Table of Contents

[Better Sentiment Analysis with BERT 1](#_Toc116646227)

[(Bidirectional Encoder Representations from Transformers) 1](#_Toc116646228)

[DistilBERT versus BERT 1](#_Toc116646229)

[Limitations 1](#_Toc116646230)

[Fine Tuning BERT 6](#_Toc116646231)

[Options for Fine Tuning BERT Locally 7](#_Toc116646232)

[Options for Fine Tuning BERT on Google Colab 7](#_Toc116646233)

[Loading a Model with Fine-Tuned Weights 14](#_Toc116646234)

[Changing the Dataset and Number of Classes 17](#_Toc116646235)

## Better Sentiment Analysis with BERT

### (Bidirectional Encoder Representations from Transformers)

|  |  |
| --- | --- |
|  | This section is based on the article at <https://colab.research.google.com/github/jalammar/jalammar.github.io/blob/master/notebooks/bert/A_Visual_Notebook_to_Using_BERT_for_the_First_Time.ipynb> |

BERT transforms text into a set of embedded features which can then be sent to another model for sentiment analysis. In effect, when performing sentiment analysis with BERT you are building two models:

1. The BERT model generates the features by transforming the original text into features.
2. Logistic regression (or some other classifier) predicts the sentiment with the features that are output from the BERT model.

BERT was released by Google in 2018. BERT implements bi-directional training with pretrained weights to develop a text feature transformer. Before this development, training would only involve sequential training from left to right or from right to left. The pre-trained weights enable faster and better training. You can customize the BERT weights as well.

### DistilBERT versus BERT

DistilBERT is a faster and lighter-weight implementation of BERT which performs almost as well as BERT.

### Limitations

The maximum sentence length is 512 tokens.

Example : Preparing the Model for BERT Transformation

This example shows steps for how to build and train the BERT transformation model and how to prepare the data for the BERT transformation:

|  |
| --- |
| import numpy as np  import pandas as pd  import torch  import transformers as ppb  import warnings  warnings.filterwarnings('ignore')  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "movie\_reviewsBERT.csv"  batch\_1 = pd.read\_csv(PATH + FILE, delimiter=',', header=None)  print(batch\_1.shape)  ROW =1  print("Review 1st column: " + batch\_1.iloc[ROW][0])  print("Rating 2nd column: " + str(batch\_1.iloc[ROW][1]))  # Show counts for review scores.  print("\*\* Showing review counts")  print(batch\_1[1].value\_counts())  # Load pretrained models.  # For DistilBERT:  model\_class, tokenizer\_class, pretrained\_weights = (ppb.DistilBertModel,  ppb.DistilBertTokenizer, 'distilbert-base-uncased')  ## Want BERT instead of distilBERT? Uncomment the following line:  #model\_class, tokenizer\_class, pretrained\_weights = (ppb.BertModel, ppb.BertTokenizer, 'bert-base-uncased')  # Load pretrained model/tokenizer  tokenizer = tokenizer\_class.from\_pretrained(pretrained\_weights)  model = model\_class.from\_pretrained(pretrained\_weights)  # Tokenize the sentences.  tokenized = batch\_1[0].apply((lambda x: tokenizer.encode(x, add\_special\_tokens=True)))  print("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Tokenized reviews ")  print(tokenized)  print(tokenized.values)  print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  # For processing we convert to 2D array.  max\_len = 0  # Get maximum number of tokens (get biggest sentence).  print("\nGetting maximum number of tokens in a sentence")  for i in tokenized.values:  if len(i) > max\_len:  max\_len = len(i)  print("Most tokens in a review (max\_len): " + str(max\_len))  # Add padding  print("------------")  print("Padded so review arrays as same size: ")  padded = np.array([i + [0]\*(max\_len-len(i)) for i in tokenized.values])  print("These are the padded reviews:")  print(padded)  print("This is the last padded sentence:")  LAST\_INDEX = len(batch\_1) -1  print(padded[LAST\_INDEX])  print("\n------------")  print("Attention mask tells BERT to ignore the padding.")  # If we directly send padded data to BERT, that would slightly confuse it.  # We need to create another variable to tell it to ignore (mask) the padding  # we've added when it's processing its input. That's what attention\_mask is  attention\_mask = np.where(padded != 0, 1, 0)  print(attention\_mask.shape)  print(attention\_mask)  print(attention\_mask[LAST\_INDEX])  print("=============")  input\_ids = torch.tensor(padded)  attention\_mask = torch.tensor(attention\_mask)  print("Input ids which are padded reviews in torch tensor format:")  print(input\_ids)  print("Attention mask in torch tensor format:")  print(attention\_mask)  print("++++++++++++++") |

Exercise (1 mark)

Examine the Example 1. Show the line of code that downloads the pretrained weights.

|  |
| --- |
| model\_class, tokenizer\_class, pretrained\_weights = (ppb.DistilBertModel,  ppb.DistilBertTokenizer, 'distilbert-base-uncased') |

Exercise (2 marks)

These next exercises examine how to prepare the data for the BERT transformation. Add this code to Example 1.

|  |
| --- |
| dfEx = pd.DataFrame(columns=[0,1])  dfEx = dfEx.append({0:"This brilliant movie is jaw-dropping.", 1:1},  ignore\_index=True)  dfEx = dfEx.append({0:"This movie is awful.", 1:0}, ignore\_index=True) |

Adjust the code to show the tokenized version of the sentence after tokenizing it with the BERT tokenizer. Show these two sentences in their tokenized form here:

|  |
| --- |
|  |

Show the code that you used to tokenize the sentences here:

|  |
| --- |
| # max sentence length is 512   import numpy as np import pandas as pd  import torch import transformers as ppb import warnings warnings.filterwarnings('ignore')  # Show all columns. pd.set\_option('display.max\_columns', None) pd.set\_option('display.width', 1000)  PATH = "C:\\datasets\\" FILE = "movie\_reviewsBERT.csv" batch\_1 = pd.read\_csv(PATH + FILE, delimiter=',', header=None)  print(batch\_1.shape) ROW =1 print("Review 1st column: " + batch\_1.iloc[ROW][0]) print("Rating 2nd column: " + str(batch\_1.iloc[ROW][1]))  # Show counts for review scores. print("\*\* Showing review counts") print(batch\_1[1].value\_counts())  # Load pretrained models. # For DistilBERT: model\_class, tokenizer\_class, pretrained\_weights = (ppb.DistilBertModel,  ppb.DistilBertTokenizer, 'distilbert-base-uncased')  ## Want BERT instead of distilBERT? Uncomment the following line: #model\_class, tokenizer\_class, pretrained\_weights = (ppb.BertModel, ppb.BertTokenizer, 'bert-base-uncased')  # Load pretrained model/tokenizer tokenizer = tokenizer\_class.from\_pretrained(pretrained\_weights) model = model\_class.from\_pretrained(pretrained\_weights)  # Tokenize the sentences. tokenized = batch\_1[0].apply((lambda x: tokenizer.encode(x, add\_special\_tokens=True))) print("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Tokenized reviews ") print(tokenized) print(tokenized.values) print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  # For processing we convert to 2D array. max\_len = 0  # Get maximum number of tokens (get the biggest sentence). print("\nGetting maximum number of tokens in a sentence") for i in tokenized.values:  if len(i) > max\_len:  max\_len = len(i)  print("Most tokens in a review (max\_len): " + str(max\_len))  # Add padding print("------------") print("Padded so review arrays as same size: ") padded = np.array([i + [0]\*(max\_len-len(i)) for i in tokenized.values]) print("These are the padded reviews:") print(padded) print("This is the last padded sentence:") LAST\_INDEX = len(batch\_1) -1 print(padded[LAST\_INDEX]) print("\n------------") print("Attention mask tells BERT to ignore the padding.")  # If we directly send padded data to BERT, that would slightly confuse it. # We need to create another variable to tell it to ignore (mask) the padding # we've added when it's processing its input. That's what attention\_mask is attention\_mask = np.where(padded != 0, 1, 0) print(attention\_mask.shape) print(attention\_mask) print(attention\_mask[LAST\_INDEX]) print("=============")  input\_ids = torch.tensor(padded) attention\_mask = torch.tensor(attention\_mask) print("Input ids which are padded reviews in torch tensor format:") print(input\_ids) print("Attention mask in torch tensor format:") print(attention\_mask) print("++++++++++++++")   #################### dfEx = pd.DataFrame(columns=[0,1]) dfEx = dfEx.append({0:"This brilliant movie is jaw-dropping.", 1:1},  ignore\_index=True) dfEx = dfEx.append({0:"This movie is awful.", 1:0}, ignore\_index=True)  # Load pretrained models. # For DistilBERT: model\_class, tokenizer\_class, pretrained\_weights = (ppb.DistilBertModel,  ppb.DistilBertTokenizer, 'distilbert-base-uncased')  ## Want BERT instead of distilBERT? Uncomment the following line: #model\_class, tokenizer\_class, pretrained\_weights = (ppb.BertModel, ppb.BertTokenizer, 'bert-base-uncased')  # Load pretrained model/tokenizer tokenizer = tokenizer\_class.from\_pretrained(pretrained\_weights) model = model\_class.from\_pretrained(pretrained\_weights)  # Tokenize the sentences. tokenized = dfEx[0].apply((lambda x: tokenizer.encode(x, add\_special\_tokens=True))) print("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Tokenized reviews ") print(tokenized) print(tokenized.values) print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  # For processing we convert to 2D array. max\_len = 0  # Get maximum number of tokens (get biggest sentence). print("\nGetting maximum number of tokens in a sentence") for i in tokenized.values:  if len(i) > max\_len:  max\_len = len(i)  print("Most tokens in a review (max\_len): " + str(max\_len))  # Add padding print("------------") print("Padded so review arrays as same size: ") padded = np.array([i + [0]\*(max\_len-len(i)) for i in tokenized.values]) print("These are the padded reviews:") print(padded) print("This is the last padded sentence:") LAST\_INDEX = len(dfEx) -1 print(padded[LAST\_INDEX]) print("\n------------") print("Attention mask tells BERT to ignore the padding.") |

Exercise (1 mark)

Two of the tokens are repeated in each set of tokens from Exercise 2. Which words do they represent?

|  |
| --- |
|  |

Exercise (2 marks)

Pad the data that was added in Exercise 2. Show your padded data here:

|  |
| --- |
|  |

Show the code that you used to pad the data here:

|  |
| --- |
|  |

Exercise (1 mark)

Why is padding added to the tokenized sentences?

|  |
| --- |
|  |

Exercise (3 marks)

Create an attention mask for the data that you added in Exercise 2. Show the attention mask output for the rows of data here:

|  |
| --- |
|  |

Show the code that you used to create the attention mask here:

|  |
| --- |
|  |

Explain what the attention mask does here:

|  |
| --- |
|  |

Exercise (2 marks)

Convert the attention mask and padded tokens to tensor format for use with PyTorch. (A tensor can be a vector or matrix) Show the tensors here:

|  |
| --- |
|  |

Show the code that you used to perform this transformation here:

|  |
| --- |
|  |

Example : Building the Features

This code shows how to use BERT to transform the tokenized data transform data to create the features for the logistic regression model. To avoid a much bigger network configuration with either Keras or Tensorflow, the PyTorch framework is used to perform the transformation. Add this code to the end of Example 1 to build it.

|  |
| --- |
| # The model() function runs our sentences through BERT. The results of the  # processing will be returned into last\_hidden\_states.  print("BERT model transforms tokens and attention mask tensors into features ")  print("for logistic regression.")  with torch.no\_grad():  last\_hidden\_states = model(input\_ids, attention\_mask=attention\_mask)  # We'll save those in the features variable, as they'll serve as the  # features to our logitics regression model.  features = last\_hidden\_states[0][:,0,:].numpy()  print("Let's see the features: ")  print(features)  print(features[1999])  print("-------------------------") |

Exercise (3 marks)

Starting with tensor values that you created in Exercise 7, use the BERT model to generate a feature set. Show the BERT features for the exercise data here:

|  |
| --- |
|  |

Show the code that you used to create the BERT features with the exercise data here:

|  |
| --- |
|  |

Example : Logistic Regression

This code splits the features and labels and then implements logistic regression to predict sentiment. Add this code to the end of Example 2 to complete the solution.

The accuracy is:

0.828

|  |
| --- |
| from sklearn.linear\_model import LogisticRegression  from sklearn.model\_selection import train\_test\_split  # The labels indicating which sentence is positive and negative  # now go into the labels variable  labels = batch\_1[1]  # Let's now split our datset into a training set and testing set (even  # though we're using 2,000 sentences from the SST2 training set).  train\_features, test\_features, train\_labels, test\_labels = train\_test\_split(features, labels)  train\_labels.describe()  lr\_clf = LogisticRegression(solver='liblinear')  lr\_clf.fit(train\_features, train\_labels)  predictions = lr\_clf.predict(test\_features)  import math  from sklearn import metrics  from sklearn.metrics import classification\_report  print(classification\_report(test\_labels, predictions))  rmse2 = math.sqrt(metrics.mean\_squared\_error(test\_labels,predictions ))  print("RMSE: " + str(rmse2)) |

Example : Multi-Class Prediction

During the lesson prior to this lab we predicted 0 to 4 with an accuracy score of 37%. With the pretrained weights from BERT we achieved 45% accuracy.

This code took about 50 minutes for me to run on my middle of the road computer. You do not need to run this but if you are interested replace the data loading code in Example 3 with this version and run the project again.

|  |
| --- |
| PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "cleanedMovieReviews.tsv"  batch\_1 = pd.read\_csv(PATH + FILE, delimiter='\t')  batch\_1 = batch\_1[['Phrase', 'Sentiment']]  batch\_1 = batch\_1.rename(columns={'Phrase': 0, 'Sentiment': 1}) |

The output from the BERT features model shows a much better accuracy, precision and recall showing compared to the results obtained with the non-BERT model.

|  |
| --- |
| precision recall f1-score support  0 0.45 0.28 0.35 286  1 0.43 0.57 0.50 534  2 0.35 0.22 0.27 431  3 0.47 0.61 0.53 563  4 0.56 0.45 0.50 319  accuracy 0.45 2133  macro avg 0.45 0.43 0.43 2133  weighted avg 0.45 0.45 0.44 2133  RMSE: 1.0715167512214394 |

## Fine Tuning BERT

Example : Fine Tuning BERT

This example shows steps needed to load a pre-tuned BERT model and fine tune it for your domain. This example is based on the example provided at:

<https://www.analyticsvidhya.com/blog/2020/07/transfer-learning-for-nlp-fine-tuning-bert-for-text-classification/>

The article provides useful detail if you are interested. To make this work, be sure to install transformers version 3.0.0. Version 4 will not work with the code.

**pip install transformers==3.0.0**

### Options for Fine Tuning BERT Locally

You can run this locally in PyCharm if you want. I have uploaded two files that you can use for input:

* **spamdata\_lite.csv**
* **spamdata\_v2.csv**

The **spamdata\_lite.csv** file contains only173 rows for training so be aware the results will not be impressive. However, you can use this file for fast training and for testing your code locally.

The **spamdata\_v2.csv** file contains 5572 rows for training will offer better results but it will also slow down the training considerably.

## Options for Fine Tuning BERT on Google Colab

You can fine tune a BERT model on Google Colab. You may want to use the GPU which is free initially so try not to use it often. To do this:

1. Login with your Google login. Then, navigate to **Runtime | Change runtime type**. When prompted in Google Colab choose GPU.
2. Then change the highlighted code below to cuda.

Also, make sure that you run the instruction to install transformers 3 from one of the code cells:

**pip install transformers==3.0.0**

Also, change the following instruction from:

**df = pd.read\_csv(PATH + "spamdata\_lite.csv")**

To:

**df = pd.read\_csv("https://github.com/prateekjoshi565/Fine-Tuning-BERT/raw/master/spamdata\_v2.csv")**

**Note 1:**

This code is more detailed and complex than we can cover in the course. For what it is worth though, I will explain how to use it to fine tune your BERT models.

**Note 2:**

When using **spamdata\_lite.csv** the prediction quality will not be good but the reduced set helps to ensure the code runs properly. Use **spamdata\_v2.csv** for better results.

Note to Windows users:

**Example 2: Windows conversion error**  
In windows (not Mac) the tensors must be set to use integer.

input\_ids = torch.tensor(padded).to(torch.int64)  
attention\_mask = torch.tensor(attention\_mask).to(torch.int64)

|  |
| --- |
| import time  import numpy as np  import pandas as pd  import torch  import torch.nn as nn  from joblib.logger import format\_time  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import classification\_report  from transformers import AutoModel, BertTokenizerFast  # specify GPU  device = torch.device("cpu")  PATH = "./sample\_data/"  df = pd.read\_csv(PATH + "spamdata\_lite.csv")  df.head()  # Create training set.  train\_text, temp\_text, train\_labels, temp\_labels =\  train\_test\_split(df['text'], df['label'], random\_state=2018, test\_size=0.3,\  stratify=df['label'])  # Use temp set from above to create validation and test set.  # Validation is used to test the model while training. Test is used  # to validate the model after training.  val\_text, test\_text, val\_labels, test\_labels = \  train\_test\_split(temp\_text, temp\_labels, random\_state=2018, test\_size=0.5,  stratify=temp\_labels)  # import BERT-base pretrained model.  # We are using weights that are suitable for uncased content. However if  # upper and lower case words are relevant for your domain use cased.  bertModel = AutoModel.from\_pretrained('bert-base-uncased')  # Load the BERT tokenizer  tokenizer = BertTokenizerFast.from\_pretrained('bert-base-uncased')  # Tokenize and encode sentences.  def prepXandY(text, labels):  textList = text.tolist()  tokens = tokenizer.batch\_encode\_plus(  textList,  max\_length=25,  pad\_to\_max\_length=True,  truncation=True  )  seq = torch.tensor(tokens['input\_ids'])  mask = torch.tensor(tokens['attention\_mask'])  y = torch.tensor(labels.tolist())  return seq, mask, y  print(test\_text)  # Prepare the data.  def getTensor(text, labels):  seq, mask, y = prepXandY(train\_text, train\_labels)  tensorData = TensorDataset(seq, mask, y)  return tensorData  from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler  # define a batch size  batch\_size = 32  # wrap tensors  train\_data = getTensor(train\_text, train\_labels)  val\_data = getTensor(val\_text, val\_labels)  # sampler for sampling the data during training  train\_sampler = RandomSampler(train\_data)  # dataLoader for train set  train\_dataloader = DataLoader(train\_data, sampler=train\_sampler, batch\_size=batch\_size)  # sampler for sampling the data during training  val\_sampler = SequentialSampler(val\_data)  # dataLoader for validation set  val\_dataloader = DataLoader(val\_data, sampler=val\_sampler, batch\_size=batch\_size)  # freeze all the parameters  for param in bertModel.parameters():  param.requires\_grad = False  class BERT\_Arch(nn.Module):  def \_\_init\_\_(self, bert):  super(BERT\_Arch, self).\_\_init\_\_()  self.bert = bert  # dropout layer  self.dropout = nn.Dropout(0.1)  # relu activation function  self.relu = nn.ReLU()  # dense layer 1  self.fc1 = nn.Linear(768, 512)  # dense layer 2 (Output layer)  self.fc2 = nn.Linear(512, 2)  # 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  \_, cls\_hs = self.bert(sent\_id, attention\_mask=mask)  x = self.fc1(cls\_hs)  x = self.relu(x)  x = self.dropout(x)  # output layer  x = self.fc2(x)  # apply softmax activation  x = self.softmax(x)  return x  # pass the pre-trained BERT to our define architecture  model = BERT\_Arch(bertModel)  # push the model to GPU  model = model.to(device)  # optimizer from hugging face transformers  from transformers import AdamW  # define the optimizer  optimizer = AdamW(model.parameters(),  lr=1e-5) # learning rate  from sklearn.utils.class\_weight import compute\_class\_weight  # compute the class weights  class\_weights = compute\_class\_weight(class\_weight='balanced', classes=np.unique(train\_labels), y=train\_labels)  print("Class Weights:", class\_weights)  # converting list of class weights to a tensor  weights = torch.tensor(class\_weights, dtype=torch.float)  # push to GPU (if it exists)  weights = weights.to(device)  # define the loss function  cross\_entropy = nn.NLLLoss(weight=weights)  # number of training epochs  epochs = 10  from sklearn.utils.class\_weight import compute\_class\_weight  # compute the class weights  class\_weights = compute\_class\_weight(class\_weight ='balanced', \  classes=np.unique(train\_labels),  y=train\_labels)  print("Class Weights:", class\_weights)  # function to train the model  def train():  model.train()  total\_loss, total\_accuracy = 0, 0  # empty list to save model predictions  total\_preds = []  # iterate over batches  for step, batch in enumerate(train\_dataloader):  # progress update after every 50 batches.  if step % 50 == 0 and not step == 0:  print(' Batch {:>5,} of {:>5,}.'.format(step, len(train\_dataloader)))  # push the batch to gpu  batch = [r.to(device) for r in batch]  sent\_id, mask, labels = batch  # clear previously calculated gradients  model.zero\_grad()  # get model predictions for the current batch  preds = model(sent\_id, mask)  # compute the loss between actual and predicted values  loss = cross\_entropy(preds, labels)  # add on to the total loss  total\_loss = total\_loss + loss.item()  # backward pass to calculate the gradients  loss.backward()  # clip the the gradients to 1.0. It helps in preventing the exploding gradient problem  torch.nn.utils.clip\_grad\_norm\_(model.parameters(), 1.0)  # update parameters  optimizer.step()  # model predictions are stored on GPU. So, push it to CPU  preds = preds.detach().cpu().numpy()  # append the model predictions  total\_preds.append(preds)  # compute the training loss of the epoch  avg\_loss = total\_loss / len(train\_dataloader)  # predictions are in the form of (no. of batches, size of batch, no. of classes).  # reshape the predictions in form of (number of samples, no. of classes)  total\_preds = np.concatenate(total\_preds, axis=0)  # returns the loss and predictions  return avg\_loss, total\_preds  # function for evaluating the model  def evaluate():  print("\nEvaluating...")  # deactivate dropout layers  model.eval()  total\_loss, total\_accuracy = 0, 0  # empty list to save the model predictions  total\_preds = []  # iterate over batches  for step, batch in enumerate(val\_dataloader):  # Progress update every 50 batches.  if step % 50 == 0 and not step == 0:  # Report progress.  print(' Batch {:>5,} of {:>5,}.'.format(step, len(val\_dataloader)))  # push the batch to gpu  batch = [t.to(device) for t in batch]  sent\_id, mask, labels = batch  # deactivate autograd  with torch.no\_grad():  # model predictions  preds = model(sent\_id, mask)  # compute the validation loss between actual and predicted values  loss = cross\_entropy(preds, labels)  total\_loss = total\_loss + loss.item()  preds = preds.detach().cpu().numpy()  total\_preds.append(preds)  # compute the validation loss of the epoch  avg\_loss = total\_loss / len(val\_dataloader)  # reshape the predictions in form of (number of samples, no. of classes)  total\_preds = np.concatenate(total\_preds, axis=0)  return avg\_loss, total\_preds  # set initial loss to infinite  best\_valid\_loss = float('inf')  # empty lists to store training and validation loss of each epoch  train\_losses = []  valid\_losses = []  # for each epoch  for epoch in range(epochs):  print('\n Epoch {:} / {:}'.format(epoch + 1, epochs))  # train model  train\_loss, \_ = train()  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')  # append training and validation 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}')  # load weights of best model  path = 'saved\_weights.pt'  model.load\_state\_dict(torch.load(path))  # Validate with test data.  test\_seq, test\_mask, test\_y = prepXandY(test\_text, test\_labels)  # get predictions for test data  with torch.no\_grad():  preds = model(test\_seq.to(device), test\_mask.to(device))  preds = preds.detach().cpu().numpy()  # %%  print("Show preds before report")  print(preds)  preds = np.argmax(preds, axis=1)  print("Preds after argmax")  print(preds)  print(classification\_report(test\_y, preds)) |

## Loading a Model with Fine-Tuned Weights

Example : Load the Data and Make Prediction

This example shows how to initialize a BERT model and load fine-tuned weights to make a prediction with new data. This implementation assumes that you had run the code in Example 5 locally and that the 'saved\_weights.pt' exist in the project folder.

**Note: The results will much more reliable if the 'saved\_weights.pt' file is trained using the larger spam file.**

|  |
| --- |
| # pip install transformers==3.0.0  import numpy as np  import pandas as pd  import torch  import torch.nn as nn  from sklearn.metrics import classification\_report  from transformers import AutoModel, BertTokenizerFast  bertModel = AutoModel.from\_pretrained('bert-base-uncased')  tokenizer = BertTokenizerFast.from\_pretrained('bert-base-uncased')  device = torch.device("cpu")  # Tokenize and encode sentences.  def prepXandY(text, labels):  textList = text.tolist()  tokens = tokenizer.batch\_encode\_plus(  textList,  max\_length=25,  pad\_to\_max\_length=True,  truncation=True  )  seq = torch.tensor(tokens['input\_ids'])  mask = torch.tensor(tokens['attention\_mask'])  y = torch.tensor(labels.tolist())  return seq, mask, y  # freeze all the parameters  for param in bertModel.parameters():  param.requires\_grad = False  # To initialize a model with weights a network  # architecture must be specified.  class BERT\_Arch(nn.Module):  def \_\_init\_\_(self, bert):  super(BERT\_Arch, self).\_\_init\_\_()  self.bert = bert  # dropout layer  self.dropout = nn.Dropout(0.1)  # relu activation function  self.relu = nn.ReLU()  # dense layer 1  self.fc1 = nn.Linear(768, 512)  # dense layer 2 (Output layer)  self.fc2 = nn.Linear(512, 2)  # 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  \_, cls\_hs = self.bert(sent\_id, attention\_mask=mask)  x = self.fc1(cls\_hs)  x = self.relu(x)  x = self.dropout(x)  # output layer  x = self.fc2(x)  # apply softmax activation  x = self.softmax(x)  return x  # pass the pre-trained BERT to our define architecture  model = BERT\_Arch(bertModel)  # push the model to GPU  model = model.to(device)  # load weights of best model  path = 'saved\_weights.pt'  model.load\_state\_dict(torch.load(path))  # Create a data frame with text and data.  dfNewData = pd.DataFrame(columns=['text', 'label'])  dfNewData = dfNewData.append({'text':'Something is wrong with your bank account. Please login here.',  'label':1}, ignore\_index=True)  dfNewData = dfNewData.append({'text':'Buy one get one FREE for a limited time only!!',  'label':1}, ignore\_index=True)  dfNewData = dfNewData.append({'text':'Hi Jane, I was hoping I could speak with you. Can you call me?',  'label':0}, ignore\_index=True)  test\_seq, test\_mask, test\_y = prepXandY(dfNewData['text'], dfNewData['label'])  preds = []  # get predictions for test data  with torch.no\_grad():  preds = model(test\_seq.to(device), test\_mask.to(device))  preds = preds.detach().cpu().numpy()  print("Show preds before report")  print(preds)  preds = np.argmax(preds, axis=1)  print("Preds after argmax")  print(preds)  print(classification\_report(test\_y, preds)) |

## Changing the Dataset and Number of Classes

Example : Changing the Dataset and Number of Classes

This example shows how to change the data set from the spam set to the cleaned movie reviews set. The movie reviews set also has 5possible ratings and not 2.

To build this revised version, replace the data loading and test train split code with the version below. Note that I am using **cleanedMovieReviews\_lite.ts**v to save time but the results will be terrible. You may want to try this on Google Colab with the full data set for a better result.

|  |
| --- |
| PATH = "/Users/pm/Desktop/DayDocs/data/"  import pandas as pd  CLEAN\_DATA = "cleanedMovieReviews\_lite.tsv"  df = pd.read\_csv(PATH + CLEAN\_DATA, skiprows=1, encoding = "ISO-8859-1",  sep='\t', names=('PhraseId','SentenceId','Phrase','Sentiment'))  print(df.head())  # Create training set.  train\_text, temp\_text, train\_labels, temp\_labels =\  train\_test\_split(df['Phrase'], df['Sentiment'], random\_state=2018,  test\_size=0.3, stratify=df['Sentiment']) |

Also, be sure to change the number of possible outcomes in the constructor for the BERT\_Arch()

class.

|  |
| --- |
| self.fc2 = nn.Linear(512, 5) |

Use the full data set for better results. My results here are terrible.

|  |
| --- |
| precision recall f1-score support  0 0.00 0.00 0.00 5  1 0.00 0.00 0.00 13  2 0.00 0.00 0.00 9  3 1.00 0.15 0.27 13  4 0.14 1.00 0.24 6  accuracy 0.17 46  macro avg 0.23 0.23 0.10 46  weighted avg 0.30 0.17 0.11 46 |