

Loading Libs :

```
import pandas as pd
import torch
import torch.nn as nn
from sklearn.model_selection import train_test_split
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from transformers import TrainingArguments, Trainer
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score, confusion_matrix, roc_curve, auc, pre
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import seaborn as sns
import json
```

Data Loading :

```
from google.colab import drive
drive.mount('/content/drive')

csv_path = "/content/drive/MyDrive/urd.csv"
df = pd.read_csv(csv_path)
df["polarization"] = df["polarization"].astype(int)
```

Mounted at /content/drive

```
train_data, val_data = train_test_split(df, test_size=0.1, random_state=42, stratify=df['polarization'])
```

Model & Tokenizer Loading :

```
MODEL_NAME = "xlm-roberta-large"
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForSequenceClassification.from_pretrained(MODEL_NAME, num_labels=2)
```

/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/tokens>), set
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.

```
warnings.warn(
tokenizer_config.json: 100% 25.0/25.0 [00:00<00:00, 2.40kB/s]
config.json: 100% 616/616 [00:00<00:00, 69.8kB/s]
sentencepiece.bpe.model: 100% 5.07M/5.07M [00:00<00:00, 13.2MB/s]
tokenizer.json: 100% 9.10M/9.10M [00:00<00:00, 20.6MB/s]
model.safetensors: 100% 2.24G/2.24G [00:20<00:00, 149MB/s]
```

Some weights of XLMRobertaForSequenceClassification were not initialized from the model checkpoint at xlm-roberta-large and
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Dataset Class and Training Arguments :

```
class PolarDataset(torch.utils.data.Dataset):
    def __init__(self, df, tokenizer, max_len=256):
        self.texts = df["text"].tolist()
        self.labels = df["polarization"].tolist()
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def __getitem__(self, idx):
        encoding = self.tokenizer(
            self.texts[idx],
            padding="max_length",
            truncation=True,
            max_length=self.max_len,
            return_tensors="pt"
        )
        return {
            "input_ids": encoding["input_ids"].squeeze(),
            "attention_mask": encoding["attention_mask"].squeeze(),
```

```

        "labels": torch.tensor(self.labels[idx], dtype=torch.long)
    }

train_dataset = PolarDataset(train_data, tokenizer)
val_dataset = PolarDataset(val_data, tokenizer)

```

```

class_counts = df["polarization"].value_counts().sort_index().tolist()
class_weights = torch.tensor(
    [sum(class_counts)/c for c in class_counts]
).float().to("cuda")

print("Class Weights:", class_weights)

class WeightedTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False, **kwargs):
        labels = inputs.get("labels")
        outputs = model(**inputs)
        logits = outputs.get("logits")

        loss_fct = nn.CrossEntropyLoss(weight=class_weights)
        loss = loss_fct(logits, labels)

        return (loss, outputs) if return_outputs else loss

```

```
Class Weights: tensor([3.2635, 1.4418], device='cuda:0')
```

```

def compute_metrics(p):
    preds = p.predictions.argmax(-1)
    labels = p.label_ids
    return {
        "accuracy": accuracy_score(labels, preds),
        "precision": precision_score(labels, preds),
        "recall": recall_score(labels, preds),
        "f1": f1_score(labels, preds)
    }

training_args = TrainingArguments(
    output_dir="./model",
    learning_rate=1e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=6,
    gradient_accumulation_steps=4,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=50,
    report_to=[],
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    metric_for_best_model="eval_f1",
    greater_is_better=True,
    fp16=True
)

trainer = WeightedTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics
)

```

Training :

```
trainer.train()
```

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.696600	0.582107	0.684211	0.833333	0.681818	0.750000
2	0.544100	0.508983	0.771930	0.848168	0.818182	0.832905
3	0.479400	0.520407	0.775439	0.825243	0.858586	0.841584
4	0.430400	0.485719	0.768421	0.866667	0.787879	0.825397
5	0.348500	0.543840	0.789474	0.863158	0.828283	0.845361
6	0.330000	0.552525	0.775439	0.860215	0.808081	0.833333

Results & Saving Model :

Saving Model to Drive :

Manual Testing :

3/6

```

        (intermediate_act_fn): GELUActivation()
    )
    (output): XLMRobertaOutput(
      (dense): Linear(in_features=4096, out_features=1024, bias=True)
      (LayerNorm): LayerNorm((1024,)), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
)
)
)
(classifier): XLMRobertaClassificationHead(
  (dense): Linear(in_features=1024, out_features=1024, bias=True)
  (dropout): Dropout(p=0.1, inplace=False)
  (out_proj): Linear(in_features=1024, out_features=2, bias=True)
)
)

```

```

def test_manual(text):
    inputs = tokenizer(
        text,
        padding=True,
        truncation=True,
        max_length=256,
        return_tensors="pt"
    )

    with torch.no_grad():
        outputs = model(**inputs)
        logits = outputs.logits
        pred = torch.argmax(logits, dim=1).item()

    label = "Polarized" if pred == 1 else "Not Polarized"
    return pred, label

```

```

text = "ٹرمپ کرپٹ اور غیر ذمہ دار ہے"
pred, label = test_manual(text)

```

```

print("Prediction:", pred)
print("Meaning:", label)

```

```

Prediction: 1
Meaning: Polarized

```

```

def get_predictions(model, dataset):
    model.eval()
    preds = []
    labels = []
    probs = []
    loader = torch.utils.data.DataLoader(dataset, batch_size=16)
    with torch.no_grad():
        for batch in loader:
            outputs = model(input_ids=batch["input_ids"], attention_mask=batch["attention_mask"])
            pred = torch.argmax(outputs.logits, dim=1)
            prob = torch.softmax(outputs.logits, dim=1)[:,-1]
            preds.extend(pred.tolist())
            labels.extend(batch["labels"].tolist())
            probs.extend(prob.tolist())
    return labels, preds, probs

y_true, y_pred, y_probs = get_predictions(model, val_dataset)

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

fpr, tpr, _ = roc_curve(y_true, y_probs)
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, label=f"ROC AUC = {roc_auc:.2f}")
plt.plot([0,1],[0,1], '--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()

precision, recall, _ = precision_recall_curve(y_true, y_probs)
pr_auc = auc(recall, precision)

```

```

pr_auc = auc(recall, precision)
plt.plot(recall, precision, label=f"PR AUC = {pr_auc:.2f}")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve")
plt.legend()
plt.show()

log_history = trainer.state.log_history
train_losses = [entry["loss"] for entry in log_history if "loss" in entry]
val_losses = [entry["eval_loss"] for entry in log_history if "eval_loss" in entry]
steps = [entry["step"] for entry in log_history if "loss" in entry]

plt.figure(figsize=(10,5))
plt.plot(steps, train_losses, label="Training Loss")
plt.plot(steps[:len(val_losses)], val_losses, label="Validation Loss")
plt.xlabel("Steps")
plt.ylabel("Loss")
plt.title("Training vs Validation Loss")
plt.legend()
plt.grid(True)
plt.show()

val_acc = [entry["eval_accuracy"] for entry in log_history if "eval_accuracy" in entry]
val_f1 = [entry["eval_f1"] for entry in log_history if "eval_f1" in entry]
val_prec = [entry["eval_precision"] for entry in log_history if "eval_precision" in entry]
val_rec = [entry["eval_recall"] for entry in log_history if "eval_recall" in entry]
epochs = list(range(1, len(val_acc)+1))

plt.figure(figsize=(10,5))
plt.plot(epochs, val_acc, label="Accuracy")
plt.plot(epochs, val_f1, label="F1 Score")
plt.plot(epochs, val_prec, label="Precision")
plt.plot(epochs, val_rec, label="Recall")
plt.xlabel("Epochs")
plt.ylabel("Metric Value")
plt.title("Validation Metrics Over Epochs")
plt.legend()
plt.grid(True)
plt.show()

metrics_dict = {
    "confusion_matrix": cm.tolist(),
    "roc_curve": {"fpr": fpr.tolist(), "tpr": tpr.tolist(), "auc": roc_auc},
    "precision_recall_curve": {"precision": precision.tolist(), "recall": recall.tolist(), "pr_auc": pr_auc},
    "train_validation_loss": {"steps": steps, "train_loss": train_losses, "val_loss": val_losses},
    "validation_metrics_over_epochs": {"epochs": epochs, "accuracy": val_acc, "f1": val_f1, "precision": val_prec, "recall":
}

with open("/content/drive/MyDrive/urd_model_metrics.json", "w") as f:
    json.dump(metrics_dict, f, indent=4)

```

