Praful_Patil_HW5

April 4, 2024

0.0.1 Setting up the environment

[ ]:
```
# If in Colab, then import the drive module from google.colab
if 'google.colab' in str(get_ipython()):
  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/'
else:
  basepath = '/home/harpreet/Insync/google_drive_shaannorr/data'
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[ ]:
```
# 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 tallying
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 manipulations
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('/content/drive/MyDrive/NLP/HW_5')
data_folder = base_folder
model_folder = base_folder/'Models'
custom_functions = base_folder/'custom_functions'
```

[ ]:
```
model_folder.mkdir(exist_ok=True, parents = True)
```

[ ]:
```
model_folder
```

[ ]:
```
PosixPath('/content/drive/MyDrive/NLP/HW_5/Models')
```

0.0.2 Load Data

[ ]:
```
import pandas as pd
train_df = pd.read_csv(data_folder / 'train.csv')
test_df = pd.read_csv(data_folder / 'test.csv')
sample_submission_df = pd.read_csv(data_folder / 'sample_submission.csv')
```

0.0.3 Splitting and converting label columns into array

[456]:
```
# Define features and labels
X = train_df['Tweet']
y = train_df[['anger', 'anticipation', 'disgust', 'fear', 'joy', 'love',␣
 ↪'optimism', 'pessimism', 'sadness', 'surprise', 'trust']].to_numpy()

# performing the split
from sklearn.model_selection import train_test_split
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

X_train = X_train.tolist()
X_valid = X_valid.tolist()

# Check the lengths
print(len(X_train), len(y_train))
```

```
6179 6179
```

[457]:
```
trainset = Dataset.from_dict({
    'texts': X_train,
    'labels': y_train
})

validset = Dataset.from_dict({
    'texts': X_valid,
    'labels': y_valid
})
```

[458]:
```
trainset[:5]['texts']
```

[458]:
```
["Going to get myself a copy of @StephenKing's CUJO for an upcoming project that
I can't talk about just yet.  #amwriting",
 "@carysmithwriter @Maria_Savva @RealRockAndRoll We're the least known band in
the World, but so glad you asked #muchlove ",
 'Unruly kids at 8am in the morning #nothanks ripping the flower beds up by the
roots while their parents watch #shocking',
 "Ok but I just got called a 'White Devil' on the train and I didnt know whether
to laugh or be offended",
 '@SXMUrbanView @karenhunter @CousinSyl you are so wrong for this!needed levity
after that recording']
```

[459]:
```
trainset.features
```

[459]:
```
{'texts': Value(dtype='string', id=None),
 'labels': Sequence(feature=Value(dtype='int64', id=None), length=-1, id=None)}
```

[460]:
```
trainset[0]['labels']
```

[460]:
```
[0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0]
```

[461]:
```
labels_type = type(trainset[0]['texts'])
print(labels_type)
```

```
<class 'str'>
```

0.0.4 Create Custom Model and Model Config Class

[462]:
```
class CustomConfig(PretrainedConfig):
  def __init__(self, vocab_size=0, embedding_dim=0, hidden_dim1=0,␣
 ↪hidden_dim2=0, num_labels=11, **kwargs):
      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
```

[463]:
```
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()
            labels = labels.float()
            loss = loss_fct(logits, labels)

        return SequenceClassifierOutput(
            loss=loss,
            logits=logits
        )
```

[470]:
```
from collections import Counter
from torchtext.vocab import vocab

def get_vocab(dataset, min_freq=1):

    counter = Counter()
    for text in dataset['texts']:
        counter.update(str(text).split())
    my_vocab = vocab(counter, min_freq=min_freq)
    my_vocab.insert_token('<unk>', 0)
    my_vocab.set_default_index(0)
    return my_vocab
```

[471]:
```
# 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()]
```

0.0.5 Collate Function for Train and Test

[472]:
```
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_ids'.
            - '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 vocabulary.
    It also converts the labels into a tensor. The 'offsets' are computed to␣
↪keep track of the
    start of each text within the 'input_ids' tensor, which is a flattened␣
↪representation of all text indices.
    """

    # 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.float64)

    # Convert the list of texts into a list of lists; each inner list contains␣
↪the vocabulary indices for a text
    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.int32) for i in␣
↪list_of_list_of_indices])

    # 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
    }
```

[473]:
```
def collate_batch_test(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_ids'.
              - '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 vocabulary.
    It also converts the labels into a tensor. The 'offsets' are computed to␣
↪keep track of the
    start of each text within the 'input_ids' tensor, which is a flattened␣
↪representation of all text indices.
    """

    # No need of labels as we would be predicting them
    #labels = [sample['labels'] for sample in batch]
    texts = [sample['texts'] for sample in batch]

    # No need of labels as we would be predicting them
    #labels = torch.tensor(labels, dtype=torch.float64)

    # Convert the list of texts into a list of lists; each inner list contains␣
↪the vocabulary indices for a text
    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.int32) for i in␣
↪list_of_list_of_indices])

    # 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
    }
```

[474]:
```
vocab = get_vocab(trainset, min_freq = 2)
collate_fn = partial(collate_batch, my_vocab=vocab)
print(len(vocab))
```

```
8134
```

[475]:
```
my_config = CustomConfig(vocab_size=len(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: 'optimism', 7: 'pessimism', 8: 'sadness' , 9:␣
 ↪'surprise', 10: 'trust'}
```

[ ]:
```
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": 8134
}
```

[ ]:
```
model = CustomMLP(config=my_config)
```

[ ]:
```
model
```

[ ]:
```
CustomMLP(
  (embedding_bag): EmbeddingBag(8134, 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)
  )
)
```

0.0.6 Compute Metrics for train and test

[ ]:
```
import numpy as np
from sklearn.metrics import accuracy_score, f1_score, precision_score,␣
 ↪recall_score, hamming_loss

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    # Apply sigmoid to the logits to get probabilities
    probabilities = 1 / (1 + np.exp(-logits))
    # Choosing appropriate threshold and converting into predictions
    predictions = (probabilities >= 0.46).astype(int)
    #print(np.average(probabilities))

    # Metrics calculation
    accuracy = accuracy_score(labels.flatten(), predictions.flatten())
    f1 = f1_score(labels, predictions, average='macro', zero_division=0)
    precision = precision_score(labels, predictions, average='macro',␣
 ↪zero_division=0)
```

```
    recall = recall_score(labels, predictions, average='macro', zero_division=0)
    hamming = hamming_loss(labels, predictions)

    return {
        'accuracy': accuracy,
        'f1': f1,
        'precision': precision,
        'recall': recall,
        'hamming_loss': hamming
    }
```

[476]:
```
import numpy as np
from sklearn.metrics import accuracy_score, f1_score, precision_score,␣
 ↪recall_score, hamming_loss

def compute_metrics_test(eval_pred):
    logits, labels = eval_pred
    # Apply sigmoid to the logits to get probabilities
    probabilities = 1 / (1 + np.exp(-logits))
    # Choosing appropriate threshold and converting into predictions
    predictions = (probabilities >= 0.46).astype(int)
    #print(np.average(probabilities))

    # Metrics calculation
    accuracy = accuracy_score(labels.flatten(), predictions.flatten())
    f1 = f1_score(labels, predictions, average='macro', zero_division=0)
    precision = precision_score(labels, predictions, average='macro',␣
 ↪zero_division=0)
    recall = recall_score(labels, predictions, average='macro', zero_division=0)
    hamming = hamming_loss(labels, predictions)

    return {
        'accuracy': accuracy,
        'f1': f1,
        'precision': precision,
        'recall': recall,
        'hamming_loss': hamming,
#Returining Predictions from here to create the datafarme later
        'predictions': predictions
    }
```

0.0.7 Training Arguments

[ ]:
```
#Configure training parameters
training_args = TrainingArguments(
    # Training-specific configurations
```

```
    num_train_epochs=10,
    per_device_train_batch_size=64,
    per_device_eval_batch_size=64,
    weight_decay=0.01,
    learning_rate=0.0005,
    optim='adamw_torch',

    # It's generally safe and clean to let the Trainer remove unused columns,
    # unless you have a specific need to access them during training.
    remove_unused_columns= False,  # Consider setting this to True

    # Checkpoint saving and model evaluation settings
    output_dir=str(model_folder),
    evaluation_strategy='steps',
    eval_steps=100,
    save_strategy="steps",
    save_steps=100,
    load_best_model_at_end=True,
    save_total_limit=3,

    # For multi-label classification, consider using 'f1' or another relevant␣
 ↪metric
    metric_for_best_model="f1",
    greater_is_better=True,  # Ensure this aligns with your chosen metric

    # Experiment logging configurations
    logging_strategy='steps',
    logging_steps=50,
    report_to='wandb',
    run_name='hf_trainer',
    warmup_steps=500,
    lr_scheduler_type='cosine_with_restarts',
)
```

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

```
/usr/local/lib/python3.10/dist-packages/accelerate/accelerator.py:432:
FutureWarning: Passing the following arguments to `Accelerator` is deprecated
and will be removed in version 1.0 of Accelerate: dict_keys(['dispatch_batches',
'split_batches', 'even_batches', 'use_seedable_sampler']). Please pass an
```

```
`accelerate.DataLoaderConfiguration` instead:
dataloader_config = DataLoaderConfiguration(dispatch_batches=None,
split_batches=False, even_batches=True, use_seedable_sampler=True)
  warnings.warn(
```

[ ]:
```
!wandb login
```

```
wandb: Currently logged in as: prafulp659
(utd659). Use `wandb login --relogin` to force relogin
```

[ ]:
```
# specify the project name where the experiment will be logged
%env WANDB_PROJECT = HW5 Multilabel Classification Using HuggingFace Trainer
```

```
env: WANDB_PROJECT=HW5 Multilabel Classification Using HuggingFace Trainer
```

[ ]:
```
#!pip install --upgrade accelerate transformers
trainer.train()
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
TrainOutput(global_step=970, training_loss=0.5093251710085525,
metrics={'train_runtime': 55.5755, 'train_samples_per_second': 1111.821,
'train_steps_per_second': 17.454, 'total_flos': 14953028801940.0, 'train_loss':
0.5093251710085525, 'epoch': 10.0})
```

[ ]:
```
trainer.evaluate()
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
{'eval_loss': 0.6595310568809509,
 'eval_accuracy': 0.4807884671962342,
 'eval_f1': 0.30421722790648825,
 'eval_precision': 0.21847237140127698,
 'eval_recall': 0.5954350993373616,
 'eval_hamming_loss': 0.5192115328037659,
 'eval_runtime': 0.3275,
 'eval_samples_per_second': 4717.875,
 'eval_steps_per_second': 76.341,
 'epoch': 10.0}
```

[ ]:
```
valid_output = trainer.predict(validset)
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
# 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 100.
```

[ ]:
```
valid_preds = (valid_output.predictions >= 0).astype(float)
valid_labels = np.array(valid_output.label_ids)
```

[ ]:
```
wandb.finish()
```

```
VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),␣
 ↪FloatProgress(value=1.0, max=1.0)))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

Load Model from checkpoint

[ ]:
```
# Define the path to the best model checkpoint
# 'model_checkpoint' variable is constructed using the model folder path and␣
 ↪the checkpoint step
# This step is identified as having the best model performance during training
model_checkpoint = model_folder/f'checkpoint-{best_model_checkpoint_step}'
```

[ ]:
```
# Instantiate the CustomMLP model with predefined configurations
# 'my_config' is an instance of the CustomConfig class, containing specific␣
 ↪model settings like
# vocabulary size, embedding dimensions, etc.
model = CustomMLP(my_config)
```

[ ]:
```
model = model.from_pretrained(model_checkpoint, config = my_config)
```

[ ]:
```
model
```

[ ]:
```
CustomMLP(
  (embedding_bag): EmbeddingBag(8134, 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)
  )
```

)

Instantiate Trainer for calculating predictions for test set

[430]:
```
# Create a partial function 'collate_fn' using 'collate_batch' with 'my_vocab' set to 'imdb_vocab'
# This function will be used by the Trainer to process batches of data during evaluation
collate_fn = partial(collate_batch, my_vocab=vocab)

# Configure training arguments for model evaluation
# 'output_dir' specifies where to save the results
# 'per_device_eval_batch_size' sets the batch size for evaluation, adjusted based on available GPU memory
# 'do_train = False' and 'do_eval=True' indicate that training is not performed, but evaluation is
# 'remove_unused_columns=False' ensures that all columns in the dataset are retained during evaluation
# 'report_to=[]' disables logging to external services like Weights & Biases

training_args = TrainingArguments(
    output_dir="./results",
    per_device_eval_batch_size=16,
    do_train=False,
    do_eval=False,
    remove_unused_columns=False,
    report_to=[]
)
```

0.0.8 Modifing testset to match trainset type

[436]:
```
# 1. Define features for test data
X_test = test_df['Tweet'].tolist()

# 2. Ensure label columns are similar to training data
label_columns = ['anger', 'anticipation', 'disgust', 'fear', 'joy', 'love', 'optimism', 'pessimism', 'sadness', 'surprise', 'trust']
for i in range(0,len(test_df)):
  for j in label_columns:
    if(test_df[j][i]=='NONE'):
      test_df[j][i]=0
test_labels = test_df[label_columns].to_numpy(dtype=np.int64)
# 3. Create Dataset object for test data
testset = Dataset.from_dict({
    'texts': X_test,
    'labels': test_labels
})
```

```
# 4. Verify the test set
print(testset[:5])
print(testset.features)

# Verify the type of features and labels
print(type(testset[0]['texts']))
print(type(testset[0]['labels']))
```

```
{'texts': ['@Adnan__786__ @AsYouNotWish Dont worry Indian army is on its ways to
dispatch all Terrorists to Hell', 'Academy of Sciences, eschews the normally
sober tone of scientific papers and calls the massive loss of wildlife a
"biological annihilation', 'I blew that opportunity -__- #mad', 'This time in 2
weeks I will be 30… ', '#Deppression is real. Partners w/ #depressed people
truly dont understand the depth in which they affect us. Add in #anxiety
&amp;amp;makes it worse'], 'labels': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]}
{'texts': Value(dtype='string', id=None), 'labels':
Sequence(feature=Value(dtype='int64', id=None), length=-1, id=None)}
<class 'str'>
<class 'list'>
```

[437]:
```
print(trainset[:5])
print(trainset.features)
```

```
{'texts': ["Going to get myself a copy of @StephenKing's CUJO for an upcoming
project that I can't talk about just yet.  #amwriting", "@carysmithwriter
@Maria_Savva @RealRockAndRoll We're the least known band in the World, but so
glad you asked #muchlove ", 'Unruly kids at 8am in the morning #nothanks ripping
the flower beds up by the roots while their parents watch #shocking', "Ok but I
just got called a 'White Devil' on the train and I didnt know whether to laugh
or be offended", '@SXMUrbanView @karenhunter @CousinSyl you are so wrong for
this!needed levity after that recording'], 'labels': [[0, 1, 0, 0, 1, 0, 1, 0,
0, 0, 0], [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0], [1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0],
[0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0], [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]]}
{'texts': Value(dtype='string', id=None), 'labels':
Sequence(feature=Value(dtype='int64', id=None), length=-1, id=None)}
```

[438]:
```
# Initialize the Trainer with the specified model and training arguments
# 'model' is the CustomMLP model loaded with pre-trained weights
# 'training_args' contains the configurations for evaluation, including batch␣
 ↪sizes and output directory
# 'eval_dataset' is set to 'testset', which is the dataset used for evaluating␣
 ↪the model
# 'data_collator' is assigned 'collate_fn', the function for processing batches␣
 ↪of data
```

```
# 'compute_metrics' is a function that calculates evaluation metrics like␣
 ↪accuracy and F1 score

trainer = Trainer(
    model=model,
    args=training_args,
    eval_dataset=testset,
    data_collator=collate_fn,
    compute_metrics=compute_metrics_test,
)
```

```
/usr/local/lib/python3.10/dist-packages/accelerate/accelerator.py:432:
FutureWarning: Passing the following arguments to `Accelerator` is deprecated
and will be removed in version 1.0 of Accelerate: dict_keys(['dispatch_batches',
'split_batches', 'even_batches', 'use_seedable_sampler']). Please pass an
`accelerate.DataLoaderConfiguration` instead:
dataloader_config = DataLoaderConfiguration(dispatch_batches=None,
split_batches=False, even_batches=True, use_seedable_sampler=True)
  warnings.warn(
```

[439]:
```
tst = trainer.evaluate()
tst['eval_predictions']
```

```
<IPython.core.display.HTML object>
```

[439]:
```
array([[0, 1, 1, …, 1, 1, 0],
       [0, 0, 1, …, 0, 0, 0],
       [1, 1, 1, …, 1, 0, 1],
       …,
       [1, 0, 0, …, 0, 0, 0],
       [0, 0, 1, …, 0, 0, 1],
       [1, 1, 1, …, 1, 1, 1]])
```

[440]:
```
test_predictions=trainer.predict(testset)
```

```
<IPython.core.display.HTML object>
```

0.0.9 Creating Submission file

[441]:
```
final_df = pd.DataFrame(tst['eval_predictions'], columns=label_columns)
final_df
```

[441]:

	anger	anticipation	disgust	fear	joy	love	optimism	pessimism	\
0	0	1	1	1	0	0	1	0	
1	0	0	1	1	1	0	1	0	
2	1	1	1	0	1	1	1	1	
3	1	0	1	1	0	0	1	1	

```
4          1              0         1     1     1     1          0           1
…      …            …       …    …   …    …        …          …
3254       0              0         1     1     1     0          1           0
3255       0              0         1     1     1     0          1           0
3256       1              0         0     1     1     0          1           1
3257       0              0         1     1     1     0          1           1
3258       1              1         1     1     1     0          1           0

      sadness  surprise  trust
0           1         1      0
1           0         0      0
2           1         0      1
3           0         1      0
4           1         1      1
…         …         …     …
3254        0         0      0
3255        0         0      0
3256        0         0      0
3257        0         0      1
3258        1         1      1

[3259 rows x 11 columns]
```

[ ]:
```
final_df['ID']=test_df['ID']
cols = ['ID']+label_columns
fnl_df = final_df[cols]
fnl_df
```

[ ]:
```
              ID  anger  anticipation  disgust  fear  joy  love  optimism  \
0     2018-01559      0             1        1     1    0     0         1
1     2018-03739      0             0        1     1    1     0         1
2     2018-00385      1             1        1     0    1     1         1
3     2018-03001      1             0        1     1    0     0         1
4     2018-01988      1             0        1     1    1     1         0
…          …    …            …       …   …  …   …        …
3254  2018-03848      0             0        1     1    1     0         1
3255  2018-00416      0             0        1     1    1     0         1
3256  2018-03717      1             0        0     1    1     0         1
3257  2018-03504      0             0        1     1    1     0         1
3258  2018-00115      1             1        1     1    1     0         1

      pessimism  sadness  surprise  trust
0             0        1         1      0
1             0        0         0      0
2             1        1         0      1
3             1        0         1      0
4             1        1         1      1
```

```
…          …       …         …     …
3254          0         0         0       0
3255          0         0         0       0
3256          1         0         0       0
3257          1         0         0       1
3258          0         1         1       1

[3259 rows x 12 columns]
```

[ ]:
```
fnl_df.to_csv('output.csv', index=False)
```