Lesson 28: Class Imbalance

Data Science with Python – BSc Course

Data Science Program

BSc Course

45 Minutes

Previously on L27...

Last Time: Measuring Quality

- Confusion matrix: TP, FP, FN, TN
- Precision, Recall, F1-score
- ROC curve and AUC
- Threshold selection

This Time: When 99% Fails

- What if 99.9% of data is ONE class?
- Why accuracy becomes meaningless
- Techniques to find the rare events
- Cost-aware decision making

L27 gave us the tools to measure – L28 shows why standard metrics fail on imbalanced data

Learning Objectives

The Problem: Fraud occurs in 0.1% of transactions. Default in 2% of loans. How do we train models when positive cases are extremely rare?

After this lesson, you will be able to:

1. **Diagnose** imbalanced datasets and explain why accuracy misleads
2. **Apply** SMOTE to create synthetic minority samples
3. **Use** class weights to rebalance training without resampling
4. **Evaluate** models fairly with PR curves and cost-based thresholds

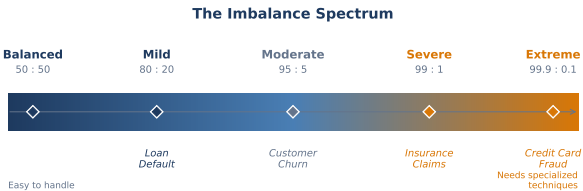
---

Finance Application: Fraud detection, default prediction, rare event modeling

The Imbalance Spectrum

Imbalance Is Everywhere in Finance

- Most real-world classification problems are imbalanced
- The harder the imbalance, the more specialized our approach must be

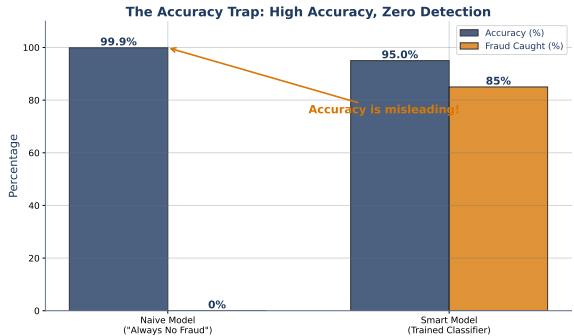


Key insight: Different imbalance levels require different techniques

Why Standard Models Fail

The Needle-in-a-Haystack Problem

- Classify everything as "hay" $\rightarrow$ right 99.99% of the time
- But you **never find the needle** – which was the whole point!

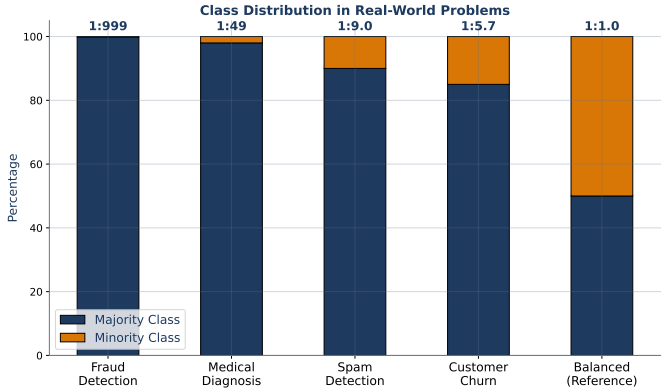


A model predicting "no fraud" always gets 99.9% accuracy – but catches zero fraud

Class Distribution in Finance

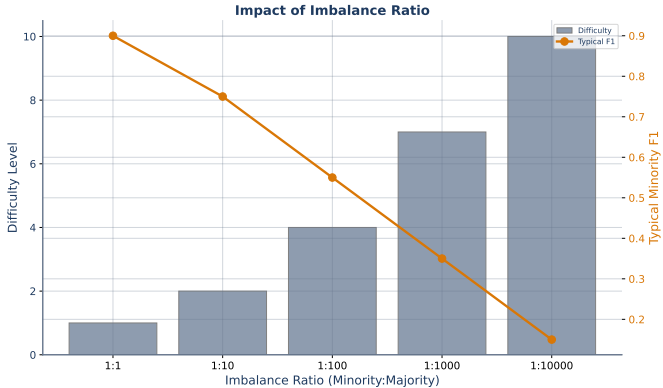
Real-World Imbalance Ratios

- Credit card fraud: 0.1% positive (1:1000)
- Loan defaults: 2–5% positive (1:20 to 1:50)



Challenge: Standard models optimize for majority class accuracy

Impact of Imbalance on Models

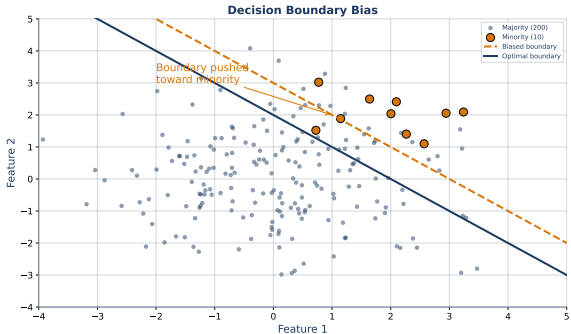


As imbalance increases, recall for minority class drops dramatically

Decision Boundary Bias

Why Standard Models Fail

- The boundary shifts toward the minority class to minimize total errors
- Result: minority samples are consistently misclassified



Standard classifiers push boundary toward minority class to minimize total errors

Three Solution Categories

How Do We Fix This?

1. **Resampling:** Change the data
   - Oversample minority (SMOTE) or undersample majority
2. **Reweighting:** Change the loss function
   - Make minority errors cost more (`class_weight`)
3. **Evaluation:** Change the metric
   - PR curves, F1-score, cost-based thresholds instead of accuracy

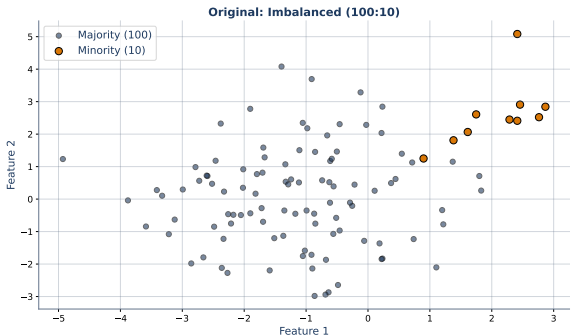
Best practice: Combine all three for robust rare-event classification.

We will cover all three approaches – each alone is often not enough

Sampling: Original Imbalanced Data

Starting Point: 100:10 Class Ratio

- Majority class dominates training
- Model learns to predict majority class by default

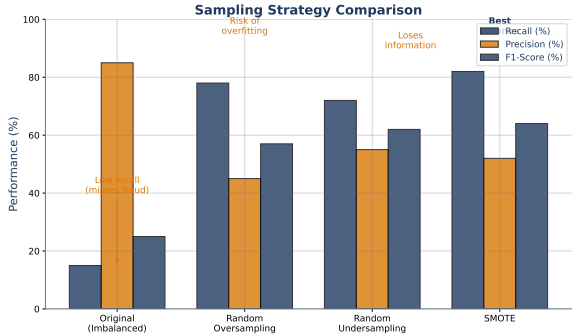


Goal: Rebalance training data while preserving test data distribution

Oversampling vs Undersampling

Two Simple Approaches

- **Oversampling:** Duplicate minority samples (risk: overfitting)
- **Undersampling:** Remove majority samples (risk: losing information)

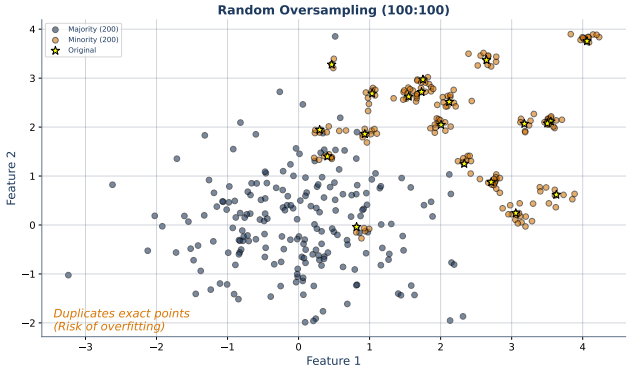


Trade-off: Oversampling risks overfitting, undersampling loses information

Random Oversampling

Simple Duplication

- Copy existing minority samples until classes are balanced
- Pro: simple. Con: model memorizes exact copies (overfitting risk)

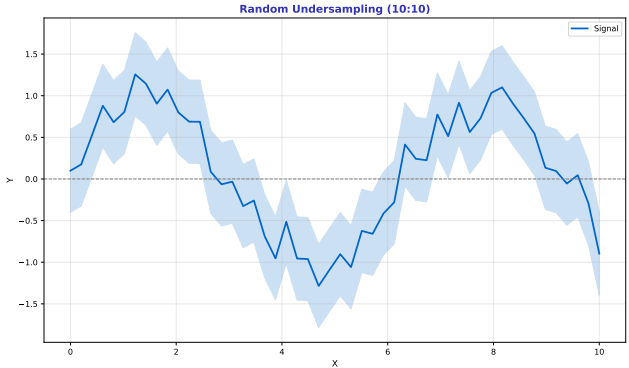


Duplicates minority samples – simple but may cause overfitting

Random Undersampling

Removing Majority Samples

- Randomly remove majority samples until classes are balanced
- Pro: fast, no synthetic data. Con: throws away potentially useful information

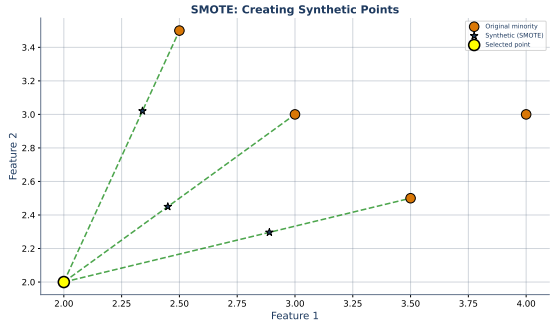


Removes majority samples – fast but loses potentially useful information

SMOTE: The Smart Solution

Instead of Copying, CREATE Synthetic Samples

- Interpolate between nearest neighbors: $x_{new} = x + \alpha(nn - x)$
- Key rule: ONLY on training data, NEVER test data

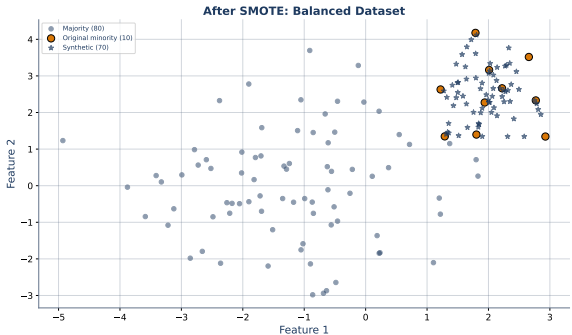


SMOTE generates points along lines connecting minority class neighbors

After SMOTE: Balanced Dataset

SMOTE Result

- Synthetic points fill the minority region naturally
- More variety than duplication – reduces overfitting



Synthetic samples fill the minority class region without exact duplication

What is K-Nearest Neighbors?

The "Nearest" Concept

- Distance is typically Euclidean: $\sqrt{\sum(x_i - y_i)^2}$
- "Nearest" = smallest distance in feature space
- k = how many neighbors to consider (default $k = 5$)

Why k Matters

- $k = 1$: Only consider the single closest neighbor (sensitive to noise)
- $k = 5$: Consider 5 closest neighbors (more robust)
- k too large: May include points from different subclusters

Important Consideration

- Features must be on similar scales (normalize first!)
- Otherwise, features with larger ranges dominate distance

Preprocessing: StandardScaler or MinMaxScaler before distance-based methods

SMOTE Algorithm Steps

For each minority sample x:

1. Find k nearest neighbors (typically $k = 5$)
2. Randomly select one neighbor nn
3. Create synthetic point: $x_{new} = x + \alpha \cdot (nn - x)$ where $\alpha \in [0, 1]$
4. Repeat until desired class balance is achieved

Key Insight: Creates NEW points along the line between existing minority points – not exact copies (avoids overfitting), stays within minority region.

Variants: SMOTE-ENN, SMOTE-Tomek, ADASYN, Borderline-SMOTE

---

1. Find k-nearest neighbors 2. Select random neighbor 3. Interpolate new point

SMOTE: Visual Intuition

The Interpolation Concept

- Interpolation = creating points BETWEEN existing points
- Like drawing a line between two dots and picking a point on that line

The Formula Unpacked: $x_{new} = x + \alpha \cdot (nn - x)$

- x = existing minority point (starting position)
- nn = nearest neighbor (another minority point)
- $(nn - x)$ = direction from x to nn (the "path")
- $\alpha = 0 \rightarrow x_{new} = x$ (at starting point)
- $\alpha = 1 \rightarrow x_{new} = nn$ (at neighbor)
- $\alpha = 0.5 \rightarrow x_{new}$ is exactly halfway

Key Insight: New points stay WITHIN the minority region

Unlike duplication, SMOTE creates variety – reduces overfitting risk

SMOTE Implementation

Using imbalanced-learn Library

- Install: `pip install imbalanced-learn`
- Import: `from imblearn.over_sampling import SMOTE`
- Key parameters: `sampling_strategy`, `k_neighbors`
- Use with Pipeline to avoid data leakage

Important: Only apply SMOTE to training data, never to test data!

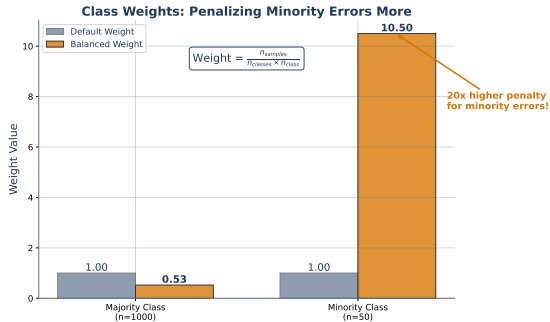
Variants: SMOTE, SMOTETomek (cleaner boundaries), ADASYN

Use `from imblearn.over_sampling import SMOTE`

Class Weights: No Resampling Needed

Multiply Minority Errors by Weight

- Use `class_weight='balanced'` in sklearn
- Model "feels" 10x more pain for minority errors

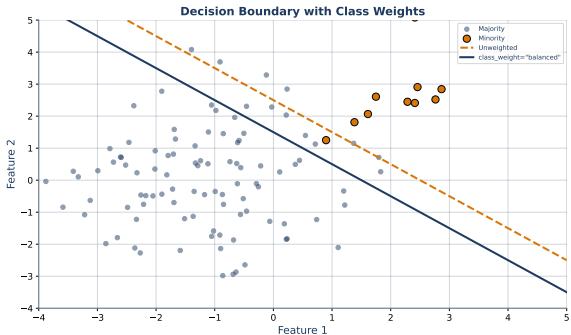


Weight = $n_{\text{samples}}/(n_{\text{classes}} \times n_{\text{samples},c})$ – each CLASS contributes equally

Weights Shift the Boundary

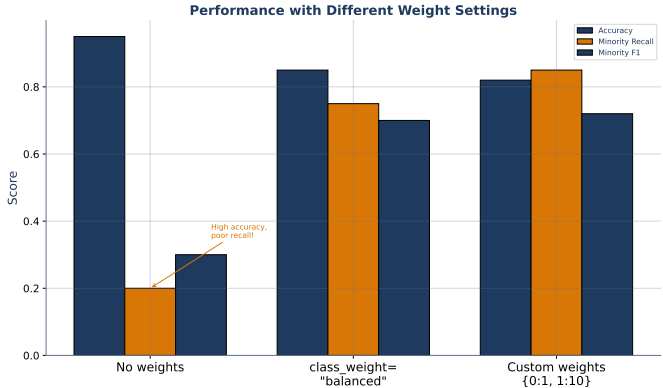
How Weights Change Classification

- Higher minority weight pushes boundary away from minority class
- More minority samples get correctly classified (higher recall)



Higher minority weight shifts decision boundary away from minority class

Performance with Different Weights



Optimal weight depends on business costs – not always “balanced”

Class Weights: The Math

The Loss Function Connection

- Models minimize a loss function during training
- Standard: Each sample contributes equally to total loss
- With weights: Multiply each sample's loss by its class weight

Weight Calculation for "Balanced"

$$w_c = \frac{n_{\text{samples}}}{n_{\text{classes}} \times n_{\text{samples},c}}$$

- For 1000 samples, 2 classes, 950 neg / 50 pos:
- $w_{neg} = 1000/(2 \times 950) = 0.53$
- $w_{pos} = 1000/(2 \times 50) = 10$

Balanced weights make each CLASS contribute equally to loss, not each sample

Class Weights in sklearn

Three Methods to Set Weights

1. **Automatic:** `class_weight='balanced'` – inversely proportional to frequency
2. **Custom dict:** `class_weight={0: 1, 1: 10}` – explicit penalty ratios
3. **Compute from data:** Use `compute_class_weight()` function

Supported classifiers: LogisticRegression, RandomForest, SVC, DecisionTree

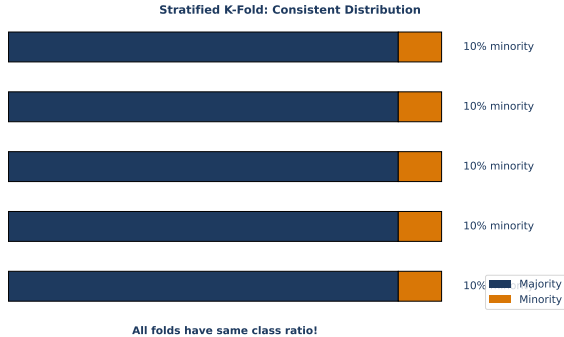
Alternative: Use `sample_weight` parameter for instance-level weights

Use `class_weight='balanced'` or provide custom dict

Stratified Cross-Validation

Problem: Random splits may put all rare events in one fold

Solution: StratifiedKFold preserves class ratios in each fold

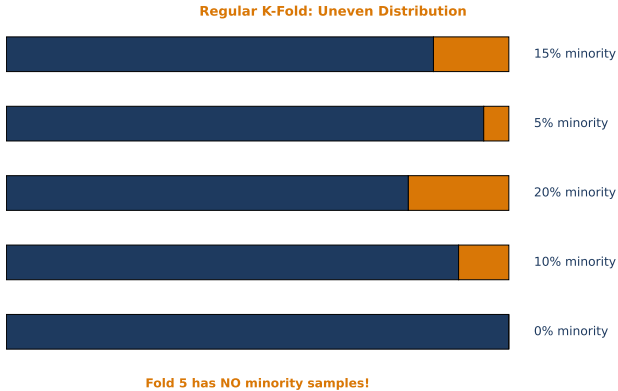


Each fold maintains the same class ratio as the full dataset

Regular K-Fold: Uneven Splits

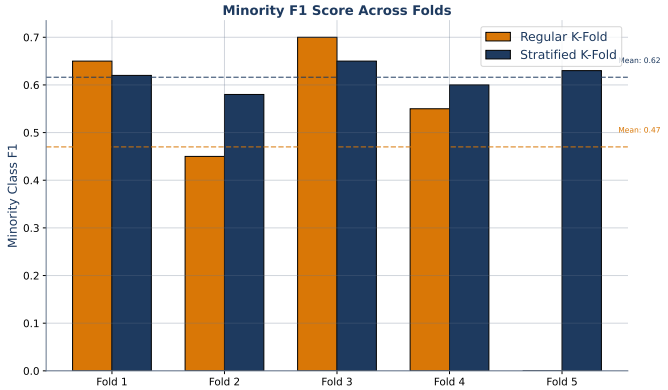
Problem with Standard CV

- Random splits may put all rare events in one fold
- Some folds have no positive examples at all



With 1% positive rate and 5 folds, some folds may have 0 positives

Minority Class F1 Score Across Folds



Stratified CV gives more stable performance estimates

Stratified CV in sklearn

Key Concepts

- **Train/test split:** Use `stratify=y` to preserve class ratios
- **Cross-validation:** Use `StratifiedKFold` instead of regular `KFold`
- Each fold maintains the same class distribution as full dataset

Best Practice: sklearn's default `cv=5` in `cross_val_score` is already stratified for classifiers, but explicit is better for clarity.

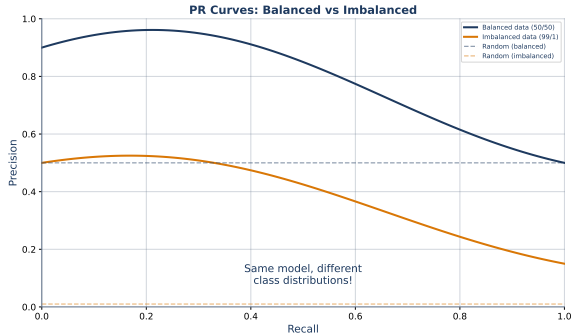
Key function: `from sklearn.model_selection import StratifiedKFold`

Use `StratifiedKFold` instead of `KFold` for classification

PR Curves Beat ROC for Imbalanced Data

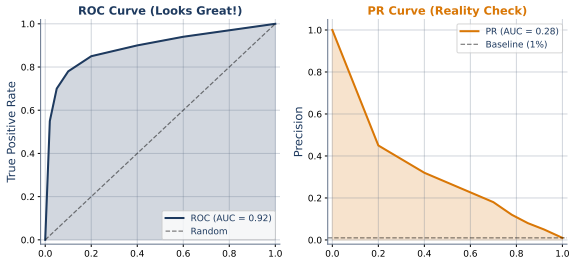
Why PR Curves Matter More

- ROC can look good even with poor minority detection
- PR curve focuses on positive class performance



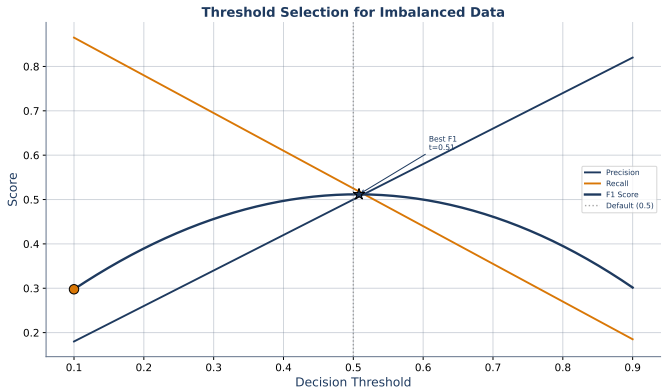
High AUC-ROC but low AUC-PR = model fails on what matters

PR vs ROC Comparison



ROC is optimistic for imbalanced data – always check PR curve

Threshold Selection for Imbalanced Data



Default 0.5 threshold is rarely optimal for imbalanced problems

PR Curve Evaluation

Key Metrics for Imbalanced Data

- **Average Precision (AP):** Area under PR curve – better than ROC-AUC for imbalanced data
- **Baseline:** Random classifier gets AP = positive class rate
- **Lift:** AP / baseline – how much better than random?

Use AP for Model Selection:

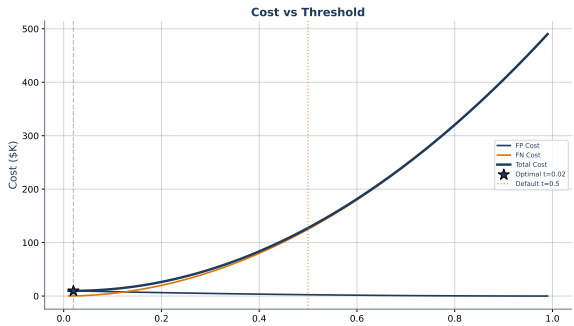
- More informative than accuracy or ROC-AUC
- Set `scoring='average_precision'` in cross-validation

Use `precision_recall_curve` and `average_precision_score`

Cost-Sensitive Threshold Selection

Threshold Based on Business Costs

- FN costs \$1000 (missed fraud) vs FP costs \$10 (investigation)
- Optimal: $t^* = \frac{C_{FP}}{C_{FP}+C_{FN}} \approx 0.01$ – flag almost everything

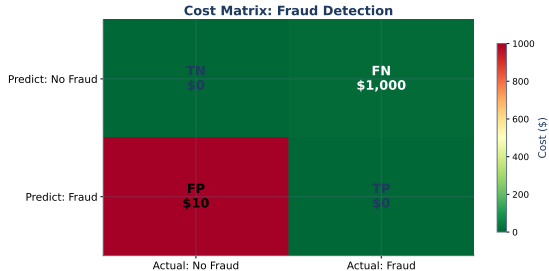


Optimal threshold minimizes expected total cost, not error rate

Expected Profit Calculation

Optimize for Profit, Not Just Metrics

- Expected cost = $FN \times C_{FN} + FP \times C_{FP}$
- Small recall improvement → millions in recovered fraud



Expected Cost = FN x $1,000 + FP x 10

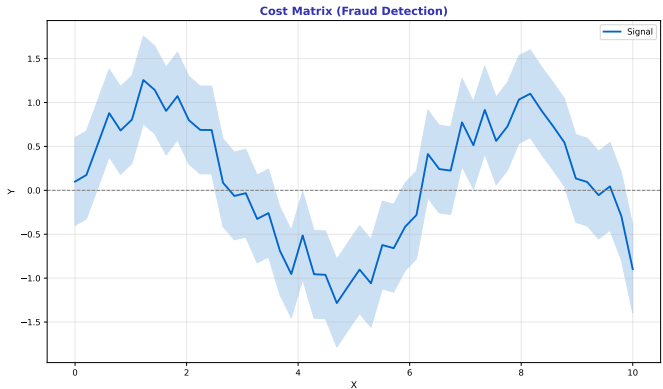
Missing fraud (FN) is 100x more costly than false alarm (FP)

Expected Cost = FN $\times$ Cost_{FN} + FP $\times$ Cost_{FP}

Cost Matrix: Fraud Detection

Encoding Business Costs

- FN (missed fraud): $1000 loss
- FP (false alarm): $10 investigation cost



Asymmetric costs: missing fraud is 100x worse than false alarm

Cost-Sensitive Threshold Selection (Detail)

Finding the Optimal Decision Threshold

1. Calculate total cost for each threshold: Cost $= FP \times C_{FP} + FN \times C_{FN}$
2. Search over thresholds (0.01 to 0.99)
3. Select threshold with minimum expected cost

Theoretical Optimal: $t^* = \frac{C_{FP}}{C_{FP}+C_{FN}}$

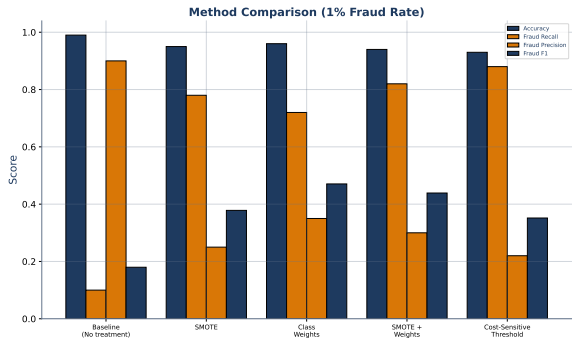
Example: If FP costs \$100 and FN costs \$10,000, optimal threshold ≈ 0.01 (flag more as positive)

Grid search over thresholds to minimize expected cost

Method Comparison on Fraud Data

Putting It All Together

- Real-world fraud rates: 0.1–1% positive
- Compare: baseline, SMOTE, class weights, combined

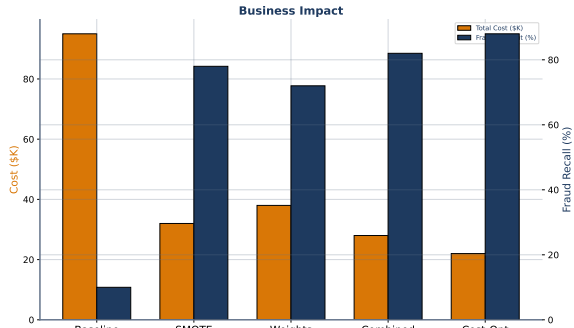


No single method is always best – test on your specific data

Business Impact Analysis

Why This Matters in Practice

- Small recall improvement can mean millions in recovered fraud
- Business stakeholders care about dollars, not F1-scores



Small recall improvement can mean millions in recovered fraud

Complete Fraud Detection Pipeline

Recommended Pipeline Components

1. **Resampling:** SMOTE in imblearn Pipeline (training only)
2. **Classifier:** RandomForest with `class_weight='balanced'`
3. **Validation:** StratifiedKFold cross-validation
4. **Metrics:** F1-score and Average Precision (not accuracy!)
5. **Threshold:** Tune based on business costs

Key Insight: Combine SMOTE + class weights for "double protection" against imbalance.

Full pipeline: SMOTE + class weights + stratified CV + PR evaluation

Five Rules for Imbalanced Classification

1. **Don't trust accuracy** – use F1, Precision, Recall, PR-AUC
2. **Combine techniques** – SMOTE + class weights + stratified CV
3. **Tune the threshold** – default 0.5 is rarely optimal
4. **Consider business costs** – optimize for profit, not just metrics
5. **Validate properly** – never resample test data

When to Use Each:

- Slight imbalance (80/20): Class weights alone
- Moderate (95/5): SMOTE + weights
- Extreme (99.9/0.1): Consider anomaly detection

Industry insight: Ensemble methods (XGBoost, LightGBM) work well for fraud

Hands-On Exercise (25 min)

Task: Build a Fraud Detection Model

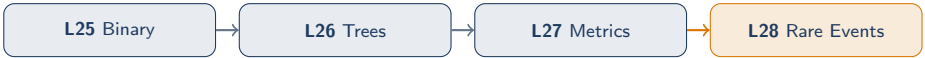
1. Create synthetic imbalanced data (1% fraud rate)
2. Train baseline model – observe the accuracy trap
3. Apply SMOTE and retrain – compare recall
4. Use `class_weight='balanced'` – compare to SMOTE
5. Plot Precision-Recall curve for best model

Deliverable: Comparison table of recall/precision across methods.

Extension: Implement cost-sensitive threshold selection

Module Wrap-Up: The Classification Journey

Module 5 Arc: From Simple Decisions to Rare Events



- L25: Binary decisions (logistic regression)
- L26: Non-linear boundaries (decision trees)
- L27: Measuring quality (confusion matrix, ROC, PR)
- L28: Handling rare events (SMOTE, weights, costs)

Next Module: What if there are NO labels? Welcome to Unsupervised Learning (L29–L32).

From Fisher's iris flowers to fraud detection – the complete classification toolkit