```
!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: regex!=2019.12.17 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: sacremoses in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: packaging in /usr/local/lib/python3.6/dist-packages (f
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: requests 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: filelock in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: click in /usr/local/lib/python3.6/dist-packages (from
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: pyparsing>=2.0.2 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dist-pa
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
```

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 accelerator, ')
  print('and then re-execute this cell.')
else:
  print(gpu_info)
```

```
print(gpu_info)
```

```
Tue Dec  1 23:36:51 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   43C    P0    27W / 250W |      0MiB / 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',
```

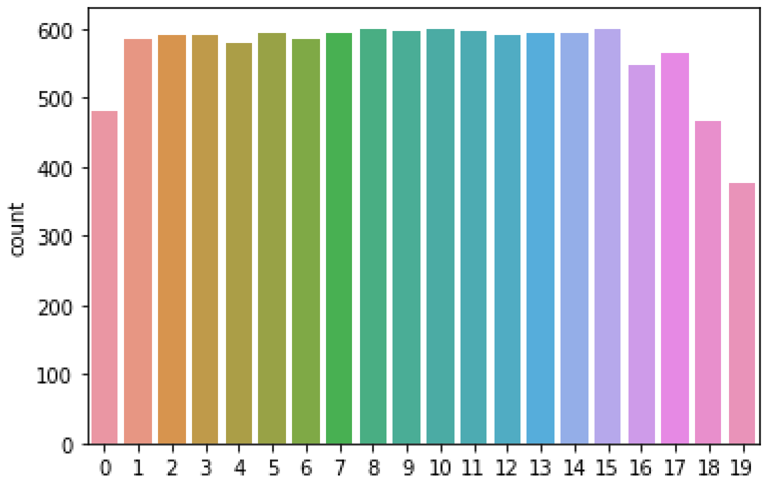
```
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',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

```
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...
```

```
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []

# 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=True, #Truncate the sentences
                        max_length = 140,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(train.target)

# Print sentence 0, now as a list of IDs.
print('Original: ', train.data[0])
print('Token IDs:', input_ids[0])
```

```
/usr/local/lib/python3.6/dist-packages/transformers/tokenization_utils_base.py:2142:
  FutureWarning,
Original:  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.
Token IDs: tensor([  101,  1045,  2001,  6603,  2065,  3087,  2041,  2045,  2071,  4
         7138,  2368,  2033,  2006,  2023,  2482,  1045,  2387,  1996,  2060,
         2154,  1012,  2009,  2001,  1037,  1016,  1011,  2341,  2998,  2482,
         1010,  2246,  2000,  2022,  2013,  1996,  2397, 20341,  1013,  2220,
        17549,  1012,  2009,  2001,  2170,  1037,  5318,  4115,  1012,  1996,
         4303,  2020,  2428,  2235,  1012,  1999,  2804,  1010,  1996,  2392,
        21519,  2001,  3584,  2013,  1996,  2717,  1997,  1996,  2303,  1012,
         2023,  2003,  2035,  1045,  2113,  1012,  2065,  3087,  2064,  2425,
```

```
         4168,  1037,  2944,  2171,  1010,  3194, 28699,  2015,  1010,  2086,
         1997,  2537,  1010,  2073,  2023,  2482,  2003,  2081,  1010,  2381,
         1010,  2030,  3649, 18558,  2017,  2031,  2006,  2023, 24151,  2559,
         2482,  1010,  3531,  1041,  1011,  5653,  1012,   102,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0])
```

```
test_input_ids = []
test_attention_masks = []

# 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=True, #Truncate the sentences
                        max_length = 140,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    test_input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    test_attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
test_input_ids = torch.cat(test_input_ids, dim=0)
test_attention_masks = torch.cat(test_attention_masks, dim=0)
test_labels = torch.tensor(test.target)

# Print sentence 0, now as a list of IDs.
print('Original: ', test.data[0])
print('Token IDs:', test_input_ids[0])
```

```
/usr/local/lib/python3.6/dist-packages/transformers/tokenization_utils_base.py:2142:
  FutureWarning,
Original:  I am a little confused on all of the models of the 88-89 bonnevilles.
I have heard of the LE SE LSE SSE SSEI. Could someone tell me the
differences are far as features or performance. I am also curious to
know what the book value is for prefereably the 89 model. And how much
less than book value can you usually get them for. In other words how
much are they in demand this time of year. I have heard that the mid-spring
early summer is the best time to buy.
Token IDs: tensor([  101,  1045,  2572,  1037,  2210,  5457,  2006,  2035,  1997,  1
         4275,  1997,  1996,  6070,  1011,  6486, 19349, 21187,  2015,  1012,
```

```
1045,  2031,  2657,  1997,  1996,  3393,  7367,  1048,  3366,  7020,
2063,  7020,  7416,  1012,  2071,  2619,  2425,  2033,  1996,  5966,
2024,  2521,  2004,  2838,  2030,  2836,  1012,  1045,  2572,  2036,
8025,  2000,  2113,  2054,  1996,  2338,  3643,  2003,  2005,  9544,
5243,  6321,  1996,  6486,  2944,  1012,  1998,  2129,  2172,  2625,
2084,  2338,  3643,  2064,  2017,  2788,  2131,  2068,  2005,  1012,
1999,  2060,  2616,  2129,  2172,  2024,  2027,  1999,  5157,  2023,
2051,  1997,  2095,  1012,  1045,  2031,  2657,  2008,  1996,  3054,
1011,  3500,  2220,  2621,  2003,  1996,  2190,  2051,  2000,  4965,
1012,   102,     0,     0,     0,     0,     0,     0,     0,     0,
   0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
   0,     0,     0,     0,     0,     0,     0,     0,     0,     0])
```

```
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)))
```

```
10,182 training samples
1,132 validation samples
7,532 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 sequentially.
            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')


bert_model
```

```
              (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)
          )
        )
        (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)
          )
```

```
# 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, dropout_
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)
```

```
      (dropout): Dropout(p=0.3, 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

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: {valid_los
```

```
Epoch 1/10, Train Loss: 2.7517    Valid Loss: 2.3021, Duration: 0:00:55.957106
Epoch 2/10, Train Loss: 2.5580    Valid Loss: 2.1465, Duration: 0:00:55.912594
Epoch 3/10, Train Loss: 2.4920    Valid Loss: 2.0520, Duration: 0:00:55.919293
Epoch 4/10, Train Loss: 2.4778    Valid Loss: 2.0167, Duration: 0:00:55.986673
Epoch 5/10, Train Loss: 2.4475    Valid Loss: 1.9217, Duration: 0:00:55.934424
Epoch 6/10, Train Loss: 2.4218    Valid Loss: 1.9241, Duration: 0:00:55.974224
Epoch 7/10, Train Loss: 2.4202    Valid Loss: 1.8931, Duration: 0:00:55.915101
Epoch 8/10, Train Loss: 2.4142    Valid Loss: 1.8560, Duration: 0:00:55.933802
Epoch 9/10, Train Loss: 2.4068    Valid Loss: 1.9132, Duration: 0:00:55.909939
Epoch 10/10, Train Loss: 2.3867    Valid Loss: 1.8397, Duration: 0:00:55.986418
```

```
# 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)
```

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

```
Train acc: 0.5084,       Valid acc: 0.4735,      Test acc: 0.4494
```

```
# Write a function to get predictions

def get_predictions(test_iter, model):
  model.eval()
  with torch.no_grad():
    predictions= np.array([])
    y_test= np.array([])

    for batch in test_iter:

      output=model(batch[0].to(device))
      _,indices = torch.max(output,dim=1)
      predictions=np.concatenate((predictions,indices.cpu().numpy()))
      y_test = np.concatenate((y_test,batch[2].numpy()))

  return y_test, predictions
```

```
y_test, predictions=get_predictions(test_dataloader, model)
```

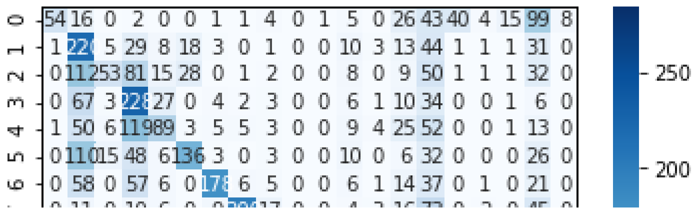
```
# Confusion Matrix
cm=confusion_matrix(y_test,predictions)
cm
```

```
array([[ 54,  16,   0,   2,   0,   0,   1,   1,   4,   0,   1,   5,   0,
         26,  43,  40,   4,  15,  99,   8],
       [  1, 220,   5,  29,   8,  18,   3,   0,   1,   0,   0,  10,   3,
         13,  44,   1,   1,   1,  31,   0],
       [  0, 112,  53,  81,  15,  28,   0,   1,   2,   0,   0,   8,   0,
          9,  50,   1,   1,   1,  32,   0],
       [  0,  67,   3, 228,  27,   0,   4,   2,   3,   0,   0,   6,   1,
         10,  34,   0,   0,   1,   6,   0],
       [  1,  50,   6, 119,  89,   3,   5,   5,   3,   0,   0,   9,   4,
         25,  52,   0,   0,   1,  13,   0],
       [  0, 110,  15,  48,   6, 136,   3,   0,   3,   0,   0,  10,   0,
          6,  32,   0,   0,   0,  26,   0],
       [  0,  58,   0,  57,   6,   0, 178,   6,   5,   0,   0,   6,   1,
         14,  37,   0,   1,   0,  21,   0],
       [  0,  11,   0,  10,   6,   0,   9, 200,  17,   0,   0,   4,   3,
         16,  73,   0,   2,   0,  45,   0],
       [  2,  22,   0,  14,   7,   0,   5,  41, 144,   1,   0,   4,   4,
         12,  63,   0,   8,   0,  70,   1],
       [  1,  17,   0,   2,   2,   0,   0,   2,   2, 242,  15,   1,   0,
          8,  48,   0,   1,   1,  54,   1],
       [  1,   9,   1,   1,   0,   1,   1,   2,   1,  10, 261,   0,   0,
         10,  53,   1,   0,   2,  45,   0],
       [  1,  29,   4,  25,   3,   2,   1,   0,   0,   1,   0, 167,   3,
         10,  55,   0,   9,   6,  80,   0],
       [  0,  58,   1,  81,  13,   0,   4,  10,   6,   0,   0,  17,  78,
         44,  54,   3,   0,   0,  24,   0],
       [  3,  23,   0,   5,   0,   1,   0,   2,   3,   0,   0,   2,   4,
        259,  58,   4,   0,   2,  29,   1],
```

```
       [  2,  16,   0,   7,   1,   0,   1,   3,   1,   0,   0,   6,   5,
         13, 284,   3,   1,   1,  50,   0],
       [ 18,  10,   0,   7,   0,   0,   0,   1,   1,   0,   0,   5,   0,
         22,  34, 235,   2,   1,  45,  17],
       [  5,  20,   0,   1,   0,   0,   0,   5,   6,   0,   0,  15,   1,
         31,  43,   1, 110,   3, 122,   1],
       [  5,   4,   1,   1,   1,   0,   0,   1,   0,   1,   0,   3,   0,
          4,  37,   2,   1, 229,  82,   4],
       [  3,   5,   0,   1,   1,   0,   0,   1,   3,   0,   0,   4,   0,
         19,  34,   5,  38,   5, 189,   2],
       [ 16,   5,   0,   0,   1,   0,   1,   3,   3,   1,   0,   3,   0,
         14,  38,  61,  14,   4,  58,  29]])
```

```
# 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



BERT with 128


```
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...
```

```
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []

# 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=True, #Truncate the sentences
                        max_length = 128,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(train.target)
```

```
# Print sentence 0, now as a list of IDs.
print('Original: ', train.data[0])
print('Token IDs:', input_ids[0])
```

```
/usr/local/lib/python3.6/dist-packages/transformers/tokenization_utils_base.py:2142:
  FutureWarning,
Original:  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.
Token IDs: tensor([  101,  1045,  2001,  6603,  2065,  3087,  2041,  2045,  2071,  43
         7138,  2368,  2033,  2006,  2023,  2482,  1045,  2387,  1996,  2060,
         2154,  1012,  2009,  2001,  1037,  1016,  1011,  2341,  2998,  2482,
         1010,  2246,  2000,  2022,  2013,  1996,  2397, 20341,  1013,  2220,
        17549,  1012,  2009,  2001,  2170,  1037,  5318,  4115,  1012,  1996,
         4303,  2020,  2428,  2235,  1012,  1999,  2804,  1010,  1996,  2392,
        21519,  2001,  3584,  2013,  1996,  2717,  1997,  1996,  2303,  1012,
         2023,  2003,  2035,  1045,  2113,  1012,  2065,  3087,  2064,  2425,
         4168,  1037,  2944,  2171,  1010,  3194, 28699,  2015,  1010,  2086,
         1997,  2537,  1010,  2073,  2023,  2482,  2003,  2081,  1010,  2381,
         1010,  2030,  3649, 18558,  2017,  2031,  2006,  2023, 24151,  2559,
         2482,  1010,  3531,  1041,  1011,  5653,  1012,   102,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0])
```

```
test_input_ids = []
test_attention_masks = []

# 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=True, #Truncate the sentences
                        max_length = 128,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    test_input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    test_attention_masks.append(encoded_dict['attention_mask'])
```

```
# Convert the lists into tensors.
test_input_ids = torch.cat(test_input_ids, dim=0)
test_attention_masks = torch.cat(test_attention_masks, dim=0)
test_labels = torch.tensor(test.target)

# Print sentence 0, now as a list of IDs.
print('Original: ', test.data[0])
print('Token IDs:', test_input_ids[0])
```

```
/usr/local/lib/python3.6/dist-packages/transformers/tokenization_utils_base.py:2142:
  FutureWarning,
Original:  I am a little confused on all of the models of the 88-89 bonnevilles.
I have heard of the LE SE LSE SSE SSEI. Could someone tell me the
differences are far as features or performance. I am also curious to
know what the book value is for prefereably the 89 model. And how much
less than book value can you usually get them for. In other words how
much are they in demand this time of year. I have heard that the mid-spring
early summer is the best time to buy.
Token IDs: tensor([  101,  1045,  2572,  1037,  2210,  5457,  2006,  2035,  1997,  1
         4275,  1997,  1996,  6070,  1011,  6486, 19349, 21187,  2015,  1012,
         1045,  2031,  2657,  1997,  1996,  3393,  7367,  1048,  3366,  7020,
         2063,  7020,  7416,  1012,  2071,  2619,  2425,  2033,  1996,  5966,
         2024,  2521,  2004,  2838,  2030,  2836,  1012,  1045,  2572,  2036,
         8025,  2000,  2113,  2054,  1996,  2338,  3643,  2003,  2005,  9544,
         5243,  6321,  1996,  6486,  2944,  1012,  1998,  2129,  2172,  2625,
         2084,  2338,  3643,  2064,  2017,  2788,  2131,  2068,  2005,  1012,
         1999,  2060,  2616,  2129,  2172,  2024,  2027,  1999,  5157,  2023,
         2051,  1997,  2095,  1012,  1045,  2031,  2657,  2008,  1996,  3054,
         1011,  3500,  2220,  2621,  2003,  1996,  2190,  2051,  2000,  4965,
         1012,   102,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0])
```

```
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)))
```

```
10,182 training samples
1,132 validation samples
7,532 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 sequentially.
            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')
```

```
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, dropout_
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

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: {valid_los
```

```
Epoch 1/10, Train Loss: 2.7181    Valid Loss: 2.1756, Duration: 0:00:52.097921
Epoch 2/10, Train Loss: 2.5172    Valid Loss: 2.0393, Duration: 0:00:51.956894
Epoch 3/10, Train Loss: 2.4376    Valid Loss: 1.9340, Duration: 0:00:51.921139
Epoch 4/10, Train Loss: 2.4005    Valid Loss: 1.8955, Duration: 0:00:51.917111
Epoch 5/10, Train Loss: 2.3927    Valid Loss: 1.8542, Duration: 0:00:51.926076
Epoch 6/10, Train Loss: 2.3700    Valid Loss: 1.7962, Duration: 0:00:51.939493
Epoch 7/10, Train Loss: 2.3622    Valid Loss: 1.7845, Duration: 0:00:51.973877
Epoch 8/10, Train Loss: 2.3467    Valid Loss: 1.7723, Duration: 0:00:51.916696
Epoch 9/10, Train Loss: 2.3558    Valid Loss: 1.7830, Duration: 0:00:51.929136
Epoch 10/10, Train Loss: 2.3251    Valid Loss: 1.7442, Duration: 0:00:51.955068
```

```
# 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)
test_acc = get_accuracy(test_dataloader ,model)
print(f'Train acc: {train_acc:.4f},\t Valid acc: {valid_acc:.4f},\t Test acc: {test_acc:.4
```

```
Train acc: 0.5166,       Valid acc: 0.4806,      Test acc: 0.4522
```

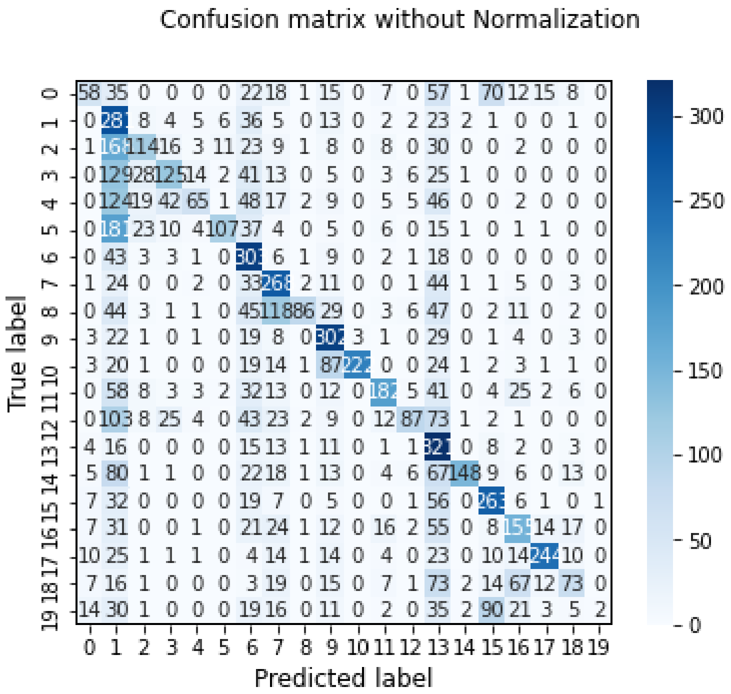
```
# Write a function to get predictions

def get_predictions(test_iter, model):
  model.eval()
  with torch.no_grad():
    predictions= np.array([])
    y_test= np.array([])

    for batch in test_iter:

      output=model(batch[0].to(device))
      _,indices = torch.max(output,dim=1)
      predictions=np.concatenate((predictions,indices.cpu().numpy()))
      y_test = np.concatenate((y_test,batch[2].numpy()))

  return y_test, predictions


y_test, predictions=get_predictions(test_dataloader, model)


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

```
array([[ 58,  35,   0,   0,   0,   0,  22,  18,   1,  15,   0,   7,   0,
         57,   1,  70,  12,  15,   8,   0],
       [  0, 281,   8,   4,   5,   6,  36,   5,   0,  13,   0,   2,   2,
         23,   2,   1,   0,   0,   1,   0],
       [  1, 168, 114,  16,   3,  11,  23,   9,   1,   8,   0,   8,   0,
         30,   0,   0,   2,   0,   0,   0],
       [  0, 129,  28, 125,  14,   2,  41,  13,   0,   5,   0,   3,   6,
         25,   1,   0,   0,   0,   0,   0],
       [  0, 124,  19,  42,  65,   1,  48,  17,   2,   9,   0,   5,   5,
```

```
        46,   0,   0,   2,   0,   0,   0],
      [  0, 181,  23,  10,   4, 107,  37,   4,   0,   5,   0,   6,   0,
        15,   1,   0,   1,   1,   0,   0],
      [  0,  43,   3,   3,   1,   0, 303,   6,   1,   9,   0,   2,   1,
        18,   0,   0,   0,   0,   0,   0],
      [  1,  24,   0,   0,   2,   0,  33, 268,   2,  11,   0,   0,   1,
        44,   1,   1,   5,   0,   3,   0],
      [  0,  44,   3,   1,   1,   0,  45, 118,  86,  29,   0,   3,   6,
        47,   0,   2,  11,   0,   2,   0],
      [  3,  22,   1,   0,   1,   0,  19,   8,   0, 302,   3,   1,   0,
        29,   0,   1,   4,   0,   3,   0],
      [  3,  20,   1,   0,   0,   0,  19,  14,   1,  87, 222,   0,   0,
        24,   1,   2,   3,   1,   1,   0],
      [  0,  58,   8,   3,   3,   2,  32,  13,   0,  12,   0, 182,   5,
        41,   0,   4,  25,   2,   6,   0],
      [  0, 103,   8,  25,   4,   0,  43,  23,   2,   9,   0,  12,  87,
        73,   1,   2,   1,   0,   0,   0],
      [  4,  16,   0,   0,   0,   0,  15,  13,   1,  11,   0,   1,   1,
       321,   0,   8,   2,   0,   3,   0],
      [  5,  80,   1,   1,   0,   0,  22,  18,   1,  13,   0,   4,   6,
        67, 148,   9,   6,   0,  13,   0],
      [  7,  32,   0,   0,   0,   0,  19,   7,   0,   5,   0,   0,   1,
        56,   0, 263,   6,   1,   0,   1],
      [  7,  31,   0,   0,   1,   0,  21,  24,   1,  12,   0,  16,   2,
        55,   0,   8, 155,  14,  17,   0],
      [ 10,  25,   1,   1,   1,   0,   4,  14,   1,  14,   0,   4,   0,
        23,   0,  10,  14, 244,  10,   0],
      [  7,  16,   1,   0,   0,   0,   3,  19,   0,  15,   0,   7,   1,
        73,   2,  14,  67,  12,  73,   0],
      [ 14,  30,   1,   0,   0,   0,  19,  16,   0,  11,   0,   2,   0,
        35,   2,  90,  21,   3,   5,   2]])
```

```
# 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_xlabel('Predicted label', fontsize=12)
  ax.set_ylabel('True label', fontsize=12)


plot_confusion_matrix(y_test,predictions)
```



BERT 512 Features

```
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...


# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []

# 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=True. #Truncate the sentences
```

```
                        max_length = 512,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(train.target)

# Print sentence 0, now as a list of IDs.
print('Original: ', train.data[0])
print('Token IDs:', input_ids[0])
```

```
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.
Token IDs: tensor([  101,  1045,  2001,  6603,  2065,  3087,  2041,  2045,  2071,
         7138,  2368,  2033,  2006,  2023,  2482,  1045,  2387,  1996,  2060,
         2154,  1012,  2009,  2001,  1037,  1016,  1011,  2341,  2998,  2482,
         1010,  2246,  2000,  2022,  2013,  1996,  2397, 20341,  1013,  2220,
        17549,  1012,  2009,  2001,  2170,  1037,  5318,  4115,  1012,  1996,
         4303,  2020,  2428,  2235,  1012,  1999,  2804,  1010,  1996,  2392,
        21519,  2001,  3584,  2013,  1996,  2717,  1997,  1996,  2303,  1012,
         2023,  2003,  2035,  1045,  2113,  1012,  2065,  3087,  2064,  2425,
         4168,  1037,  2944,  2171,  1010,  3194, 28699,  2015,  1010,  2086,
         1997,  2537,  1010,  2073,  2023,  2482,  2003,  2081,  1010,  2381,
         1010,  2030,  3649, 18558,  2017,  2031,  2006,  2023, 24151,  2559,
         2482,  1010,  3531,  1041,  1011,  5653,  1012,   102,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,

            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
```

```
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
0,     0])
```

```python
test_input_ids = []
test_attention_masks = []

# 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=True, #Truncate the sentences
                        max_length = 512,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    test_input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    test_attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
test_input_ids = torch.cat(test_input_ids, dim=0)
test_attention_masks = torch.cat(test_attention_masks, dim=0)
test_labels = torch.tensor(test.target)

# Print sentence 0, now as a list of IDs.
print('Original: ', test.data[0])
```

```
print('Original: ', test.data[0])
print('Token IDs:', test_input_ids[0])
```

```
Original:  I am a little confused on all of the models of the 88-89 bonnevilles.
I have heard of the LE SE LSE SSE SSEI. Could someone tell me the
differences are far as features or performance. I am also curious to
know what the book value is for prefereably the 89 model. And how much
less than book value can you usually get them for. In other words how
much are they in demand this time of year. I have heard that the mid-spring
early summer is the best time to buy.
Token IDs: tensor([  101,  1045,  2572,  1037,  2210,  5457,  2006,  2035,  1997,
         4275,  1997,  1996,  6070,  1011,  6486, 19349, 21187,  2015,  1012,
         1045,  2031,  2657,  1997,  1996,  3393,  7367,  1048,  3366,  7020,
         2063,  7020,  7416,  1012,  2071,  2619,  2425,  2033,  1996,  5966,
         2024,  2521,  2004,  2838,  2030,  2836,  1012,  1045,  2572,  2036,
         8025,  2000,  2113,  2054,  1996,  2338,  3643,  2003,  2005,  9544,
         5243,  6321,  1996,  6486,  2944,  1012,  1998,  2129,  2172,  2625,
         2084,  2338,  3643,  2064,  2017,  2788,  2131,  2068,  2005,  1012,
         1999,  2060,  2616,  2129,  2172,  2024,  2027,  1999,  5157,  2023,
         2051,  1997,  2095,  1012,  1045,  2031,  2657,  2008,  1996,  3054,
         1011,  3500,  2220,  2621,  2003,  1996,  2190,  2051,  2000,  4965,
         1012,   102,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,

            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0])
```

```
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)))
```

```
10,182 training samples
1,132 validation samples
7,532 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 sequentially.
            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')
```

```
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, dropout_
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)
```

```
            (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)
```

```
    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

    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: {valid_los
```

```
Epoch 1/10, Train Loss: 3.0419    Valid Loss: 2.7815, Duration: 0:03:39.286841
Epoch 2/10, Train Loss: 2.9293    Valid Loss: 2.7077, Duration: 0:03:39.195885
Epoch 3/10, Train Loss: 2.8992    Valid Loss: 2.6527, Duration: 0:03:39.192333
Epoch 4/10, Train Loss: 2.8657    Valid Loss: 2.6342, Duration: 0:03:39.245665
Epoch 5/10, Train Loss: 2.8500    Valid Loss: 2.5605, Duration: 0:03:39.192875
Epoch 6/10, Train Loss: 2.8464    Valid Loss: 2.5700, Duration: 0:03:39.200640
Epoch 7/10, Train Loss: 2.8387    Valid Loss: 2.5003, Duration: 0:03:39.328104
Epoch 8/10, Train Loss: 2.8185    Valid Loss: 2.4943, Duration: 0:03:39.246588
Epoch 9/10, Train Loss: 2.8241    Valid Loss: 2.5048, Duration: 0:03:39.224581
Epoch 10/10, Train Loss: 2.8186    Valid Loss: 2.4282, Duration: 0:03:39.227243
```

```
# 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)
test_acc = get_accuracy(test_dataloader ,model)
print(f'Train acc: {train_acc:.4f},\t Valid acc: {valid_acc:.4f},\t Test acc: {test_acc:.4
```

```
Train acc: 0.3407,       Valid acc: 0.3145,      Test acc: 0.2872
```

```
# Write a function to get predictions

def get_predictions(test_iter, model):
  model.eval()
  with torch.no_grad():
    predictions= np.array([])
    y_test= np.array([])
```

```
    for batch in test_iter:

      output=model(batch[0].to(device))
      _,indices = torch.max(output,dim=1)
      predictions=np.concatenate((predictions,indices.cpu().numpy()))
      y_test = np.concatenate((y_test,batch[2].numpy()))

  return y_test, predictions


y_test, predictions=get_predictions(test_dataloader, model)


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

```
array([[  9,   6,   0,   0,   0,   0,  19,  22,   0,  33,   0,   3,   0,
         55,   0,  69,   9,   8,  75,  11],
       [  0, 111,  26,  17,   0,   5,  69,  36,   0,  32,   1,   2,   2,
         53,   2,   1,   0,   1,  31,   0],
       [  0,  59,  54,  41,   0,  12,  65,  33,   0,  41,   0,   1,   2,
         48,   0,   1,   0,   0,  37,   0],
       [  0,  42,  28,  85,   0,   2,  59,  61,   0,  34,   0,   2,   8,
         41,   1,   0,   0,   1,  28,   0],
       [  0,  51,  10,  48,   3,   0,  68,  59,   0,  41,   0,   0,   4,
         64,   0,   1,   0,   0,  36,   0],
       [  0,  68,  33,  39,   0,  49,  74,  27,   0,  28,   0,   2,   2,
         52,   0,   1,   0,   0,  20,   0],
       [  0,   8,   2,  13,   0,   0, 283,  28,   0,  29,   2,   0,   0,
         17,   0,   0,   0,   0,   8,   0],
       [  0,   9,   0,   2,   0,   0,  51, 205,   0,  55,   1,   1,   1,
         40,   0,   0,   1,   0,  30,   0],
       [  0,  25,   1,   1,   0,   0,  56, 148,  10,  62,   0,   1,   1,
         39,   0,   1,   4,   1,  48,   0],
       [  0,  12,   1,   0,   0,   0,  34,  26,   0, 231,   8,   0,   0,
         35,   0,   4,   0,   4,  42,   0],
       [  0,   7,   0,   1,   0,   0,  24,  40,   0,  85, 139,   0,   1,
         43,   0,   3,   0,   3,  53,   0],
       [  0,  15,  14,   9,   0,   1,  32,  26,   0,  48,   0,  55,   6,
         68,   0,   8,  15,   6,  93,   0],
       [  0,  22,  10,  18,   0,   3,  64,  79,   2,  36,   0,   4,  32,
         78,   0,   7,   3,   4,  31,   0],
       [  0,  10,   0,   0,   0,   0,  28,  33,   0,  43,   0,   0,   0,
        231,   0,   7,   0,   2,  42,   0],
       [  0,   8,   5,   2,   0,   1,  30,  64,   0,  37,   0,   0,   2,
         82,  76,   7,   4,   2,  74,   0],
       [  1,   8,   2,   1,   0,   0,  18,   8,   0,  21,   0,   0,   0,
         47,   0, 225,   1,   5,  54,   7],
       [  0,  10,   1,   1,   0,   0,  26,  36,   0,  33,   0,   2,   0,
         74,   0,   9,  68,   6,  97,   1],
       [  0,   7,   0,   1,   0,   0,  15,  14,   0,  36,   1,   1,   0,
         33,   0,  26,   4, 175,  61,   2],
       [  2,   3,   3,   1,   0,   0,   5,  19,   0,  27,   0,   1,   0,
         73,   0,  13,  34,   9, 120,   0],
       [  3,   1,   0,   0,   0,   0,  18,  25,   0,  35,   0,   0,   0,
         38,   1,  70,   3,   7,  48,   2]])
```

```
# 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)
```