DataMining Lab2 Homework Report

NTPU 711233125 林君騰

First, I used Torch to verify that CUDA was running correctly. Then, I read the 'tweet_DM.json' file line by line and merged the sentiment classifications, training set, and test set according to their IDs.

```
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

```
device(type='cuda')
```

I created a stop word function that includes converting text to lowercase, removing punctuation, removing stop words, and lemmatizing words, then applied this function.

```
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english')).union(set(ENGLISH_STOP_WORDS))

def preprocess_text(text):
    text = text.lower()
    text = re.sub(f"[{re.escape(string.punctuation)}]", "", text)
    text = ' '.join([word for word in text.split() if word not in stop_words])
    text = ' '.join([lemmatizer.lemmatize(word) for word in text.split()])
    return text

merged_df['text'] = merged_df['text'].apply(preprocess_text)
```

Next, I assigned data to the training and test sets based on the labels in the identification, encoded the target y using LabelEncoder, and split the training set into an 8:2 ratio for training and validation sets.

```
df_train = merged_df[merged_df['identification'] == 'train']
df_test = merged_df[merged_df['identification'] == 'test']

label_encoder = LabelEncoder()
df_train['emotion'] = label_encoder.fit_transform(df_train['emotion'].astype(str))

df_train, df_val = train_test_split(df_train, test_size=0.2, random_state=42)
```

I used RoBERTa as our tokenizer and converted the training, test, and validation sets into numerical sequences.

```
model_name = 'roberta-base'
tokenizer = RobertaTokenizer.from_pretrained(model_name)

train_encodings = tokenizer(df_train['text'].tolist(), max_length=128, truncation=True,
                            padding='max_length', return_tensors='pt')
val_encodings = tokenizer(df_val['text'].tolist(), max_length=128, truncation=True,
                          padding='max_length', return_tensors='pt')
test_encodings = tokenizer(df_test['text'].tolist(), max_length=128, truncation=True,
                           padding='max_length', return_tensors='pt')
```

I chose to use PyTorch to train this model. Firstly, I created a custom EmotionDataset as my Dataset, adding an attention_mask in the _getitem_ method and setting the dtype of labels to torch.long.

```
class EmotionDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

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

    def __getitem__(self, idx):
        input_ids = self.encodings['input_ids'][idx]
        attention_mask = self.encodings['attention_mask'][idx]
        labels = torch.tensor(self.labels[idx], dtype=torch.long)
        return input_ids, attention_mask, labels
```

I set the batch size of the DataLoader to 64. Then, I built a custom EmotionClassifier model, configured the RoBERTa parameters, and set up the pool output in the forward function.

```python
class EmotionClassifier(nn.Module):
    def __init__(self, model_name, num_labels):
        super(EmotionClassifier, self).__init__()
        self.roberta = RobertaModel.from_pretrained(model_name)
        self.classifier = nn.Linear(self.roberta.config.hidden_size, num_labels)

    def forward(self, input_ids, attention_mask):
        outputs = self.roberta(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs[1]
        logits = self.classifier(pooled_output)
        return logits
```

Before starting the training, I set the learning rate to 3e-5, used AdamW as our optimizer, CrossEntropy as our loss function, and Accuracy and F1-Score as our evaluation metrics.

```python
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

model = EmotionClassifier(model_name, num_labels).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-5)
criterion = nn.CrossEntropyLoss()

acc = Accuracy(task='multiclass', num_classes=num_labels).to(device)
f1 = F1Score(task='multiclass', num_classes=num_labels, average='macro').to(device)
```

```python
def calculate_accuracy(preds, labels):
    _, preds_max = torch.max(preds, 1)
    correct = (preds_max == labels).sum().item()
    accuracy = correct / labels.size(0)
    return accuracy

def calculate_f1(preds, labels):
    _, preds_max = torch.max(preds, 1)
    f1 = f1_score(labels.cpu().numpy(), preds_max.cpu().numpy(), average='weighted')
    return f1
```

Then I moved the entire device to CUDA, set the epochs to 3, and initialized accuracy to 0. I created a for loop with tqdm to display the training progress bar, set input, attention, and label, cleared gradients with zero_grad(), performed forward, calculated loss, backward, and updated parameters.

```
epochs = 3
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

best_val_acc = 0

for epoch in range(epochs):
    model.train()
    bar_train = tqdm(train_loader, desc=f'Training Epoch {epoch + 1}/{epochs}')
    for batch in bar_train:
        input_ids, attention_mask, labels = [x.to(device) for x in batch]

        optimizer.zero_grad()
        logits = model(input_ids, attention_mask)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
```

After that, I calculated the training set evaluation metrics, Accuracy, F1-Score, etc. Next, I entered the validation phase, initialized the validation set loss, Accuracy, F1-Score, etc., directly entered the validation set evaluation metrics, and added these calculated metrics into the previously created validation set list one by one.

```
model.eval()
bar_val = tqdm(val_loader, desc=f'Validation Epoch {epoch + 1}/{epochs}')
val_losses = []
val_accs = []
val_f1s = []

with torch.no_grad():
    for batch in bar_val:
        input_ids, attention_mask, labels = [x.to(device) for x in batch]
        logits = model(input_ids, attention_mask)
        loss = criterion(logits, labels)

        val_loss = loss.item()
        val_acc = acc(logits, labels)
        val_f1 = f1(logits, labels)

        val_losses.append(val_loss)
        val_accs.append(val_acc)
        val_f1s.append(val_f1)

        bar_val.set_postfix(loss=val_loss, acc=val_acc, f1=val_f1)
```

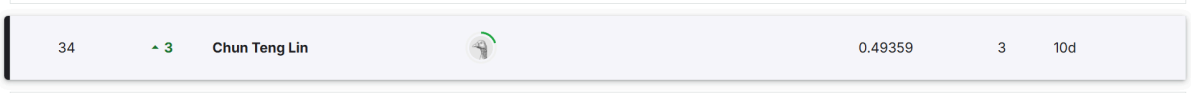
I divided the total sum of loss, accuracy, and F1 by the length to get the presented values and printed out the best result. The best result was determined by the highest Accuracy. Thus, the model training was completed.

```
avg_val_loss = sum(val_losses) / len(val_losses)
avg_val_acc = sum(val_accs) / len(val_accs)
avg_val_f1 = sum(val_f1s) / len(val_f1s)

print(f"Epoch {epoch + 1}/{epochs}:")
print(f"  Train Loss: {loss.item():.4f}, Train Acc: {train_acc:.4f}, Train F1: {train_f1:.4f}")
print(f"  Val Loss: {avg_val_loss:.4f}, Val Acc: {avg_val_acc:.4f}, Val F1: {avg_val_f1:.4f}")

if avg_val_acc > best_val_acc:
    best_val_acc = avg_val_acc
    torch.save(model.state_dict(), 'best_model_1125.pth')
    print(f"Best model saved with val accuracy: {best_val_acc:.4f}")
```

Finally, I predicted the test data and uploaded the results to the Kaggle competition, achieving an accuracy of 0.49359.



I found that the choice of model has a significant impact on the final training results. The newer and better the model, the higher the accuracy when applied to this task. However, hardware limitations are a constraint for most people, as not everyone has access to powerful resources like an RTX 4090 GPU. Therefore, I chose to use a standard RoBERTa model. The parameters within also affect training accuracy. After several attempts, I found that a learning rate of 3e-5 performed the best. Other parameters were similarly fine-tuned through step-by-step training and testing, leading to better results with specific configurations. I believe that, with societal and technological advancements, computational resources will become increasingly accessible to everyone in the future.