Link to Presentation: https://docs.google.com/presentation/d/1V8hxflZYdcIonr7nF7swYjx8bbf4BihpolLrBHgHo8c/edit?usp=sharing

Data 440 Machine Learning Project 2

The assignment

↳ 1 cell hidden

The deliverables

↳ 1 cell hidden

The rubric

↳ 1 cell hidden

Import and Clean Data Set

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
adult_1 = pd.read_csv('https://github.com/cari-lin/class_data/raw/refs/heads/main/adult.data', header=None)
adult_2 = pd.read_csv('https://github.com/cari-lin/class_data/raw/refs/heads/main/adult.test', skiprows=1, header=None)
```

```
adult_m1 = pd.concat([adult_1, adult_2])
adult_m1.columns = ['age', 'workclass', 'fnlwgt', 'education', 'education-num', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'capital-gain', 'capital-loss', 'hours-per-week', 'native-country', 'income']
display(adult_m1.shape)
display(adult_m1.head())
```

Show hidden output

```
adult_m1 = adult_m1.replace(' ?', np.nan)
adult_m1 = adult_m1.dropna(subset=['income']) # Drop rows with NaN in the 'income' column
adult_m1 = adult_m1.dropna() # Drop remaining rows with NaN in any column
adult_m1 = adult_m1.reset_index(drop=True) # Reset index after cleaning

display(adult_m1.head())
print(f"Shape after cleaning: {adult_m1.shape}")
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

```
Shape after cleaning: (45222, 15)
```

```
adult_m1.isnull().sum()
```

Show hidden output

```
adult_m1.shape
```

```
(45222, 15)
```

First Attempt: Creating Sentences

```python
# Creating Sentences
def row_to_text(row):
    return (
        f"This person is {row['age']} years old, works in {row['workclass']}, "
        f"has {row['education']} education, is {row['marital-status']}, "
        f"works in the {row['occupation']} occupation, identifies as {row['sex']}, "
        f"is of {row['race']} race, works {row['hours-per-week']} hours per week, "
        f"and is from {row['native-country']}."
    )

adult_m1['text'] = adult_m1.apply(row_to_text, axis=1)

adult_m1[['text', 'income']].head()
```

	text	income
0	This person is 39 years old, works in State-g...	<=50K
1	This person is 50 years old, works in Self-em...	<=50K
2	This person is 38 years old, works in Private...	<=50K
3	This person is 53 years old, works in Private...	<=50K
4	This person is 28 years old, works in Private...	<=50K

```python
# Encode labels
adult_m1['label'] = adult_m1['income'].apply(lambda x: 1 if '>50K' in x else 0)
```

```python
# Split the data
from sklearn.model_selection import train_test_split

train_df, test_df = train_test_split(adult_m1[['text', 'label']],
                                     test_size=0.2,
                                     random_state=42,
                                     stratify=adult_m1['label'])
```

```python
# Load a Transformer (DistilBERT)
from transformers import DistilBertTokenizerFast, DistilBertForSequenceClassification

tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

model = DistilBertForSequenceClassification.from_pretrained(
    'distilbert-base-uncased',
    num_labels=2
)
```

```
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight', 'pre_classifier.bias', 'pre_classifier.we
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

```python
# Tokenize the text
def tokenize(batch):
    return tokenizer(batch['text'], padding=True, truncation=True)

from datasets import Dataset

train_ds = Dataset.from_pandas(train_df)
test_ds = Dataset.from_pandas(test_df)

train_ds = train_ds.map(tokenize, batched=True)
test_ds = test_ds.map(tokenize, batched=True)
```

```
Map: 100%  36177/36177 [00:05<00:00, 6229.72 examples/s]
Map: 100%  9045/9045 [00:02<00:00, 4349.67 examples/s]
```

```python
# Train with huggingface trainer
from transformers import TrainingArguments, Trainer
import os
os.environ["WANDB_DISABLED"] = "true"

training_args = TrainingArguments(
    output_dir='./results',
    eval_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir='./logs'
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=test_ds,
    tokenizer=tokenizer
)
```

```
Using the `WANDB_DISABLED` environment variable is deprecated and will be removed in v5. Use the --report_to flag to control the integrations used for logging result (for instance --report_to none).
/tmp/ipython-input-1519299961.py:17: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

```
# Evaluate
trainer.evaluate()
```

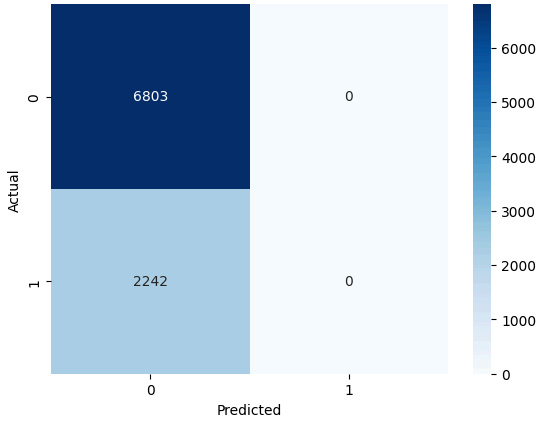
```
[566/566 00:19]
{'eval_loss': 0.6318663954734802,
 'eval_model_preparation_time': 0.0013,
 'eval_runtime': 19.3201,
 'eval_samples_per_second': 468.164,
 'eval_steps_per_second': 29.296}
```

```
# Make some predictions
import numpy as np

# Get predictions
preds_output = trainer.predict(test_ds)

# The predictions are logits; take the argmax to get class labels
preds = np.argmax(preds_output.predictions, axis=1)

# True labels
labels = preds_output.label_ids
```

```
# Compute accuracy, precision, recall, F1
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

accuracy = accuracy_score(labels, preds)
precision = precision_score(labels, preds)
recall = recall_score(labels, preds)
f1 = f1_score(labels, preds)
cm = confusion_matrix(labels, preds)

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
print("Confusion Matrix:")
print(cm)
```

```
Accuracy: 0.7521
Precision: 0.0000
Recall: 0.0000
F1 Score: 0.0000
Confusion Matrix:
[[6803    0]
 [2242    0]]
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```



Training with the weighted model

```
# Adding Class Weights
import torch

# Count how many samples are in each class
labels = torch.tensor(train_df['label'].values)
class_counts = torch.bincount(labels)
class_weights = 1.0 / class_counts.float()
class_weights = class_weights.to(torch.float32)

print("Class weights:", class_weights)
```

```
Class weights: tensor([3.6750e-05, 1.1153e-04])
```

```
from transformers import DistilBertForSequenceClassification

# Redefine the model with class weights
class WeightedDistilBert(DistilBertForSequenceClassification):
    def __init__(self, config, class_weights):
        super().__init__(config)
        self.class_weights = class_weights

    def forward(self, input_ids=None, attention_mask=None, labels=None):
        outputs = super().forward(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        logits = outputs.logits
        loss = None
        if labels is not None:
            # Use weighted cross entropy
            loss_fct = torch.nn.CrossEntropyLoss(weight=self.class_weights.to(logits.device))
            loss = loss_fct(logits, labels)
        return {"loss": loss, "logits": logits}
```

```
from transformers import DistilBertConfig

config = DistilBertConfig.from_pretrained('distilbert-base-uncased', num_labels=2)
model = WeightedDistilBert(config, class_weights)
```

```
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
    output_dir='./results',
    eval_strategy="epoch",
    learning_rate=3e-5,           # slightly higher LR can help
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=2,           # train a bit longer
    weight_decay=0.01,
    logging_dir='./logs',
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=test_ds,
    tokenizer=tokenizer
)
trainer.train()
```

```
Using the `WANDB_DISABLED` environment variable is deprecated and will be removed in v5. Use the --report_to flag to control the integrations used for logging result (for instance --report_to none).
/tmp/ipython-input-3261885379.py:14: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[4524/4524 10:32, Epoch 2/2]

Epoch	Training Loss	Validation Loss
1	0.451400	0.464605
2	0.422100	0.421757

```
TrainOutput(global_step=4524, training_loss=0.4516870136497086, metrics={'train_runtime': 633.2139, 'train_samples_per_second': 114.265, 'train_steps_per_second': 7.145, 'total_flos': 1365273433170348.0, 'train_loss':
0.4516870136497086, 'epoch': 2.0})
```

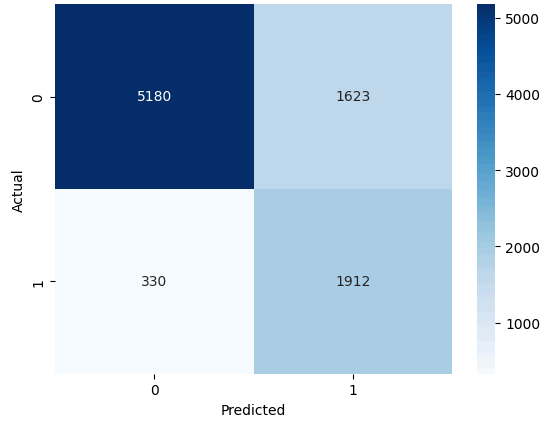
```
preds_output = trainer.predict(test_ds)
preds = np.argmax(preds_output.predictions, axis=1)
labels = preds_output.label_ids

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

accuracy = accuracy_score(labels, preds)
precision = precision_score(labels, preds)
recall = recall_score(labels, preds)
f1 = f1_score(labels, preds)
cm = confusion_matrix(labels, preds)

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
```

```
print(f"F1 Score: {f1:.4f}")
print("Confusion Matrix:")
print(cm)
```

```
Accuracy: 0.7841
Precision: 0.5409
Recall: 0.8528
F1 Score: 0.6619
Confusion Matrix:
[[5180 1623]
 [ 330 1912]]
```

```
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
print("Confusion Matrix:")
print(cm)
import seaborn as sns
import matplotlib.pyplot as plt

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

```
Accuracy: 0.7841
Precision: 0.5409
Recall: 0.8528
F1 Score: 0.6619
Confusion Matrix:
[[5180 1623]
 [ 330 1912]]
```



Key Takeaways:

Improved Minority Class Detection: The Recall for the positive class (income >50K) shot up from 0.2623 to 0.8528! This means your model is now much better at correctly identifying individuals who earn more than 50K. This is precisely what you aimed for by introducing class weights. Balanced Performance: The F1 Score for the positive class more than doubled, going from 0.3133 to 0.6619. This indicates a much better balance between precision and recall for that crucial minority class. Overall Accuracy Boost: Even the overall Accuracy improved from 0.7150 to 0.7841, which is a great bonus. Trade-off in False Positives: As expected with class weighting, to achieve better recall for the minority class, the model is now more willing to predict the positive class. This has resulted in an increase in False Positives (from 924 to 1623), meaning more people earning <=50K are now incorrectly classified as >50K.

Run New Optuna Study with 3 Trials

Reasoning: The first set of instructions involves preparing the data for the Optuna study: creating a copy of the DataFrame, enhancing the text feature, encoding the target variable, splitting the data, converting it to Dataset objects, and tokenizing it. This also includes calculating the new class weights for the Optuna study, which are crucial for addressing class imbalance during training.

```
import torch
from sklearn.model_selection import train_test_split
from datasets import Dataset

# 1. Create a copy of the adult_m1 DataFrame
adult_m1_optuna = adult_m1.copy()

# 2. Define an enhanced row_to_text_enhanced function
def row_to_text_enhanced(row):
```

```
    return (
        f"Age is {row['age']}, Workclass is {row['workclass']}, Fnlwgt is {row['fnlwgt']}, "
        f"Education is {row['education']}, Education-num is {row['education-num']}, "
        f"Marital-status is {row['marital-status']}, Occupation is {row['occupation']}, "
        f"Relationship is {row['relationship']}, Race is {row['race']}, "
        f"Sex is {row['sex']}, Capital-gain is {row['capital-gain']}, "
        f"Capital-loss is {row['capital-loss']}, Hours-per-week is {row['hours-per-week']}, "
        f"Native-country is {row['native-country']}."
    )

adult_m1_optuna['text'] = adult_m1_optuna.apply(row_to_text_enhanced, axis=1)

# 3. Encode the income column into numerical labels
adult_m1_optuna['label'] = adult_m1_optuna['income'].apply(lambda x: 1 if '>50K' in x else 0)

# 4. Split the adult_m1_optuna DataFrame into training and testing sets
train_df_optuna, test_df_optuna = train_test_split(adult_m1_optuna[['text', 'label']],
                                                   test_size=0.2,
                                                   random_state=42,
                                                   stratify=adult_m1_optuna['label'])

# 5. Convert train_df_optuna and test_df_optuna into datasets.Dataset objects
train_ds_optuna = Dataset.from_pandas(train_df_optuna)
test_ds_optuna = Dataset.from_pandas(test_df_optuna)

# 6. Tokenize the text column
def tokenize(batch):
    return tokenizer(batch['text'], padding=True, truncation=True)

train_ds_optuna = train_ds_optuna.map(tokenize, batched=True)
test_ds_optuna = test_ds_optuna.map(tokenize, batched=True)

# 7. Calculate new class weights
labels_optuna = torch.tensor(train_df_optuna['label'].values)
class_counts_optuna = torch.bincount(labels_optuna)
class_weights_optuna = 1.0 / class_counts_optuna.float()
class_weights_optuna = class_weights_optuna.to(torch.float32)

print("Data preparation and tokenization complete for Optuna study.")
print("New class weights for Optuna study:", class_weights_optuna)

```

```
Map: 100%                           36177/36177 [00:09<00:00, 4564.01 examples/s]
Map: 100%                           9045/9045 [00:02<00:00, 4592.43 examples/s]
Data preparation and tokenization complete for Optuna study.
New class weights for Optuna study: tensor([3.6750e-05, 1.1153e-04])
```

Reasoning: The next step is to define the `compute_metrics` function, which will be used by the `Trainer` to evaluate the model's performance during the Optuna study, specifically focusing on metrics like accuracy, precision, recall, and F1 score for the positive class.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np

# 8. Define a compute_metrics function
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = np.argmax(predictions, axis=1)

    accuracy = accuracy_score(labels, predictions)
    # Use 'binary' for precision, recall, and f1 for the positive class (1)
    precision = precision_score(labels, predictions, pos_label=1, average='binary', zero_division=0)
    recall = recall_score(labels, predictions, pos_label=1, average='binary', zero_division=0)
    f1 = f1_score(labels, predictions, pos_label=1, average='binary', zero_division=0)

    return {
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1': f1
    }

print("compute_metrics function defined.")
```

```
compute_metrics function defined.
```

Reasoning: Now that the data is prepared and the `compute_metrics` function is defined, the next step is to define the Optuna `objective` function. This function will encapsulate the model training and evaluation process, including hyperparameter suggestion, model instantiation, `TrainingArguments` setup, `Trainer` initialization, training, evaluation, and returning the target metric (F1 score) for Optuna to optimize.

```
!pip install optuna
print("Optuna installed.")
```

Show hidden output

```
import optuna # Ensure Optuna is imported
from transformers import TrainingArguments, Trainer, DistilBertConfig
import os

# Ensure WANDB is disabled for Optuna trials if not explicitly needed
os.environ["WANDB_DISABLED"] = "true"

# 9. Define an objective function for Optuna
def objective(trial):
    # a. Suggest hyperparameters
    learning_rate = trial.suggest_float('learning_rate', 1e-5, 5e-5, log=True)
    per_device_train_batch_size = trial.suggest_categorical('per_device_train_batch_size', [16, 32])
    num_train_epochs = trial.suggest_categorical('num_train_epochs', [2, 3])

    # b. Instantiate the WeightedDistilBert model
    config = DistilBertConfig.from_pretrained('distilbert-base-uncased', num_labels=2)
    # Make sure to pass class_weights_optuna which was calculated in the previous step
    model = WeightedDistilBert(config, class_weights_optuna)

    # c. Set up TrainingArguments
    training_args = TrainingArguments(
        output_dir=f'./results_optuna/trial_{trial.number}',
        eval_strategy="epoch",
        learning_rate=learning_rate,
        per_device_train_batch_size=per_device_train_batch_size,
        per_device_eval_batch_size=16, # Fixed eval batch size
        num_train_epochs=num_train_epochs,
        weight_decay=0.01,
        logging_dir=f'./logs_optuna/trial_{trial.number}',
        report_to="none" # Explicitly disable reporting to external services like W&B
    )

    # d. Initialize a Trainer instance
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_ds_optuna,
        eval_dataset=test_ds_optuna,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics
    )

    # e. Train the model
    trainer.train()

    # f. Evaluate the model
    eval_results = trainer.evaluate()

    # g. Return the 'eval_f1' score for optimization
    return eval_results['eval_f1']

print("Optuna objective function defined.")
```

```
Optuna objective function defined.
```

Reasoning: The `objective` function has been defined. The next step is to create an Optuna study, run it for 3 trials, and then print the best result, which aligns with instructions 11, 12, and 13 of the subtask.

```
import optuna

# 11. Create an Optuna study
study = optuna.create_study(direction='maximize', study_name='distilbert_weighted_f1_study')

# 12. Run the Optuna study with 3 trials
study.optimize(objective, n_trials=3)

# 13. Print the best trial's value and parameters
print("Optuna study complete.")
print(f"Best trial value (F1 score): {study.best_value:.4f}")
print("Best trial parameters:")
for key, value in study.best_params.items():
    print(f"  {key}: {value}")
```

```
[I 2025-12-05 21:07:31,376] A new study created in memory with name: distilbert_weighted_f1_study
/tmp/ipython-input-2763113740.py:34: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[3393/3393 21:37, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.439100	0.430860	0.830956	0.641188	0.722123	0.679253
2	0.410800	0.397237	0.818463	0.601283	0.794380	0.684473
3	0.386800	0.392608	0.805860	0.573414	0.846566	0.683718

[566/566 00:23]

```
[I 2025-12-05 21:29:36,116] Trial 0 finished with value: 0.6837175792507204 and parameters: {'learning_rate': 1.7387201011757042e-05, 'per_device_train_batch_size': 32, 'num_train_epochs': 3}. Best is trial 0 with value: 0.6837175792507
/tmp/ipython-input-2763113740.py:34: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[4524/4524 17:47, Epoch 2/2]

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.433600	0.414012	0.816473	0.597065	0.798394	0.683206
2	0.372600	0.384325	0.806412	0.572914	0.860393	0.687823

[566/566 00:24]

```
[I 2025-12-05 21:47:49,415] Trial 1 finished with value: 0.6878231413799252 and parameters: {'learning_rate': 2.474945739792247e-05, 'per_device_train_batch_size': 16, 'num_train_epochs': 2}. Best is trial 1 with value: 0.6878231413799252.
/tmp/ipython-input-2763113740.py:34: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[4524/4524 15:41, Epoch 2/2]

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.442900	0.406232	0.792482	0.552761	0.852810	0.670760
2	0.392800	0.400749	0.788281	0.545966	0.866191	0.669771

[566/566 00:24]

```
[I 2025-12-05 22:03:57,808] Trial 2 finished with value: 0.6697706501120883 and parameters: {'learning_rate': 1.6441410395454183e-05, 'per_device_train_batch_size': 16, 'num_train_epochs': 2}. Best is trial 1 with value: 0.6878231413799252.
Optuna study complete.
Best trial value (F1 score): 0.6878
Best trial parameters:
  learning_rate: 2.474945739792247e-05
  per_device_train_batch_size: 16
  num_train_epochs: 2
```

```python
########################################
# 8. OPTUNA: Define Objective Function
########################################

def objective(trial):
    # Suggest hyperparameters
    learning_rate = trial.suggest_float("learning_rate", 1e-6, 5e-5, log=True)
    per_device_batch_size = trial.suggest_categorical("per_device_batch_size", [8, 16, 32])
    weight_decay = trial.suggest_float("weight_decay", 0.0, 0.3)
    num_train_epochs = trial.suggest_int("num_train_epochs", 1, 3)

    # Training arguments
    training_args = TrainingArguments(
        output_dir=f"results_optuna_{trial.number}",
        eval_strategy="epoch",
        learning_rate=learning_rate,
        per_device_train_batch_size=per_device_batch_size,
        per_device_eval_batch_size=per_device_batch_size,
        num_train_epochs=num_train_epochs,
        weight_decay=weight_decay,
        logging_steps=50,
        save_strategy="no",
        report_to=[]
    )

    # Trainer
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_ds,
        eval_dataset=test_ds,
        tokenizer=tokenizer
    )

    # Train
    trainer.train()

    # Evaluate
    eval_results = trainer.evaluate()
    return eval_results["eval_loss"]


########################################
# 9. RUN OPTUNA STUDY
########################################

study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=5)
```

```

print("Best Trial:")
print(study.best_trial)
print("Best Parameters:", study.best_params)


########################################
# 10. RETRAIN MODEL WITH BEST PARAMETERS
########################################

best_params = study.best_params

training_args = TrainingArguments(
    output_dir="./best_model",
    eval_strategy="epoch",
    learning_rate=best_params["learning_rate"],
    per_device_train_batch_size=best_params["per_device_batch_size"],
    per_device_eval_batch_size=best_params["per_device_batch_size"],
    num_train_epochs=best_params["num_train_epochs"],
    weight_decay=best_params["weight_decay"],
    logging_steps=50,
    save_strategy="epoch",
    report_to=[]
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=test_ds,
    tokenizer=tokenizer
)

trainer.train()

# Save best model
trainer.save_model("./best_model")


########################################
# 11. MAKE PREDICTIONS
########################################

predictions_output = trainer.predict(test_ds)
pred_logits = predictions_output.predictions.argmax(axis=-1)
true_labels = predictions_output.label_ids


########################################
# 12. METRICS
########################################

accuracy = accuracy_score(true_labels, pred_logits)
precision = precision_score(true_labels, pred_logits, zero_division=0)
recall = recall_score(true_labels, pred_logits, zero_division=0)
f1 = f1_score(true_labels, pred_logits, zero_division=0)

print("Accuracy:", round(accuracy, 4))
print("Precision:", round(precision, 4))
print("Recall:", round(recall, 4))
print("F1 Score:", round(f1, 4))
```

```
[I 2025-12-05 22:11:13,009] A new study created in memory with name: no-name-ce85cb9e-f61a-4ae7-b2e8-fd7875e26a6f
/tmp/ipython-input-211700294.py:27: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[2262/2262 04:26, Epoch 1/1]

Epoch	Training Loss	Validation Loss
1	0.410400	0.426339

[566/566 00:18]

```
[I 2025-12-05 22:15:59,138] Trial 0 finished with value: 0.42633941769599915 and parameters: {'learning_rate': 1.7930584988801 05e-06, 'per_device_batch_size': 16, 'weight_decay': 0.12012108171621652, 'num_train_epochs': 1}. Best is tria
/tmp/ipython-input-211700294.py:27: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[2262/2262 08:14, Epoch 2/2]

Epoch	Training Loss	Validation Loss
1	0.401000	0.417861
2	0.447800	0.418053

[283/283 00:17]

```
[I 2025-12-05 22:24:31,879] Trial 1 finished with value: 0.4180528521537781 and parameters: {'learning_rate': 4.640871909706181e-06, 'per_device_batch_size': 32, 'weight_decay': 0.05668881691070582, 'num_train_epochs': 2}. Best is trial
/tmp/ipython-input-211700294.py:27: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[4523/4523 05:19, Epoch 1/1]

Epoch	Training Loss	Validation Loss
1	0.432100	0.440755

[1131/1131 00:19]

```
[I 2025-12-05 22:30:11,683] Trial 2 finished with value: 0.4407546818256378 and parameters: {'learning_rate': 8.142274356486695e-06, 'per_device_batch_size': 8, 'weight_decay': 0.012593103038710663, 'num_train_epochs': 1}. Best is trial
/tmp/ipython-input-211700294.py:27: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[13569/13569 15:56, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	0.447200	0.503041
2	0.475400	0.474761
3	0.491300	0.456200

[1131/1131 00:20]

```
[I 2025-12-05 22:46:29,120] Trial 3 finished with value: 0.4562000632286072 and parameters: {'learning_rate': 2.0634395496200123e-05, 'per_device_batch_size': 8, 'weight_decay': 0.08873669003134965, 'num_train_epochs': 3}. Best is trial
/tmp/ipython-input-211700294.py:27: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

[4523/4523 05:19, Epoch 1/1]

Epoch	Training Loss	Validation Loss
1	0.429000	0.454317

[1131/1131 00:20]

```
[I 2025-12-05 22:52:09,329] Trial 4 finished with value: 0.45431700348854065 and parameters: {'learning_rate': 1.7173454146066268e-06, 'per_device_batch_size': 8, 'weight_decay': 0.10598844334047726, 'num_train_epochs': 1}. Best is tria
/tmp/ipython-input-211700294.py:74: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
Best Trial:
FrozenTrial(number=1, state=<TrialState.COMPLETE: 1>, values=[0.4180528521537781], datetime_start=datetime.datetime(2025, 12, 5, 22, 15, 59, 139221), datetime_complete=datetime.datetime(2025, 12, 5, 22, 24, 31, 879064), params={'learnir
Best Parameters: {'learning_rate': 4.640871909706181e-06, 'per_device_batch_size': 32, 'weight_decay': 0.05668881691070582, 'num_train_epochs': 2}
```

[2262/2262 08:43, Epoch 2/2]

Epoch	Training Loss	Validation Loss
1	0.385300	0.416136
2	0.434100	0.418866

```
Accuracy: 0.8001
Precision: 0.5676
Recall: 0.8131
F1 Score: 0.6685

Classification Report:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-211700294.py in <cell line: 0>()
    110
    111 print("\nClassification Report:")
--> 112 print(classification_report(true_labels, pred_logits, zero_division=0))
    113
    114 print("\nConfusion Matrix:")

NameError: name 'classification_report' is not defined
```

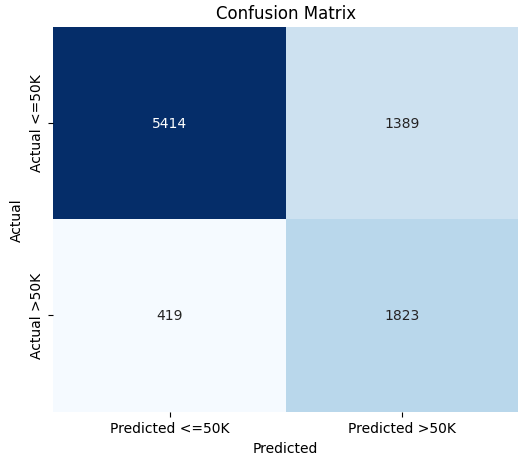
Next steps: Explain error

```
1 from sklearn.metrics import classification_report, confusion_matrix
2 import pandas as pd
3
4 # Classification Report
```

```
print("\nClassification Report:")
print(classification_report(true_labels, pred_logits, zero_division=0))

# Confusion Matrix (numeric)
cm = confusion_matrix(true_labels, pred_logits)
print("\nConfusion Matrix:")
print(cm)

# Pretty Confusion Matrix Table
cm_df = pd.DataFrame(
    cm,
    index=["Actual <=50K", "Actual >50K"],
    columns=["Predicted <=50K", "Predicted >50K"]
)

print("\nConfusion Matrix (Detailed):")
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.80      0.86      6803
           1       0.57      0.81      0.67      2242

    accuracy                           0.80      9045
   macro avg       0.75      0.80      0.76      9045
weighted avg       0.84      0.80      0.81      9045


Confusion Matrix:
[[5414 1389]
 [ 419 1823]]

Confusion Matrix (Detailed):
              Predicted <=50K  Predicted >50K
Actual <=50K             5414            1389
Actual >50K               419            1823
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

cm = confusion_matrix(true_labels, pred_logits)

plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False,
            xticklabels=["Predicted <=50K", "Predicted >50K"],
            yticklabels=["Actual <=50K", "Actual >50K"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```



.