IMDB BERT Sentiment Classification

March 14, 2022

1 IMDB Sentiment Classification

1.0.1 Alexander Bricken

1.0.2 Minerva University

The goal of this notebook is to take a pre-trained recent NLP model and apply it to classify the sentiment of movie reviews. The data is collected from the Kaggle IMDB dataset.

The classification will be binary. Hence, either the movie review is 0 (bad) or 1 (good).

The approach I take includes the following: - Data exploration: using Pandas Profiling to examine for any missing rows, duplicates, and other anomalies in the dataset. - BertTokenization: I tokenise the text for using Bert to capture the most information in embeddings. This allows me to record attention and standardise the length of input. Converts my words to numbers for BERT to interpret. - Train the model: forward and backward pass (feeding the input through the model and backpropagating it). Use DistilBERT. Update parameters accordingly. Track variables to monitor progress. - Evaluate the model: Using validation dataset get accuracy and loss. Perform fine-tuning and test results using test dataset. Output descriptive statistics and ROC curve to evaluate.

1.0.3 Import Libraries

[1]:
```
!pip install wget
!pip install transformers
```

```
Requirement already satisfied: wget in /usr/local/lib/python3.7/dist-packages
(3.2)
Requirement already satisfied: transformers in /usr/local/lib/python3.7/dist-
packages (4.17.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.7/dist-
packages (from transformers) (1.21.5)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from transformers) (2.23.0)
Requirement already satisfied: sacremoses in /usr/local/lib/python3.7/dist-
packages (from transformers) (0.0.47)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.7/dist-
packages (from transformers) (21.3)
```

```
Requirement already satisfied: huggingface-hub<1.0,>=0.1.0 in
/usr/local/lib/python3.7/dist-packages (from transformers) (0.4.0)
Requirement already satisfied: tokenizers!=0.11.3,>=0.11.1 in
/usr/local/lib/python3.7/dist-packages (from transformers) (0.11.6)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-
packages (from transformers) (3.6.0)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/dist-
packages (from transformers) (4.63.0)
Requirement already satisfied: importlib-metadata in
/usr/local/lib/python3.7/dist-packages (from transformers) (4.11.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.7/dist-
packages (from transformers) (6.0)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.7/dist-packages (from transformers) (2019.12.20)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.7/dist-packages (from huggingface-
hub<1.0,>=0.1.0->transformers) (3.10.0.2)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from packaging>=20.0->transformers)
(3.0.7)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-
packages (from importlib-metadata->transformers) (3.7.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->transformers) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (2021.10.8)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (3.0.4)
Requirement already satisfied: click in /usr/local/lib/python3.7/dist-packages
(from sacremoses->transformers) (7.1.2)
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist-packages
(from sacremoses->transformers) (1.1.0)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages
(from sacremoses->transformers) (1.15.0)
```

[54]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import nltk
import seaborn as sns
import sys
import os
import re
import torch
import random
```

```python
import tensorflow as tf
import sklearn
from transformers import BertModel, BertConfig, BertTokenizer,␣
 ↪DistilBertForSequenceClassification, BertConfig, AdamW
from transformers import get_linear_schedule_with_warmup
from tensorflow import keras
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler
from torch.utils.data import TensorDataset, random_split
from sklearn.metrics import f1_score, confusion_matrix, plot_confusion_matrix,␣
 ↪classification_report, accuracy_score, roc_curve, auc
from pandas_profiling import ProfileReport
from tqdm import tqdm
import time
import datetime

# settings
sns.set_theme(style="whitegrid")
seed_val = 42
random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)
```

[3]:
```python
# GOOGLE COLLAB GPU CONNECTION CHECK
device_name = tf.test.gpu_device_name()
if device_name:
    print(f'GPU: {device_name}')
else:
    print("Device not found.")

# CUDA connection. Tell PyTorch to use GPU.
# taken from https://wandb.ai/wandb/common-ml-errors/reports/
 ↪How-To-Use-GPU-with-PyTorch---VmlldzozMzAxMDk
if torch.cuda.is_available():
    device = torch.device("cuda")
    print('GPU in use:', torch.cuda.get_device_name(0))
else:
    print('No GPU available, using the CPU instead.')
    device = torch.device("cpu")
```

```
GPU: /device:GPU:0
GPU in use: Tesla T4
```

1.0.4 Import Data

```
[4]: train_data = pd.read_csv('Train.csv')
     test_data = pd.read_csv('Test.csv')
     valid_data = pd.read_csv('Valid.csv')
```

```
[5]: print('number of train:', len(train_data))
     print('number of valid:', len(valid_data))
     print('number of test:', len(test_data))
```

```
number of train: 40000
number of valid: 5000
number of test: 5000
```

1.0.5 Data Exploration

```
[ ]: # can be used in jupyter notebook/lab only. not Collab.
     profile = ProfileReport(train_data, title="Pandas Profiling Report")
     profile.to_notebook_iframe()
```

Our labels are split pretty equal at 20019 “0s” and 19981 “1s”. We can see from the report that there are a 269 duplicate rows. Also, upon close examination of the general text itself, there are frequent $< br\ / >< br\ / >$ HTML tags that we will have to remove.

1.0.6 Data Cleaning

In this section, as discovered in the EDA, we need to do the following: - Delete duplicate rows - Remove $< br\ / >< br\ / >$ tagging - Lowercase everything

We don’t perform too much text cleaning, because BERT actually uses certain aspects of the text as contextual information. It is better, as explored in Bricken (2021), to only perform light cleaning when using a BERT model.

```
[6]: # remove duplicates
     train_data = train_data.drop_duplicates().reset_index(drop=True)
```

```
[7]: def text_clean(x):
         """
         A function using regex to remove unwanted pieces from the text for each␣
      ↪review.
         Input: a review.
         Output: the cleaned review.
         """
         # first we lowercase everything
         x = x.lower()
         # remove unicode characters
         x = x.encode('ascii', 'ignore').decode()
         # remove urls
         x = re.sub(r'https*\S+', ' ', x)
```

```
    x = re.sub(r'http*\S+', ' ', x)
    # remove <br></br> tags
    x = re.sub(r'<br \/><br \/>', ' ', x)
    # remove ' \ "
    x = re.sub(r"\\", "", x)
    x = re.sub(r"\'", "", x)
    x = re.sub(r"\"", "", x)
    # remove *
    x = re.sub(r"\*", "", x)
    return x
```

[8]:
```
train_data['text'] = train_data['text'].apply(text_clean)
test_data['text'] = test_data['text'].apply(text_clean)
valid_data['text'] = valid_data['text'].apply(text_clean)
```

[9]:
```
# output some results
list(train_data['text'].sample(3))
```

[9]:
```
['there are similarities between ray lawrences jindabyne and his last movie
lantana  a dead body and its repercussions for already dysfunctional lives. but
whereas lantana offered some hope and resolution, jindabyne leaves everything
unresolved in a bleak way that will leave most viewers unsatisfied, perhaps even
cheated. the storyline - the aftermath of a fishermans discovery of a corpse
floating in a remote river - is based on a short story by raymond carver. it
became an element in robert altmans classic 1993 ensemble short cuts. lawrence
uses this theme for an exploration and exposition of relationships within a
small australian community under stress. the movie poses some moral questions
would you let the discovery of a dead body ruin your good weekend? and more
poignantly for australians would it make any difference if the dead person was
an aboriginal? the acting, especially by gabriel byrne and laura linney, is
commendable. and there are elements of mysticism reinforced by haunting music,
not unlike picnic at hanging rock. if all this sounds like the basis for a great
movie - be prepared for a let down, the pace is very slow and the murder is
shown near the beginning, thereby eliminating the element of mystery. and so we
are left with these desolate lives and a blank finale.',
 'hammer house of horror: witching time is set in rural england on woodstock
farm where stressed musician david winter (jon finch) lives with his actress
wife mary (prunella gee) & is currently composing the music for a horror film.
one night while looking for his dog billy david finds a mysterious woman in his
barn, calling herself lucinda jessop (patricia quinn) she claims to be a witch
who has transported herself from 300 years in the past to now. obviously rather
sceptical david has a hard time believing her so he locks her in a room in his
farmhouse & calls his doctor charles (ian mcculloch) to come examine her,
however once he arrives & they enter the room lucinda has disappeared. charles
puts it down to david drinking too much but over the next few day strange &
disturbing things begin to happen to david & mary… witching time was episode 1
from the short lived british anthology horror series produced by hammer studios
for tv & originally aired here in the uk during september 1980, the first of two
hammer house of horror episodes to be directed by don leaver (episode 13 the
mark of satan being the other) i actually rather liked this. as a series hammer
house of horror dealt with various different themes & were all unconnected to
each other except in name & unsurprisingly watching time is a sinister &
effective little tale about a witch, the script by anthony read benefits from
its slight 50 odd minute duration & moves along at a nice pace. the characters
are pretty good as is the dialogue, there are some nice scenes here & i liked
the way it never quite reveals whether david & mary are going crazy or not. i
think its a well structured, entertaining & reasonably creepy horror themed tv
show that i enjoyed more than i thought i would. being made for british tv meant
the boys at hammer had a lower budget than usual, if that was even possible, &
as such there is no gorgeous period settings here as in their most well know
frankenstein & dracula films although the contemporary english setting does give
it a certain atmosphere that you can relate to a bit more. another tv based
restriction is that the exploitation levels are lower than you might hope for,
theres some nudity & gore but not much although i didnt mind too much as the
story here is pretty good. its well made for what it is & hammers experience on
their feature films probably helped make these look pretty good, the acting is
good as well with genre favourite ian mcculloch making a bit-part appearance.
witching time is a good start to the hammer house of horror series, as a 50
minute piece of british tv its pretty damned good, now why dont they make shows
like this over here anymore?',
 'what a great cast for this movie. the timing was excellent and there were so
many clever lines-several times i was still laughing minutes after they were
delivered. i found manna from heaven to have some surprising moments and while
there were things i was thinking would happen, the way they came together was
anything but predictable. this movie is about hope and righting wrongs. i left
the theater feeling inspired to do the right thing. bravo to the five sisters.']
```

1.0.7 Data Preparation: Tokenisation

[10]:
```python
PRE_TRAINED_MODEL_NAME = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(PRE_TRAINED_MODEL_NAME)
```

To start with, because BERT works with fixed-length sequences, we need to choose the maximum length of the sequences to best represent the model. By storing the length of each Tweet, we can do this and evaluate the coverage.

The maximum length for standard BERT is 512, so we can try that and see how much truncation we get (information loss should be recorded). However, as tested in retrospect, it is highly computationally expensive to increase the token length. As such, we restrain our truncation to 256 (half of 512) and test our model.

Realistically, we are losing quite a lot of information here, but for sake of demonstration this is fine. If we were optimising for accuracy, we would of course increase this length of tokens, use a better model of BERT, and do even more fine-tuning.

[11]:
```
# we run this cell if we are able to optimise for max length
# if majority of input strings are less than 512 in length
# but here we max out because most are above.
#
# token_lens = []
# for txt in list(train_data['text']):
#     tokens = tokenizer.encode(txt, max_length=MAX_LENGTH, truncation=True)
#     token_lens.append(len(tokens))
```

[12]:
```
# sns.displot(token_lens)
# plt.xlim([0, MAX_LENGTH])
# plt.xlabel('Token count')
# plt.show()

# we see that a lot of information is lost,
# but perhaps the model still will have enough for sentiment analysis
```

[13]:
```
MAX_LENGTH = 256
def bert_tokenizer(text):
    """
    Function that runs the necessary steps for BERT tokenization.
    Input: the text that we want to tokenize.
    Output: the associated encoded input_ids and attention mask.
    """
    encoding = tokenizer.encode_plus(
    text,
    max_length=MAX_LENGTH,
    truncation=True,
    return_token_type_ids=False,
    add_special_tokens=True, # Add '[CLS]' and '[SEP]'
    padding='max_length',
    return_attention_mask=True,
    return_tensors='pt',  # Return PyTorch tensors
    )
    return encoding['input_ids'][0], encoding['attention_mask'][0]
```

We add the following four important specifications to our BertTokeniser to improve the performance and make BERT feasible: - Add special tokens to the start and end of each sentence [SEP] and [CLS]. - We add [SEP] as an artifact of knowing when a sentence ends. - We add [CLS] to the beginnging of the sentence to make the model aware that we are performing a classification task. - Output the attention mask. This allows the model to differentiate between real tokens and padding tokens.

[14]:
```
# we shrink the data to 10000 train and
# 2000 val/test because 39723 for train is too computationally expensive
train_text = train_data['text']
# train data tokenization
train_tokenized_list = []
```

```
train_attn_mask_list = []
for text in list(train_text[0:10000]):
    tokenized_text, attn_mask = bert_tokenizer(text)
    train_tokenized_list.append(tokenized_text.numpy())
    train_attn_mask_list.append(attn_mask.numpy())
```

[15]:
```
test_text = test_data['text']
# test data tokenization
test_tokenized_list = []
test_attn_mask_list = []
for text in list(test_text[0:2000]):
    tokenized_text, attn_mask = bert_tokenizer(text)
    test_tokenized_list.append(tokenized_text.numpy())
    test_attn_mask_list.append(attn_mask.numpy())
```

[16]:
```
valid_text = valid_data['text']
# valid data tokenization
valid_tokenized_list = []
valid_attn_mask_list = []
for text in list(valid_text[0:2000]):
    tokenized_text, attn_mask = bert_tokenizer(text)
    valid_tokenized_list.append(tokenized_text.numpy())
    valid_attn_mask_list.append(attn_mask.numpy())
```

[17]:
```
# configure everything we have thus far
X_train = torch.tensor(train_tokenized_list)
X_test = torch.tensor(test_tokenized_list)
X_valid = torch.tensor(valid_tokenized_list)

# attention mask
attn_train = torch.tensor(train_attn_mask_list)
attn_test = torch.tensor(test_attn_mask_list)
attn_valid = torch.tensor(valid_attn_mask_list)

# labels
y_train = torch.tensor(train_data['label'][0:10000])
y_test = torch.tensor(test_data['label'][0:2000])
y_valid = torch.tensor(valid_data['label'][0:2000])
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: UserWarning:
Creating a tensor from a list of numpy.ndarrays is extremely slow. Please
consider converting the list to a single numpy.ndarray with numpy.array() before
converting to a tensor. (Triggered internally at
../torch/csrc/utils/tensor_new.cpp:201.)
```

```
[18]: print(f"Length of X_train: {len(X_train)}")
      print(f"Length of attn_train: {len(attn_train)}")
      print(f"Length of y_train: {len(y_train)} \n")
      print(f"Length of X_test: {len(X_test)}")
      print(f"Length of attn_test: {len(attn_test)}")
      print(f"Length of y_test: {len(y_test)} \n")
      print(f"Length of X_valid: {len(X_valid)}")
      print(f"Length of attn_valid: {len(attn_valid)}")
      print(f"Length of X_test: {len(y_valid)}")
```

```
Length of X_train: 10000
Length of attn_train: 10000
Length of y_train: 10000

Length of X_test: 2000
Length of attn_test: 2000
Length of y_test: 2000

Length of X_valid: 2000
Length of attn_valid: 2000
Length of X_test: 2000
```

```
[19]: # final configurations for input

      # Combine the inputs into a TensorDataset.
      train_dataset = TensorDataset(X_train, attn_train, y_train)

      # set up val dataset in the same way
      val_dataset = TensorDataset(X_valid, attn_valid, y_valid)
```

```
[20]: print(f'{len(train_dataset)} training samples')
      print(f'{len(val_dataset)} validation samples')
```

```
10000 training samples
2000 validation samples
```

1.0.8 Modelling

We start by making a dataloader. This way, we can save training memory by iterating through only some of the data as loaded memory instead of all at once. This is called batching.

```
[21]: batch_size = 16
      # batch size 16 as per https://wandb.ai/jack-morris/david-vs-goliath/reports/
       ↪Does-Model-Size-Matter-A-Comparison-of-BERT-and-DistilBERT--VmlldzoxMDUxNzU

      # initialise dataloader for train and validation datasets
      train_dataloader = DataLoader(train_dataset, sampler =␣
       ↪RandomSampler(train_dataset), batch_size = batch_size)
```

```
validation_dataloader = DataLoader(val_dataset, sampler =␣
 ↪SequentialSampler(val_dataset), batch_size = batch_size)
```

We use DistilBERT here for computational efficiency. It is smaller, faster, cheaper, and lighter, and still maintains up to 95% of the performance of normal BERT.

If we wanted to increase our model's performance, we would use a larger BERT, but for this purpose, as we have done in truncating token length and limiting the size of the dataset, we are trying to demonstrate the feasibility of the model and not necessarily achieve highest accuracy.

[22]:
```
bert_model_name = "distilbert-base-uncased"
model = DistilBertForSequenceClassification.from_pretrained(
    bert_model_name, # we use distilbert for computational efficiency.
    num_labels = 2, # binary classification
    output_attentions = False, # we don't need attention weights or hidden␣
 ↪states
    output_hidden_states = False,
)

# we initialise model to cuda (connection to GPU)
model.cuda()
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertForSequenceClassification:
['vocab_layer_norm.weight', 'vocab_transform.weight', 'vocab_projector.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias', 'vocab_transform.bias']
- This IS expected if you are initializing DistilBertForSequenceClassification
from the checkpoint of a model trained on another task or with another
architecture (e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing
DistilBertForSequenceClassification from the checkpoint of a model that you
expect to be exactly identical (initializing a BertForSequenceClassification
model from a BertForSequenceClassification model).
Some weights of DistilBertForSequenceClassification were not initialized from
the model checkpoint at distilbert-base-uncased and are newly initialized:
['classifier.weight', 'pre_classifier.bias', 'classifier.bias',
'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

[22]:
```
DistilBertForSequenceClassification(
  (distilbert): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0): TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
        (1): TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
        (2): TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
        (3): TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
        (4): TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
        (5): TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
      )
    )
  )
  (pre_classifier): Linear(in_features=768, out_features=768, bias=True)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

We use AdamW (Adaptive Moment Estimation) optimizer here because it works with momentums of first and second order. Essentially, it is a good optimizer for gradient descent because we don't want to jump over the minimum, we want to decrease the velocity a little bit for a careful search. AdamW does this.

[23]:
```
# we use AdamW optimizer.
optimizer = AdamW(model.parameters(),
                  lr = 3e-5, # our learning rate is initialised to be as␣
 ↪recommended for DistilBERT
                  eps = 1e-8 # keep eps normal.
                )
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309:
FutureWarning: This implementation of AdamW is deprecated and will be removed in
a future version. Use the PyTorch implementation torch.optim.AdamW instead, or
set `no_deprecation_warning=True` to disable this warning
  FutureWarning,
```

[30]:
```
# if we start with 3 epochs and examine our loss, we can figure out if
# we need early stopping, or need more or less epochs/warmups.
epochs = 3

# number of batches x number of epochs is the total steps we need to make
```

```
total_steps = len(train_dataloader) * epochs

# this initialises our learning schedule
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0,
                                            num_training_steps = total_steps)
```

[31]:
```
def accuracy_out(preds, labels):
    """
    Use argmax function to turn our logit probabilities into final␣
 ↪classification.
    Input: predictions as logit probabilities for each 1 or 0 classification␣
 ↪and true labels.
    Output: The accuracy of the predictions as a decimal between 0 and 1.
    """
    predictions = np.argmax(preds, axis=1).flatten() # take the argmax
    labelling = labels.flatten() # flatten the labels for each batch
    return np.sum(predictions == labelling) / len(labelling) # calculate␣
 ↪accuracy (number right) / total
```

[32]:
```
# initialise storage and time for training.
training_stats = []
total_t0 = time.time()
```

This next code cell is where we train and test out model on a validation set to record the performance on newly seen data for each epoch. By doing so, we can find the best number of epochs, check if the model is overfitting, and evaluate our hyperparameters etc.

[33]:
```
# pytorch training structure taken from:
# https://mccormickml.com/2019/07/22/BERT-fine-tuning/

for epoch_i in range(0, epochs):

    ################
    ### TRAINING ###
    ################

    print("")
    print(f'======== Epoch {epoch_i + 1} / {epochs} ========')
    print('Training...')

    # start timer
    t0 = time.time()
    # reset loss for this epoch
    total_train_loss = 0

    # model mode into train.
    # https://stackoverflow.com/questions/51433378/
↪what-does-model-train-do-in-pytorch
    model.train()

    # now go by batch within epochs.
    # we use the dataloader here
    for step, batch in enumerate(train_dataloader):
        # progress report
        if step % 50 == 0 and not step == 0:
            elapsed = time.time() - t0
            print(f"Batch {step} of {len(train_dataloader)}. Elapsed:␣
↪{round(elapsed, 2)} seconds.")

        # unpack and send each tensor to the GPU
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)

        # clear gradients before backward pass
        # https://stackoverflow.com/questions/48001598/
↪why-do-we-need-to-call-zero-grad-in-pytorch
        model.zero_grad()

        # call model() for forward pass
        # https://huggingface.co/transformers/model_doc/bert.
↪html#bertforsequenceclassification
        result = model(b_input_ids,
                       attention_mask=b_input_mask,
                       labels=b_labels,
                       return_dict=True)

        # we get the loss because we know true labels.
        # and the logits which are the outputs of the model
        # we just need to turn from log to normal probabilities
        # and then use argmax to find the predicted probability of each
        # classification 0 or 1 for sentiment classification
        loss = result.loss
        logits = result.logits

        # we need total training loss to then calc average loss across epochs
        total_train_loss += loss.item()

        # backprop
        loss.backward()

        # clip the norm of the gradients to 1.0 to help prevent the "exploding␣
↪gradients" problem
```

```
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # we use AdamW optimiser because parameters are modified based on their␣
↪gradients, the learning rate, etc.
        # this takes a step in the calculated gradient
        optimizer.step()

        # update learning rate
        scheduler.step()

    # calculate the average loss over all of the batches
    avg_train_loss = total_train_loss / len(train_dataloader)
    # measure how long this epoch took
    training_time = time.time() - t0

    print("\n")
    print("Average training loss:", avg_train_loss)
    print("Training epoch took: ", round(training_time, 2))

    ##################
    ### Validation ###
    ##################

    print("")
    print("Running Validation...")

    t0 = time.time()

    # we change the model to evaluation mode for validation (PyTorch)
    model.eval()

    # initialise accurancy, loss, and step count
    total_eval_accuracy = 0
    total_eval_loss = 0
    nb_eval_steps = 0

    # we use validation dataloader
    for batch in validation_dataloader:

        # again unpack pieces of data
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)

        # grad not needed for val (only for backprop training)
        with torch.no_grad():
            # run model forward pass
            result = model(b_input_ids,
                            attention_mask=b_input_mask,
                            labels=b_labels,
                            return_dict=True)

        # again output loss and logits for val dataset
        loss = result.loss
        logits = result.logits

        # sum up all losses
        total_eval_loss += loss.item()

        # move logits and labels to CPU
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()

        # calculate total accuracy using acc function
        total_eval_accuracy += accuracy_out(logits, label_ids)

    ####################
    ### EPOCH OUTPUT ###
    ####################

    # average valid dataset accuracy and loss calculations
    avg_val_accuracy = total_eval_accuracy / len(validation_dataloader)
    print("Validation Accuracy:", round(avg_val_accuracy, 2))
    avg_val_loss = total_eval_loss / len(validation_dataloader)
    print("Validation Loss: ", round(avg_val_loss, 2))

    # record the states of this epoch for performance analysis
    training_stats.append(
        {
            'epoch': epoch_i + 1,
            'Training Loss': avg_train_loss,
            'Valid. Loss': avg_val_loss,
            'Valid. Accur.': avg_val_accuracy,
            'Training Time': training_time,
        }
    )

print("Training complete.")

print("Total training taken (sec):", round(time.time() - total_t0, 2))
```

```
======== Epoch 1 / 3 ========
Training…
```

```
Batch 50 of 625. Elapsed: 20.24 seconds.
Batch 100 of 625. Elapsed: 40.24 seconds.
Batch 150 of 625. Elapsed: 59.29 seconds.
Batch 200 of 625. Elapsed: 78.47 seconds.
Batch 250 of 625. Elapsed: 98.21 seconds.
Batch 300 of 625. Elapsed: 117.94 seconds.
Batch 350 of 625. Elapsed: 137.37 seconds.
Batch 400 of 625. Elapsed: 156.8 seconds.
Batch 450 of 625. Elapsed: 176.49 seconds.
Batch 500 of 625. Elapsed: 196.08 seconds.
Batch 550 of 625. Elapsed: 215.59 seconds.
Batch 600 of 625. Elapsed: 235.02 seconds.


Average training loss: 0.2599819464504719
Training epoch took:  244.69

Running Validation…
Validation Accuracy: 0.91
Validation Loss:  0.24

======== Epoch 2 / 3 ========
Training…
Batch 50 of 625. Elapsed: 19.61 seconds.
Batch 100 of 625. Elapsed: 39.2 seconds.
Batch 150 of 625. Elapsed: 58.75 seconds.
Batch 200 of 625. Elapsed: 78.37 seconds.
Batch 250 of 625. Elapsed: 98.23 seconds.
Batch 300 of 625. Elapsed: 118.06 seconds.
Batch 350 of 625. Elapsed: 137.65 seconds.
Batch 400 of 625. Elapsed: 157.1 seconds.
Batch 450 of 625. Elapsed: 176.52 seconds.
Batch 500 of 625. Elapsed: 195.89 seconds.
Batch 550 of 625. Elapsed: 215.3 seconds.
Batch 600 of 625. Elapsed: 234.73 seconds.


Average training loss: 0.13168398306928575
Training epoch took:  244.47

Running Validation…
Validation Accuracy: 0.9
Validation Loss:  0.35

======== Epoch 3 / 3 ========
Training…
Batch 50 of 625. Elapsed: 19.54 seconds.
Batch 100 of 625. Elapsed: 38.97 seconds.
Batch 150 of 625. Elapsed: 58.43 seconds.
Batch 200 of 625. Elapsed: 77.85 seconds.
Batch 250 of 625. Elapsed: 97.32 seconds.
Batch 300 of 625. Elapsed: 116.81 seconds.
Batch 350 of 625. Elapsed: 136.35 seconds.
Batch 400 of 625. Elapsed: 155.98 seconds.
Batch 450 of 625. Elapsed: 175.53 seconds.
Batch 500 of 625. Elapsed: 195.07 seconds.
Batch 550 of 625. Elapsed: 214.67 seconds.
Batch 600 of 625. Elapsed: 234.23 seconds.


Average training loss: 0.05595344244129956
Training epoch took:  244.02

Running Validation…
Validation Accuracy: 0.9
Validation Loss:  0.45
Training complete.
Total training taken (sec): 786.87
```

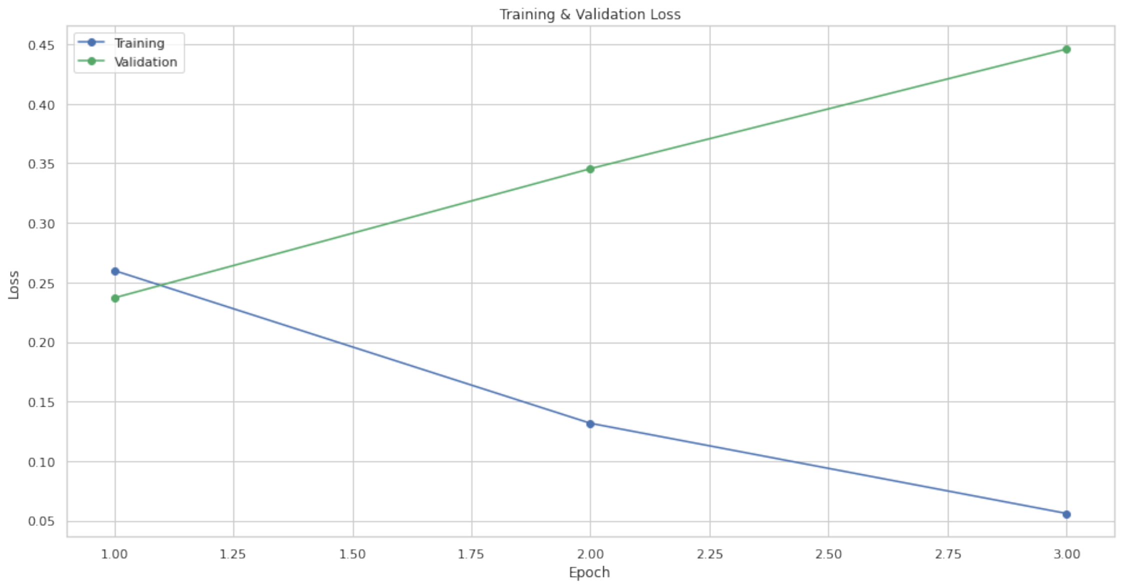
1.0.9 Model Evaluation

We need to evaluate the performance of our model by outputting the associated loss and accuracy across validation data for each epoch. By examining the overall scores and the trend over time, we can see if the model is overfitting and how well it is performing in general.

[34]:
```python
# training statistics dataframe
df_stats = pd.DataFrame(data=training_stats)
df_stats = df_stats.set_index('epoch')
df_stats
```

[34]:
```
       Training Loss  Valid. Loss  Valid. Accur.  Training Time
epoch
1           0.259982     0.236977         0.9130     244.687517
2           0.131684     0.345489         0.9025     244.468211
3           0.055953     0.445941         0.9045     244.015770
```

[35]:
```python
%matplotlib inline
# Plot the learning curve.
plt.figure(figsize=(16,8))
plt.plot(df_stats['Training Loss'], 'b-o', label="Training")
plt.plot(df_stats['Valid. Loss'], 'g-o', label="Validation")

# Label the plot.
plt.title("Training & Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



We can see immediately that the validation loss increases every epoch. This suggests that our model is overfitting: it is performing less well on unseen data as it is further trained. This corresponds to the better validation accuracy of 91.3% in the first epoch as well.

To solve for this, we should take the parameters associated with the first epoch. We can implement early stopping or save each model in order to do this. However, given we have only trained on 3 epochs and they are all relatively similar, for simplicity we will go ahead with the 3rd epoch model that is only slightly worse.

Overall, however, we are getting roughly 90% of our predictions right on the validation dataset. This is a strong performance and much better than just random guessing (which would be 50%). This performance is aided by the various batches we separate our data into and the tuning of our model as we progress through training. This is done via the AdamW optimizer and the linear learning rate scheduler.

We can evaluate this performance on an unseen test dataset below:

1.0.10 Test Set Performance

[36]:
```
# we have our test data from above that has alread been tokenized
# X_test, attn_test, y_test, and converted into tensors.

test_dataset = TensorDataset(X_test, attn_test, y_test)
test_sampler = SequentialSampler(test_dataset)
# we use same batch size as above
```

```
test_dataloader = DataLoader(test_dataset, sampler=test_sampler,␣
 ↪batch_size=batch_size)
```

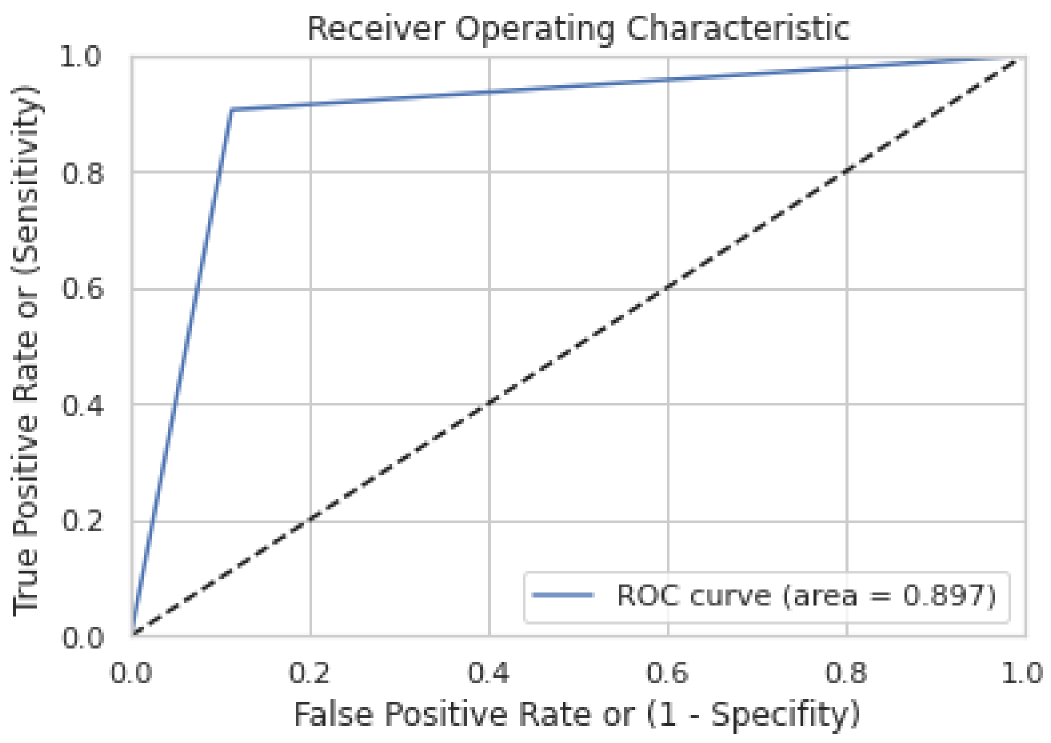
[38]:
```
# model in eval mode
model.eval()

# need storage for predictions and true labels
predictions = []
true_labels = []

print("Testing model on test data...")
for batch in test_dataloader:

    batch = tuple(t.to(device) for t in batch)
    # unpack
    b_input_ids, b_input_mask, b_labels = batch

    # because not doing backprop again, no grad calcs
    with torch.no_grad():
        # forward pass
        result = model(b_input_ids,
                       attention_mask=b_input_mask,
                       return_dict=True)

    # get logits
    logits = result.logits

    # move to cpu
    logits = logits.detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()

    # store
    predictions.append(logits)
    true_labels.append(label_ids)

print("Predictions output.")
print('Model testing done.')
```

```
DONE.
```

[53]:
```
# final prediction calc
final_pred_list = []
for batch in predictions:
    for j in batch:
        final_pred_list.append(np.argmax(j))

# flatten nested true label array
true_label_list = list(np.concatenate(true_labels).flat)
```

[56]:
```
# model test function
def eval_model(predictions, true_labels):
    print(accuracy_score(true_labels, predictions))
    # Compute fpr, tpr, thresholds and roc auc
    fpr, tpr, thresholds = roc_curve(np.array(true_labels), np.
 ↪array(predictions))
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.plot(fpr, tpr, label='ROC curve (area = %0.3f)' % roc_auc)
    plt.plot([0, 1], [0, 1], 'k--')  # random predictions curve
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.xlabel('False Positive Rate or (1 - Specifity)')
    plt.ylabel('True Positive Rate or (Sensitivity)')
    plt.title('Receiver Operating Characteristic')
    plt.legend(loc="lower right")
    plt.show()
    print(classification_report(true_labels, np.array(predictions),␣
 ↪target_names=["bad review", "good review"]))

eval_model(final_pred_list, true_label_list)
```

```
0.8965
```



```
              precision    recall  f1-score   support

  bad review       0.91      0.89      0.90      1036
 good review       0.88      0.91      0.89       964

    accuracy                           0.90      2000
   macro avg       0.90      0.90      0.90      2000
weighted avg       0.90      0.90      0.90      2000
```

[61]:
```
# example classification
print("Review:", test_data['text'][0])
print("True Label:", true_label_list[0])
print("Predicted Label:", final_pred_list[0])
print("Tokenisation: \n", X_test[0])
```

```
Review: i always wrote this series off as being a complete stink-fest because
jim belushi was involved in it, and heavily. but then one day a tragic
happenstance occurred. after a white sox game ended i realized that the remote
was all the way on the other side of the room somehow. now i could have just
gotten up and walked across the room to get the remote, or even to the tv to
turn the channel. but then why not just get up and walk across the country to
watch tv in another state? nuts to that, i said. so i decided to just hang tight
on the couch and take whatever fate had in store for me. what fate had in store
was an episode of this show, an episode about which i remember very little
except that i had once again made a very broad, general sweeping blanket
judgment based on zero objective or experiential evidence with nothing
whatsoever to back my opinions up with, and once again i was completely right!
this show is a total crud-pie! belushi has all the comedic delivery of a hairy
lighthouse foghorn. the women are physically attractive but too stepford-is to
elicit any real feeling from the viewer. there is absolutely no reason to stop
yourself from running down to the local tv station with a can of gasoline and a
flamethrower and sending every copy of this mutt howling back to hell.  except..
except for the wonderful comic sty lings of larry joe campbell, americas
greatest comic character actor. this guy plays belushis brother-in-law, andy,
and he is gold. how good is he really? well, aside from being funny, his job is
to make belushi look good. thats like trying to make butt warts look good. but
campbell pulls it off with style. someone should invent a nobel prize in comic
buffoonery so he can win it every year. without larry joe this show would
consist of a slightly vacant looking courtney thorne-smith smacking belushi over
the head with a frying pan while he alternately beats his chest and plays with
the straw on the floor of his cage. 5 stars for larry joe campbell designated
comedic bacon because he improves the flavor of everything hes in!
True Label: 0
Predicted Label: 0
Tokenisation:
 tensor([  101,  1045,  2467,  2626,  2023,  2186,  2125,  2004,  2108,  1037,
        3143, 27136,  1011, 17037,  2138,  3958, 19337, 20668,  2072,  2001,
        2920,  1999,  2009,  1010,  1998,  4600,  1012,  2021,  2059,  2028,
        2154,  1037, 13800,  6433, 26897,  4158,  1012,  2044,  1037,  2317,
        9175,  2208,  3092,  1045,  3651,  2008,  1996,  6556,  2001,  2035,
        1996,  2126,  2006,  1996,  2060,  2217,  1997,  1996,  2282,  5064,
        1012,  2085,  1045,  2071,  2031,  2074,  5407,  2039,  1998,  2939,
        2408,  1996,  2282,  2000,  2131,  1996,  6556,  1010,  2030,  2130,
        2000,  1996,  2694,  2000,  2735,  1996,  3149,  1012,  2021,  2059,
        2339,  2025,  2074,  2131,  2039,  1998,  3328,  2408,  1996,  2406,
        2000,  3422,  2694,  1999,  2178,  2110,  1029, 12264,  2000,  2008,
        1010,  1045,  2056,  1012,  2061,  1045,  2787,  2000,  2074,  6865,
        4389,  2006,  1996,  6411,  1998,  2202,  3649,  6580,  2018,  1999,
        3573,  2005,  2033,  1012,  2054,  6580,  2018,  1999,  3573,  2001,
        2019,  2792,  1997,  2023,  2265,  1010,  2019,  2792,  2055,  2029,
        1045,  3342,  2200,  2210,  3272,  2008,  1045,  2018,  2320,  2153,
        2081,  1037,  2200,  5041,  1010,  2236, 12720,  8768,  8689,  2241,
        2006,  5717,  7863,  2030,  4654,  4842, 11638,  4818,  3350,  2007,
        2498, 18971,  2000,  2067,  2026, 10740,  2039,  2007,  1010,  1998,
        2320,  2153,  1045,  2001,  3294,  2157,   999,  2023,  2265,  2003,
        1037,  2561, 13675,  6784,  1011, 11345,   999, 19337, 20668,  2072,
        2038,  2035,  1996, 21699,  6959,  1997,  1037, 15892, 10171,  9666,
        9769,  1012,  1996,  2308,  2024,  8186,  8702,  2021,  2205,  3357,
        3877,  1011,  2003,  2000, 12005, 26243,  2151,  2613,  3110,  2013,
        1996, 13972,  1012,  2045,  2003,  7078,  2053,  3114,  2000,  2644,
        4426,  2013,  2770,  2091,  2000,   102])
```

Here we measure the performance of our final model on a completely unseen test dataset. We do so by outputting an ROC curve and the associated accuracy, recall, precision, and f1-score. We see our accuracy for this epoch 3 model is roughly 90%. The precision for a bad review is 91% whereas for a good review it is 88%. This means out of the positive values (1), the model does better at getting true postitives for bad reviews versus good reviews. Inverse values are seen in recall to reflect this as well.

The ROC curve is near the top left of the graph. This is good, indicating few type 1 and type 2 errors.

Finally, we see an example review with the true and predicted label. As we can see, the review is quite convoluted. The author describes the film as being bad but certain actors' performances being good. This demonstrates the potential difficulty of the task at hand, despite my model getting it right with use of the tokenized input shown as well.

1.0.11 Interpretation

This model is performing particularly well given all of computational constraints we had to consider in developing the model. These weaknesses include:

- Using DistilBERT rather than base BERT or even large BERT.
- Limiting the max length when tokenizing to only 256. While 512 is the max for BERT and even that would have truncated a lot of information from our model (some text lengths were up to and over 1000 tokens), we had to constrain to 256 for computational reasons.

- Limiting the amount of data. We had to take a quarter of the total data available, again for computational reasons.
- Limiting number of epochs. If we have a bigger batch size and potentially lower the learning rate, we might be able to tune our model further, however large amounts of cross validation would be necessary, also creating computational costs for our GPU.

Had we not made these "sacrifices" in our model's performance for making this demonstration project feasible within Collab's environment, we likely would see an even better performance. The model does have some strengths as well, however:

- DistilBERT is still a very robust model and performs to a high standard relative to other non-BERT models.
- It is modular. We can easily change the hyperparameters we use to increase the performance of our model depending on the computational performance we have in any circumstance.
- It is relatively interpretable. We know what is happening as the BERT trains, we can see what parts include the forward pass and backpropagation, and the output of the model is easy to evaluate.

If I had more time and resources, I would use large BERT, use and gather even more IMDB data, use a BERT model that doesn't have a tokenization length constraint of 512, test the best max_length value and use it so that we truncate a limited amount of data, and cross validate using different hyperparameters to test the performance of my model and find the best configuration.

If working on a similar problem but different dataset, I would also be concious of the variations in types of text that BERT works with and perform very specific text cleaning to optimise the input of that text. I likely could have done this more here as well, however, my cleaning was up to and beyond the standard necessary for this demonstration

---

1.0.12 References

aerin. (2019). What does model.train() do in PyTorch? Retrieved from https://stackoverflow.com/questions/51433378/what-does-model-train-do-in-pytorch

Bricken, A. (2021). Does BERT Need Clean Data? Part 1: Data Cleaning. Retrieved from https://medium.com/towards-data-science/part-1-data-cleaning-does-bert-need-clean-data-6a50c9c6e9fd

Bricken, A. (2021). Does BERT Need Clean Data? Part 2: Classification. Retrieved from https://medium.com/@Alexander.Bricken/does-bert-need-clean-data-part-2-classification-d29adf9f745a

HuggingFace. (2022). BERT. Retrieved from https://huggingface.co/transformers/model_doc/bert.html#bertfors

Jain, K. (2021). Movie Review Sentiment Analysis EDA | BERT. Retrieved from https://www.kaggle.com/kritanjalijain/movie-review-sentiment-analysis-eda-bert

McCormick, C. (2019). BERT Fine-Tuning Tutorial with PyTorch. Retrieved ftom https://mccormickml.com/2019/07/22/BERT-fine-tuning/

Morris, J. (n.d.) Does Model Size Matter? A Comparison of BERT and DistilBERT.

Retrieved from https://wandb.ai/jack-morris/david-vs-goliath/reports/Does-Model-Size-Matter-A-Comparison-of-BERT-and-DistilBERT–VmlldzoxMDUxNzU

Thakur, A. (n.d.) How To Use GPU with PyTorch. Retrieved from https://wandb.ai/wandb/common-ml-errors/reports/How-To-Use-GPU-with-PyTorch—VmlldzozMzAxMDk

user1424739. (2017). Why do we need to call zero_grad() in PyTorch? Retrieved from https://stackoverflow.com/questions/48001598/why-do-we-need-to-call-zero-grad-in-pytorch

Yuan, Z. (2020). IMDB dataset (Sentiment analysis) in CSV format. Retrieved from https://www.kaggle.com/columbine/imdb-dataset-sentiment-analysis-in-csv-format