Kaggle private competition PDF report detailing each step and approach

Student infomation
Name:林澤宇
Student id:112064528
Github id:98965138
kaggle name:NTHU112064528

42	▲3	NTHU112064528	0.46940	2	2d

CELL1

```
import pandas as pd
import json
import re
import emoji
from nltk.corpus import stopwords
from transformers import BertTokenizer, BertModel
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.optim import AdamW
from torchmetrics import Accuracy
from tqdm import tqdm
import nltk
import numpy as np
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Download NLTK resources for text preprocessing
nltk.download("punkt")
nltk.download("stopwords")

# Step 1: Load emotion and data identification datasets
# Read the CSV file containing emotion labels
emotion = pd.read_csv("/kaggle/input/dm-2024-isa-5810-lab-2-homework/emotion.csv")
data_identification = pd.read_csv("/kaggle/input/dm-2024-isa-5810-lab-2-homework/data_identification.csv")

# Load and process tweets from the JSON file
with open("/kaggle/input/dm-2024-isa-5810-lab-2-homework/tweets_DM.json", "r") as file:
    tweets_raw = [json.loads(line) for line in file]

tweets = pd.DataFrame([{
    "tweet_id": tweet["_source"]["tweet"]["tweet_id"],
    "text": tweet["_source"]["tweet"]["text"]
} for tweet in tweets_raw])
```

Explain:Load the necessary Python packages for text processing, model building, data processing and other functions.
Set up the computing device (GPU or CPU).

CELL2

```python
# Merge `data_identification` with `tweets` on the common column "tweet_id"
merged_data = pd.merge(data_identification, tweets, on="tweet_id", how="left")
#Merge the result with `emotion` on "tweet_id"
merged_data = pd.merge(merged_data, emotion, on="tweet_id", how="left")

# 2.  Separate the merged dataset into training and testing sets
train_data = merged_data[merged_data["identification"] == "train"].copy()
test_data = merged_data[merged_data["identification"] == "test"].copy()
```

Explain:Load tweet data and label data and integrate them

Read the emotion label (emotion.csv) and data identification label (data_identification.csv).

Extract tweet content from a JSON file (tweets_DM.json).

Merge three data sets to form a complete data frame (merged_data)

CELL3

```python
# Step 3: Text preprocessing
stop_words = set(stopwords.words("english"))

#Steps:
#1. Convert emojis to descriptive words using `emoji.demojize`.
#2. Remove URLs (e.g., "http://example.com").
#3. Remove user mentions (e.g., "@username").
#4. Retain hashtags but remove other punctuation and special characters.
#5. Convert text to lowercase to ensure consistency.
#6. Remove stopwords to focus on meaningful words.

def preprocess_text(text):
    text = emoji.demojize(text)
    text = re.sub(r"http\S+", "", text)
    text = re.sub(r"@\w+", "", text)
    text = re.sub(r"[^\w\s#\U0001F600-\U0001F64F]", "", text)
    text = text.lower()
    words = [word for word in text.split() if word not in stop_words]
    return " ".join(words)  # required for BERT input

train_data["cleaned_text"] = train_data["text"].apply(preprocess_text)
test_data["cleaned_text"] = test_data["text"].apply(preprocess_text)
```

#Steps:

#1. Convert emojis to descriptive words using `emoji.demojize`.

#2. Remove URLs (e.g., "http://example.com").

#3. Remove user mentions (e.g., "@username").

#4. Retain hashtags but remove other punctuation and special characters.

#5. Convert text to lowercase to ensure consistency.

#6. Remove stopwords to focus on meaningful words.

CELL4,5

```
#Step 4: Label processing
emotion_mapping = {emotion: idx for idx, emotion in enumerate(train_data["emotion"].unique())}
train_data["label"] = train_data["emotion"].map(emotion_mapping)
test_data["label"] = test_data["emotion"].map(emotion_mapping)
```

```
from sklearn.model_selection import train_test_split

# train_data20% for val
train_data, val_data = train_test_split(train_data, test_size=0.2, random_state=42)

print(f"Train size: {len(train_data)}, Validation size: {len(val_data)}")
```

Explain: This code converts emotion labels into numeric values for training and testing datasets using a mapping. Then, it splits the training data into 80% for training and 20% for validation, ensuring reproducibility with a fixed random state. Finally, it prints the sizes of the training and validation sets.

CELL6

```python
# 5. Define datasets and load data
class TwitterDataset(Dataset):
    #-data (DataFrame): The input data containing 'cleaned_text' and 'label' columns.
    #-tokenizer: The Hugging Face tokenizer for tokenizing text.
    #-max_length (int): The maximum sequence length for padding and truncation.
    def __init__(self, data, tokenizer, max_length):
        self.data = data
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __getitem__(self, index):
        row = self.data.iloc[index]
        text = row['cleaned_text'] # Preprocessed text
        label = row['label']# Numeric label for the emotion
        # Tokenize the text and prepare it for the model
        encoding = self.tokenizer(
            text,
            padding="max_length",# Pad to the maximum sequence length
            truncation=True, # Truncate text longer than the maximum length
            max_length=self.max_length,# Maximum sequence length
            return_tensors="pt"# Return PyTorch tensors
        )

        return {
            'input_ids': encoding['input_ids'].squeeze(0),# Tokenized IDs
            'attention_mask': encoding['attention_mask'].squeeze(0),# Attention mask
            'label': torch.tensor(label, dtype=torch.long) # Emotion label as a tensor
        }

    def __len__(self):
        return len(self.data)

def collate_fn(batch):
    #- 'input_ids' (Tensor): Stacked input IDs for the batch.
    #- 'attention_mask' (Tensor): Stacked attention masks for the batch.
    #- 'labels' (Tensor): Stacked labels for the batch.
    input_ids = torch.stack([item['input_ids'] for item in batch])
    attention_mask = torch.stack([item['attention_mask'] for item in batch])
    labels = torch.tensor([item['label'] for item in batch])
    return {'input_ids': input_ids, 'attention_mask': attention_mask, 'labels': labels}
```

Explain: This code defines the TwitterDataset class and a collate_fn function to handle data preparation for the model:

TwitterDataset Class:

Custom PyTorch dataset for preprocessed tweets (cleaned_text) and their numeric emotion labels (label).

Text is tokenized using the Hugging Face tokenizer with padding, truncation, and conversion to PyTorch tensors.

Provides __getitem__ for accessing individual samples and __len__ for the dataset size.

collate_fn Function:

Used to batch multiple samples during data loading.

Stacks input_ids and attention_mask tensors, and collects labels into a single batch-ready dictionary.

This setup ensures that the data is properly tokenized and batched for input to the model.

CELL7:

```
# Step 6: Initialize BERT model and classifier
class EmotionClassifier(nn.Module):
    def __init__(self, bert_model_name="bert-base-uncased", num_classes=8):
        #Initialize
        super(EmotionClassifier, self).__init__()
        self.bert = BertModel.from_pretrained(bert_model_name)
        self.classifier = nn.Linear(self.bert.config.hidden_size, num_classes)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(
            input_ids=input_ids,
            attention_mask=attention_mask
        )
        # Extract the [CLS] token output (first token) from the last hidden state
        cls_output = outputs.last_hidden_state[:, 0, :]
        # Pass the [CLS] token output through the classifier to get logits
        logits = self.classifier(cls_output)
        return logits
```

Explain: This code defines the EmotionClassifier, a PyTorch model that combines a pre-trained BERT model (bert-base-uncased) with a linear classification layer. The BERT model processes the input text, and the [CLS] token's output is passed to the classifier to predict emotion labels.

CELL8

```
# Step 7: Initialize model, optimizer, and loss function
model = EmotionClassifier(num_classes=len(emotion_mapping)).to(device)
# Load the pre-trained BERT tokenizer for text preprocessing
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
optimizer = AdamW(model.parameters(), lr=2e-5)
loss_fn = nn.CrossEntropyLoss()

# Step 8: Construct data loaders
max_length = 64
batch_size = 8
# Create DataLoader objects for batch processing
train_dataset = TwitterDataset(train_data, tokenizer, max_length)
#test_dataset = TwitterDataset(test_data, tokenizer, max_length)
val_dataset = TwitterDataset(val_data, tokenizer, max_length)

train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, collate_fn=collate_fn)
#test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, collate_fn=collate_fn)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, collate_fn=collate_fn)
# - `train_dataloader` is ready to provide batches for training.
# - `val_dataloader` is used for validating the model during training.
```

Explain: This code initializes the EmotionClassifier model with the number of emotion classes and prepares it for training on the selected device. It also sets up the pre-trained BERT tokenizer, AdamW optimizer, and cross-entropy loss function. The training and validation datasets are wrapped into DataLoader objects, which handle batch processing and data shuffling during training and validation.

CELL9

```
# Step 9: Training and evaluation functions
#Function to train the model for one epoch.
def train(model, dataloader, optimizer, loss_fn, device):
    model.train()
    total_loss = 0
    for batch in tqdm(dataloader, desc="Training"):
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        # Zero out the gradients for the optimizer
        optimizer.zero_grad()
        # Forward pass through the model
        logits = model(input_ids, attention_mask)
        # Compute the loss between predictions and true labels
        loss = loss_fn(logits, labels)
        # Backward pass to compute gradients
        loss.backward()
        # Update model parameters using the optimizer
        optimizer.step()
        # Accumulate the loss for reporting
        total_loss += loss.item()
    return total_loss / len(dataloader)
```

```
def evaluate(model, dataloader, loss_fn, device):
    model.eval()
    total_loss = 0
    # Initialize
    accuracy_metric = Accuracy(task="multiclass", num_classes=len(emotion_mapping)).to(device)
    with torch.no_grad():
        for batch in tqdm(dataloader, desc="Evaluating"):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)
            # Forward pass through the model
            logits = model(input_ids, attention_mask)
            # Compute the loss between predictions and true labels
            loss = loss_fn(logits, labels)
            # Accumulate the loss for reporting
            total_loss += loss.item()
            # Update the accuracy metric with the current batch predictions and labels
            accuracy_metric.update(logits.argmax(dim=1), labels)
    accuracy = accuracy_metric.compute().item()
    accuracy_metric.reset()
    # Return the average loss per batch and the overall accuracy
    return total_loss / len(dataloader), accuracy
```

Explain: This code defines two functions for training and evaluating the model:

train Function:

Trains the model for one epoch.

Performs forward and backward passes, calculates loss, and updates model parameters using the optimizer.

Returns the average training loss.

evaluate Function:

Evaluates the model on a validation or test dataset without updating weights.

Calculates loss and computes accuracy using the torchmetrics.Accuracy metric.

Returns the average validation/test loss and accuracy.

These functions handle the core logic for training and validating the model.

CELL10

```python
epochs = 1 #I only train for one epochs because of time. one epochs 5hr
for epoch in range(epochs):
    print(f"Epoch {epoch + 1}/{epochs}")

    # train model
    train_loss = train(model, train_dataloader, optimizer, loss_fn, device)

    # val model
    val_loss, val_accuracy = evaluate(model, val_dataloader, loss_fn, device)
    # Print the training loss for this epoch
    print(f"Train Loss: {train_loss:.4f}")
    # Print the validation loss and accuracy for this epoch
    print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_accuracy:.4f}")

print("Training Complete!")
```

Explain: This code conducts model training and evaluation for one epoch due to time constraints (each epoch takes approximately 5 hours).

Summary:

Epoch Loop:

Iterates through the specified number of epochs (in this case, 1).

Training:

Calls the train function to update the model parameters using the training dataset and computes the training loss.

Validation:

Calls the evaluate function to calculate the validation loss and accuracy on the validation dataset.

Results Logging:

Prints the training loss, validation loss, and validation accuracy after each epoch.

Completion Message:

Confirms when the training process is complete.

CELL11

```python
#Same as class TwitterDataset but return tweet_id
class TestTwitterDataset(Dataset):
    def __init__(self, data, tokenizer, max_length):
        self.data = data
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __getitem__(self, index):
        row = self.data.iloc[index]
        text = row['cleaned_text']
        tweet_id = row['tweet_id']

        encoding = self.tokenizer(
            text,
            padding="max_length",
            truncation=True,
            max_length=self.max_length,
            return_tensors="pt"
        )

        return {
            'input_ids': encoding['input_ids'].squeeze(0),
            'attention_mask': encoding['attention_mask'].squeeze(0),
            'id': tweet_id  # return tweet_id
        }

    def __len__(self):
        return len(self.data)
```

Explain: This code defines the TestTwitterDataset class, which is similar to the TwitterDataset class but includes the tweet ID (tweet_id) in the output.
Purpose:
Prepares test data for the model, including the tweet_id to associate predictions with specific tweets.
Key Features:
Accepts data, tokenizer, and max_length as inputs.
Processes each tweet by tokenizing the text and padding/truncating it to the specified maximum length.
Returns input_ids, attention_mask, and tweet_id for each data sample.
Difference from TwitterDataset:
Adds the tweet_id to the output, which is useful for tracking predictions for specific tweets in the test set.
This class is tailored for handling test datasets where the original tweet IDs need to be preserved for output or submission purposes.

CELL12

```
def predict(model, dataloader, device):
    model.eval()
    predictions = [] # List to store predicted class labels
    ids = []   # List to store tweet IDs from the test dataset
    with torch.no_grad():
        for batch in tqdm(dataloader, desc="Predicting"):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)

            # Forward pass through the model to get prediction logits
            logits = model(input_ids, attention_mask)
            preds = logits.argmax(dim=1).cpu().numpy()  #Get predicted class labels
            predictions.extend(preds)
            ids.extend(batch['id'])  #Append tweet IDs to the list

    return ids, predictions
```

Explain: This function, predict, is crucial for generating predictions on the test dataset using the trained model.

Purpose:

Makes predictions on the test dataset and associates them with their respective tweet IDs.

Key Features:

Evaluation Mode: The model is set to evaluation mode (model.eval()) to disable gradient computation.

No Gradient Calculation: Uses torch.no_grad() for efficient inference.

Process:

Iterates through the batches in the test dataset using the DataLoader.

Performs a forward pass through the model to obtain logits (raw predictions).

Converts the logits to predicted class labels (argmax on dimension 1).

Collects predictions and their corresponding tweet IDs into separate lists.

Output:

Returns:ids: A list of tweet IDs from the test dataset.

predictions: A list of predicted class labels corresponding to each tweet ID.

This function is essential for generating model predictions in a structured way, enabling submission or further evaluation of results.

CELL13

```python
model.load_state_dict(torch.load("bert_emotion_model.pth")) # Load model weights from a file
model.eval()    # Set the model to evaluation mode
print("Model loaded and ready for testing.")

# Use the model to make predictions on the test dataset
test_ids, test_predictions = predict(model, test_dataloader, device)

# Convert predicted numeric labels into emotion names
inverse_emotion_mapping = {v: k for k, v in emotion_mapping.items()}
test_emotions = [inverse_emotion_mapping[pred] for pred in test_predictions]
```

```python
# Create submission file
submission = pd.DataFrame({
    "id": test_ids,
    "emotion": test_emotions
})

# Save as CSV file
submission.to_csv("sampleSubmission.csv", index=False)
print("Submission file saved as sampleSubmission.csv")
```

Explain:This final step handles model loading, prediction generation, and submission file creation.

Loads the trained model's weights from the file bert_emotion_model.pth.

Sets the model to evaluation mode (model.eval()) to prepare for inference.

Calls the predict function to generate predictions for the test dataset.

Returns:

test_ids: Tweet IDs from the test dataset.

test_predictions: Numeric predictions for the emotion labels.and maps numeric predictions back to their corresponding emotion labels using the inverse_emotion_mapping.

Finally Creates a DataFrame with two columns:

id: The tweet IDs.

emotion: The predicted emotion labels.

Saves the DataFrame as a CSV file named sampleSubmission.csv.