Python Files

File: emotion_dataset.py

```python
import torch
from torch.utils.data import Dataset

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

   def __len__(self):
       return len(self.data) # return number of samples

   def __getitem__(self, idx):
       return torch.LongTensor(self.data[idx]), torch.tensor(self.labels[idx], dtype=torch.long) # return sample and l
```

File: loader.py

```python
import os
import pandas as pd
from datasets import load_dataset

def loader(train_csv_path, val_csv_path, test_csv_path):
   # Check if the files exist; if not, load from the remote source
   if not (os.path.exists(train_csv_path) and os.path.exists(val_csv_path) and os.path.exists(test_csv_path)):
       print("Data files not found. Loading dataset from remote source...")

       os.makedirs("data", exist_ok=True)

       # Load the dataset from Hugging Face
       ds = load_dataset("dair-ai/emotion", "split")

       label_names = ds["train"].features["label"].names

       # Save train data
       train_data = {
           "text": ds["train"]["text"],
           "label": ds["train"]["label"],
           # Convert label indices to label names
           "label_name": [label_names[label] for label in ds["train"]["label"]]
       }
       pd.DataFrame(train_data).to_csv(train_csv_path, index=True)

       # Save validation data
       val_data = {
           "text": ds["validation"]["text"],
           "label": ds["validation"]["label"],
           "label_name": [label_names[label] for label in ds["validation"]["label"]]
       }
       pd.DataFrame(val_data).to_csv(val_csv_path, index=True)

       # Save test data
       test_data = {
           "text": ds["test"]["text"],
           "label": ds["test"]["label"],
           "label_name": [label_names[label] for label in ds["test"]["label"]]
       }
       pd.DataFrame(test_data).to_csv(test_csv_path, index=True)

   train_df = pd.read_csv(train_csv_path, index_col=0)
   val_df = pd.read_csv(val_csv_path, index_col=0)
   test_df = pd.read_csv(test_csv_path, index_col=0)

   return train_df, val_df, test_df
```

File: metrics.py

```python
import os
from sklearn.metrics import (
   accuracy_score,
   f1_score,
```

```
    precision_score,
    recall_score,
    classification_report,
    confusion_matrix,
)

def compute_metrics(true_labels, predicted_labels, labels):
    accuracy = round(accuracy_score(true_labels, predicted_labels), 4)
    f1 = round(f1_score(true_labels, predicted_labels, average="weighted", zero_division=0), 4)
    precision = round(precision_score(true_labels, predicted_labels, average="weighted", zero_division=0), 4)
    recall = round(recall_score(true_labels, predicted_labels, average="weighted", zero_division=0), 4)
    class_report = classification_report(true_labels, predicted_labels, target_names=labels, zero_division=0)
    conf_matrix = confusion_matrix(true_labels, predicted_labels)
    return {
        "accuracy": accuracy,
        "f1": f1,
        "precision": precision,
        "recall": recall,
        "class_report": class_report,
        "conf_matrix": conf_matrix.tolist()
    }

def print_metrics(metrics):
    for key, value in metrics.items():
        if key == "conf_matrix":
            print("confusion matrix:")
            for row in value:
                print(row)
            print()
        elif key == "class_report":
            print("classification report:")
            print(value)
        else:
            print(f"{key}: {value}")

def save_metrics(label, metrics):
    os.makedirs("results", exist_ok=True)
    with open(f"results/{label}_metrics.txt", "w") as f:
        f.write(f"Accuracy Score: {metrics['accuracy']}\n")
        f.write(f"F1-Score: {metrics['f1']}\n")
        f.write(f"Precision: {metrics['precision']}\n")
        f.write(f"Recall: {metrics['recall']}\n\n")
        f.write("Classification Report:\n")
        f.write(metrics['class_report'] + "\n")
        f.write("Confusion Matrix:\n")
        for row in metrics['conf_matrix']:
            f.write(str(row) + "\n")
```

File: plot_scores.py

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

def plot_scores(results, label):
    _, axes = plt.subplots(1, 3, figsize=(9, 3))  # Adjust the layout for 3 subplots

    epochs = range(1, results["num_epochs"] + 1)

    # Plot loss
    axes[0].plot(epochs, results["train_losses"], marker="o", markersize=3, label="Train Loss")
    axes[0].plot(epochs, results["val_losses"], marker="o", markersize=3, linestyle="--", label="Val Loss")
    axes[0].set_title(f"{label}: Loss")
    axes[0].set_xlabel("Epoch")
    axes[0].set_ylabel("Loss")
    axes[0].legend(loc="best")

    # Convert accuracy and F1-score to percentages
    train_accuracies = [acc * 100 for acc in results["train_accuracies"]]
    val_accuracies = [acc * 100 for acc in results["val_accuracies"]]
    train_f1_scores = [f1 * 100 for f1 in results["train_f1_scores"]]
    val_f1_scores = [f1 * 100 for f1 in results["val_f1_scores"]]

    # Plot accuracy
    axes[1].plot(epochs, train_accuracies, marker="o", markersize=3, label="Train Accuracy")
    axes[1].plot(epochs, val_accuracies, marker="o", markersize=3, linestyle="--", label="Val Accuracy")
    axes[1].set_title(f"{label}: Accuracy")
```

```
    axes[1].set_xlabel("Epoch")
    axes[1].set_ylabel("Accuracy (%)")
    axes[1].legend(loc="best")

    # Plot F1-score
    axes[2].plot(epochs, train_f1_scores, marker="o", markersize=3, label="Train F1-Score")
    axes[2].plot(epochs, val_f1_scores, marker="o", markersize=3, linestyle="--", label="Val F1-Score")
    axes[2].set_title(f"{label}: F1-Score")
    axes[2].set_xlabel("Epoch")
    axes[2].set_ylabel("F1-Score (%)")
    axes[2].legend(loc="best")

    # Set the number of ticks on the x and y axes for all plots
    for ax in axes:
        ax.yaxis.set_major_locator(MaxNLocator(nbins=4))
        ax.xaxis.set_major_locator(MaxNLocator(nbins=4))

    # Adjust layout and display the plots
    plt.tight_layout()
    plt.show()
```

File: predict.py

```
import os
import torch
import pandas as pd
from metrics import compute_metrics, print_metrics, save_metrics

def decode_tokens(token_ids, reverse_vocab):
    words = [reverse_vocab[token] for token in token_ids if token in reverse_vocab]
    return " ".join(words)

def predict(label, model, device, loader, label_map, reverse_vocab):
    model.to(device)
    model.eval()

    # Store predictions
    predictions = []
    true_labels = []
    predicted_labels = []

    # Ensure the results directory exists
    os.makedirs("results", exist_ok=True)

    with torch.no_grad():
        for batch in loader:
            sentences, labels = batch
            labels = labels.to(device)

            # Forward pass
            outputs = model(sentences.to(device))
            probabilities = torch.softmax(outputs, dim=1)
            predicted = torch.argmax(probabilities, dim=1)

            # Collect true and predicted labels for metrics
            true_labels.extend(labels.cpu().numpy())
            predicted_labels.extend(predicted.cpu().numpy())

            # Convert tokenized tensors back to text using decode_tokens()
            decoded_sentences = [
                decode_tokens(sentence.tolist(), reverse_vocab)
                for sentence in sentences
            ]

            for i, decoded_sentence in enumerate(decoded_sentences):
                true_label = label_map[labels[i].item()]
                pred_label = label_map[predicted[i].item()]
                confidence = probabilities[i][predicted[i].item()].item()
                correct = true_label == pred_label

                predictions.append({
                    "Sentence": decoded_sentence,
                    "Correct": correct,
                    "True Label": true_label,
                    "Predicted Label": pred_label,
                    "Confidence (%)": f"{confidence * 100:.2f}"
```

```python
            })

    # Save predictions to a CSV file
    predictions_df = pd.DataFrame(predictions)
    predictions_file = f"results/{label}_predictions.csv"
    predictions_df.to_csv(predictions_file, index=False)

    # Compute metrics
    metrics = compute_metrics(true_labels, predicted_labels, label_map.values())
    print_metrics(metrics)
    save_metrics(label, metrics)
```

File: train_model.py

```python
import os
import time
import numpy as np
from collections import Counter

from torch import nn
import torch
import torch.optim as optim
from torch.nn.utils import clip_grad_norm_
from torchinfo import summary

from sklearn.metrics import accuracy_score, f1_score

from metrics import compute_metrics, print_metrics, save_metrics

def log_undefined(predicted_labels, labels):
    counts = Counter(predicted_labels)
    for idx, label in enumerate(labels):
        if counts[idx] == 0:
            print(f"Warning: No predictions for label '{label}' (index {idx}).")


def log_undefined(predicted_labels, labels):
    """Logs warnings for labels that are not predicted."""
    counts = Counter(predicted_labels)
    for idx, label in enumerate(labels):
        if counts[idx] == 0:
            print(f"Warning: No predictions for label '{label}' (index {idx}).")


def train_model(
    label,
    model,
    train_loader,
    val_loader,
    label_map,
    device,
    optimizer_type="Adam",
    learning_rate=0.001,
    momentum=0.9,
    weight_decay=0.0,
    step_size=None,
    gamma=0.5,
    reg_type=None,
    reg_lambda=0.0,
    num_epochs=30,
    grad_clip=0.0,
):
    """Trains a PyTorch model and logs metrics for each epoch."""
    # Move the model to the device
    model = model.to(device)

    # Define the loss function
    criterion = nn.CrossEntropyLoss()

    # Select optimizer
    if optimizer_type == "SGD":
        optimizer = optim.SGD(
            model.parameters(), lr=learning_rate, momentum=momentum, weight_decay=weight_decay
        )
    elif optimizer_type == "Adam":
        optimizer = optim.Adam(
```

```python
            model.parameters(), lr=learning_rate, weight_decay=weight_decay
        )
    elif optimizer_type == "AdamW":
        optimizer = optim.AdamW(
            model.parameters(), lr=learning_rate, weight_decay=weight_decay
        )
    else:
        raise ValueError(f"Unknown optimizer type: {optimizer_type}")

    # Learning rate scheduler
    scheduler = None
    if step_size is not None and gamma is not None:
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)

    # Metrics storage
    train_losses, train_accuracies, train_f1_scores = [], [], []
    val_losses, val_accuracies, val_f1_scores = [], [], []

    total_start_time = time.time()

    for epoch in range(num_epochs):
        start_time = time.time()

        # Training phase
        model.train()

        epoch_total_train_loss = 0.0
        epoch_total_train_samples = 0
        epoch_train_true_labels = []
        epoch_train_predicted_labels = []

        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)

            # Apply regularization
            if reg_lambda > 0.0 and reg_type is not None:
                if reg_type == "L1":
                    l1_norm = sum(param.abs().sum() for param in model.parameters())
                    loss += reg_lambda * l1_norm
                elif reg_type == "L2":
                    l2_norm = sum(param.pow(2).sum() for param in model.parameters())
                    loss += reg_lambda * l2_norm

            loss.backward()

            if grad_clip > 0:
                clip_grad_norm_(model.parameters(), grad_clip)

            optimizer.step()

            epoch_total_train_loss += loss.item() * inputs.size(0)
            epoch_total_train_samples += inputs.size(0)

            # Collect true labels and predictions
            _, predicted = torch.max(outputs, dim=1)
            epoch_train_true_labels.extend(targets.cpu().numpy())
            epoch_train_predicted_labels.extend(predicted.cpu().numpy())

        # Calculate training metrics
        avg_epoch_train_loss = round(epoch_total_train_loss / epoch_total_train_samples, 4)
        epoch_train_accuracy = round(accuracy_score(epoch_train_true_labels, epoch_train_predicted_labels), 4)
        epoch_train_f1 = round(f1_score(epoch_train_true_labels, epoch_train_predicted_labels, average='weighted'), 4)
        train_losses.append(avg_epoch_train_loss)
        train_accuracies.append(epoch_train_accuracy)
        train_f1_scores.append(epoch_train_f1)

        # Validation phase
        model.eval()

        epoch_total_val_loss = 0.0
        epoch_total_val_samples = 0
        all_val_true_labels = []
        all_val_predicted_labels = []
```

```
        with torch.no_grad():
            for inputs, targets in val_loader:
                inputs, targets = inputs.to(device), targets.to(device)
                outputs = model(inputs)

                avg_epoch_val_loss = criterion(outputs, targets)
                epoch_total_val_loss += avg_epoch_val_loss.item() * inputs.size(0)
                epoch_total_val_samples += inputs.size(0)

                # Collect true labels and predictions
                _, predicted = torch.max(outputs, dim=1)
                all_val_true_labels.extend(targets.cpu().numpy())
                all_val_predicted_labels.extend(predicted.cpu().numpy())

        # Calculate validation metrics
        avg_epoch_val_loss = round(epoch_total_val_loss / epoch_total_val_samples, 4)
        epoch_val_accuracy = round(accuracy_score(all_val_true_labels, all_val_predicted_labels), 4)
        epoch_val_f1 = round(f1_score(all_val_true_labels, all_val_predicted_labels, average='weighted'), 4)
        val_losses.append(avg_epoch_val_loss)
        val_accuracies.append(epoch_val_accuracy)
        val_f1_scores.append(epoch_val_f1)

        # Update learning rate
        if scheduler:
            scheduler.step()

        epoch_duration = round(time.time() - start_time)
        print(
            f"Epoch {epoch + 1}/{num_epochs} ({epoch_duration}s) | "
            f"Train: loss {avg_epoch_train_loss}, acc {epoch_train_accuracy*100:.2f}%, f1 {epoch_train_f1*100:.2f}% | "
            f"Val: loss {avg_epoch_val_loss}, acc {epoch_val_accuracy*100:.2f}%, f1 {epoch_val_f1*100:.2f}%"
        )

    # Log undefined predictions
    log_undefined(all_val_predicted_labels, label_map.values())

    # Total training time
    total_training_time = round(time.time() - total_start_time)
    print(f"Total Training Time: {total_training_time}s\n")

    # Save model summary and metrics
    os.makedirs("models", exist_ok=True)
    with open(f"models/{label}.txt", "w", encoding="utf-8") as f:
        f.write(str(summary(model, verbose=0)))

    # Compute and save metrics
    metrics = compute_metrics(all_val_true_labels, all_val_predicted_labels, label_map.values())
    print_metrics(metrics)
    save_metrics(label, metrics)

    # Save model state
    torch.save(model.state_dict(), f"models/{label}.pth")

    # Return training history
    return {
        "num_epochs": num_epochs,
        "train_losses": train_losses,
        "train_accuracies": train_accuracies,
        "train_f1_scores": train_f1_scores,
        "val_losses": val_losses,
        "val_accuracies": val_accuracies,
        "val_f1_scores": val_f1_scores,
    }
```

File: transformer_model.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
import random
# the self attention is just like described in deep learning by Bishop, so i will not change it.
class SelfAttention(nn.Module):
    def __init__(self, d_model, d_key):
        super().__init__()
        # Three separate linear layers for the queries, keys, and values
```

```
        self.w_q = nn.Linear(d_model, d_key)
        self.w_k = nn.Linear(d_model, d_key)
        self.w_v = nn.Linear(d_model, d_model)
    def forward(self, x):
        q = self.w_q(x)
        k = self.w_k(x)
        v = self.w_v(x)
        # Compute the attention weights
        a = q @ k.transpose(-2, -1) / (k.shape[-1] ** 0.5)
        a = F.softmax(a, dim=-1)
        # Apply the attention weights
        z = a @ v
        return z
# Same as in book, shouldn't need any change
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, d_key, n_heads):
        super().__init__()
        self.heads = nn.ModuleList([SelfAttention(d_model, d_key) for _ in range(n_heads)])
        # Down projection back to model dimension
        self.w_o = nn.Linear(n_heads * d_model, d_model)
    def forward(self, x):
        return self.w_o(torch.cat([h(x) for h in self.heads], dim=-1))
# maybe change siLU activation function?
class TransformerBlock(nn.Module):
    def __init__(self, d_model, d_key, n_heads, dropout1, dropout2, mlp_factor=4, ):
        super().__init__()
        # We need to init two layer norms because they have parameters
        self.ln1 = nn.LayerNorm(d_model)
        self.attn = MultiHeadSelfAttention(d_model, d_key, n_heads)
        self.ln2 = nn.LayerNorm(d_model)
        # a feedforward module
        if dropout1 > 0:
            self.mlp = nn.Sequential(
                nn.Linear(d_model, mlp_factor * d_model),
                nn.Dropout(p = dropout1),
                nn.SiLU(),  # Swish activation function, f(x) = x * sigmoid(x)
                nn.Linear(mlp_factor * d_model, d_model),
                nn.Dropout(p = dropout2)
            )
        else:
            self.mlp = nn.Sequential(
                nn.Linear(d_model, mlp_factor * d_model),
                nn.SiLU(),  # Swish activation function, f(x) = x * sigmoid(x)
                nn.Linear(mlp_factor * d_model, d_model),
                nn.SiLU(),
                nn.Linear(d_model, mlp_factor * d_model),
                nn.SiLU(),
                nn.Linear(d_model, mlp_factor * d_model),
            )
    def forward(self, x):
        # Residual connections and pre-layernorm
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x
class TransformerClassifier(nn.Module):
    def __init__(self, n_embeds, n_classes, d_model=256, d_key=64, n_heads=2, mlp_factor=4, n_layers=2, device = "cpu",
        super().__init__()
        self.device = device
        self.d_model = d_model
        self.token_embedding = nn.Embedding(n_embeds, d_model)
        self.transformer_model = nn.Sequential(*[TransformerBlock(d_model, d_key, n_heads, dropout1, dropout2, mlp_fact
        self.final_layer_norm = nn.LayerNorm(d_model)
        self.classifier = nn.Sequential(nn.Linear(d_model, d_model), nn.SiLU(), nn.Linear(d_model, n_classes))

    def sinusoidalPositionEncoding(self, input):
        # create empty matrix
        r_n_matrix = torch.empty((input.size(dim=1), self.d_model))
        r_n_matrix = r_n_matrix.to(self.device)
        # fill all areas of empty matrix
        for n in range(input.size(dim=1)):
            for i in range(self.d_model):
                if i % 2 == 0:
                    r_n_matrix[n, i] =  math.sin(n / 10000 ** (i / self.d_model))
                if i % 2 == 1:
                    r_n_matrix[n, i] = math.cos(n / 10000 ** (i / self.d_model))
        # add with input
        input_hat = input + r_n_matrix
        # return modified input
```

```python
        return input_hat

    def forward(self, x):
        e = self.token_embedding(x)
        # sinusoidal positional encoding
        s = self.sinusoidalPositionEncoding(e)
        h = self.transformer_model(s)
        h = h.mean(dim=1) # Average pooling on the sequence dimension
        y = self.classifier(self.final_layer_norm(h))
        return y
```