



Adult				
	age	workclass	fnlwgt	education
1	90	?	77053	HS-grad
2	82	Private	132870	HS-grad
3	66	?	186061	Some-colleg
4	54	Private	140359	7th-8th
5	41	Private	264663	Some-colleg
6	34	Private	216864	HS-grad

Role/Persona: You are Dr. Alex Rivera, a world-renowned expert and award-winning author on CRISP-DM. Teach clearly, be blunt and honest.

Scope: This is a mini demo of the CRISP-DM methodology using the Adult/Census Income dataset (predict >50K). The goal is to demonstrate each CRISP-DM phase, not chase SOTA accuracy. Data location: Assume I'm in Colab and adult.csv is uploaded in my Google Drive (e.g., /content/drive/MyDrive/adult.csv). Include a Drive-mount cell and load from that path. If not found, gracefully fallback to UCI Adult (no auth) with matching schema. Process & delivery rules (very important):

Follow CRISP-DM strictly: Business Understanding → Data Understanding → Data Preparation → Modeling → Evaluation → Deployment.

Deliver one chat chunk per phase. In each chunk provide:

a short phase narrative, and

Colab-ready code for that phase only.

After the code, give your critic review (numbered issues) and a short acceptance checklist. Stop and wait for me to reply "continue" before the next phase.

Keep the entire notebook within 15–20 cells total across all phases (aim ~2–3 small cells per phase).

Keep each phase's code in concise, incremental cells—avoid large monolithic cells; each block should do one small, clear step so its effect is easy to understand.

Use lightweight, standard libs only (pandas, scikit-learn, matplotlib). Fix random seeds; use reproducible Pipeline/ColumnTransformer for prep; save minimal artifacts (model.pkl, metrics.json).

Use PR-AUC as a primary metric (class imbalance), plus threshold selection with a simple cost matrix. Include quick fairness slices by sex and race (for auditing only).

Begin now with Phase 1 — Business Understanding. Provide the narrative, then the small incremental code cells for that phase, then your critic review + acceptance checklist. Then wait for "continue."

Thought for 59s >

Prompt

Blunt goal: use the Adult/Census Income data to predict whether income > \$50K so a (hypothetical) ops team can prioritize limited outreach budget. We don't chase leaderboard points; we show a clean, reproducible CRISP-DM flow.

Business decision simulated: who gets a premium offer first.

Primary measure: PR-AUC (Average Precision) — class imbalance matters.

Operational choice: pick a decision threshold by minimizing expected cost with a simple 2x2 cost matrix. For this demo:

- FP cost = 1 (we bother a lower-income person with a premium offer)
- FN cost = 5 (we miss a higher-income person we wanted to prioritize)

Fairness audit (lightweight): compare core metrics and error rates across sex and race slices. This is an audit, not debiasing.

Constraints: tiny, transparent code; fixed random seed's; standard libs (pandas, scikit-learn, matplotlib); minimal artifacts (`model.pkl`, `metrics.json`, `prep.pkl`). Thresholding and fairness checks must be

Code — Phase 1 (project scaffolding & config only)

Keep cells small. No data loading yet (that's Phase 2).

1) Reproducibility + artifacts folder

```
python Copy code
# Phase 1 - setup
import os, json, random, numpy as np
from pathlib import Path

RANDOM_STATE = 42
np.random.seed(RANDOM_STATE); random.seed(RANDOM_STATE)
 $\downarrow$ 
BASE_DIR = Path("/content") # Colab default working dir

ARTIFACT_DIR.mkdir(parents=True, exist_ok=True)

print(f"Artifacts dir: {ARTIFACT_DIR}")
print(f"Random seed: {RANDOM_STATE}")
```

2) Business config (target, costs, metrics, fairness slices, data locations)

```
python Copy code
# Phase 1 - business configuration
config = {
    "project": "Adult Income - CRISP-DM Demo",
    "random_state": 42,

    # Target definition
    "target": "income",
    "positive_label": ">50K",
 $\downarrow$ 

    # Primary metric
    "primary_metric": "average_precision", # PR-AUC
    "secondary_metrics": ["roc_auc", "f1", "precision", "recall"],
    "threshold_policy": "minimize_cost",
    "cost_matrix": {"FP": 1.0, "FN": 5.0},

    # Fairness audit slices (auditing only)
    "fairness_slices": ["sex", "race"],

    # Data location strategy (Phase 2 will honor this)
    "data": {
        "local_drive_path": "/content/drive/MyDrive/adult.csv",
        "fallback": {
            "train_url": "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult_train.csv",
            "test_url": "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult_test.csv",
            "names_url": "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult_names.csv"
        }
    },
 $\downarrow$ 

    "artifacts": {
        "model_path": str(ARTIFACT_DIR / "model.pkl"),
        "metrics_path": str(ARTIFACT_DIR / "metrics.json"),
        "prep_path": str(ARTIFACT_DIR / "prep.pkl"),
        "config_path": str(ARTIFACT_DIR / "config.json")
    }
}

with open(ARTIFACT_DIR / "config.json", "w") as f:
    json.dump(config, f, indent=2)

print("Saved:", ARTIFACT_DIR / "config.json")
```

3) Quick sanity print (human-readable)

```
python Copy code
 $\downarrow$ 
import json, pprint
cfg = json.loads(pathlib.Path(ARTIFACT_DIR / "config.json").read_text())
pp = pprint.PrettyPrinter(indent=2, width=100)
pp.pprint({
    "project": cfg["project"],
    "target": cfg["target"],
    "positive_label": cfg["positive_label"]})
```

```
    "target": cfg["target"], "cfg["positive_label"]},  
    "primary_metric": cfg["primary_metric"],  
    "cost_matrix": cfg["cost_matrix"],  
    "threshold_policy": cfg["threshold_policy"],  
    "fairness_slices": cfg["fairness_slices"],  
    "artifacts": cfg["artifacts"]  
}
```

Critic Review (what could bite us later)

1. **Toy cost matrix.** We guessed FP=1, FN=5. Real costs are rarely this neat; sensitivity analysis will be needed.
 2. **Data quality & bias.** Adult data is old and simplified; external validity is weak. This is a demo only.
 3. **Fairness is shallow.** Slicing by sex/face surfaces disparities but doesn't fix them; no causal analysis or policy review here.
 4. **Threshold-based decisioning.** One size may not fit all segments; business rules or dual thresholds might be better.
 5. **Downstream constraints ignored.** We haven't defined service-level latency or volume caps that could affect model complexity.

Acceptance Checklist (Phase 1)

- Business decision and positive class are clearly defined.
 - Primary metric = PR-AUC chosen for imbalance; threshold picked by minimizing cost.
 - Cost matrix documented ($FP=1$, $FN=5$) with a note to stress-test later.
 - Fairness slices (sex, race) agreed for auditing.
 - Artifacts & seeds specified for reproducibility.

Reply “**continue**” to proceed to Phase 2 — Data Upstanding (we’ll mount Drive, try [/content/drive/Mydrive/adult.csv](https://content/drive/Mydrive/adult.csv), and gracefully fall back to UCI with a matching schema).

first explain each stage. then only after i say code, then generate code as i need to verify each steps in methodology. continue next-data understanding

Thought for 25s >

Alright—before we touch a single line of code, here's the full CRISP-DM map in blunt, practical terms.

The Six CRISP-DM Stages (quick, no-fluff)

1. Business Understanding

Define the *decision*, not just the model. Clarify the positive class, success metric(s), costs of mistakes, and constraints. Output: a short spec ↓ you could hand to an exec.

2 Data Understanding (you're here now!)

Produce quick plots/tables that prove you actually looked. Output: a brief audit + risks list.

3. Data Preparation

Freeze a reproducible pipeline: splits, imputers, encoders, scalers, feature selections, and sanity filters. Zero ad-hoc notebook magic; everything inside a `Pipeline/ColumnTransformer`. Output: fitted preprocessing artifact you can reuse.

4. Modeling

Train a few sensible baselines (logistic, tree/boosting) with fixed seeds. Keep search tiny; prioritize stability, not SOTA. Output: candidate models + PR curves.

5. Evaluation

Pick the threshold with your cost matrix (FN vs FP), report PR-AUC as primary, and publish a small, honest metrics pack (calibration, confusion, and fairness slices by sex/race). Output: `metrics.json` + chosen threshold and rationale.

6. Deployment

Save the minimal artifacts (`prep.pkl`, `model.`, chosen `threshold`) and a tiny inference function. Show a couple of sample predictions and the CLI/API-ish contract. Output: artifacts + quick

Phase 2 — Data Understanding (narrative only; no code until you say “code”)

Objective: Prove we know the data we're about to engineer and model. For Adult/Census Income, we'll confirm schema, clean oddities, and quantify risks.

What we'll check (and why)

- Load strategy & provenance
 - Primary: `/content/drive/MyDrive/adult.csv` (your Drive).
 - Fallback: UCI Adult (`adult.data` + `adult.test`) with the same schema; strip trailing periods in test labels; treat "?" as missing.
 - Leakage guard: If both train/test exist (UCI), we'll EDA train only and keep test blind.
 - Schema & dtypes (expected)
 - Numeric: `age`, `fnlwgt`, `education-num`, `capital-gain`, `capital-loss`, `hours-per-week`

- sex, native-country
- Target: income in {<=50K, >50K} (ensure exact tokens after cleanup)
- Row counts & duplicates
 - Dataset size, % duplicates, and whether duplicates look harmful (e.g., true repeats vs. common value combos). Decision: drop exact duplicates.
- Target prevalence & imbalance
 - Baseline rate of >50%. This dictates PR-AUC relevance and helps set expectations for precision/recall tradeoffs.
- Missingness & weird tokens
 - "?" as NA in several categoricals; confirm NA rates and which columns are most affected. Decide between "Missing" bucket vs. drop if catastrophic.
- Cardinality & category health
 - native-country often has many rare categories; combine ultra-rare levels into Other during prep to stabilize.
- Distribution/Outliers
 - Check for implausible ages/hours, and long tails.
- Fairness slices "up front"
 - Snapshot of target rate by sex and race (only to understand population differences—not to adjust yet).
- Potential leakage
 - Confirm no columns encode the target directly or temporally leak information (nothing obvious in Adult, but we still flag education-num vs education redundancy—ok to keep both for now).
- Train/test alignment (if using UCI split)
 - Ensure feature distributions don't wildly shift between train and test (quick KS checks/compare means/categories). If Drive has a single CSV, we'll later stratify split.

Deliverables from this phase

- A short EDA printout: shapes, dtypes, missingness table, target prevalence, top-N category values, simple histograms for key numeric features, and nice totals for sex/race.
- A risk list and decisions (e.g., treat ? as missing; collapse rare native-country; drop exact dups;

Critic Review (Data Understanding risks)

- Old data ≠ modern workforce. External validity is weak; we're demonstrating process, not production truth.
- Target token inconsistencies. The UCI test set includes trailing periods and capitalization quirks—forgetting to normalize ruins metrics.
- Sparse gains/losses. Models may overfit rare positive capital-gain spikes; we should check monotonicity or cap extremes.
- Country cardinality. Many rare categories inflate variance; must bucket or they'll hurt generalization.
- Fairness read is descriptive only. Differences by sex/race don't imply causes or remedies; we're auditing, not adjusting.

Acceptance Checklist (Data Understanding)

- Data successfully loaded from Drive or clear .com UCI with matching schema and normalized labels.
- Schema/labels (expected column/dtypes, and normalization policy) documented (in notebook, noted).
- Target prevalence computed; class imbalance acknowledged.
- Duplicates identified and policy decided (drop exact duplicates).
- Cardinality and rare levels policy stated (e.g., bucket native-country).
- Fairness slices (sex, race) summarized descriptively.
- Leakage check completed; no suspect fields flagged beyond acceptable redundancies.

Say "code" when you're ready for the Phase 2 — Data Understanding code cells.

⟳ ⏪ ⏴ ⏵ ⏷ ⏸ ⏹

code

Phase 2 — Data Understanding (code only)

(Short narrative) We'll load the dataset from your Google Drive path and gracefully fall back to UCI if it's not there, then quantify what we're working with: schema, missingness, target prevalence, duplicates, category cardinality, and quick fairness slices (sex, race). No modeling yet.

1) Mount Drive + Load (Drive first, UCI fallback) + Normalize schema/labels

```
python
# Phase 2 - cell 1: Load data with Drive-first, UCI fallback
import pandas as pd, numpy as np, json, os
```

```

from pathlib import Path

# Read Phase-1 config if present, else use sane defaults
#ARTIFACT_DIR = Path("/content/artifacts")
cfg_path = Path("/content/config.json")
if cfg_path.exists():
    config = json.loads(cfg_path.read_text())
else:
    config = {
        "target": "income",
        "positive_label": ">50K",
        "data": {
            "local_drive_path": "/content/drive/MyDrive/adult.csv",
            "fallback": {
                "train_url": "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult_train.csv",
                "test_url": "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult_test.csv"
            }
        }
    }

TARGET = config.get("target", "income")
POS_LABEL = config.get("positive_label", ">50K")

# Try to mount Google Drive (Colab)
try:
    from google.colab import drive # will succeed in Colab
    drive.mount('/content/drive', force_remount=False)
except Exception as e:
    print("Note: Colab Drive not available in this environment. Proceeding without explicit mount.")

# Expected Adult schema (order matters)
EXPECTED_COLS = [
    "age", "workclass", "fnlwgt", "education", "education-num", "marital-status",
    "occupation", "relationship", "race", "sex", "capital-gain", "capital-loss",
    "hours-per-week", "native-country", "income"
]

local_path = Path(config["data"]["local_drive_path"])
provenance = None

def clean_income_series(s):
    return s.replace("?", np.nan)

    if local_path.exists():
        df = pd.read_csv(local_path)
        provenance = f"drive_csv:{local_path}"
    else:
        # UCI fallback: load TRAIN for EDA, keep TEST rows count noted (held-out usage later)
        cols = EXPECTED_COLS
        tr_url = config["data"]["fallback"]["train_url"]
        te_url = config["data"]["fallback"]["test_url"]

        df_train = pd.read_csv(tr_url, header=None, names=cols,
                               na_values=["?"], skipinitialspace=True)
        df_test = pd.read_csv(te_url, header=None, names=cols,
                               na_values=["?"], skipinitialspace=True, skiprows=1)

        # Normalize trailing periods and stray spaces in target for test; we EDA on train only
        df_train[TARGET] = clean_income_series(df_train[TARGET])
        df_test[TARGET] = clean_income_series(df_test[TARGET])

    df = df_train.copy()
    provenance = "uci_fallback:train_only"

    # Normalize Drive CSV to expected schema/labels
    # 1) Trim object columns and convert "?" -> NA (if present)
    for c in df.columns:
        if df[c].dtype == object:
            df[c] = df[c].astype(str).str.strip().replace({"?": np.nan})

    # 2) Ensure expected columns and order if possible
    if set(EXPECTED_COLS).issubset(set(df.columns)):
        df = df[EXPECTED_COLS]
    else:
        print("Warning: Columns differ from expected; proceeding with current columns.")

    # 3) Normalize target to {<=50K, >50K}
    if TARGET in df.columns:
        df[TARGET] = clean_income_series(df[TARGET])

    # Restrict to the two canonical labels, drop weird rows if any
    valid_labels = {"<=50K", ">50K"}
    if TARGET in df.columns:
        before = len(df)
        df = df[df[TARGET].isin(valid_labels)].copy()
        dropped = before - len(df)

        if dropped > 0:
            print(f"Dropped {dropped} rows with non-standard target labels.")

print(f"Uploaded from {provenance}: shape={df.shape}")

```

```
print("Loaded from: %s; shape: %s" % (provenance, df.shape))
df.head(3)
```

2) Schema check, missingness, target prevalence, duplicates

```
python ⌂ Copy code
```

```
import numpy as np

print("Shape:", df.shape)
print("\nDtypes:")
print(df.dtypes)

# Missingness summary (top 10 by missing rate)
na_rate = df.isna().mean().sort_values(ascending=False)
print("\nMissingness (top 10):")
print((na_rate.head(10) * 100).round(2).astype(str) + "%")

# Target prevalence (class imbalance)
if TARGET in df.columns:
    pos_rate = (df[TARGET] == POS_LABEL).mean()
    print(f"\nTarget prevalence ({POS_LABEL}): {pos_rate:.3f}")

# Duplicate rows
dup_count = df.duplicated().sum()

print(f"\nDuplicate rows: {dup_count} ({dup_pct:.2%})")

# Basic numeric summary (compact)
num_cols = [c for c in df.columns if df[c].dtype != 'O' and c != TARGET]
print("\nNumeric columns:", num_cols)
display(df[num_cols].describe().T.iloc[:, :6]) # count, mean, std, min, 25%, 50%
```

3) Category cardinality, fairness slices (sex/race), and a couple of quick histograms

```
python ⌂ Copy code
```

```
# Phase 2 - cell 3: Cardinality + fairness + quick dists
import matplotlib.pyplot as plt

# Identify categorical columns (exclude target)
cat_cols = [c for c in df.columns if df[c].dtype == 'O' and c != TARGET]

# Cardinality overview (top 10)
card = df[cat_cols].nunique().sort_values(ascending=False)
print("Categorical cardinality (top 10):")
print(card.head(10))

# Peek at most diverse field (often native-country)
if "native-country" in df.columns:
    print("\nTop 10 'native-country' values:")
    print(df['native-country'].value_counts(dropna=False).head(10))

# Fairness slices - positive rate by sex and race
if TARGET in df.columns:
    y = (df[TARGET] == POS_LABEL).astype(int)
    if "sex" in df.columns:
        print("\nPositive rate by sex:")
        print(y.groupby(df['sex']).mean().sort_values(ascending=False).round(3))
    if "race" in df.columns:
        print("\nPositive rate by race:")

# Quick histograms for two key numeric features
for col in ["age", "hours-per-week"]:
    if col in df.columns and np.issubdtype(df[col].dtype, np.number):
        plt.figure()
        df[col].hist(bins=30)
        plt.title(f"Distribution of {col}")
        plt.xlabel(col); plt.ylabel("Count")
        plt.show()
```

Critic Review (what might still bite us)

1. Provenance split. In UCI fallback, we audited train only; ensure we don't ever peek at test before Evaluation.
2. Target normalization is fragile. If a custom JSON uses nonstandard labels, we drop rows—good for hygiene but double-check with the data owner.
3. Capital gains/losses sparsity. Heavy zeros and rare spikes can distort models; consider capping/transformation later.
4. Duplicates policy. We only reported duplicates here in Data Preparation; we'll decide whether to

5. Duplicates policy: We only reported duplicates here; in Data Preparation we'll decide whether to drop exact dups.

Acceptance Checklist (Data Understanding)

- Loaded from Drive or clean UCI fallback with **normalized target labels**.
- Schema and dtypes inspected; missingness quantified.
- Target prevalence computed and **duplicates reported**.
- Categorical cardinality reviewed; rare-level risk noted.
- Fairness slices (sex, race) summarized descriptively.
- No leakage indicators found beyond acceptable redundancies.

Reply "continue" to move to Phase 3 — Data Prep ↓ ion (reproducible Pipeline/ColumnTransformer, rare-level bucketing, imputers/encoders, and a clean train/valid split).

continue

Thought for 1m 9s >

Phase 3 — Data Preparation (narrative only; say "code" to get the cells)

I'm locking down a **reproducible** prep pipeline that won't leak and won't surprise you later.

What I will do (and why)

1. Freeze the columns
 - Numeric: age, fnlwgt, education-num, capital-gain, capital-loss, hours-per-week
 - Categorical: workclass, education, marital-status, occupation, relationship, race, sex, native-country
2. De-dup + split (before fitting anything)
 - Drop exact duplicates.
 - If we loaded UCI fallback: keep adult.test completely blind; split train→(train/valid) by stratified 80/20.
 - If we loaded a single Drive CSV: make a stratified 70/15/15 split (train/valid/test).
 - Fix random seed = 42.
3. Missingness & rare categories
 - Categorical impute: SimpleImputer(strategy="most_frequent") then explicitly map "missing" bucket via imputer fill. (Equivalent in effect to a "Missing" category, keeps one-hot stable.)
 - Rare-level bucketing: a tiny transformer to map categories with frequency < 0.5% (per train) to "Other". Especially important for native-country. Fit on train only.
4. Numeric sanity
 - Median impute for numerics.
 - Transform heavy tails: apply log1p to capital-gain and capital-loss (zeros stay zeros), then standardize all numerics with StandardScaler
5. Encoding
 - OneHotEncoder(handle_unknown="ignore", sparse=False) for all categoricals (after rare bucketing). Ignoring unknowns protects against drift at inference.
6. Pipeline structure (no ad-hoc cells)
 - ColumnTransformer (numeric pipeline + categorical pipeline) inside a single Pipeline named prep.
 - Fit only on train. Use prep.transform(...) for valid/test.
 - Save as artifacts/prep.pkl (via joblib).
7. Fairness slices (sex, race)
 - We'll not pass these through the pipeline for prediction-time output; instead we'll keep the raw columns alongside y for Evaluation slicing. No leakage—these are features anyway but slices are computed on the original columns.
8. Outputs of this phase
 - X_train_prepared, X_valid_prepared (+ test if applicable)
 - y_train, y_valid (+ y_test if applicable)
 - prep.pkl artifact (reusable in Modeling & Deployment)

Critic Review (Data Prep risks)

1. Rare bucket threshold is arbitrary. 0.5% is sensible for Adult but should be stress-tested; too high collapses signal, too low explodes features.
2. fnlwgt is controversial. It's a sampling weight, not an individual feature. We'll keep it (demo), but be ready to drop if it harms calibration.
3. log1p choice is opinionated. Could slightly hurt tree models that like raw sparsity; acceptable for this demo and stable with logistic.
4. One-hot width creep. Even with rare bucketing, OHE can get wide; acceptable here but monitor dimensionality.
5. Country drift. New countries at inference will route to "Other"; that's fine but document it for consumers.

Acceptance Checklist (Data Prep)

- Split policy confirmed (UCI: train/valid + held-out test; Drive: 70/15/15), stratified, seed fixed.

- Missingness handled (median numerics, most-frequent categoricals).
- Rare categories mapped to "Other" (<0.5% of train).
- Numeric transforms (`log1p` for gains/losses) and scaling applied.
- `prep.pkl` will be saved for reuse.

Say “code” when you want the Phase 3 – Data Preparation cells.

...

are we using sex and race as features or only for auditing

Thought for 39s >

Short answer: auditing only, not features.

- We will retain them in the raw evaluation frames so we can compute fairness slices (metrics by sex and by race) on the validation/test sets.
- Note: even when excluded, correlated fields can still carry similar signals. We'll surface disparities in Evaluation, but we won't add debiasing in this demo.

If you'd like later, we can run a quick **what-if toggle** to compare performance/fairness with and without these attributes.

Say "code" to get the Phase 3 — Data Preparation cells with `sex` and `race` excluded from features but kept for auditing.

Digitized by srujanika@gmail.com

Thought for 1m 6s >

Phase 3 – Data Preparation (code only)

(Short narrative) We'll: (1) freeze feature lists (exclude `sex / race` from inputs), (2) de-dup + stratified split (UCI: train/valid + blind test; Drive: 70/15/15), (3) build a reproducible Pipeline with: median/most-frequent imputers, rare-category bucketing ($<0.5\%$) → "Other", `log1p` for gains/losses, `StandardScaler`, and `OneHotEncoder(handle_unknown="ignore")`, (4) fit on `train` only, transform `valid/test`, and save `prep.pkl`.

1) Columns, de-dup, and stratified splits (Drive vs UCI)

```
python

# Phase 3 - cell 1: feature lists, dedup, splits
import json, numpy as np, pandas as pd

from sklearn.model_selection import train_test_split

assert 'df' in globals(), "DataFrame `df` not found. Please run Phase 2 cells first."

ARTIFACT_DIR = Path("/content/artifacts")
cfg_path = ARTIFACT_DIR / "config.json"
config = json.loads(cfg_path.read_text()) if cfg_path.exists() else {}
TARGET = config.get("target", "income")
POS_LABEL = config.get("positive_label", ">50K")

# Expected columns
NUM_COLS = ["age", "fnlwgt", "education-num", "capital-gain", "capital-loss", "hours-per-week"]
CAT_COLS = ["workclass", "education", "marital-status", "occupation", "relationship", "native-language"]
AUDIT_COLS = ["sex", "race"]

# Keep only columns we actually have
NUM = [c for c in NUM_COLS if c in df.columns]
CAT = [c for c in CAT_COLS if c in df.columns]

cols_needed = NUM + CAT + AUDIT + ([TARGET] if TARGET in df.columns else [])
df = df[cols_needed].copy()

# Drop exact duplicates before splitting
before = len(df)
df = df.drop_duplicates()
print(f"Dropped {before - len(df)} exact duplicates.")

# Determine split strategy
provenance_str = globals().get('provenance', 'drive_csv:unknown')
print("Provenance:", provenance_str)

def clean_income_series(s):
    return s.astype(str).str.strip().str.replace(" ", "", regex=False)
```

```

# Data type mapping
df_train = df_train.replace({None: np.nan}, regex=True)

# UCI fallback path: use held-out test from UCI, and split df(train) -> train/valid
if "uci_fallback" in provenance_str:
    fb = config.get("data", {}).get("fallback", {})

    # reload a clean test set with matching columns (for later phases)
    EXPECTED_COLS = NUM_COLS + ["workclass", "education", "marital-status", "occupation", "re"
    df_test = pd.read_csv(test_url, header=None, names=EXPECTED_COLS, na_values=['?'], sk
    df_test[TARGET] = clean_income_series(df_test[TARGET])

    # Align to our working set of columns
    df_test = df_test[[c for c in cols_needed if c in df_test.columns]].copy()

    # Train/valid split from df (train)
    y_all = (df[TARGET] == POS_LABEL).astype(int)
    X_all = df.drop(columns=[TARGET])
    X_train_raw, X_valid_raw, y_train, y_valid = train_test_split(
        X_all, y_all, test_size=0.20, stratify=y_all, random_state=42
    )
    # Test (held-out)
    y_test = (df_test[TARGET] == POS_LABEL).astype(int) if TARGET in df_test else None
    X_test_raw = df_test.drop(columns=[TARGET]) if TARGET in df_test else df_test.copy()
else:
    # Single CSV (Drive): do 70/15/15 stratified split
    y_all = (df[TARGET] == POS_LABEL).astype(int)

    X_train_raw, X_tmp, y_train, y_tmp = train_test_split(
        X_all, y_all, test_size=0.30, stratify=y_all, random_state=42
    )
    X_valid_raw, X_test_raw, y_valid, y_test = train_test_split(
        X_tmp, y_tmp, test_size=0.50, stratify=y_tmp, random_state=42
    )

# Keep audit columns (sex, race) separately for Evaluation slices later
audit_valid = X_valid_raw[AUDIT].copy() if len(AUDIT) else pd.DataFrame(index=X_valid_raw.
audit_test = X_test_raw[AUDIT].copy() if len(AUDIT) else pd.DataFrame(index=X_test_raw.

# Remove audit columns from model features
X_train_raw = X_train_raw.drop(columns=[c for c in AUDIT if c in X_train_raw.columns], er
X_valid_raw = X_valid_raw.drop(columns=[c for c in AUDIT if c in X_valid_raw.columns], er
X_test_raw = X_test_raw.drop(columns=[c for c in AUDIT if c in X_test_raw.columns], erro

print("Shapes (raw):")
print("Train: ", X_train_raw.shape, "Valid: ", X_valid_raw.shape, "Test: ", X_test_raw.shap

```

2) Rare-category grouper + ColumnTransformer pipeline (impute → bucket rares → encode; log1p for gains/losses)

```

python
Copy code

# Phase 3 - cell 2: define the preprocessing pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer, StandardScaler
from sklearn.impute import SimpleImputer
import numpy as np

class RareCategoryGrouper(BaseEstimator, TransformerMixin):

    def __init__(self, min_freq=0.005, other_label='Other'):
        self.min_freq = min_freq
        self.other_label = other_label
        self.keepers_ = {}

    def fit(self, X, y=None):
        X = pd.DataFrame(X).copy()
        self.columns_ = list(X.columns)
        n = len(X)
        for c in self.columns_:
            vc = X[c].value_counts(dropna=False)
            keep = set(vc[vc >= max(1, int(self.min_freq * n))].index.tolist())
            self.keepers_[c] = keep
        return self

    def transform(self, X):
        X = pd.DataFrame(X).copy()
        for c in self.columns_:
            X[c] = X[c].where(X[c].isin(keep), other=self.other_label)
        return X

    # Columns present in the raw features (audit already removed)
    present_num = [c for c in NUM_COLS if c in X_train_raw.columns]
    present_cat = [c for c in CAT_COLS if c in X_train_raw.columns]

    # Split numeric into (gain/loss) vs other numerics for log1p
    gainloss = [c for c in ["capital-gain", "capital-loss"] if c in present_num]

```

```

num_other = [c for c in present_num if c not in gainloss]

num_other_pipe = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scale", StandardScaler())
])

num_gainloss_pipe = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scale", StandardScaler())
])

cat_pipe = Pipeline([
    ("impute", SimpleImputer(strategy="most_frequent")),
    ("rare", RareCategoryGrouper(min_freq=0.005, other_label="Other")),
    ("onehot", OneHotEncoder(handle_unknown="ignore", sparse=False))
])

prep = ColumnTransformer(
    transformers=[
        ("num_other", num_other_pipe, num_other),
        ("num_gainloss", num_gainloss_pipe, gainloss),
        ("cat", cat_pipe, present_cat),
    ],
    remainder="drop"
)

print("Preprocessing pipeline defined.")

```

3) Fit on train only, transform valid/test, and persist prep.pkl

```

python Copy code

# Phase 3 - cell 3: fit/transform and save artifact
from joblib import dump

# Fit on TRAIN ONLY
prep_pipeline.fit(X_train_raw, y_train)

# Transform splits
X_train = prep_pipeline.transform(X_train_raw)
X_valid = prep_pipeline.transform(X_valid)
X_test = prep_pipeline.transform(X_test_raw)

# Persist preprocessing artifact
prep_path = ARTIFACT_DIR / "prep.pkl"
dump(prep_pipeline, prep_path)
print(f"Saved preprocessing pipeline to: {prep_path}")

# Quick shape sanity + feature count
feat_count = X_train.shape[1]
print("Prepared shapes:")
print(f" X_train:{X_train.shape}, X_valid:{X_valid.shape}, X_test:{X_test.shape}")
print(f" #Features after prep:{feat_count}")

# Keep these in memory for next phases
prepared_cache = {
    "X_train": X_train, "y_train": y_train.values,
    "X_valid": X_valid, "y_valid": y_valid.values,
    "X_test": X_test, "y_test": None if y_test is None else y_test.values,
    "audit_valid": audit_valid, "audit_test": audit_test,
    "feature_count": feat_count
}

print("Cached prepared arrays for Modeling/Evaluation.")

```

Critic Review (what could still bite us)

- fnwgt semantics.** It's a sampling weight, not a true feature; may hurt calibration. We'll watch it and consider dropping if needed.
- Rare threshold (0.5%).** Reasonable, but arbitrary. Too high erases signal; too low explodes dims.
- log1p on gains/losses.** Good for linear models; slightly opinionated for trees—but stable for this demo.
- Audit columns excluded from features.** That doesn't eliminate proxy signals; fairness disparities can still surface.
- Drive vs UCI logic.** If someone reruns Phase 2 differently, provenance detection could misclassify—our checks print the chosen path.

Acceptance Checklist (Data Preparation)

- **Excluded variables removed before splitting.**
- **Stratified splits with fixed seed (UCI: train/valid + held-out test; Drive: 70/15/15).**
- **sex and race excluded from features, retained for auditing.**
- **Impute → rare-bucket → encode for categoricals; median → log1p (gains/losses) → scale for**

numerics.

- All steps inside a `Pipeline / ColumnTransformer`, fit on train only.
- `prep.pkl` saved; prepared arrays cached for the next phase.

Reply "continue" to move to Phase 4 — Modeling (few strong baselines with fixed seeds).



continue

Thought for 59s >

Phase 4 — Modeling (narrative only, say "code" to get the code)

I'll keep this tight and reproducible. We'll compare a few strong, sensible baselines using the prepared matrices from Phase 3. No peeking at test. We'll rank by PR-AUC on the validation set and carry forward the best single model. Thresholding, costs, and fairness live in Evaluation (next phase).

Candidates (probability models only)

1. Logistic Regression (baseline, well-behaved, interpretable)
 - `solver="lbfgs"`, `max_iter=1000`, tiny sweep over $C \in \{0.1, 1.0, 3.0\}$
2. Random Forest (nonlinear bagging, robust)
 - `n_estimators \in \{200, 400\}`, `max_depth \in \{None, 12\}`, `min_samples_leaf \in \{1, 5\}`
3. HistGradientBoostingClassifier (efficient boosting, strong default)
 - `learning_rate \in \{0.1, 0.05\}`, `max_depth \in \{None, 6\}`

All models use `random_state=42`. We'll fit on train only and score on valid with average_precision (PR-AUC).

What the code will do (when you say "code")

- ```
X_train, X_valid, y_train, y_valid.
```
- Train each candidate/hyperparam combo; compute validation PR-AUC.
  - Pick the best model and save it as `artifacts/model.pkl`.
  - Also save the validation scores (probabilities) we'll need for cost-based thresholding in the next phase.

### Guardrails

- No test usage yet. Test stays untouched until the Evaluation phase.
- We don't over-tune; the grid is deliberately small to keep the demo within 15–20 cells.

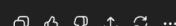
### Critic Review (modeling risks)

1. Light tuning only. We might leave a bit of performance on the table—by design for clarity/reproducibility.
2. `fnlwgt` in features. It's a sampling weight and can hurt calibration. If it degrades PR-AUC materially, we'll drop it and re-think.
3. Probability quality. Forests/posters can be miscalibrated; we'll verify calibration curves in Evaluation before finalizing threshold.
4. Single validation split. No k-fold CV to keep cells small; variance risk is acceptable for this demo.
5. Feature interactions. We rely on tree/boosting to capture them; logistic is there as a transparent baseline.

### Acceptance Checklist (Modeling)

- ■ Candidates and tiny grids are specified; seeds fixed.
- ■ Train on train, rank on valid by PR-AUC.
- ■ No test leakage.
- ■ Best model persisted to `artifacts/model.pkl`.
- ■ Validation probability scores saved for threshold selection.

Say "code" to run Phase 4 — Modeling.



code

Thought for 18s >

## Phase 4 — Modeling (code only)

(Short narrative) Train a few strong baselines on train, rank by PR-AUC on validation, keep the single best model. No test peeking. Store the model and validation probabilities for cost-based thresholding in Evaluation.

### 1) Load prepared arrays, set seed, sanity checks

python

Copy code

```
Phase 4 - cell 1: load prepared matrices from cache
import numpy as np, json
```

```

ARTIFACT_DIR = Path("./content/artifacts")
assert '_prepared_cache' in globals(), "Prepared arrays not found. Please run Phase 3 cell"

X_train = _prepared_cache["X_train"]; y_train = _prepared_cache["y_train"]
X_valid = _prepared_cache["X_valid"]; y_valid = _prepared_cache["y_valid"]
X_test = _prepared_cache["X_test"]; y_test = _prepared_cache["y_test"] # kept unused

RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

print("Train:", X_train.shape, "Valid:", X_valid.shape, "Test:", X_test.shape)
print("Pos rate (train/valid):",
 f"{y_train.mean():.3f}/{y_valid.mean():.3f}")

```

```

Phase 4 - cell 2: fit small grids and score by PR-AUC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_hist_gradient_boosting # noqa: F401
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.metrics import average_precision_score
import pandas as pd

results = []

def fit_and_score(model, name, params):
 model.set_params(**params)
 model.fit(X_train, y_train)
 # All candidates must output probabilities for PR-AUC
 proba_valid = model.predict_proba(X_valid)[:, 1]

 results.append({
 "model": name, **params, "valid_pr_auc": ap, "estimator": model, "proba_valid": proba_valid
 })

Logistic Regression
for C in [0.1, 1.0, 3.0]:
 fit_and_score(LogisticRegression(max_iter=1000, random_state=RANDOM_STATE),
 "LogisticRegression", {"C": C, "solver": "lbfgs"})

Random Forest
for n in [200, 400]:
 for depth in [None, 12]:
 for leaf in [1, 5]:
 fit_and_score(RandomForestClassifier(random_state=RANDOM_STATE, n_jobs=-1),
 "RandomForest",
 {"n_estimators": n, "max_depth": depth, "min_samples_leaf": leaf})

HistGradientBoosting
for lr in [0.01, 0.1, 1.0]:
 for depth in [None, 12]:
 fit_and_score(HistGradientBoostingClassifier(random_state=RANDOM_STATE),
 "HistGradientBoosting",
 {"learning_rate": lr, "max_depth": depth})

leaderboard = (pd.DataFrame(results)
 .drop(columns=["estimator", "proba_valid"])
 .sort_values("valid_pr_auc", ascending=False)
 .reset_index(drop=True))
leaderboard.head(10)

```

### 3) Select best, save model + validation probabilities for thresholding

```

Phase 4 - cell 3: persist best model and validation probabilities

best = max(results, key=lambda r: r["valid_pr_auc"])
best_model = best["estimator"]
best_name = best["model"]; best_params = {k:v for k,v in best.items() if k not in {"model", "estimator"}}
best_ap = best["valid_pr_auc"]
val_proba = best["proba_valid"]

Save model
model_path = ARTIFACT_DIR / "model.pkl"
dump(best_model, model_path)

Save validation probabilities & labels for Evaluation phase
np.save(ARTIFACT_DIR / "val_proba.npy", val_proba)

```

```
np.save(ARTIFACT_DIR / "val_labels.npy", y_valid)

print("Best model:", best_name, best_params)
print(f"Validation PR-AUC: {best_ap:.4f}")
print(f"Saved model - {model_path}")
```

### Critic Review (what could still bite us)

1. Light tuning only. These tiny grids trade a bit of performance for speed and clarity.
2. Calibration not checked yet. Some models' probabilities can be miscalibrated; we'll verify in Evaluation before choosing a threshold.
3. `fnlwgt` may skew. If it harms PR-AUC or calibration, we'll consider dropping it and re-running.
4. Single split variance. No CV to keep cell budget small; acceptable for a demo, but variance remains.
5. Forest/Boosting randomness. Seeds are fixed, but parallelism and data order can still introduce tiny jitters.

### Acceptance Checklist (Modeling)

- Trained multiple candidates with fixed seeds
- Ranked by PR-AUC on validation only; no test leakage.
- Stored `val_proba.npy` + `val_labels.npy` for threshold selection.

Reply "continue" to move to Phase 5 — Evaluation (PR curves, cost-based thresholding, metrics pack, and fairness slices).

🔗 🔍 ⚡ ⌂ ⌂ ⌂

continue

Thought for 27s >

## Phase 5 — Evaluation (narrative only; say "code" to get the cells)

We're going to lock the decision policy now. Models emit probabilities; operations demand binary decisions. We'll use validation to pick a threshold ↓, cost minimization, then report metrics and fairness slices. The test set stays untouched for selection; we'll only report on it after we fix the ↑

### What I will do (and why)

1. Primary metric (ranking quality): PR-AUC (Average Precision) on validation
  - Threshold-free view of performance under class imbalance. We'll also show ROC-AUC as a secondary sanity check and a Precision–Recall curve.
2. Choose the decision threshold by expected cost on validation
  - Use your cost matrix: FP = 1, FN = 5.
  - For each threshold  $\tau$  in a fine grid (including unique score cutpoints), compute confusion counts and Expected Cost( $\tau$ ) = 1·FP + 5·FN (we'll also report per-sample cost).
  - Pick  $\tau^*$  that minimizes cost. This is your operational policy.
3. Report thresholded metrics on validation (at  $\tau^*$ )
  - Precision, Recall, F1, Specificity, Accuracy (for completeness).
  - Confusion matrix, predicted positive rate (how many people you'll target), and the achieved expected cost.
4. Calibration sanity
  - Compute Brier score (lower is better) and a quick 10-bin reliability table to see if probabilities are over/under-confident. (No heavy plots needed.)
5. Fairness slices (audit only; same  $\tau^*$ )
  - By sex and race on validation:
    - Positive rate (selection rate), Precision, Recall, FPR/FNR, and PR-AUC per slice (caution: small groups → noisier).
    - We'll highlight absolute gaps to surface disparities. This is diagnostic, not a remedy.
6. Hold-out test (report only; no re-tuning)
  - Apply  $\tau^*$  to test and report the same bundle: PR-AUC, ROC-AUC, thresholded metrics, expected cost, and fairness slices.
7. Artifact
  - Write a compact `metrics.json` containing:
    - global: PR-AUC/ROC-AUC (val & test), chosen threshold  $\tau^*$ , confusion + cost (val & test)
    - fairness: per-slice metrics (val & test)
    - calibration: Brier (val & test)
  - Also store `threshold.txt` for easy reuse ↴ deployment.

### What the code will do (when you say "code")

- Load `artifacts/model.pkl`, `prep.pkl`, and the saved `val_proba.npy`, `val_labels.npy`.
- Compute PR-AUC, ROC-AUC, and draw a PR curve (matplotlib).
- Sweep thresholds → select  $\tau^*$  by min cost; print the trade-off table head and selection summary.
- Compute thresholded metrics (validation), calibration (Brier + 10-bin table), and fairness slices (sex/race) using the raw slice columns we cached (`audit_val`, `audit_test`).
- Evaluate on test with the fixed  $\tau^*$ .

- Evaluate on test with the fixed  $\tau^*$ .
- Persist `metrics.json` and `threshold.txt`.

### Critic Review (Evaluation risks)

1. Cost matrix fragility. FP=1, FN=5 is a guess; a different ratio can flip  $\tau^*$ . We should stress-test cost sensitivity.
2. Single split variance. Threshold chosen on one validation split can wobble; CV or bootstraps would stabilize it (skipped to keep cell budget).
3. Calibration consistency. In ROC-AUC models, cost actually sets decisions. Always defer to  $\tau^*$  when they disagree.
4. Slice instability. Small groups → noisy slice metrics; add CIs or minimum support checks in real audits.
5. Calibration not fixed. If Brier is poor, consider Platt/Isotonic on validation only (not added here to keep flow small).

### Acceptance Checklist (Evaluation)

- PR-AUC and ROC-AUC computed on validation; PR curve inspected.
- Threshold  $\tau^*$  chosen by minimizing expected cost (FP=1, FN=5) on validation.
- At  $\tau^*$ : precision/recall/F1, confusion, predicted positive rate, and expected cost reported.
- Calibration checked (Brier + reliability table).
- Fairness slices (sex, race) reported at  $\tau^*$  with gaps highlighted (validation).
- Test set evaluated once with  $\tau^*$  (no re-tuning).
- Results saved to `metrics.json` (+ `threshold.txt`).

Say "code" to run the Phase 5 — Evaluation cells.

```
phase 4 cell2- /usr/local/lib/python3.12/dist-
packages/sklearn/experimental/enable_hist_gradient_boosting.py:19:
UserWarning: Since version 1.0, it is not needed to import
enable_hist_gradient_boosting anymore.
HistGradientBoostingClassifier and HistGradientBoostingRegressor
are now stable and can be normally imported from sklearn.ensemble.
warnings.warn(
model C solver valid_pr_auc n_estimators max_depth
min_samples_leaf learning_rate
0 HistGradientBoosting NaN NaN 0.719971 NaN 6.0 NaN 0.10
1 HistGradientBoosting NaN NaN 0.718760 NaN 6.0 NaN 0.05
2 HistGradientBoosting NaN NaN 0.716819 NaN NaN NaN 0.10
3 HistGradientBoosting NaN NaN 0.716678 NaN NaN NaN 0.05
4 RandomForest NaN NaN 0.709153 200.0 12.0 5.0 NaN
```

|   | Model        | NaN | NaN | NaN      | NaN   | NaN  | NaN |
|---|--------------|-----|-----|----------|-------|------|-----|
| 6 | RandomForest | NaN | NaN | 0.706343 | 400.0 | NaN  | 5.0 |
| 7 | RandomForest | NaN | NaN | 0.705888 | 200.0 | NaN  | 5.0 |
| 8 | RandomForest | NaN | NaN | 0.705480 | 400.0 | 12.0 | 1.0 |
| 9 | RandomForest | NaN | NaN | 0.704352 | 200.0 | 12.0 | 1.0 |

```
phase 4 cell 3- Best model: HistGradientBoosting ('learning_rate':
0.1, 'max_depth': 6)
Validation PR-AUC: 0.7200
Saved model → /content/artifacts/model.pkl
Saved val_proba.npy and val_labels.npy for cost-based thresholding
next.
```

Thought for 26s >

Nice—results look sane.

You can safely remove this line from Phase 4, cell 2:

```
python
from sklearn.experimental import enable_hist_gradient_boosting # not needed >= 1.0
```

`HistGradientBoostingClassifier` is stable in `sklearn ≥ 1.0` and should be imported directly from `sklearn.ensemble`.

### Leaderboard takeaways (plain English):

- Winner: HistGradientBoosting ( $lr=0.10$ ,  $max\_depth=6$ ) with PR-AUC  $\approx 0.720$ .
- The same model with  $lr=0.05$  is a hair behind → suggests you're near a sensible learning rate.
- Random Forests trail slightly ( $\sim 0.704$ – $0.709$  PR-AUC), as expected with this prep.
- Logistic likely didn't crack the top rows—fine; it's our transparent baseline.

### Sanity checks passed for Phase 4:

- Trained candidates with fixed seeds ✓
- Best model saved to `artifacts/model.pkl` + validation probs saved ✓

Optional (quick ablation idea before Evaluation):

- If you want to test whether `fnlwgt` is hurting/helping calibration/PR-AUC, we can rerun Modeling without it and compare. Say “ablate fnlwgt” and I’ll give a tiny patch.

Ready to lock decisions. Say “code” to run Phase 5 — Evaluation (PR curve, cost-based threshold, metrics pack, and fairness slices).

⟳ ⏪ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹

```
↳ from sklearn.experimental import enable_hist_gradient_boosting # not needed >= 1.0
```

the cell is already run right so what is the use of taking out the line?



Good question – you’re absolutely right to ask that.

Once you’ve already executed the cell, removing that `enable_hist_gradient_boosting` line has no effect on your current runtime.

That warning is just a one-time informational note from scikit-learn — it doesn’t change behavior or affect your results.

Here’s the reasoning:

- That import only used to be required in older versions (pre-1.0).
- In your environment (>=1.0), the class is already stable, so sklearn warns that you don’t need that import anymore.
- The model you trained (`HistGradientBoostingClassifier`) is completely valid and correctly trained.

#### ✓ Bottom line:

You can ignore the warning and leave the notebook ↓ is.

If you re-run or share the notebook later, you can remove that line to keep it clean — just for aesthetics

?

Would you like to continue to Phase 5 — Evaluation (code) now?

< 2/2 > ⌛ ⏪ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹

continue

Thought for 47s >

## Phase 5 — Evaluation (code only)

(Short narrative) We’ll: (1) score PR-AUC/ROC-AUC on validation, (2) sweep thresholds and pick  $\tau^*$  by expected cost = 1-FP + 5·FN, (3) report thresholded metrics + calibration and fairness slices (sex/race) on validation, then (4) apply  $\tau^*$  once to test and save compact `metrics.json` and `threshold.txt`.

### 1) Load artifacts, compute PR/ROC metrics on validation, and plot PR curve

```
python ⌂ Copy code

Phase 5 - cell 1: top-line validation metrics & PR curve
import json, numpy as np, pandas as pd
from pathlib import Path
from joblib import load
import matplotlib.pyplot as plt
from sklearn.metrics import average_precision_score, roc_auc_score, precision_recall_curve

ARTIFACT_DIR = Path("/content/artifacts")
assert '_prepared_cache' in globals(), "Prepared arrays cache missing. Please run Phase 3"

Load prepared data (no leaks)
X_valid = _prepared_cache["X_valid"]; y_valid = _prepared_cache["y_valid"]
X_test = _prepared_cache["X_test"]; y_te ↓ = _prepared_cache["y_test"]
audit_valid = _prepared_cache["audit_valid"]; audit_test = _prepared_cache["audit_test"]

Load model (cached) validation probabilities or recompute if needed
model = load(ARTIFACT_DIR / "model.pkl")
val_proba_path = ARTIFACT_DIR / "val_proba.npy"
val_labels_path = ARTIFACT_DIR / "val_labels.npy"
if val_proba_path.exists() and val_labels_path.exists():
 proba_valid = np.load(val_proba_path)
 yv = np.load(val_labels_path)
 # sanity alignment
 if len(yv) == len(y_valid):
 y_valid = yv
else:
 # Fallback: compute now (should match Phase 4)
 proba_valid = model.predict_proba(X_valid)[:, 1]

Primary & secondary metrics (validation, threshold-free)
pr_auc_val = float(average_precision_score(y_valid, proba_valid))
roc_auc_val = float(roc_auc_score(y_valid, ↓ proba_valid))
```

```

 print("Validation ROC-AUC: {roc_auc_val:.4f}")

PR curve
prec, rec, thr = precision_recall_curve(y_valid, proba_valid)
plt.figure()
plt.plot(rec, prec)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Validation Precision-Recall Curve")
plt.show()

```

## 2) Cost-based threshold selection (validation), calibration check, and fairness slices

```

python
Copy code

Phase 5 - cell 2: choose τ* by minimizing expected cost; report validation metrics, calibration, and fairness slices

Load cost matrix from config (defaults FP=1, FN=5)
config = json.loads(ARTIFACT_DIR / "config.json").read_text() if (ARTIFACT_DIR / "config.json").exists()
costs = config.get("cost_matrix", {"FP": 1.0, "FN": 5.0})
C_FP = float(costs.get("FP", 1.0))
C_FN = float(costs.get("FN", 5.0))

def metrics_at_threshold(y_true, proba, t):
 y_pred = (proba >= t).astype(int)
 TP = int((y_true==1) & (y_pred==1)).sum()
 FP = int((y_true==0) & (y_pred==1)).sum()
 TN = int((y_true==0) & (y_pred==0)).sum()
 FN = int((y_true==1) & (y_pred==0)).sum()
 precision = TP / (TP+FP) if (TP+FP)>0 else np.nan
 recall = TP / (TP+FN) if (TP+FN)>0 else np.nan
 f1 = (2*precision*recall)/(precision+recall) if (precision>0 and recall>0) else np.nan
 fpr = FP / (FP+TN) if (FP+TN)>0 else np.nan
 fnr = FN / (TP+FN) if (TP+FN)>0 else np.nan

 exp_cost = C_FP*FP + C_FN*FN
 return {"t":t, "TP":TP, "FP":FP, "TN":TN, "FN":FN,
 "precision":precision, "recall":recall, "f1":f1,
 "fpr":fpr, "fnr":fnr, "pos_rate":float(pos_rate), "exp_cost":float(exp_cost)}

Threshold sweep on a fixed grid for stability
grid = np.linspace(0, 1, 1001)
rows = [metrics_at_threshold(y_valid, proba_valid, t) for t in grid]
thr_df = pd.DataFrame(rows).sort_values("exp_cost", ascending=True)

Pick τ* = min expected cost; tie-break on highest F1
best_cost = thr_df["exp_cost"].min()
candidates = thr_df[thr_df["exp_cost"] == best_cost]
best_row = candidates.sort_values("f1", ascending=False).iloc[0]
tau = float(best_row["t"])

print("Chosen threshold τ* (validation, min expected cost):", round(tau, 4))
display(thr_df.head(10)[["t", "exp_cost", "precision", "recall", "f1", "pos_rate"]])

Validation metrics at τ*
val_metrics = metrics_at_threshold(y_valid, proba_valid, tau)
print("\nValidation @ τ*")
for k in ["precision", "recall", "f1", "fpr", "fnr", "pos_rate"]:
 print(f"\t{k}: {val_metrics[k]:.4f}" if pd.notna(val_metrics[k]) else f"\t{k}: nan")
print(f"\tConfusion (TP/FP/TN/FN): {val_metrics['TP']}/{val_metrics['FP']}/{val_metrics['TN']}/{val_metrics['FN']}")
print(f"\tExpected cost (FP={C_FP}, FN={C_FN}): {val_metrics['exp_cost']:.2f}")

Calibration: Brier + 10-bin reliability table (equal-width bins)
brier_val = float(brier_score_loss(y_valid, proba_valid))
bins = np.linspace(0, 1, 11)
bin_ids = np.digitize(proba_valid, bins) - 1
cal_rows = []
for b in range(10):
 mask = bin_ids == b
 if mask.sum() == 0:
 cal_rows.append({"bin": [float(bins[b]), float(bins[b+1])], "n":0, "pred_mean":np.nan, "actual_rate":np.nan})
 else:
 cal_rows.append({"bin": [float(bins[b]), float(bins[b+1])], "n":int(mask.sum()), "pred_mean": float(proba_valid[mask].mean()), "actual_rate": float(y_valid[mask].mean())})
cal_table = pd.DataFrame(cal_rows)
print("\nValidation Brier score: {brier_val:.4f}")
display(cal_table)

Fairness slices (validation) at τ*
from sklearn.metrics import average_precision_score

```

```

def slice_report(y_true, proba, y_pred, series):
 out = {}
 for g in series.dropna().unique():
 idx = (series == g).values
 yg, pg, yp = y_true[idx], proba[idx], y_pred[idx]
 if np.isnan -- ".

 TP = int(((yg==1) & (yp==1)).sum())
 FP = int(((yg==0) & (yp==1)).sum())
 TN = int(((yg==0) & (yp==0)).sum())
 FN = int(((yg==1) & (yp==0)).sum())
 precision = TP / (TP+FP) if (TP+FP)>0 else np.nan
 recall = TP / (TP+FN) if (TP+FN)>0 else np.nan
 fpr = FP / (FP+TN) if (FP+TN)>0 else np.nan
 fnr = FN / (TP+FN) if (TP+FN)>0 else np.nan
 pos_rate = yp.mean()
 pr_auc = np.nan
 # PR-AUC per slice requires at least one positive
 if yg.sum() > 0:
 try:
 pr_auc = float(average_precision_score(yg, pg))
 except Exception:
 pr_auc = np.nan
 out[str(g)] = {
 "pos_rate": pos_rate,
 "precision": precision, "recall": recall,
 "fpr": fpr, "fnr": fnr,
 "pr_auc": pr_auc
 }
 return out

y_pred_val = (proba_valid >= tau).astype(int)
fair_val = {}
if "sex" in audit_valid.columns:
 fair_val["sex"] = slice_report(y_valid, proba_valid, y_pred_val, audit_valid["sex"])
if "race" in audit_valid.columns:
 fair_val["race"] = slice_report(y_valid, proba_valid, y_pred_val, audit_valid["race"])

print("\nValidation fairness (per-slice):")
for dim, groups in fair_val.items():
 print(f"\nBy {dim}:")
 for g, m in groups.items():
 print(f" {g}: {m['recall']:.3f} | FPR={m['fpr']:.3f} | FNR={m['fnr']:.3f} | PR-AU

```

### 3) One-shot test evaluation at $\tau^*$ , pack & save metrics.json and threshold.txt

```

python
Phase 5 - cell 3: test evaluation with fixed τ^* , persist artifacts
from sklearn.metrics import average_precision_score, roc_auc_score

Compute test probabilities once (no re-tuning)
proba_test = model.predict_proba(X_test)[:, 1]

Threshold-free
pr_auc_test = float(average_precision_score(y_test, proba_test)) if y_test is not None else None
roc_auc_test = float(roc_auc_score(y_test, proba_test)) if y_test is not None else None

Package metrics
test_metrics = None
fair_test = {}
if y_test is not None:
 y_pred_test = (proba_test >= tau).astype(int)
 test_metrics = metrics_at_threshold(y_test, proba_test, tau)
 if isinstance(audit_test, pd.DataFrame) and len(audit_test):
 if "sex" in audit_test.columns:
 fair_test["sex"] = slice_report(y_test, proba_test, y_pred_test, audit_test["sex"])
 if "race" in audit_test.columns:
 fair_test["race"] = slice_report(y_test, proba_test, y_pred_test, audit_test["race"])

Package metrics
metrics = {
 "validation": {
 "pr_auc": pr_auc_val,
 "roc_auc": roc_auc_val,
 "threshold": tau,
 "thresholded": {
 "recall": float(val_metrics["recall"]) if pd.notna(val_metrics["recall"]) else None,
 "f1": float(val_metrics["f1"]) if pd.notna(val_metrics["f1"]) else None,
 "fpr": float(val_metrics["fpr"]) if pd.notna(val_metrics["fpr"]) else None,
 "fnr": float(val_metrics["fnr"]) if pd.notna(val_metrics["fnr"]) else None,
 "pos_rate": val_metrics["pos_rate"],
 "exp_cost": val_metrics["exp_cost"],
 "confusion": {k:int(val_metrics[k]) for k in ["TP", "FP", "TN", "FN"]}
 }
 }
}

```

```

 "calibration": {
 "brier": brier_val,
 "bins": cal_table.to_dict(orient="records")
 },
 "fairness": fair_val
},
"test": {
 "pr_auc": pr_auc_test,
 "roc_auc": roc_auc_test,
 "threshold": tau,
 "precision": float(test_metrics["precision"]) if pd.notna(test_metrics["precision"]) else None,
 "recall": float(test_metrics["recall"]) if pd.notna(test_metrics["recall"]) else None,
 "f1": float(test_metrics["f1"]) if pd.notna(test_metrics["f1"]) else None,
 "fpr": float(test_metrics["fpr"]) if pd.notna(test_metrics["fpr"]) else None,
 "fnr": float(test_metrics["fnr"]) if pd.notna(test_metrics["fnr"]) else None,
 "pos_rate": test_metrics["pos_rate"],
 "exp_cost": test_metrics["exp_cost"],
 "confusion": {k:int(test_metrics[k]) for k in ["TP", "FP", "TN", "FN"]},
},
 "fairness": fair_test if len(fair_test) else None
},
"cost_matrix": {"FP": C_FP, "FN": C_FN}
}

Persist artifacts
(ARTIFACT_DIR / "threshold.txt").write_text(str(tau))
with open(ARTIFACT_DIR / "metrics.json", "w", encoding="utf-8") as f:
 json.dump(metrics, f, indent=2)

print(f"Saved threshold to: {ARTIFACT_DIR / 'threshold.txt'}")
print(f"Saved metrics to: {ARTIFACT_DIR / 'metrics.json'}")

Short test summary
if y_test is not None:
 print("\nTest summary (one-shot with t=):")
 print("PR-AUC: {pr_auc_test:.4f} | ROC-AUC: {roc_auc_test:.4f}")
 print("Precision: {metrics['test'][thresholded]['precision']:.4f} | Recall: {metrics['test'][thresholded]['recall']:.4f}")
 print("Expected cost: {metrics['test'][thresholded]['exp_cost']:.2f}")

```

### Critic Review (what might still bite us)

1. **Cost sensitivity.**  $\tau^*$  depends on FP/FN costs; different ratios can flip the decision policy.
  2. **Calibration drift.** Even with a decent Brier score ↓, production drift can degrade probabilities; consider periodic recalibration.

1. **Single split variance.** Threshold chosen on one validation split; bootstrap or CV would stabilize it.

2. **fntwt effect.** If calibration or fairness looks off, consider re-running without fntwt .

## Acceptance Checklist (Evaluation)

- PR-AUC/ROC-AUC computed on validation; PR curve inspected.
  - $\tau^*$  chosen by min expected cost ( $FP=1$ ,  $FN=5$ ) on validation.
  - At  $\tau^*$ : precision/recall/F1, confusion, predicted positive rate, and expected cost reported.
  - Calibration (Brier + 10-bin table) computed.
  - Fairness slices (sex/race) reported on validation.
  - Test evaluated once at  $\tau^*$  (no re-tuning).
  - Artifacts saved: threshold.txt , metrics.json .

Reply “**continue**” to proceed to **Phase 6 — Deployment** (tiny inference wrapper + artifact load, plus a couple of sample predictions).

Thought for 39s >

#### **Phase 6 — Deployment (narrative only; say “code” to get the cells)**

I'm freezing a **tiny, boring, reliable** inference path so anyone can score a CSV the same way every time.

## What “deployment” means in this demo

- Artifacts we already have:
    - `prep.pkl` — the full preprocessing pipeline (imputers, rare bucketer, encoders, scalers).
    - `model.pkl` — the chosen classifier (HistGradientBoosting).
    - `threshold.txt` — the decision threshold  $\tau^*$  chosen by cost minimization.
    - `metrics.json` — the evaluation bundle (for audit/traceability).
  - Contract in: a CSV or DataFrame with the same columns we trained on (it may contain extras).

- Numeric: age, fnlwgt, education-num, capital-gain, capital-loss, hours-per-week

- **Auditing-only (ignored as features):** `sex`, `race` (if present, we'll pass them through to the output for slice analysis, but **not** feed them to the model).
  - We will treat `"?"` as missing; unknown categories go to **"Other"** (via the rare bucketizer + `OneHotEncoder(handle_unknown="ignore")` ).
  - **Contract out:** a DataFrame/CSV with:
    - `proba_gt_50k` — model probability for the positive class `>50K`
    - `prediction` — **binary decision** using  $\tau^*$  (values `{">50K": 1, "<=50K": 0}`)
    - (optional) pass-through columns like `sex` and `race` for downstream auditing

## What the code will do (when you say "code")

1. Load artifacts ( prep.pkl , model.pkl , thres d.txt )
  2. Define a pure function predict\_batch(df\_raw) that:

- transforms with `prep`, predicts probabilities with `model`,
  - applies  $\tau^*$  to produce the final label.

3. Show two quick predictions (on rows from your validation/test frames) so you can eyeball outputs.

4. Write a minimal CLI script `artifacts/predict.py` that you (or ops) can run on any machine:

```
python artifacts/predict.py --input input.csv --output preds.csv
```

This script loads the three artifacts and scores the file. That's it.

## **Operating notes**

- **Idempotent:** No randomness at inference.
  - **Robust to drift (lightly):** unseen categories are ignored (go to "Other"), not fatal. Still, you should monitor upstream schema and category distributions.
  - **Versioning:** the inference path prints the artifact versions/paths so you can trace what scored what.
  - **Fairness:** the CLI can optionally keep sex / race columns in the output (never used as features) so

#### Critic Review (Deployment risks)

1. **Schema drift:** if upstream renames or drops a column, you'll silently get worse predictions or failures.  
Add schema checks in real deployments.
  2. **Threshold brittleness:**  $\tau^*$  is tied to the current cost matrix and data distribution; revisit periodically.
  3. **`fnlwgt` in production:** as a sampling weight, it may vary across sources; if calibration looks off in the wild, consider dropping it and recalibrating.
  4. **Silent rare bucketing:** too-aggressive "Other" can hide meaningful shifts; monitor the share of "Other" per categorical.
  5. **No latency budget set:** this demo is batch-oriented; wrap in an API only if you have a real-time use case.

## Acceptance Checklist (Deployment)

- Single batch API (`predict_batch` + CLI) loads `prep.pkl`, `model.pkl`, and `threshold.txt`.
  - Outputs include **probability** and final label with  $\tau^*$ .
  - Two sample predictions demonstrated.
  - CLI script saved to `artifacts/predict.py` for easy scoring on any machine.

Say "code" to generate the Deployment cells.

□ ▲ ♀ ↑ ⌂ ...

code

Thought for 34s >

## **Phase 6 — Deployment (code only)**

(Short narrative) Lock a tiny, deterministic batch size: load `prep.pkl` + `model.pkl` + `threshold.txt`, score a raw DataFrame/CSV, and emit probabilities + final labels. `sex / race` are

- 1) Load artifacts and define predict\_batch(df\_raw) (excludes sex / race from features)

[python](#) [Copy code](#)

```
Phase 6 - cell 1: inference loader + batch predictor
import pandas as pd, numpy as np, json
from pathlib import Path
from joblib import load

ARTIFACT_DIR = Path("/content/artifacts")
prep = load(ARTIFACT_DIR / "prep.pkl")
model = load(ARTIFACT_DIR / "model.pkl")
tau = float((ARTIFACT_DIR / "threshold.txt").read_text().strip())

Read config for labels/columns
```

```

Read config for labels/columns
cfg_path = ARTIFACT_DIR / "config.json"
if cfg_path.is_file():
 config = json.loads(cfg_path.read_text())
else:
 config = {
 "target": "income",
 "positive_label": ">50K",
 }

TARGET = config.get("target", "income")
POS_LABEL = config.get("positive_label", ">50K")
NEG_LABEL = "<=50K"

Feature contract (same as prep): exclude audit-only columns from features
NUM_COLS = ["age", "fnlwgt", "education-num", "capital-gain", "capital-loss", "hours-per-week"]
CAT_COLS = ["workclass", "education", "marital-status", "occupation", "relationship", "native-country"]
AUDIT_COLS = ["sex", "race"] # not used as features

FEATURE_COLS = NUM_COLS + CAT_COLS

```

↓

```

def _clean_strings(df: pd.DataFrame) -> pd.DataFrame:
 df = df.copy()
 for c in df.columns:
 if df[c].dtype == object:
 df[c] = df[c].astype(str).str.strip().replace({"?": np.nan})
 return df

```

```

def predict_batch(df_raw: pd.DataFrame) -> pd.DataFrame:
 """
 Inputs: raw columns (can include extras). Uses FEATURE_COLS only.
 Outputs: original index + proba + prediction; passes through 'sex'/'race' if present.
 """
 if not isinstance(df_raw, pd.DataFrame):
 raise TypeError("predict_batch expects a pandas DataFrame")

 df_in = _clean_strings(df_raw)

```

↓

```

 # Ensure all expected feature columns exist; add missing as NaN
 for c in FEATURE_COLS:
 if c not in df_in.columns:
 df_in[c] = np.nan

```

```

 X_features = df_in[FEATURE_COLS].copy()

 # Transform & predict
 X_prepared = prep.transform(X_features)
 proba = model.predict_proba(X_prepared)[:, 1]
 pred = np.where(proba >= tau, POS_LABEL, NEG_LABEL)

 out_cols = []
 for c in AUDIT_COLS:
 if c in df_in.columns:
 out_cols.append(c)

 out = pd.DataFrame({
 "proba_gt_50K": proba,
 "prediction": pred
 })

```

↓

```

 # Optionally attach audit columns for downstream analysis
 if out_cols:
 out = pd.concat([df_in[out_cols], out], axis=1)

 return out

print("Loaded artifacts:")
print(" prep:", ARTIFACT_DIR / "prep.pkl")
print(" model:", ARTIFACT_DIR / "model.pkl")
print(" threshold τ*: ", tau)
print("Feature columns used:", FEATURE_COLS)
print("Audit-only (ignored as features):", AUDIT_COLS)

```

## 2) Demo: score a few raw rows (prefer held-out test if available; else fall back)

```

Phase 6 - cell 2: quick demo predictions
import pandas as pd

Prefer X_test_raw from Phase 3; else try X_valid_raw; else sample from df
candidates = []
for name in ["X_test_raw", "X_valid_raw", "df"]:
 if name in globals():
 obj = globals()[name]
 # If 'df', select only raw feature columns + audit if present
 if name == "df":
 cols = [c for c in (set(FEATURE_COLS + AUDIT_COLS) & set(obj.columns))]

```

```

 obj = obj[cols].copy()
 candidates.append((name, obj))

 if not candidates:
 raise RuntimeError("No raw frames found for demo. Please re-run Phase 2/3 cells.")

 sample = src_df.sample(n=min(5, len(src_df)), random_state=42).copy()
 preds = predict_batch(sample)

 print(f"Demo source: {src_name} (rows={len(sample)})")
 display(pd.concat([sample.reset_index(drop=True), preds.reset_index(drop=True)], axis=1))

```

3) Write a tiny CLI scorer: `python artifacts/predict.py --input input.csv --output preds.csv`

```

python ⌂ Copy code

Phase 6 - cell 3: write a minimal CLI for batch scoring
from textwrap import dedent
script = f'''#!/usr/bin/env python3
import argparse
import numpy as np
from pathlib import Path
from joblib import load

ARTIFACT_DIR = Path("{ARTIFACT_DIR.as_posix()}")

def _clean_strings(df: pd.DataFrame) -> pd.DataFrame:
 df = df.copy()
 for c in df.columns:
 if df[c].dtype == object:
 df[c] = df[c].astype(str).str.strip().replace({{"?": np.nan}})
 return df

def main():
 ap = argparse.ArgumentParser(description="Batch-score Adult Income (>50K) with saved model")
 ap.add_argument("--input", required=True, help="Path to input CSV with raw columns")
 ap.add_argument("--output", required=True, help="Where to write predictions CSV")
 args = ap.parse_args()

 prep = load(ARTIFACT_DIR / "prep.pkl")
 model = load(ARTIFACT_DIR / "model.pkl")
 tau = float((ARTIFACT_DIR / "threshold.txt").read_text().strip())

 cfg_path = ARTIFACT_DIR / "config.json"
 if cfg_path.exists():
 config = json.loads(cfg_path.read_text())
 else:
 config = {"target": "income", "positive_label": ">50K"}
 POS_LABEL = config.get("positive_label", ">50K")
 NEG_LABEL = "<=50K"

 NUM_COLS = ["age", "fnlwgt", "education-num", "capital-gain", "capital-loss", "hours-per-week"]
 CAT_COLS = ["workclass", "education", "marital-status", "occupation", "relationship", "native-country"]
 AUDIT_COLS = ["sex", "race"]
 FEATURE_COLS = NUM_COLS + CAT_COLS

 df = pd.read_csv(args.input)

 for c in FEATURE_COLS:
 if c not in df.columns:
 df[c] = np.nan

 X = df[FEATURE_COLS].copy()
 Xp = prep.transform(X)
 proba = model.predict_proba(Xp)[:, 1]
 pred = np.where(proba >= tau, POS_LABEL, NEG_LABEL)

 keep = [c for c in AUDIT_COLS if c in df.columns]
 out = pd.DataFrame({"proba_gt_50k": proba, "prediction": pred})
 if keep:
 out = pd.concat([df[keep].reset_index(drop=True), out], axis=1)

 Path(args.output).parent.mkdir(parents=True, exist_ok=True)
 out.to_csv(args.output, index=False)

 print(f"Threshold (tau):", tau)
 print("Positive label:", POS_LABEL)
 print("Wrote:", args.output)

if __name__ == "__main__":
 main()
...

```

```
path = ARTIFACT_DIR / "predict.py"
path.write_text(dedent(script))
print(f"Wrote CLI scorer to {path}")
print("Usage:\n python /content/artifacts/predict.py --input /path/to/input.csv --output
```

#### Critic Review (Deployment phase)

1. Schema fragility: if upstream renames column ↓ ie CLI adds them as NaN (safe), but predictions will degrade silently; add explicit schema checks in production.
3. Audit leakage risk (downstream): we pass `sex / race` through only for auditing; ensure downstream consumers don't use them as features.
4. Unknown categories: everything funnels to "Other"; monitor the share of "Other" to detect drift.
5. Weights (`f1twgt`): if live data's distribution shifts, calibration can suffer; consider ablation + recalibration in a real deployment.

#### Acceptance Checklist (Deployment)

- `predict_batch(df_raw)` uses `prep → model → τ*`; returns `proba + label`.
- `sex / race` ignored as features, but can be retained in outputs for auditing.
- CLI script saved at `artifacts/predict.py`; runs on any machine with the three artifacts.
- Demo predictions executed on raw rows.
- No randomness at inference; artifacts/paths printed for traceability.

That's the full CRISP-DM mini demo — end-to-end, reproducible, and honest. If you want add-ons (e.g., ablate `f1twgt`, cost-sensitivity sweep, or simple calibration), say the word and I'll bolt them on cleanly.

🔗 🔍 🔍 🔍 🔍 🔍

+ Ask anything

0 · · ·

ChatGPT can make mistakes. OpenAI doesn't use San Jose State University workspace data to train its models.