```
!pip install transformers
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from datetime import datetime
from pathlib import Path
import pandas as pd
import torchtext.data as ttd
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: sacremoses in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: dataclasses; python_version < "3.7" in /usr/local/lib/
Requirement already satisfied: filelock in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: tokenizers==0.9.4 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: packaging in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from sa
Requirement already satisfied: joblib in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: click in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: pyparsing>=2.0.2 in /usr/local/lib/python3.6/dist-pack
```

## Loading Dataset

We will use The 20 Newsgroups dataset Dataset homepage:

Scikit-learn includes some nice helper functions for retrieving the 20 Newsgroups dataset-- https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_20newsgroups.html. We'll use them below to retrieve the dataset.

Also look at results fron non- neural net models here : https://scikit-learn.org/stable/auto_examples/text/plot_document_classification_20newsgroups.html#sphx-glr-auto-examples-text-plot-document-classification-20newsgroups-py

```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Select the Runtime > "Change runtime type" menu to enable a GPU accelera
  print('and then re-execute this cell.')
else:
  print(gpu_info)
```

```
Wed Dec  2 04:21:04 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 455.38       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   41C    P0    28W / 250W |     10MiB / 16280MiB |      0%      Default |
|                               |                      |                 ERR! |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

```
from sklearn.datasets import fetch_20newsgroups

train = fetch_20newsgroups(subset='train',
                           remove=('headers', 'footers', 'quotes'))

test = fetch_20newsgroups(subset='test',
                           remove=('headers', 'footers', 'quotes'))
```

```
print(train.data[0])
```

```
I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.
```

```
print(train.target[0])
```

```
7
```

```
train.target_names
```

```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
```

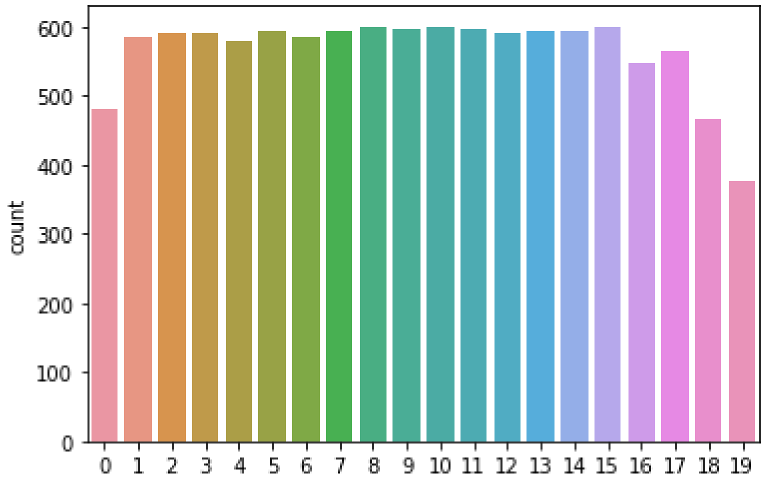
```
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

```
len(train.target_names)
```

```
20
```

```
import seaborn as sns

# Plot the number of tokens of each length.
sns.countplot(train.target);
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass
  FutureWarning
```




## BERT with 140 features and chunking

```
from transformers import BertTokenizer

# Load the BERT tokenizer.
print('Loading BERT tokenizer...')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True
```

```
Loading BERT tokenizer...
```

Downloading: 100% 232k/232k [00:00<00:00, 2.65MB/s]

```
from collections import defaultdict
# Creating Train Chunks
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []
labels = []
maxlen=140
train_chunks = defaultdict(list)
index = 0
label_index = 0
# For every sentence...
for sent in train.data:
    # `encode_plus` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        truncation=False,
                        padding=False,
                        #max_length = maxlen,           # Pad & truncate all sent
                        return_attention_mask = True,   # Construct attn. masks.
                        #return_tensors = 'pt',     # Return pytorch tensors.
                   )

    values = []
    rem_ids = encoded_dict['input_ids']
    rem_attn_ids = encoded_dict['attention_mask']
    while len(rem_ids)> maxlen:
      values.append(index)
      index += 1
      ids = rem_ids[0:int(maxlen)-1]+[102]
      rem_ids = [rem_ids[0]] +rem_ids[int(maxlen):]
      encoded_dict['input_ids']=torch.tensor([ids])
      ids = rem_attn_ids[0:int(maxlen)-1]+[1]
      rem_attn_ids = [1] + rem_attn_ids[int(maxlen):]
      encoded_dict['attention_mask']=torch.tensor([ids])
      # Add the encoded sentence to the list.
      input_ids.append(encoded_dict['input_ids'])
      #print(input_ids)
      # And its attention mask (simply differentiates padding from non-padding).
      attention_masks.append(encoded_dict['attention_mask'])
      labels.append(train.target[label_index])
    else:
      values.append(index)
      index += 1
      ids = rem_ids + ([0] * (maxlen-len(rem_ids)))
      encoded_dict['input_ids']=torch.tensor([ids])
```

```
        encoded_dict['input_ids']=torch.tensor([ids])
        ids = rem_attn_ids + ([0] * (maxlen-len(rem_ids)))
        encoded_dict['attention_mask']=torch.tensor([ids])
        input_ids.append(encoded_dict['input_ids'])
        #print(input_ids)
        # And its attention mask (simply differentiates padding from non-padding).
        attention_masks.append(encoded_dict['attention_mask'])
        labels.append(train.target[label_index])
      train_chunks[label_index] = values
      label_index += 1
# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
#print(input_ids)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(labels)
```

```
    Token indices sequence length is longer than the specified maximum sequence length fc
```

```
from collections import defaultdict
# Creating Train Chunks
# Tokenize all of the sentences and map the tokens to thier word IDs.
test_input_ids = []
test_attention_masks = []
test_labels = []
maxlen=140
test_chunks = defaultdict(list)
index = 0
label_index = 0
# For every sentence...
for sent in test.data:
    # `encode_plus` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        truncation=False,
                        padding=False,
                        #max_length = maxlen,           # Pad & truncate all sent
                        return_attention_mask = True,   # Construct attn. masks.
                        #return_tensors = 'pt',     # Return pytorch tensors.
                   )

    values = []
    rem_ids = encoded_dict['input_ids']
    rem_attn_ids = encoded_dict['attention_mask']
    while len(rem_ids)> maxlen:
      values.append(index)
      index += 1
      ids = rem_ids[0:int(maxlen)-1]+[102]
```

```
        rem_ids = [rem_ids[0]] +rem_ids[int(maxlen):]
        encoded_dict['input_ids']=torch.tensor([ids])
        ids = rem_attn_ids[0:int(maxlen)-1]+[1]
        rem_attn_ids = [1] + rem_attn_ids[int(maxlen):]
        encoded_dict['attention_mask']=torch.tensor([ids])
        # Add the encoded sentence to the list.
        test_input_ids.append(encoded_dict['input_ids'])
        #print(input_ids)
        # And its attention mask (simply differentiates padding from non-padding).
        test_attention_masks.append(encoded_dict['attention_mask'])
        test_labels.append(test.target[label_index])
      else:
        values.append(index)
        index += 1
        ids = rem_ids + ([0] * (maxlen-len(rem_ids)))
        encoded_dict['input_ids']=torch.tensor([ids])
        ids = rem_attn_ids + ([0] * (maxlen-len(rem_ids)))
        encoded_dict['attention_mask']=torch.tensor([ids])
        test_input_ids.append(encoded_dict['input_ids'])
        #print(input_ids)
        # And its attention mask (simply differentiates padding from non-padding).
        test_attention_masks.append(encoded_dict['attention_mask'])
        test_labels.append(test.target[label_index])
      test_chunks[label_index] = values
      label_index += 1
# Convert the lists into tensors.
test_input_ids = torch.cat(test_input_ids, dim=0)
#print(input_ids)
test_attention_masks = torch.cat(test_attention_masks, dim=0)
test_labels = torch.tensor(test_labels)


from torch.utils.data import TensorDataset, random_split

# Combine the training inputs into a TensorDataset.
dataset = TensorDataset(input_ids, attention_masks, labels)
test_dataset = TensorDataset(test_input_ids, test_attention_masks, test_labels)

# Create a 90-10 train-validation split.

# Calculate the number of samples to include in each set.
train_size = int(0.9 * len(dataset))
val_size = len(dataset) - train_size

# Divide the dataset by randomly selecting samples.
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

print('{:>5,} training samples'.format(train_size))
print('{:>5,} validation samples'.format(val_size))
print('{:>5,} test samples'.format(len(test_dataset)))
```

```
31,424 training samples
3,492 validation samples
19,438 test samples
```

```
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler

# The DataLoader needs to know our batch size for training, so we specify it
# here. For fine-tuning BERT on a specific task, the authors recommend a batch
# size of 16 or 32.
batch_size = 8

# Create the DataLoaders for our training and validation sets.
# We'll take training samples in random order.
train_dataloader = DataLoader(
            train_dataset,  # The training samples.
            sampler = RandomSampler(train_dataset), # Select batches randomly
            batch_size = batch_size # Trains with this batch size.
        )

# For validation the order doesn't matter, so we'll just read them sequentially.
validation_dataloader = DataLoader(
            val_dataset, # The validation samples.
            sampler = SequentialSampler(val_dataset), # Pull out batches sequenti
            batch_size = batch_size # Evaluate with this batch size.
        )

test_dataloader = DataLoader(
            test_dataset,  # The training samples.
            sampler = RandomSampler(test_dataset), # Select batches randomly
            batch_size = batch_size # Trains with this batch size.
        )


from transformers import  BertModel

bert_model = BertModel.from_pretrained('bert-base-uncased')
```

Downloading: 100% 433/433 [00:00<00:00, 1.37kB/s]

Downloading: 100% 440M/440M [00:08<00:00, 53.2MB/s]

```
bert_model
```

```
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (10): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
```

```
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
```

```
# Define the model
class linear(nn.Module):

  def __init__(self, bert_model, n_outputs, dropout_rate):

    super(linear, self).__init__()

    self.D = bert_model.config.to_dict()['hidden_size']
    self.bert_model = bert_model
    self.K = n_outputs
    self.dropout_rate=dropout_rate

    # embedding layer
    #self.embed = nn.Embedding(self.V, self.D)
```

```
    # dense layer
    self.fc = nn.Linear(self.D , self.K)

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

  def forward(self, X):

    with torch.no_grad():
      embedding = self.bert_model(X)[0][:,0,:]

    #embedding= self.dropout(embedding)

    output = self.fc(embedding)
    output= self.dropout(output)

    return output


n_outputs = 20
dropout_rate = 0.5


#model = RNN(n_vocab, embed_dim, n_hidden, n_rnnlayers, n_outputs, bidirectional,
model = linear(bert_model, n_outputs, dropout_rate)
model.to(device)
```

```
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (10): BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (query): Linear(in_features=768, out_features=768, bias=True)
                (key): Linear(in_features=768, out_features=768, bias=True)
                (value): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
              (output): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=True)
            )
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (11): BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (query): Linear(in_features=768, out_features=768, bias=True)
                (key): Linear(in_features=768, out_features=768, bias=True)
```

```
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (output): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
  )
  (pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
  )
)
(fc): Linear(in_features=768, out_features=20, bias=True)
(dropout): Dropout(p=0.5, inplace=False)
)
```

```
print(model)
```

```
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (10): BertLayer(
        (attention): BertAttention(
          (self): BertSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
            (key): Linear(in_features=768, out_features=768, bias=True)
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (output): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (11): BertLayer(
        (attention): BertAttention(
          (self): BertSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
```

```
            (key): Linear(in_features=768, out_features=768, bias=True)
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (output): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
  )
  (pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
  )
)
(fc): Linear(in_features=768, out_features=20, bias=True)
(dropout): Dropout(p=0.5, inplace=False)
)
```

```
for name, param in model.named_parameters():
  print(name, param.shape)
```

```
bert_model.encoder.layer.8.attention.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.8.intermediate.dense.weight torch.Size([3072, 768])
bert_model.encoder.layer.8.intermediate.dense.bias torch.Size([3072])
bert_model.encoder.layer.8.output.dense.weight torch.Size([768, 3072])
bert_model.encoder.layer.8.output.dense.bias torch.Size([768])
bert_model.encoder.layer.8.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.8.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.9.attention.self.query.weight torch.Size([768, 768])
bert_model.encoder.layer.9.attention.self.query.bias torch.Size([768])
bert_model.encoder.layer.9.attention.self.key.weight torch.Size([768, 768])
bert_model.encoder.layer.9.attention.self.key.bias torch.Size([768])
bert_model.encoder.layer.9.attention.self.value.weight torch.Size([768, 768])
bert_model.encoder.layer.9.attention.self.value.bias torch.Size([768])
bert_model.encoder.layer.9.attention.output.dense.weight torch.Size([768, 768])
bert_model.encoder.layer.9.attention.output.dense.bias torch.Size([768])
bert_model.encoder.layer.9.attention.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.9.attention.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.9.intermediate.dense.weight torch.Size([3072, 768])
bert_model.encoder.layer.9.intermediate.dense.bias torch.Size([3072])
bert_model.encoder.layer.9.output.dense.weight torch.Size([768, 3072])
bert_model.encoder.layer.9.output.dense.bias torch.Size([768])
bert_model.encoder.layer.9.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.9.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.10.attention.self.query.weight torch.Size([768, 768])
bert_model.encoder.layer.10.attention.self.query.bias torch.Size([768])
bert_model.encoder.layer.10.attention.self.key.weight torch.Size([768, 768])
bert_model.encoder.layer.10.attention.self.key.bias torch.Size([768])
```

```
bert_model.encoder.layer.10.attention.self.value.weight torch.Size([768, 768])
bert_model.encoder.layer.10.attention.self.value.bias torch.Size([768])
bert_model.encoder.layer.10.attention.output.dense.weight torch.Size([768, 768])
bert_model.encoder.layer.10.attention.output.dense.bias torch.Size([768])
bert_model.encoder.layer.10.attention.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.10.attention.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.10.intermediate.dense.weight torch.Size([3072, 768])
bert_model.encoder.layer.10.intermediate.dense.bias torch.Size([3072])
bert_model.encoder.layer.10.output.dense.weight torch.Size([768, 3072])
bert_model.encoder.layer.10.output.dense.bias torch.Size([768])
bert_model.encoder.layer.10.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.10.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.11.attention.self.query.weight torch.Size([768, 768])
bert_model.encoder.layer.11.attention.self.query.bias torch.Size([768])
bert_model.encoder.layer.11.attention.self.key.weight torch.Size([768, 768])
bert_model.encoder.layer.11.attention.self.key.bias torch.Size([768])
bert_model.encoder.layer.11.attention.self.value.weight torch.Size([768, 768])
bert_model.encoder.layer.11.attention.self.value.bias torch.Size([768])
bert_model.encoder.layer.11.attention.output.dense.weight torch.Size([768, 768])
bert_model.encoder.layer.11.attention.output.dense.bias torch.Size([768])
bert_model.encoder.layer.11.attention.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.11.attention.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.11.intermediate.dense.weight torch.Size([3072, 768])
bert_model.encoder.layer.11.intermediate.dense.bias torch.Size([3072])
bert_model.encoder.layer.11.output.dense.weight torch.Size([768, 3072])
bert_model.encoder.layer.11.output.dense.bias torch.Size([768])
bert_model.encoder.layer.11.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.11.output.LayerNorm.bias torch.Size([768])
bert_model.pooler.dense.weight torch.Size([768, 768])
bert_model.pooler.dense.bias torch.Size([768])
fc.weight torch.Size([20, 768])
fc.bias torch.Size([20])
```

```
import random

seed = 123

random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

learning_rate = 0.001
epochs=10

# STEP 5: INSTANTIATE LOSS CLASS
criterion = nn.CrossEntropyLoss()

# STEP 6: INSTANTIATE OPTIMIZER CLASS

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Freeze embedding Layer

#freeze embeddings
#model.embed.weight.requires_grad  = False

# STEP 7: TRAIN THE MODEL
```

```
# STEP 7: TRAIN THE MODEL

train_losses= np.zeros(epochs)
valid_losses= np.zeros(epochs)


for epoch in range(epochs):
  t0= datetime.now()
  train_loss=[]

  model.train()
  for batch in train_dataloader:

    # forward pass
    output= model(batch[0].to(device))
    loss=criterion(output,batch[2].to(device))

    # set gradients to zero
    optimizer.zero_grad()

    # backward pass
    loss.backward()
    optimizer.step()
    train_loss.append(loss.item())

  train_loss=np.mean(train_loss)

  valid_loss=[]
  model.eval()
  with torch.no_grad():
    for batch in validation_dataloader:

      # forward pass
      output= model(batch[0].to(device))
      loss=criterion(output,batch[2].to(device))

      valid_loss.append(loss.item())

    valid_loss=np.mean(valid_loss)

  # save Losses
  train_losses[epoch]= train_loss
  valid_losses[epoch]= valid_loss
  dt= datetime.now()-t0
  print(f'Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}    Valid Loss: {
```

```
Epoch 1/10, Train Loss: 2.3017    Valid Loss: 1.5224, Duration: 0:02:56.166814
Epoch 2/10, Train Loss: 2.1739    Valid Loss: 1.4355, Duration: 0:02:56.013125
Epoch 3/10, Train Loss: 2.1419    Valid Loss: 1.4144, Duration: 0:02:55.931495
Epoch 4/10, Train Loss: 2.1274    Valid Loss: 1.3814, Duration: 0:02:55.973924
Epoch 5/10, Train Loss: 2.1361    Valid Loss: 1.3163, Duration: 0:02:55.929460
Epoch 6/10, Train Loss: 2.1258    Valid Loss: 1.3437, Duration: 0:02:55.988769
Epoch 7/10, Train Loss: 2.1261    Valid Loss: 1.2965, Duration: 0:02:56.029110
Epoch 8/10, Train Loss: 2.1219    Valid Loss: 1.3007, Duration: 0:02:56.030751
Epoch 9/10, Train Loss: 2.1134    Valid Loss: 1.3114, Duration: 0:02:56.041194
Epoch 10/10, Train Loss: 2.1219    Valid Loss: 1.3253, Duration: 0:02:55.974106
```

```
# Accuracy- write a function to get accuracy
# use this function to get accuracy and print accuracy
def get_accuracy(data_iter, model):
  model.eval()
  with torch.no_grad():
    correct =0
    total =0

    for batch in data_iter:

      output=model(batch[0].to(device))
      _,indices = torch.max(output,dim=1)
      correct+= (batch[2].to(device)==indices).sum().item()
      total += batch[2].shape[0]

    acc= correct/total

    return acc


train_acc = get_accuracy(train_dataloader, model)
valid_acc = get_accuracy(validation_dataloader, model)
print(f'Train acc: {train_acc:.4f},\t Valid acc: {valid_acc:.4f}')
```

```
Train acc: 0.6364,       Valid acc: 0.6220
```

```
# Accuracy- write a function to get accuracy
# use this function to get accuracy and print accuracy
def get_accuracy_v1(dataset, chunks, model):
  model.eval()
  indices = chunks.keys()
  print(len(indices))
  correct = 0
  total = 0
  for idx in indices:
    values = chunks[idx]
    logits = torch.zeros(1,20).to(device)
    for ix in values:
      with torch.no_grad():
        output=model(dataset[ix][0].view(1, -1).to(device))
        logits = logits + output
    _,index = torch.max(logits,dim=1)
    if index.item() == dataset[values[0]][2].item():
       correct += 1
    total += 1
  acc= correct/total
  return acc


test_acc = get_accuracy_v1(test_dataset, test_chunks, model)
print(f'Test acc: {test_acc:.4f}')
```

```
7532
Test acc: 0.4101
```

```
# Write a function to get predictions

def get_predictions(dataset, chunks, model):
  model.eval()
  indices = chunks.keys()
  print(len(indices))
  correct = 0
  total = 0
  predictions = []
  y_test = []
  for idx in indices:
    values = chunks[idx]
    logits = torch.zeros(1,20).to(device)
    for ix in values:
      with torch.no_grad():
        output=model(dataset[ix][0].view(1,-1).to(device))
        logits = logits + output
    _,index = torch.max(logits,dim=1)
    predictions.append(index.item())
    y_test.append(dataset[values[0]][2].item())
  return np.array(y_test), np.array(predictions)
```

```
y_test, predictions=get_predictions(test_dataset, test_chunks, model)
```

```
7532
```

```
predictions
```

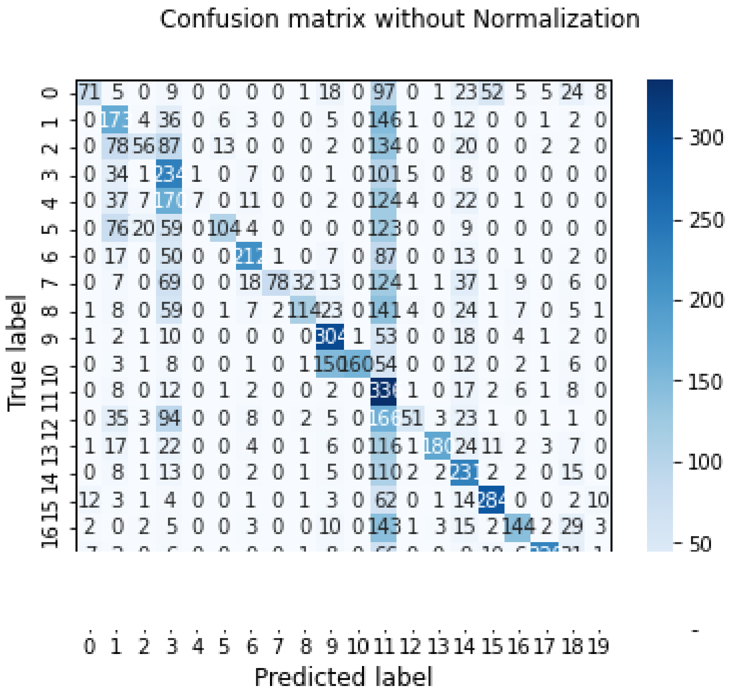
```
array([ 3,  1,  9, ...,  9,  3, 15])
```

```
# Confusion Matrix
cm=confusion_matrix(y_test,predictions)
cm
```

```
array([[ 71,   5,   0,   9,   0,   0,   0,   0,   1,  18,   0,  97,   0,
          1,  23,  52,   5,   5,  24,   8],
       [  0, 173,   4,  36,   0,   6,   3,   0,   0,   5,   0, 146,   1,
          0,  12,   0,   0,   1,   2,   0],
       [  0,  78,  56,  87,   0,  13,   0,   0,   0,   2,   0, 134,   0,
          0,  20,   0,   0,   2,   2,   0],
       [  0,  34,   1, 234,   1,   0,   7,   0,   0,   1,   0, 101,   5,
          0,   8,   0,   0,   0,   0,   0],
       [  0,  37,   7, 170,   7,   0,  11,   0,   0,   2,   0, 124,   4,
          0,  22,   0,   1,   0,   0,   0],
       [  0,  76,  20,  59,   0, 104,   4,   0,   0,   0,   0, 123,   0,
          0,   9,   0,   0,   0,   0,   0],
       [  0,  17,   0,  50,   0,   0, 212,   1,   0,   7,   0,  87,   0,
          0,  13,   0,   1,   0,   2,   0],
       [  0,   7,   0,  69,   0,   0,  18,  78,  32,  13,   0, 124,   1,
          1,  37,   1,   9,   0,   6,   0],
       [  1,   8,   0,  59,   0,   1,   7,   2, 114,  23,   0, 141,   4,
          0,  24,   1,   7,   0,   5,   1],
       [  1,   2,   1,  10,   0,   0,   0,   0,   0, 304,   1,  53,   0,
          0,  18,   0,   4,   1,   2,   0],
```

```
       [  0,   3,   1,   8,   0,   0,   1,   0,   1, 150, 160,  54,   0,
          0,  12,   0,   2,   1,   6,   0],
       [  0,   8,   0,  12,   0,   1,   2,   0,   0,   2,   0, 336,   1,
          0,  17,   2,   6,   1,   8,   0],
       [  0,  35,   3,  94,   0,   0,   8,   0,   2,   5,   0, 166,  51,
          3,  23,   1,   0,   1,   1,   0],
       [  1,  17,   1,  22,   0,   0,   4,   0,   1,   6,   0, 116,   1,
        180,  24,  11,   2,   3,   7,   0],
       [  0,   8,   1,  13,   0,   0,   2,   0,   1,   5,   0, 110,   2,
          2, 231,   2,   2,   0,  15,   0],
       [ 12,   3,   1,   4,   0,   0,   1,   0,   1,   3,   0,  62,   0,
          1,  14, 284,   0,   0,   2,  10],
       [  2,   0,   2,   5,   0,   0,   3,   0,   0,  10,   0, 143,   1,
          3,  15,   2, 144,   2,  29,   3],
       [  7,   2,   0,   6,   0,   0,   0,   0,   1,   8,   0,  66,   0,
          0,   9,  10,   6, 229,  31,   1],
       [  1,   4,   3,   3,   0,   0,   0,   0,   1,  11,   0,  95,   0,
          1,  13,   5,  54,   6, 110,   3],
       [ 14,   3,   0,   4,   0,   0,   2,   0,   0,   7,   0,  92,   0,
          0,  17,  73,  18,   0,  10,  11]])
```

```
# Write a function to print confusion matrix
# plot confusion matrix
# need to import confusion_matrix from sklearn for this function to work
# need to import seaborn as sns
# import seaborn as sns
# import matplotlib.pyplot as plt
# from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(y_true,y_pred,normalize=None):
  cm=confusion_matrix(y_true,y_pred,normalize=normalize)
  fig, ax = plt.subplots(figsize=(6,5))
  if normalize == None:
    fmt='d'
    fig.suptitle('Confusion matrix without Normalization', fontsize=12)

  else :
    fmt='0.2f'
    fig.suptitle('Normalized confusion matrix', fontsize=12)

  ax=sns.heatmap(cm,cmap=plt.cm.Blues,annot=True,fmt=fmt)
  ax.axhline(y=0, color='k',linewidth=1)
  ax.axhline(y=cm.shape[1], color='k',linewidth=2)
  ax.axvline(x=0, color='k',linewidth=1)
  ax.axvline(x=cm.shape[0], color='k',linewidth=2)

  ax.set_xlabel('Predicted label', fontsize=12)
  ax.set_ylabel('True label', fontsize=12)


plot_confusion_matrix(y_test,predictions)
```


Confusion matrix without Normalization
True label
Predicted label
300
250
200
150
100
50