 0.2007 - val_accuracy: 0.9200
 val_auc_score:  0.9357305426118744
Epoch 29/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1899 -
accuracy: 0.9235 - val_loss: 0.2104 - val_accuracy: 0.9162
 val_auc_score:  0.938470431053272
Epoch 30/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1912 -
accuracy: 0.9240 - val_loss: 0.2042 - val_accuracy: 0.9176
 val_auc_score:  0.9380638988454203
Epoch 31/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1928 -
```

```
accuracy: 0.9210 - val_loss: 0.1978 - val_accuracy: 0.9209
 val_auc_score:  0.9400515739024167
Epoch 32/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1917 -
accuracy: 0.9229 - val_loss: 0.1936 - val_accuracy: 0.9227
 val_auc_score:  0.9380109748107466
Epoch 33/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1877 -
accuracy: 0.9257 - val_loss: 0.1993 - val_accuracy: 0.9234
 val_auc_score:  0.9334270259374647
Epoch 34/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1913 -
accuracy: 0.9220 - val_loss: 0.2067 - val_accuracy: 0.9143
 val_auc_score:  0.9357852797950099
Epoch 35/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1892 -
accuracy: 0.9247 - val_loss: 0.1951 - val_accuracy: 0.9252
 val_auc_score:  0.9389515455752886
Epoch 36/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1865 -
accuracy: 0.9257 - val_loss: 0.1943 - val_accuracy: 0.9201
 val_auc_score:  0.9390947179690684
Epoch 37/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1880 -
accuracy: 0.9237 - val_loss: 0.2094 - val_accuracy: 0.9196
 val_auc_score:  0.9402820427611607
Epoch 38/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1869 -
accuracy: 0.9249 - val_loss: 0.2200 - val_accuracy: 0.9211
 val_auc_score:  0.9393981107744823
Epoch 39/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1859 -
accuracy: 0.9251 - val_loss: 0.2143 - val_accuracy: 0.9219
 val_auc_score:  0.9406000955352347
Epoch 40/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1847 -
accuracy: 0.9261 - val_loss: 0.1961 - val_accuracy: 0.9247
 val_auc_score:  0.9420003547137514
Epoch 41/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1842 -
accuracy: 0.9263 - val_loss: 0.2041 - val_accuracy: 0.9154
 val_auc_score:  0.9407376184237464
Epoch 42/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1841 -
accuracy: 0.9253 - val_loss: 0.2251 - val_accuracy: 0.9089
 val_auc_score:  0.9396649310487277
Epoch 43/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1843 -
accuracy: 0.9257 - val_loss: 0.2024 - val_accuracy: 0.9227
 val_auc_score:  0.9367205216719997
Epoch 44/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1838 -
accuracy: 0.9254 - val_loss: 0.1909 - val_accuracy: 0.9241
 val_auc_score:  0.9416954472660253
Epoch 45/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1827 -
accuracy: 0.9272 - val_loss: 0.1951 - val_accuracy: 0.9212
 val_auc_score:  0.9418865575362091
Epoch 46/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1799 -
accuracy: 0.9292 - val_loss: 0.1985 - val_accuracy: 0.9220
```

```
 val_auc_score:  0.9420448321299814
Epoch 47/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1778 -
accuracy: 0.9282 - val_loss: 0.1941 - val_accuracy: 0.9218
 val_auc_score:  0.9420066896775844
Epoch 48/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1847 -
accuracy: 0.9249 - val_loss: 0.2053 - val_accuracy: 0.9175
 val_auc_score:  0.9305378733656872
Epoch 49/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1800 -
accuracy: 0.9282 - val_loss: 0.1998 - val_accuracy: 0.9190
 val_auc_score:  0.9403370121157456
Epoch 50/50
1250/1250 [==============================] - 3s 3ms/step - loss: 0.1780 -
accuracy: 0.9297 - val_loss: 0.2180 - val_accuracy: 0.9154
 val_auc_score:  0.9412586332677054
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fec40dfe090>
```

In [ ]:

```
auc_callback.history
```

Out[ ]:


```
{'auc_score': [0.9021875282406164,
  0.9142916872516139,
  0.9018239366951155,
  0.9200900033629481,
  0.9246574570075412,
  0.9243068007285055,
  0.9255489400942161,
  0.9310637859216496,
  0.9324471186959288,
  0.9297069428041654,
  0.9313628227484417,
  0.9266902286046326,
  0.929448149028598,
  0.9313873444758087,
  0.9351206392701557,
  0.933495494403463,
  0.9361055437257446,
  0.9361527298088838,
  0.9358966115328757,
  0.9368504381815987,
  0.9352905777153131,
  0.935445336569995,
  0.9366285596664814,
  0.9321717854685394,
  0.9232044396311003,
  0.9386123873108884,
  0.9407367781842154,
  0.9357305426118744,
  0.938470431053272,
  0.9380638988454203,
  0.9400515739024167,
  0.9380109748107466,
  0.9334270259374647,
  0.9357852797950099,
  0.9389515455752886,
  0.9390947179690684,
  0.9402820427611607,
  0.9393981107744823,
  0.9406000955352347,
  0.9420003547137514,
  0.9407376184237464,
  0.9396649310487277,
  0.9367205216719997,
  0.9416954472660253,
  0.9418865575362091,
  0.9420448321299814,
  0.9420066896775844,
  0.9305378733656872,
  0.9403370121157456,
  0.9412586332677054]}
```

In [ ]:

```
%tensorboard --logdir logs/fit
```

In [ ]:

```
# model.save("BERT_NN_24_11:39")
# new_m = tf.keras.models.load_model("BERT_NN_24_11:39")
# new_m.summary()

# !zip -r "BERT_NN_24_11:39.zip" "BERT_NN_24_11:39"

# !cp "BERT_NN_24_11:39.zip" "/content/drive/MyDrive/Datasets"
```

In [ ]:

```

```

# Part-6: Creating a Data pipeline for BERT Model

1. Download data from here (https://drive.google.com/file/d/1QwjqTsqTX2vdy7fTmeXjxP3dq8IAVLpo/view?usp=sharing)
2. Read the csv file
3. Remove all the html tags
4. Now do tokenization [Part 3 as mentioned above]
   - Create tokens,mask array and segment array
5. Get Embeddings from BERT Model [Part 4 as mentioned above] , let it be X_test
   - Print the shape of output(X_test.shape).You should get (352,768)
6. Predit the output of X_test with the Neural network model which we trained earlier.
7. Print the occurences of class labels in the predicted output

</pre>

In [ ]:

```
!gdown --id "1QwjqTsqTX2vdy7fTmeXjxP3dq8IAVLpo"
```

```
Downloading...
From: https://drive.google.com/uc?id=1QwjqTsqTX2vdy7fTmeXjxP3dq8IAVLpo
To: /content/test.csv
100% 62.1k/62.1k [00:00<00:00, 23.4MB/s]
```

In [ ]:

```
def predict_output(test_file_name):
  # read test_file
  df_test = pd.read_csv(test_file_name)

  # Remove all the html tags
  df_test["Text"] = df_test["Text"].apply(lambda x: BeautifulSoup(x, "lxml").text)

  #tokenization
  X_test_data_tokens, X_test_data_mask, X_test_data_segment = create_token_dataset(df_t
est["Text"].values)

  # Get Embeddings from BERT Model
  X_test = bert_model.predict([X_test_data_tokens, X_test_data_mask, X_test_data_segmen
t])
  print("Shape of embedings from BERT model: ",X_test.shape)

  # predict using the trained neural network
  y_prediction = model.predict(X_test)

  # distribution of predictions
  y_pred_classes = np.where(y_prediction >0.5, 1, 0)
  res = Counter(list([x[0] for x in y_pred_classes]))
  print("Distribution of predictions: ",res)

  df = pd.DataFrame.from_dict(res, orient='index')
  df.plot(kind='bar')
  plt.title("y_test")
  plt.show()

  return y_prediction
```

In [ ]:

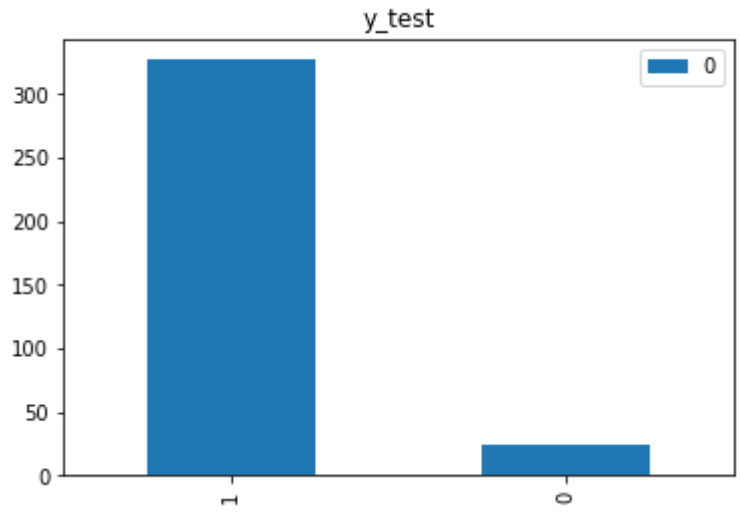
```

```

In [ ]:

```
result = predict_output("test.csv")
```

```
Shape of embedings from BERT model:  (352, 768)
Distribution of predictions:  Counter({1: 327, 0: 25})
```



In [ ]:

```
!jupyter nbconvert --to html "BERT_Assignment_Final.ipynb"
```

```
[NbConvertApp] Converting notebook BERT_Assignment_Final.ipynb to html
[NbConvertApp] Writing 398351 bytes to BERT_Assignment_Final.html
```

In [ ]:

```

```

# Description of the workflow

## 1. PRE PROCESSING

- We pick only 2 columns for our task : Text, Score
- Scores range from 0-5, but here we convert it to 0 or 1. If score > 3 we consider it as 1, if its <3 we consider it 0 else we ignore the data row.
- To remove all html tags we use BeautifulSoup module. It removes all the html tags present in text
- We calculate one more feature which is len of each text sentence
- We then split the dataset into a 80-20 split

## 2. CREATING BERT MODEL

- We use BERT uncased model here which has hidden size of 768
- We use tfhub to load the BERT model. We make the bert layer frozen by applying trainable = False
- The bert layer takes 3 input
    1. word ids : vector of each words represented as integers
    2. segment_ids : segment id first sentence is 0, 1 for next sentence and so on. In our case we only have one sentence so it will be 1
    3. mask_vector : it indicates if we are padding
- The BERT layer gives an output of 768 dimensions.

## 3. TOKENIZATION

- From the trained BERT layer we get the vocab file and import tokenization from the tokenization.py file
- Steps for tokenization
    1. We convert the text into tokens
    2. We pad or truncate the tokens to make the token length as 55. If len(tokens) > maxLen + 2, then truncate else if len(tokens) < maxLen +2 then padding ( here +2 is used to incorporporate the [CLS] amd [SEP] tokens.
    3. Start of the tokens is indicated by [CLS] and ends with [SEP]
    4. In case padding was used we use the [PAD] token.
    5. Based on the paddings done create the mask array and the segment array
- We create the tokens, masks and segments for test and train data which acts like the Train and Test dataset

## 4. GETTING EMBEDDINGS FROM BERT MODEL

- We use the loaded bert model to make the predictions. The bert model gives 768 dimensional output
- After predictions we get train_pooled_output and test_pooled_output of size (len(data),768)
- We then use these embeddings as input to train our Neural Network which takes these embeddins as input and the predict the score for each text embedding
- The NN takes 768 features are input and predicts score. We use AUC as metric to evaluate our model.

## 5. FINAL PIPELINE FOR BERT MODEL

- The pipeline function takes input as as csv file which contains all the text and predicts the score (0 or 1)
- Steps in function:
    1. Read CSV
    2. Remove HTML tags
    3. Pick the relevant columns and get the BERT model embeddings
    4. Use the embeddings as input to the NN and get the predicted scores
    5. Print the distribution of predictions

In [ ]:

```
for i in range(1, 10):
  print(i)
  a = [0]*10
```