```
# ==============================
# Adult Income (Census) – Colab Setup & Load
# ==============================
```

```
import os, io, sys, textwrap, numpy as np, pandas as pd
from pathlib import Path
```

```
def ensure_files_present():
    """
    Ensures adult.data.csv and adult.test.csv exist in /content.
    If not found, prompts manual upload.
    """
    need_upload = []
    for fn in ["adult.data.csv", "adult.test.csv"]:
        if not Path(f"/content/{fn}").exists():
            need_upload.append(fn)

    if need_upload:
        print("Files not found in /content. Please upload the following files:", need_upload)
        try:
            from google.colab import files
        except Exception:
            print("Not running in Colab? Place the files in the working directory and rerun.")
            return

        uploaded = files.upload()  # user selects files
        for name in uploaded.keys():
            print(f"Uploaded: {name}")

ensure_files_present()

# 5) Column names from UCI Adult dataset
COLUMNS = [
    "age", "workclass", "fnlwgt", "education", "education_num",
    "marital_status", "occupation", "relationship", "race", "sex",
    "capital_gain", "capital_loss", "hours_per_week", "native_country", "income"
]

# 6) Robust loaders (handle ?, spaces, and the dotted labels in test)
def load_adult_train(path="/content/adult.data.csv"):
    return pd.read_csv(
        path, header=None, names=COLUMNS,
        na_values=" ?", skipinitialspace=True
    )

def load_adult_test(path="/content/adult.test.csv"):
    df = pd.read_csv(
        path, header=None, names=COLUMNS,
        na_values=" ?", skipinitialspace=True, skiprows=1  # skip the non-data first line
    )
    # Remove trailing '.' from income labels in test
    if df["income"].dtype == object:
        df["income"] = df["income"].str.replace(".", "", regex=False).str.strip()
    return df

# 7) Load
train_df = load_adult_train()
test_df  = load_adult_test()

# 8) Basic sanity
print("Train shape:", train_df.shape)
print("Test  shape:", test_df.shape)
print("\nTrain dtypes:")
print(train_df.dtypes)

# 9) Quick target distribution
```

```
print("\nTarget value counts (Train):")
print(train_df["income"].value_counts(dropna=False))
print("\nTarget value counts (Test):")
print(test_df["income"].value_counts(dropna=False))

# 10) Missing values overview (after na_values handling)
def missing_summary(df, name):
    na_counts = df.isna().sum().sort_values(ascending=False)
    print(f"\n=== Missing summary: {name} ===")
    print(na_counts[na_counts > 0])

missing_summary(train_df, "train")
missing_summary(test_df, "test")

# 11) Categorical cardinalities
cat_cols = ["workclass","education","marital_status","occupation","relationship","race","sex","native_country","income"]
card = {c: train_df[c].nunique(dropna=True) for c in cat_cols}
print("\nCategorical cardinalities (train):")
for k,v in card.items():
    print(f"{k:16s}: {v}")

# 12) Numeric feature summary
num_cols = ["age","fnlwgt","education_num","capital_gain","capital_loss","hours_per_week"]
print("\nNumeric summary (train):")
display(train_df[num_cols].describe())

# 13) Peek at data
print("\nHead (train):")
display(train_df.head(5))
print("\nHead (test):")
display(test_df.head(5))
```

```
Train shape: (32561, 15)
Test  shape: (16281, 15)

Train dtypes:
age                int64
workclass         object
fnlwgt             int64
education         object
education_num      int64
marital_status    object
occupation        object
relationship      object
race              object
sex               object
capital_gain       int64
capital_loss       int64
hours_per_week     int64
native_country    object
income            object
dtype: object

Target value counts (Train):
income
<=50K    24720
>50K      7841
Name: count, dtype: int64

Target value counts (Test):
income
<=50K    12435
>50K      3846
Name: count, dtype: int64

=== Missing summary: train ===
Series([], dtype: int64)

=== Missing summary: test ===
Series([], dtype: int64)

Categorical cardinalities (train):
workclass       : 9
education       : 16
marital_status  : 7
occupation      : 15
relationship    : 6
race            : 5
sex             : 2
native_country  : 42
income          : 2

Numeric summary (train):
```

	age	fnlwgt	education_num	capital_gain	capital_loss	hours_per_week
count	32561.000000	3.256100e+04	32561.000000	32561.000000	32561.000000	32561.000000
mean	38.581647	1.897784e+05	10.080679	1077.648844	87.303830	40.437456
std	13.640433	1.055500e+05	2.572720	7385.292085	402.960219	12.347429
min	17.000000	1.228500e+04	1.000000	0.000000	0.000000	1.000000
25%	28.000000	1.178270e+05	9.000000	0.000000	0.000000	40.000000
50%	37.000000	1.783560e+05	10.000000	0.000000	0.000000	40.000000
75%	48.000000	2.370510e+05	12.000000	0.000000	0.000000	45.000000
max	90.000000	1.484705e+06	16.000000	99999.000000	4356.000000	99.000000

```
Head (train):
```

	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	cap:
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	

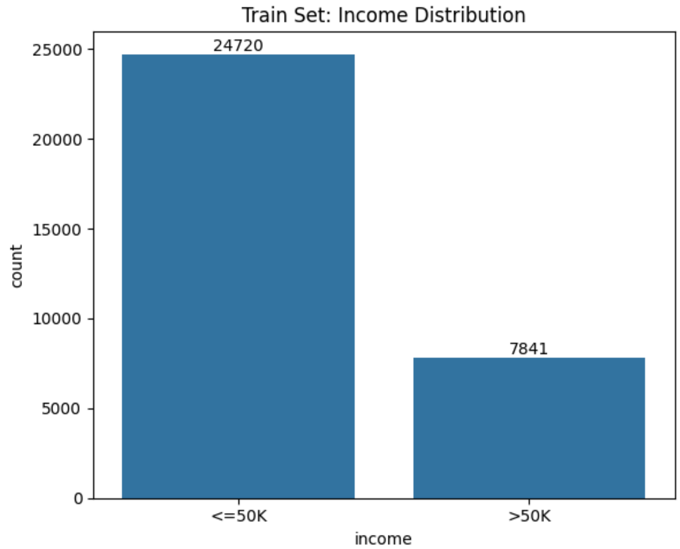
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0

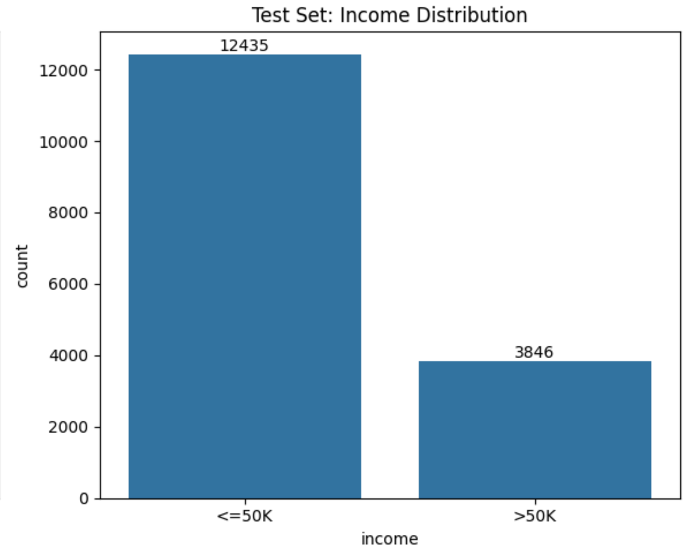
Head (test):

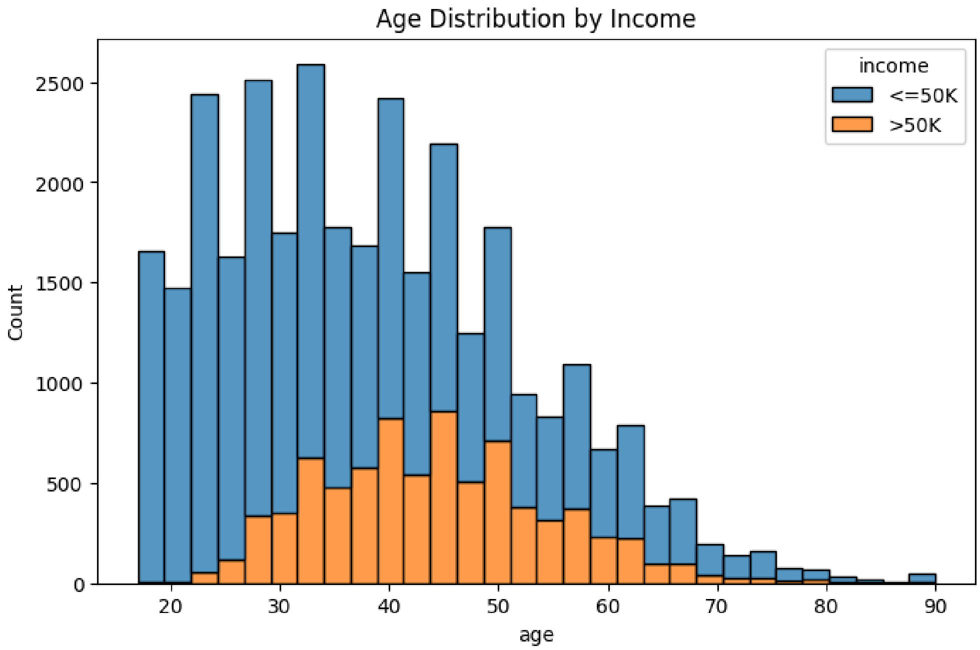
	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	cap:
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	
4	18	?	103497	Some-college	10	Never-married	?	Own-child	White	Female	0	

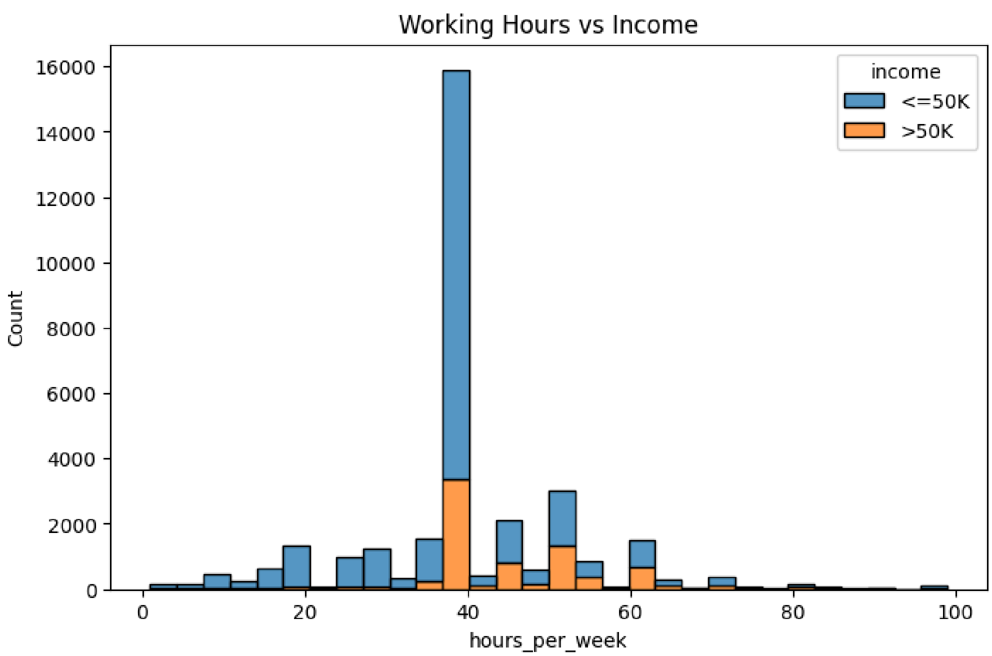
```
# ===============================
# EDA : visuals + numeric-only correlation
# ===============================
import matplotlib.pyplot as plt
import seaborn as sns

# 1) Target Distribution (Train vs Test)
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

sns.countplot(x='income', data=train_df, ax=axes[0])
axes[0].set_title("Train Set: Income Distribution")
axes[0].bar_label(axes[0].containers[0])

sns.countplot(x='income', data=test_df, ax=axes[1])
axes[1].set_title("Test Set: Income Distribution")
axes[1].bar_label(axes[1].containers[0])

plt.tight_layout()
plt.show()

# 2) Age Distribution vs Income
plt.figure(figsize=(8, 5))
sns.histplot(data=train_df, x="age", hue="income", bins=30, kde=False, multiple="stack")
plt.title("Age Distribution by Income")
plt.show()

# 3) Hours-per-week vs Income
plt.figure(figsize=(8, 5))
sns.histplot(data=train_df, x="hours_per_week", hue="income", bins=30, kde=False, multiple="stack")
plt.title("Working Hours vs Income")
plt.show()

# 4) Categorical Features vs Income (Top 5 Features)
cat_features = ["workclass", "education", "marital_status", "occupation", "sex"]

fig, axes = plt.subplots(3, 2, figsize=(15, 15))
axes = axes.flatten()

for i, col in enumerate(cat_features):
    sns.countplot(x=col, hue="income", data=train_df, ax=axes[i])
    axes[i].set_title(f"Income by {col}")
    axes[i].tick_params(axis='x', rotation=45)

# remove empty subplot
fig.delaxes(axes[-1])
```

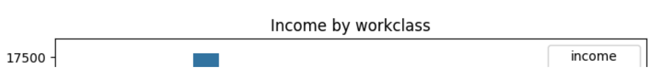
```
plt.tight_layout()
plt.show()

# 5) Correlation Heatmap (NUMERIC FEATURES ONLY)
num_cols = ["age","fnlwgt","education_num","capital_gain","capital_loss","hours_per_week"]
corr = train_df[num_cols].corr(numeric_only=True)

plt.figure(figsize=(8,6))
sns.heatmap(corr, annot=True, cmap="coolwarm", fmt=".2f", square=True)
plt.title("Correlation Heatmap (Numeric Features)")
plt.show()

# (Optional) Quick sanity: average income rate by marital_status (sorted)
rate = (
    train_df
      .assign(y=(train_df["income"]==">50K").astype(int))
      .groupby("marital_status")["y"]
      .mean()
      .sort_values(ascending=False)
)
print("\nShare of >50K by marital_status (train):")
print(rate)
```

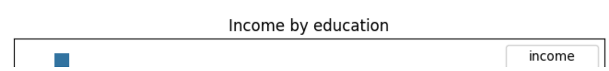

Train Set: Income Distribution
24720
7841
count
0
5000
10000
15000
20000
25000
<=50K
>50K
income

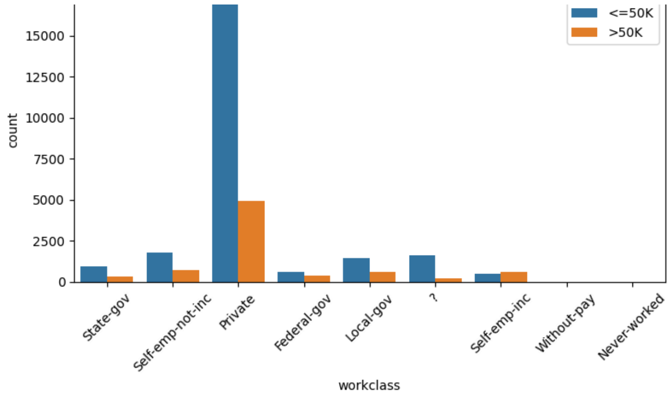

Test Set: Income Distribution
12435
3846
count
0
2000
4000
6000
8000
10000
12000
<=50K
>50K
income

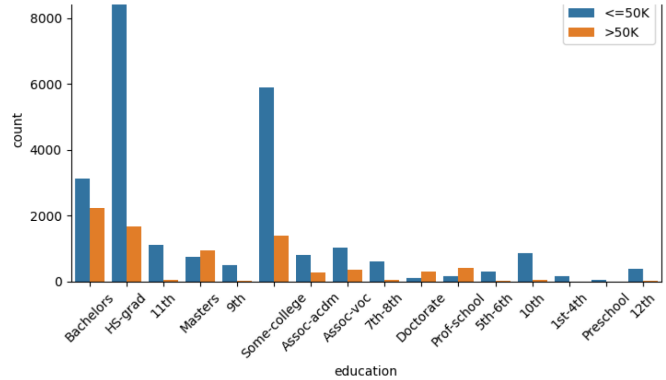

Age Distribution by Income
income
<=50K
>50K
Count
0
500
1000
1500
2000
2500
20
30
40
50
60
70
80
90
age

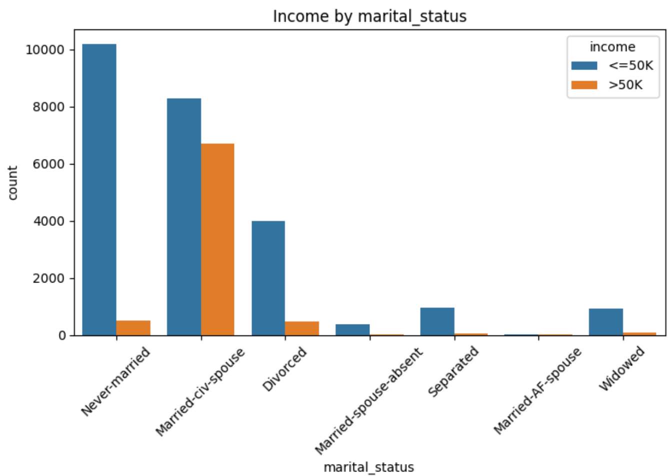

Working Hours vs Income
income
<=50K
>50K
Count
0
2000
4000
6000
8000
10000
12000
14000
16000
0
20
40
60
80
100
hours_per_week

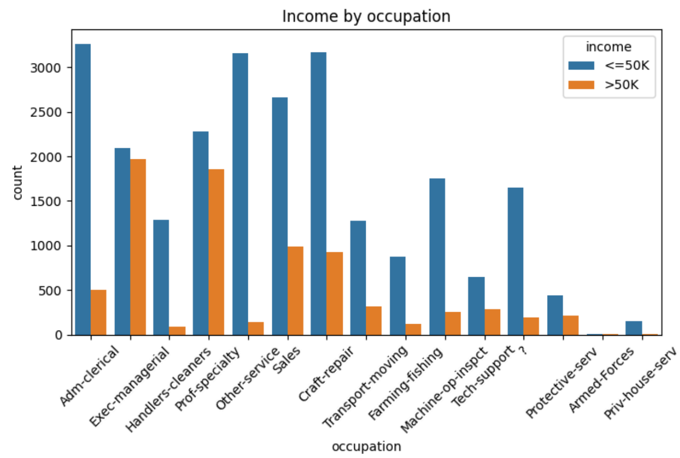

Income by workclass
17500
income

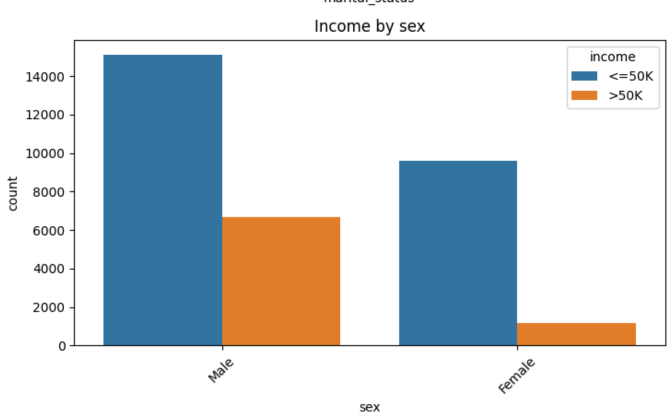

Income by education
income

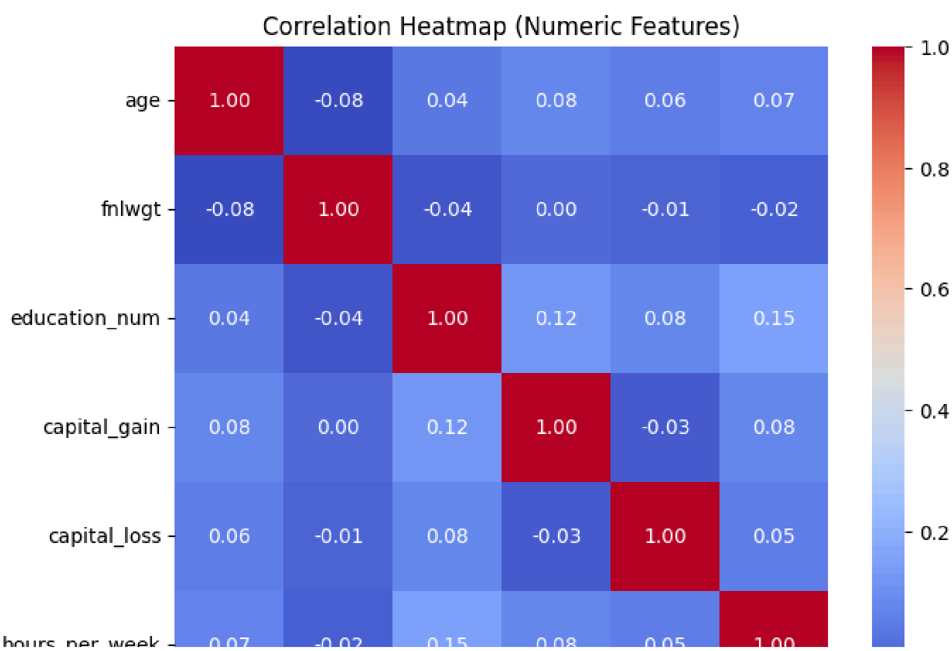

count
<=50K
>50K
State-gov
Self-emp-not-inc
Private
Federal-gov
Local-gov
?
Self-emp-inc
Without-pay
Never-worked
workclass

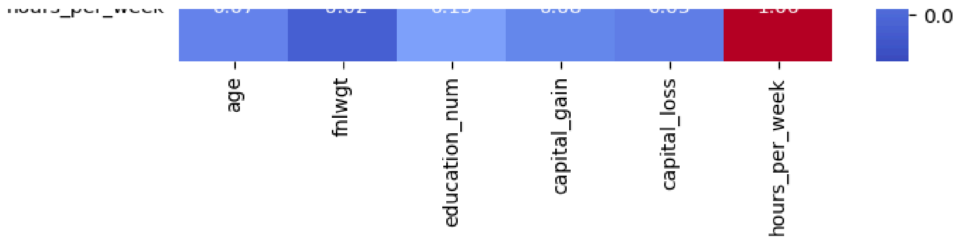

count
<=50K
>50K
Bachelors
HS-grad
11th
Masters
9th
Some-college
Assoc-acdm
Assoc-voc
7th-8th
Doctorate
Prof-school
5th-6th
10th
1st-4th
Preschool
12th
education


Income by marital_status
count
income
<=50K
>50K
Never-married
Married-civ-spouse
Divorced
Married-spouse-absent
Separated
Married-AF-spouse
Widowed
marital_status


Income by occupation
count
income
<=50K
>50K
Adm-clerical
Exec-managerial
Handlers-cleaners
Prof-specialty
Other-service
Sales
Craft-repair
Transport-moving
Farming-fishing
Machine-op-inspct
Tech-support
?
Protective-serv
Armed-Forces
Priv-house-serv
occupation


Income by sex
count
income
<=50K
>50K
Male
Female
sex


Correlation Heatmap (Numeric Features)
age 1.00 -0.08 0.04 0.08 0.06 0.07
fnlwgt -0.08 1.00 -0.04 0.00 -0.01 -0.02
education_num 0.04 -0.04 1.00 0.12 0.08 0.15
capital_gain 0.08 0.00 0.12 1.00 -0.03 0.08
capital_loss 0.06 -0.01 0.08 -0.03 1.00 0.05
hours_per_week 0.07 -0.02 0.15 0.08 0.05 1.00




```
Share of >50K by marital_status (train):
marital_status
Married-civ-spouse       0.446848
Married-AF-spouse        0.434783
Divorced                 0.104209
Widowed                  0.085599
Married-spouse-absent    0.081340
Separated                0.064390
Never-married            0.045961
Name: y, dtype: float64
```

EDA Insights and Interpretation — Adult Income Prediction

1. Target Distribution

Train Set ⟶

<=50K: 24,720 (~76%)

> 50K: 7,841 (~24%)

Test Set ⟶

<=50K: 12,435 (~76%)

> 50K: 3,846 (~24%)

Interpretation: The dataset is highly imbalanced. Most individuals earn ≤50K. We will need to address this imbalance using:

SMOTE oversampling, or

class_weight="balanced" during model training.

2. Age Distribution vs Income

Individuals earning >50K are typically between 30 and 55 years old.

Most ≤50K earners are below 35.

Income probability increases sharply after age 30 and peaks between 35–50.

Implication for Modeling: age is a strong predictor for income classification.

3. Working Hours vs Income

Majority work around 40 hours/week (normal full-time).

Higher income (>50K) earners:

Often work 50–60+ hours/week.

Very few high earners work less than 35 hours/week.

Insight: hours_per_week is an important predictor. Higher weekly hours generally correlate positively with higher income.

4. Workclass vs Income

Most individuals work in the Private sector.

Higher income is more common in:

Federal-gov

Self-employed (incorporated)

State-gov roles.

Lower income is common among:

Without-pay and Never-worked categories.

Individuals with missing (?) workclass data.

Implication: workclass affects income predictions significantly.

5. Education vs Income

Higher education levels lead to higher income:

Doctorate and Prof-school ⟶ highest percentage of >50K.

Masters and Bachelors ⟶ strong contributors.

Lower income groups dominate:

HS-grad, Some-college, 11th grade, and below.

Insight: education_num (numeric) and education (categorical) are critical features.

6. Marital Status vs Income

Married-civ-spouse → nearly 45% earn >50K.

Never-married → only 4.5% earn >50K.

Divorced and widowed individuals fall in between.

Conclusion: marital_status has a strong relationship with income level.

7. Occupation vs Income

Higher-income occupations include:

Exec-managerial

Prof-specialty

Tech-support

Lower-income occupations include:

Handlers-cleaners

Other-service

Priv-house-serv

Takeaway: occupation is an influential categorical feature and should be properly encoded.

8. Gender (Sex) vs Income

Males dominate the >50K category.

Females are mostly in the <=50K category.

Gender bias is evident and should be considered when analyzing fairness.

9. Correlation Heatmap (Numeric Features) Feature Correlation with Income education_num +0.34 (moderate) hours_per_week +0.15 (weak) capital_gain +0.12 (weak) age +0.08 (weak)

capital_loss +0.05 (very weak) fnlwgt ~0 (negligible)

Insight:

education_num is the most informative numeric predictor.

fnlwgt has almost no impact → we can consider dropping it.

10. Key Findings

Dataset is imbalanced → need oversampling or class-weight adjustment.

Strong predictors:

education_num, education

age

hours_per_week

marital_status

occupation

Features to handle carefully:

native_country → group rare categories.

workclass → clean missing values (? → "Unknown").

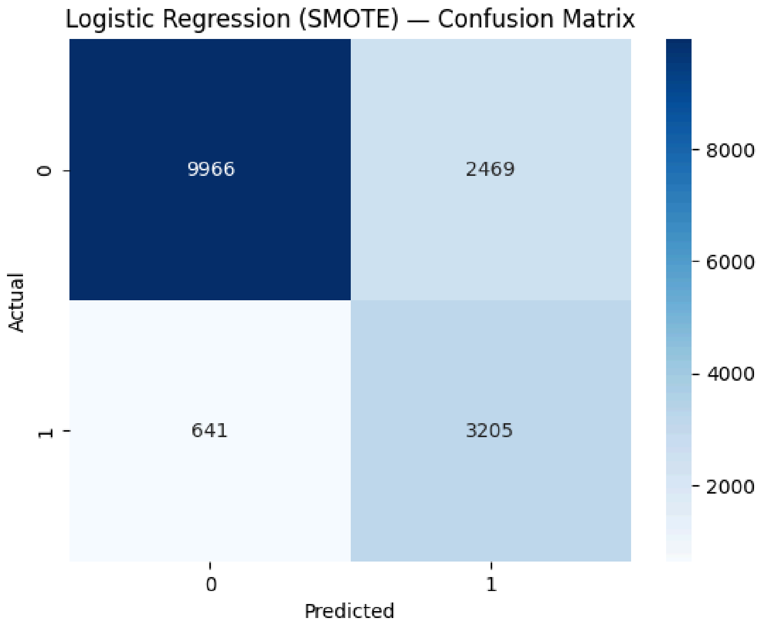
Potentially drop or deprioritize fnlwgt due to low correlation.

```
# ===============================
# Adult Income - Models: LR(SMOTE), RF, XGBoost
# ===============================

# 0) Install xgboost (Colab)
!pip -q install xgboost

# 1) Imports
import os, joblib, numpy as np
```

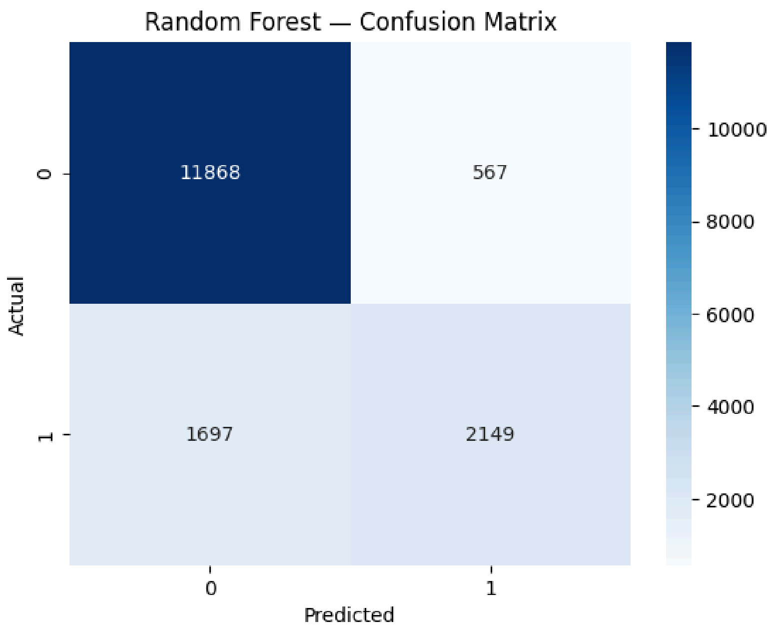
```
import seaborn as sns, matplotlib.pyplot as plt

from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline as SkPipeline
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

from sklearn.metrics import classification_report, confusion_matrix
from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE

# 2) Ensure X_train / y_train / X_test / y_test exist (fallback build if needed)
try:
    X_train, y_train, X_test, y_test
except NameError:
    assert "train_df" in globals() and "test_df" in globals(), "Load train_df and test_df first."

    X_train = train_df.drop("income", axis=1).copy()
    y_train = (train_df["income"] == ">50K").astype(int).copy()
    X_test  = test_df.drop("income", axis=1).copy()
    y_test  = (test_df["income"] == ">50K").astype(int).copy()

# 3) Feature groups
numeric_features = ["age","fnlwgt","education_num","capital_gain","capital_loss","hours_per_week"]
categorical_features = ["workclass","education","marital_status","occupation",
                        "relationship","race","sex","native_country"]

# 4) Preprocessor
numeric_transformer = SkPipeline(steps=[
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler())
])

categorical_transformer = SkPipeline(steps=[
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown="ignore", sparse_output=False))
])

preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, numeric_features),
        ("cat", categorical_transformer, categorical_features),
    ]
)

# 5) Pipelines
# Logistic Regression WITH SMOTE
pipe_lr = ImbPipeline(steps=[
    ("preprocessor", preprocessor),
    ("smote", SMOTE(random_state=42)),
    ("classifier", LogisticRegression(max_iter=1000, class_weight="balanced",
                                      solver="lbfgs", n_jobs=-1))
])

# Random Forest (no SMOTE)
pipe_rf = SkPipeline(steps=[
    ("preprocessor", preprocessor),
    ("classifier", RandomForestClassifier(
        n_estimators=200, max_depth=12, random_state=42, n_jobs=-1
    ))
])

# XGBoost (handle imbalance via scale_pos_weight; no SMOTE)
pos_weight = float((len(y_train) - y_train.sum()) / y_train.sum())
pipe_xgb = SkPipeline(steps=[
    ("preprocessor", preprocessor),
    ("classifier", XGBClassifier(
        objective="binary:logistic",
        eval_metric="logloss",
        n_estimators=300,
```

```
        max_depth=6,
        learning_rate=0.1,
        subsample=0.8,
        colsample_bytree=0.8,
        reg_lambda=1.0,
        scale_pos_weight=pos_weight,
        n_jobs=-1,
        random_state=42
    ))
])

# 6) Train
print("Training Logistic Regression (SMOTE)...")
pipe_lr.fit(X_train, y_train)

print("Training Random Forest...")
pipe_rf.fit(X_train, y_train)

print("Training XGBoost...")
pipe_xgb.fit(X_train, y_train)

# 7) Evaluate helper
def evaluate_model(name, model, X_test, y_test):
    y_pred = model.predict(X_test)
    print(f"\n{name} — Classification Report")
    print(classification_report(y_test, y_pred))
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
    plt.title(f"{name} — Confusion Matrix")
    plt.xlabel("Predicted"); plt.ylabel("Actual")
    plt.show()

# 8) Evaluate all
evaluate_model("Logistic Regression (SMOTE)", pipe_lr, X_test, y_test)
evaluate_model("Random Forest",               pipe_rf, X_test, y_test)
evaluate_model("XGBoost",                     pipe_xgb, X_test, y_test)

# 9) Save models
os.makedirs("models", exist_ok=True)
joblib.dump(pipe_lr, "models/logistic_regression_smote.pkl")
joblib.dump(pipe_rf, "models/random_forest.pkl")
joblib.dump(pipe_xgb, "models/xgboost.pkl")
print(" All models saved under /content/models/")
```

```
Training Logistic Regression (SMOTE)...
Training Random Forest...
Training XGBoost...

Logistic Regression (SMOTE) — Classification Report
              precision    recall  f1-score   support

           0       0.94      0.80      0.87     12435
           1       0.56      0.83      0.67      3846

    accuracy                           0.81     16281
   macro avg       0.75      0.82      0.77     16281
weighted avg       0.85      0.81      0.82     16281
```

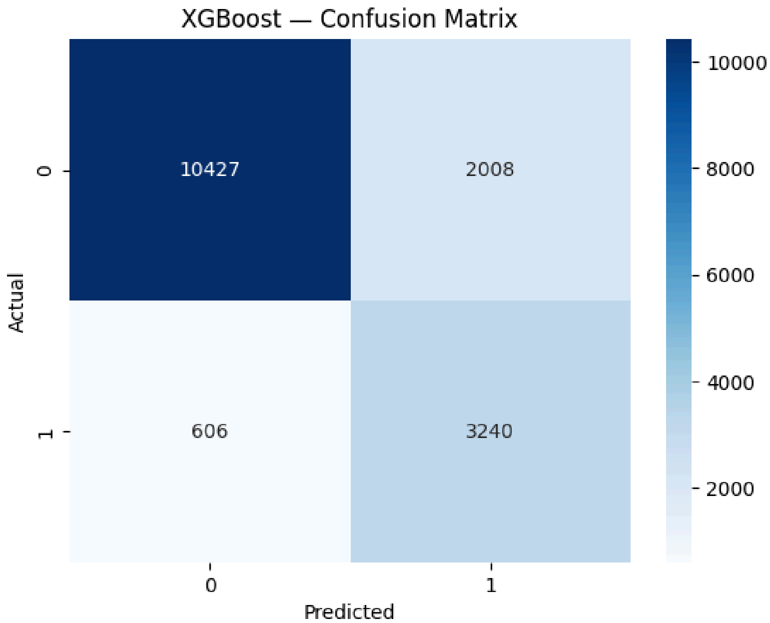



```
Random Forest — Classification Report
              precision    recall  f1-score   support

           0       0.87      0.95      0.91     12435
           1       0.79      0.56      0.65      3846

    accuracy                           0.86     16281
   macro avg       0.83      0.76      0.78     16281
weighted avg       0.86      0.86      0.85     16281
```




```
XGBoost — Classification Report
```

```
              precision    recall  f1-score   support

           0       0.95      0.84      0.89     12435
           1       0.62      0.84      0.71      3846

    accuracy                           0.84     16281
   macro avg       0.78      0.84      0.80     16281
weighted avg       0.87      0.84      0.85     16281
```




```
All models saved under /content/models/
```

Key Insights A. Logistic Regression (SMOTE)

High recall → catches most high-income individuals.

Precision is low → many false positives.

Best if minimizing missed positive cases is critical.

B. Random Forest

Best overall accuracy and precision.

However, misses many positives (low recall).

Best for conservative predictions.

C. XGBoost

More balanced between precision and recall.

Lower accuracy than RF but best F1-score.

Handles class imbalance well (scale_pos_weight worked great).

```
# ===============================
# Balanced 500/500 Search → Refit on Full: LR(SMOTE), RF, XGBoost (GPU)
# ===============================

!pip -q install xgboost

import os, numpy as np, seaborn as sns, matplotlib.pyplot as plt, joblib
from sklearn.metrics import classification_report, confusion_matrix, f1_score, make_scorer
from sklearn.model_selection import StratifiedKFold, GridSearchCV, RandomizedSearchCV, cross_val_predict
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline as SkPipeline
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
```

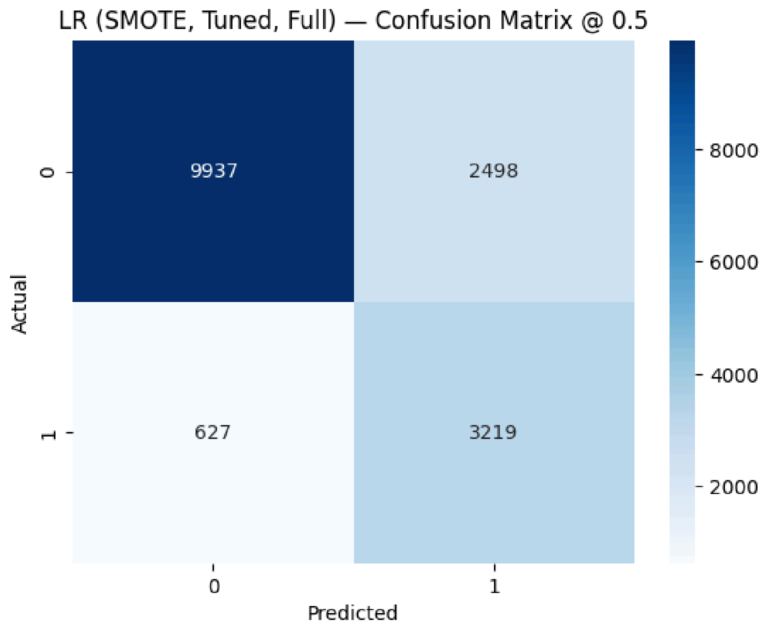
```
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE

# ---------- Ensure data present ----------
try:
    X_train, y_train, X_test, y_test
except NameError:
    assert "train_df" in globals() and "test_df" in globals(), "Load train_df and test_df first."
    X_train = train_df.drop("income", axis=1).copy()
    y_train = (train_df["income"] == ">50K").astype(int).copy()
    X_test  = test_df.drop("income", axis=1).copy()
    y_test  = (test_df["income"] == ">50K").astype(int).copy()

# ---------- Build a balanced search subset: 500 negatives & 500 positives ----------
def balanced_subset(X, y, n_per_class=500, random_state=42):
    import pandas as pd
    df = X.copy()
    df["_y_"] = y.values
    # split by class
    neg = df[df["_y_"] == 0]
    pos = df[df["_y_"] == 1]
    n_neg = min(n_per_class, len(neg))
    n_pos = min(n_per_class, len(pos))
    neg_s = neg.sample(n=n_neg, random_state=random_state)
    pos_s = pos.sample(n=n_pos, random_state=random_state)
    sub = pd.concat([neg_s, pos_s], axis=0).sample(frac=1.0, random_state=random_state)  # shuffle
    y_sub = sub.pop("_y_")
    return sub, y_sub

X_search, y_search = balanced_subset(X_train, y_train, n_per_class=500, random_state=42)
print(f"Search subset size: {len(X_search)} (neg={ (y_search==0).sum() }, pos={ (y_search==1).sum() })")

# ---------- (Optional) drop 'fnlwgt' for speed — has near-zero correlation ----------
drop_fnlwgt = True
numeric_features = ["age","education_num","capital_gain","capital_loss","hours_per_week"] if drop_fnlwgt else \
                   ["age","fnlwgt","education_num","capital_gain","capital_loss","hours_per_week"]
categorical_features = ["workclass","education","marital_status","occupation",
                        "relationship","race","sex","native_country"]

# ---------- Preprocessor ----------
numeric_transformer = SkPipeline(steps=[
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler())
])
categorical_transformer = SkPipeline(steps=[
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown="ignore", sparse_output=False))
])
preprocessor = ColumnTransformer(
    transformers=[("num", numeric_transformer, numeric_features),
                  ("cat", categorical_transformer, categorical_features)]
)

# ---------- CV + scorer ----------
cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)  # 3-fold for speed
f1_pos = make_scorer(f1_score, pos_label=1)

def evaluate_with_thresholds(name, model, X_test, y_test, tuned_threshold=None):
    # Default 0.5
    y_pred_05 = model.predict(X_test)
    print(f"\n{name} — Test @ 0.5")
    print(classification_report(y_test, y_pred_05))
    cm = confusion_matrix(y_test, y_pred_05)
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
    plt.title(f"{name} — Confusion Matrix @ 0.5"); plt.xlabel("Predicted"); plt.ylabel("Actual")
    plt.show()
    # Tuned threshold
    if tuned_threshold is not None:
        try:
            probs = model.predict_proba(X_test)[:, 1]
        except Exception:
```

```
            scores = model.decision_function(X_test)
            probs = 1 / (1 + np.exp(-scores))
        y_pred_tuned = (probs >= tuned_threshold).astype(int)
        print(f"\n{name} — Test @ tuned threshold={tuned_threshold:.3f}")
        print(classification_report(y_test, y_pred_tuned))
        cm = confusion_matrix(y_test, y_pred_tuned)
        sns.heatmap(cm, annot=True, fmt="d", cmap="Greens")
        plt.title(f"{name} — Confusion Matrix @ tuned={tuned_threshold:.3f}")
        plt.xlabel("Predicted"); plt.ylabel("Actual")
        plt.show()

def choose_threshold_via_cv(estimator, X, y, cv, lo=0.3, hi=0.7, steps=9):
    # Narrow sweep for speed
    try:
        cv_scores = cross_val_predict(estimator, X, y, cv=cv, method="decision_function", n_jobs=-1)
        probs_like = 1 / (1 + np.exp(-cv_scores))
    except Exception:
        probs_like = cross_val_predict(estimator, X, y, cv=cv, method="predict_proba", n_jobs=-1)[:, 1]
    thresholds = np.linspace(lo, hi, steps)
    f1s = [(t, f1_score(y, (probs_like >= t).astype(int))) for t in thresholds]
    best_t, best_f1 = max(f1s, key=lambda x: x[1])
    return best_t, best_f1

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

# ==========================================================
# 1) Logistic Regression (SMOTE) — Search on BALANCED subset ➜ Refit on FULL
# ==========================================================
pipe_lr_search = ImbPipeline(steps=[
    ("preprocessor", preprocessor),
    ("smote", SMOTE(random_state=42)),  # harmless on balanced set; keeps pipeline consistent
    ("classifier", LogisticRegression(max_iter=1500, n_jobs=-1))
])

param_grid_lr = [
    {"classifier__solver": ["lbfgs"], "classifier__penalty": ["l2"],
     "classifier__C": [0.1, 0.5, 1.0, 2.0]},
    {"classifier__solver": ["liblinear"], "classifier__penalty": ["l1","l2"],
     "classifier__C": [0.1, 0.5, 1.0, 2.0]},
    {"classifier__solver": ["saga"], "classifier__penalty": ["l1","l2","elasticnet"],
     "classifier__l1_ratio": [0.0, 0.5], "classifier__C": [0.1, 0.5, 1.0]},
]

gs_lr = GridSearchCV(
    estimator=pipe_lr_search, param_grid=param_grid_lr, cv=cv,
    scoring={"f1_pos": f1_pos, "roc_auc": "roc_auc"}, refit="f1_pos",
    n_jobs=-1, verbose=1
)
print("Searching LR (SMOTE) on balanced subset...")
gs_lr.fit(X_search, y_search)
print("Best LR (subset) params:", gs_lr.best_params_, " | Best CV F1:", gs_lr.best_score_)

# Rebuild LR with best params and FIT on FULL
pipe_lr_full = ImbPipeline(steps=[
    ("preprocessor", preprocessor),
    ("smote", SMOTE(random_state=42)),
    ("classifier", LogisticRegression(max_iter=1500, n_jobs=-1))
]).set_params(**gs_lr.best_params_)

print("Refitting LR best config on FULL data...")
pipe_lr_full.fit(X_train, y_train)

best_t_lr, best_f1_lr = choose_threshold_via_cv(pipe_lr_full, X_train, y_train, cv)
print(f"Chosen LR threshold via CV (full): {best_t_lr:.3f} (CV F1={best_f1_lr:.3f})")
evaluate_with_thresholds("LR (SMOTE, Tuned, Full)", pipe_lr_full, X_test, y_test, tuned_threshold=best_t_lr)
joblib.dump(pipe_lr_full, "models/logistic_regression_smote_tuned.pkl")


# ==========================================================
# 2) Random Forest — Search on BALANCED subset ➜ Refit on FULL
# ==========================================================
pipe_rf_search = SkPipeline(steps=[
    ("preprocessor", preprocessor),
```

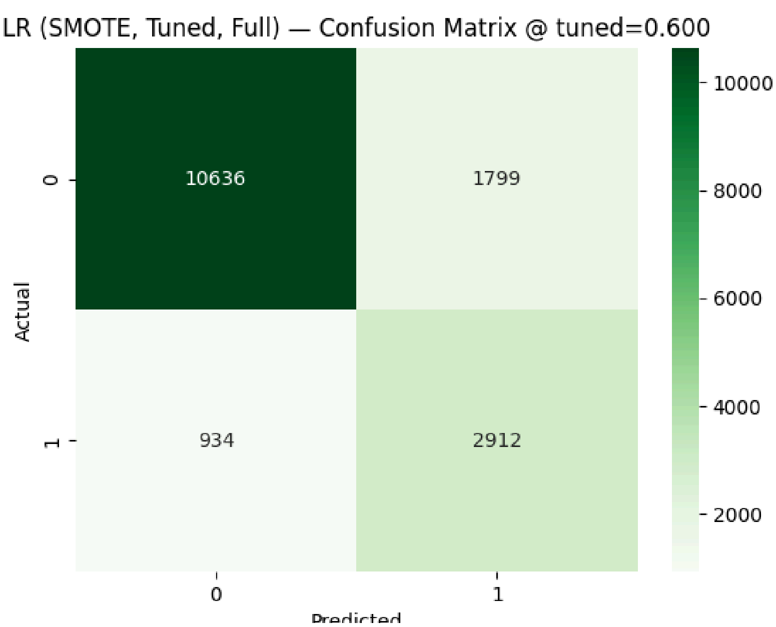
```
        ("classifier", RandomForestClassifier(random_state=42, n_jobs=-1))
    ])

    param_dist_rf = {
        "classifier__n_estimators": [200, 300, 400],
        "classifier__max_depth": [None, 12, 16],
        "classifier__min_samples_split": [2, 5, 10],
        "classifier__min_samples_leaf": [1, 2, 3],
        "classifier__max_features": ["sqrt", 0.5],
        "classifier__class_weight": [None, "balanced"],
    }
    rs_rf = RandomizedSearchCV(
        estimator=pipe_rf_search, param_distributions=param_dist_rf, n_iter=12,
        cv=cv, scoring={"f1_pos": f1_pos, "roc_auc": "roc_auc"}, refit="f1_pos",
        n_jobs=-1, verbose=1, random_state=42
    )
    print("\nSearching Random Forest on balanced subset...")
    rs_rf.fit(X_search, y_search)
    print("Best RF (subset) params:", rs_rf.best_params_, " | Best CV F1:", rs_rf.best_score_)

    pipe_rf_full = SkPipeline(steps=[
        ("preprocessor", preprocessor),
        ("classifier", RandomForestClassifier(random_state=42, n_jobs=-1))
    ]).set_params(**rs_rf.best_params_)

    print("Refitting RF best config on FULL data...")
    pipe_rf_full.fit(X_train, y_train)

    best_t_rf, best_f1_rf = choose_threshold_via_cv(pipe_rf_full, X_train, y_train, cv)
    print(f"Chosen RF threshold via CV (full): {best_t_rf:.3f} (CV F1={best_f1_rf:.3f})")
    evaluate_with_thresholds("Random Forest (Tuned, Full)", pipe_rf_full, X_test, y_test, tuned_threshold=best_t_rf)
    joblib.dump(pipe_rf_full, "models/random_forest_tuned.pkl")


    # =========================================================
    # 3) XGBoost (GPU) — Search on BALANCED subset → Refit on FULL
    # =========================================================
    pos_weight = float((len(y_train) - y_train.sum()) / y_train.sum())  # keep class prior for full refit

    pipe_xgb_search = SkPipeline(steps=[
        ("preprocessor", preprocessor),
        ("classifier", XGBClassifier(
            objective="binary:logistic",
            eval_metric="logloss",
            n_jobs=-1,
            random_state=42,
            scale_pos_weight=1.0,        # subset is balanced; don't distort during SEARCH
            tree_method="gpu_hist",
            predictor="gpu_predictor",
        ))
    ])

    param_dist_xgb = {
        "classifier__n_estimators": [200, 300, 400],
        "classifier__max_depth": [4, 6, 8],
        "classifier__learning_rate": [0.05, 0.1],
        "classifier__subsample": [0.8, 1.0],
        "classifier__colsample_bytree": [0.7, 0.9],
        "classifier__min_child_weight": [1, 5],
        "classifier__reg_lambda": [1.0, 2.0],
    }
    rs_xgb = RandomizedSearchCV(
        estimator=pipe_xgb_search, param_distributions=param_dist_xgb, n_iter=12,
        cv=cv, scoring={"f1_pos": f1_pos, "roc_auc": "roc_auc"}, refit="f1_pos",
        n_jobs=-1, verbose=1, random_state=42
    )
    print("\nSearching XGBoost (GPU) on balanced subset...")
    rs_xgb.fit(X_search, y_search)
    print("Best XGB (subset) params:", rs_xgb.best_params_, " | Best CV F1:", rs_xgb.best_score_)

    # Rebuild for FULL with original class prior (pos_weight) and FIT
    pipe_xgb_full = SkPipeline(steps=[
        ("preprocessor", preprocessor),
```

```
        ("classifier", XGBClassifier(
            objective="binary:logistic",
            eval_metric="logloss",
            n_jobs=-1,
            random_state=42,
            scale_pos_weight=pos_weight,  # <-- use real prior on full data
            tree_method="gpu_hist",
            predictor="gpu_predictor",
        ))
    ]).set_params(**rs_xgb.best_params_)

    print("Refitting XGB best config on FULL data...")
    pipe_xgb_full.fit(X_train, y_train)

    best_t_xgb, best_f1_xgb = choose_threshold_via_cv(pipe_xgb_full, X_train, y_train, cv)
    print(f"Chosen XGB threshold via CV (full): {best_t_xgb:.3f} (CV F1={best_f1_xgb:.3f})")
    evaluate_with_thresholds("XGBoost (Tuned, GPU, Full)", pipe_xgb_full, X_test, y_test, tuned_threshold=best_t_xgb)
    joblib.dump(pipe_xgb_full, "models/xgboost_tuned.pkl")


    print("\n Saved final (refit-on-full) models to:")
    print(" - models/logistic_regression_smote_tuned.pkl")
    print(" - models/random_forest_tuned.pkl")
    print(" - models/xgboost_tuned.pkl")
```

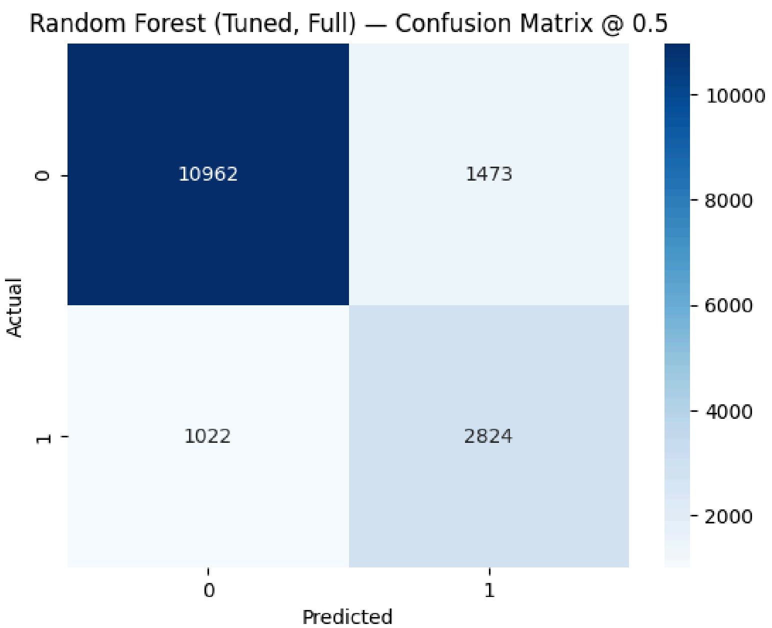
```
Search subset size: 1000 (neg=500, pos=500)
Searching LR (SMOTE) on balanced subset...
Fitting 3 folds for each of 30 candidates, totalling 90 fits
Best LR (subset) params: {'classifier__C': 0.1, 'classifier__l1_ratio': 0.5, 'classifier__penalty': 'elasticnet', 'classifier_
Refitting LR best config on FULL data...
Chosen LR threshold via CV (full): 0.600 (CV F1=0.689)

LR (SMOTE, Tuned, Full) — Test @ 0.5
              precision    recall  f1-score   support

           0       0.94      0.80      0.86     12435
           1       0.56      0.84      0.67      3846

    accuracy                           0.81     16281
   macro avg       0.75      0.82      0.77     16281
weighted avg       0.85      0.81      0.82     16281
```



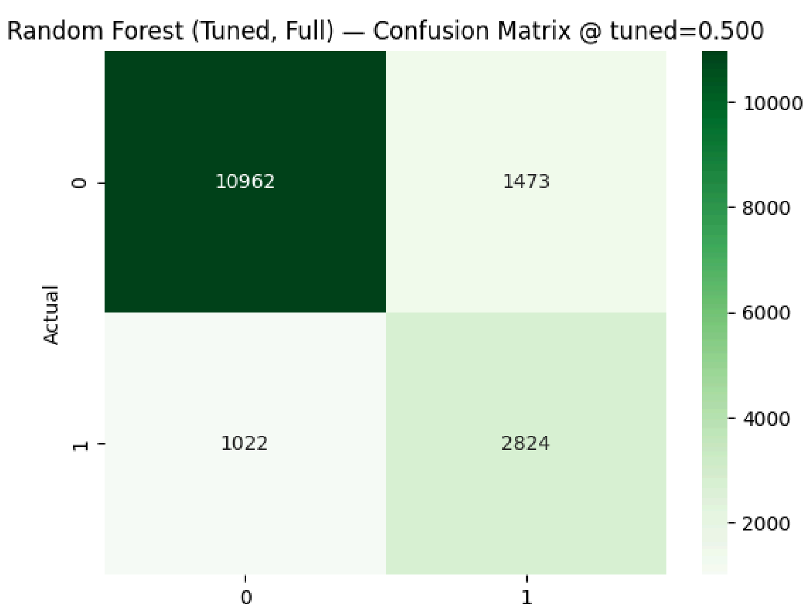

```
LR (SMOTE, Tuned, Full) — Test @ tuned threshold=0.600
              precision    recall  f1-score   support

           0       0.92      0.86      0.89     12435
           1       0.62      0.76      0.68      3846

    accuracy                           0.83     16281
   macro avg       0.77      0.81      0.78     16281
weighted avg       0.85      0.83      0.84     16281
```



```
Searching Random Forest on balanced subset...
Fitting 3 folds for each of 12 candidates, totalling 36 fits
Best RF (subset) params: {'classifier__n_estimators': 300, 'classifier__min_samples_split': 10, 'classifier__min_samples_leaf'
Refitting RF best config on FULL data...
Chosen RF threshold via CV (full): 0.500 (CV F1=0.707)

Random Forest (Tuned, Full) — Test @ 0.5
              precision    recall  f1-score   support

           0       0.91      0.88      0.90     12435
           1       0.66      0.73      0.69      3846

    accuracy                           0.85     16281
   macro avg       0.79      0.81      0.80     16281
weighted avg       0.85      0.85      0.85     16281
```

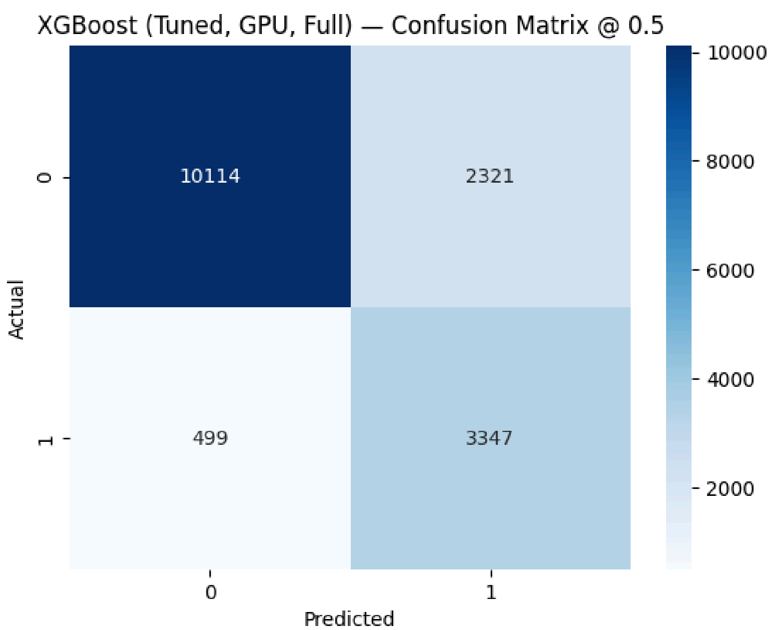



```
Random Forest (Tuned, Full) — Test @ tuned threshold=0.500
              precision    recall  f1-score   support

           0       0.91      0.88      0.90     12435
           1       0.66      0.73      0.69      3846

    accuracy                           0.85     16281
   macro avg       0.79      0.81      0.80     16281
weighted avg       0.85      0.85      0.85     16281
```



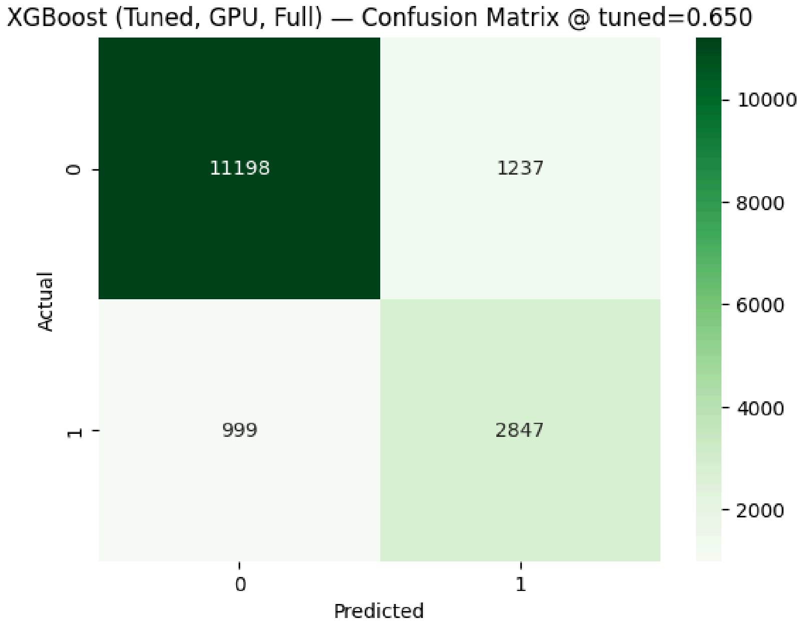
Predicted

```
Searching XGBoost (GPU) on balanced subset...
Fitting 3 folds for each of 12 candidates, totalling 36 fits
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [23:54:58] WARNING: /workspace/src/common/error_

    E.g. tree_method = "hist", device = "cuda"

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [23:54:58] WARNING: /workspace/src/learner.cc:73
Parameters: { "predictor" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
Best XGB (subset) params: {'classifier__subsample': 1.0, 'classifier__reg_lambda': 1.0, 'classifier__n_estimators': 200, 'clas
Refitting XGB best config on FULL data...
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [23:54:58] WARNING: /workspace/src/common/error_

    E.g. tree_method = "hist", device = "cuda"

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [23:54:58] WARNING: /workspace/src/learner.cc:73
Parameters: { "predictor" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
Chosen XGB threshold via CV (full): 0.650 (CV F1=0.724)

XGBoost (Tuned, GPU, Full) — Test @ 0.5
              precision    recall  f1-score   support

           0       0.95      0.81      0.88     12435
           1       0.59      0.87      0.70      3846

    accuracy                           0.83     16281
   macro avg       0.77      0.84      0.79     16281
weighted avg       0.87      0.83      0.84     16281

/usr/local/lib/python3.12/dist-packages/xgboost/core.py:2676: UserWarning: [23:55:04] WARNING: /workspace/src/common/error_msg

    E.g. tree_method = "hist", device = "cuda"

  if len(data.shape) != 1 and self.num_features() != data.shape[1]:
/usr/local/lib/python3.12/dist-packages/xgboost/core.py:729: UserWarning: [23:55:04] WARNING: /workspace/src/common/error_msg.
Potential solutions:
- Use a data structure that matches the device ordinal in the booster.
- Set the device for booster before call to inplace_predict.

This warning will only be shown once.

  return func(**kwargs)
```




```
XGBoost (Tuned, GPU, Full) — Test @ tuned threshold=0.650
              precision    recall  f1-score   support
```

```
           0       0.92      0.90      0.91     12435
           1       0.70      0.74      0.72      3846

    accuracy                           0.86     16281
   macro avg       0.81      0.82      0.81     16281
weighted avg       0.87      0.86      0.86     16281
```




```
Saved final (refit-on-full) models to:
- models/logistic_regression_smote_tuned.pkl
- models/random_forest_tuned.pkl
- models/xgboost_tuned.pkl
```

```
# ===============================
# SHAP Explainability — LR(SMOTE), RF, XGBoost (tuned models)
# Robust version that coerces tree SHAP values to (n_samples, n_features)
# ===============================

!pip -q install shap

import os, numpy as np, pandas as pd
import joblib, shap, matplotlib.pyplot as plt

# 0) Output dir
os.makedirs("reports/shap", exist_ok=True)

# 1) Load tuned models
lr_path  = "models/logistic_regression_smote_tuned.pkl"
rf_path  = "models/random_forest_tuned.pkl"
xgb_path = "models/xgboost_tuned.pkl"

pipe_lr  = joblib.load(lr_path)
pipe_rf  = joblib.load(rf_path)
pipe_xgb = joblib.load(xgb_path)

# 2) Helper: transform & names from ColumnTransformer
def get_transformed_and_names(pipeline, X):
    pre = pipeline.named_steps["preprocessor"]
    X_trans = pre.transform(X)
    try:
        feat_names = pre.get_feature_names_out()
    except Exception:
        feat_names = np.array([f"f{i}" for i in range(X_trans.shape[1])])
    feat_names = list(np.asarray(feat_names).ravel().astype(str))
    return X_trans, feat_names

# Expect globals from training
assert all(k in globals() for k in ["X_train","y_train","X_test","y_test"]), \
    "Run training/tuning first."

# Sample sizes
N_GLOBAL = min(2000, len(X_test))
N_BG     = min(500,  len(X_train))
rng = np.random.RandomState(42)

sample_idx     = rng.choice(len(X_test), size=N_GLOBAL, replace=False)
X_test_sample  = X_test.iloc[sample_idx].copy()
y_test_sample  = y_test.iloc[sample_idx].copy()
bg_idx         = rng.choice(len(X_train), size=N_BG, replace=False)
X_bg           = X_train.iloc[bg_idx].copy()

# ---------- Plot helpers (Explanation API) ----------
def save_beeswarm(exp, path_png, max_display=20):
    plt.figure()
    shap.plots.beeswarm(exp, max_display=max_display, show=False)
    plt.tight_layout(); plt.savefig(path_png, dpi=150, bbox_inches="tight"); plt.close()

def save_bar_from_exp(exp, path_png, top_k=20):
    mean_abs = np.abs(exp.values).mean(axis=0)
    idx = np.argsort(-mean_abs)[:top_k]
    names = np.array(exp.feature_names)[idx]
    vals  = mean_abs[idx]
    plt.figure(figsize=(8,6))
    y = np.arange(len(idx))
    plt.barh(y, vals); plt.yticks(y, names); plt.gca().invert_yaxis()
    plt.xlabel("mean |SHAP value|"); plt.title("Global Feature Importance")
    plt.tight_layout(); plt.savefig(path_png, dpi=150, bbox_inches="tight"); plt.close()

def save_dependence(exp, feature_name, path_png):
    plt.figure()
    shap.plots.scatter(exp[:, feature_name], show=False)
    plt.tight_layout(); plt.savefig(path_png, dpi=150, bbox_inches="tight"); plt.close()

def save_waterfall(exp, i, path_png, max_display=15):
    if i is None: return
```

```
        plt.figure()
        shap.plots.waterfall(exp[i], max_display=max_display, show=False)
        plt.tight_layout(); plt.savefig(path_png, dpi=150, bbox_inches="tight"); plt.close()

    # ---------- Utilities to normalize tree outputs ----------
    def ensure_2d_positive_class(sv, n_samples, n_features):
        """
        Coerce SHAP values to shape (n_samples, n_features) selecting the positive class.
        Handles shapes:
          - (n_samples, n_features)                 -> return as-is
          - (2, n_samples, n_features)              -> sv[1]
          - (n_samples, 2, n_features)              -> sv[:, 1, :]
          - (n_samples, n_features, 1)              -> sv[:, :, 0]
          - any singleton dims -> squeeze if safe
        """
        arr = np.asarray(sv)
        if arr.ndim == 2 and arr.shape == (n_samples, n_features):
            return arr
        if arr.ndim == 3:
            if arr.shape[0] == 2 and arr.shape[1] == n_samples:
                return arr[1, :, :]                     # (classes, samples, features)
            if arr.shape[1] == 2 and arr.shape[0] == n_samples:
                return arr[:, 1, :]                     # (samples, classes, features)
            if arr.shape[2] == 1 and arr.shape[0] == n_samples:
                return arr[:, :, 0]                     # (samples, features, 1)
        # fallback: try squeeze and reshape conservatively
        arr2 = np.squeeze(arr)
        if arr2.ndim == 2 and arr2.shape[0] == n_samples:
            return arr2
        raise ValueError(f"Unexpected SHAP values shape {arr.shape}; cannot coerce to (samples, features).")

    def positive_class_base(expected_value):
        """Return a scalar base value for the positive class from TreeExplainer."""
        base = expected_value
        if isinstance(base, (list, tuple, np.ndarray)):
            base = np.asarray(base)
            if base.ndim == 0:
                base = float(base)
            else:
                # typically [base_class0, base_class1]
                base = float(base[1])
        else:
            base = float(base)
        return base

    # ---------- Main runner ----------
    def run_shap_for_model(pipeline, model_name, is_tree=False, is_linear=False):
        print(f"\n=== SHAP for {model_name} ===")

        # Transform through preprocessor
        X_test_trans_np, feature_names = get_transformed_and_names(pipeline, X_test_sample)
        X_bg_trans_np,   _             = get_transformed_and_names(pipeline, X_bg)

        clf = pipeline.named_steps["classifier"]

        if is_tree:
            # Tree models: use interventional + probability with background data
            explainer = shap.TreeExplainer(
                clf,
                data=X_bg_trans_np,
                model_output="probability",
                feature_perturbation="interventional"
            )
            raw_sv = explainer.shap_values(X_test_trans_np)
            sv_2d  = ensure_2d_positive_class(raw_sv, X_test_trans_np.shape[0], X_test_trans_np.shape[1])
            base_scalar = positive_class_base(explainer.expected_value)
            base_values = np.full(X_test_trans_np.shape[0], base_scalar, dtype=float)

        elif is_linear:
            masker = shap.maskers.Independent(X_bg_trans_np)
            explainer   = shap.LinearExplainer(clf, masker, link=shap.links.logit)
            exp_linear  = explainer(X_test_trans_np)
            sv_2d       = np.asarray(exp_linear.values)  # already 2D
```

```
            base_scalar = float(explainer.expected_value)
            base_values = np.full(X_test_trans_np.shape[0], base_scalar, dtype=float)

        else:
            masker = shap.maskers.Independent(X_bg_trans_np)
            explainer = shap.KernelExplainer(lambda d: clf.predict_proba(d)[:,1], masker)
            raw_sv = explainer.shap_values(X_test_trans_np, nsamples=500)
            sv_2d  = ensure_2d_positive_class(raw_sv, X_test_trans_np.shape[0], X_test_trans_np.shape[1])
            base_scalar = float(explainer.expected_value)
            base_values = np.full(X_test_trans_np.shape[0], base_scalar, dtype=float)

        # Build Explanation for plotting
        exp = shap.Explanation(
            values=sv_2d,
            base_values=base_values,
            data=X_test_trans_np,
            feature_names=feature_names
        )

        # ---- Global plots
        save_beeswarm(exp, f"reports/shap/{model_name}_summary_beeswarm.png")
        save_bar_from_exp(exp, f"reports/shap/{model_name}_summary_bar.png")

        # ---- Export mean |SHAP|
        mean_abs = np.abs(exp.values).mean(axis=0)
        imp_df = pd.DataFrame({"feature": feature_names, "mean_abs_shap": mean_abs}) \
                   .sort_values("mean_abs_shap", ascending=False)
        imp_df.to_csv(f"reports/shap/{model_name}_mean_abs_shap.csv", index=False)

        # ---- Local explanations: pick TP, FP, FN on the original sample
        y_pred = pipeline.predict(X_test_sample)
        idx_tp = idx_fp = idx_fn = None
        for i in range(len(X_test_sample)):
            if idx_tp is None and y_pred[i]==1 and y_test_sample.iloc[i]==1: idx_tp = i
            if idx_fp is None and y_pred[i]==1 and y_test_sample.iloc[i]==0: idx_fp = i
            if idx_fn is None and y_pred[i]==0 and y_test_sample.iloc[i]==1: idx_fn = i
            if idx_tp is not None and idx_fp is not None and idx_fn is not None: break

        save_waterfall(exp, idx_tp, f"reports/shap/{model_name}_local_tp_waterfall.png")
        save_waterfall(exp, idx_fp, f"reports/shap/{model_name}_local_fp_waterfall.png")
        save_waterfall(exp, idx_fn, f"reports/shap/{model_name}_local_fn_waterfall.png")

        # ---- Dependence for top feature
        top_feature = imp_df.iloc[0]["feature"]
        safe_name = str(top_feature).replace("/", "-")
        save_dependence(exp, top_feature, f"reports/shap/{model_name}_dependence_{safe_name}.png")

        print(f"Saved SHAP artifacts for {model_name} → reports/shap/")
        return imp_df

    # 4) Run for each model
    imp_lr  = run_shap_for_model(pipe_lr,  "lr_smote_tuned",        is_tree=False, is_linear=True)
    imp_rf  = run_shap_for_model(pipe_rf,  "random_forest_tuned",   is_tree=True)
    imp_xgb = run_shap_for_model(pipe_xgb, "xgboost_tuned",         is_tree=True)

    # 5) Top-10 tables
    def top10(df): return df.head(10).reset_index(drop=True)

    print("\nTop 10 features by mean |SHAP| (LR):")
    display(top10(imp_lr))
    print("\nTop 10 features by mean |SHAP| (RF):")
    display(top10(imp_rf))
    print("\nTop 10 features by mean |SHAP| (XGB):")
    display(top10(imp_xgb))

    print("\n SHAP plots saved under reports/shap/:")
    print("  - *_summary_beeswarm.png (global)")
    print("  - *_summary_bar.png (global)")
    print("  - *_mean_abs_shap.csv (global table)")
    print("  - *_local_tp/fp/fn_waterfall.png (local)")
    print("  - *_dependence_<topfeature>.png (interaction)")
```

```
=== SHAP for lr_smote_tuned ===
Saved SHAP artifacts for lr_smote_tuned → reports/shap/

=== SHAP for random_forest_tuned ===
100%|===================| 3997/4000 [11:13<00:00]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-1560810884.py in <cell line: 0>()
    196 # 4) Run for each model
    197 imp_lr  = run_shap_for_model(pipe_lr,  "lr_smote_tuned",        is_tree=False, is_linear=True)
--> 198 imp_rf  = run_shap_for_model(pipe_rf,  "random_forest_tuned",   is_tree=True)
    199 imp_xgb = run_shap_for_model(pipe_xgb, "xgboost_tuned",         is_tree=True)
    200

1 frames
/tmp/ipython-input-1560810884.py in ensure_2d_positive_class(sv, n_samples, n_features)
    100     if arr2.ndim == 2 and arr2.shape[0] == n_samples:
    101         return arr2
--> 102     raise ValueError(f"Unexpected SHAP values shape {arr.shape}; cannot coerce to (samples, features).")
    103
    104 def positive_class_base(expected_value):

ValueError: Unexpected SHAP values shape (2000, 107, 2); cannot coerce to (samples, features).
```

Next steps: Explain error

```
# ===============================
# SHAP for Tuned XGBoost (only)
# Robust handling of SHAP shapes + saved plots/CSV
# ===============================

!pip -q install shap

import os, numpy as np, pandas as pd
import joblib, shap, matplotlib.pyplot as plt

# --- Output dir
os.makedirs("reports/shap_xgb", exist_ok=True)

# --- Load tuned XGBoost pipeline you saved earlier
xgb_path = "models/xgboost_tuned.pkl"
pipe_xgb = joblib.load(xgb_path)

# --- Helpers to transform with pipeline's preprocessor
def get_transformed_and_names(pipeline, X):
    pre = pipeline.named_steps["preprocessor"]
    X_trans = pre.transform(X)
    try:
        feat_names = pre.get_feature_names_out()
    except Exception:
        feat_names = np.array([f"f{i}" for i in range(X_trans.shape[1])])
    feat_names = list(np.asarray(feat_names).ravel().astype(str))
    return X_trans, feat_names

# --- Expect training globals
assert all(k in globals() for k in ["X_train","y_train","X_test","y_test"]), \
    "Run training/tuning first to define X_train/y_train/X_test/y_test."

# --- Moderate sample sizes (increase if you like)
N_GLOBAL = min(2000, len(X_test))
N_BG     = min(500,  len(X_train))
rng = np.random.RandomState(42)

sample_idx     = rng.choice(len(X_test), size=N_GLOBAL, replace=False)
X_test_sample  = X_test.iloc[sample_idx].copy()
y_test_sample  = y_test.iloc[sample_idx].copy()
bg_idx         = rng.choice(len(X_train), size=N_BG, replace=False)
X_bg           = X_train.iloc[bg_idx].copy()

# --- Plot helpers (Explanation API)
```

```
def save_beeswarm(exp, path_png, max_display=20):
    plt.figure()
    shap.plots.beeswarm(exp, max_display=max_display, show=False)
    plt.tight_layout(); plt.savefig(path_png, dpi=150, bbox_inches="tight"); plt.close()

def save_bar_from_exp(exp, path_png, top_k=20):
    mean_abs = np.abs(exp.values).mean(axis=0)
    idx = np.argsort(-mean_abs)[:top_k]
    names = np.array(exp.feature_names)[idx]
    vals  = mean_abs[idx]
    plt.figure(figsize=(8,6))
    y = np.arange(len(idx))
    plt.barh(y, vals); plt.yticks(y, names); plt.gca().invert_yaxis()
    plt.xlabel("mean |SHAP value|"); plt.title("Global Feature Importance")
    plt.tight_layout(); plt.savefig(path_png, dpi=150, bbox_inches="tight"); plt.close()

def save_dependence(exp, feature_name, path_png):
    plt.figure()
    shap.plots.scatter(exp[:, feature_name], show=False)
    plt.tight_layout(); plt.savefig(path_png, dpi=150, bbox_inches="tight"); plt.close()

def save_waterfall(exp, i, path_png, max_display=15):
    if i is None: return
    plt.figure()
    shap.plots.waterfall(exp[i], max_display=max_display, show=False)
    plt.tight_layout(); plt.savefig(path_png, dpi=150, bbox_inches="tight"); plt.close()

# --- Normalize tree SHAP outputs to (n_samples, n_features), selecting positive class
def ensure_2d_positive_class(sv, n_samples, n_features):
    """
    Handles shapes:
      - (n_samples, n_features)                  -> return as-is
      - (2, n_samples, n_features)               -> sv[1, :, :]
      - (n_samples, 2, n_features)               -> sv[:, 1, :]
      - (n_samples, n_features, 2)               -> sv[:, :, 1]
      - (n_samples, n_features, 1)               -> sv[:, :, 0]
      - squeezes singleton dims when safe
    """
    arr = np.asarray(sv)
    if arr.ndim == 2 and arr.shape == (n_samples, n_features):
        return arr
    if arr.ndim == 3:
        # classes axis first
        if arr.shape[0] == 2 and arr.shape[1] == n_samples and arr.shape[2] == n_features:
            return arr[1, :, :]
        # classes axis second
        if arr.shape[0] == n_samples and arr.shape[1] == 2 and arr.shape[2] == n_features:
            return arr[:, 1, :]
        # classes axis last (this is what you saw: (samples, features, 2))
        if arr.shape[0] == n_samples and arr.shape[1] == n_features and arr.shape[2] == 2:
            return arr[:, :, 1]
        # singleton last
        if arr.shape[0] == n_samples and arr.shape[1] == n_features and arr.shape[2] == 1:
            return arr[:, :, 0]
    # final fallback
    arr2 = np.squeeze(arr)
    if arr2.ndim == 2 and arr2.shape[0] == n_samples and arr2.shape[1] == n_features:
        return arr2
    raise ValueError(f"Unexpected SHAP values shape {arr.shape}; cannot coerce to (samples, features).")

def positive_class_base(expected_value):
    """Return scalar baseline for the positive class from TreeExplainer expected_value."""
    base = expected_value
    if isinstance(base, (list, tuple, np.ndarray)):
        base = np.asarray(base)
        if base.ndim == 0:
            base = float(base)
        else:
            # [base_neg, base_pos]
            base = float(base[1])
    else:
        base = float(base)
    return base
```

```
# --- Run SHAP for XGBoost
def run_shap_xgb(pipeline, model_name="xgboost_tuned"):
    print(f"\n=== SHAP for {model_name} ===")

    X_test_trans_np, feature_names = get_transformed_and_names(pipeline, X_test_sample)
    X_bg_trans_np,   _             = get_transformed_and_names(pipeline, X_bg)

    clf = pipeline.named_steps["classifier"]

    # Use TreeExplainer with background for interventional, probability output
    explainer = shap.TreeExplainer(
        clf,
        data=X_bg_trans_np,
        model_output="probability",
        feature_perturbation="interventional"
    )
    raw_sv = explainer.shap_values(X_test_trans_np)

    # Normalize to (n_samples, n_features) for the positive class
    sv_2d = ensure_2d_positive_class(raw_sv, X_test_trans_np.shape[0], X_test_trans_np.shape[1])

    # Base values (positive class)
    base_scalar = positive_class_base(explainer.expected_value)
    base_values = np.full(X_test_trans_np.shape[0], base_scalar, dtype=float)

    # Build Explanation for plotting
    exp = shap.Explanation(
        values=sv_2d,
        base_values=base_values,
        data=X_test_trans_np,
        feature_names=feature_names
    )

    # Global plots
    save_beeswarm(exp, f"reports/shap_xgb/{model_name}_summary_beeswarm.png")
    save_bar_from_exp(exp, f"reports/shap_xgb/{model_name}_summary_bar.png")

    # Export mean |SHAP|
    mean_abs = np.abs(exp.values).mean(axis=0)
    imp_df = pd.DataFrame({"feature": feature_names, "mean_abs_shap": mean_abs}) \
                .sort_values("mean_abs_shap", ascending=False)
    imp_df.to_csv(f"reports/shap_xgb/{model_name}_mean_abs_shap.csv", index=False)

    # Local explanations: choose TP, FP, FN from sample
    y_pred = pipeline.predict(X_test_sample)
    idx_tp = idx_fp = idx_fn = None
    for i in range(len(X_test_sample)):
        if idx_tp is None and y_pred[i]==1 and y_test_sample.iloc[i]==1: idx_tp = i
        if idx_fp is None and y_pred[i]==1 and y_test_sample.iloc[i]==0: idx_fp = i
        if idx_fn is None and y_pred[i]==0 and y_test_sample.iloc[i]==1: idx_fn = i
        if idx_tp is not None and idx_fp is not None and idx_fn is not None: break

    save_waterfall(exp, idx_tp, f"reports/shap_xgb/{model_name}_local_tp_waterfall.png")
    save_waterfall(exp, idx_fp, f"reports/shap_xgb/{model_name}_local_fp_waterfall.png")
    save_waterfall(exp, idx_fn, f"reports/shap_xgb/{model_name}_local_fn_waterfall.png")

    # Dependence plot for most important feature
    top_feature = imp_df.iloc[0]["feature"]
    safe_name = str(top_feature).replace("/", "-")
    save_dependence(exp, top_feature, f"reports/shap_xgb/{model_name}_dependence_{safe_name}.png")

    print(f"Saved SHAP artifacts for {model_name} → reports/shap_xgb/")
    return imp_df

# --- Run it
imp_xgb = run_shap_xgb(pipe_xgb, "xgboost_tuned")

print("\nTop 10 features by mean |SHAP| (XGB):")
display(imp_xgb.head(10).reset_index(drop=True))

print("\n XGBoost SHAP saved under reports/shap_xgb/:")
print("  - xgboost_tuned_summary_beeswarm.png")
```

```
print("  - xgboost_tuned_summary_bar.png")
print("  - xgboost_tuned_mean_abs_shap.csv")
print("  - xgboost_tuned_local_tp/fp/fn_waterfall.png")
print("  - xgboost_tuned_dependence_<topfeature>.png")
```

```
=== SHAP for xgboost_tuned ===
/usr/local/lib/python3.12/dist-packages/shap/explainers/_tree.py:2043: UserWarning: [01:41:41] WARNING: /workspace/src/common/er

    E.g. tree_method = "hist", device = "cuda"

  raw = xgb_model.save_raw(raw_format="ubj")
Saved SHAP artifacts for xgboost_tuned → reports/shap_xgb/

Top 10 features by mean |SHAP| (XGB):
```

	feature	mean_abs_shap
0	cat__marital_status_Married-civ-spouse	0.132575
1	num__age	0.078894
2	num__education_num	0.063183
3	num__hours_per_week	0.046578
4	num__capital_gain	0.037345
5	cat__occupation_Exec-managerial	0.015859
6	cat__sex_Female	0.014623
7	cat__marital_status_Never-married	0.013399
8	cat__occupation_Prof-specialty	0.011220
9	cat__occupation_Other-service	0.010866

```
XGBoost SHAP saved under reports/shap_xgb/:
 - xgboost_tuned_summary_beeswarm.png
 - xgboost_tuned_summary_bar.png
 - xgboost_tuned_mean_abs_shap.csv
 - xgboost_tuned_local_tp/fp/fn_waterfall.png
 - xgboost_tuned_dependence_<topfeature>.png
```

```
# ==============================
# Metrics pack (ROC/PR/Calibration + Threshold sweep) & Fairness audit
# for tuned XGBoost pipeline
# ==============================

import os, json, numpy as np, pandas as pd, joblib, matplotlib.pyplot as plt
from sklearn.metrics import (
    roc_curve, auc, precision_recall_curve, average_precision_score,
    brier_score_loss, precision_score, recall_score, f1_score, accuracy_score
)
from sklearn.calibration import CalibrationDisplay

# ---- Paths & preconditions
MODEL_PATH = "models/xgboost_tuned.pkl"
os.makedirs("reports/metrics", exist_ok=True)

# Expect these globals from your earlier notebook:
#   train_df, test_df, X_test, y_test
assert "X_test" in globals() and "y_test" in globals(), "Please define X_test/y_test first."
assert "test_df" in globals(), "Please keep test_df available for fairness slices."

# ---- Load tuned pipeline
pipe_xgb = joblib.load(MODEL_PATH)

# ---- Probabilities on test set
probs = pipe_xgb.predict_proba(X_test)[:, 1]

# ==============================
# (1) METRICS PACK
```

```
# ==============================

# --- ROC curve & AUC
fpr, tpr, _ = roc_curve(y_test, probs)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, lw=2, label=f"XGB (AUC = {roc_auc:.3f})")
plt.plot([0,1], [0,1], "--", color="gray", lw=1)
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve — Tuned XGBoost")
plt.legend(loc="lower right")
plt.tight_layout()
plt.savefig("reports/metrics/roc_curve_xgb.png", dpi=150, bbox_inches="tight")
plt.close()

# --- PR curve & Average Precision (PR-AUC)
prec, rec, _ = precision_recall_curve(y_test, probs)
ap = average_precision_score(y_test, probs)

plt.figure()
plt.plot(rec, prec, lw=2, label=f"XGB (AP = {ap:.3f})")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision–Recall Curve — Tuned XGBoost")
plt.legend(loc="lower left")
plt.tight_layout()
plt.savefig("reports/metrics/pr_curve_xgb.png", dpi=150, bbox_inches="tight")
plt.close()

# --- Calibration curve & Brier score
plt.figure()
CalibrationDisplay.from_predictions(y_test, probs, n_bins=10)
plt.title("Reliability Curve — Tuned XGBoost")
plt.tight_layout()
plt.savefig("reports/metrics/calibration_curve_xgb.png", dpi=150, bbox_inches="tight")
plt.close()

brier = brier_score_loss(y_test, probs)

# --- Threshold sweep to select operating point (maximize F1 for positive class)
def threshold_sweep(y_true, y_prob, lo=0.2, hi=0.9, steps=71):
    ths = np.linspace(lo, hi, steps)
    rows = []
    for t in ths:
        yp = (y_prob >= t).astype(int)
        p  = precision_score(y_true, yp, zero_division=0)
        r  = recall_score(y_true, yp, zero_division=0)
        f1 = f1_score(y_true, yp, zero_division=0)
        acc= accuracy_score(y_true, yp)
        rows.append((t, p, r, f1, acc))
    return pd.DataFrame(rows, columns=["threshold","precision","recall","f1","accuracy"])

sweep_df = threshold_sweep(y_test, probs, lo=0.2, hi=0.9, steps=71)
best_row = sweep_df.loc[sweep_df["f1"].idxmax()].to_dict()
best_threshold = float(best_row["threshold"])

plt.figure()
plt.plot(sweep_df["threshold"], sweep_df["f1"], lw=2, label="F1 (pos class)")
plt.plot(sweep_df["threshold"], sweep_df["precision"], lw=1, alpha=0.7, label="Precision")
plt.plot(sweep_df["threshold"], sweep_df["recall"], lw=1, alpha=0.7, label="Recall")
plt.axvline(best_threshold, color="red", linestyle="--", label=f"Best t={best_threshold:.3f}")
plt.xlabel("Threshold")
plt.ylabel("Score")
plt.title("Threshold Sweep — Tuned XGBoost")
plt.legend(loc="best")
plt.tight_layout()
plt.savefig("reports/metrics/threshold_sweep_xgb.png", dpi=150, bbox_inches="tight")
plt.close()

# Save config with chosen threshold
config = {
```

```
        "model_path": MODEL_PATH,
        "threshold": best_threshold,
        "metrics": {
            "roc_auc": float(roc_auc),
            "pr_ap": float(ap),
            "brier": float(brier),
            "best_f1": float(best_row["f1"]),
            "best_precision": float(best_row["precision"]),
            "best_recall": float(best_row["recall"]),
            "best_accuracy": float(best_row["accuracy"])
        }
    }
    with open("reports/metrics/config.json", "w") as f:
        json.dump(config, f, indent=2)

    # ==============================
    # (2) FAIRNESS / BIAS AUDIT
    # ==============================

    # We audit on the test split using original test_df attributes
    df_eval = test_df.copy()
    df_eval = df_eval.assign(
        y_true = y_test.values,
        y_prob = probs,
        y_pred = (probs >= best_threshold).astype(int)
    )

    def group_metrics(df, group_col, ref_value=None):
        """
        Returns per-group metrics:
          - positive_rate (Statistical Parity)
          - TPR (Equal Opportunity)
          - PPV (Predictive Parity / Precision)
          - DI (Disparate Impact: group positive rate / reference positive rate)
        """
        rows = []
        # choose reference (default: majority group by count, or provided)
        if ref_value is None:
            ref_value = df[group_col].value_counts().idxmax()
        ref_rate = (df.loc[df[group_col]==ref_value, "y_pred"]==1).mean()
        for g, sub in df.groupby(group_col):
            pr = (sub["y_pred"]==1).mean()                                       # Statistical Parity (positive rate)
            # TPR
            tp = ((sub["y_pred"]==1) & (sub["y_true"]==1)).sum()
            fn = ((sub["y_pred"]==0) & (sub["y_true"]==1)).sum()
            tpr = tp / (tp + fn) if (tp+fn) > 0 else 0.0
            # PPV
            pp = ((sub["y_pred"]==1) & (sub["y_true"]==1)).sum()
            fp = ((sub["y_pred"]==1) & (sub["y_true"]==0)).sum()
            ppv = pp / (pp + fp) if (pp+fp) > 0 else 0.0
            di  = pr / ref_rate if ref_rate > 0 else np.nan
            rows.append((group_col, g, pr, tpr, ppv, di, len(sub)))
        return pd.DataFrame(rows, columns=["attribute","group","positive_rate","tpr","ppv","disparate_impact","count"]).sort_values(["at

    # Sex audit
    sex_audit  = group_metrics(df_eval, "sex", ref_value="Male" if "Male" in df_eval["sex"].unique() else None)
    # Race audit
    race_audit = group_metrics(df_eval, "race", ref_value="White" if "White" in df_eval["race"].unique() else None)

    fairness_df = pd.concat([sex_audit, race_audit], ignore_index=True)
    fairness_df.to_csv("reports/metrics/fairness_audit.csv", index=False)

    # Pretty print summary
    print("=== XGBoost — Metrics Summary ===")
    print(f"ROC AUC: {roc_auc:.3f} | PR AP: {ap:.3f} | Brier: {brier:.4f}")
    print(f"Best threshold (F1+): {best_threshold:.3f}  "
          f"| F1: {best_row['f1']:.3f}  Prec: {best_row['precision']:.3f}  Rec: {best_row['recall']:.3f}  Acc: {best_row['accuracy']:.3f

    print("\n=== Fairness Audit (saved to reports/metrics/fairness_audit.csv) ===")
    display(fairness_df)

    print("\n Saved files:")
    print(" - reports/metrics/roc_curve_xgb.png")
```

```
print(" - reports/metrics/pr_curve_xgb.png")
print(" - reports/metrics/calibration_curve_xgb.png")
print(" - reports/metrics/threshold_sweep_xgb.png")
print(" - reports/metrics/config.json")
print(" - reports/metrics/fairness_audit.csv")
```

```
/usr/local/lib/python3.12/dist-packages/xgboost/core.py:2676: UserWarning: [01:48:56] WARNING: /workspace/src/common/error_msg.c

    E.g. tree_method = "hist", device = "cuda"

  if len(data.shape) != 1 and self.num_features() != data.shape[1]:
=== XGBoost — Metrics Summary ===
ROC AUC: 0.925 | PR AP: 0.820 | Brier: 0.1130
Best threshold (F1+): 0.640  | F1: 0.718  Prec: 0.690  Rec: 0.749  Acc: 0.861

=== Fairness Audit (saved to reports/metrics/fairness_audit.csv) ===
```

	attribute	group	positive_rate	tpr	ppv	disparate_impact	count
0	sex	Female	0.098690	0.649153	0.715888	0.294606	5421
1	sex	Male	0.334991	0.766585	0.686091	1.000000	10860
2	race	Amer-Indian-Eskimo	0.132075	0.789474	0.714286	0.484080	159
3	race	Asian-Pac-Islander	0.314583	0.766917	0.675497	1.153004	480
4	race	Black	0.112108	0.675978	0.691429	0.410894	1561
5	race	Other	0.155556	0.720000	0.857143	0.570139	135
6	race	White	0.272838	0.751576	0.689356	1.000000	13946

```
Saved files:
- reports/metrics/roc_curve_xgb.png
- reports/metrics/pr_curve_xgb.png
- reports/metrics/calibration_curve_xgb.png
- reports/metrics/threshold_sweep_xgb.png
- reports/metrics/config.json
- reports/metrics/fairness_audit.csv
```

Next steps: Generate code with fairness_df | View recommended plots | New interactive sheet

```
# ===============================
# Fairness Audit — sex & race (parity, TPR, PPV, DI)
# Saves CSV and bar plots
# ===============================

import os, json, numpy as np, pandas as pd, matplotlib.pyplot as plt
from sklearn.metrics import precision_score, recall_score, confusion_matrix
import joblib

# --- Paths
MODEL_PATH = "models/xgboost_tuned.pkl"
METRICS_DIR = "reports/metrics"
os.makedirs(METRICS_DIR, exist_ok=True)

# --- Load model & (optional) saved threshold
pipe_xgb = joblib.load(MODEL_PATH)

cfg_path = os.path.join(METRICS_DIR, "config.json")
if os.path.exists(cfg_path):
    with open(cfg_path, "r") as f:
        cfg = json.load(f)
    threshold = float(cfg.get("threshold", 0.65))
else:
    threshold = 0.65  # fallback if config not saved yet

# --- Preconditions: we need these from your previous steps
assert "X_test" in globals() and "y_test" in globals(), "Please define X_test/y_test first."
assert "test_df" in globals(), "Please keep test_df available for fairness slices."

# --- Predict on test
```

```
probs = pipe_xgb.predict_proba(X_test)[:, 1]
y_pred = (probs >= threshold).astype(int)

# --- Build evaluation frame (keep original attributes for slicing)
df_eval = test_df.copy()
df_eval = df_eval.assign(y_true=y_test.values, y_prob=probs, y_pred=y_pred)

# --- Metrics per group
def group_metrics(df, group_col, ref_value=None):
    """
    Returns per-group metrics:
      - positive_rate (Statistical Parity)
      - TPR (Equal Opportunity)
      - PPV (Predictive Parity / Precision)
      - DI (Disparate Impact: group positive rate / reference positive rate)
      - counts (support)
    """
    rows = []
    # choose reference: provided or majority-count group
    if ref_value is None:
        ref_value = df[group_col].value_counts().idxmax()
    ref_rate = (df.loc[df[group_col]==ref_value, "y_pred"]==1).mean()
    for g, sub in df.groupby(group_col):
        pr  = (sub["y_pred"]==1).mean()
        tp  = ((sub["y_pred"]==1) & (sub["y_true"]==1)).sum()
        fn  = ((sub["y_pred"]==0) & (sub["y_true"]==1)).sum()
        fp  = ((sub["y_pred"]==1) & (sub["y_true"]==0)).sum()
        # TPR, PPV
        tpr = tp / (tp + fn) if (tp + fn) > 0 else 0.0
        ppv = tp / (tp + fp) if (tp + fp) > 0 else 0.0
        di  = pr / ref_rate if ref_rate > 0 else np.nan
        rows.append((group_col, ref_value, g, pr, tpr, ppv, di, len(sub)))
    out = pd.DataFrame(rows, columns=[
        "attribute","reference_group","group","positive_rate","tpr","ppv","disparate_impact","count"
    ]).sort_values(["attribute","group"]).reset_index(drop=True)
    return out

# --- Compute audits
sex_ref  = "Male"  if "Male"  in df_eval["sex"].unique()  else None
race_ref = "White" if "White" in df_eval["race"].unique() else None

sex_audit  = group_metrics(df_eval, "sex",  ref_value=sex_ref)
race_audit = group_metrics(df_eval, "race", ref_value=race_ref)

fairness_df = pd.concat([sex_audit, race_audit], ignore_index=True)
fairness_path = os.path.join(METRICS_DIR, "fairness_audit.csv")
fairness_df.to_csv(fairness_path, index=False)

# --- Plot helpers
def bar_plot(metric_df, attr, metric_col, title, fname, ref_line=None, ylim=(0,1.0)):
    sub = metric_df[metric_df["attribute"] == attr].copy()
    labels = sub["group"].astype(str).tolist()
    values = sub[metric_col].values.astype(float)

    plt.figure(figsize=(8,4.5))
    x = np.arange(len(labels))
    plt.bar(x, values)
    plt.xticks(x, labels, rotation=30, ha="right")
    plt.ylim(*ylim)
    plt.ylabel(metric_col)
    plt.title(title)
    if ref_line is not None:
        plt.axhline(ref_line, linestyle="--")
    plt.tight_layout()
    out = os.path.join(METRICS_DIR, fname)
    plt.savefig(out, dpi=150, bbox_inches="tight")
    plt.close()
    return out

# --- Make plots for sex & race: positive_rate, tpr, ppv, disparate_impact
plots = []
for attr in ["sex","race"]:
    # Positive rate (Statistical Parity)
```

```
        plots.append(bar_plot(fairness_df, attr, "positive_rate",
                              f"{attr} — Positive Rate (Statistical Parity)",
                              f"{attr}_positive_rate.png", ref_line=None))
        # TPR (Equal Opportunity)
        plots.append(bar_plot(fairness_df, attr, "tpr",
                              f"{attr} — True Positive Rate (Equal Opportunity)",
                              f"{attr}_tpr.png", ref_line=None))
        # PPV (Predictive Parity)
        plots.append(bar_plot(fairness_df, attr, "ppv",
                              f"{attr} — Precision / PPV (Predictive Parity)",
                              f"{attr}_ppv.png", ref_line=None))
        # Disparate Impact with 0.8 rule line
        plots.append(bar_plot(fairness_df, attr, "disparate_impact",
                              f"{attr} — Disparate Impact (vs. reference)",
                              f"{attr}_disparate_impact.png", ref_line=0.8, ylim=(0,1.4)))

    # --- Quick console summary: flag DI below 0.8
    flags = fairness_df[(~fairness_df["disparate_impact"].isna()) & (fairness_df["disparate_impact"] < 0.8)]
    print("=== Fairness Audit — Summary ===")
    print(f"Threshold used: {threshold:.3f}")
    print(f"Saved table: {fairness_path}")
    print("Saved plots:")
    for p in plots: print(" -", p)

    if len(flags):
        print("\n  Groups with Disparate Impact < 0.8 (potential concern):")
        display(flags[["attribute","reference_group","group","disparate_impact","count"]])
    else:
        print("\n No groups with DI < 0.8 at this threshold.")

    # Also print the full table (top)
    print("\nFull fairness table (head):")
    display(fairness_df.head(20))
```

```
/usr/local/lib/python3.12/dist-packages/xgboost/core.py:2676: UserWarning: [01:54:43] WARNING: /workspace/src/common/error_msg.c

    E.g. tree_method = "hist", device = "cuda"

  if len(data.shape) != 1 and self.num_features() != data.shape[1]:
=== Fairness Audit — Summary ===
Threshold used: 0.640
Saved table: reports/metrics/fairness_audit.csv
Saved plots:
 - reports/metrics/sex_positive_rate.png
 - reports/metrics/sex_tpr.png
 - reports/metrics/sex_ppv.png
```