

Machine Learning in Finance (RSM338)

Week 7: Linear Classification

Table of contents

1	Introduction	2
2	Why Not Just Use Regression?	2
2.1	The Linear Probability Model	2
2.2	The Problem with Linear Regression for Classification	3
3	Logistic Regression	4
3.1	The Logistic Function	4
3.2	The Logistic Regression Model	5
3.3	Odds and Log-Odds	6
3.4	Fitting Logistic Regression	7
3.5	Logistic Regression in Practice	8
3.6	Multiple Features and Extensions	9
3.7	Regularized Logistic Regression	10
4	Decision Boundaries	11
4.1	What Is a Decision Boundary?	11
4.2	Nonlinear Decision Boundaries	11
5	Linear Discriminant Analysis	18
5.1	Why Another Classifier?	18
5.2	A Bayesian Approach to Classification	18
5.3	The Normality Assumption	19
5.4	The Discriminant Function	20
5.5	Decision Boundaries and Parameter Estimation	21
5.6	The LDA Recipe	21
5.7	LDA in Python	22
5.8	Quadratic Discriminant Analysis	22
5.9	When to Use LDA vs. Logistic Regression	23
6	Evaluating Classification Models	23
6.1	Beyond Accuracy	23
6.2	The Confusion Matrix	24
6.3	The Threshold Trade-off	26
6.4	The ROC Curve	27
6.5	Choosing the Optimal Threshold	29
6.6	Comparing Classifiers	29
7	Summary	29
8	References	30

1 Introduction

This week we move from predicting continuous outcomes (regression) to predicting **categorical outcomes**. This is the classification problem: given features $\mathbf{x}$, predict which class an observation belongs to.

Classification is everywhere in finance. A bank reviewing a loan application wants to predict whether the applicant will default. A credit card company monitors transactions to flag potential fraud. An equity analyst might classify stocks into “buy,” “hold,” or “sell” categories. A risk manager needs to identify which corporate bonds are likely to be downgraded. Insurance companies classify applicants into risk tiers to set premiums. In each case, the goal is the same: use observable characteristics to predict a categorical outcome.

The simplest version is **binary classification**, where there are only two possible outcomes. We encode these as $y \in \{0, 1\}$ —for example, $y = 1$ for “default” and $y = 0$ for “no default.” Given n observations, each with features $\mathbf{x}_i$ and a binary class label y_i , we want to learn a function that predicts y for new observations.

There are two fundamentally different ways to approach classification. The first is to predict the class directly—output 0 or 1 with no uncertainty attached. The second is to predict the *probability* of each class, $P(y = 1 | \mathbf{x})$, and then threshold that probability to make a decision. The second approach is far more useful in practice because it tells us how confident we are in the prediction, not just the prediction itself.

Why does confidence matter? Consider two loan applicants, both classified as “likely to default.” If one has a 51% predicted default probability and the other has a 95% probability, a bank would treat them very differently. The first might receive the loan with extra monitoring; the second would be rejected outright. A model that outputs only “default” or “no default” loses this crucial information.

This chapter covers the two most common linear classification methods: **logistic regression** and **linear discriminant analysis (LDA)**. Both produce linear decision boundaries, but they approach the problem from different angles. We’ll also study **decision boundaries**—the geometric interpretation of classification—and the **evaluation metrics** used to assess how well a classifier performs, which turn out to be surprisingly subtle when classes are imbalanced.

2 Why Not Just Use Regression?

2.1 The Linear Probability Model

If classification is about predicting a 0/1 outcome, why not just use the regression tools we already know? This is a natural first idea: encode the class labels as numbers— $y = 0$ for one class, $y = 1$ for the other—and fit ordinary linear regression. The predicted values $\hat{y}$ can then be interpreted as the probability that $y = 1$.

This approach is called the **linear probability model**, and it has a long history in economics and the social sciences. It’s simple, interpretable, and uses familiar tools. But it has a fundamental flaw that becomes apparent as soon as we try it.

To illustrate, we’ll use credit card default data throughout this chapter. The dataset contains information on 1 million credit card holders: their current balance, their annual income, and whether they defaulted on their payments (1) or not (0). Our goal is to predict default from balance and income.

Python

```
import numpy as np
import pandas as pd

# Load credit default data
data = pd.read_csv('../Slides/credit_default.csv')

balance = data['balance'].values
income = data['income'].values
default = data['default'].values
```

```
print(f"Observations: {len(data)}")
print(f"Default rate: {default.mean():.1%}")
```

Output

```
Observations: 1000000
Default rate: 3.3%
```

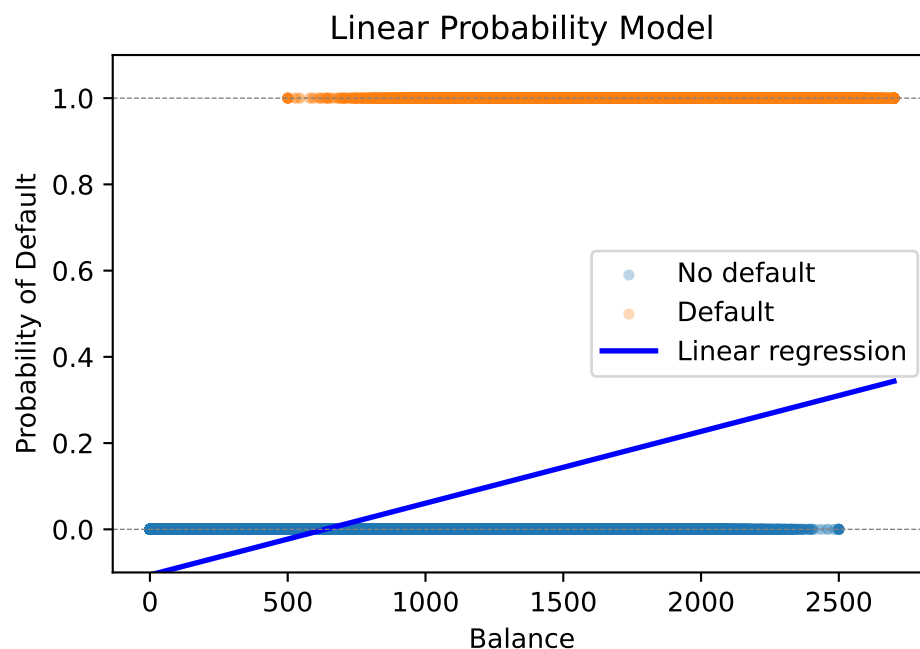
Python

```
from sklearn.linear_model import LinearRegression

# Fit linear regression
X = balance.reshape(-1, 1)
y = default

lr = LinearRegression()
lr.fit(X, y)

# Predictions
balance_grid = np.linspace(0, 2700, 100).reshape(-1, 1)
prob_linear = lr.predict(balance_grid)
```



2.2 The Problem with Linear Regression for Classification

The linear probability model has a fundamental flaw: it can produce “probabilities” that aren’t valid probabilities. By definition, probabilities must lie between 0 and 1. But linear regression is just a line (or hyperplane), and a line extends infinitely in both directions. For sufficiently extreme values of the features, the predicted “probability” will inevitably fall outside the valid range.

Python

```
prob_all = lr.predict(balance.reshape(-1, 1))
```

```
print(f"Minimum predicted probability: {prob_all.min():.3f}")
print(f"Maximum predicted probability: {prob_all.max():.3f}")
print(f"Number of predictions < 0: {(prob_all < 0).sum()}")
print(f"Number of predictions > 1: {(prob_all > 1).sum()}")
```

Output

```
Minimum predicted probability: -0.106
Maximum predicted probability: 0.344
Number of predictions < 0: 330069
Number of predictions > 1: 0
```

What does a probability of -0.05 mean? Or 1.15? These predictions are nonsensical. A probability cannot be negative (that would mean “less likely than impossible”), nor can it exceed 1 (that would mean “more likely than certain”). Yet linear regression happily produces such values because it has no concept of the output being bounded.

This isn’t just a theoretical concern. In our credit default example, people with very low balances get negative predicted probabilities, while those with very high balances might get probabilities above 1. If we’re using these predictions to make decisions—approving or denying credit—we need values we can actually interpret.

We need a function that naturally maps any input to the interval (0, 1), no matter how extreme the features become. This function should be monotonic (higher input means higher or equal output), smooth, and easy to work with mathematically. The logistic function satisfies all these requirements, and logistic regression is built around it.

3 Logistic Regression

3.1 The Logistic Function

Instead of modeling probability as a linear function, logistic regression uses the **logistic function** (also called the sigmoid function):

$$\sigma(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}$$

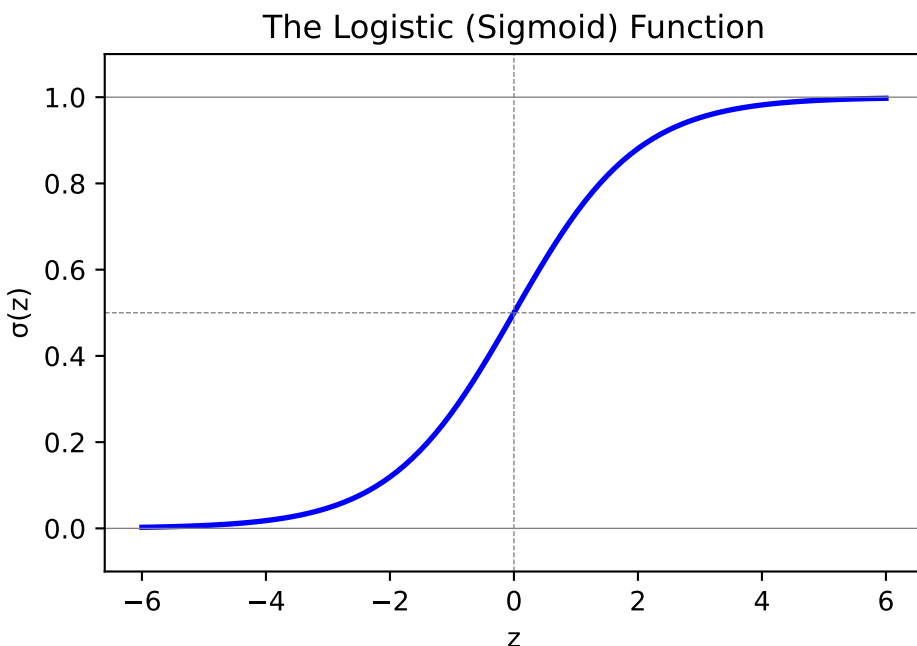
The two expressions are equivalent—you can verify this by multiplying the first form by e^{-z}/e^{-z} . The second form is often more convenient computationally.

This S-shaped function has exactly the properties we need for modeling probabilities. First, the output is always between 0 and 1, no matter what value we plug in for z . As $z \rightarrow -\infty$, the denominator $1 + e^{-z}$ becomes huge (because $e^{-z} \rightarrow \infty$), so $\sigma(z) \rightarrow 0$. As $z \rightarrow +\infty$, we have $e^{-z} \rightarrow 0$, so the denominator approaches 1 and $\sigma(z) \rightarrow 1$. The function smoothly interpolates between these extremes.

Second, the function is monotonically increasing: larger values of z always give larger probabilities. This is intuitive—if the “evidence” for class 1 (as captured by z) increases, the probability of class 1 should increase.

Third, the function is symmetric around $z = 0$. When $z = 0$, we have $\sigma(0) = 1/(1 + 1) = 0.5$ —a coin flip, maximum uncertainty. The function rises from near 0 to near 1, with the steepest part centered at this midpoint.

Fourth, the function has a nice mathematical property: its derivative has a simple form. Specifically, $\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$. This makes optimization (finding the best coefficients) more tractable.



3.2 The Logistic Regression Model

In logistic regression, we model the probability of the positive class as:

$$P(y = 1 | \mathbf{x}) = \sigma(\beta_0 + \beta' \mathbf{x}) = \frac{1}{1 + e^{-(\beta_0 + \beta' \mathbf{x})}}$$

Let's unpack this notation carefully. The expression $P(y = 1 | \mathbf{x})$ reads “the probability that $y = 1$ given $\mathbf{x}$.” This is a **conditional probability**—the probability of default, given that we observe a particular set of feature values. The vertical bar “|” means “given” or “conditional on.” For a credit card holder with balance \$2,000, we're asking: what is the probability this specific person defaults, given their balance?

Here $\mathbf{x} = (x_1, x_2, \dots, x_p)'$ is a p -vector of features (attributes) for an observation. In our credit example, this might be balance and income, so $p = 2$. The vector $\beta = (\beta_1, \beta_2, \dots, \beta_p)'$ contains the corresponding coefficients we need to learn from data. The scalar β_0 is the intercept (also called the bias term).

It helps to define $z = \beta_0 + \beta' \mathbf{x}$ as the **linear predictor**. Writing out the dot product explicitly:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p$$

This should look familiar—it's exactly the same linear combination we used in linear regression. The difference is what we do with it. In linear regression, z is directly the predicted output. In logistic regression, z is the input to the logistic function, which then produces a probability.

The model becomes $P(y = 1 | \mathbf{x}) = 1/(1 + e^{-z})$. Think of z as a “score” that can range from $-\infty$ to $+\infty$. The logistic function compresses this infinite range into the interval $(0, 1)$:

- When $z = 0$: $P(y = 1) = 0.5$. This is the point of maximum uncertainty—a coin flip.
- When $z > 0$: $P(y = 1) > 0.5$. Positive scores indicate the observation is more likely to be class 1.
- When $z < 0$: $P(y = 1) < 0.5$. Negative scores indicate the observation is more likely to be class 0.
- When z is large and positive: $P(y = 1)$ approaches 1. We're very confident it's class 1.
- When z is large and negative: $P(y = 1)$ approaches 0. We're very confident it's class 0.

The coefficients $\beta_0, \beta_1, \dots, \beta_p$ determine how the features translate into the score z . If $\beta_1 > 0$, then higher values of x_1 increase the score, which increases the probability of class 1. If $\beta_1 < 0$, higher x_1 decreases the probability. The magnitude of β_1 determines how strongly x_1 affects the probability.

3.3 Odds and Log-Odds

To interpret logistic regression coefficients, we need to understand odds. You’ve encountered odds in sports betting: “the Leafs are 3-to-1 to win” means that for every 3 times they win, they lose once. Put another way, they win 3 out of every 4 games, or 75% of the time.

Formally, odds are the ratio of the probability of an event to the probability of non-event:

$$\text{Odds} = \frac{P(\text{event})}{P(\text{no event})} = \frac{P(\text{event})}{1 - P(\text{event})}$$

Odds and probability contain the same information but express it differently. If you know one, you can compute the other. The table below shows some correspondences:

Probability	Odds	Interpretation
50%	1:1	Even money—equally likely
75%	3:1	3 times more likely to happen than not
20%	1:4	4 times more likely <i>not</i> to happen
90%	9:1	Very likely
99%	99:1	Almost certain

To go from odds to probability: $P = \text{Odds}/(1 + \text{Odds})$. To go from probability to odds: $\text{Odds} = P/(1 - P)$.

Odds range from 0 (impossible) to ∞ (certain), with 1 being the neutral point (50-50). This asymmetry is awkward. An event with probability 90% has odds 9:1, while an event with probability 10% has odds 1:9 (or 0.111). The odds scale is “stretched out” above 1 and “compressed” below 1.

Taking the logarithm fixes this asymmetry. The **log-odds** (also called the **logit**) is simply $\ln(\text{Odds})$:

- Log-odds of 0 means odds = 1, which means probability = 50% (maximum uncertainty)
- Positive log-odds means odds > 1, which means probability > 50% (more likely than not)
- Negative log-odds means odds < 1, which means probability < 50% (less likely than not)
- Log-odds can range from $-\infty$ to $+\infty$, symmetric around 0

Now here’s the connection to logistic regression. If we solve the logistic regression model for the log-odds, we get:

$$\ln \left(\frac{P(y = 1 | \mathbf{x})}{1 - P(y = 1 | \mathbf{x})} \right) = \beta_0 + \beta' \mathbf{x}$$

In other words, **logistic regression models the log-odds as a linear function of the features**. This is why it’s sometimes called a “linear model”—not because the probability is linear (it isn’t), but because the log-odds is.

This gives us a clean interpretation of the coefficients. The coefficient β_j tells us how a one-unit increase in x_j changes the log-odds, holding other features constant:

- If $\beta_j > 0$: higher x_j increases the log-odds, which increases the probability of $y = 1$
- If $\beta_j < 0$: higher x_j decreases the log-odds, which decreases the probability of $y = 1$
- If $\beta_j = 0$: x_j has no effect on the probability

We can also interpret coefficients as **odds ratios** by exponentiating. If we increase x_j by one unit, the log-odds changes by β_j , which means the odds get multiplied by e^{β_j} . For example:

- If $\beta_j = 0.5$, then $e^{0.5} \approx 1.65$: each one-unit increase in x_j multiplies the odds by 1.65 (a 65% increase in odds)
- If $\beta_j = -0.3$, then $e^{-0.3} \approx 0.74$: each one-unit increase in x_j multiplies the odds by 0.74 (a 26% decrease in odds)
- If $\beta_j = 0$, then $e^0 = 1$: the odds are unchanged

This odds ratio interpretation is common in medical research and epidemiology. You might read that “each additional year of smoking increases the odds of lung cancer by 8%,” which means the coefficient on years-smoked is $\ln(1.08) \approx 0.077$.

3.4 Fitting Logistic Regression

How do we find the best coefficients β_0, β ? As with any machine learning model, we define a **loss function** that measures how badly the model fits the data, then find the coefficients that minimize this loss.

For observation i with features $\mathbf{x}_i$ and label y_i , let $\hat{p}_i = P(y_i = 1 | \mathbf{x}_i)$ be our predicted probability. What makes a good prediction? The intuition is straightforward:

- If $y_i = 1$ (the observation actually belongs to class 1), we want $\hat{p}_i$ close to 1
- If $y_i = 0$ (the observation actually belongs to class 0), we want $\hat{p}_i$ close to 0

In other words, a good model assigns high probability to the class that actually occurred.

Why not just use squared error, $(y_i - \hat{p}_i)^2$, like we did in regression? We could, but it turns out to have some undesirable properties for classification. The optimization landscape has flat regions that make gradient descent slow, and the loss doesn’t penalize confident wrong predictions as severely as it should.

Instead, logistic regression uses the **binary cross-entropy loss** (also called log loss):

$$\mathcal{L}(\beta) = -\frac{1}{n} \sum_{i=1}^n [y_i \ln(\hat{p}_i) + (1 - y_i) \ln(1 - \hat{p}_i)]$$

This formula looks complicated, but it’s actually quite intuitive once you break it down. For each observation, only one of the two terms is “active” (the other is multiplied by zero):

- When $y_i = 1$: the loss contribution is $-\ln(\hat{p}_i)$. If $\hat{p}_i = 0.99$, this is $-\ln(0.99) \approx 0.01$ (very small, good!). If $\hat{p}_i = 0.01$, this is $-\ln(0.01) \approx 4.6$ (large, bad!).
- When $y_i = 0$: the loss contribution is $-\ln(1 - \hat{p}_i)$. If $\hat{p}_i = 0.01$, this is $-\ln(0.99) \approx 0.01$ (very small, good!). If $\hat{p}_i = 0.99$, this is $-\ln(0.01) \approx 4.6$ (large, bad!).

The logarithm has a crucial property: as the predicted probability for the correct class approaches 0, the loss goes to infinity. This severely penalizes confident wrong predictions. If the model says “I’m 99% sure this is class 0” but it’s actually class 1, the loss is enormous. This encourages the model to be well-calibrated—to only be confident when it should be.

Both linear and logistic regression fit into the same ML framework: define a loss function, then minimize it. The table below compares them:

	Linear Regression	Logistic Regression
Output	Continuous $\hat{y}$	Probability $\hat{p} \in (0, 1)$
Loss function	Mean squared error (MSE)	Binary cross-entropy
Formula	$\frac{1}{n} \sum_i (y_i - \hat{y}_i)^2$	$-\frac{1}{n} \sum_i [y_i \ln(\hat{p}_i) + (1 - y_i) \ln(1 - \hat{p}_i)]$
Optimization	Closed-form solution	Iterative (gradient descent)

The key difference in the last row: linear regression has a closed-form solution (we can write down a formula for the optimal coefficients), but logistic regression does not. The cross-entropy loss with the logistic function creates a nonlinear optimization problem. We solve it using iterative methods like gradient descent or Newton's method, starting from an initial guess and repeatedly improving until convergence. Fortunately, the optimization problem is **convex**, meaning there's a single global minimum and we're guaranteed to find it.

Connection to statistics

In statistics, minimizing cross-entropy loss is equivalent to **maximum likelihood estimation**. The idea is to find parameters that make the observed data most probable. If we assume each observation is an independent Bernoulli trial with probability $\hat{p}_i$, then the likelihood of observing our data is $\prod_i \hat{p}_i^{y_i} (1 - \hat{p}_i)^{1-y_i}$. Taking the negative log gives exactly the cross-entropy loss. Minimizing MSE is equivalent to maximum likelihood under the assumption that errors are normally distributed. Both approaches—ML and statistics—arrive at the same answer through different reasoning.

3.5 Logistic Regression in Practice

Let's fit logistic regression to our credit default data:

Python

```
from sklearn.linear_model import LogisticRegression

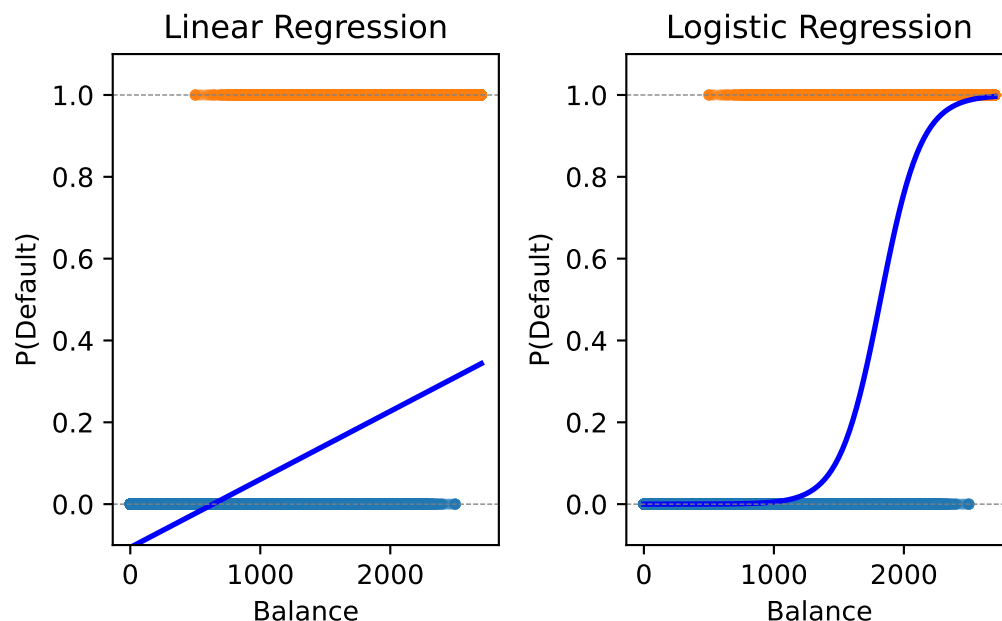
# Fit logistic regression
log_reg = LogisticRegression()
log_reg.fit(X, y)

# Predictions
prob_logistic = log_reg.predict_proba(balance_grid)[: , 1]

print(f"Logistic Regression:")
print(f" Intercept: {log_reg.intercept_[0]:.4f}")
print(f" Coefficient on balance: {log_reg.coef_[0, 0]:.6f}")
```

Output

```
Logistic Regression:
Intercept: -11.6163
Coefficient on balance: 0.006383
```



The logistic curve stays within $[0, 1]$ and captures the S-shaped relationship between balance and default probability.

Logistic regression gives us a probability. To make a classification decision, we need a **threshold** (also called a cutoff). The default rule is to predict class 1 if $P(y = 1 | \mathbf{x}) > 0.5$.

Python

```
# Predict probabilities and classes
prob_pred = log_reg.predict_proba(X)[: , 1]
class_pred = (prob_pred > 0.5).astype(int)

# Compare to actual
print(f"Using threshold = 0.5:")
print(f" Predicted defaults: {class_pred.sum()}")
print(f" Actual defaults: {int(default.sum())}")
print(f" Correctly classified: {(class_pred == default).sum()} / {len(default)}")
print(f" Accuracy: {(class_pred == default).mean():.1%}")
```

Output

```
Using threshold = 0.5:
Predicted defaults: 19005
Actual defaults: 33000
Correctly classified: 975591 / 1000000
Accuracy: 97.6%
```

The 0.5 threshold isn't always optimal—we'll return to this when discussing evaluation metrics.

3.6 Multiple Features and Extensions

Logistic regression easily extends to multiple predictors:

$$P(y = 1 | \mathbf{x}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p)}}$$

Python

```
# Fit with both balance and income
X_both = np.column_stack([balance, income])
log_reg_both = LogisticRegression()
log_reg_both.fit(X_both, default)

print(f"Logistic Regression with Balance and Income:")
print(f"  Intercept: {log_reg_both.intercept_[0]:.4f}")
print(f"  Coefficient on balance: {log_reg_both.coef_[0, 0]:.6f}")
print(f"  Coefficient on income: {log_reg_both.coef_[0, 1]:.9f}")
```

Output

```
Logistic Regression with Balance and Income:
  Intercept: -11.2532
  Coefficient on balance: 0.006383
  Coefficient on income: -0.000009279
```

The coefficient on income is tiny—income adds little predictive power beyond balance.

When we have $K > 2$ classes, we can extend logistic regression using the **softmax function**. For each class k , we define a linear predictor $z_k = \beta_{k,0} + \beta'_k \mathbf{x}$. The probability of class k is:

$$P(y = k | \mathbf{x}) = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}$$

This is called **multinomial logistic regression** or **softmax regression**. The softmax function ensures that each probability is between 0 and 1 and that the probabilities sum to 1 across all classes.

3.7 Regularized Logistic Regression

Just like linear regression, logistic regression can overfit—especially with many features. We can add regularization. **Lasso logistic regression** maximizes:

$$\mathcal{L}(\beta) - \lambda \sum_{j=1}^p |\beta_j|$$

The penalty $\lambda \sum |\beta_j|$ shrinks coefficients toward zero and can set some exactly to zero (variable selection). This prevents overfitting when p is large relative to n , identifies which features matter most, and improves out-of-sample prediction. The regularization parameter λ is chosen by cross-validation.

Python

```
from sklearn.linear_model import LogisticRegressionCV

# Fit Lasso logistic regression with CV
log_reg_lasso = LogisticRegressionCV(penalty='l1', solver='saga', cv=5, max_iter=1000)
log_reg_lasso.fit(X_both, default)

print(f"Lasso Logistic Regression ( chosen by CV):")
print(f"  Best C (inverse of ): {log_reg_lasso.C_[0]:.4f}")
print(f"  Coefficient on balance: {log_reg_lasso.coef_[0, 0]:.6f}")
print(f"  Coefficient on income: {log_reg_lasso.coef_[0, 1]:.9f}")
```

Output

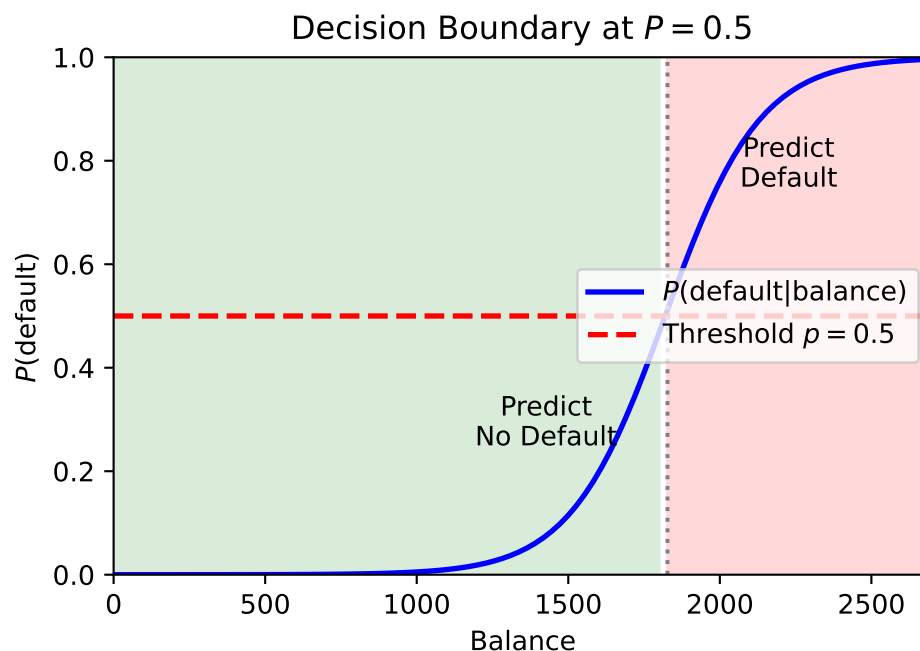
```
Lasso Logistic Regression ( chosen by CV):
Best C (inverse of ): 0.0001
Coefficient on balance: 0.001064
Coefficient on income: -0.000123499
```

4 Decision Boundaries

4.1 What Is a Decision Boundary?

Think of a classifier as drawing a line (or curve) through feature space that separates the classes. The **decision boundary** is this dividing line—observations on one side get predicted as Class 0, observations on the other side as Class 1.

For logistic regression with threshold 0.5, we predict Class 1 when $P(y = 1 | \mathbf{x}) > 0.5$. The boundary is where $P(y = 1 | \mathbf{x}) = 0.5$ exactly—the point of maximum uncertainty.



The horizontal red line marks $P = 0.5$. Where the probability curve crosses this threshold defines the decision boundary in feature space—observations with balance above this point are predicted to default.

With multiple features, the boundary is where $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots = 0$ —a linear equation in the features. This is why logistic regression is called a **linear classifier**: the decision boundary is a line (in 2D) or hyperplane (in higher dimensions).

4.2 Nonlinear Decision Boundaries

Sometimes a straight line can't separate the classes well. We can create curved boundaries by adding transformed features to our model. If we include x_1^2 , x_2^2 , and $x_1 x_2$ as additional features, the model is still logistic regression (linear in these new features), but the boundary is now curved in the original (x_1, x_2) space.

Consider data where Class 0 forms an inner ring and Class 1 forms an outer ring. No straight line can separate these classes—we need a circular boundary. A circle centered at the origin has equation $x_1^2 + x_2^2 = r^2$. If we add squared terms as features, the decision boundary becomes:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_1 x_2 + \beta_5 x_2^2 = 0$$

This is a quadratic equation in x_1 and x_2 —it can represent circles, ellipses, or other curved shapes.

Python

```
from sklearn.preprocessing import PolynomialFeatures

# Linear: uses only x1, x2
log_reg_linear = LogisticRegression()
log_reg_linear.fit(X_ring, y_ring)

# Quadratic: add x1^2, x2^2, x1*x2 as new features
poly = PolynomialFeatures(degree=2)
X_ring_poly = poly.fit_transform(X_ring)
log_reg_quad = LogisticRegression()
log_reg_quad.fit(X_ring_poly, y_ring)
```

penalty penalty: {'l1', 'l2',
'elasticnet', None}, default='l2' 'deprecated'

Specify the norm of the penalty:

- 'None': no penalty is added;
- 'l2': add a L2 penalty term and it is the default choice;
- 'l1': add a L1 penalty term;
- 'elasticnet': both L1 and L2 penalty terms are added.

.. warning::
Some penalties may not work with some solvers. See the parameter 'solver' below, to know the compatibility between the penalty and solver.

.. versionadded:: 0.19
l1 penalty with SAGA solver
(allowing 'multinomial' + L1)

.. deprecated:: 1.8
'penalty' was deprecated in version 1.8 and will be removed in 1.10. Use 'l1_ratio' instead.
'l1_ratio=0' for 'penalty='l2',
'l1_ratio=1' for 'penalty='l1' and 'l1_ratio' set to any float between 0 and 1 for 'penalty='elasticnet'.

C	C: float, default=1.0	1.0
	Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization. ‘C=np.inf’ results in unpenalized logistic regression. For a visual example on the effect of tuning the ‘C’ parameter with an L1 penalty, see: :ref:‘sphx_glr_auto_examples_linear_model_plot_logistic_path.py’.	
l1_ratio	l1_ratio: float, default=0.0	0.0
	The Elastic-Net mixing parameter, with ‘0 <= l1_ratio <= 1’. Setting ‘l1_ratio=1’ gives a pure L1-penalty, setting ‘l1_ratio=0’ a pure L2-penalty. Any value between 0 and 1 gives an Elastic-Net penalty of the form ‘l1_ratio * L1 + (1 - l1_ratio) * L2’.	
	.. warning:: Certain values of ‘l1_ratio’, i.e. some penalties, may not work with some solvers. See the parameter ‘solver’ below, to know the compatibility between the penalty and solver.	
	.. versionchanged:: 1.8 Default value changed from None to 0.0.	
	.. deprecated:: 1.8 ‘None’ is deprecated and will be removed in version 1.10. Always use ‘l1_ratio’ to specify the penalty type.	

dual dual: bool, default=False	False
Dual (constrained) or primal (regularized, see also :ref:'this equation ') formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer 'dual=False' when n_samples > n_features.	
tol tol: float, default=1e-4	0.0001
Tolerance for stopping criteria.	
fit_intercept fit_intercept: bool, default=True	True
Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.	
intercept_scaling intercept_scaling: float, default=1	1
Useful only when the solver 'liblinear' is used and 'self.fit_intercept' is set to 'True'. In this case, 'x' becomes '[x, self.intercept_scaling]', i.e. a "synthetic" feature with constant value equal to 'intercept_scaling' is appended to the instance vector. The intercept becomes "intercept_scaling * synthetic_feature_weight".	
.. note:: The synthetic feature weight is subject to L1 or L2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) 'intercept_scaling' has to be increased.	

`class_weight` `class_weight`: dict or `None`
'balanced', default=None

Weights associated with classes in the form “{class_label: weight}”. If not given, all classes are supposed to have weight one.

The “balanced” mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as “`n_samples / (n_classes * np.bincount(y))`”.

Note that these weights will be multiplied with `sample_weight` (passed through the fit method) if `sample_weight` is specified.

.. versionadded:: 0.17
`class_weight='balanced'`
`random_state` `random_state`: int, `None`
RandomState instance,
default=None

Used when “`solver`” == ‘sag’, ‘saga’ or ‘liblinear’ to shuffle the data. See :term:‘Glossary ‘ for details.

```
solver solver: {'lbfgs', 'liblinear', 'lbfgs',
'newton-cg', 'newton-cholesky',
'sag', 'saga'}, default='lbfgs'
```

Algorithm to use in the optimization problem. Default is 'lbfgs'.

To choose a solver, you might want to consider the following aspects:

- 'lbfgs' is a good default solver because it works reasonably well for a wide class of problems.
- For :term:'multiclass' problems ('n_classes' >= 3), all solvers except 'liblinear' minimize the full multinomial loss, 'liblinear' will raise an error.
- 'newton-cholesky' is a good choice for 'n_samples' » 'n_features * n_classes', especially with one-hot encoded categorical features with rare categories. Be aware that the memory usage of this solver has a quadratic dependency on 'n_features * n_classes' because it explicitly computes the full Hessian matrix.
- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones;
- 'liblinear' can only handle binary classification by default. To apply a one-versus-rest scheme for the multiclass setting one can wrap it with the :class:'~sklearn.multiclass.OneVsRestClassifier'.

```
.. warning::
The choice of the algorithm
depends on the penalty chosen
('l1_ratio=0'
for L2-penalty, 'l1_ratio=1' for
L1-penalty and '0 < l1_ratio < 1'
for
Elastic-Net) and on (multinomial
multiclass supp
```

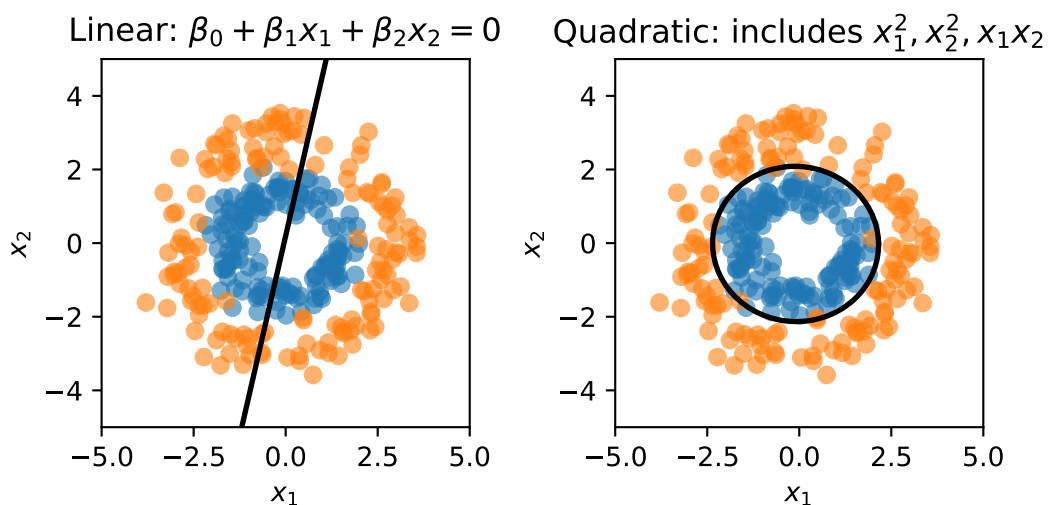
<code>max_iter</code>	<code>max_iter: int, default=100</code>	100
Maximum number of iterations taken for the solvers to converge.		
<code>verbose</code>	<code>verbose: int, default=0</code>	0
For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.		
<code>warm_start</code>	<code>warm_start: bool, default=False</code>	False
When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver. See :term:`the Glossary`.		
.. versionadded:: 0.17 *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.		
<code>n_jobs</code>	<code>n_jobs: int, default=None</code>	None
Does not have any effect.		
.. deprecated:: 1.8 ‘n_jobs’ is deprecated in version 1.8 and will be removed in 1.10.		

To visualize the decision boundary, we create a grid of points, compute $P(y = 1)$ at each point, then draw the contour where $P = 0.5$.

Python

```
# Create grid covering the feature space
x1_range = np.linspace(-5, 5, 100)
x2_range = np.linspace(-5, 5, 100)
X1, X2 = np.meshgrid(x1_range, x2_range)
grid = np.column_stack([X1.ravel(), X2.ravel()])

# Predict P(y=1) at every grid point
probs_linear = log_reg_linear.predict_proba(grid)[: , 1].reshape(X1.shape)
probs_quad = log_reg_quad.predict_proba(poly.transform(grid))[: , 1].reshape(X1.shape)
```



The linear model is forced to draw a straight line through the rings. The quadratic model can learn a circular boundary that actually separates the classes.

5 Linear Discriminant Analysis

5.1 Why Another Classifier?

Before diving into the mechanics, it helps to see where LDA fits relative to what we've already covered. The table below compares three approaches:

	Clustering (Week 5)	Logistic Regression	LDA
Type	Unsupervised	Supervised	Supervised
Labels	Unknown — discover them	Known — learn a boundary	Known — learn distributions
Strategy	Assume each group is a distribution; find the groups	Directly model $P(y \mathbf{x})$	Model $P(\mathbf{x} y)$ per class, then apply Bayes' theorem

LDA is the supervised version of the distributional thinking used in clustering. In clustering, we assumed each group came from a distribution and tried to discover the groups. In LDA, we already know the groups and want to learn what makes each group different so we can classify new observations. In practice, LDA and logistic regression often give similar answers — the value is in understanding both ways of thinking about classification.

5.2 A Bayesian Approach to Classification

Logistic regression directly models $P(y|\mathbf{x})$ —the probability of the class given the features. This is called the **discriminative** approach: we learn to discriminate between classes without modeling where the data comes from.

Discriminant analysis takes a fundamentally different approach. Instead of directly modeling the probability of class given features, it asks: “If I knew an observation came from class k , what would its features look like?” This is called the **generative** approach because we're modeling how the data is generated within each class.

Why would we do this? The key insight is Bayes' theorem, which lets us flip conditional probabilities:

$$P(y = k|\mathbf{x}) = \frac{P(\mathbf{x}|y = k) \cdot P(y = k)}{P(\mathbf{x})}$$

In words:

$$\text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}}$$

Let's unpack each term:

- **Posterior** $P(y = k|\mathbf{x})$: What we want—the probability of class k given the observed features. This is what we'll use to classify new observations.
- **Likelihood** $P(\mathbf{x}|y = k)$: The probability density of the features, given that the observation comes from class k . This answers “what do class- k observations look like?”
- **Prior** $P(y = k)$: How common is class k in the population, before we observe any features? If 3% of loans default, the prior probability of default is 0.03.
- **Evidence** $P(\mathbf{x})$: The overall probability density of the features, regardless of class. This is a normalizing constant that ensures the posteriors sum to 1.

The generative approach has a nice interpretation. Imagine you're trying to identify whether an email is spam or legitimate. The discriminative approach (logistic regression) asks: “Given these words, how likely is spam?” The generative approach asks: “What words do spam emails typically contain? What words do legitimate emails contain? Given my observation, which pattern is it more consistent with?”

Let's establish notation. We have K classes labeled $1, 2, \dots, K$. Let $\pi_k = P(y = k)$ be the prior probability of class k , and let $f_k(\mathbf{x}) = P(\mathbf{x}|y = k)$ be the probability density of $\mathbf{x}$ given class k . Bayes' theorem gives us the posterior probability:

$$P(y = k|\mathbf{x}) = \frac{f_k(\mathbf{x})\pi_k}{\sum_{j=1}^K f_j(\mathbf{x})\pi_j}$$

The denominator $\sum_{j=1}^K f_j(\mathbf{x})\pi_j$ is the total probability of observing $\mathbf{x}$ across all classes. It's the same for all classes k , so it's just a normalizing constant that ensures the posteriors sum to 1. For classification purposes, we can often ignore it and just compare the numerators.

We classify to the class with the highest posterior probability: $\hat{y} = \arg \max_k P(y = k|\mathbf{x})$. Since the denominator is constant across classes, this is equivalent to $\hat{y} = \arg \max_k f_k(\mathbf{x})\pi_k$.

5.3 The Normality Assumption

The Bayesian framework tells us *how* to combine the likelihood and prior, but it doesn't tell us *what* the likelihood looks like. We need to specify a model for $f_k(\mathbf{x})$ —the distribution of features within each class.

Linear Discriminant Analysis (LDA) makes a specific assumption: within each class, the features follow a **multivariate normal distribution**. In notation:

$$\mathbf{X} | y = k \sim \mathcal{N}(\mu_k, \Sigma)$$

This says: “If an observation belongs to class k , its features are drawn from a multivariate normal distribution with mean μ_k and covariance matrix Σ .”

The density function is:

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2} |\Sigma|^{1/2}} \exp \left(-\frac{1}{2} (\mathbf{x} - \mu_k)' \Sigma^{-1} (\mathbf{x} - \mu_k) \right)$$

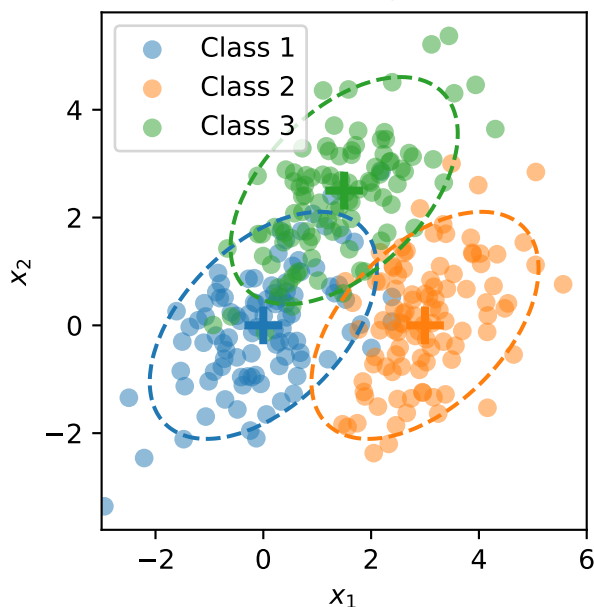
This looks intimidating, but the key parts are:

- μ_k is the mean vector of class k . Different classes have different centers.
- Σ is the covariance matrix. It describes the shape (how spread out, how correlated) of each class.
- The term $(\mathbf{x} - \mu_k)' \Sigma^{-1} (\mathbf{x} - \mu_k)$ is the squared **Mahalanobis distance** from $\mathbf{x}$ to the class center μ_k . Points far from the center have lower density.

The critical assumption in LDA is that **all classes share the same covariance matrix** Σ . This means each class is a normal “blob” with its own center μ_k , but all the blobs have the same shape—the same orientation, the same spread in each direction, the same correlations between features. Only the centers differ.

Why make this assumption? It dramatically reduces the number of parameters we need to estimate. A covariance matrix for p features has $p(p+1)/2$ unique entries. With K classes, allowing separate covariances would require estimating $K \times p(p+1)/2$ parameters. With shared covariance, we only estimate $p(p+1)/2$ parameters (plus the K mean vectors). When p is large or n is small, this parsimony matters a lot.

LDA: Same Covariance, Different Means



The dashed ellipses show the 95% probability contours—they have the same shape (orientation and spread) but different centers.

5.4 The Discriminant Function

Start with the posterior from Bayes' theorem:

$$P(y = k | \mathbf{x}) = \frac{f_k(\mathbf{x})\pi_k}{\sum_{j=1}^K f_j(\mathbf{x})\pi_j}$$

Taking the log and plugging in the normal density:

$$\ln P(y = k | \mathbf{x}) = \ln f_k(\mathbf{x}) + \ln \pi_k - \underbrace{\ln \sum_j f_j(\mathbf{x})\pi_j}_{\text{same for all } k}$$

The normal density gives $\ln f_k(\mathbf{x}) = -\frac{p}{2} \ln(2\pi) - \frac{1}{2} \ln |\Sigma| - \frac{1}{2} (\mathbf{x} - \mu_k)' \Sigma^{-1} (\mathbf{x} - \mu_k)$.

The first two terms don't depend on k (shared covariance!). Expanding the quadratic and dropping terms that don't depend on k , we get the **discriminant function**:

$$\delta_k(\mathbf{x}) = \mathbf{x}'\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k'\Sigma^{-1}\mu_k + \ln \pi_k$$

This is a scalar—one number for each class k . We classify $\mathbf{x}$ to the class with the largest discriminant: $\hat{y} = \arg \max_k \delta_k(\mathbf{x})$.

The discriminant function is linear in $\mathbf{x}$ —that's why it's called *Linear* Discriminant Analysis.

5.5 Decision Boundaries and Parameter Estimation

The decision boundary between classes k and ℓ is where $\delta_k(\mathbf{x}) = \delta_\ell(\mathbf{x})$. This simplifies to:

$$\mathbf{x}'\Sigma^{-1}(\mu_k - \mu_\ell) = \frac{1}{2}(\mu_k'\Sigma^{-1}\mu_k - \mu_\ell'\Sigma^{-1}\mu_\ell) + \ln \frac{\pi_\ell}{\pi_k}$$

This is a linear equation in $\mathbf{x}$ —so the boundary is a line (in 2D) or hyperplane (in higher dimensions). With K classes, we have $K(K-1)/2$ pairwise boundaries, but only $K-1$ of them matter for defining the decision regions.

In practice, we estimate the parameters from training data. The prior probabilities are estimated as $\hat{\pi}_k = n_k/n$, where n_k is the number of training observations in class k . The class means are estimated as $\hat{\mu}_k = \frac{1}{n_k} \sum_{i:y_i=k} \mathbf{x}_i$. The pooled covariance matrix is:

$$\hat{\Sigma} = \frac{1}{n-K} \sum_{k=1}^K \sum_{i:y_i=k} (\mathbf{x}_i - \hat{\mu}_k)(\mathbf{x}_i - \hat{\mu}_k)'$$

The pooled covariance averages within-class covariances, weighted by class size.

5.6 The LDA Recipe

It's worth stepping back and summarizing how LDA works as a complete method:

	LDA
Model	Each class k is a multivariate normal: $\mathbf{x} \mid y = k \sim \mathcal{N}(\mu_k, \Sigma)$
Parameters	Priors $\hat{\pi}_k = n_k/n$, means $\hat{\mu}_k$, pooled covariance $\hat{\Sigma}$
“Loss function”	Not a loss function — parameters are estimated directly from the data (sample proportions, sample means, pooled covariance)
Classification rule	Assign $\mathbf{x}$ to the class with the largest discriminant $\delta_k(\mathbf{x})$

No optimization loop, no gradient descent. LDA computes its parameters in closed form — plug in the training data and you're done. This is fundamentally different from logistic regression, which iteratively searches for the coefficients that minimize cross-entropy loss.

5.7 LDA in Python

Python

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Combine the 3-class data
X_lda = np.vstack([X1, X2, X3])
y_lda = np.array([0] * n_per_class + [1] * n_per_class + [2] * n_per_class)

# Fit LDA
lda = LinearDiscriminantAnalysis()
lda.fit(X_lda, y_lda)

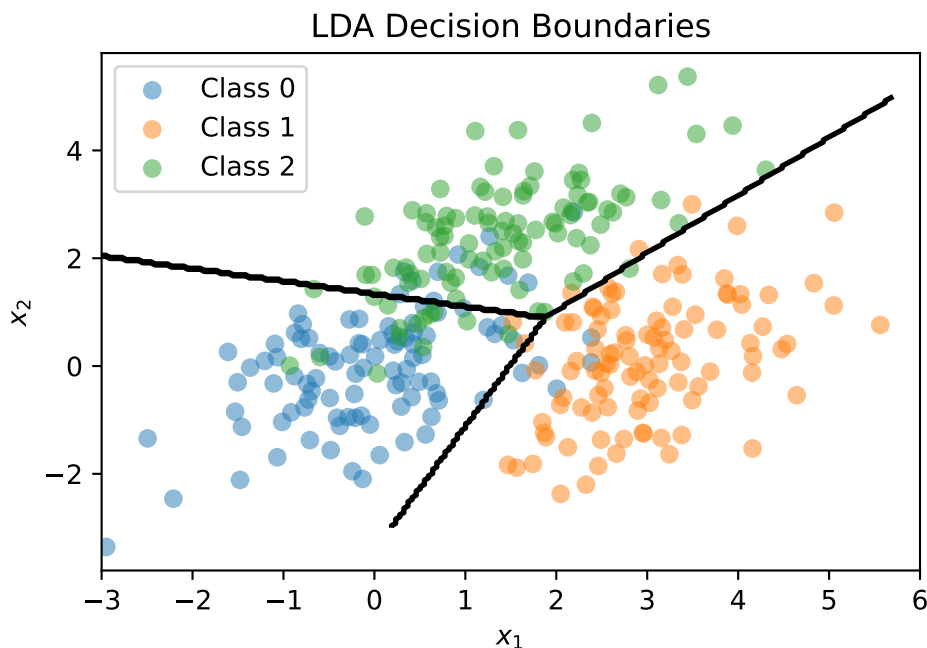
print("LDA Class Means:")
for k in range(3):
    print(f"  Class {k}: {lda.means_[k]}")

print(f"\nClass Priors: {lda.priors_}")
```

Output

```
LDA Class Means:
Class 0: [ 0.00962094 -0.02608294]
Class 1: [3.00835928  0.12294623]
Class 2: [1.40152169  2.3715806 ]

Class Priors: [0.33333333 0.33333333 0.33333333]
```

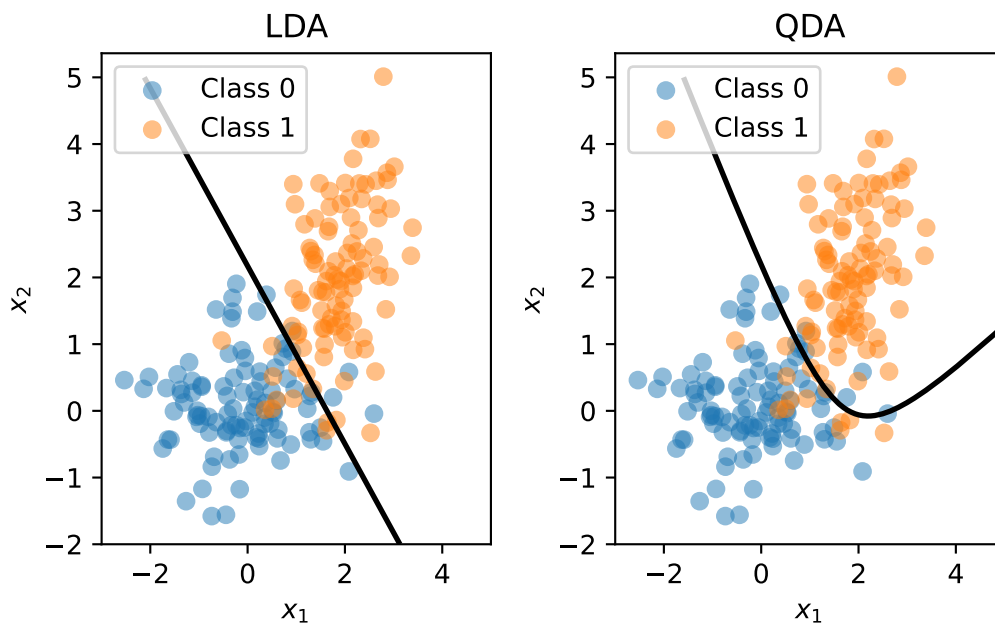


5.8 Quadratic Discriminant Analysis

LDA assumes all classes share the same covariance matrix Σ . **Quadratic Discriminant Analysis (QDA)** relaxes this: each class has its own covariance Σ_k . The discriminant function becomes:

$$\delta_k(\mathbf{x}) = -\frac{1}{2} \ln |\Sigma_k| - \frac{1}{2} (\mathbf{x} - \mu_k)' \Sigma_k^{-1} (\mathbf{x} - \mu_k) + \ln \pi_k$$

This is quadratic in $\mathbf{x}$, giving curved decision boundaries. There is a trade-off between the methods: LDA has more restrictive assumptions, fewer parameters to estimate, and is more stable; QDA is more flexible, has more parameters, and can overfit with small samples.



When classes have different covariances, QDA captures the curved boundary while LDA is forced to use a straight line.

5.9 When to Use LDA vs. Logistic Regression

LDA and QDA have good track records as classifiers, not necessarily because the normality assumption is correct, but because estimating fewer parameters (shared Σ) reduces variance, the decision boundary often works well even when normality is violated, and there is a closed-form solution with no iterative optimization needed.

When to use which: use LDA when you have limited data or classes are well-separated, use QDA when you have more data and suspect different class shapes, and consider regularized discriminant analysis to blend between LDA and QDA.

Both LDA and logistic regression produce linear decision boundaries. Logistic regression makes no assumption about the distribution of $\mathbf{x}$, is more robust when the normality assumption is violated, and is preferred when you have binary features or mixed feature types. LDA is more efficient when normality holds (uses information about class distributions), can be more stable with small samples, and naturally handles multi-class problems. In practice, they often give similar results. Try both and compare via cross-validation.

6 Evaluating Classification Models

6.1 Beyond Accuracy

For regression, we use MSE or R^2 to measure how well a model predicts continuous outcomes. Classification requires different metrics because the outcomes are categories, not numbers.

The most obvious metric is **accuracy**—the percentage of predictions that are correct:

$$\text{Accuracy} = \frac{\text{Number Correct}}{\text{Total}} = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP, TN, FP, and FN are the counts in the confusion matrix (defined below). An accuracy of 95% sounds good: we’re right 19 times out of 20!

But accuracy can be deeply misleading, especially when **classes are imbalanced**—meaning one class is much more common than the other. In finance, imbalanced classes are the norm, not the exception:

- Credit card fraud: ~0.1% of transactions are fraudulent
- Loan defaults: ~3-5% of loans default
- Bankruptcy: <1% of firms go bankrupt in a given year
- Extreme market moves: <5% of days have returns beyond 2 standard deviations

Consider a fraud detection model where only 0.1% of transactions are actually fraudulent. A model that predicts “not fraud” for every single transaction achieves 99.9% accuracy! It’s correct on all the legitimate transactions (99.9% of the data) and wrong only on the frauds (0.1% of the data). This “model” is useless—it catches zero frauds, which was the entire point.

The problem is that accuracy treats all errors equally. In fraud detection, a false negative (missing a fraud) costs far more than a false positive (investigating a legitimate transaction). In medical diagnosis, missing a cancer might be fatal, while a false alarm leads to more tests. We need metrics that distinguish between types of errors.

6.2 The Confusion Matrix

The **confusion matrix** organizes all the ways a classifier can be right or wrong. For binary classification, there are exactly four possibilities:

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

Let’s define each cell:

- **True Positive (TP):** We predicted positive, and it actually was positive. In fraud detection: we flagged a transaction as fraud, and it really was fraud. This is a correct positive prediction.
- **False Positive (FP):** We predicted positive, but it was actually negative. In fraud detection: we flagged a transaction as fraud, but it was legitimate. This is a false alarm, also called a Type I error. The “positive” part refers to what we predicted, not whether it’s good.
- **True Negative (TN):** We predicted negative, and it actually was negative. In fraud detection: we said “not fraud,” and it was indeed not fraud. This is a correct negative prediction.
- **False Negative (FN):** We predicted negative, but it was actually positive. In fraud detection: we said “not fraud,” but it actually was fraud. We missed a real positive. This is also called a Type II error.

Memory tip: The first word (True/False) tells you if the prediction was correct. The second word (Positive/Negative) tells you what you predicted.

Different applications care about different cells. In credit card fraud detection, false negatives are very costly (the bank loses money on undetected fraud), while false positives are just annoying (a legitimate customer gets a call). In airport security screening, false negatives are catastrophic (a threat gets through), so we tolerate many false positives (innocent passengers get extra screening).

From the confusion matrix, we can compute several metrics that capture different aspects of performance:

Accuracy is the overall correct rate:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

This is the fraction of all predictions that were correct. As discussed, it can be misleading with imbalanced classes.

Precision answers: “Of those we predicted positive, how many actually are?”

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

High precision means few false alarms. If our fraud detector has 90% precision, then 9 out of 10 transactions we flag are actually fraudulent.

Recall (also called **sensitivity** or **true positive rate**) answers: “Of the actual positives, how many did we catch?”

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

High recall means we catch most of the positives. If our fraud detector has 80% recall, we catch 80% of actual frauds.

Specificity (also called **true negative rate**) answers: “Of the actual negatives, how many did we correctly identify?”

$$\text{Specificity} = \frac{TN}{TN + FP}$$

High specificity means few legitimate observations are flagged as positive.

False Positive Rate is the complement of specificity:

$$\text{FPR} = \frac{FP}{TN + FP} = 1 - \text{Specificity}$$

This is the fraction of negatives that we incorrectly labeled as positive.

Python

```
from sklearn.metrics import confusion_matrix, classification_report

# Using our credit default data with logistic regression
y_pred = log_reg.predict(X)

# Confusion matrix
cm = confusion_matrix(default, y_pred)
print("Confusion Matrix:")
print(f"                Predicted No   Predicted Yes")
print(f"Actual No         {cm[0,0]:5d}       {cm[0,1]:5d}")
print(f"Actual Yes        {cm[1,0]:5d}       {cm[1,1]:5d}")
```

Output

```
Confusion Matrix:
                Predicted No   Predicted Yes
Actual No         961793       5207
Actual Yes        19202        13798
```

Python

```
# Metrics
TP = cm[1, 1]
TN = cm[0, 0]
```

```

FP = cm[0, 1]
FN = cm[1, 0]

print(f"\nMetrics:")
print(f" Accuracy:      {(TP + TN) / (TP + TN + FP + FN):.1%}")
print(f" Precision:     {TP / (TP + FP):.1%}" if (TP + FP) > 0 else " Precision: N/A")
print(f" Recall:       {TP / (TP + FN):.1%}")
print(f" Specificity:  {TN / (TN + FP):.1%}")

```

Output

```

Metrics:
Accuracy:      97.6%
Precision:     72.6%
Recall:       41.8%
Specificity: 99.5%

```

6.3 The Threshold Trade-off

In our credit default example, about 97% of cardholders don't default and only 3% do. Using the standard threshold of 0.5, the model rarely predicts "default" because the probability of default is below 0.5 for most people. We get high accuracy (97%+) but low recall—we miss most actual defaults.

Is this a good model? It depends on what you care about. For a credit card company, missing a default is costly—the company loses the unpaid balance. A false alarm (flagging someone who won't default) is much cheaper—maybe the company offers them a lower credit limit or higher interest rate. Given this asymmetry, the company would rather catch more actual defaults (higher recall) even if it means more false alarms (lower precision).

The key insight is that **the 0.5 threshold isn't sacred**. Logistic regression gives us a probability, and we choose how to convert that probability into a decision. Instead of predicting "default" when $P(\text{default}) > 0.5$, we can use any threshold τ :

$$\text{Predict default if } P(\text{default}) > \tau$$

Lowering the threshold has predictable effects:

- More observations cross the threshold, so we predict "positive" (default) more often
- We catch more actual positives → **higher recall**
- We also flag more negatives by mistake → **more false positives, lower precision**
- Overall accuracy may drop because we're wrong on many negatives

Raising the threshold has the opposite effects:

- Fewer observations cross the threshold, so we predict "positive" less often
- We catch fewer actual positives → **lower recall**
- We have fewer false positives → **higher precision**

There's a fundamental trade-off between precision and recall. A very low threshold (like 0.01) gives nearly perfect recall—we catch almost all defaults—but terrible precision because we flag almost everyone. A very high threshold (like 0.99) gives nearly perfect precision—when we flag someone, they almost certainly will default—but terrible recall because we catch almost nobody.

The "right" threshold depends on the relative costs of false positives and false negatives, which varies by application.

Python

```
# try different thresholds
thresholds = [0.5, 0.2, 0.1, 0.05]
prob_pred = log_reg.predict_proba(X)[: , 1]

print("Effect of Threshold on Confusion Matrix Metrics:\n")
print(f"{'Threshold':>10} {'Accuracy':>10} {'Precision':>10} {'Recall':>10} {'FPR':>10}")
print("-" * 52)

for thresh in thresholds:
    y_pred_thresh = (prob_pred > thresh).astype(int)
    cm = confusion_matrix(default, y_pred_thresh)
    TP, TN, FP, FN = cm[1,1], cm[0,0], cm[0,1], cm[1,0]

    acc = (TP + TN) / (TP + TN + FP + FN)
    prec = TP / (TP + FP) if (TP + FP) > 0 else 0
    rec = TP / (TP + FN)
    fpr = FP / (TN + FP)

    print(f"{'thresh':>10.2f} {'acc':>10.1%} {'prec':>10.1%} {'rec':>10.1%} {'fpr':>10.1%}")
```

Output

Effect of Threshold on Confusion Matrix Metrics:

Threshold	Accuracy	Precision	Recall	FPR
0.50	97.6%	72.6%	41.8%	0.5%
0.20	96.7%	50.2%	66.0%	2.2%
0.10	94.9%	37.0%	78.1%	4.5%
0.05	91.7%	26.7%	86.7%	8.1%

Lowering the threshold from 0.5 to 0.1 dramatically increases recall (catching defaults) at the cost of more false positives.

6.4 The ROC Curve

How do we visualize the precision-recall (or equivalently, TPR-FPR) trade-off across all possible thresholds? The **Receiver Operating Characteristic (ROC) curve** does exactly this.

The ROC curve plots:

- **X-axis:** False Positive Rate (FPR) = $FP / (FP + TN)$
- **Y-axis:** True Positive Rate (TPR) = Recall = $TP / (TP + FN)$

Each point on the curve corresponds to a different threshold. As we vary the threshold from 1 (predict nothing positive) to 0 (predict everything positive), we trace out the curve.

At threshold = 1, we predict everything negative: TPR = 0 (catch no positives) and FPR = 0 (no false positives). This is the bottom-left corner.

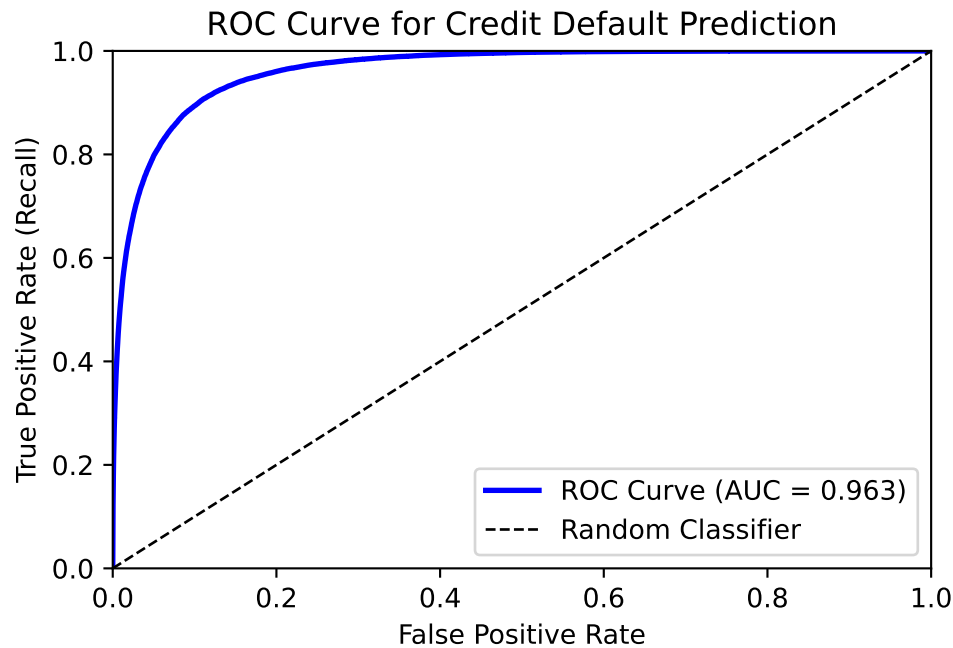
At threshold = 0, we predict everything positive: TPR = 1 (catch all positives) and FPR = 1 (all negatives are false positives). This is the top-right corner.

As we lower the threshold, we move along the curve from bottom-left toward top-right. The shape of this curve tells us about classifier quality:

- **Good classifiers** hug the top-left corner. They achieve high TPR (catch most positives) with low FPR (few false positives). The curve rises steeply at first.

- **Random classifiers** follow the diagonal line from (0,0) to (1,1). A random guess has no discriminative power—increasing TPR requires proportionally increasing FPR.
- **Perfect classifiers** go straight up to (0,1) and then across to (1,1). They can catch all positives (TPR=1) with zero false positives (FPR=0).

The ROC curve separates model quality from threshold choice. Two models might have the same accuracy at threshold 0.5, but very different ROC curves. The one with the better curve has more flexibility—it can achieve higher recall without sacrificing as much precision.



The **Area Under the Curve (AUC)** summarizes the entire ROC curve in a single number. It's literally the area under the ROC curve:

- **AUC = 1.0:** Perfect classifier. The ROC curve goes straight up and then across, enclosing the entire unit square.
- **AUC = 0.5:** Random guessing. The ROC curve follows the diagonal, enclosing exactly half the square.
- **AUC < 0.5:** Worse than random. This usually means the predictions are inverted—swap 0 and 1 and you'd do better.

There's a beautiful probabilistic interpretation: **AUC equals the probability that a randomly chosen positive example has a higher predicted probability than a randomly chosen negative example.** In other words, if we picked one defaulter and one non-defaulter at random, what's the chance our model ranks the defaulter higher?

This interpretation makes AUC intuitive. An AUC of 0.85 means: pick a random defaulter and a random non-defaulter; 85% of the time, the model assigns higher default probability to the defaulter. The model usually ranks correctly.

Python

```
from sklearn.metrics import roc_auc_score

auc = roc_auc_score(default, probab_pred)
print(f"AUC for credit default model: {auc:.3f}")
```

Output

```
AUC for credit default model: 0.963
```


AUC is useful for comparing models because it's **threshold-independent**. Two models with the same AUC have equally good ranking ability, even if their optimal thresholds differ. It answers: “How well does the model separate positives from negatives?” rather than “How accurate is the model at a specific threshold?”

However, AUC has limitations. It treats all false positive rates equally, but in practice you might only care about the low-FPR region (because you can't tolerate many false alarms). A model that's great at low FPR but poor at high FPR might have the same AUC as a model with the opposite pattern. For this reason, some practitioners prefer partial AUC (area under a specific region) or precision-recall curves.

6.5 Choosing the Optimal Threshold

The “best” threshold depends on the costs of different errors. Let c_{FN} be the cost of a false negative (missing a default) and c_{FP} be the cost of a false positive (false alarm). If missing defaults is very costly (e.g., the bank loses the loan amount), we want a lower threshold to maximize recall.

The objective is to minimize total cost:

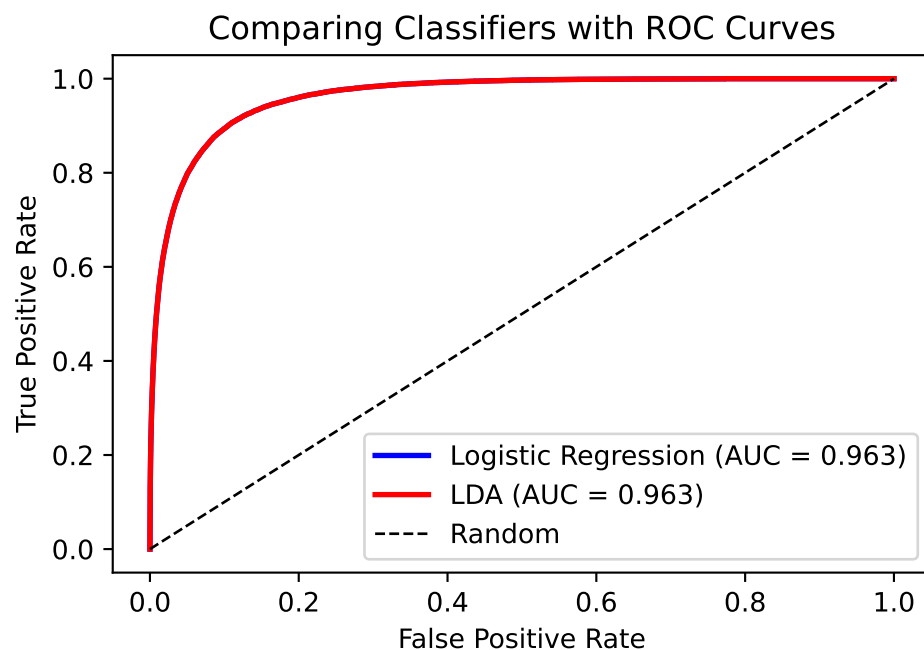
$$\text{Total Cost} = c_{FN} \cdot FN + c_{FP} \cdot FP$$

Or equivalently, maximize:

$$\text{Benefit} = TP - c \cdot FP$$

where $c = c_{FP}/c_{FN}$ is the relative cost ratio.

6.6 Comparing Classifiers



The ROC curves are nearly identical—both methods perform similarly on this dataset. The AUC provides a single number for comparison.

7 Summary

Classification predicts categorical outcomes and is a core supervised learning task. The linear probability model (regression with 0/1 outcome) fails because it can produce impossible probabilities.

Logistic regression uses the sigmoid function to model probabilities properly. Outputs are always in $(0, 1)$, coefficients measure effect on log-odds, and the model can be regularized (Lasso) for variable selection.

Linear Discriminant Analysis takes a Bayesian approach. It models class-conditional distributions as normal and assumes shared covariance (LDA) or class-specific covariance (QDA). LDA often performs similarly to logistic regression in practice.

Accuracy can be misleading with imbalanced classes. The confusion matrix breaks down predictions into TP, FP, TN, FN. Precision (of predicted positives, how many are correct?) and Recall (of actual positives, how many did we catch?) capture different aspects of performance. The ROC curve shows the trade-off across all thresholds, and AUC summarizes discriminative ability in a single number. The optimal threshold depends on the costs of different types of errors.

8 References

- Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2), 179-188.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning* (2nd ed.). Springer. Chapter 4.
- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An Introduction to Statistical Learning* (2nd ed.). Springer. Chapter 4.