Start coding or generate with AI.

```
from google.colab import drive
# Mount the Google Drive to access files stored there
drive.mount('/content/drive')

# Install the latest version of torchtext library quietly without showing output
!pip install torchtext -qq
!pip install transformers evaluate wandb datasets accelerate -U -qq ## NEW LINES ##
basepath = '/content/drive/MyDrive/data/'
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mour
```

```
import pandas as pd
import numpy as np
```

```
# Importing PyTorch library for tensor computations and neural network modules
import torch
import torch.nn as nn

# For working with textual data vocabularies and for displaying model summaries
from torchtext.vocab import vocab

# General-purpose Python libraries for random number generation and numerical operations
import random
import numpy as np

# Utilities for efficient serialization/deserialization of Python objects and for element ta
import joblib
from collections import Counter

# For creating lightweight attribute classes and for partial function application
from functools import partial

# For filesystem path handling, generating and displaying confusion matrices, and date-time
from pathlib import Path
from sklearn.metrics import confusion_matrix
from datetime import datetime

# For plotting and visualization
import matplotlib.pyplot as plt
import seaborn as sns
# %matplotlib inline

### NEW ##########################
# imports from Huggingface ecosystem
from transformers.modeling_outputs import SequenceClassifierOutput
from transformers import PreTrainedModel, PretrainedConfig
from transformers import TrainingArguments, Trainer
from datasets import Dataset
import evaluate

# wandb library
import wandb


base_folder = Path(basepath)
data_folder = base_folder/'datasets'
model_folder = base_folder/'models'
custom_functions = base_folder/'custom-functions'
model_folder.mkdir(exist_ok=True, parents = True)
```

```
train_df = pd.read_csv(data_folder/'train_twitter.csv')
train_texts = train_df["Tweet"]
train_labels = train_df.drop(columns=['ID','Tweet'])

test_df = pd.read_csv(data_folder/'test_twitter.csv')
test_texts = test_df["Tweet"].values
test_labels = test_df.drop(columns=['ID','Tweet']).values


from sklearn.model_selection import train_test_split
train_texts, valid_texts, train_labels, valid_labels = train_test_split(train_texts, train_l
train_texts = train_texts.values
valid_texts = valid_texts.values
train_labels = train_labels.values
valid_labels = valid_labels.values


trainset = Dataset.from_dict({
    'texts': train_texts,
    'labels': train_labels
})

validset = Dataset.from_dict({
    'texts': valid_texts,
    'labels': valid_labels
})

testset = Dataset.from_dict({
    'texts': test_texts,
    'labels': test_labels
})


print(trainset)
print(trainset.features)
print(trainset[1])
```

```
Dataset({
    features: ['texts', 'labels'],
    num_rows: 4634
})
{'texts': Value(dtype='string', id=None), 'labels': Sequence(feature=Value(dtype='int64'
{'texts': 'I got a short fuse when im sober.', 'labels': [1, 0, 1, 0, 0, 0, 0, 0, 1, 0,
```

```
class CustomConfig(PretrainedConfig):
  def __init__(self, vocab_size=0, embedding_dim=0, hidden_dim1=0, hidden_dim2=0, num_labels
      super().__init__()
      self.vocab_size = vocab_size
      self.embedding_dim = embedding_dim
      self.hidden_dim1 = hidden_dim1
      self.hidden_dim2 = hidden_dim2
      self.num_labels = num_labels


class CustomMLP(PreTrainedModel):
    config_class = CustomConfig

    def __init__(self, config):
        super().__init__(config)

        self.embedding_bag = nn.EmbeddingBag(config.vocab_size, config.embedding_dim)
        self.layers = nn.Sequential(
            nn.Linear(config.embedding_dim, config.hidden_dim1),
            nn.BatchNorm1d(num_features=config.hidden_dim1),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(config.hidden_dim1, config.hidden_dim2),
            nn.BatchNorm1d(num_features=config.hidden_dim2),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(config.hidden_dim2, config.num_labels)
        )

    def forward(self, input_ids, offsets, labels=None):
        embed_out = self.embedding_bag(input_ids, offsets)
        logits = self.layers(embed_out)
        loss = None
        if labels is not None:
            loss_fct = nn.BCEWithLogitsLoss()
            loss = loss_fct(logits, labels)
        return SequenceClassifierOutput(
            loss=loss,
            logits=logits
        )
```

```
def get_vocab(dataset, min_freq=1):
    """
    Generate a vocabulary from a dataset.

    Args:
        dataset (Dataset): A Hugging Face Dataset object. The dataset should
                           have a key 'texts' that contains the text data.
        min_freq (int): The minimum frequency for a token to be included in
                        the vocabulary.

    Returns:
        torchtext.vocab.Vocab: Vocabulary object containing tokens from the
                               dataset that meet or exceed the specified
                               minimum frequency. It also includes a special
                               '<unk>' token for unknown words.
    """
    # Initialize a counter object to hold token frequencies
    counter = Counter()

    # Update the counter with tokens from each text in the dataset
    # Iterating through texts in the dataset
    for text in dataset['texts']:  ###### Change from previous function ####
        counter.update(str(text).split())

    # Create a vocabulary using the counter object
    # Tokens that appear fewer times than `min_freq` are excluded
    my_vocab = vocab(counter, min_freq=min_freq)

    # Insert a '<unk>' token at index 0 to represent unknown words
    my_vocab.insert_token('<unk>', 0)

    # Set the default index to 0
    # This ensures that any unknown word will be mapped to '<unk>'
    my_vocab.set_default_index(0)

    return my_vocab


# Creating a function that will be used to get the indices of words from vocab
def tokenizer(text, vocab):
    """Converts text to a list of indices using a vocabulary dictionary"""
    return [vocab[token] for token in str(text).split()]
```

```
def collate_batch(batch, my_vocab):
    """
    Prepares a batch of data by transforming texts into indices based on a vocabulary and
    converting labels into a tensor.

    Args:
        batch (list of dict): A batch of data where each element is a dictionary with keys
                              'labels' and 'texts'. 'labels' are the sentiment labels, and
                              'texts' are the corresponding texts.
        my_vocab (torchtext.vocab.Vocab): A vocabulary object that maps tokens to indices.

    Returns:
        dict: A dictionary with three keys:
              - 'input_ids': a tensor containing concatenated indices of the texts.
              - 'offsets': a tensor representing the starting index of each text in 'input_i
              - 'labels': a tensor of the labels for each text in the batch.

    The function transforms each text into a list of indices based on the provided vocabular
    It also converts the labels into a tensor. The 'offsets' are computed to keep track of t
    start of each text within the 'input_ids' tensor, which is a flattened representation of
    """

    # Get labels and texts from batch dict samples
    labels = [sample['labels'] for sample in batch]
    texts = [sample['texts'] for sample in batch]

    # Convert the list of labels into a tensor of dtype int32
    labels = torch.tensor(labels, dtype=torch.float32)

    # Convert the list of texts into a list of lists; each inner list contains the vocabular
    list_of_list_of_indices = [tokenizer(text, my_vocab) for text in texts]

    # Concatenate all text indices into a single tensor
    input_ids = torch.cat([torch.tensor(i, dtype=torch.int64) for i in list_of_list_of_indic

    # Compute the offsets for each text in the concatenated tensor
    offsets = [0] + [len(i) for i in list_of_list_of_indices]
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)

    return {
        'input_ids': input_ids,
        'offsets': offsets,
        'labels': labels
    }


emo_vocab = get_vocab(trainset, min_freq=2)
collate_fn = partial(collate_batch, my_vocab=emo_vocab)
```

```
my_config = CustomConfig(vocab_size=len(emo_vocab),
                         embedding_dim=300,
                         hidden_dim1=200,
                         hidden_dim2=100,
                         num_labels=11)


my_config.id2label = {0:'anger', 1: 'anticipation', 2:'disgust',3:'fear',4:'joy',5:'love',6:

# Generating id_to_label by reversing the key-value pairs in label_to_id
my_config.label2id = {v: k for k, v in my_config.id2label .items()}

my_config
```

```
CustomConfig {
  "embedding_dim": 300,
  "hidden_dim1": 200,
  "hidden_dim2": 100,
  "id2label": {
    "0": "anger",
    "1": "anticipation",
    "2": "disgust",
    "3": "fear",
    "4": "joy",
    "5": "love",
    "6": "optimism",
    "7": "pessimism",
    "8": "sadness",
    "9": "surprise",
    "10": "trust"
  },
  "label2id": {
    "anger": 0,
    "anticipation": 1,
    "disgust": 2,
    "fear": 3,
    "joy": 4,
    "love": 5,
    "optimism": 6,
    "pessimism": 7,
    "sadness": 8,
    "surprise": 9,
    "trust": 10
  },
  "transformers_version": "4.39.3",
  "vocab_size": 6062
}
```

```
model = CustomMLP(config=my_config)
model
```

```
CustomMLP(
  (embedding_bag): EmbeddingBag(6062, 300, mode='mean')
  (layers): Sequential(
```

```
        (0): Linear(in_features=300, out_features=200, bias=True)
        (1): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True,
    track_running_stats=True)
        (2): ReLU()
        (3): Dropout(p=0.5, inplace=False)
        (4): Linear(in_features=200, out_features=100, bias=True)
        (5): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True,
    track_running_stats=True)
        (6): ReLU()
        (7): Dropout(p=0.5, inplace=False)
        (8): Linear(in_features=100, out_features=11, bias=True)
      )
    )
```

```
def compute_metrics(eval_pred):
    combined_metrics = evaluate.combine([evaluate.load("accuracy"),
                                         evaluate.load("f1", average="macro")])

    logits, labels = eval_pred
    predictions = (logits>0.5).astype(int).reshape(-1)
    evaluations = combined_metrics.compute(
        predictions=predictions, references=labels.astype(int).reshape(-1))
    return evaluations
```

```
# Configure training parameters
training_args = TrainingArguments(

    # Training-specific configurations
    num_train_epochs=5,
    per_device_train_batch_size=128, # Number of samples per training batch
    per_device_eval_batch_size=128, # Number of samples per validation batch
    weight_decay=0.1, # weight decay (L2 regularization)
    learning_rate=0.001, # learning arte
    optim='adamw_torch', # optimizer
    remove_unused_columns=False, # flag to retain unused columns

    # Checkpoint saving and model evaluation settings
    output_dir=str(model_folder),  # Directory to save model checkpoints
    evaluation_strategy='steps',  # Evaluate model at specified step intervals
    ##################################
    eval_steps=10,  # Perform evaluation every 50 training steps
    save_strategy="steps",  # Save model checkpoint at specified step intervals
    save_steps=50,  # Save a model checkpoint every 50 training steps
    load_best_model_at_end=True,  # Reload the best model at the end of training
    save_total_limit=2,  # Retain only the best and the most recent model checkpoints
    # Use 'accuracy' as the metric to determine the best model
    metric_for_best_model="f1",
    greater_is_better=True,  # A model is 'better' if its accuracy is higher


    # Experiment logging configurations
    logging_strategy='steps',
    logging_steps=10,
    report_to='wandb',  # Log metrics and results to Weights & Biases platform
    run_name='imdb_hf_trainer',  # Experiment name for Weights & Biases
)


trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=trainset,
    eval_dataset = validset,
    data_collator=collate_fn,
    compute_metrics=compute_metrics,
)
print(trainer)
```

```
<transformers.trainer.Trainer object at 0x7a64fb2606d0>
/usr/local/lib/python3.10/dist-packages/accelerate/accelerator.py:436: FutureWarning: Pa
dataloader_config = DataLoaderConfiguration(dispatch_batches=None, split_batches=False,
  warnings.warn(
```

```
!wandb login 75a22b5a5c4de4706fb1be6e842e13687283d10c
# specify the project name where the experiment will be logged
%env project_WANDB = nlp_course_spring_2024-HW5
```

```
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
env: project_WANDB=nlp_course_spring_2024-HW5
```

```
trainer.train()
```

wandb: Currently logged in as: samanojvan (manojcompany). Use `wandb login --relogin` t(
Tracking run with wandb version 0.16.6
Run data is saved locally in /content/wandb/run-20240406_033511-muc3ld1o
Syncing run **imdb_hf_trainer** to Weights & Biases (docs)
View project at https://wandb.ai/manojcompany/huggingface
View run at https://wandb.ai/manojcompany/huggingface/runs/muc3ld1o
[185/185 00:47, Epoch 5/5]

Step	Training Loss	Validation Loss	Accuracy	F1
10	0.681200	0.657856	0.786849	0.000000
20	0.604700	0.612528	0.786849	0.000000
30	0.556800	0.576016	0.786849	0.000000
40	0.521600	0.548343	0.786849	0.000000
50	0.508800	0.528223	0.786849	0.000000
60	0.497600	0.509645	0.786849	0.000000
70	0.490500	0.502097	0.786878	0.000276
80	0.477000	0.495359	0.786878	0.000276
90	0.483500	0.493399	0.787290	0.004132
100	0.473900	0.488392	0.787320	0.004407
110	0.475000	0.485233	0.787320	0.004407
120	0.472700	0.483286	0.787290	0.004132
130	0.462800	0.482534	0.787320	0.004407
140	0.468300	0.480037	0.787349	0.004682
150	0.463800	0.477517	0.787408	0.005231
160	0.463800	0.477328	0.787408	0.005231
170	0.468300	0.477003	0.787437	0.005506
180	0.467000	0.476472	0.787408	0.005231

Downloading builder script: 100% 4.20k/4.20k [00:00<00:00, 63.7kB/s]

Downloading builder script: 100% 6.77k/6.77k [00:00<00:00, 177kB/s]

```
TrainOutput(global_step=185, training_loss=0.5011747398891965, metrics=
```

```
trainer.evaluate()
```

[25/25 00:00]

```
{'eval_loss': 0.47751718759536743,
 'eval_accuracy': 0.7874080611944689,
 'eval_f1': 0.005231277533039648,
 'eval_runtime': 3.1655,
 'eval_samples_per_second': 976.164,
 'eval_steps_per_second': 7.898,
 'epoch': 5.0}
```

```
valid_output = trainer.predict(validset)
```

```
valid_output._fields
```

```
('predictions', 'label_ids', 'metrics')
```

```
# After training, let us check the best checkpoint
# We need this for Inference
best_model_checkpoint_step = trainer.state.best_model_checkpoint.split('-')[-1]
print(f"The best model was saved at step {best_model_checkpoint_step}.")
```

```
The best model was saved at step 150.
```

```
wandb.finish()
```

Run history:

```
# Define the path to the best model checkpoint
# 'model checkpoint' variable is constructed using the model folder path and the checkpoint
```