IMEN574/IDSC574 Programming for Data Science

10. **Classification**

Spring 2026

Instructor: **Sangwoo Mo**

Today’s Agenda

01 **What Is Classification?**

Definition, examples, and where it fits in ML

02 **Logistic Regression**

Sigmoid, cross-entropy, gradient descent, multi-class

03 **Regularization**

L2 / L1 penalties and tuning C with CV

04 **SVM**

Maximum margin, support vectors, and the kernel trick

05 **Tree-Based & Boosting**

DT/RF/Boosting recap, entropy & uncertainty, XGBoost & LightGBM

06 **Evaluation & Practice**

Confusion matrix, ROC-AUC, class imbalance, full example

What Is Classification?

Given input features X, predict a discrete categorical output y.

$$y = f(X) \in \{C1, C2, ..., Ck\}$$

X = input features (pixel values, text embeddings, lab results, ...)

y = categorical target (spam/ham, cat/dog, disease type, ...)

f = the decision function we want to learn from labeled data

k = number of classes (2 = binary, 3+ = multi-class)

Regression → continuous values ($350K, 22.5°C) | Classification → discrete categories (spam / not spam)

Classification Is Everywhere

Email Filtering

Spam vs. ham detection
from email content

Medical Diagnosis

Disease classification from
patient symptoms & lab data

Fraud Detection

Flagging fraudulent
transactions in real-time

Image Recognition

Object/face detection,
digit recognition (MNIST)

Sentiment Analysis

Positive/negative/neutral
from text reviews

Customer Churn

Predict which customers
will leave next month

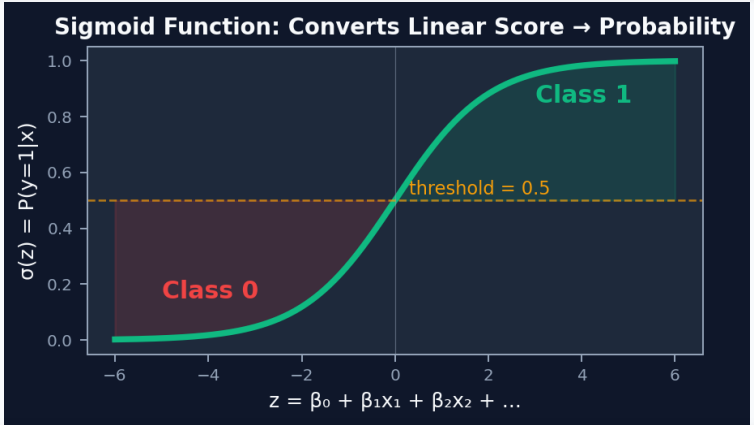
Logistic Regression

The foundation of classification modeling

Logistic Regression — Core Idea

$$P(y=1|X) = \sigma(\beta^T X)$$

- $\sigma(z) = 1/(1+\exp(-z))$ — the sigmoid function
- Maps any real $z \rightarrow (0,1)$: valid probability output
- Predict class 1 if $\sigma(\beta^T X) \geq 0.5$ (threshold adjustable)



Key Insight: the Logit (Log-Odds)

$$\log\left(\frac{p}{1-p}\right) = \beta^T X$$

- $\beta^T X$ predicts log-odds, not probability directly — sigmoid converts it to probability
- Each β_j: a 1-unit increase in x_j multiplies the odds by $\exp(\beta_j)$ → basis of odds-ratio interpretation
- This is why it's called logistic regression: logit + regression

Why not linear regression?

Linear outputs can be < 0 or > 1 — not valid probabilities. Sigmoid squashes output to always be in (0,1).

From Probability to Prediction

How sigmoid output becomes a classification decision:

Decision Rule

- If $P(y=1|X) \geq$ threshold → predict class 1
- Else → predict class 0
- Default threshold: 0.5

Threshold Flexibility

- Lower threshold (e.g., 0.3) → predict class 1 more often
- Higher threshold (e.g., 0.7) → predict class 1 less often
- Tune to balance precision/recall based on business needs

This decision boundary determines how LR separates the classes.

Decision Boundary

Logistic regression finds a linear boundary that separates the classes.

Binary Classification (2D)

- The decision boundary is where $P(y=1) = 0.5$
- This happens when $\beta TX = 0$
- In 2D: a straight line
- In nD: a hyperplane
- Points on one side → class 0
- Points on other side → class 1

Key Implication

Logistic regression can only learn linear boundaries.

If classes aren't linearly separable, you need:

- SVM (kernel trick)
- Tree-based models
- Feature engineering

Same limitation as linear regression for regression tasks!

The sigmoid outputs probabilities, not just hard predictions — this is a major advantage for ranking and thresholding.

Cross-Entropy Loss (Log Loss)

We can’t use MSE for classification — the loss surface is non-convex. Instead, use cross-entropy:

Binary Cross-Entropy

$$L = -1/n \sum [\, y_i \log(\hat{y}_i) + (1-y_i) \log(1-\hat{y}_i) \,]$$

- When y=1: loss = −log(ŷ) → penalizes low predicted probability for true positives
- When y=0: loss = −log(1−ŷ) → penalizes high predicted probability for true negatives

Probabilistic View (MLE)

Assume yi ~ Bernoulli(pi) where pi = σ(βTxi)

Likelihood: ∏ pi^yi (1-pi)^(1-yi)

Negative log-likelihood = Binary Cross-Entropy!

Just like regression: MLE naturally gives us the right loss function.

Tip: Bernoulli = coin flip with probability p. MLE finds the p that makes our observed labels most probable.

Training Logistic Regression

No closed-form solution — must use iterative optimization (gradient descent or variants).

Gradient of Cross-Entropy

$$\partial L/\partial \beta_j = 1/n \sum (\hat{y}_i - y_i) \cdot x_{ij}$$

- Looks just like the linear regression gradient! (with ŷi = σ(βTxi) instead of βTxi)
- Cross-entropy is convex → gradient descent is guaranteed to find the global minimum
- scikit-learn uses L-BFGS, Newton-CG, or SAG/SAGA solvers (faster than plain GD)

vs. Linear Regression

Linear Reg: closed-form exists (Normal Eq.)
Logistic Reg: no closed-form, must iterate.

But both can use gradient descent!

sklearn Solvers

```
'lbfgs'     - default, good for small data
'saga'      - best for large data, supports L1
'newton-cg' - good for multi-class
'liblinear' - fast, but no multi-class OvR only
```

Logistic Regression — Code

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)
model = make_pipeline(
    StandardScaler(),
    LogisticRegression(random_state=42)  # C=1.0 by default
)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]  # probability of class 1
print(f"Accuracy: {accuracy_score(y_test, y_pred):.3f}")
print(classification_report(y_test, y_pred))
```

Note: stratify=y ensures class proportions are preserved in train/test split — important for imbalanced data.

Multi-Class Classification

Binary LR uses sigmoid for 2 classes. For K > 2, softmax generalizes this naturally.

Softmax (Multinomial)

- Each class gets its own score $z_k = \beta_k^T x$ — K logits total
- Softmax normalizes: $P(\text{class } k) = \exp(z_k) / \sum \exp(z_j)$ → all sum to 1
- Learns all K weight vectors jointly — more principled than K separate binary models
- *Now the sklearn default (solver='lbfgs' uses multinomial automatically)*

```
# Softmax (multinomial) — now the sklearn default
lr = LogisticRegression(solver='lbfgs')  # auto = multinomial
lr.fit(X_train, y_train)
probs = lr.predict_proba(X_test)  # shape (n_samples, K)
print(probs[0])    # e.g. [0.05, 0.82, 0.13] — sums to 1
print(lr.classes_)  # class labels
```

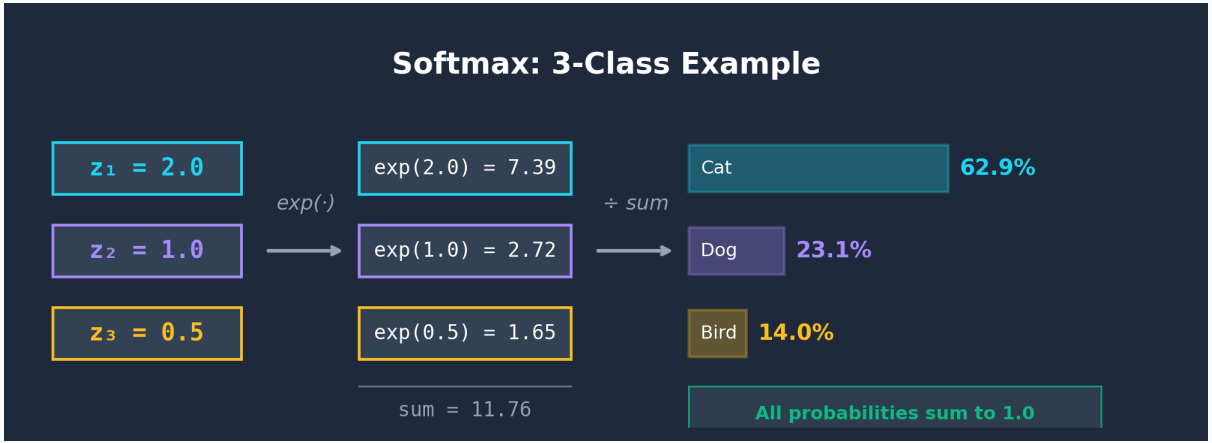
Note: One-vs-Rest (OvR) trains K binary classifiers separately — simpler but less elegant. Still available via multi_class='ovr'.

Softmax — Multi-Class Probabilities

Convert K raw scores (logits) into a valid probability distribution that sums to 1.

$$P(y = k \mid x) = \exp(z_k) / \sum_j \exp(z_j) \quad \text{where } z_k = \beta_k \hat{} T\, x$$

exp(z) makes all scores positive — then dividing by the sum normalizes them into probabilities.



Categorical Cross-Entropy Loss

$$L = -\sum_k y_k \cdot \log P(y=k|x)$$

Direct extension of binary cross-entropy to K classes. Each class's true label penalizes wrong predictions.

Interpreting Coefficients — Odds Ratio

In logistic regression, coefficients relate to odds ratios:

$$\text{odds} = P(y=1) / P(y=0) = e^{(\beta 0 + \beta 1 x 1 + \ldots)}$$

- $e^{\beta j}$ = odds multiplier when xj increases by 1 unit
- $e^{\beta} > 1 \rightarrow$ feature increases odds of class 1
- $e^{\beta} < 1 \rightarrow$ feature decreases odds of class 1
- $e^{\beta} = 1 \rightarrow$ feature has no effect on odds

```
import numpy as np
import pandas as pd
# After fitting logistic regression
coefs = model[1].coef_[0]  # pipeline: model[1] is LogisticRegression
odds_ratios = np.exp(coefs)
odds_df = pd.DataFrame({
    'feature': feature_names,
    'coefficient': coefs,
    'odds_ratio': odds_ratios
}).sort_values('odds_ratio', ascending=False)
print(odds_df.to_string(index=False))
```

Logistic Regression — Limitations

Linear Boundary Only

Can only learn straight decision boundaries. Non-linear patterns need kernel SVM or tree-based models.

Sensitive to Outliers

Extreme values in features can disproportionately affect the decision boundary.

Feature Scaling Required

Coefficients are scale-dependent. Always use StandardScaler for fair regularization.

Assumes Independence

Assumes features contribute independently. Interactions must be manually engineered.

Despite these limitations, logistic regression is the go-to baseline for any classification task!

Regularization

Controlling complexity in classification models

Regularization in Logistic Regression

Same idea as regression: add a penalty to prevent large coefficients and overfitting.

L2 Penalty (Ridge, default)

$$\text{Loss} = \text{CE} + (1/2C) \sum \beta j^2$$

Shrinks all coefficients.
Keeps all features.
Default in sklearn (penalty='l2').
Great for most cases.

L1 Penalty (Lasso)

$$\text{Loss} = \text{CE} + (1/C) \sum |\beta j|$$

Can zero out coefficients.
Automatic feature selection.
Use penalty='l1', solver='saga'.
Best with many irrelevant features.

C Parameter (inverse of α)

sklearn uses C = 1/α. Larger C = less regularization (more complex model). Smaller C = more regularization (simpler model).

- C = 0.01 → strong regularization, underfitting risk
- C = 1.0 → default, balanced
- C = 100 → weak regularization, overfitting risk

Regularization — Code

```python
from sklearn.linear_model import LogisticRegression
# L2 regularization (default)
lr_l2 = make_pipeline(StandardScaler(),
    LogisticRegression(penalty='l2', C=1.0))
# L1 regularization (feature selection)
lr_l1 = make_pipeline(StandardScaler(),
    LogisticRegression(penalty='l1', C=1.0, solver='saga'))
# Compare
for name, m in [('L2', lr_l2), ('L1', lr_l1)]:
    m.fit(X_train, y_train)
    acc = m.score(X_test, y_test)
    n_nonzero = (m[1].coef_ != 0).sum()
    total = m[1].coef_.size
    print(f"{name}: Accuracy={acc:.3f}, "
          f"Non-zero coefs: {n_nonzero}/{total}")
```

Tuning C with Cross-Validation:

```python
from sklearn.linear_model import LogisticRegressionCV
lr_cv = make_pipeline(StandardScaler(),
    LogisticRegressionCV(Cs=[0.01,0.1,1,10,100],
        cv=5, scoring='accuracy'))
lr_cv.fit(X_train, y_train)
print(f"Best C: {lr_cv[1].C_[0]:.2f}")
```

Support Vector Machine

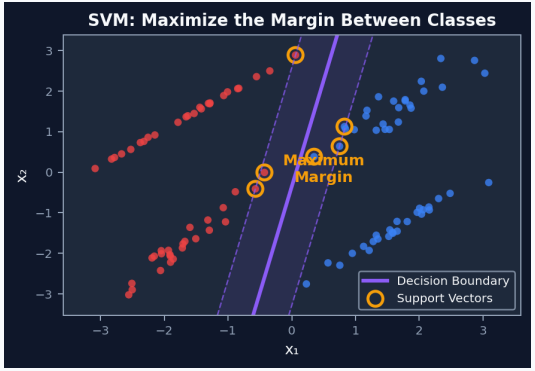
Maximum margin classification

Support Vector Machine — Core Idea

SVM finds the decision boundary that maximizes the margin between classes.

Key Concepts:

- Margin: the distance between the boundary and the nearest points of each class
- Support Vectors: the data points closest to the boundary — they "support" and define it
- Maximizing margin → better generalization (larger gap = more robust to noise)
- Only support vectors matter — other points don't affect the boundary at all
- C parameter: trade-off between margin width and misclassification



Small C

Wide margin, allows some misclassification (soft margin). More regularized.

Large C

Narrow margin, tries to classify all points correctly. Can overfit.

Tip: Why maximize margin? Wider gap between classes = more robust to noise in test data = better generalization.

The Kernel Trick

Linear SVM only draws straight boundaries. The kernel trick lets SVM work in higher dimensions without going there.

Intuition

- Replace dot product x·z with kernel K(x,z)
- K computes similarity in high-dim space cheaply
- SVM boundary: sign(β0 + ∑αi·K(xi, x))

Common Kernels

- Linear: K(x,z) = x^Tz (use when data is separable)
- RBF: K(x,z) = exp(−γ‖x−z‖²) — most popular default
- Poly: K(x,z) = (γx^Tz + r)^d (captures interactions)

In practice: start with RBF kernel. It works well for most non-linear problems and has only two hyperparameters (C, gamma).

```
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
# Always scale first! SVM is distance-based
svm = make_pipeline(StandardScaler(),
        SVC(kernel='rbf', C=1.0, gamma='scale', probability=True))
svm.fit(X_train, y_train)
print(f"Accuracy: {svm.score(X_test, y_test):.3f}")
# Tune: C=[0.1,1,10,100], gamma=['scale',0.01,0.1]
```

Tree-Based Models for Classification

What changes from regression: split criteria, loss, and evaluation

Measuring Uncertainty

Trees split data to reduce uncertainty. A good split makes child nodes purer. Two common measures:

Entropy

$$H(S) = -\sum p_i \cdot \log_2(p_i)$$

- From information theory: measures how “surprising” a node is
- H = 0: perfectly pure node (one class only)
- H = 1: maximum confusion (50/50 binary)
- `criterion='entropy'`
- *Tree picks the split that reduces H the most*

Gini Impurity

$$G(S) = 1 - \sum p_i^2$$

- Probability of misclassifying a random sample
- G = 0: pure node
- G = 0.5: maximum uncertainty (binary)
- `criterion='gini'` (sklearn default)
- *Nearly identical to entropy in practice*

Both measures give almost the same trees. The tree picks the split that drops uncertainty the most (Information Gain). Use whichever your framework defaults to.

Tree-Based Models — Recap & Differences

You already learned DT, RF, and Boosting for regression. The core algorithms are identical — here's what changes:

	Regression	Classification
Leaf prediction	Mean of samples	Majority class (or class probs)
Split criterion	MSE reduction	Gini impurity / Entropy
Boosting loss	MSE (squared error)	Log loss (cross-entropy)
Pseudo-residuals	yi − F(xi)	yi − σ(F(xi))
Final output	Predicted value directly	Apply sigmoid → probability
sklearn class	*Regressor	*Classifier
Eval metric	R^2, RMSE	Accuracy, F1, ROC-AUC

Everything else — bagging, boosting, hyperparameters, early stopping, feature importance — works exactly the same way.

XGBoost & LightGBM for Classification



Just swap Regressor → Classifier. Everything else stays the same (hyperparams, tuning, early stopping).

```python
import xgboost as xgb
import lightgbm as lgb
# XGBoost
xgb_clf = xgb.XGBClassifier(
    n_estimators=500, learning_rate=0.05, max_depth=6,
    subsample=0.8, colsample_bytree=0.8,
    eval_metric='logloss',
    random_state=42)
xgb_clf.fit(X_train, y_train,
            eval_set=[(X_test, y_test)], verbose=False,
            callbacks=[xgb.callback.EarlyStopping(rounds=20)])
# LightGBM
lgb_clf = lgb.LGBMClassifier(
    n_estimators=500, learning_rate=0.05,
    num_leaves=31, min_child_samples=20, random_state=42)
lgb_clf.fit(X_train, y_train, eval_set=[(X_test, y_test)],
            callbacks=[lgb.early_stopping(20, verbose=False)])
for name, m in [('XGBoost', xgb_clf), ('LightGBM', lgb_clf)]:
    print(f"{name}: Acc={m.score(X_test, y_test):.3f}")
```

Hyperparameter tuning: same RandomizedSearchCV approach as regression. Use scoring='roc_auc' or 'f1' instead of 'r2'.

Evaluation & Metrics

Beyond accuracy — measuring what matters

Confusion Matrix

The 4 quadrants show: what we predicted vs. what actually happened.

		Predicted: Negative	Predicted: Positive
Actual	Actual Neg	TN (True Negative)	FP (False Positive)
Actual	Actual Pos	FN (False Negative)	TP (True Positive)

TP: model predicted positive and it was correct. FP: model predicted positive but it was wrong.

TN: model predicted negative and it was correct. FN: model predicted negative but it was wrong.

Computing the Confusion Matrix

How to compute and interpret the confusion matrix in scikit-learn:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Get predictions
y_pred = model.predict(X_test)
# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)
# [[TN  FP]
#  [FN  TP]]
# Visualize
ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
plt.show()
```

Each row = actual label, each column = predicted label. Diagonal = correct predictions.

Precision, Recall & F1 Score

Precision	$TP / (TP + FP)$	Of all predicted positive, how many are actually positive? High when FP is costly (spam filter).
Recall	$TP / (TP + FN)$	Of all actual positive, how many did we catch? High when FN is costly (cancer detection).
F1 Score	$2 \times P \times R / (P + R)$	Harmonic mean of precision & recall. Balanced metric when classes are imbalanced.
Accuracy	$(TP+TN) / Total$	Overall correctness. Misleading when classes are imbalanced! (99% acc with 99% majority class)

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

ROC Curve & AUC

ROC Curve

- Plots True Positive Rate (Recall) vs False Positive Rate at all thresholds
- Threshold: P ≥ t → predict positive
- Top-left corner = perfect classifier
- Diagonal = random guessing

AUC (Area Under Curve)

- AUC = 1.0: perfect separation
- AUC = 0.5: random (no discrimination)
- AUC > 0.9: excellent
- Threshold-independent: evaluates ranking quality, not a specific cutoff

```
from sklearn.metrics import roc_auc_score, roc_curve
y_prob = model.predict_proba(X_test)[:, 1]
auc = roc_auc_score(y_test, y_prob)
print(f"ROC AUC: {auc:.3f}")
fpr, tpr, _ = roc_curve(y_test, y_prob)
import matplotlib.pyplot as plt
plt.plot(fpr, tpr, label=f'AUC={auc:.3f}')
plt.plot([0,1],[0,1], 'k--', label='Random')
plt.xlabel('FPR'); plt.ylabel('TPR')
plt.legend(); plt.show()
```

Tip: AUC = probability that model ranks a random positive higher than a random negative.

Probability Calibration

predict_proba() doesn't always return true probabilities. Calibration fixes this.

Well-Calibrated?

If model says P=0.8 for 100 samples,
about 80 should actually be positive.

✓ Logistic Reg. — naturally well-calibrated
✗ SVM, RF, Boosting — often not

Matters for: medical risk scores,
ranking, threshold tuning

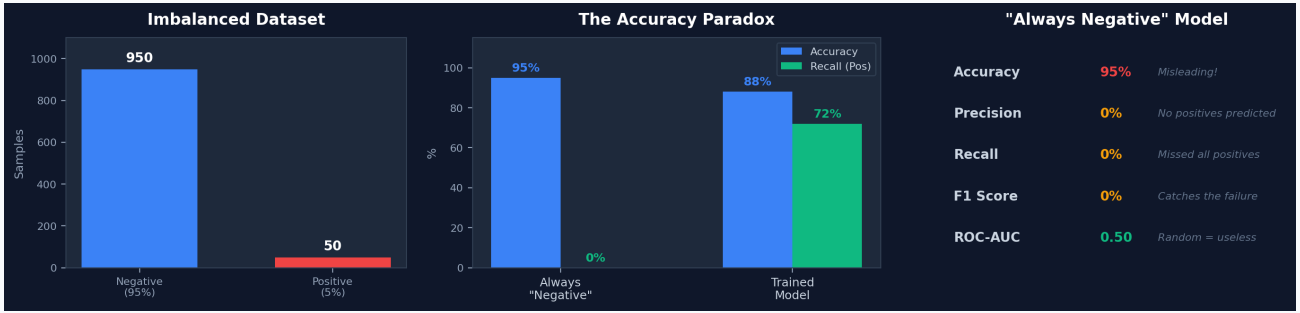
Calibration Methods

- Platt Scaling: fit sigmoid on outputs (SVM default with probability=True)
- Isotonic Regression: non-parametric, flexible, needs more data
- CalibratedClassifierCV: sklearn wrapper that handles it all
- **Always calibrate on held-out data (CV), never on training set!**

```
from sklearn.calibration import CalibratedClassifierCV
svm = SVC(kernel='rbf', probability=False)
cal_svm = CalibratedClassifierCV(
    svm, cv=5, method='sigmoid')
cal_svm.fit(X_train, y_train)
y_prob = cal_svm.predict_proba(X_test)[:, 1]
from sklearn.calibration import calibration_curve
frac_pos, mean_pred = calibration_curve(
    y_test, y_prob, n_bins=10)
```

Class Imbalance

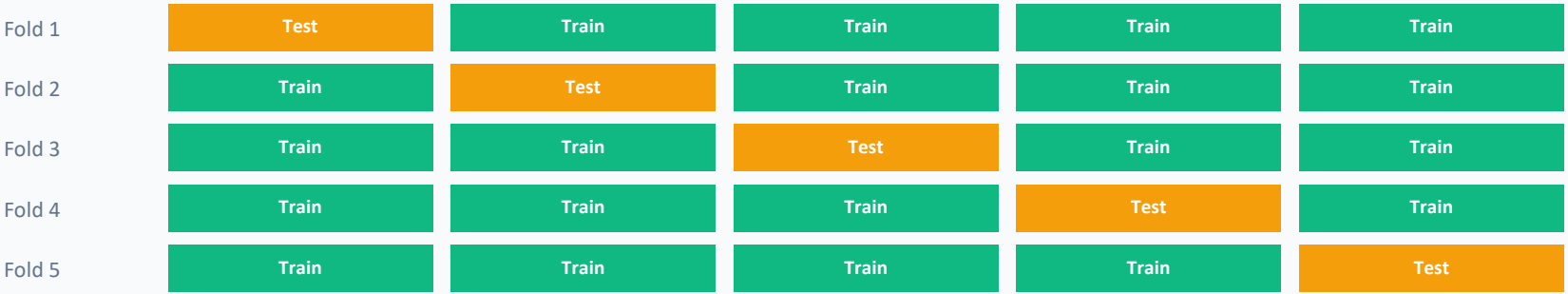
When one class dominates (e.g., 95/5), accuracy becomes meaningless. A model predicting all-negative gets 95% accuracy but catches zero positives.



`class_weight='balanced'`	Automatically upweights minority class in the loss. Supported by LR, SVC, RF, XGBoost. Try this first.
`Threshold tuning`	Lower predict_proba threshold from 0.5 to 0.3–0.4. Increases recall without resampling.
`SMOTE`	Synthesize minority samples via interpolation (imblearn). Apply only on training set inside a Pipeline.
`AUC-ROC / F1`	Use AUC or class-specific F1 as eval metric, not accuracy. ROC curve is imbalance-insensitive.

Cross-Validation for Classification

Same K-Fold CV as regression, but use StratifiedKFold (preserves class ratios in each fold).



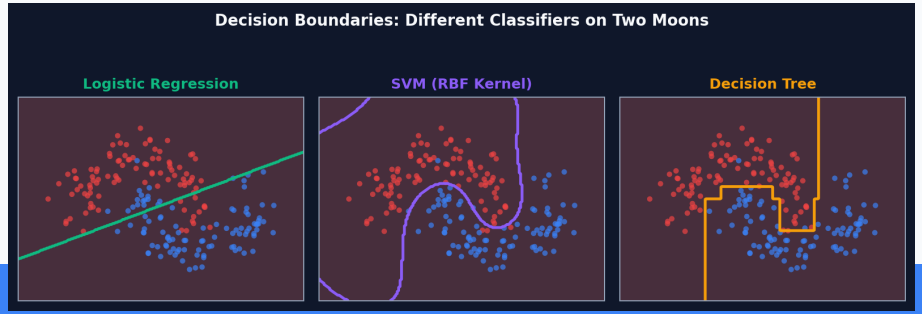

```python
from sklearn.model_selection import cross_val_score, StratifiedKFold
# Stratified: preserves class proportions in each fold
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
# Try different scoring metrics
for metric in ['accuracy', 'f1', 'roc_auc']:
    scores = cross_val_score(model, X, y, cv=cv, scoring=metric)
    print(f"{metric:10s}  {scores.mean():.3f} (+/- {scores.std():.3f})")
```

Putting It All Together

Comparison, full example, and best practices

Algorithm Comparison

Algorithm	Non-linear	Scaling?	Probabilities	Best for
Logistic Reg.	✗	Yes	Native	Baseline, interpretability
SVM (Kernel)	✓	Yes	Platt (approx)	Small data, non-linear
Decision Tree	✓	No	Class fractions	Interpretability
Random Forest	✓	No	Vote fractions	Robust default
XGBoost	✓	No	Native	Best accuracy (tabular)
LightGBM	✓	No	Native	Large data + categoricals



Full Example: Breast Cancer (1/3)



```python
import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import classification_report, roc_auc_score
# 1. Load data
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target  # 0=malignant, 1=benign
print(f"Samples: {X.shape[0]}, Features: {X.shape[1]}")
print(f"Class distribution: {np.bincount(y)}")
print(f"  Malignant: {(y==0).sum()}, Benign: {(y==1).sum()}")
# 2. Split (stratified!)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)
```

Full Example: Breast Cancer (2/3)



```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
# 3. Compare models
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
models = {
    'Logistic': make_pipeline(StandardScaler(), LogisticRegression()),
    'SVM(RBF)': make_pipeline(StandardScaler(),
                   SVC(kernel='rbf', probability=True)),
    'RF':       RandomForestClassifier(n_estimators=200,
                   random_state=42),
    'XGBoost':  xgb.XGBClassifier(n_estimators=300,
                   learning_rate=0.1, eval_metric='logloss',
                   random_state=42),
}
for name, model in models.items():
    acc = cross_val_score(model, X_train, y_train,
                          cv=cv, scoring='accuracy').mean()
    auc = cross_val_score(model, X_train, y_train,
                          cv=cv, scoring='roc_auc').mean()
    print(f"{name:10s}  Acc={acc:.3f}  AUC={auc:.3f}")
```

Full Example: Breast Cancer (3/3)



```
# 4. Final evaluation with best model
best_model = make_pipeline(StandardScaler(), LogisticRegression())
best_model.fit(X_train, y_train)
y_pred = best_model.predict(X_test)
y_prob = best_model.predict_proba(X_test)[:, 1]
print(classification_report(y_test, y_pred,
      target_names=['Malignant', 'Benign']))
print(f"ROC AUC: {roc_auc_score(y_test, y_prob):.3f}")
# 5. Feature importance (top 10)
coefs = best_model[1].coef_[0]
feat_imp = pd.DataFrame({
    'feature': data.feature_names,
    'coefficient': coefs,
    'odds_ratio': np.exp(coefs)
}).sort_values('coefficient', key=abs, ascending=False)
print(feat_imp.head(10).to_string(index=False))
```

Note: For this dataset, logistic regression often matches or beats complex models because the relationship is roughly linear and features are well-engineered. Always start with the simple baseline!

Feature Importance for Classification

Same approaches as regression, with one classification-specific addition:

Logistic Reg.

model.coef_
Odds ratios (e^β)
Sign = direction

Tree-Based

feature_importances_
Based on split gain
No direction info

Permutation

Model-agnostic
Shuffle each feature
Measure accuracy drop

```
# Permutation importance (works for any model)
from sklearn.inspection import permutation_importance
result = permutation_importance(model, X_test, y_test,
                                n_repeats=10, random_state=42)
for name, imp in sorted(zip(feature_names, result.importances_mean),
                        key=lambda x: -x[1])[:10]:
    print(f"  {name}: {imp:.4f}")
```

Permutation importance is the most reliable method — works for any model, measures actual prediction impact.

Threshold Tuning in Practice

Default threshold 0.5 is often not optimal. Tune it based on your business objective.

Medical Diagnosis

Missing cancer (FN) is much worse than a false alarm (FP).
→ Lower threshold (e.g., 0.3)
→ Higher recall, lower precision

Spam Filter

Blocking a real email (FP) is worse than letting spam through (FN).
→ Higher threshold (e.g., 0.8)
→ Higher precision, lower recall

```python
from sklearn.metrics import precision_recall_curve, f1_score
y_prob = model.predict_proba(X_test)[:, 1]
prec, rec, thresholds = precision_recall_curve(
    y_test, y_prob)
# Find threshold that maximizes F1
f1s = 2*prec*rec / (prec + rec + 1e-8)
best_t = thresholds[f1s[:-1].argmax()]
print(f"Optimal threshold: {best_t:.3f}")
y_tuned = (y_prob >= best_t).astype(int)
print(f"F1 (0.5):   {f1_score(y_test, y_pred):.3f}")
print(f"F1 (tuned): {f1_score(y_test, y_tuned):.3f}")
```

How to Choose Your Algorithm

Need interpretable model?	→ Logistic Regression (odds ratios)
Many irrelevant features?	→ Logistic Reg. with L1 (feature selection)
Non-linear + small data?	→ Kernel SVM (RBF)
Want best accuracy (tabular)?	→ XGBoost / LightGBM + tune hyperparams
Class imbalance?	→ class_weight='balanced' + F1/AUC metric

Golden rule: Always start with Logistic Regression baseline, then compare more complex models with CV.

Key Takeaways

- Logistic Regression is your baseline — interpretable, fast, outputs probabilities
- Regularization (L2/L1 via C parameter) prevents overfitting, L1 enables feature selection
- SVM with kernel trick captures non-linear boundaries for small/medium datasets
- XGBoost/LightGBM dominate tabular data classification in practice
- Go beyond accuracy: use F1, ROC-AUC, and confusion matrix for real evaluation
- Handle class imbalance: class_weight='balanced', threshold tuning, proper metrics
- Always use StratifiedKFold and Pipelines to prevent data leakage

```
Consistent API:  pipe.fit(X_train, y_train)  >>  pipe.predict_proba(X_test)  >>  roc_auc_score(y_test, y_prob)
```