Importing Libraries

```
import pandas as pd
import torch
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

import matplotlib.pyplot as plt
import seaborn as sns
import json
from sklearn.metrics import confusion_matrix
from torch.utils.data import DataLoader
```

Loading Data

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", for
```

```
csv_path = "/content/drive/MyDrive/prep2_train.csv"
```

```
df = pd.read_csv(csv_path)
df["polarization"] = df["polarization"].astype(int)
```

Train Split

```
train_data, val_data = train_test_split(df, test_size=0.1, random_state=42)
```

Tokenizer Loading

```
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-multilingual-cased")
```

```
/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
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(
```

Load the Model

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-multilingual-cased", num_labels=2)
```

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

Dataset Class

```
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)
```

Metrics and Trainging Arguments

```
    from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score

    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=2e-5,
        per_device_train_batch_size=8,
        per_device_eval_batch_size=8,
        num_train_epochs=3,
        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_accuracy",
        greater_is_better=True,

        fp16=True
    )
```

Trainer

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

Training

```
    trainer.train()
```

[1365/1365 03:30, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.545200	0.585100	0.688119	0.735632	0.615385	0.670157
2	0.446500	0.554199	0.725248	0.734300	0.730769	0.732530
3	0.333400	0.688744	0.710396	0.700441	0.764423	0.731034

Results

```
results = trainer.evaluate()
print(results)
```

[51/51 00:00]

```
{'eval_loss': 0.5541993975639343, 'eval_accuracy': 0.7252475247524752, 'eval_precision': 0.7342995169082126, 'e
```

Save Model

```
trainer.save_model("./model")
tokenizer.save_pretrained("./model")
```

```
('./model/tokenizer_config.json',
 './model/special_tokens_map.json',
 './model/vocab.txt',
 './model/added_tokens.json',
 './model/tokenizer.json')
```

Manual Test

```
model = AutoModelForSequenceClassification.from_pretrained("./model")
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-multilingual-cased")
```

```
model.eval()
```

```
DistilBertForSequenceClassification(
  (distilbert): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(119547, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0-5): 6 x TransformerBlock(
          (attention): DistilBertSdpaAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
  )
  (pre_classifier): Linear(in_features=768, out_features=768, bias=True)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

```
text = "911 is done by israel and israel should die and it is corrupt. trump should die"
```

```
inputs = tokenizer(
    text,
    padding=True,
```

```
    truncation=True,
    max_length=256,
    return_tensors="pt"
)
```

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

print(predicted_class)
if predicted_class == 1:
    print("Polarized")
else:
    print("Not Polarized")
```

```
1
Polarized
```

```
model.save_pretrained("/content/drive/MyDrive/my_model_3")
tokenizer.save_pretrained("/content/drive/MyDrive/my_model_3")
```

```
('/content/drive/MyDrive/my_model_3/tokenizer_config.json',
 '/content/drive/MyDrive/my_model_3/special_tokens_map.json',
 '/content/drive/MyDrive/my_model_3/vocab.txt',
 '/content/drive/MyDrive/my_model_3/added_tokens.json',
 '/content/drive/MyDrive/my_model_3/tokenizer.json')
```

Plots and Metrics

```
log_history = trainer.state.log_history

train_losses = []
val_losses = []
val_accuracies = []
val_precisions = []
val_recalls = []
steps = []

for entry in log_history:
    if "loss" in entry:
        train_losses.append(entry["loss"])
        steps.append(entry["step"])
    if "eval_loss" in entry:
        val_losses.append(entry["eval_loss"])
    if "eval_accuracy" in entry:
        val_accuracies.append(entry["eval_accuracy"])
    if "eval_precision" in entry:
        val_precisions.append(entry["eval_precision"])
    if "eval_recall" in entry:
        val_recalls.append(entry["eval_recall"])

plt.figure(figsize=(10,5))
plt.plot(steps[:len(train_losses)], 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()

plt.figure(figsize=(10,5))
plt.plot(steps[:len(val_accuracies)], val_accuracies, label="Validation Accuracy")
plt.xlabel("Steps")
plt.ylabel("Accuracy")
plt.title("Validation Accuracy Curve")
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(10,5))
plt.plot(steps[:len(val_precisions)], val_precisions, label="Validation Precision")
```

```
plt.xlabel("Steps")
plt.ylabel("Precision")
plt.title("Validation Precision Curve")
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(10,5))
plt.plot(steps[:len(val_recalls)], val_recalls, label="Validation Recall")
plt.xlabel("Steps")
plt.ylabel("Recall")
plt.title("Validation Recall Curve")
plt.legend()
plt.grid(True)
plt.show()

def get_predictions(model, dataset):
    model.eval()
    preds = []
    labels = []
    loader = 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)
            preds.extend(pred.tolist())
            labels.extend(batch["labels"].tolist())
    return labels, preds

true_labels, pred_labels = get_predictions(model, val_dataset)

cm = confusion_matrix(true_labels, pred_labels)

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()

from sklearn.metrics import roc_curve, auc, precision_recall_curve

probs = []
labels_list = []
model.eval()
loader = DataLoader(val_dataset, batch_size=16)
with torch.no_grad():
    for batch in loader:
        outputs = model(
            input_ids=batch["input_ids"],
            attention_mask=batch["attention_mask"]
        )
        prob = torch.softmax(outputs.logits, dim=1)[:,1]
        probs.extend(prob.tolist())
        labels_list.extend(batch["labels"].tolist())

fpr, tpr, _ = roc_curve(labels_list, probs)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, color='darkorange', label='ROC curve (AUC = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend(loc="lower right")
plt.grid(True)
plt.show()

precision, recall, _ = precision_recall_curve(labels_list, probs)
plt.figure(figsize=(8,6))
plt.plot(recall, precision, color='green')
plt.xlabel("Recall")
```

```
plt.ylabel("Precision")
plt.title("Precision-Recall Curve")
plt.grid(True)
plt.show()

metrics_data = {
    "train_loss": train_losses,
    "val_loss": val_losses,
    "val_accuracy": val_accuracies,
    "val_precision": val_precisions,
    "val_recall": val_recalls,
    "confusion_matrix": cm.tolist()
}

with open("training_metrics.json", "w") as f:
    json.dump(metrics_data, f, indent=4)
```