A Comprehensive Field Guide for Data Scientists

February 2026

Contents

Preface: Why Evaluation Is the Most Important Skill

Building a machine learning model that produces numbers is easy. Building one you can *trust*, defend in a board meeting, and rely on to drive real decisions is an entirely different discipline. That discipline is **Model Evaluation & Validation**.

Many data science projects fail not because the algorithm was wrong, but because the practitioner optimised for the wrong metric, let data from the future leak into training, reported accuracy on a wildly imbalanced test set, or could not explain to a stakeholder why the model made a particular decision. This tutorial is designed to close those gaps.

The guide is structured around four pillars, each building on the previous one:

1. **Cross-Validation** — learning how to estimate generalisation performance honestly, and how to avoid the most common pitfalls like data leakage and overfitting to a validation fold.
2. **Hyperparameter Tuning** — systematically searching the space of model configurations rather than guessing, using methods that scale from exhaustive search to Bayesian optimisation.
3. **Metrics for Imbalanced Datasets** — understanding why accuracy is misleading when classes are unequal and which metrics actually reflect business cost.
4. **Model Interpretability** — opening the black box with SHAP, LIME, and partial dependence plots so that the model can be scrutinised, debugged, and explained.

Throughout, every concept is grounded in runnable Python examples using mainstream libraries (`scikit-learn`, `shap`, `lime`, `scikit-optimize`). Code is annotated line by line wherever a concept needs emphasis.

1 Cross-Validation: Honest Performance Estimation

1.1 The Core Problem: Why a Single Train–Test Split Is Not Enough

Suppose you train a classifier and measure its accuracy on a held-out test set. You get 87%. Should you celebrate? Not yet. That number reflects one particular random partition of your data. Had you used a different random seed, you might have gotten 82% or 91%. You have measured the performance of *one* model on *one* subset of your data, not the expected performance of your modelling *approach* on unseen data.

Cross-validation resolves this by evaluating the model on multiple, non-overlapping validation subsets and averaging the results. The variance of those scores tells you how stable your model is; the mean tells you the unbiased estimate of generalisation performance.

Just as a model can overfit the training data, an *evaluation procedure* can overfit the validation data. Every time you make a modelling decision based on a fixed validation set, you implicitly "train" on that set. Cross-validation delays this leakage by rotating which data is held out.

1.2 K-Fold Cross-Validation

1.2.1 How It Works

K-Fold partitions the dataset into k equally sized ***folds***. The model is trained k times; each time, one fold serves as the validation set and the remaining $k-1$ folds form the training set. The final performance estimate is the mean (and standard deviation) across all k scores.

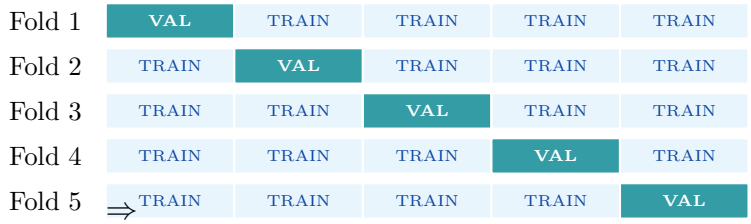

Figure 1: 5-Fold Cross-Validation: the teal fold rotates through each iteration.

1.2.2 Choosing k

- **k = 5 or 10** is the standard choice backed by empirical research (Kohavi 1995). It balances bias and variance in the estimator.
- **k = n** (Leave-One-Out CV) is nearly unbiased but has very high variance and is computationally expensive.
- **Stratified K-Fold** preserves the class distribution in each fold — always use this for classification.

1.2.3 Python Example: K-Fold and Stratified K-Fold

K-Fold Cross-Validation — Full Walkthrough

```python
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import (
    KFold, StratifiedKFold, cross_val_score, cross_validate
)
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# ------------------------------------------------------------------
# 1. Synthetic dataset: 1000 samples, 20 features, binary target
#    weights=(0.7, 0.3) means 70% class 0, 30% class 1  mild imbalance
# ------------------------------------------------------------------
X, y = make_classification(
    n_samples=1000,
    n_features=20,
    n_informative=10,
    n_redundant=4,
    weights=[0.7, 0.3],
    random_state=42
)

# ------------------------------------------------------------------
# 2. Build a Pipeline
#    Always wrap preprocessing inside the pipeline so that the scaler
#    is fit ONLY on training folds  never on the validation fold.
#    This is the single most common source of data leakage.
# ------------------------------------------------------------------
pipeline = Pipeline([
    ('scaler', StandardScaler()),          # zero-mean, unit-variance
    ('clf', RandomForestClassifier(
        n_estimators=100, random_state=42
    ))
])

# ------------------------------------------------------------------
# 3. Plain K-Fold (ignores class distribution in each fold)
# ------------------------------------------------------------------
kf = KFold(n_splits=5, shuffle=True, random_state=42)

scores_kfold = cross_val_score(
    pipeline, X, y,
    cv=kf,
    scoring='roc_auc',     # area under ROC curve: threshold-independent
    n_jobs=-1              # use all CPU cores
)

print("=== Standard K-Fold ===")
print(f"Fold AUC scores : {np.round(scores_kfold, 4)}")
```

```python
print(f"Mean AUC        : {scores_kfold.mean():.4f}")
print(f"Std  AUC        : {scores_kfold.std():.4f}")

# ------------------------------------------------------------------------
# 4. Stratified K-Fold (preserves class ratio in every fold)
#    For classification, ALWAYS prefer StratifiedKFold.
# ------------------------------------------------------------------------
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

scores_stratified = cross_val_score(
    pipeline, X, y,
    cv=skf,
    scoring='roc_auc',
    n_jobs=-1
)

print("\n=== Stratified K-Fold ===")
print(f"Fold AUC scores : {np.round(scores_stratified, 4)}")
print(f"Mean AUC        : {scores_stratified.mean():.4f}")
print(f"Std  AUC        : {scores_stratified.std():.4f}")

# ------------------------------------------------------------------------
# 5. cross_validate returns richer information:
#    train scores, test scores, and fit/score times.
# ------------------------------------------------------------------------
cv_results = cross_validate(
    pipeline, X, y,
    cv=skf,
    scoring=['roc_auc', 'f1', 'precision', 'recall'],
    return_train_score=True,   # detect overfitting by comparing
    n_jobs=-1
)

results_df = pd.DataFrame({
    'train_roc_auc': cv_results['train_roc_auc'],
    'test_roc_auc' : cv_results['test_roc_auc'],
    'test_f1'      : cv_results['test_f1'],
    'fit_time_s'   : cv_results['fit_time']
})
print("\n=== Detailed CV Results ===")
print(results_df.round(4).to_string(index=False))

# If train AUC >> test AUC, the model is overfitting.
gap = cv_results['train_roc_auc'].mean() - cv_results['test_roc_auc'
    ].mean()
print(f"\nOverfit gap (train - test AUC): {gap:.4f}")
if gap > 0.05:
    print("WARNING: Possible overfitting detected.")
```

What this code does, step by step:

Step 1 creates a synthetic binary classification dataset with mild class imbalance, which mirrors real business scenarios where one outcome (e.g., fraud, churn) is rarer than the other.

Step 2 is the most important architectural decision: the `Pipeline` ensures that the `StandardScaler` is fitted *only* on the training portion of each fold. If you scaled all data

first and then cross-validated, the validation fold's statistics would have influenced the scaler — a subtle form of data leakage that inflates your score.

Steps 3 and 4 compare plain `KFold` with `StratifiedKFold`. In practice, stratification reduces the variance of the CV estimator when class imbalance is present.

Step 5 uses

1.3 Time-Series Cross-Validation

Standard K-Fold **must not be used with time-series data**. The reason is fundamental: the future cannot cause the past. If your validation fold contains data from January and your training fold contains data from March, your model learns from the future to predict the past. This produces wildly optimistic estimates that will not hold in live deployment.

1.3.1 Walk-Forward Validation (Expanding Window)

`TimeSeriesSplit` implements ***walk-forward validation***, also called *expanding window* or *rolling origin* cross-validation. The training window always ends before the validation window begins, and the training set grows with each split.

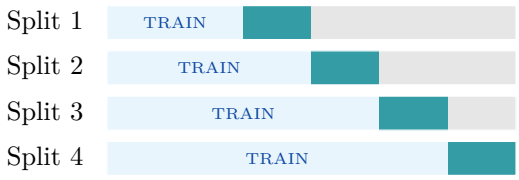

Figure 2: Time-Series Walk-Forward Validation: training always precedes validation temporally.

Time-Series Cross-Validation

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import TimeSeriesSplit, cross_validate
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# ---------------------------------------------------------------------
# 1. Simulate a time-ordered dataset (e.g., daily transaction data)
#    n_samples = 2000 days; features represent behavioural signals.
# ---------------------------------------------------------------------
np.random.seed(42)
n = 2000

# Create a temporal trend in the signal to mimic real time-series
time_idx = np.arange(n)
signal   = 0.3 * np.sin(2 * np.pi * time_idx / 365)  # seasonal
    component

X = np.column_stack([
    signal + np.random.randn(n) * 0.5,   # feature 1: noisy signal
    np.random.randn(n),                   # feature 2: noise
```

```
    time_idx / n + np.random.randn(n)*0.1 # feature 3: trend
])
y = (signal + np.random.randn(n) * 0.4 > 0).astype(int)

# ------------------------------------------------------------------
# 2. Configure TimeSeriesSplit
#    n_splits=5 creates 5 folds, each with progressively more
    training
#    data.  gap=30 inserts a 30-day gap between training and
    validation
#    to simulate realistic deployment delay.
# ------------------------------------------------------------------
tscv = TimeSeriesSplit(n_splits=5, gap=30)

# ------------------------------------------------------------------
# 3. Inspect what each split looks like
# ------------------------------------------------------------------
print("Split summary (train size | val size):")
for fold_num, (train_idx, val_idx) in enumerate(tscv.split(X), start
    =1):
    print(f"  Fold {fold_num}: train [{train_idx[0]}..{train_idx[-1]
        }] "
          f"({len(train_idx)} samples) | "
          f"val [{val_idx[0]}..{val_idx[-1]}] ({len(val_idx)}
             samples)")

# ------------------------------------------------------------------
# 4. Cross-validate a gradient boosting classifier
# ------------------------------------------------------------------
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', GradientBoostingClassifier(
        n_estimators=100, learning_rate=0.05,
        max_depth=4, random_state=42
    ))
])

cv_results = cross_validate(
    pipeline, X, y,
    cv=tscv,
    scoring=['roc_auc', 'f1'],
    return_train_score=True
)

print("\n=== Time-Series CV Results ===")
for fold_i in range(5):
    print(f"  Fold {fold_i+1}: "
          f"train AUC={cv_results['train_roc_auc'][fold_i]:.4f}  "
          f"val AUC={cv_results['test_roc_auc'][fold_i]:.4f}  "
          f"val F1={cv_results['test_f1'][fold_i]:.4f}")

print(f"\nMean val AUC : {cv_results['test_roc_auc'].mean():.4f} "
      f"+/- {cv_results['test_roc_auc'].std():.4f}")

# ------------------------------------------------------------------
# 5. Trend analysis: does performance degrade over time?
#    In live systems, concept drift causes later folds to score
    lower.
```

```
# ------------------------------------------------------------------------
fold_aucs = cv_results['test_roc_auc']
trend = np.polyfit(range(len(fold_aucs)), fold_aucs, deg=1)[0]
print(f"\nPerformance trend across folds: {trend:+.4f} per fold")
if trend < -0.01:
    print("ALERT: Declining performance  potential concept drift.")
else:
    print("Performance stable across time folds.")
```

The gap parameter is critical for production systems. If your model is retrained monthly and predictions are served for the following 30 days, inserting a 30-day gap into your CV scheme ensures that your offline evaluation mimics live conditions. Without it, validation samples immediately adjacent to the training cutoff may share patterns (such as the same weekly seasonality cycle) that won't exist at actual deployment time.

Trend analysis across folds is a uniquely time-series concern. If AUC falls monotonically from fold 1 to fold 5, your model is experiencing ***concept drift*** — the statistical relationship between features and target is changing over time. This is a signal to explore re-training schedules or online learning approaches.

- **Shuffling data** before splitting — destroys temporal ordering and guarantees leakage.
- **Using aggregated features** (e.g., 30-day rolling mean computed on the full dataset) before splitting. Always compute such features inside the pipeline or after splitting.
- **Ignoring the deployment gap** between training cutoff and live prediction.

2 Hyperparameter Tuning: Systematic Model Configuration

2.1 What Are Hyperparameters?

Machine learning models have two kinds of parameters. ***Model parameters*** are learned during training (e.g., the weights in a neural network, the split thresholds in a decision tree). ***Hyperparameters*** are set *before* training begins and control the learning process itself: how many trees to grow, how deep each tree can be, the learning rate, regularisation strength, and so on.

The goal of hyperparameter tuning is to find the configuration that maximises the model's generalisation performance, as measured by cross-validation, not by training error.

Every tuning method follows the same loop: **propose** a configuration → **train** the model with that configuration → **evaluate** using cross-validation → **record** the score → repeat until budget exhausted → **select** the configuration with the best CV score. The methods differ in how they *propose* the next candidate.

2.2 Grid Search

Grid search exhaustively evaluates every combination of a pre-specified set of hyperparameter values. If you specify 3 values for `max_depth` and 4 values for `n_estimators`, grid search trains $3 \times 4 = 12$ models (each cross-validated k times), giving $12k$ total fits.

This is guaranteed to find the best configuration *within the grid*, but it scales exponentially with the number of hyperparameters – a phenomenon called the ***curse of dimensionality in search***.

Grid Search with Cross-Validation

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

X, y = make_classification(
    n_samples=1500, n_features=20, n_informative=12,
    weights=[0.65, 0.35], random_state=42
)

# ------------------------------------------------------------------
# Pipeline: scaler + classifier
# Hyperparameters for pipeline steps are referenced as
# "stepname__parametername"
# ------------------------------------------------------------------
pipeline = Pipeline([
```

```
    ('scaler', StandardScaler()),
    ('clf', GradientBoostingClassifier(random_state=42))
])

# ----------------------------------------------------------------------
# Define the grid
# 3 x 4 x 3 = 36 configurations, each with 5-fold CV = 180 fits.
# For production models, this grid is modest; real searches often
# involve hundreds of configurations.
# ----------------------------------------------------------------------
param_grid = {
    'clf__n_estimators':  [50, 100, 200],
    'clf__max_depth':     [2, 3, 4, 5],
    'clf__learning_rate': [0.01, 0.05, 0.1],
}

cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state
    =42)

grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    scoring='roc_auc',          # optimise for AUC
    cv=cv_strategy,
    n_jobs=-1,                  # parallelise across all cores
    verbose=1,                  # print progress
    refit=True                  # refit on full data with best params
)

grid_search.fit(X, y)

# ----------------------------------------------------------------------
# Results exploration
# ----------------------------------------------------------------------
print(f"Best CV AUC   : {grid_search.best_score_:.4f}")
print(f"Best params   : {grid_search.best_params_}")

# Convert the full results to a DataFrame for analysis
results_df = pd.DataFrame(grid_search.cv_results_)

# Show top-5 configurations
top5 = results_df[['params', 'mean_test_score', 'std_test_score',
                   'rank_test_score']]\
         .sort_values('rank_test_score').head(5)

print("\nTop 5 configurations:")
for _, row in top5.iterrows():
    print(f"  Rank {int(row['rank_test_score'])}: "
          f"AUC={row['mean_test_score']:.4f} "
          f"+/-{row['std_test_score']:.4f}  "
          f"{row['params']}")

# ----------------------------------------------------------------------
# The best estimator is already refitted on the full dataset.
# It can be used directly for prediction.
# ----------------------------------------------------------------------
best_model = grid_search.best_estimator_
```

2.3 Random Search

Random search samples hyperparameter combinations uniformly at random from a defined distribution. Bergstra and Bengio (2012) demonstrated mathematically that random search is more efficient than grid search when the number of hyperparameters is large and only a few are truly important, because random search explores more of the *marginal* space of each parameter with the same number of evaluations.

The intuition: if 2 out of 5 hyperparameters matter but you don't know which 2, a $5 \times 5 \times 5 \times 5 \times 5$ grid evaluates each important parameter at only 5 distinct values. Fifty random samples explore each important parameter at up to 50 distinct values, while still covering the full joint space.

Random Search with Continuous Distributions

```python
import numpy as np
from scipy.stats import uniform, randint, loguniform
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1500, n_features=20, n_informative=12,
    weights=[0.65, 0.35], random_state=42
)

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', GradientBoostingClassifier(random_state=42))
])

# ---------------------------------------------------------------------
# Distributions instead of grids
#
# loguniform(a, b): draws from a log-uniform distribution between a
#    and b, which is appropriate for learning rates and regularisation
#    strengths that span orders of magnitude.
#
# randint(low, high): draws integers from [low, high).
#    Best for counts like n_estimators, max_depth.
#
# uniform(loc, scale): draws from [loc, loc+scale].
#    Best for continuous values with no strong scale preference.
# ---------------------------------------------------------------------
param_distributions = {
    'clf__n_estimators':  randint(50, 500),          # integers in [50,500)
    'clf__max_depth':     randint(2, 8),              # integers in [2,8)
    'clf__learning_rate': loguniform(1e-3, 0.5),     # log-scale [0.001, 0.5]
    'clf__subsample':     uniform(0.5, 0.5),          # uniform in [0.5, 1.0]
```

```
    'clf__min_samples_leaf': randint(1, 30),
    'clf__max_features':  uniform(0.3, 0.7),          # fraction of
        features
}

cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state
    =42)

random_search = RandomizedSearchCV(
    estimator=pipeline,
    param_distributions=param_distributions,
    n_iter=60,           # evaluate 60 random configurations
    scoring='roc_auc',
    cv=cv_strategy,
    n_jobs=-1,
    random_state=42,
    verbose=1,
    refit=True
)

random_search.fit(X, y)

print(f"Best CV AUC   : {random_search.best_score_:.4f}")
print(f"Best params   :")
for k, v in random_search.best_params_.items():
    print(f"  {k:<35}: {v}")

# ---------------------------------------------------------------------
# Compare grid search vs random search coverage
# For 60 evaluations:
#   Grid: constrained to pre-specified values
#   Random: explores continuous ranges, likely finds better optima
# ---------------------------------------------------------------------
```

2.4 Bayesian Optimisation

Bayesian optimisation is the state of the art for expensive hyperparameter tuning. Unlike grid and random search, which treat each evaluation independently, Bayesian optimisation *learns* from previous evaluations and uses a probabilistic ***surrogate model*** (typically a Gaussian Process or Tree-structured Parzen Estimator) to predict which regions of the hyperparameter space are most promising.

The surrogate model balances ***exploitation*** (evaluating where the model predicts the best performance) with ***exploration*** (evaluating uncertain regions that might harbour better optima). This trade-off is managed by an ***acquisition function***, most commonly Expected Improvement (EI):

$$\mathrm{EI}(\mathbf{x}) = \mathbb{E}\big[\max\big(f(\mathbf{x}) - f(\mathbf{x}^+), 0\big)\big]$$

where $f(\mathbf{x}^+)$ is the best observed score so far. The next evaluation point is $\mathbf{x}_{\text{next}} = \arg\max_{\mathbf{x}} \mathrm{EI}(\mathbf{x})$.

Bayesian Optimisation with scikit-optimize

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import StratifiedKFold, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# scikit-optimize provides BayesSearchCV with a scikit-learn API
# Install: pip install scikit-optimize
from skopt import BayesSearchCV
from skopt.space import Real, Integer, Categorical

X, y = make_classification(
    n_samples=1500, n_features=20, n_informative=12,
    weights=[0.65, 0.35], random_state=42
)

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', GradientBoostingClassifier(random_state=42))
])

# ------------------------------------------------------------------
# Search space using skopt types
#   Real(low, high, prior='log-uniform'): continuous log-scale range
#   Integer(low, high): integer range
#   Categorical([...]): discrete set of choices
# ------------------------------------------------------------------
search_space = {
    'clf__n_estimators' : Integer(50, 400),
    'clf__max_depth'    : Integer(2, 7),
    'clf__learning_rate': Real(1e-3, 0.5, prior='log-uniform'),
    'clf__subsample'    : Real(0.4, 1.0),
    'clf__min_samples_leaf': Integer(1, 25),
}

cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state
    =42)

# ------------------------------------------------------------------
# BayesSearchCV: n_iter controls total evaluations.
# Fewer evaluations than grid/random search can achieve better
    results
# because each evaluation is informed by prior results.
# ------------------------------------------------------------------
bayes_search = BayesSearchCV(
    estimator=pipeline,
    search_spaces=search_space,
    n_iter=40,                # 40 informed evaluations
    scoring='roc_auc',
    cv=cv_strategy,
    n_jobs=1,                 # skopt internals require n_jobs=1
    random_state=42,
    verbose=0,
    refit=True
)
```

```python

bayes_search.fit(X, y)

print(f"Best Bayesian CV AUC : {bayes_search.best_score_:.4f}")
print("Best parameters:")
for k, v in bayes_search.best_params_.items():
    print(f"  {k:<35}: {v}")

# ---------------------------------------------------------------------
# Convergence plot: track how AUC improves iteration by iteration.
# A flat curve means the search has converged; more budget won't help.
# ---------------------------------------------------------------------
import matplotlib.pyplot as plt

scores_over_time = []
best_so_far = -np.inf
for score in bayes_search.cv_results_['mean_test_score']:
    best_so_far = max(best_so_far, score)
    scores_over_time.append(best_so_far)

plt.figure(figsize=(8, 4))
plt.plot(range(1, len(scores_over_time)+1), scores_over_time,
         marker='o', color='#0d47a1', linewidth=2, markersize=4)
plt.xlabel("Evaluation number")
plt.ylabel("Best CV AUC so far")
plt.title("Bayesian Optimisation Convergence")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.savefig("bayesian_convergence.png", dpi=150)
plt.close()
print("Convergence plot saved.")
```

2.4.1 Choosing a Tuning Strategy

The following table summarises when to use each approach:

Method	Best for	Weakness	Budget
Grid Search	Small grids, interpretable search space	Exponential scaling	Low
Random Search	Many hyperparameters, few matter	May miss fine structure	Medium
Bayesian Opt.	Expensive models, large spaces	Sequential (harder to parallelise)	High

Table 1: Hyperparameter tuning method comparison.

When you tune hyperparameters with 5-fold stratified CV, you must also report your final performance estimate using a separate *outer* cross-validation loop (nested CV), or a hold-out test set that was never touched during tuning. Using the same data for both tuning and final reporting is a form of overfitting.

3 Metrics for Imbalanced Datasets

3.1 Why Accuracy Fails

Imagine a fraud detection model. Fraudulent transactions represent 0.5% of all transactions. A model that predicts "not fraud" for every single transaction achieves 99.5% accuracy. This is a useless model that catches zero fraud.

This is why ***class imbalance*** is one of the most practically important challenges in applied machine learning. The choice of evaluation metric must reflect the cost structure of the problem, not just raw prediction accuracy.

3.2 The Confusion Matrix: A Foundation

All classification metrics derive from four counts:

	Predicted Positive	**Predicted Negative**
Actual Positive	green!15 TP (True Positive)	red!15 FN (False Negative)
Actual Negative	red!15 FP (False Positive)	green!15 TN (True Negative)

Table 2: The 2Œ2 confusion matrix.

From these four numbers, we derive every metric in this section.

3.3 Core Metrics

Precision answers the question: "Of all the cases I predicted positive, what fraction actually were positive?" It penalises false alarms.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall (also called Sensitivity or True Positive Rate) answers: "Of all the actual positive cases, what fraction did I catch?" It penalises missed detections.

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1-Score is the harmonic mean of precision and recall. The harmonic mean is used (rather than arithmetic) because it punishes extreme imbalances between precision and recall.

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The $\mathbf{F}_\beta$ **score** generalises F1 by weighting recall β times more than precision. For fraud detection where missing a fraud (false negative) is far more costly than a false alarm, use $\beta > 1$:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}$$

Balanced Accuracy averages recall across all classes, correcting for imbalance without requiring threshold tuning:

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right)$$

Comprehensive Imbalanced Classification Evaluation

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold, cross_val_predict
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    f1_score,
    balanced_accuracy_score,
    precision_recall_curve,
    average_precision_score,
    roc_auc_score,
    ConfusionMatrixDisplay
)

# ------------------------------------------------------------------
# 1. Heavily imbalanced dataset: 95% class 0, 5% class 1
#    This simulates fraud, disease, or rare equipment failure.
# ------------------------------------------------------------------
X, y = make_classification(
    n_samples=5000,
    n_features=20,
    n_informative=10,
    weights=[0.95, 0.05],   # 4750 negatives, 250 positives
    random_state=42
)
print(f"Class distribution: {np.bincount(y)}  "
      f"({100*y.mean():.1f}% positive)")

# ------------------------------------------------------------------
# 2. Train three models and get out-of-fold predictions
#    cross_val_predict generates predictions for every sample using
#    only models trained without that sample  proper OOF evaluation.
# ------------------------------------------------------------------
```

```python
models = {
    'Logistic Regression': Pipeline([
        ('scaler', StandardScaler()),
        ('clf', LogisticRegression(
            class_weight='balanced',  # upweight minority
            C=0.1, max_iter=1000
        ))
    ]),
    'Random Forest': Pipeline([
        ('scaler', StandardScaler()),
        ('clf', RandomForestClassifier(
            n_estimators=200,
            class_weight='balanced_subsample',
            random_state=42
        ))
    ]),
    'Gradient Boosting': Pipeline([
        ('scaler', StandardScaler()),
        ('clf', GradientBoostingClassifier(
            n_estimators=200, learning_rate=0.05,
            max_depth=4, random_state=42
        ))
    ]),
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

results = {}
for name, pipe in models.items():
    # predict_proba: probability scores for each class
    proba  = cross_val_predict(pipe, X, y, cv=cv, method='predict_
       proba')[:,1]
    # predict: hard class labels at 0.5 threshold
    labels = cross_val_predict(pipe, X, y, cv=cv, method='predict')

    results[name] = {
        'proba'  : proba,
        'labels' : labels,
        'ap'     : average_precision_score(y, proba),    # area under
            P-R curve
        'auc'    : roc_auc_score(y, proba),
        'f1'     : f1_score(y, labels),
        'f2'     : f1_score(y, labels, beta=2.0),        # recall-
           weighted
        'bal_acc': balanced_accuracy_score(y, labels),
    }

# ------------------------------------------------------------------
# 3. Summary table
# ------------------------------------------------------------------
print("\n{:<22} {:>8} {:>8} {:>8} {:>8} {:>8}".format(
    "Model", "AUC", "Avg Prec", "F1", "F2", "BalAcc"))
print("-" * 70)
for name, r in results.items():
    print(f"{name:<22} {r['auc']:>8.4f} {r['ap']:>8.4f} "
          f"{r['f1']:>8.4f} {r['f2']:>8.4f} {r['bal_acc']:>8.4f}")

# ------------------------------------------------------------------
```

```
# 4. Precision-Recall curves  the right plot for imbalanced data
#    ROC curves are optimistic when the negative class dominates;
#    P-R curves expose the real difficulty of finding true positives.
# ------------------------------------------------------------------
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

colors = ['#0d47a1', '#00838f', '#e65100']
for (name, r), color in zip(results.items(), colors):
    prec, rec, thresh = precision_recall_curve(y, r['proba'])
    axes[0].plot(rec, prec, color=color, linewidth=2,
                 label=f"{name} (AP={r['ap']:.3f})")

axes[0].axhline(y.mean(), color='gray', linestyle='--', linewidth=1.2,
                label=f"Random (AP={y.mean():.3f})")
axes[0].set_xlabel("Recall", fontsize=12)
axes[0].set_ylabel("Precision", fontsize=12)
axes[0].set_title("Precision-Recall Curves", fontsize=13, fontweight='bold')
axes[0].legend(fontsize=9)
axes[0].grid(alpha=0.3)

# ------------------------------------------------------------------
# 5. Threshold analysis for the best model
#    The default 0.5 threshold is rarely optimal for imbalanced data.
#    Plot F1 and F2 as a function of threshold to find the sweet spot.
# ------------------------------------------------------------------
best_name = max(results, key=lambda n: results[n]['ap'])
proba_best = results[best_name]['proba']
prec, rec, thresholds = precision_recall_curve(y, proba_best)

f1_scores = 2 * prec[:-1] * rec[:-1] / (prec[:-1] + rec[:-1] + 1e-9)
f2_scores = 5 * prec[:-1] * rec[:-1] / (4*prec[:-1] + rec[:-1] + 1e-9)

axes[1].plot(thresholds, f1_scores, color='#0d47a1', label='F1 score', lw=2)
axes[1].plot(thresholds, f2_scores, color='#e65100', label='F2 score', lw=2)
axes[1].axvline(0.5, color='gray', linestyle='--', label='Default threshold')

best_f1_thresh = thresholds[np.argmax(f1_scores)]
axes[1].axvline(best_f1_thresh, color='#0d47a1', linestyle=':', lw=1.5,
                label=f'Best F1 threshold={best_f1_thresh:.2f}')

axes[1].set_xlabel("Decision Threshold", fontsize=12)
axes[1].set_ylabel("Score", fontsize=12)
axes[1].set_title(f"F1/F2 vs Threshold  {best_name}", fontsize=13,
                  fontweight='bold')
axes[1].legend(fontsize=9)
axes[1].grid(alpha=0.3)
axes[1].set_xlim(0, 1)

```

```
plt.tight_layout()
plt.savefig("imbalanced_metrics.png", dpi=150)
plt.close()

# ------------------------------------------------------------------
# 6. Optimal threshold classification report
# ------------------------------------------------------------------
optimal_labels = (proba_best >= best_f1_thresh).astype(int)
print(f"\nClassification report at optimal threshold ({best_f1_
    thresh:.2f}):")
print(classification_report(y, optimal_labels,
                            target_names=['No Fraud', 'Fraud']))
```

3.3.1 Reading Precision-Recall Curves

The ***Precision-Recall (PR) curve*** plots precision on the y-axis against recall on the x-axis as the decision threshold sweeps from 0 to 1. The ***Average Precision (AP)*** score is the area under this curve.

For imbalanced datasets, the PR curve is more informative than the ROC curve because it directly measures performance on the minority class. The ROC curve can appear excellent (AUC > 0.9) even when the model performs poorly at actually finding positive cases, because the large number of true negatives inflates the TN count.

The dashed horizontal line at the baseline class frequency is the AP score of a random classifier. Any model must substantially exceed this baseline to be useful.

3.3.2 Threshold Optimisation

The decision threshold of 0.5 is an arbitrary convention. For imbalanced problems, the optimal threshold is almost never 0.5. The threshold should be chosen based on the cost asymmetry of the problem: if false negatives are 5 times more costly than false positives, use $\beta = \sqrt{5} \approx 2.2$ in the F_β score to find the optimal threshold.

Metric	**Use when**	**Range**
Accuracy	Classes are balanced AND errors are equally costly	[0, 1]
Balanced Accuracy	Imbalanced, quick threshold-free comparison	[0, 1]
F1 Score	False positives and negatives equally costly	[0, 1]
F2 Score	Missing positives more costly than false alarms	[0, 1]
Average Precision	Need full threshold-independent evaluation	[0, 1]
ROC-AUC	Comparing discriminative ability across thresholds	[0.5, 1]

Table 3: Metric selection guide for imbalanced classification problems.

4 Model Interpretability

4.1 Why Interpretability Is Not Optional

A model that cannot be explained is a model that cannot be trusted, audited, debugged, or improved. In a business context, interpretability matters for three distinct reasons:

Regulatory compliance: Legislation such as GDPR (right to explanation), the EU AI Act, and US financial regulations often require that models provide explanations for decisions that affect individuals.

Model debugging: If your model performs worse on a demographic subgroup or in a specific time period, interpretability methods reveal which features are driving that behaviour.

Stakeholder trust: A data scientist who can say "the model flags this customer as high-risk primarily because their last payment was 60 days late and their credit utilisation is 94%" earns far more trust than one who shows a confusion matrix.

4.2 SHAP: SHapley Additive exPlanations

4.2.1 The Theory

SHAP is grounded in cooperative game theory. A ***Shapley value*** is the average marginal contribution of a feature across all possible orderings (coalitions) of features. For feature j in a prediction $f(\mathbf{x})$:

$$\phi_j = \sum_{S \subseteq F \setminus \{j\}} \frac{|S|!\,(|F| - |S| - 1)!}{|F|!} \left[f_{S \cup \{j\}}(\mathbf{x}_{S \cup \{j\}}) - f_S(\mathbf{x}_S) \right]$$

where F is the set of all features, S is any subset not containing j, and f_S is the model's prediction using only features in S. The SHAP value ϕ_j represents the feature's fair share of the difference between the model's prediction and the global base rate.

The key property that makes SHAP uniquely appealing is that the SHAP values sum exactly to the model's output relative to the expected output:

$$f(\mathbf{x}) = \mathbb{E}[f(X)] + \sum_{j=1}^{|F|} \phi_j$$

This additive decomposition means every prediction is fully accounted for, with no residual "unexplained" component.

SHAP: Global and Local Explanations

```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import shap
```

```
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# ---------------------------------------------------------------------
# 1. Dataset with interpretable feature names
#    (simulating a credit scoring context)
# ---------------------------------------------------------------------
np.random.seed(42)
n = 2000
X_raw = pd.DataFrame({
    'days_late_last_payment' : np.random.exponential(10, n).clip(0,
       120),
    'credit_utilisation_pct' : np.random.beta(2, 5, n) * 100,
    'num_open_accounts'      : np.random.poisson(4, n),
    'income_annual_k'        : np.random.lognormal(4, 0.5, n),
    'years_credit_history'   : np.random.gamma(3, 3, n).clip(0.5,
       30),
    'num_hard_inquiries'     : np.random.poisson(1.5, n),
    'debt_to_income_ratio'   : np.random.beta(2, 4, n),
    'months_since_derog'     : np.random.exponential(20, n).clip(0,
       120),
})

# Logistic rule with noise  creates realistic credit default target
logit = (
    0.05 * X_raw['days_late_last_payment']
    + 0.03 * X_raw['credit_utilisation_pct']
    - 0.02 * X_raw['income_annual_k'] / 10
    + 0.5  * X_raw['debt_to_income_ratio']
    + 0.4  * np.random.randn(n)
    - 3.0
)
y = (1 / (1 + np.exp(-logit)) > 0.5).astype(int)

X_train, X_test, y_train, y_test = train_test_split(
    X_raw, y, test_size=0.25, random_state=42, stratify=y
)

# ---------------------------------------------------------------------
# 2. Train a gradient boosting model
#    We do NOT use a Pipeline here because SHAP needs the raw
#    model object (not wrapped), though TreeExplainer handles this.
# ---------------------------------------------------------------------
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s  = scaler.transform(X_test)

model = GradientBoostingClassifier(
    n_estimators=200, learning_rate=0.05,
    max_depth=4, random_state=42
)
model.fit(X_train_s, y_train)

# ---------------------------------------------------------------------
# 3. Create SHAP explainer
```

```
#     TreeExplainer is optimised for tree-based models (O(TLD^2) where
#     T=trees, L=leaves, D=depth)  much faster than brute-force.
# ------------------------------------------------------------------------
explainer = shap.TreeExplainer(model)

# Compute SHAP values for the test set
# shap_values shape: (n_test_samples, n_features) for binary
# For binary, index [1] gives SHAP for the positive class.
shap_values = explainer.shap_values(X_test_s)

# ------------------------------------------------------------------------
# 4. Global explanation: Beeswarm summary plot
#     Each dot = one test sample. X-axis = SHAP value (impact on
#     model output). Colour = feature value (red=high, blue=low).
#     Features are sorted by mean absolute SHAP value.
# ------------------------------------------------------------------------
shap.summary_plot(
    shap_values, X_test,    # use un-scaled X for readable values
    plot_type='dot',
    show=False,
    max_display=8
)
plt.tight_layout()
plt.savefig("shap_beeswarm.png", dpi=150, bbox_inches='tight')
plt.close()

# ------------------------------------------------------------------------
# 5. Global feature importance (bar chart version)
# ------------------------------------------------------------------------
shap.summary_plot(
    shap_values, X_test,
    plot_type='bar',
    show=False,
    max_display=8
)
plt.tight_layout()
plt.savefig("shap_importance.png", dpi=150, bbox_inches='tight')
plt.close()

# ------------------------------------------------------------------------
# 6. Local explanation: a single prediction
#     Interpret the decision for one high-risk customer.
# ------------------------------------------------------------------------
sample_idx = np.where(y_test == 1)[0][0]   # first true positive
sample     = X_test_s[sample_idx:sample_idx+1]

predicted_prob = model.predict_proba(sample)[0, 1]
print(f"\n--- Local Explanation for Sample {sample_idx} ---")
print(f"Predicted probability of default: {predicted_prob:.3f}")
print(f"Base value (mean prediction)    : {explainer.expected_value:.3f}")
print(f"\nSHAP contribution of each feature:")

feature_names = X_raw.columns.tolist()
sample_shap   = shap_values[sample_idx]
sample_orig   = X_test.iloc[sample_idx]

```

```
contributions = pd.DataFrame({
    'feature'     : feature_names,
    'value'       : sample_orig.values,
    'shap_value'  : sample_shap
}).sort_values('shap_value', key=abs, ascending=False)

for _, row in contributions.iterrows():
    direction = "increases" if row['shap_value'] > 0 else "decreases"
    print(f"  {row['feature']:<30}: {direction} risk by "
          f"{abs(row['shap_value']):.4f} "
          f"(feature value = {row['value']:.2f})")

total = explainer.expected_value + sample_shap.sum()
print(f"\nBase rate ({explainer.expected_value:.3f}) + "
      f"sum of contributions ({sample_shap.sum():.3f}) "
      f"= {total:.3f} (model output before sigmoid)")

# ------------------------------------------------------------------
# 7. SHAP Dependence Plot: non-linear relationships
#    Shows SHAP value of one feature as a function of its raw value,
#    coloured by an interacting feature chosen automatically by SHAP.
# ------------------------------------------------------------------
shap.dependence_plot(
    'days_late_last_payment',
    shap_values, X_test,
    interaction_index='auto',
    show=False
)
plt.tight_layout()
plt.savefig("shap_dependence.png", dpi=150, bbox_inches='tight')
plt.close()
print("\nSHAP plots saved.")
```

4.2.2 Reading SHAP Output

The **beeswarm plot** is the primary diagnostic for global interpretability. Each row is a feature. Each dot is a sample. The position on the x-axis is the SHAP value: positive values push the prediction toward the positive class; negative values push it away. The colour represents the feature's raw value. For example: if points with high `credit_utilisation_pct` (red) have large positive SHAP values, the model has learned that high utilisation predicts default — a result that is consistent with domain knowledge, which builds confidence.

The **local waterfall** for a single prediction shows how each feature's SHAP value adds to or subtracts from the base rate to produce the final score. This is the explanation you would present to a credit officer reviewing a declined application.

4.3 LIME: Local Interpretable Model-agnostic Explanations

LIME takes a philosophically different approach. Rather than computing exact Shapley values, LIME constructs a *local linear approximation* of the model's behaviour in the neighbourhood of a specific prediction.

The algorithm works in three steps:

Step 1. Perturb the input. Generate N synthetic samples by randomly turning features on and off (for tabular data, this means replacing feature values with values sampled from the training distribution).

Step 2. Weight by proximity. Weight each synthetic sample by its distance from the original input using an exponential kernel.

Step 3. Fit a sparse linear model. Train a regularised linear model (typically LASSO) on the weighted synthetic samples. The coefficients of this linear model are the local explanation.

LIME Tabular Explainer

```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import lime
import lime.lime_tabular
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Reuse the credit scoring dataset from the SHAP section
np.random.seed(42)
n = 2000
X_raw = pd.DataFrame({
    'days_late_last_payment' : np.random.exponential(10, n).clip(0,
       120),
    'credit_utilisation_pct' : np.random.beta(2, 5, n) * 100,
    'num_open_accounts'      : np.random.poisson(4, n),
    'income_annual_k'        : np.random.lognormal(4, 0.5, n),
    'years_credit_history'   : np.random.gamma(3, 3, n).clip(0.5,
       30),
    'num_hard_inquiries'     : np.random.poisson(1.5, n),
    'debt_to_income_ratio'   : np.random.beta(2, 4, n),
    'months_since_derog'     : np.random.exponential(20, n).clip(0,
       120),
})
logit = (0.05*X_raw['days_late_last_payment']
         + 0.03*X_raw['credit_utilisation_pct']
         - 0.02*X_raw['income_annual_k']/10
         + 0.5*X_raw['debt_to_income_ratio']
         + 0.4*np.random.randn(n) - 3.0)
y = (1/(1+np.exp(-logit)) > 0.5).astype(int)

X_train, X_test, y_train, y_test = train_test_split(
    X_raw.values, y, test_size=0.25, random_state=42, stratify=y
)

scaler   = StandardScaler()
X_tr_s   = scaler.fit_transform(X_train)
X_te_s   = scaler.transform(X_test)

```

```
model = GradientBoostingClassifier(
    n_estimators=200, learning_rate=0.05, max_depth=4, random_state=
        42
)
model.fit(X_tr_s, y_train)

# ----------------------------------------------------------------------
# 1. Build the LIME explainer
#    training_data: the SCALED training set, so LIME samples from
    the
#    correct distribution.
#    feature_names: column names for readable output.
#    class_names: label strings.
#    discretize_continuous: if True, LIME bins continuous features
#    this makes explanations more readable but less precise.
# ----------------------------------------------------------------------
explainer = lime.lime_tabular.LimeTabularExplainer(
    training_data=X_tr_s,
    feature_names=X_raw.columns.tolist(),
    class_names=['No Default', 'Default'],
    mode='classification',
    discretize_continuous=True,
    random_state=42
)

# ----------------------------------------------------------------------
# 2. Explain a single prediction
#    num_features: how many features to include in the local model.
#    num_samples: how many perturbed samples to generate (more =
    stable).
# ----------------------------------------------------------------------
idx = np.where(y_test == 1)[0][3]   # a true default case
sample = X_te_s[idx]

explanation = explainer.explain_instance(
    data_row=sample,
    predict_fn=model.predict_proba,
    num_features=8,
    num_samples=3000,
    top_labels=1
)

pred_prob = model.predict_proba(sample.reshape(1,-1))[0,1]
print(f"Predicted probability of default: {pred_prob:.3f}")
print("\nLIME local explanation (feature: weight):")
for feat, weight in explanation.as_list(label=1):
    direction = "increases risk" if weight > 0 else "decreases risk"
    print(f"  {feat:<50}: {weight:+.4f}  ({direction})")

# ----------------------------------------------------------------------
# 3. Save the LIME explanation as an HTML file
#    This is LIME's native format and renders feature weights as
#    a horizontal bar chart.
# ----------------------------------------------------------------------
html_str = explanation.as_html()
with open("lime_explanation.html", "w") as f:
    f.write(html_str)
print("\nLIME HTML explanation saved.")
```

```
95
96  # ------------------------------------------------------------------
97  # 4. Manual bar chart (for embedding in reports)
98  # ------------------------------------------------------------------
99  lime_list = sorted(explanation.as_list(label=1), key=lambda x: x[1])
100 features_l = [item[0] for item in lime_list]
101 weights_l  = [item[1] for item in lime_list]
102 colors_l   = ['#e65100' if w > 0 else '#0d47a1' for w in weights_l]
103
104 fig, ax = plt.subplots(figsize=(9, 5))
105 bars = ax.barh(features_l, weights_l, color=colors_l, edgecolor='
       white')
106 ax.axvline(0, color='black', linewidth=0.8)
107 ax.set_xlabel("LIME weight (positive = increases P(Default))",
       fontsize=11)
108 ax.set_title(f"LIME Local Explanation  Sample {idx}\n"
109              f"P(Default)={pred_prob:.3f}", fontsize=12, fontweight=
                    'bold')
110 ax.grid(axis='x', alpha=0.3)
111 plt.tight_layout()
112 plt.savefig("lime_explanation.png", dpi=150, bbox_inches='tight')
113 plt.close()
114 print("LIME bar chart saved.")
```

4.3.1 SHAP vs LIME: When to Use Each

Property	SHAP	LIME
Mathematical foundation	Shapley values (game theory)	Local linear surrogate
Consistency	Globally consistent; local explanations sum to prediction	Locally consistent only; can vary between runs
Model agnosticism	TreeExplainer requires tree models; KernelExplainer is model-agnostic but slow	Fully model-agnostic
Computational cost	Fast for trees, slow for deep models	Moderate; depends on num_samples
Stability	Deterministic (for TreeExplainer)	Stochastic; run multiple times to check variance
Best for	Audits, global analysis, regulatory documentation	Quick local explanations, non-tree models

Table 4: SHAP versus LIME comparison.

4.4 Partial Dependence Plots

Partial Dependence Plots (PDP) show the *marginal* effect of one or two features on the model's predicted outcome, averaging over all other features. Formally, for feature j:

$$\hat{f}_j(x_j) = \mathbb{E}_{X_{-j}}\left[\hat{f}(x_j, X_{-j})\right] \approx \frac{1}{n}\sum_{i=1}^{n} \hat{f}(x_j, \mathbf{x}_{-j}^{(i)})$$

For each value of x_j, the model prediction is computed for every training sample (fixing x_j while using each sample's actual values for all other features), and the results are averaged. This removes the dependence on all other features, showing the isolated effect of x_j.

Partial Dependence Plots and ICE Curves

```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sklearn.inspection import PartialDependenceDisplay
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

np.random.seed(42)
n = 2000
X_raw = pd.DataFrame({
    'days_late_last_payment' : np.random.exponential(10, n).clip(0,
       120),
    'credit_utilisation_pct' : np.random.beta(2, 5, n) * 100,
    'num_open_accounts'      : np.random.poisson(4, n),
    'income_annual_k'        : np.random.lognormal(4, 0.5, n),
    'years_credit_history'   : np.random.gamma(3, 3, n).clip(0.5,
       30),
    'num_hard_inquiries'     : np.random.poisson(1.5, n),
    'debt_to_income_ratio'   : np.random.beta(2, 4, n),
    'months_since_derog'     : np.random.exponential(20, n).clip(0,
       120),
})
logit = (0.05*X_raw['days_late_last_payment']
         + 0.03*X_raw['credit_utilisation_pct']
         - 0.02*X_raw['income_annual_k']/10
         + 0.5*X_raw['debt_to_income_ratio']
         + 0.4*np.random.randn(n) - 3.0)
y = (1/(1+np.exp(-logit)) > 0.5).astype(int)

X_train, X_test, y_train, y_test = train_test_split(
    X_raw, y, test_size=0.25, random_state=42, stratify=y
)

scaler   = StandardScaler()
X_tr_s   = pd.DataFrame(scaler.fit_transform(X_train), columns=X_raw
   .columns)
X_te_s   = pd.DataFrame(scaler.transform(X_test),  columns=X_raw.
   columns)

model = GradientBoostingClassifier(
    n_estimators=200, learning_rate=0.05, max_depth=4, random_state=
       42
)
model.fit(X_tr_s, y_train)
```

```
42
43 # ----------------------------------------------------------------------
44 # 1. PDP for the top 4 features
45 #    kind='both': overlay ICE (individual conditional expectation)
46 #    curves on top of the mean PDP line.
47 #    ICE curves reveal heterogeneity: if all ICE curves are parallel,
48 #    the feature has the same effect on everyone.  If they fan out,
49 #    there are interaction effects with other features.
50 # ----------------------------------------------------------------------
51 features_to_plot = [
52     'days_late_last_payment',
53     'credit_utilisation_pct',
54     'debt_to_income_ratio',
55     'income_annual_k'
56 ]
57
58 fig, axes = plt.subplots(2, 2, figsize=(13, 9))
59 axes_flat = axes.flatten()
60
61 for ax, feat in zip(axes_flat, features_to_plot):
62     disp = PartialDependenceDisplay.from_estimator(
63         model,
64         X_tr_s,
65         features=[feat],
66         kind='both',              # PDP + ICE
67         subsample=200,            # use 200 samples for ICE (speed)
68         n_jobs=-1,
69         grid_resolution=60,
70         ax=ax,
71         ice_lines_kw={'color': '#90caf9', 'alpha': 0.3, 'linewidth':
               0.5},
72         pd_line_kw={'color': '#0d47a1', 'linewidth': 2.5, 'label': '
            PDP mean'}
73     )
74     ax.set_title(f"PDP + ICE: {feat}", fontsize=11, fontweight='bold
        ')
75     ax.set_xlabel(feat, fontsize=9)
76     ax.set_ylabel("Partial dependence", fontsize=9)
77     ax.grid(alpha=0.3)
78
79 plt.suptitle("Partial Dependence Plots with ICE Curves",
80              fontsize=14, fontweight='bold', y=1.01)
81 plt.tight_layout()
82 plt.savefig("pdp_ice.png", dpi=150, bbox_inches='tight')
83 plt.close()
84
85 # ----------------------------------------------------------------------
86 # 2. 2D PDP: interaction between two features
87 #    If the surface is not a simple sum of two 1D functions, there
88 #    is a genuine statistical interaction between the two features.
89 # ----------------------------------------------------------------------
90 fig, ax = plt.subplots(figsize=(8, 6))
91 disp_2d = PartialDependenceDisplay.from_estimator(
92     model,
93     X_tr_s,
94     features=[('days_late_last_payment', 'credit_utilisation_pct')],
95     kind='average',
```

```
    grid_resolution=30,
    ax=ax,
    n_jobs=-1
)
ax.set_title("2D PDP: Late Payment vs Credit Utilisation",
             fontsize=12, fontweight='bold')
plt.tight_layout()
plt.savefig("pdp_2d.png", dpi=150, bbox_inches='tight')
plt.close()
print("PDP plots saved.")
```

Individual Conditional Expectation (ICE) curves expose something that PDPs hide. A PDP averages over all samples; if some samples see a positive effect from a feature and others see a negative effect, those effects cancel out and the PDP appears flat. ICE curves plot the predicted outcome for each individual sample as the feature varies, revealing this heterogeneity. When ICE curves cross each other, a feature interaction is present.

The **2D PDP** extends this to pairs of features. The contour plot shows the model's predicted outcome as a function of two features simultaneously. Non-linear contours that cannot be decomposed into a sum of a row pattern and a column pattern indicate an interaction effect.

Interpretability methods are not only for explainability to stakeholders — they are essential debugging tools. If a SHAP summary plot shows a feature contributing positively that you know from domain expertise should contribute negatively, you have found a bug: possibly a data leakage issue, a target encoding done incorrectly, or a labelling error. Always validate model behaviour against domain knowledge before deployment.

5 Putting It All Together: End-to-End Evaluation Workflow

This section demonstrates a complete, production-grade evaluation workflow that combines all four pillars from a single coherent script. The pattern shown here is the template you should follow for any classification problem.

Full Evaluation Pipeline — Production Template

```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import shap
import warnings
warnings.filterwarnings('ignore')

from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import (
    StratifiedKFold, cross_validate, RandomizedSearchCV, cross_val_
        predict
)
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    classification_report, average_precision_score,
    roc_auc_score, balanced_accuracy_score, f1_score,
    precision_recall_curve
)
from scipy.stats import randint, loguniform, uniform

print("=" * 65)
print("  MODEL EVALUATION & VALIDATION  PRODUCTION WORKFLOW")
print("=" * 65)

#
# STEP 1: Data
#
X, y = make_classification(
    n_samples=3000, n_features=20, n_informative=10,
    weights=[0.88, 0.12], random_state=42
)
feature_names = [f"feature_{i:02d}" for i in range(X.shape[1])]
print(f"\nStep 1  Dataset: {X.shape[0]} samples, "
      f"{X.shape[1]} features, {100*y.mean():.1f}% positive")

# Reserve a true holdout set  touch this ONLY at the very end.
from sklearn.model_selection import train_test_split
X_dev, X_holdout, y_dev, y_holdout = train_test_split(
    X, y, test_size=0.15, random_state=42, stratify=y
)
print(f"       Dev set: {X_dev.shape[0]} samples | "
      f"Holdout: {X_holdout.shape[0]} samples")
```

```
46
47  #
48  # STEP 2: Hyperparameter tuning (inner CV loop)
49  #
50  print("\nStep 2  Hyperparameter Tuning (RandomizedSearchCV)...")
51
52  inner_cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=
        42)
53
54  pipeline = Pipeline([
55      ('scaler', StandardScaler()),
56      ('clf', GradientBoostingClassifier(random_state=42))
57  ])
58
59  param_dist = {
60      'clf__n_estimators' : randint(50, 300),
61      'clf__max_depth'    : randint(2, 6),
62      'clf__learning_rate': loguniform(1e-2, 0.3),
63      'clf__subsample'    : uniform(0.5, 0.5),
64      'clf__min_samples_leaf': randint(5, 30),
65  }
66
67  search = RandomizedSearchCV(
68      pipeline, param_dist,
69      n_iter=30,
70      scoring='average_precision',
71      cv=inner_cv,
72      n_jobs=-1,
73      random_state=42,
74      refit=True
75  )
76  search.fit(X_dev, y_dev)
77  best_pipeline = search.best_estimator_
78
79  print(f"       Best tuning AP (inner CV): {search.best_score_:.4f}")
80  print(f"       Best params: n_est={search.best_params_['clf__n_
        estimators']}, "
81        f"depth={search.best_params_['clf__max_depth']}, "
82        f"lr={search.best_params_['clf__learning_rate']:.4f}")
83
84  #
85  # STEP 3: Unbiased performance estimation (outer CV loop)
86  #
87  print("\nStep 3  Outer CV: Unbiased Performance Estimation...")
88
89  outer_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
90
91  # For the outer loop we re-run the search inside each fold to
92  # avoid optimism, but for speed we demonstrate with the best
        pipeline.
93  # In practice, use a nested CV (GridSearchCV inside cross_validate).
94  cv_results = cross_validate(
95      best_pipeline, X_dev, y_dev,
96      cv=outer_cv,
97      scoring={
98          'ap'      : 'average_precision',
99          'auc'     : 'roc_auc',
100         'f1'      : 'f1',
```

```
        'bal_acc': 'balanced_accuracy'
    },
    return_train_score=True,
    n_jobs=-1
)

print(f"\n       {'Metric':<18} {'Mean':>8} {'Std':>8} {'Overfit Gap':>12}")
print(f"       {'-'*48}")
for m in ['ap', 'auc', 'f1', 'bal_acc']:
    train_mean = cv_results[f'train_{m}'].mean()
    test_mean  = cv_results[f'test_{m}'].mean()
    test_std   = cv_results[f'test_{m}'].std()
    gap        = train_mean - test_mean
    print(f"       {m:<18} {test_mean:>8.4f} {test_std:>8.4f} {gap:>12.4f}")

#
# STEP 4: Threshold optimisation
#
print("\nStep 4  Threshold Optimisation...")

oof_proba = cross_val_predict(
    best_pipeline, X_dev, y_dev, cv=outer_cv,
    method='predict_proba'
)[:, 1]

prec, rec, thresholds = precision_recall_curve(y_dev, oof_proba)
# Use F2: recall matters 2x more than precision (missing positives costly)
f2_scores = (5 * prec[:-1] * rec[:-1]) / (4*prec[:-1] + rec[:-1] + 1e-9)
best_thresh_idx = np.argmax(f2_scores)
optimal_threshold = thresholds[best_thresh_idx]

print(f"       Optimal threshold (max F2): {optimal_threshold:.3f}")
print(f"       F2 at optimal: {f2_scores[best_thresh_idx]:.4f}")
print(f"       F2 at 0.50   : {f2_scores[np.argmin(np.abs(thresholds-0.5))]:.4f}")

#
# STEP 5: Final holdout evaluation (one-time, no peeking before this)
#
print("\nStep 5  Final Holdout Evaluation...")

best_pipeline.fit(X_dev, y_dev)
holdout_proba  = best_pipeline.predict_proba(X_holdout)[:, 1]
holdout_labels = (holdout_proba >= optimal_threshold).astype(int)

print(f"\n       *** FINAL HOLDOUT RESULTS ***")
print(f"       AUC-ROC           : {roc_auc_score(y_holdout, holdout_proba):.4f}")
print(f"       Average Precision: {average_precision_score(y_holdout, holdout_proba):.4f}")
print(f"       Balanced Accuracy: {balanced_accuracy_score(y_holdout, holdout_labels):.4f}")
```

```
print(f"       F1 Score          : {f1_score(y_holdout, holdout_
    labels):.4f}")
print()
print(classification_report(y_holdout, holdout_labels,
                            target_names=['Negative', 'Positive']))

#
# STEP 6: SHAP interpretability on holdout set
#
print("Step 6  SHAP Global Interpretability...")

fitted_clf    = best_pipeline.named_steps['clf']
fitted_scaler = best_pipeline.named_steps['scaler']
X_holdout_s   = fitted_scaler.transform(X_holdout)

explainer  = shap.TreeExplainer(fitted_clf)
shap_vals  = explainer.shap_values(X_holdout_s)

# Compute mean absolute SHAP and rank features
mean_abs_shap = np.abs(shap_vals).mean(axis=0)
feat_importance = pd.Series(mean_abs_shap, index=feature_names)\
                    .sort_values(ascending=False)

print("\n       Top 5 features by mean |SHAP|:")
for feat, val in feat_importance.head(5).items():
    print(f"         {feat:<20}: {val:.5f}")

shap.summary_plot(shap_vals, X_holdout,
                  feature_names=feature_names,
                  plot_type='bar', show=False, max_display=10)
plt.tight_layout()
plt.savefig("final_shap_importance.png", dpi=150, bbox_inches='tight
    ')
plt.close()
print("\n       SHAP importance plot saved.")
print("\n" + "="*65)
print("  EVALUATION COMPLETE")
print("="*65)
```

6 The Evaluation Checklist

Before declaring a model production-ready, work through the following checklist. Each item addresses a common failure mode described in this tutorial.

Data Integrity

- ☐ All preprocessing fitted inside a Pipeline or on training data only.
- ☐ No future data in training splits (especially for time-series).
- ☐ Class distribution verified in each fold (use StratifiedKFold).
- ☐ Holdout test set segregated at the start and not inspected until the final evaluation.

Cross-Validation

- ☐ CV strategy matches deployment scenario (StratifiedKFold for classification, TimeSeriesSplit for temporal data).
- ☐ Both mean and standard deviation of CV scores reported.
- ☐ Training scores compared to validation scores to diagnose overfitting.
- ☐ Performance trend across time-series folds inspected for concept drift.

Hyperparameter Tuning

- ☐ Tuning performed on development set only; holdout never used during tuning.
- ☐ Nested cross-validation used if reporting unbiased performance alongside tuning.
- ☐ Convergence plot inspected to confirm budget was sufficient.

Metrics

- ☐ Metric chosen based on business cost structure, not convenience.
- ☐ For imbalanced datasets: accuracy not used alone; AP, F1, and balanced accuracy reported.
- ☐ Decision threshold optimised for the chosen business objective.
- ☐ Precision-recall curve inspected, not just the single-threshold result.

Interpretability

- ☐ SHAP global summary consistent with domain knowledge.
- ☐ At least one local explanation generated and reviewed for a representative positive prediction and a representative negative prediction.
- ☐ PDP or ICE curves used to verify expected monotonicity of key features.

- ☐ Any surprising model behaviour investigated for data errors or leakage before proceeding.

7 Regression Metrics: Measuring Continuous Predictions

7.1 The Problem with a Single Number

Classification has the luxury of a confusion matrix — a two-dimensional structure that separates types of errors. Regression collapses the comparison between predicted and actual values into a single error distribution. The danger is that different metrics collapse that distribution differently, each hiding a different kind of problem. A model can have a low RMSE but a terrible MAPE on small values, or a good R^2 but systematically biased predictions in one region. Understanding each metric's blind spots is what separates a careful practitioner from someone who reports the first number that looks good.

7.2 Core Regression Metrics

Let y_i be the true value and $\hat{y}_i$ be the prediction, with n samples and mean true value $\bar{y}$.

Mean Absolute Error (MAE) is the average of the absolute differences. It is in the same unit as the target and gives every error equal weight:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Mean Squared Error (MSE) squares each error before averaging, which means large errors are penalised disproportionately. This is appropriate when large errors are especially costly:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Root Mean Squared Error (RMSE) is the square root of MSE, restoring the original unit and making it directly comparable to MAE:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Mean Absolute Percentage Error (MAPE) expresses error as a percentage of the true value, making it scale-independent and useful for comparing models across datasets with different magnitudes. It is undefined or explosive when $y_i \approx 0$:

$$\text{MAPE} = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

Symmetric MAPE (sMAPE) fixes the division-by-zero problem by dividing by the mean of the actual and predicted values:

$$\text{sMAPE} = \frac{100}{n} \sum_{i=1}^{n} \frac{2|y_i - \hat{y}_i|}{|y_i| + |\hat{y}_i|}$$

R^2 **(Coefficient of Determination)** measures the proportion of variance in the target explained by the model. A baseline model that always predicts $\bar{y}$ achieves $R^2 = 0$; a perfect model achieves $R^2 = 1$. Negative values mean the model is worse than the mean:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Adjusted R^2 penalises for the number of features p, preventing R^2 from artificially increasing when irrelevant features are added:

$$\bar{R}^2 = 1 - (1 - R^2) \cdot \frac{n - 1}{n - p - 1}$$

Metric	Use when	Caution
MAE	Robust to outliers; median-optimising	Not differentiable at 0; less sensitive to large errors
RMSE	Large errors are disproportionately costly	Heavily influenced by outliers; unit-sensitive
MAPE	Comparing across scales; percentage makes sense	Blows up near zero; biased toward under-prediction
sMAPE	MAPE context but targets near zero exist	Can still behave oddly for very small values
R^2	Explaining variance; comparing to baseline	Scale-free but can be misleading with non-linear models
Adj. R^2	Feature selection; multivariate regression	Still a summary statistic; inspect residuals

Table 5: Regression metric selection guide.

7.3 Residual Analysis: Beyond a Single Number

No single metric captures the full picture. Residual analysis — examining the distribution and structure of errors — is the gold standard for diagnosing regression models.

Comprehensive Regression Evaluation with Residual Diagnostics

```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
```

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_predict, KFold, cross_
    validate
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    mean_absolute_error, mean_squared_error,
    r2_score, mean_absolute_percentage_error
)

# ---------------------------------------------------------------------
# 1. California housing dataset: predict median house value
#    Target is in units of $100,000. ~20,000 samples, 8 features.
# ---------------------------------------------------------------------
housing = fetch_california_housing(as_frame=True)
X, y = housing.data, housing.target
print(f"Dataset: {X.shape[0]} samples, {X.shape[1]} features")
print(f"Target range: [{y.min():.2f}, {y.max():.2f}]  "
      f"Mean: {y.mean():.2f}  Std: {y.std():.2f}")

# ---------------------------------------------------------------------
# 2. Two models: linear baseline and gradient boosting
# ---------------------------------------------------------------------
models = {
    'Linear Regression': Pipeline([
        ('scaler', StandardScaler()),
        ('reg', LinearRegression())
    ]),
    'Gradient Boosting': Pipeline([
        ('scaler', StandardScaler()),
        ('reg', GradientBoostingRegressor(
            n_estimators=200, learning_rate=0.05,
            max_depth=5, random_state=42
        ))
    ])
}

cv = KFold(n_splits=5, shuffle=True, random_state=42)

def smape(y_true, y_pred):
    """Symmetric Mean Absolute Percentage Error."""
    denom = (np.abs(y_true) + np.abs(y_pred)) / 2
    return np.mean(np.abs(y_true - y_pred) / denom) * 100

results = {}
for name, pipe in models.items():
    # Out-of-fold predictions for unbiased metric computation
    oof_pred = cross_val_predict(pipe, X, y, cv=cv, n_jobs=-1)

    mae   = mean_absolute_error(y, oof_pred)
    rmse  = np.sqrt(mean_squared_error(y, oof_pred))
    mape  = mean_absolute_percentage_error(y, oof_pred) * 100
    smap  = smape(y.values, oof_pred)
    r2    = r2_score(y, oof_pred)

    # Adjusted R^2
    n, p  = X.shape
```

```
    adj_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)

    results[name] = {
        'oof_pred': oof_pred,
        'MAE': mae, 'RMSE': rmse,
        'MAPE': mape, 'sMAPE': smap,
        'R2': r2, 'Adj_R2': adj_r2
    }

# ------------------------------------------------------------------
# 3. Metric comparison table
# ------------------------------------------------------------------
print(f"\n{'Metric':<12}", end="")
for name in models:
    print(f"  {name:<22}", end="")
print()
print("-" * 60)
for metric in ['MAE', 'RMSE', 'MAPE', 'sMAPE', 'R2', 'Adj_R2']:
    print(f"{metric:<12}", end="")
    for name in models:
        val = results[name][metric]
        unit = "%" if metric in ['MAPE', 'sMAPE'] else ""
        print(f"  {val:>8.4f}{unit:<14}", end="")
    print()

# ------------------------------------------------------------------
# 4. Four-panel residual diagnostic plot (Gold Standard)
#    Panel A: Residuals vs Fitted   checks heteroscedasticity
#    Panel B: QQ plot               checks normality of errors
#    Panel C: Scale-Location        detects variance instability
#    Panel D: Error distribution    checks symmetry and tails
# ------------------------------------------------------------------
from scipy import stats

fig, axes = plt.subplots(2, 2, figsize=(13, 10))
best_name = 'Gradient Boosting'
oof   = results[best_name]['oof_pred']
resid = y.values - oof
std_resid = resid / resid.std()

# Panel A: Residuals vs Fitted
axes[0,0].scatter(oof, resid, alpha=0.3, s=8, color='#0d47a1')
axes[0,0].axhline(0, color='red', linewidth=1.5, linestyle='--')
# Add LOWESS-style trend
from numpy.polynomial import polynomial as P
z = np.polyfit(oof, resid, 3)
p = np.poly1d(z)
x_line = np.linspace(oof.min(), oof.max(), 200)
axes[0,0].plot(x_line, p(x_line), color='orange', linewidth=2)
axes[0,0].set_xlabel("Fitted Values", fontsize=11)
axes[0,0].set_ylabel("Residuals", fontsize=11)
axes[0,0].set_title("Residuals vs Fitted\n(should be random around
    0)",
                    fontsize=11, fontweight='bold')
axes[0,0].grid(alpha=0.3)

# Panel B: Normal Q-Q Plot
```

```
(osm, osr), (slope, intercept, r) = stats.probplot(std_resid, dist='norm')
axes[0,1].scatter(osm, osr, alpha=0.4, s=8, color='#0d47a1')
axes[0,1].plot(osm, slope*np.array(osm)+intercept, 'r--', linewidth=1.5)
axes[0,1].set_xlabel("Theoretical Quantiles", fontsize=11)
axes[0,1].set_ylabel("Sample Quantiles", fontsize=11)
axes[0,1].set_title("Normal Q-Q Plot\n(deviations = non-normal errors)",
                    fontsize=11, fontweight='bold')
axes[0,1].grid(alpha=0.3)

# Panel C: Scale-Location (sqrt of abs residuals vs fitted)
axes[1,0].scatter(oof, np.sqrt(np.abs(std_resid)), alpha=0.3, s=8,
                  color='#00838f')
axes[1,0].set_xlabel("Fitted Values", fontsize=11)
axes[1,0].set_ylabel(r"$\sqrt{|\text{Std. Residuals}|}$", fontsize=11)
axes[1,0].set_title("Scale-Location\n(flat line = homoscedastic)",
                    fontsize=11, fontweight='bold')
axes[1,0].grid(alpha=0.3)

# Panel D: Error distribution
axes[1,1].hist(resid, bins=60, color='#0d47a1', alpha=0.75, edgecolor='white',
               linewidth=0.3, density=True)
xr = np.linspace(resid.min(), resid.max(), 300)
axes[1,1].plot(xr, stats.norm.pdf(xr, resid.mean(), resid.std()),
               'r--', linewidth=2, label='Normal fit')
axes[1,1].set_xlabel("Residual", fontsize=11)
axes[1,1].set_ylabel("Density", fontsize=11)
axes[1,1].set_title(f"Error Distribution\n"
                    f"Skew={stats.skew(resid):.3f}   "
                    f"Kurt={stats.kurtosis(resid):.3f}",
                    fontsize=11, fontweight='bold')
axes[1,1].legend(fontsize=9)
axes[1,1].grid(alpha=0.3)

plt.suptitle(f"Residual Diagnostics  {best_name}",
             fontsize=14, fontweight='bold', y=1.01)
plt.tight_layout()
plt.savefig("residual_diagnostics.png", dpi=150, bbox_inches='tight')
plt.close()
print("\nResidual diagnostic plots saved.")

# ------------------------------------------------------------------
# 5. Prediction interval width analysis
#    Pinball / quantile loss: for each quantile tau, a model that
#    outputs the tau-quantile should have tau fraction of actuals below.
# ------------------------------------------------------------------
from sklearn.ensemble import GradientBoostingRegressor as GBR

quantiles = [0.1, 0.5, 0.9]
coverage_results = {}

for q in quantiles:
```

```
    q_model = Pipeline([
        ('scaler', StandardScaler()),
        ('reg', GBR(loss='quantile', alpha=q,
                    n_estimators=150, max_depth=4, random_state=42))
    ])
    q_pred = cross_val_predict(q_model, X, y, cv=cv, n_jobs=-1)
    coverage = np.mean(y.values <= q_pred)
    coverage_results[q] = coverage

print("\nQuantile calibration (target coverage vs actual coverage):")
for q, cov in coverage_results.items():
    print(f"  Q{int(q*100):>2}: target={q:.2f}  actual={cov:.3f}  "
          f"{'OK' if abs(q - cov) < 0.03 else 'MISCALIBRATED'}")
```

The four-panel residual plot is the equivalent of the confusion matrix for regression. The *Residuals vs Fitted* panel should show a random cloud centred at zero — any curve or funnel shape indicates the model is missing systematic structure. The *Q-Q plot* should follow the diagonal line — heavy tails mean the model is underpredicting the frequency of large errors. The *Scale-Location* plot should be flat — a rising trend means variance grows with the prediction (heteroscedasticity), which invalidates confidence intervals.

8 Model Calibration: When Probabilities Must Mean Something

8.1 The Calibration Problem

A classifier's predicted probability is not just a ranking device — in many business contexts it is used directly as a probability estimate. A credit scoring model that outputs 0.30 for a customer is implicitly claiming that 30% of customers with that profile will default. If in reality 60% of those customers default, the model is severely ***miscalibrated***, and any downstream decision made using the raw probability (e.g., expected loss calculation, premium pricing) will be systematically wrong.

Calibration is the statistical property that a model's predicted probability of p should match the observed frequency of the event across all predictions in the neighbourhood of p.

Formally, a model is perfectly calibrated if:

$$P(Y = 1 \mid \hat{p} = p) = p \quad \forall p \in [0, 1]$$

8.2 The Reliability Diagram

The ***reliability diagram*** (also called the calibration curve) is the primary visual tool. Predictions are grouped into bins by their predicted probability; for each bin, the fraction of actual positive outcomes is plotted against the mean predicted probability. A perfectly calibrated model lies on the $y = x$ diagonal.

8.3 The Brier Score

The ***Brier score*** is the mean squared error of probability predictions:

$$\text{BS} = \frac{1}{n}\sum_{i=1}^{n}(\hat{p}_i - y_i)^2$$

It ranges from 0 (perfect) to 1 (perfectly wrong). It penalises confident wrong predictions heavily and decomposes into:

$$\text{BS} = \underbrace{\text{Reliability}}_{\text{calibration error}} - \underbrace{\text{Resolution}}_{\text{sharpness}} + \underbrace{\text{Uncertainty}}_{\text{base rate variance}}$$

A model can have a low Brier score either by being well calibrated or by being sharp (confident). Reporting both Brier score and the reliability diagram separates the two contributions.

Calibration Analysis, Platt Scaling, and Isotonic Regression

```
import numpy as np
import matplotlib
matplotlib.use('Agg')
```

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.calibration import CalibratedClassifierCV, calibration_curve
from sklearn.model_selection import train_test_split, cross_val_predict, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import brier_score_loss

# ------------------------------------------------------------------------
# 1. Dataset and train/test split
# ------------------------------------------------------------------------
X, y = make_classification(
    n_samples=5000, n_features=20, n_informative=10,
    weights=[0.75, 0.25], random_state=42
)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)

# ------------------------------------------------------------------------
# 2. Gradient Boosting is often poorly calibrated because it was
#    optimised for ranking (log-loss), not for probability accuracy.
#    Random Forest is typically overconfident (predicted probs
#    cluster toward 0 and 1, not matching actual frequencies).
# ------------------------------------------------------------------------
models = {
    'Gradient Boosting (raw)': Pipeline([
        ('scaler', StandardScaler()),
        ('clf', GradientBoostingClassifier(
            n_estimators=200, max_depth=4, random_state=42
        ))
    ]),
    'Random Forest (raw)': Pipeline([
        ('scaler', StandardScaler()),
        ('clf', RandomForestClassifier(
            n_estimators=200, random_state=42
        ))
    ]),
}

# ------------------------------------------------------------------------
# 3. Calibrated versions
#    Platt Scaling (method='sigmoid'): fits a logistic regression
#    on top of the model's raw scores.  Works well when the
#    calibration curve is S-shaped.
#
#    Isotonic Regression (method='isotonic'): fits a non-parametric
#    monotonic function.  More flexible but needs more data to avoid
#    overfitting.  Use when the calibration error is non-monotonic.
#
#    cv='prefit': the base model is already fitted; only the
#    calibration layer is fitted on the calibration set.
# ------------------------------------------------------------------------
```

```
calibrated_models = {}
for name, pipe in models.items():
    pipe.fit(X_train, y_train)
    # Platt scaling
    calib_platt = CalibratedClassifierCV(pipe, method='sigmoid', cv='prefit')
    calib_platt.fit(X_test[:len(X_test)//2], y_test[:len(y_test)//2])
    calibrated_models[name + ' + Platt'] = calib_platt

    # Isotonic regression
    calib_iso = CalibratedClassifierCV(pipe, method='isotonic', cv='prefit')
    calib_iso.fit(X_test[:len(X_test)//2], y_test[:len(y_test)//2])
    calibrated_models[name + ' + Isotonic'] = calib_iso

# Use the remaining half of test set for evaluation
X_eval, y_eval = X_test[len(X_test)//2:], y_test[len(y_test)//2:]

# ------------------------------------------------------------------
# 4. Compute calibration curves and Brier scores
# ------------------------------------------------------------------
all_models = {**models, **calibrated_models}
eval_results = {}

for name, model in all_models.items():
    try:
        proba = model.predict_proba(X_eval)[:, 1]
    except Exception:
        proba = model.predict_proba(X_eval)[:, 1]

    frac_pos, mean_pred = calibration_curve(
        y_eval, proba, n_bins=10, strategy='uniform'
    )
    brier = brier_score_loss(y_eval, proba)
    eval_results[name] = {
        'proba': proba, 'frac_pos': frac_pos,
        'mean_pred': mean_pred, 'brier': brier
    }

# ------------------------------------------------------------------
# 5. Reliability diagram: 2 x 2 grid comparing raw vs calibrated
# ------------------------------------------------------------------
fig, axes = plt.subplots(1, 2, figsize=(13, 6))

colors_raw  = ['#0d47a1', '#00838f']
colors_cal  = ['#1565c0', '#e65100', '#1b5e20', '#880e4f']

for ax, base_name, color_raw in zip(
        axes, ['Gradient Boosting', 'Random Forest'], colors_raw):

    ax.plot([0, 1], [0, 1], 'k--', linewidth=1.2, label='Perfect calibration')

    # Raw model
    r = eval_results[base_name + ' (raw)']
    ax.plot(r['mean_pred'], r['frac_pos'], 's-', color=color_raw,
            linewidth=2, markersize=6,
```

```
            label=f"Raw  (Brier={r['brier']:.4f})")

    # Platt
    rp = eval_results[base_name + ' (raw) + Platt']
    ax.plot(rp['mean_pred'], rp['frac_pos'], 'o-', color='#e65100',
            linewidth=2, markersize=6,
            label=f"Platt (Brier={rp['brier']:.4f})")

    # Isotonic
    ri = eval_results[base_name + ' (raw) + Isotonic']
    ax.plot(ri['mean_pred'], ri['frac_pos'], '^-', color='#1b5e20',
            linewidth=2, markersize=6,
            label=f"Isotonic (Brier={ri['brier']:.4f})")

    ax.set_xlabel("Mean Predicted Probability", fontsize=11)
    ax.set_ylabel("Fraction of Positives", fontsize=11)
    ax.set_title(f"Reliability Diagram  {base_name.split(' (')[0]}",
                 fontsize=12, fontweight='bold')
    ax.legend(fontsize=8.5)
    ax.grid(alpha=0.3)
    ax.set_xlim(0, 1); ax.set_ylim(0, 1)

plt.tight_layout()
plt.savefig("calibration_curves.png", dpi=150, bbox_inches='tight')
plt.close()

# ------------------------------------------------------------------
# 6. Summary: Brier scores
# ------------------------------------------------------------------
print("Brier Score Summary (lower = better):")
for name, r in eval_results.items():
    print(f"  {name:<42}: {r['brier']:.5f}")
print("\nNote: calibration only improves probability accuracy, not")
print("discrimination (AUC). Always report both AUC and Brier score.
    ")
```

When calibration matters most: any time probabilities feed into a downstream system that treats them as actual probabilities — expected value calculations, risk-adjusted pricing, multi-model ensembles that average probabilities, or Bayesian updating. When the model output is used only as a ranking (e.g., top-K recommendations, ranked lists), calibration is irrelevant and AUC is the right metric.

Calibration and discrimination (AUC) are orthogonal properties. You can have a perfectly discriminating model with terrible calibration (a monotonic transformation of the probabilities preserves AUC but destroys calibration). Always report both. Use AUC to assess whether the model separates classes; use the Brier score and reliability diagram to assess whether the probabilities are trustworthy.

9 Statistical Significance Testing for Model Comparison

9.1 The Core Question: Is Model A Actually Better?

When Model A achieves an AUC of 0.872 and Model B achieves 0.861 on the same cross-validation folds, the natural question is: is this difference real, or is it sampling noise? Without a statistical test, you cannot answer that question. Choosing the better-looking model on a single evaluation is like flipping a coin twice and concluding it is biased toward heads.

The challenge in model comparison is that the observations (CV fold scores) are *not independent* — they share training data. Standard two-sample t-tests assume independence and are anti-conservative (too likely to declare a significant difference) in this setting.

9.2 McNemar's Test: Comparing Classifiers on the Same Test Set

McNemar's test compares two classifiers on the same set of test examples. It asks: of the examples where the two models disagree, does one model get more right than the other? It operates on the contingency table of per-sample correct/incorrect decisions and requires no distributional assumptions.

$$\chi^2 = \frac{(n_{01} - n_{10})^2}{n_{01} + n_{10}}$$

where n_{01} is the number of examples Model A gets wrong but Model B gets right, and n_{10} is the reverse.

9.3 The Corrected Resampled t-Test

For comparing models across cross-validation folds, Nadeau and Bengio (2003) proposed a correction to the paired t-test that accounts for the overlap in training data between folds. For k-fold CV with n total samples and $n_{\text{test}} = n/k$ test samples per fold:

$$t = \frac{\bar{d}}{\sqrt{\left(\frac{1}{k} + \frac{n_{\text{test}}}{n_{\text{train}}}\right) s_d^2}}$$

where $\bar{d}$ is the mean per-fold score difference and s_d^2 is the variance of those differences.

Statistical Model Comparison: McNemar and Corrected t-Test

```
import numpy as np
import pandas as pd
from scipy import stats
from scipy.stats import chi2
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier,
```

```
    RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold, cross_val_
    predict, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
from itertools import combinations

X, y = make_classification(
    n_samples=3000, n_features=20, n_informative=10,
    weights=[0.7, 0.3], random_state=42
)

cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

models = {
    'Logistic Regression': Pipeline([
        ('sc', StandardScaler()),
        ('clf', LogisticRegression(C=0.1, max_iter=1000))
    ]),
    'Random Forest': Pipeline([
        ('sc', StandardScaler()),
        ('clf', RandomForestClassifier(n_estimators=200, random_
            state=42))
    ]),
    'Gradient Boosting': Pipeline([
        ('sc', StandardScaler()),
        ('clf', GradientBoostingClassifier(
            n_estimators=200, learning_rate=0.05,
            max_depth=4, random_state=42
        ))
    ]),
}

# ------------------------------------------------------------------
# 1. Collect per-fold AUC scores and per-sample hard predictions
# ------------------------------------------------------------------
fold_scores  = {}
oof_labels   = {}
n = len(y)
n_splits = cv.get_n_splits()
n_test_per_fold = n // n_splits

for name, pipe in models.items():
    fold_aucs = cross_val_score(pipe, X, y, cv=cv, scoring='roc_auc'
        , n_jobs=-1)
    oof_preds = cross_val_predict(pipe, X, y, cv=cv, n_jobs=-1)  #
        hard labels
    fold_scores[name] = fold_aucs
    oof_labels[name]  = oof_preds
    print(f"{name:<25}: AUC={fold_aucs.mean():.4f} +/- {fold_aucs.
        std():.4f}")

# ------------------------------------------------------------------
# 2. Corrected Resampled t-Test (Nadeau-Bengio)
#    Compares two models using their per-fold score differences.
#    The correction term accounts for the non-independence of CV
```

```
    folds.
# ------------------------------------------------------------------
def corrected_resampled_t_test(scores_a, scores_b, n_total, n_test):
    """
    Nadeau-Bengio corrected paired t-test for cross-validation
       comparison.

    Parameters
    ----------
    scores_a, scores_b : arrays of per-fold scores
    n_total  : total number of samples in dataset
    n_test   : number of test samples per fold

    Returns
    -------
    t_stat, p_value
    """
    k      = len(scores_a)
    d      = scores_a - scores_b            # per-fold differences
    d_bar  = d.mean()
    s2_d   = d.var(ddof=1)                  # sample variance

    n_train = n_total - n_test
    # Correction factor: 1/k + n_test/n_train
    correction = (1.0 / k) + (n_test / n_train)
    se     = np.sqrt(correction * s2_d)

    if se == 0:
        return 0.0, 1.0
    t_stat = d_bar / se
    p_val  = 2 * stats.t.sf(np.abs(t_stat), df=k - 1)
    return t_stat, p_val

print("\n--- Corrected Resampled t-Test (Nadeau-Bengio) ---")
print(f"{'Comparison':<45} {'t-stat':>8} {'p-value':>10} {'
   Significant':>12}")
print("-" * 78)

model_names = list(models.keys())
for a, b in combinations(model_names, 2):
    t, p = corrected_resampled_t_test(
        fold_scores[a], fold_scores[b],
        n_total=n, n_test=n_test_per_fold
    )
    sig = "YES (p<0.05)" if p < 0.05 else "no"
    print(f"{a} vs {b:<20} {t:>8.3f} {p:>10.4f} {sig:>12}")

# ------------------------------------------------------------------
# 3. McNemar's Test: comparing hard-label decisions on same samples
#    Only meaningful for samples where the two models disagree.
# ------------------------------------------------------------------
def mcnemar_test(labels_a, labels_b, y_true):
    """
    Test whether two classifiers differ significantly in their
       errors.
    n01: A wrong, B right
    n10: A right, B wrong
    Returns chi-squared statistic and p-value (with continuity
```

```
        correction).
    """
    n01 = np.sum((labels_a != y_true) & (labels_b == y_true))
    n10 = np.sum((labels_a == y_true) & (labels_b != y_true))
    # Continuity-corrected McNemar
    if n01 + n10 == 0:
        return 0.0, 1.0
    chi2_stat = (abs(n01 - n10) - 1)**2 / (n01 + n10)
    p_val = 1 - chi2.cdf(chi2_stat, df=1)
    return chi2_stat, p_val

print("\n--- McNemar's Test (on out-of-fold hard labels) ---")
print(f"{'Comparison':<45} {'chi2':>8} {'p-value':>10} "
      f"{'n01':>6} {'n10':>6} {'Significant':>12}")
print("-" * 95)

for a, b in combinations(model_names, 2):
    chi2_val, p = mcnemar_test(oof_labels[a], oof_labels[b], y)
    n01 = np.sum((oof_labels[a] != y) & (oof_labels[b] == y))
    n10 = np.sum((oof_labels[a] == y) & (oof_labels[b] != y))
    sig = "YES (p<0.05)" if p < 0.05 else "no"
    print(f"{a} vs {b:<20} {chi2_val:>8.3f} {p:>10.4f} "
          f"{n01:>6} {n10:>6} {sig:>12}")

# ------------------------------------------------------------------
# 4. Multiple comparisons correction (Bonferroni)
#    When testing k pairs, the chance of a false positive rises.
#    Bonferroni adjusts the significance threshold to alpha/k.
# ------------------------------------------------------------------
n_comparisons = len(list(combinations(model_names, 2)))
alpha_bonf = 0.05 / n_comparisons
print(f"\nBonferroni-corrected significance threshold for {n_
    comparisons} "
      f"comparisons: {alpha_bonf:.4f}")
```

If you test 20 model variants and declare significance at $p < 0.05$, you expect roughly one false positive by chance alone. Always apply a correction: Bonferroni (divide threshold by number of tests, conservative) or Benjamini-Hochberg (controls the false discovery rate, less conservative). Report corrected p-values whenever comparing more than two models.

10 Bias & Fairness Evaluation

10.1 Why Fairness Is a Measurement Problem

Fairness is not a single property — it is a family of mathematical definitions that are often mutually contradictory. The practitioner's job is to: (1) understand which definition of fairness is appropriate for the business and regulatory context, (2) measure it, and (3) make explicit trade-offs when definitions conflict.

Before any fairness analysis, a critical distinction must be made. A ***protected attribute*** is a feature (race, gender, age, disability) that must not be the basis for a decision. Such features may be excluded from the model, but they must still be used to *evaluate* whether the model produces disparate outcomes.

10.2 Core Fairness Definitions

Demographic Parity (also: statistical parity): the model's positive prediction rate should be equal across groups. Used in hiring and lending where equal rates of opportunity are required.

$$P(\hat{Y} = 1 \mid A = 0) = P(\hat{Y} = 1 \mid A = 1)$$

Equalized Odds: both the true positive rate (recall) and false positive rate must be equal across groups. Used in criminal justice and medical screening where errors of both types must not be concentrated in one group.

$$P(\hat{Y} = 1 \mid Y = y, A = 0) = P(\hat{Y} = 1 \mid Y = y, A = 1) \quad \forall y \in \{0, 1\}$$

Equal Opportunity: a relaxation of equalized odds that only requires equal recall (true positive rate). Appropriate when false negatives are the primary concern.

Disparate Impact Ratio: a ratio metric used in US employment law (the "80% rule"). If the positive rate for the disadvantaged group is less than 80% of the advantaged group's rate, disparate impact is indicated.

$$\text{DIR} = \frac{P(\hat{Y} = 1 \mid A = 1)}{P(\hat{Y} = 1 \mid A = 0)} \quad (\text{should be} \geq 0.8)$$

Fairness Audit: Measuring and Visualising Bias

```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split,
    StratifiedKFold, cross_val_predict
from sklearn.pipeline import Pipeline
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (confusion_matrix, roc_auc_score,
                             precision_score, recall_score)

np.random.seed(42)
n = 4000

# ------------------------------------------------------------------
# 1. Simulated loan application dataset
#    Group A=0: majority group (60%)
#    Group A=1: minority group (40%)
#    Introduce historical bias: minority group has artificially
#    lower income in training data (reflecting past discrimination).
# ------------------------------------------------------------------
group = np.random.binomial(1, 0.4, n)   # 0=majority, 1=minority

income   = np.where(group == 0,
                    np.random.normal(65, 15, n),    # majority: mean $65k
                    np.random.normal(52, 14, n))    # minority: mean $52k

credit   = np.random.normal(680, 80, n) - 30 * group
dti      = np.random.beta(2, 5, n) + 0.05 * group
history  = np.random.gamma(4, 2, n) - 1.5 * group

X = np.column_stack([income, credit, dti, history,
                     np.random.randn(n, 6)])   # 6 noise features
feature_names = ['income', 'credit_score', 'dti', 'credit_history',
                 *[f'noise_{i}' for i in range(6)]]

# True default driven by financial fundamentals (fair ground truth)
logit = -0.04*income + 0.005*(750-credit) + 2*dti - 0.05*history
y     = (1/(1+np.exp(-logit + np.random.randn(n)*0.5)) > 0.5).astype
    (int)

print(f"Default rate  majority: {y[group==0].mean():.3f}  "
      f"minority: {y[group==1].mean():.3f}")

# ------------------------------------------------------------------
# 2. Train a model WITHOUT group as a feature (group-blind)
#    Fairness analysis uses group ONLY in evaluation, not training.
# ------------------------------------------------------------------
X_train, X_test, y_train, y_test, g_train, g_test = train_test_split
    (
    X, y, group, test_size=0.3, stratify=y, random_state=42
)

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', GradientBoostingClassifier(
        n_estimators=200, max_depth=4,
        learning_rate=0.05, random_state=42
    ))
])
pipeline.fit(X_train, y_train)
proba_test = pipeline.predict_proba(X_test)[:, 1]
pred_test  = pipeline.predict(X_test)
```

```
64
65 # ------------------------------------------------------------------
66 # 3. Per-group metrics
67 # ------------------------------------------------------------------
68 def group_metrics(y_true, y_pred, y_proba, groups):
69     """Compute fairness-relevant metrics per group."""
70     rows = []
71     for g_val in sorted(np.unique(groups)):
72         mask = groups == g_val
73         yt   = y_true[mask]
74         yp   = y_pred[mask]
75         ypr  = y_proba[mask]
76         cm   = confusion_matrix(yt, yp, labels=[0,1])
77         tn, fp, fn, tp = cm.ravel()
78         rows.append({
79             'group'         : g_val,
80             'n'             : mask.sum(),
81             'pos_rate'      : yp.mean(),        # demographic parity
                metric
82             'tpr'           : tp/(tp+fn+1e-9),  # recall / equal
                opportunity
83             'fpr'           : fp/(fp+tn+1e-9),  # false positive
                rate
84             'precision'     : precision_score(yt, yp, zero_division=
                0),
85             'auc'           : roc_auc_score(yt, ypr),
86             'false_neg_rate': fn/(fn+tp+1e-9),
87         })
88     return pd.DataFrame(rows)
89
90 gm = group_metrics(y_test, pred_test, proba_test, g_test)
91 print("\n--- Per-Group Metrics ---")
92 print(gm.set_index('group').round(4).to_string())
93
94 # ------------------------------------------------------------------
95 # 4. Fairness metrics derived from group metrics
96 # ------------------------------------------------------------------
97 g0 = gm[gm['group']==0].iloc[0]
98 g1 = gm[gm['group']==1].iloc[0]
99
100 print("\n--- Fairness Metrics ---")
101 dir_ratio  = g1['pos_rate'] / (g0['pos_rate'] + 1e-9)
102 demo_gap   = abs(g0['pos_rate'] - g1['pos_rate'])
103 tpr_gap    = abs(g0['tpr']      - g1['tpr'])
104 fpr_gap    = abs(g0['fpr']      - g1['fpr'])
105 eq_odds    = max(tpr_gap, fpr_gap)
106
107 print(f"  Disparate Impact Ratio (>=0.8 required) : {dir_ratio:.4f}
    "
108       f"{'PASS' if dir_ratio >= 0.8 else 'FAIL'}")
109 print(f"  Demographic Parity Gap  (|g0 - g1|)     : {demo_gap:.4f} "
110       f"{'OK' if demo_gap < 0.05 else 'CONCERN'}")
111 print(f"  Equal Opportunity Gap   (TPR diff)      : {tpr_gap:.4f} "
112       f"{'OK' if tpr_gap < 0.05 else 'CONCERN'}")
113 print(f"  Equalized Odds Violation (max gap)      : {eq_odds:.4f} "
114       f"{'OK' if eq_odds < 0.05 else 'CONCERN'}")
115
116 # ------------------------------------------------------------------
```

```
# 5. Threshold adjustment for equalized odds
#    Use a different decision threshold per group to equalise TPR.
#    This is the most common post-processing fairness intervention.
# ------------------------------------------------------------------
from sklearn.metrics import roc_curve

thresholds_explored = np.linspace(0.1, 0.9, 200)

# Find threshold per group that achieves TPR close to target
target_tpr = 0.70   # desired recall for positive class in both groups

best_thresh = {}
for g_val in [0, 1]:
    mask = g_test == g_val
    yt   = y_test[mask]
    ypr  = proba_test[mask]
    tprs = [recall_score(yt, (ypr >= t).astype(int), zero_division=0)
            for t in thresholds_explored]
    best_idx = np.argmin(np.abs(np.array(tprs) - target_tpr))
    best_thresh[g_val] = thresholds_explored[best_idx]

print(f"\n  Threshold adjustment for TPR={target_tpr}:")
print(f"    Group 0 threshold: {best_thresh[0]:.3f}")
print(f"    Group 1 threshold: {best_thresh[1]:.3f}")

adj_pred = np.where(
    g_test == 0,
    (proba_test >= best_thresh[0]).astype(int),
    (proba_test >= best_thresh[1]).astype(int)
)

gm_adj = group_metrics(y_test, adj_pred, proba_test, g_test)
print("\n--- After Threshold Adjustment ---")
print(gm_adj.set_index('group')[['pos_rate','tpr','fpr','precision']].round(4))

# ------------------------------------------------------------------
# 6. Visualise per-group metric comparison
# ------------------------------------------------------------------
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

metrics_to_plot = ['pos_rate', 'tpr', 'fpr', 'precision', 'auc']
labels = ['Pos. Rate', 'TPR', 'FPR', 'Precision', 'AUC']

for ax, (gm_df, title) in zip(axes, [
    (gm,     'Before Adjustment'),
    (gm_adj, 'After Threshold Adjustment')
]):
    x = np.arange(len(metrics_to_plot))
    w = 0.35
    vals0 = [gm_df[gm_df['group']==0].iloc[0][m] for m in metrics_to_plot]
    vals1 = [gm_df[gm_df['group']==1].iloc[0][m] for m in metrics_to_plot]

    ax.bar(x - w/2, vals0, w, label='Group 0 (majority)',
```

```
            color='#0d47a1', alpha=0.85)
    ax.bar(x + w/2, vals1, w, label='Group 1 (minority)',
            color='#e65100', alpha=0.85)
    ax.set_xticks(x); ax.set_xticklabels(labels, fontsize=10)
    ax.set_ylim(0, 1.1)
    ax.axhline(0.8, color='red', linestyle='--', linewidth=1,
               label='80% rule threshold')
    ax.set_title(title, fontsize=12, fontweight='bold')
    ax.legend(fontsize=8); ax.grid(axis='y', alpha=0.3)

plt.suptitle("Fairness Metrics Before and After Threshold Adjustment
    ",
             fontsize=13, fontweight='bold')
plt.tight_layout()
plt.savefig("fairness_metrics.png", dpi=150, bbox_inches='tight')
plt.close()
print("\nFairness visualisation saved.")
```

Chouldechova (2017) and Kleinberg et al. (2017) independently proved that demographic parity, equalized odds, and predictive parity (precision equality) *cannot all be satisfied simultaneously* unless base rates are equal across groups. Fairness analysis requires an explicit choice of which criterion to prioritise, determined by the ethical and legal context of the application, not by the algorithm.

11 Production Monitoring & Drift Detection

11.1 The Life Cycle Does Not End at Deployment

A model that performs well on the test set at deployment time may degrade over weeks or months as the world changes. This degradation has two root causes.

Data drift (or covariate shift) occurs when the distribution of input features $P(X)$ changes while the relationship $P(Y \mid X)$ remains stable. Example: a pandemic shifts the income distribution of loan applicants, but the relationship between income and default probability is unchanged.

Concept drift occurs when $P(Y \mid X)$ itself changes — the fundamental relationship between features and target has shifted. Example: a new regulation changes what constitutes fraud, so past labelling conventions no longer apply.

Both require monitoring, but they call for different responses: data drift may be handled by retraining on new data; concept drift may require redesigning the feature set or the label definition.

11.2 Population Stability Index (PSI)

The ***Population Stability Index*** is the most widely used metric for monitoring input drift in production. It was developed by credit risk practitioners and measures the divergence between a reference distribution and a current distribution.

$$\text{PSI} = \sum_{b=1}^{B} (q_b - p_b) \ln\left(\frac{q_b}{p_b}\right)$$

where p_b is the proportion of reference data in bin b and q_b is the corresponding proportion in the current data. PSI is a symmetrised version of the KL divergence. The standard thresholds are: PSI < 0.1 (stable), $0.1 \leq$ PSI < 0.2 (minor shift, monitor closely), PSI ≥ 0.2 (significant shift, investigate and consider retraining).

PSI, KS Drift Detection, and Performance Monitoring

```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from scipy import stats
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score

# ------------------------------------------------------------------
# 1. Simulate production lifecycle
#    Month 0-2: stable data (training distribution)
```

```
#    Month 3-5: gradual covariate shift (data drift)
#    Month 6-8: concept drift (relationship changes)
# ----------------------------------------------------------------------
np.random.seed(42)

def generate_batch(n, drift_factor=0.0, concept_drift=False):
    """Generate a batch of data with optional drift."""
    X = np.random.randn(n, 10)
    X[:, 0] += drift_factor * 2      # feature 0 shifts with drift
    X[:, 1] += drift_factor * 1.5    # feature 1 also shifts

    if not concept_drift:
        # Original relationship: feature 0 and 2 predict y
        logit = 1.5*X[:,0] + X[:,2] + 0.3*np.random.randn(n)
    else:
        # Concept drift: feature 0 now has OPPOSITE sign
        logit = -1.5*X[:,0] + X[:,2] + 0.3*np.random.randn(n)

    y = (1/(1 + np.exp(-logit)) > 0.5).astype(int)
    return X, y

# Training data (reference)
X_train, y_train = generate_batch(3000, drift_factor=0.0)

# Production batches (one per month)
months = list(range(9))
monthly_data = {
    m: generate_batch(
        300,
        drift_factor=max(0, (m-2)*0.4),        # drift starts month
            3
        concept_drift=(m >= 6)                   # concept drift
            from month 6
    )
    for m in months
}

# ----------------------------------------------------------------------
# 2. Train model on reference data
# ----------------------------------------------------------------------
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', GradientBoostingClassifier(n_estimators=150, random_
        state=42))
])
pipeline.fit(X_train, y_train)

# ----------------------------------------------------------------------
# 3. PSI function
# ----------------------------------------------------------------------
def compute_psi(reference, current, n_bins=10):
    """
    Compute Population Stability Index between two samples.
    Uses percentile-based bins from the reference distribution.
    """
    breakpoints = np.percentile(reference, np.linspace(0, 100, n_
        bins+1))
    breakpoints = np.unique(breakpoints)  # remove duplicates
```

```
71
72      ref_counts = np.histogram(reference, bins=breakpoints)[0]
73      cur_counts = np.histogram(current,   bins=breakpoints)[0]
74
75      # Convert to proportions, add small epsilon to avoid log(0)
76      eps = 1e-6
77      ref_pct = ref_counts / len(reference) + eps
78      cur_pct = cur_counts / len(current)   + eps
79
80      psi = np.sum((cur_pct - ref_pct) * np.log(cur_pct / ref_pct))
81      return psi
82
83  # ------------------------------------------------------------------
84  # 4. Month-by-month monitoring
85  # ------------------------------------------------------------------
86  reference_feature = X_train[:, 0]   # monitor feature 0
87  monitoring = []
88
89  for m in months:
90      X_m, y_m = monthly_data[m]
91      proba_m = pipeline.predict_proba(X_m)[:, 1]
92
93      # PSI on feature 0
94      psi_f0 = compute_psi(reference_feature, X_m[:, 0])
95
96      # KS test on feature 0 (alternative to PSI)
97      ks_stat, ks_pval = stats.ks_2samp(reference_feature, X_m[:, 0])
98
99      # AUC requires ground truth labels (only available with delay)
100     auc = roc_auc_score(y_m, proba_m)
101
102     # Score distribution PSI (model output drift)
103     ref_scores = pipeline.predict_proba(X_train[:300])[:, 1]
104     psi_score  = compute_psi(ref_scores, proba_m)
105
106     monitoring.append({
107         'month': m,
108         'psi_feature_0': psi_f0,
109         'ks_statistic' : ks_stat,
110         'ks_pvalue'    : ks_pval,
111         'psi_score'    : psi_score,
112         'auc'          : auc,
113     })
114
115 mon_df = pd.DataFrame(monitoring).set_index('month')
116
117 print("Monthly Monitoring Dashboard:")
118 print(mon_df.round(4).to_string())
119
120 # ------------------------------------------------------------------
121 # 5. Monitoring dashboard plot
122 # ------------------------------------------------------------------
123 fig, axes = plt.subplots(2, 2, figsize=(13, 9))
124
125 # PSI feature
126 ax = axes[0,0]
127 colors = ['#e65100' if v >= 0.2 else '#f57c00' if v >= 0.1
128           else '#1b5e20' for v in mon_df['psi_feature_0']]
```

```
ax.bar(mon_df.index, mon_df['psi_feature_0'], color=colors, alpha=0.85)
ax.axhline(0.1, color='orange', linestyle='--', linewidth=1.5, label='PSI=0.1 (caution)')
ax.axhline(0.2, color='red',    linestyle='--', linewidth=1.5, label='PSI=0.2 (alert)')
ax.set_title("PSI  Feature 0 (Data Drift)", fontweight='bold')
ax.set_xlabel("Month"); ax.set_ylabel("PSI")
ax.legend(fontsize=8); ax.grid(alpha=0.3)
ax.set_xticks(months)

# KS statistic
ax = axes[0,1]
ax.plot(mon_df.index, mon_df['ks_statistic'],
        marker='o', color='#0d47a1', linewidth=2)
ax.axhline(0.1, color='red', linestyle='--', label='KS=0.1 (alert)')
ax.fill_between(mon_df.index, 0, mon_df['ks_statistic'],
                alpha=0.15, color='#0d47a1')
ax.set_title("KS Statistic  Feature 0", fontweight='bold')
ax.set_xlabel("Month"); ax.set_ylabel("KS Statistic")
ax.legend(fontsize=8); ax.grid(alpha=0.3)
ax.set_xticks(months)

# Score distribution PSI
ax = axes[1,0]
ax.bar(mon_df.index, mon_df['psi_score'], color='#00838f', alpha=0.85)
ax.axhline(0.1, color='orange', linestyle='--', linewidth=1.5)
ax.axhline(0.2, color='red',    linestyle='--', linewidth=1.5)
ax.set_title("PSI  Model Score Distribution", fontweight='bold')
ax.set_xlabel("Month"); ax.set_ylabel("PSI")
ax.grid(alpha=0.3); ax.set_xticks(months)

# AUC over time
ax = axes[1,1]
ax.plot(mon_df.index, mon_df['auc'],
        marker='s', color='#1b5e20', linewidth=2.5, markersize=8)
ax.fill_between(mon_df.index, mon_df['auc'].iloc[0] - 0.03,
                mon_df['auc'], alpha=0.2, color='#1b5e20')
ax.axhline(mon_df['auc'].iloc[0] * 0.95, color='red', linestyle='--',
           label='5% degradation threshold')
ax.set_title("Model AUC Over Time", fontweight='bold')
ax.set_xlabel("Month"); ax.set_ylabel("AUC")
ax.legend(fontsize=9); ax.grid(alpha=0.3)
ax.set_xticks(months)

plt.suptitle("Production Monitoring Dashboard", fontsize=14,
             fontweight='bold', y=1.01)
plt.tight_layout()
plt.savefig("monitoring_dashboard.png", dpi=150, bbox_inches='tight')
plt.close()
print("\nMonitoring dashboard saved.")
```

The monitoring dashboard separates the *early warning signals* (PSI on raw features, visible immediately in production) from the *lagging performance signal* (AUC, which requires

ground truth labels that may arrive weeks later). The PSI on the model's output score distribution is the most valuable early warning: it detects shifts in how the model is scoring the population even before labels arrive.

In a real deployment, true labels are almost never available immediately. Monitor the model's score distribution using PSI on a weekly or daily cadence. When PSI ≥ 0.2, trigger a manual review. When PSI ≥ 0.2 and labels arrive confirming AUC degradation, trigger retraining. Build this alerting into your MLOps pipeline from day one.

12 Learning Curves & Diagnosing Underfitting vs Overfitting

12.1 The Bias-Variance Trade-off in Practice

Every model lives on a spectrum between two failure modes.

High bias (underfitting): the model is too simple to capture the patterns in the data. Both training error and test error are high. The model makes systematic errors — the same type of mistake regardless of how much data it sees. Solution: increase model complexity or engineer better features.

High variance (overfitting): the model is too complex and has memorised the training data. Training error is low but test error is high. Performance improves with more data. Solution: regularisation, dropout, pruning, or more data.

Learning curves are the primary tool for diagnosing which failure mode dominates.

12.2 Reading Learning Curves

A ***learning curve*** plots training and validation performance as a function of training set size. The diagnosis is visual:

- **Both curves converge to a high error plateau**: high bias. More data will not help. The model needs to be more complex.
- **Large gap between training and validation curves**: high variance. The model would benefit from more data. If data cannot be obtained, apply regularisation.
- **Curves converge to a low error**: good fit. The model has learned the signal without overfitting.

Learning Curves and Complexity Curves

```
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import learning_curve, validation_curve
    , StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

X, y = make_classification(
    n_samples=3000, n_features=20, n_informative=8,
    n_redundant=4, weights=[0.65, 0.35], random_state=42
)

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

```
19
20  # ----------------------------------------------------------------------
21  # 1. Learning curves for three models
22  #    train_sizes: fractions of training data to use in each step.
23  #    Returns arrays of shape (n_steps, n_cv_folds).
24  # ----------------------------------------------------------------------
25  models = {
26      'Logistic Regression\n(High Bias)': Pipeline([
27          ('sc', StandardScaler()),
28          ('clf', LogisticRegression(C=0.001, max_iter=1000))  # over-
                regularised
29      ]),
30      'Gradient Boosting\n(Good Fit)': Pipeline([
31          ('sc', StandardScaler()),
32          ('clf', GradientBoostingClassifier(
33              n_estimators=100, max_depth=3,
34              learning_rate=0.1, random_state=42
35          ))
36      ]),
37      'Decision Tree\n(High Variance)': Pipeline([
38          ('sc', StandardScaler()),
39          ('clf', DecisionTreeClassifier(max_depth=None, random_state=
                42))
40      ]),
41  }
42
43  train_sizes = np.linspace(0.05, 1.0, 15)
44
45  fig, axes = plt.subplots(1, 3, figsize=(15, 5), sharey=True)
46
47  for ax, (name, pipe) in zip(axes, models.items()):
48      sizes, train_scores, val_scores = learning_curve(
49          pipe, X, y,
50          train_sizes=train_sizes,
51          cv=cv,
52          scoring='roc_auc',
53          n_jobs=-1,
54          return_times=False
55      )
56
57      # Mean and std across folds
58      train_mean = train_scores.mean(axis=1)
59      train_std  = train_scores.std(axis=1)
60      val_mean   = val_scores.mean(axis=1)
61      val_std    = val_scores.std(axis=1)
62
63      ax.plot(sizes, train_mean, 'o-', color='#0d47a1', linewidth=2,
64              markersize=5, label='Training AUC')
65      ax.fill_between(sizes, train_mean - train_std,
66                      train_mean + train_std, alpha=0.15, color='#0
                            d47a1')
67
68      ax.plot(sizes, val_mean, 's-', color='#e65100', linewidth=2,
69              markersize=5, label='Validation AUC')
70      ax.fill_between(sizes, val_mean - val_std,
71                      val_mean + val_std, alpha=0.15, color='#e65100')
72
73      gap = (train_mean[-1] - val_mean[-1])
```

```
    ax.set_title(f"{name}\nFinal gap: {gap:.3f}", fontsize=11,
                 fontweight='bold')
    ax.set_xlabel("Training set size", fontsize=10)
    ax.legend(fontsize=9)
    ax.grid(alpha=0.3)
    ax.set_ylim(0.4, 1.05)

axes[0].set_ylabel("AUC", fontsize=11)
plt.suptitle("Learning Curves: High Bias vs Good Fit vs High Variance",
             fontsize=13, fontweight='bold', y=1.02)
plt.tight_layout()
plt.savefig("learning_curves.png", dpi=150, bbox_inches='tight')
plt.close()

# ------------------------------------------------------------------
# 2. Validation curve (Complexity Curve)
#    Vary a single hyperparameter and observe how training and
#    validation performance change.  This reveals the sweet spot
#    between underfitting and overfitting for that parameter.
# ------------------------------------------------------------------
param_range = np.arange(1, 10)   # max_depth from 1 to 9

pipe_dt = Pipeline([
    ('sc', StandardScaler()),
    ('clf', DecisionTreeClassifier(random_state=42))
])

train_scores_vc, val_scores_vc = validation_curve(
    pipe_dt, X, y,
    param_name='clf__max_depth',
    param_range=param_range,
    cv=cv,
    scoring='roc_auc',
    n_jobs=-1
)

fig, ax = plt.subplots(figsize=(9, 5))

tr_m = train_scores_vc.mean(axis=1)
tr_s = train_scores_vc.std(axis=1)
va_m = val_scores_vc.mean(axis=1)
va_s = val_scores_vc.std(axis=1)

ax.plot(param_range, tr_m, 'o-', color='#0d47a1', lw=2, label='Training AUC')
ax.fill_between(param_range, tr_m - tr_s, tr_m + tr_s,
                alpha=0.15, color='#0d47a1')
ax.plot(param_range, va_m, 's-', color='#e65100', lw=2, label='Validation AUC')
ax.fill_between(param_range, va_m - va_s, va_m + va_s,
                alpha=0.15, color='#e65100')

best_depth = param_range[np.argmax(va_m)]
ax.axvline(best_depth, color='green', linestyle=':', lw=1.5,
           label=f'Best depth={best_depth}')

ax.set_xlabel("max_depth", fontsize=12)
```

```
ax.set_ylabel("AUC", fontsize=12)
ax.set_title("Validation Curve  Decision Tree max\_depth\n"
             "(Left=underfit, Right=overfit, Peak=sweet spot)",
             fontsize=12, fontweight='bold')
ax.legend(fontsize=10)
ax.grid(alpha=0.3)
ax.set_xticks(param_range)
plt.tight_layout()
plt.savefig("validation_curve.png", dpi=150, bbox_inches='tight')
plt.close()
print("Learning and validation curves saved.")

# ------------------------------------------------------------------
# 3. Diagnosis summary
# ------------------------------------------------------------------
print("\n--- Bias-Variance Diagnosis ---")
for name, pipe in models.items():
    sizes, tr, val = learning_curve(
        pipe, X, y, train_sizes=[0.8, 1.0],
        cv=cv, scoring='roc_auc', n_jobs=-1
    )
    tr_final  = tr[-1].mean()
    val_final = val[-1].mean()
    gap       = tr_final - val_final
    if tr_final < 0.75 and val_final < 0.75:
        diagnosis = "HIGH BIAS (underfit)"
    elif gap > 0.10:
        diagnosis = "HIGH VARIANCE (overfit)"
    else:
        diagnosis = "Good fit"
    clean = name.replace('\n', ' ')
    print(f"  {clean:<35}: train={tr_final:.3f}  val={val_final:.3f}  "
          f"gap={gap:.3f}  -> {diagnosis}")
```

12.3 The Validation Curve: Finding the Complexity Sweet Spot

The ***validation curve*** is the natural complement to the learning curve. Instead of varying the amount of data, it varies a single hyperparameter controlling complexity. The training AUC always increases with complexity; the validation AUC rises, peaks, and then falls as the model begins to overfit. The peak of the validation curve is the optimal complexity setting — this is what hyperparameter tuning is searching for, but the validation curve makes it visually explicit.

13 Multiclass & Multi-label Metrics

13.1 Beyond Binary Classification

Many real problems involve more than two classes: document categorisation, product defect type classification, disease subtype identification, sentiment with multiple levels. The metrics from binary classification extend to multiclass settings, but the averaging strategy becomes a critical decision that changes what is being measured.

13.2 Averaging Strategies

Given per-class precision P_c, recall R_c, and F_1^c for each class $c \in \{1, \ldots, K\}$:

Macro averaging: unweighted mean across classes. Every class contributes equally regardless of size. Use this when you care equally about rare and common classes.

$$F_1^{\text{macro}} = \frac{1}{K} \sum_{c=1}^{K} F_1^c$$

Weighted averaging: weighted mean by class support (number of true instances). Reflects the contribution of each class proportionally. Use this when larger classes matter more.

$$F_1^{\text{weighted}} = \frac{1}{\sum_c n_c} \sum_{c=1}^{K} n_c \cdot F_1^c$$

Micro averaging: aggregates TP, FP, FN across all classes before computing. For precision and recall, micro-average equals accuracy when all predictions are considered.

13.3 Cohen's Kappa: Agreement Beyond Chance

Cohen's Kappa measures agreement between predicted and actual labels, corrected for what would be expected by chance. It ranges from -1 (perfect disagreement) through 0 (chance agreement) to 1 (perfect agreement):

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

where p_o is the observed accuracy and p_e is the expected accuracy under chance (computed from marginal distributions). Kappa is especially useful for imbalanced multiclass problems where accuracy is misleading.

Comprehensive Multiclass Evaluation

```
import numpy as np
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

```
import seaborn as sns
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import StratifiedKFold, cross_val_
    predict
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, label_binarize
from sklearn.metrics import (
    classification_report, confusion_matrix,
    cohen_kappa_score, roc_auc_score,
    f1_score, ConfusionMatrixDisplay
)

# ------------------------------------------------------------------
# 1. 4-class dataset: imbalanced (mirrors real document
    categorisation)
# ------------------------------------------------------------------
X, y = make_classification(
    n_samples=3000, n_features=20, n_informative=12,
    n_classes=4, n_clusters_per_class=1,
    weights=[0.4, 0.3, 0.2, 0.1],   # class 3 is rare
    random_state=42
)
class_names = ['Class A', 'Class B', 'Class C', 'Class D']
print(f"Class distribution: {np.bincount(y)}")

# ------------------------------------------------------------------
# 2. Model and out-of-fold predictions
# ------------------------------------------------------------------
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', GradientBoostingClassifier(
        n_estimators=200, max_depth=4,
        learning_rate=0.05, random_state=42
    ))
])

oof_labels = cross_val_predict(pipeline, X, y, cv=cv, n_jobs=-1)
oof_proba  = cross_val_predict(pipeline, X, y, cv=cv,
                               method='predict_proba', n_jobs=-1)

# ------------------------------------------------------------------
# 3. Full classification report (per-class precision/recall/F1)
# ------------------------------------------------------------------
print("\nClassification Report:")
print(classification_report(y, oof_labels, target_names=class_names)
    )

# ------------------------------------------------------------------
# 4. Aggregate metrics with different averaging strategies
# ------------------------------------------------------------------
for avg in ['macro', 'weighted', 'micro']:
    f1 = f1_score(y, oof_labels, average=avg)
    print(f"F1 ({avg:<8}): {f1:.4f}")

kappa = cohen_kappa_score(y, oof_labels)
print(f"Cohen's Kappa  : {kappa:.4f}  "
```

```
        f"({'good' if kappa > 0.6 else 'moderate' if kappa > 0.4 else 'poor'})")

# ------------------------------------------------------------------------
# 5. Multiclass ROC-AUC (One-vs-Rest)
#    For multiclass AUC, we treat each class as binary (one vs. rest)
#    and average.  macro averages class AUCs equally; weighted by support.
# ------------------------------------------------------------------------
auc_macro    = roc_auc_score(y, oof_proba, multi_class='ovr', average='macro')
auc_weighted = roc_auc_score(y, oof_proba, multi_class='ovr', average='weighted')
print(f"AUC-OVR (macro)   : {auc_macro:.4f}")
print(f"AUC-OVR (weighted): {auc_weighted:.4f}")

# ------------------------------------------------------------------------
# 6. Normalised confusion matrix
#    Normalising by true labels (rows) shows recall per class.
#    Normalising by predicted labels (cols) shows precision per class.
# ------------------------------------------------------------------------
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

cm_raw  = confusion_matrix(y, oof_labels)
cm_norm = confusion_matrix(y, oof_labels, normalize='true')

for ax, cm, title, fmt in zip(
    axes,
    [cm_raw, cm_norm],
    ['Confusion Matrix (counts)', 'Confusion Matrix (recall-normalised)'],
    ['d', '.2f']
):
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                                  display_labels=class_names)
    disp.plot(ax=ax, colorbar=False, cmap='Blues', values_format=fmt)
    ax.set_title(title, fontsize=12, fontweight='bold')
    ax.tick_params(axis='x', rotation=20)

plt.tight_layout()
plt.savefig("multiclass_confusion.png", dpi=150, bbox_inches='tight')
plt.close()

# ------------------------------------------------------------------------
# 7. Per-class metrics visualisation
# ------------------------------------------------------------------------
report = classification_report(y, oof_labels,
                               target_names=class_names, output_dict=True)
metrics_df = pd.DataFrame({
    cls: {
        'Precision': report[cls]['precision'],
        'Recall'   : report[cls]['recall'],
        'F1'       : report[cls]['f1-score'],
```

```
        'Support'   : report[cls]['support'],
    }
    for cls in class_names
}).T

fig, ax = plt.subplots(figsize=(9, 5))
x = np.arange(len(class_names))
w = 0.25

ax.bar(x - w, metrics_df['Precision'], w, label='Precision',
       color='#0d47a1', alpha=0.85)
ax.bar(x,     metrics_df['Recall'],    w, label='Recall',
       color='#00838f', alpha=0.85)
ax.bar(x + w, metrics_df['F1'],        w, label='F1 Score',
       color='#e65100', alpha=0.85)

ax.set_xticks(x)
ax.set_xticklabels(
    [f"{c}\n(n={int(metrics_df.loc[c,'Support'])})" for c in class_
       names],
    fontsize=10
)
ax.set_ylim(0, 1.1)
ax.set_ylabel("Score", fontsize=11)
ax.set_title(f"Per-class Metrics (Kappa={kappa:.3f})",
             fontsize=12, fontweight='bold')
ax.legend(fontsize=10)
ax.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.savefig("multiclass_perclass.png", dpi=150, bbox_inches='tight')
plt.close()
print("\nMulticlass visualisations saved.")

# ------------------------------------------------------------------
# 8. Which averaging strategy to report?
# ------------------------------------------------------------------
print("\nAveraging Strategy Guide:")
print(f"  macro    F1={f1_score(y,oof_labels,average='macro'):.4f}
    "
      "-- equal weight to all classes (good for rare class
         monitoring)")
print(f"  weighted F1={f1_score(y,oof_labels,average='weighted'):.4f
    }  "
      "-- weight by support (good for overall performance)")
print(f"  micro    F1={f1_score(y,oof_labels,average='micro'):.4f}
    "
      "-- global TP/FP/FN (equals accuracy when all samples counted)
         ")
print(f"  Kappa={kappa:.4f}  -- corrected for chance (robust to
    imbalance)")
```

13.4 The Normalised Confusion Matrix as a Debugging Tool

The raw confusion matrix shows counts; the normalised version (row-normalised by true class) shows recall per class. For a 4-class problem, reading down each column of the normalised matrix reveals which classes are being confused with each other. If Class

D (rare) is being predicted as Class A 60% of the time, the model has not learned a discriminative representation of Class D — a signal to collect more examples, oversample, or apply class-weighted loss.

Situation	**Recommended Metric**	**Averaging**
All classes equally important, balanced	Accuracy or Macro F1	Macro
Rare class matters most	Macro F1, per-class recall	Macro
Overall performance for a mixed audience	Weighted F1	Weighted
Comparing to human annotators	Cohen's Kappa	N/A
Probabilistic ranking across classes	Multiclass AUC (OVR)	Macro/Weighted

Table 6: Metric selection guide for multiclass problems.

Regression Models

- ☐ Reported MAE *and* RMSE (sensitivity to outliers differs).
- ☐ R^2 reported alongside residual plots — not in isolation.
- ☐ Four-panel residual diagnostic completed (patterns indicate missing structure).
- ☐ MAPE or sMAPE used only when target values are not near zero.

Calibration

- ☐ If probabilities are used directly (not just for ranking): reliability diagram inspected.
- ☐ Brier score reported alongside AUC.
- ☐ Calibration applied (Platt or isotonic) if reliability curve deviates from diagonal.

Statistical Testing

- ☐ Model comparisons confirmed with corrected resampled t-test or McNemar's test.
- ☐ Bonferroni correction applied when comparing more than two models.
- ☐ Difference in CV means reported with confidence interval, not just point estimate.

Fairness & Bias

- ☐ Protected attributes identified and excluded from model training.

- ☐ Per-group metrics computed for all relevant protected groups.
- ☐ Disparate Impact Ratio checked (≥ 0.8 required in most jurisdictions).
- ☐ Chosen fairness criterion (demographic parity vs equalized odds) documented and justified.

Production Monitoring

- ☐ PSI monitoring pipeline built before deployment, not after.
- ☐ Score distribution PSI and feature PSI monitored separately.
- ☐ Retraining triggers defined (e.g., PSI ≥ 0.2 or AUC degradation $> 5\%$).
- ☐ Baseline reference distributions saved at deployment time.

Bias-Variance and Learning Curves

- ☐ Learning curves generated and inspected before finalising model complexity.
- ☐ Validation curve run for at least one key hyperparameter.
- ☐ Diagnosis (high bias / high variance / good fit) documented and acted upon.

Multiclass

- ☐ Averaging strategy (macro/weighted/micro) chosen based on class balance and business importance.
- ☐ Per-class precision, recall, and F1 inspected — not just aggregate metrics.
- ☐ Normalised confusion matrix inspected for inter-class confusions.
- ☐ Cohen's Kappa reported when comparing to human baseline or annotator agreement.

Appendix: Quick Reference

Key Formulas

$$\text{Precision} = \frac{TP}{TP + FP} \tag{1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2}$$

$$F_\beta = (1 + \beta^2) \cdot \frac{P \cdot R}{\beta^2 P + R} \tag{3}$$

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right) \tag{4}$$

$$\text{SHAP: } f(\mathbf{x}) = \mathbb{E}[f(X)] + \textstyle\sum_j \phi_j \tag{5}$$

$$\text{PDP: } \hat{f}_j(x_j) = \frac{1}{n}\sum_{i=1}^{n} \hat{f}(x_j, \mathbf{x}_{-j}^{(i)}) \tag{6}$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{7}$$

$$\text{Brier Score} = \frac{1}{n}\sum_{i=1}^{n}(\hat{p}_i - y_i)^2 \tag{8}$$

$$\text{PSI} = \sum_b (q_b - p_b)\ln(q_b/p_b) \tag{9}$$

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{10}$$

Library Quick-Install

Required packages

```
pip install scikit-learn shap lime scikit-optimize scipy matplotlib
    pandas numpy seaborn
```

Key Classes and Functions

Class / Function	Purpose
`StratifiedKFold`	K-fold with class distribution preserved
`TimeSeriesSplit`	Walk-forward CV for temporal data
`cross_validate`	Multi-metric CV with train/test scores
`cross_val_predict`	Out-of-fold predictions for full-dataset evaluation
`Pipeline`	Chain preprocessing + model (prevents leakage)
`GridSearchCV`	Exhaustive hyperparameter search
`RandomizedSearchCV`	Stochastic hyperparameter search
`BayesSearchCV`	Bayesian hyperparameter optimisation (skopt)
`average_precision_score`	Area under precision-recall curve
`balanced_accuracy_score`	Accuracy corrected for class imbalance
`precision_recall_curve`	Threshold-free P-R analysis
`shap.TreeExplainer`	Fast SHAP for tree models
`shap.summary_plot`	Global SHAP visualisation
`LimeTabularExplainer`	Local LIME explanation for tabular data
`PartialDependenceDisplay`	PDP and ICE plots
`mean_absolute_error`	MAE for regression
`r2_score`	Coefficient of determination (R^2)
`CalibratedClassifierCV`	Platt scaling / isotonic regression
`calibration_curve`	Reliability diagram data
`brier_score_loss`	Probability calibration quality
`learning_curve`	Train/val score vs training set size
`validation_curve`	Train/val score vs hyperparameter value
`cohen_kappa_score`	Agreement metric for multiclass
`roc_auc_score`	AUC-ROC, supports multiclass OVR
`ConfusionMatrixDisplay`	Normalised multiclass confusion matrix

Acknowledgments

Esmaeil Rezaei, Ph.D.

Thank you for reading this guide! I hope this tutorial has helped you understand how to preprocess text data for transformer models. If you found this useful, I would greatly appreciate your support by connecting with me on my professional networks.

GitHub
esmaeil-rezaei

LinkedIn
esmaeil-rezaei

Feel free to explore my repositories on GitHub for more machine learning projects and connect with me on LinkedIn to stay updated on my latest work. Your feedback and suggestions are always welcome!