Share Comment Star ◦◦◦

Assignment 6: Tweet emotion analysis

HXD220000

Harikrishna Dev

Custom *MultiLabelClassifier* class built to run multiple models without repeating multiple lines of code

The MultiLabelClassifier class is designed for training and evaluating multi-label text classification models using the Hugging Face Transformers library. It supports fine-tuning pre-trained models for multi-label classification tasks and provides prediction and hyperparameter optimization methods.

- *model_name* (str): The pre-trained model name from Hugging Face Transformers.
- *labels* (list of str): The list of labels for classification
- *batch_size* (int): Batch size for training (default is 8)
- *learning_rate* (float): Learning rate for training (default is 2e-5)
- *num_epochs* (int): Number of epochs for training (default is 5)
- *metric_name* (str): The name of the evaluation metric (default is "f1")
- *threshold* (float): Threshold for binary classification (default is 0.5)

```
# Initialize the classifier
classifier = MultiLabelClassifier(
```

```
    model_name="distilbert-base-uncased",
    labels=["positive", "negative"],
    batch_size=8,
    learning_rate=2e-5,
    num_epochs=10,
    metric_name="f1",
    threshold=0.5
)

# Train the classifier
classifier.train(train_dataset, valid_dataset)

# Optimize threshold
best_threshold = classifier.optimize_threshold(valid_dataset)

# Make predictions
predictions = classifier.predict(["This is a positive sentence", "This is a negative sentence"],
```

Code for class creation

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification, TrainingArguments, Tr
import numpy as np
from sklearn.metrics import f1_score, roc_auc_score, accuracy_score
import torch
from transformers import EvalPrediction
import optuna
from datetime import date
from sklearn.metrics import multilabel_confusion_matrix

class MultiLabelClassifier:
```

```
    def __init__(self, model_name, labels, batch_size=8, learning_rate=2e-5, num_epochs=5, metric
        """
        Initializes the MultiLabelClassifier.

        Args:
        - model_name (str): The pre-trained model name.
        - labels (list of str): The list of labels for classification.
        - batch_size (int): Batch size for training.
        - learning_rate (float): Learning rate for training.
        - num_epochs (int): Number of epochs for training.
        - metric_name (str): The name of the evaluation metric.
        - threshold (float): Threshold for binary classification.

        Returns:
        - None
        """
        self.model_name = model_name
        self.labels = labels
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.num_epochs = num_epochs
        self.metric_name = metric_name
        self.threshold = threshold
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForSequenceClassification.from_pretrained(model_name, problem_type=
        self.id2label = {str(i): label for i, label in enumerate(labels)}
        self.label2id = {label: i for i, label in enumerate(labels)}
        self.model.to(self.device)

    def preprocess_data(self, examples):
```

```
        """
        Preprocesses the input data.

        Args:
        - examples (dict): Dictionary containing input data.

        Returns:
        - dict: Preprocessed input data.
        """
        text = examples["Tweet"]
        encoding = self.tokenizer(text, padding="max_length", truncation=True, max_length=128)
        labels_batch = {k: examples[k] for k in examples.keys() if k in self.labels}
        labels_matrix = np.zeros((len(text), len(self.labels)))
        for idx, label in enumerate(self.labels):
            labels_matrix[:, idx] = labels_batch[label]
        encoding["labels"] = labels_matrix.tolist()
        return encoding

    def multi_label_metrics(self, predictions, labels, threshold=None):
        """
        Computes multi-label classification metrics.

        Args:
        - predictions (torch.Tensor): Model predictions.
        - labels (np.ndarray): Ground truth labels.
        - threshold (float): Threshold for binary classification.

        Returns:
        - dict: Dictionary containing computed metrics.
        """
        if threshold is None:
```

```python
        threshold = self.threshold
    sigmoid = torch.nn.Sigmoid()
    probs = sigmoid(torch.Tensor(predictions))
    y_pred = np.zeros(probs.shape)
    y_pred[np.where(probs >= threshold)] = 1
    y_true = labels
    f1_micro_average = f1_score(y_true=y_true, y_pred=y_pred, average='micro')
    roc_auc = roc_auc_score(y_true, y_pred, average='micro')
    accuracy = accuracy_score(y_true, y_pred)
    metrics = {'f1': f1_micro_average, 'roc_auc': roc_auc, 'accuracy': accuracy}
    return metrics

def multilabel_confusion_matrix(self, predictions, labels, threshold=None):
    """
    Computes multilabel confusion matrix.

    Args:
    - predictions (torch.Tensor): Model predictions.
    - labels (np.ndarray): Ground truth labels.
    - threshold (float): Threshold for binary classification.

    Returns:
    - np.ndarray: Multilabel confusion matrix.
    """
    if threshold is None:
        threshold = self.threshold
    sigmoid = torch.nn.Sigmoid()
    probs = sigmoid(torch.Tensor(predictions))
    y_pred = np.zeros(probs.shape)
    y_pred[np.where(probs >= threshold)] = 1
    y_true = labels
```

```python
        return multilabel_confusion_matrix(y_true, y_pred)

    def compute_metrics(self, p: EvalPrediction):
        """
        Computes evaluation metrics.

        Args:
        - p (EvalPrediction): Evaluation predictions.

        Returns:
        - dict: Dictionary containing computed metrics.
        """
        preds = p.predictions[0] if isinstance(p.predictions, tuple) else p.predictions
        result = self.multi_label_metrics(predictions=preds, labels=p.label_ids)
        return result

    def train(self, train_dataset, valid_dataset, push_to_huggingface=True):
        """
        Trains the model.

        Args:
        - train_dataset (Dataset): Training dataset.
        - valid_dataset (Dataset): Validation dataset.

        Returns:
        - None
        """
        args = TrainingArguments(
            f"{self.model_name}-finetuned",
            # evaluation_strategy="epoch",
            # save_strategy="epoch",
```

```
        learning_rate=self.learning_rate,
        per_device_train_batch_size=self.batch_size,
        per_device_eval_batch_size=self.batch_size,
        num_train_epochs=self.num_epochs,
        weight_decay=0.01,
        load_best_model_at_end=True,
        metric_for_best_model="f1",  # Use F1 score as the metric to determine the best model
        optim='adamw_torch',  # Optimizer
        # output_dir=str(model_folder),  # Directory to save model checkpoints
        evaluation_strategy='steps',  # Evaluate model at specified step intervals
        eval_steps=50,  # Perform evaluation every 50 training steps
        save_strategy="steps",  # Save model checkpoint at specified step intervals
        save_steps=1000,  # Save model checkpoint every 1000 training steps
        save_total_limit=2,  # Retain only the best and the most recent model checkpoints
        greater_is_better=True,  # A model is 'better' if its F1 score is higher
        logging_strategy='steps',  # Log metrics and results to Weights & Biases platform
        logging_steps=50,  # Log metrics and results every 50 steps
        report_to='wandb',  # Log metrics and results to Weights & Biases platform
        run_name=f"emotion_tweet_{self.model_name}_{date.today().strftime('%Y-%m-%d')}",  # E
        fp16=True  # Use mixed precision training (FP16)
        )

    train_dataset = train_dataset.map(self.preprocess_data, batched=True, remove_columns=trai
    valid_dataset = valid_dataset.map(self.preprocess_data, batched=True, remove_columns=vali

    train_dataset.set_format("torch")
    valid_dataset.set_format("torch")

    trainer = Trainer(
        self.model,
        args,
```

```
            train_dataset=train_dataset,
            eval_dataset=valid_dataset,
            tokenizer=self.tokenizer,
            compute_metrics=self.compute_metrics,
        )

        trainer.train()
        eval_results = trainer.evaluate()
        print(f"Evaluation results: {eval_results}")

        # Pushing model to Huggingface
        if push_to_huggingface:
          model_name_hf = f"emotion_tweet_{self.model_name}_{date.today().strftime('%Y-%m-%d')}"
          self.model.push_to_hub(model_name_hf)
          print(f"Model pushed to Huggingface: harikrishnad1997/{model_name_hf}")


        # Log evaluation results to Weights & Biases platform
        wandb.log({"eval_accuracy": eval_results["eval_accuracy"], "eval_loss": eval_results["eva

        # # Compute and plot confusion matrix
        # preds = trainer.predict(valid_dataset)
        # y_labels = valid_dataset[self.labels]
        # confusion_matrix = self.multilabel_confusion_matrix(preds, y_labels)
        # plt.figure(figsize=(10, 7))
        # sns.heatmap(confusion_matrix, annot=True, cmap="Blues")
        # plt.xlabel("Predicted Labels")
        # plt.ylabel("True Labels")
        # plt.title("Multilabel Confusion Matrix")
        # plt.show()
```

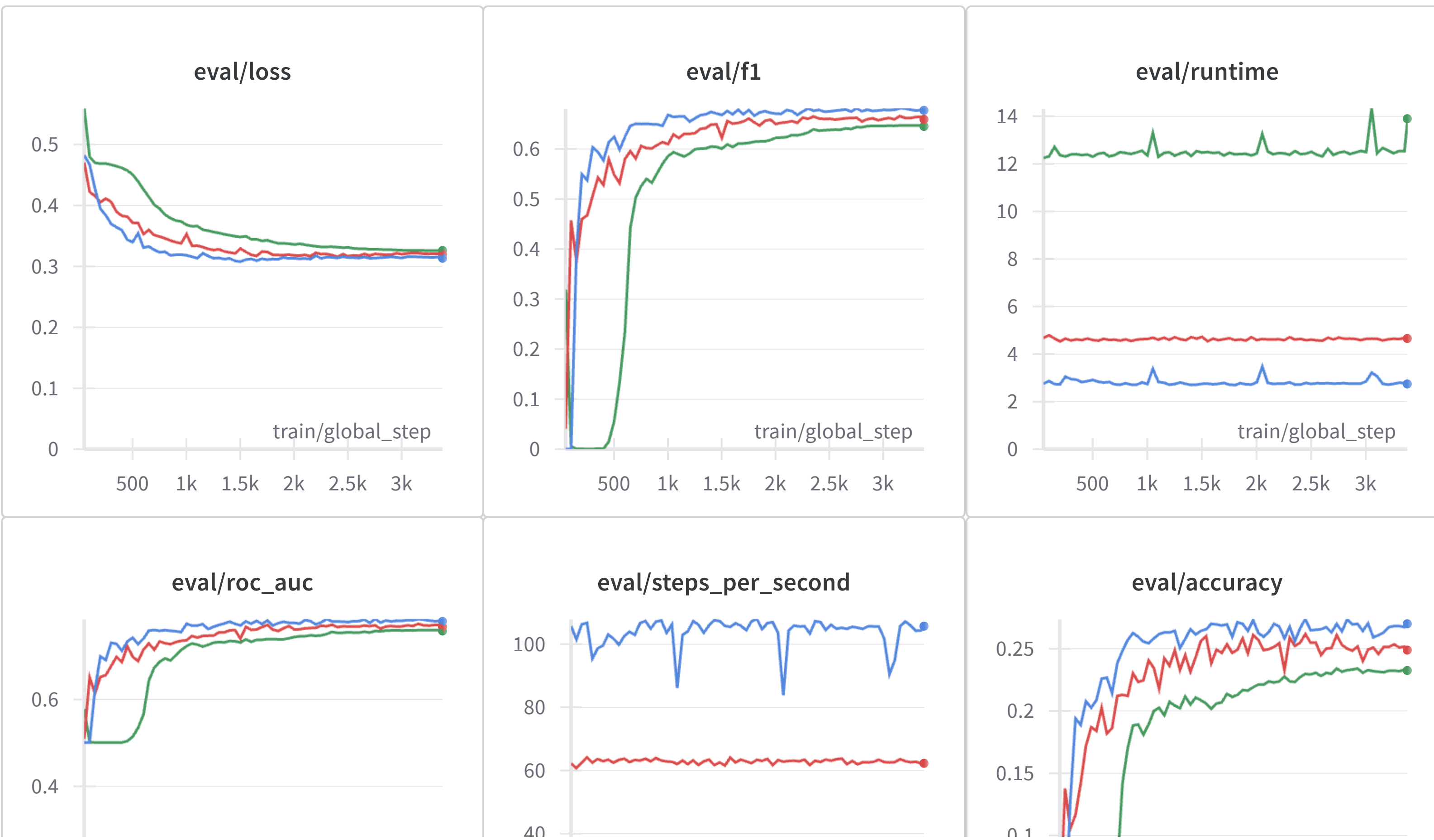
```
    # # Log confusion matrix to Weights & Biases platform
    # wandb.log({"confusion_matrix": wandb.Image(plt)})

def predict(self, texts, threshold=0.5, load_from_huggingface=False):
    """
    Generates predictions for a list of texts.

    Args:
    - texts (list of str): List of input texts.
    - threshold (float): Threshold for binary classification.

    Returns:
    - dict: Dictionary containing predicted labels for each input text.
    """
    if threshold is None:
        threshold = self.threshold

    # Load the model from Hugging Face if specified
    if load_from_huggingface:
      self.model = AutoModelForSequenceClassification.from_pretrained(load_from_huggingface)
      # self.tokenizer = AutoTokenizer.from_pretrained(load_from_huggingface)
      self.model.to("cpu")
    else:
      # Use the model from training
      self.model.to("cpu")



    # Preprocess input texts
    encoding = self.tokenizer(texts, padding="max_length", truncation=True, max_length=128, r

    # Make predictions
```

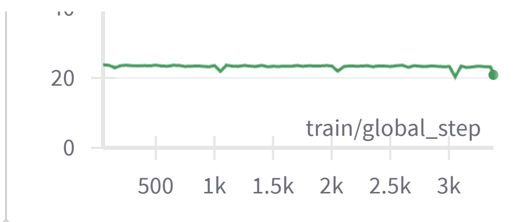
```
        with torch.no_grad():
            output = self.model(**encoding)

        # Convert logits to probabilities
        sigmoid = torch.nn.Sigmoid()
        probs = sigmoid(output.logits)

        # Apply threshold for binary classification
        threshold_tensor = torch.tensor([threshold], device="cpu")
        binary_preds = (probs >= threshold_tensor).int()

        # Convert binary predictions to label names
        label_preds = []
        for pred in binary_preds:
            label_pred = [self.id2label[str(i)] for i, val in enumerate(pred) if val == 1]
            label_preds.append(label_pred)

        return label_preds, binary_preds.cpu().numpy()

    def objective(self, trial, valid_dataset):
        """
        Objective function for hyperparameter optimization.

        Args:
        - trial (Trial): Optuna trial object.
        - valid_dataset (Dataset): Validation dataset.

        Returns:
        - float: Computed metric value.
        """
        threshold = trial.suggest_float("threshold", 0.1, 0.9)
```

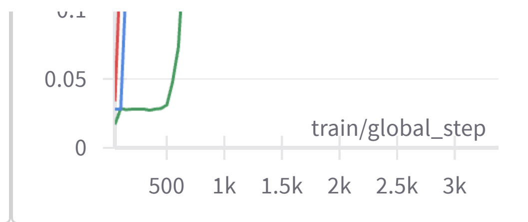
```python
        valid_dataset = valid_dataset.map(self.preprocess_data, batched=True)
        valid_dataset.set_format("torch")

        # Get the correct labels from the dataset
        labels = np.array([valid_dataset[column] for column in self.labels]).T

        # Model to cpu
        self.model.to("cpu")

        # Make predictions
        with torch.no_grad():
            logits = self.model(valid_dataset["input_ids"].to(torch.device("cpu")))['logits']
            predictions = torch.sigmoid(logits).cpu().numpy()

            # Apply threshold for binary classification
            binary_preds = (predictions >= threshold).astype(int)

            # Compute metrics
            f1_micro_average = f1_score(y_true=labels, y_pred=binary_preds, average='micro')
            roc_auc = roc_auc_score(labels, predictions, average='micro')
            accuracy = accuracy_score(labels, binary_preds)

            result = {'f1': f1_micro_average, 'roc_auc': roc_auc, 'accuracy': accuracy}
            return -result["f1"]

    def optimize_threshold(self, valid_dataset):
        """
        Optimizes the threshold for binary classification.

        Args:
        - valid_dataset (Dataset): Validation dataset.
```

```
    Returns:
    - float: Best threshold value.
    """
    study = optuna.create_study(direction="maximize")
    study.optimize(lambda trial: self.objective(trial, valid_dataset), n_trials=10)
    self.threshold = study.best_params["threshold"]
    return study.best_params["threshold"]
```

Evaluation Reports







Notes on the models:

- `distilbert-base-uncased` has the best results in terms of F1 score.
- `google/t5-base` metrics drip and then normalise around the same values.

https://wandb.ai/harikrishnad/nlp_course_spring_2024-emotion-analysis-hf-trainer-hw6/reports/Assignment-6-Tweet-emotion-analysis--Vmlldzo3NTQzNTIz