RSM338: Machine Learning in Finance

Lecture 7: Classification | March 4–5, 2026

Kevin Mott
Rotman School of Management

Rotman Commerce

Today's Goal

this lecture we move from predicting continuous outcomes (regression) to predicting **categorical outcomes** (classification).

Classification in finance: credit default, fraud detection, market direction, bankruptcy prediction, sector assignment.

Today's roadmap:

1. **Logistic regression:** The workhorse linear classifier
2. **Decision boundaries:** Linear and nonlinear
3. **k-Nearest Neighbors:** A flexible, nonparametric classifier
4. **Decision trees:** Rule-based classification
5. **Evaluation metrics:** Confusion matrix, ROC curves, AUC

The Binary Classification Setup

We have n observations, each with features $\mathbf{x}_i$ and a binary class label $y_i \in \{0, 1\}$.

Our goal: learn a function that predicts y for new observations.

Two approaches:

1. **Predict the class directly:** Output 0 or 1
2. **Predict the probability:** Estimate $P(y = 1 \mid \mathbf{x})$, then threshold

The second approach is more flexible—it tells us *how confident* we are, not just the prediction.

Part I: Why Not Just Use Regression?

The Linear Probability Model

A natural first idea: encode $y \in \{0, 1\}$ and fit linear regression. This is the **linear probability model**.

We'll use credit card default data throughout this lecture: 1 million individuals with balance, income, and default status.

```python
import numpy as np
import pandas as pd

# Load credit default data
data = pd.read_csv('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%}")
```

```
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)
```

Rotman Commerce

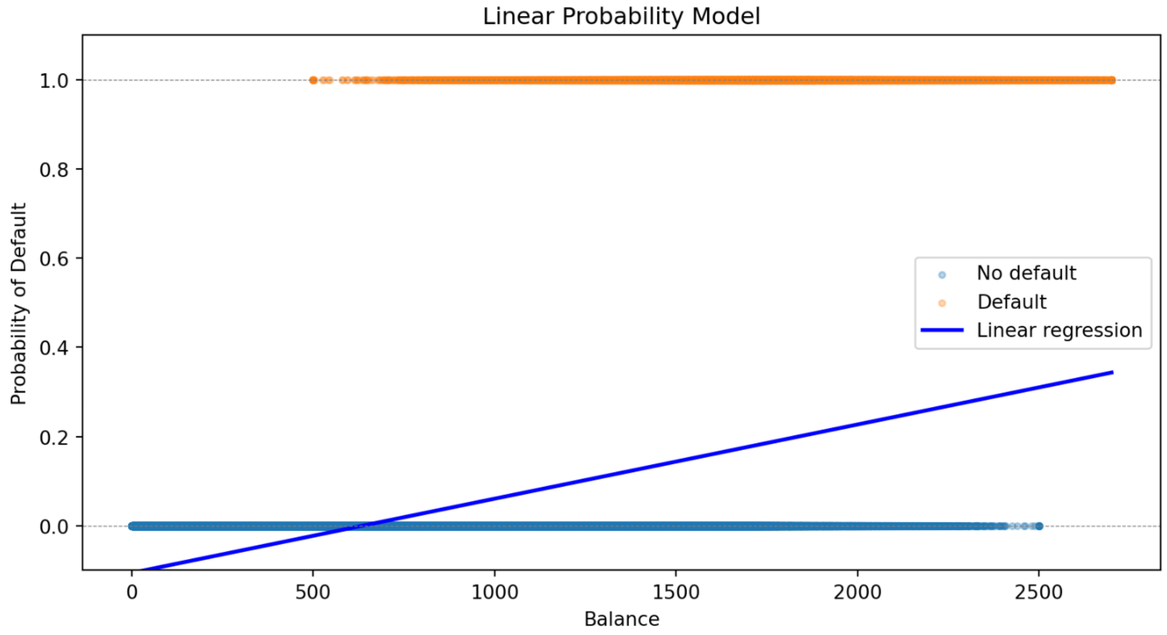
Default rate: 3.3%

```
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)
```



The Problem: Predictions Outside [0, 1]

The linear probability model can produce “probabilities” that aren’t valid probabilities.

```
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()}")
```

```
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? These are nonsensical.

We need a function that **naturally** maps any input to the interval (0, 1). This is exactly what logistic regression does.

Part II: Logistic Regression

The Logistic Function

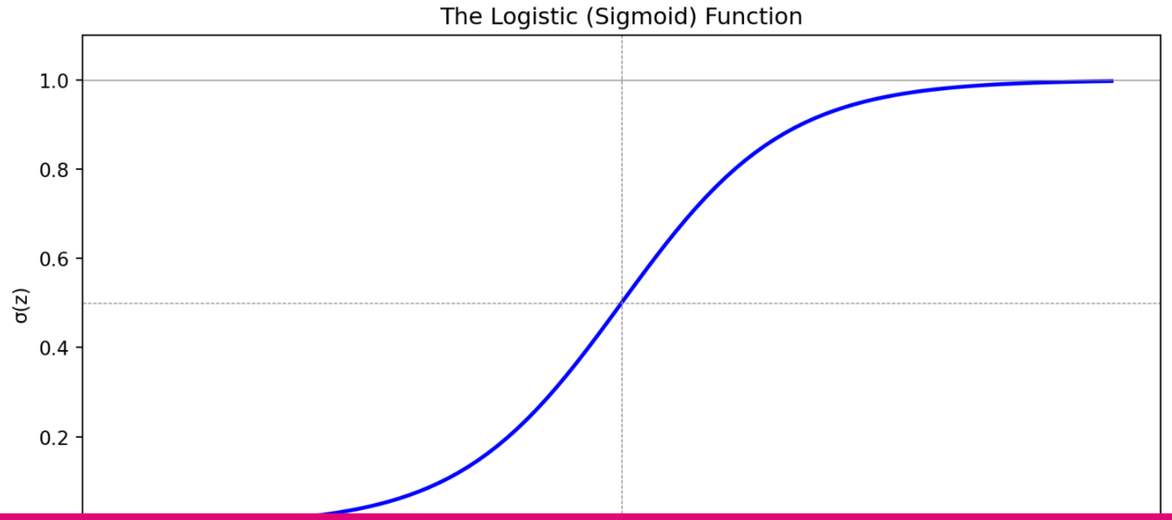
Instead of modeling probability as a linear function, we use the **logistic function** (also called the sigmoid function):

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

This S-shaped function has two properties:

1. **Output is always between 0 and 1:** $\lim_{z \to -\infty} \sigma(z) = 0; \lim_{z \to +\infty} \sigma(z) = 1$

2. **Monotonic:** Larger z always means larger probability



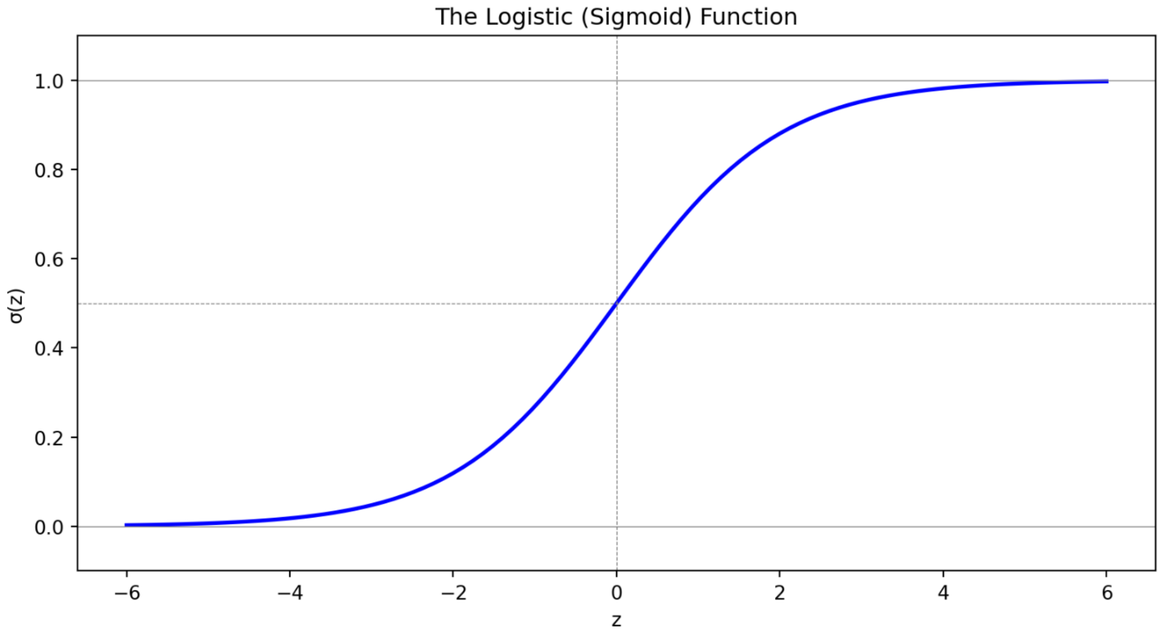
Instead of modeling probability as a linear function, we use the **logistic function** (also called the sigmoid function):

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

This S-shaped function has two properties:

1. **Output is always between 0 and 1:** $\lim_{z \to -\infty} \sigma(z) = 0; \lim_{z \to +\infty} \sigma(z) = 1$

2. **Monotonic:** Larger z always means larger probability



The Logistic Regression Model

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

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

The notation $P(y = 1 \mid \mathbf{x})$ 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.

Here $\mathbf{x} = (x_1, x_2, \ldots, x_p)'$ is a p-vector of features (attributes) for an observation, and $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_p)'$ is the corresponding p-vector of coefficients we need to learn.

The Linear Predictor

Let's define $z = \beta_0 + \boldsymbol{\beta}'\mathbf{x}$ as the **linear predictor**. Writing out the dot product:

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

Then the model becomes:

$$P(y = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-z}}$$

The linear combination z can be any real number, but the logistic function squashes it to (0, 1).

- If $z = 0$: $P(y = 1) = 0.5$ (coin flip)
- If $z > 0$: $P(y = 1) > 0.5$ (more likely positive)
- If $z < 0$: $P(y = 1) < 0.5$ (more likely negative)

What Are Odds?

You've seen odds in sports betting: "the Leafs are 3-to-1 to win" means for every 1 time they win, they lose 3 times.

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

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 *not* to happen
90%	9:1	Very likely

Odds range from 0 to ∞, with 1 being the "neutral" point (50-50). This asymmetry is awkward—log-odds fixes it.

The Log-Odds (Logit) Interpretation

Taking the log of odds gives us **log-odds** (also called the **logit**):

- Log-odds of 0 means 50-50 (odds = 1)
- Positive log-odds means more likely than not
- Negative log-odds means less likely than not

In logistic regression, the log-odds is linear in the features:

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

The coefficient β_j tells us how a one-unit increase in x_j affects the log-odds:

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

We can also interpret coefficients as **odds ratios**: e^{β_j} is the multiplicative change in odds for a one-unit increase in x_j. If

Taking the log of odds gives us **log-odds** (also called the **logit**):

- Log-odds of 0 means 50-50 (odds = 1)
- Positive log-odds means more likely than not
- Negative log-odds means less likely than not

In logistic regression, the log-odds is linear in the features:

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

The coefficient β_j tells us how a one-unit increase in x_j affects the log-odds:

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

We can also interpret coefficients as **odds ratios**: e^{β_j} is the multiplicative change in odds for a one-unit increase in x_j. If $\beta_j = 0.5$, then $e^{0.5} \approx 1.65$: each one-unit increase multiplies the odds by 1.65 (a 65% increase).

Fitting Logistic Regression: The Loss Function

How do we find the best coefficients β_0, $\boldsymbol{\beta}$? Like any ML model, we define a **loss function** and minimize it.

For observation i with features $\mathbf{x}_i$ and label y_i, let $\hat{p_i} = P(y_i = 1 \mid \mathbf{x}_i)$ be our predicted probability. Intuitively:

- If $y_i = 1$: we want $\hat{p_i}$ close to 1 (predict high probability for actual positives)
- If $y_i = 0$: we want $\hat{p_i}$ close to 0 (predict low probability for actual negatives)

The **binary cross-entropy loss** (also called log loss) captures this:

$$\square(\boldsymbol{\beta}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \ln(\hat{p_i}) + (1 - y_i)\ln(1 - \hat{p_i})\right]$$

When $y_i = 1$, the loss is $-\ln(\hat{p_i})$, which is small when $\hat{p_i}$ is close to 1. When $y_i = 0$, the loss is $-\ln(1 - \hat{p_i})$, which is small when $\hat{p_i}$ is close to 0.

Comparing Loss Functions: Logistic vs. Linear Regression

Both linear and logistic regression fit into the same ML framework: **define a loss function, then minimize it**.

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

The choice of loss function depends on the problem: squared error makes sense for continuous outcomes, cross-entropy makes sense for probabilities.

Connection to statistics

In statistics, minimizing cross-entropy loss is equivalent to **maximum likelihood estimation**. 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.

Rotman Commerce

Logistic Regression for Credit Default

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

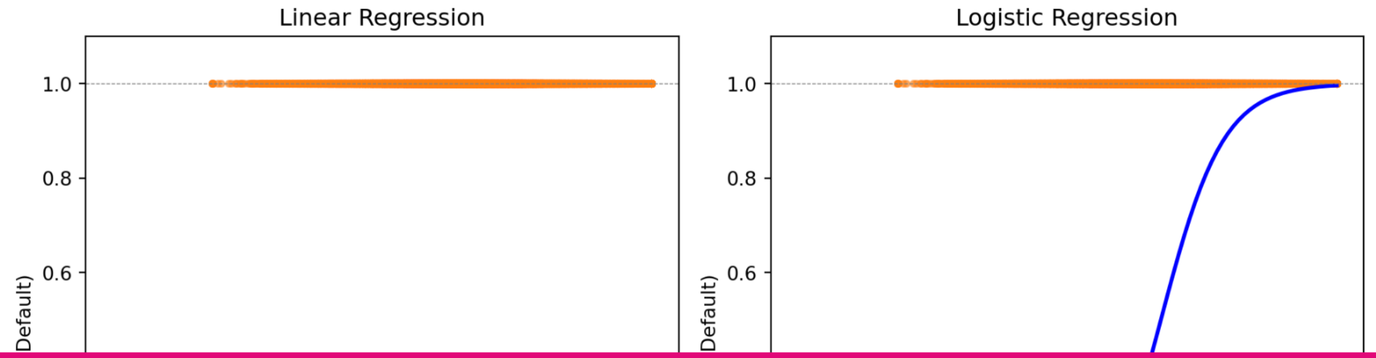
```python
from sklearn.linear_model import LogisticRegression

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

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

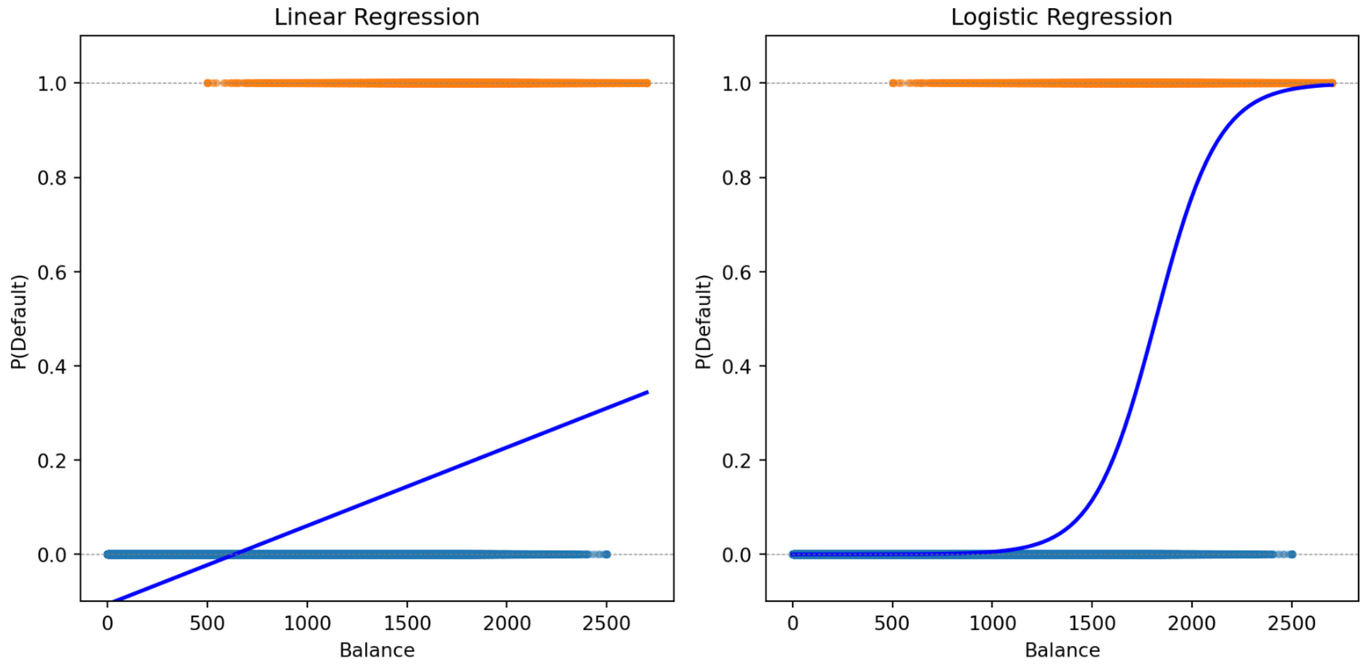
print(f"Logistic Regression:")
print(f"  Intercept: {log_reg.intercept_[0]:.4f}")
print(f"  Coefficient on balance: {log_reg.coef_[0, 0]:.6f}")
```

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



```
6
7  # Predictions
8  prob_logistic = log_reg.predict_proba(balance_grid)[:, 1]
9
10 print(f"Logistic Regression:")
11 print(f"  Intercept: {log_reg.intercept_[0]:.4f}")
12 print(f"  Coefficient on balance: {log_reg.coef_[0, 0]:.6f}")
```

```
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.

Making Predictions

Logistic regression gives us a probability. To make a classification decision, we need a **threshold** (also called a cutoff).

The default rule: predict class 1 if $P(y = 1 \mid \mathbf{x}) > 0.5$

```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%}")
```

```
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.

Multiple Features

Logistic regression easily extends to multiple predictors:

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

```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}")
```

```
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.

Multi-Class Logistic Regression

When we have $K > 2$ classes, we can extend logistic regression using the **softmax function**.

For each class k, we define a linear predictor:

$$z_k = \beta_{k,0} + \boldsymbol{\beta}_k' \mathbf{x}$$

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:

- Each probability is between 0 and 1
- The probabilities sum to 1 across all classes

Regularized Logistic Regression

Just like linear regression, logistic regression can overfit—especially with many features.

We can add regularization. **Lasso logistic regression** maximizes:

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

The penalty $\lambda \sum |\beta_j|$ shrinks coefficients toward zero and can set some exactly to zero (variable selection).

Benefits:

- Prevents overfitting when p is large relative to n
- Identifies which features matter most
- Improves out-of-sample prediction

The regularization parameter λ is chosen by cross-validation.

```
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)
```

$$j=1$$

The penalty $\lambda \sum |\beta_j|$ shrinks coefficients toward zero and can set some exactly to zero (variable selection).

Benefits:

- Prevents overfitting when p is large relative to n
- Identifies which features matter most
- 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 (λ chosen by CV):")
print(f"  Best C (inverse of λ): {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}")
```

```
Lasso Logistic Regression (λ chosen by CV):
  Best C (inverse of λ): 0.0001
  Coefficient on balance: 0.001064
  Coefficient on income: -0.000123556
```

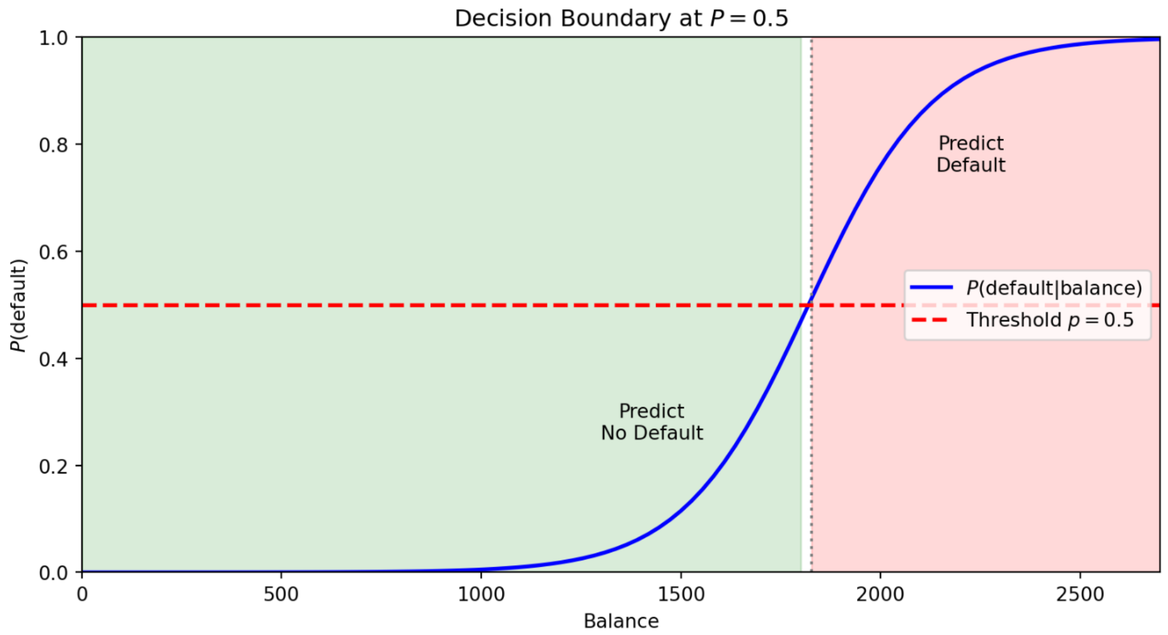
Part III: Decision Boundaries

The 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 $P(y = 1 \mid \mathbf{x}) > 0.5$.

The boundary is where $P(y = 1 \mid \mathbf{x}) = 0.5$ exactly—the point of maximum uncertainty.



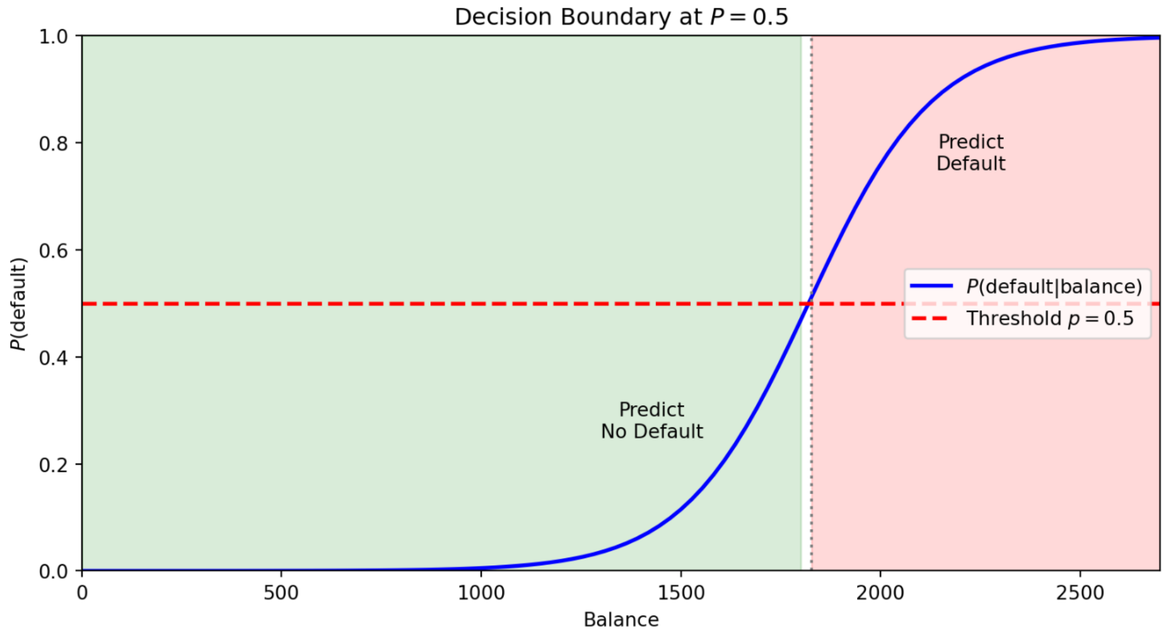
The horizontal red line marks $P = 0.5$. Where the probability curve crosses this threshold defines the **decision boundary** in feature space—observations with balance above this point are predicted to default.

Rotman Commerce

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 $P(y = 1 \mid \mathbf{x}) > 0.5$.

The boundary is where $P(y = 1 \mid \mathbf{x}) = 0.5$ 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 $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots = 0$—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).

What If Classes Aren't Linearly Separable?

Sometimes a straight line can't separate the classes well. If we know the structure of the problem, we can create **curved boundaries** by engineering the right features:

- Include $x_1^2, x_2^2, x_1 x_2$ 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

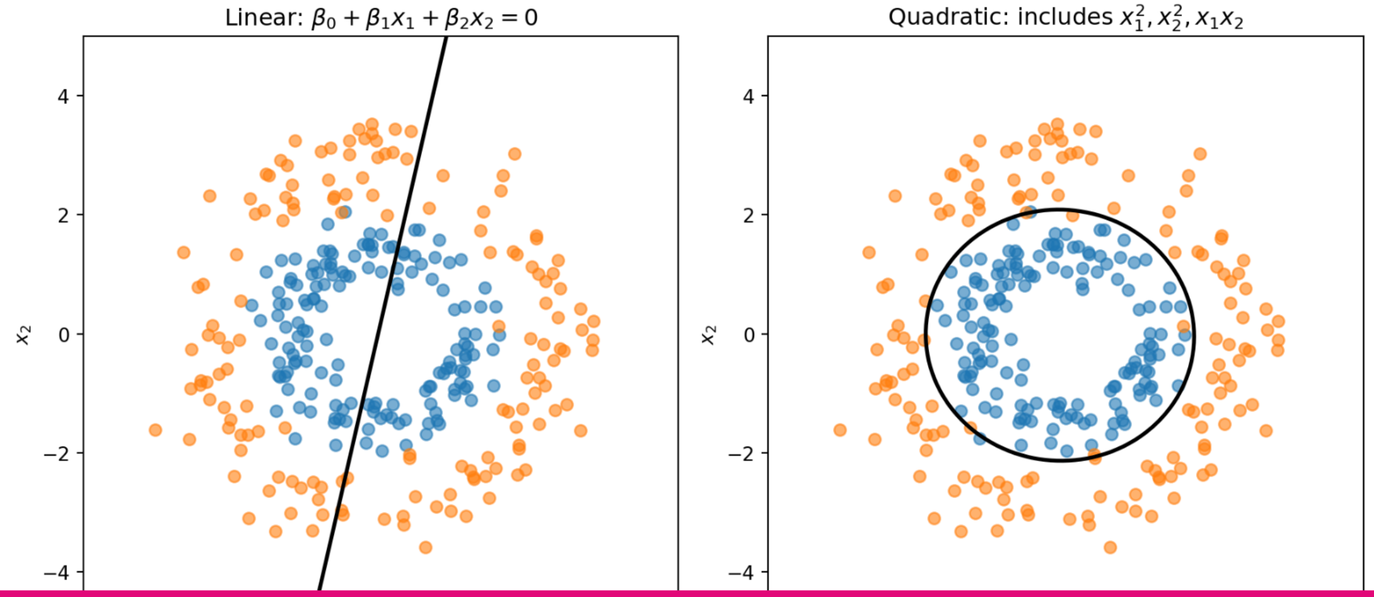
Linear vs. Quadratic Boundaries

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:

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

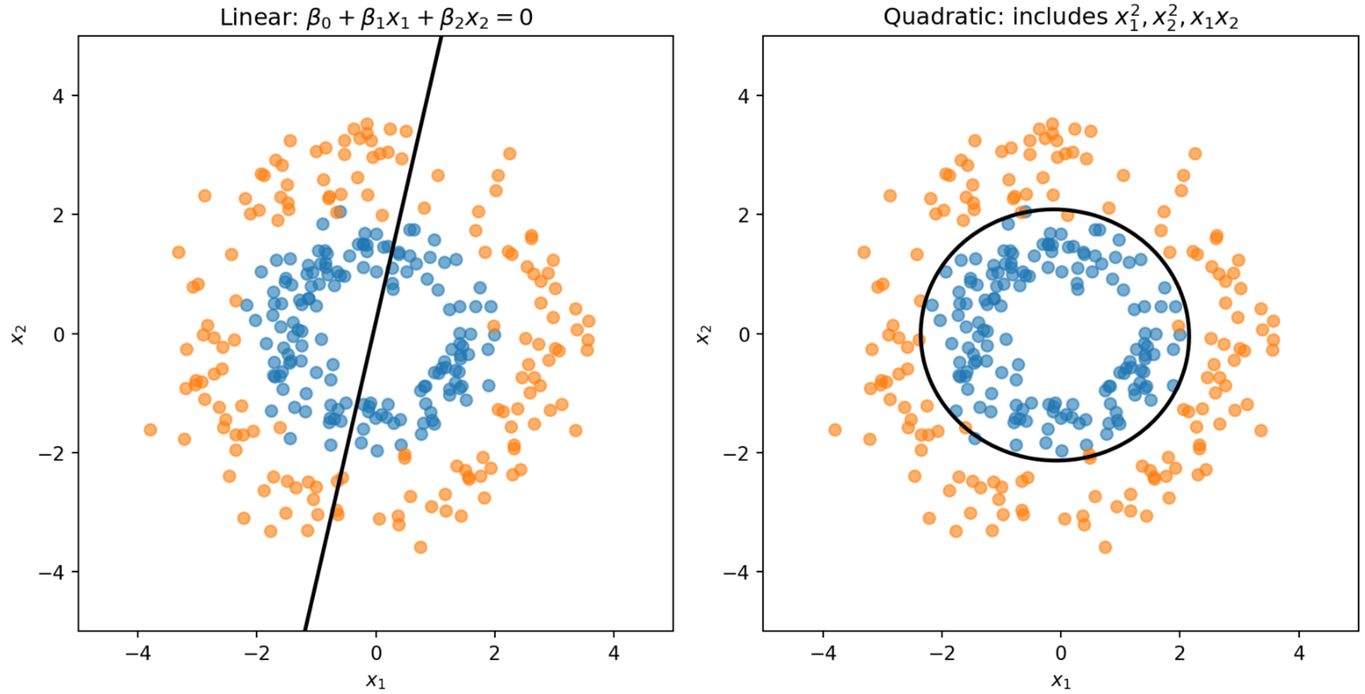
This is a **quadratic equation** in x_1 and x_2—it can represent circles, ellipses, or other curved shapes.



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.



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.

Another Approach: Classification as Supervised Clustering

Logistic regression directly models $P(y = 1 \mid \mathbf{x})$. **Linear Discriminant Analysis (LDA)** takes the opposite approach — it models what each class looks like and uses Bayes' theorem to classify.

The idea is the same as clustering (Lecture 4), except now we **know the labels**:

1. Assume each class k follows a multivariate normal: $\mathbf{x} \mid y = k \sim \square(\boldsymbol{\mu}_k, \boldsymbol{\Sigma})$
2. Estimate the class means $\boldsymbol{\mu}_k$ and a **shared** covariance $\boldsymbol{\Sigma}$ from training data
3. For a new observation, ask: which class's distribution is it most likely to have come from?

Bayes' theorem turns this into a scoring rule. The **discriminant function** for class k is:

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

where $\pi_k = P(y = k)$ is the prior (just the fraction of training data in class k). Classify to whichever class has the highest δ_k. Because δ_k is linear in $\mathbf{x}$, the decision boundary is a line — just like logistic regression.

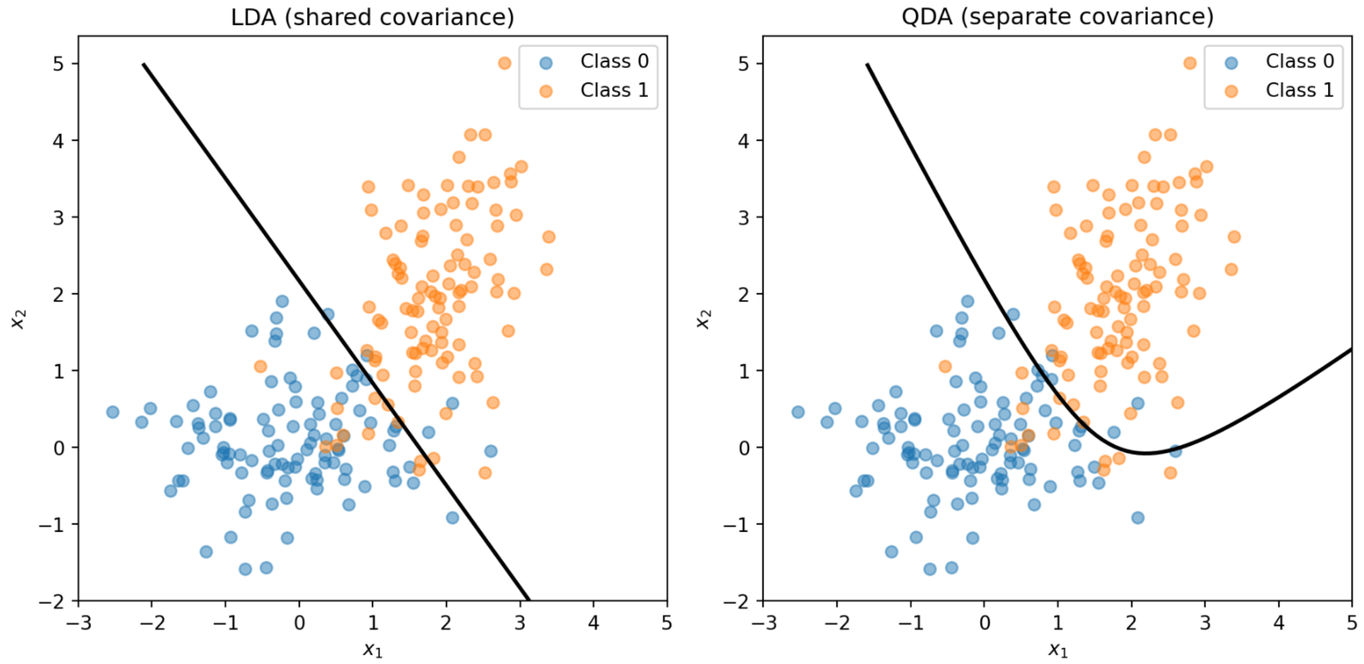
No optimization loop, no gradient descent — LDA computes its parameters in closed form. See the appendix for the full derivation.

Rotman Commerce

LDA vs. QDA: Shared or Separate Covariance?

LDA assumes all classes share the same covariance $\boldsymbol{\Sigma}$ — same shape, different centres. **Quadratic Discriminant Analysis (QDA)** relaxes this: each class gets its own $\boldsymbol{\Sigma}_k$.

- **Shared covariance → linear boundary** (LDA)
- **Separate covariance → quadratic boundary** (QDA)



The trade-off: LDA has fewer parameters (more stable, less overfitting); QDA is more flexible (captures curved

The Feature Engineering Problem

The ring example worked because we knew the right transformation: add x_1^2 and x_2^2. But that approach has a big limitation: **we have to know which transformations to use**.

With 2 features, adding squares and interactions is easy. With 50 features? There are 1,275 pairwise interactions and 50 squared terms—and we have no guarantee that quadratic terms are the right choice. Maybe the boundary depends on $\log(x_3)$, or x_7/x_{12}, or something we'd never think to try.

We want methods that can learn nonlinear boundaries **directly from the data**, without us having to guess the right feature transformations in advance.

Parametric vs. Nonparametric Models

Parametric models (like logistic regression) assume the data follows a specific functional form. We estimate a fixed set of parameters ($\beta_0, \beta_1, \ldots, \beta_p$), and these parameters define the model completely.

Nonparametric models make fewer assumptions about the functional form. Instead, they let the data determine the structure of the decision boundary.

	Parametric	Nonparametric
Structure	Fixed form (e.g., linear)	Flexible, data-driven
Parameters	Fixed number	Grows with data
Examples	Logistic regression, LDA	k-NN, Decision Trees
Risk	Bias if form is wrong	Overfitting with limited data

Both k-NN and decision trees are nonparametric—they don't assume a linear (or any particular) decision boundary.

Part IV: k-Nearest Neighbors

The Intuition Behind k-NN

k-Nearest Neighbors (k-NN) is based on a simple idea: similar observations should have similar outcomes.

To classify a new observation:

1. Find the k training observations closest to it
2. Take a vote among those k neighbors
3. Assign the majority class

If you want to know if a new loan applicant will default, look at applicants in the training data who are most similar to them. If most of those similar applicants defaulted, predict default.

No training phase is needed—k-NN stores all the training data and does the work at prediction time. This is sometimes called a “lazy learner.”

Distance Recap (Lecture 4)

k-NN needs to measure how far apart two observations are. Same idea as clustering:

$$d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\| = \sqrt{\sum_{k=1}^{p} (x_{ik} - x_{jk})^2}$$

Two reminders from Lecture 4:

1. **Standardize first.** Features on different scales (income in dollars vs. DTI as a ratio) will make distance meaningless. Standardize each feature to mean 0, standard deviation 1.
2. **Distance = norm of a difference.** The L_2 (Euclidean) norm is the default. Manhattan (L_1) is an alternative but Euclidean works well for most applications.

The k-NN Algorithm

Input: Training data $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, a new point $\mathbf{x}$, and the number of neighbors k.

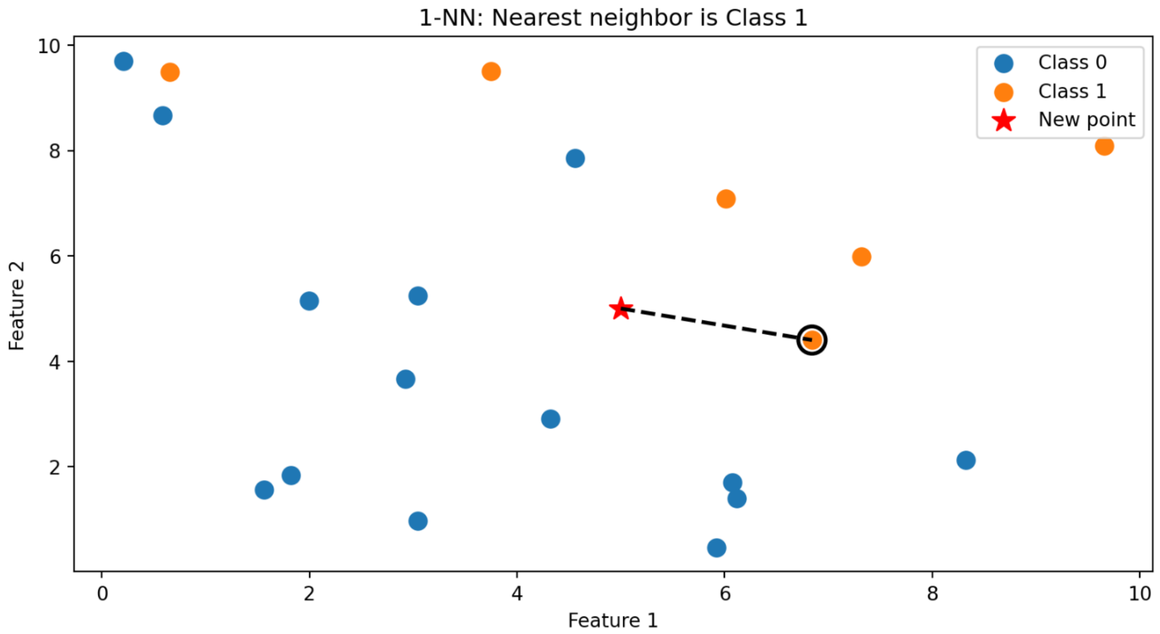
Algorithm:

1. Compute the distance from $\mathbf{x}$ to every training observation $\mathbf{x}_i$
2. Identify the k training observations with the smallest distances—call this set $\square_k(\mathbf{x})$
3. Assign the class that appears most frequently among the k neighbors:

$$\hat{y} = \arg\max_c \sum_{i \in \square_k(\mathbf{x})} \mathbb{1}_{\{y_i = c\}}$$

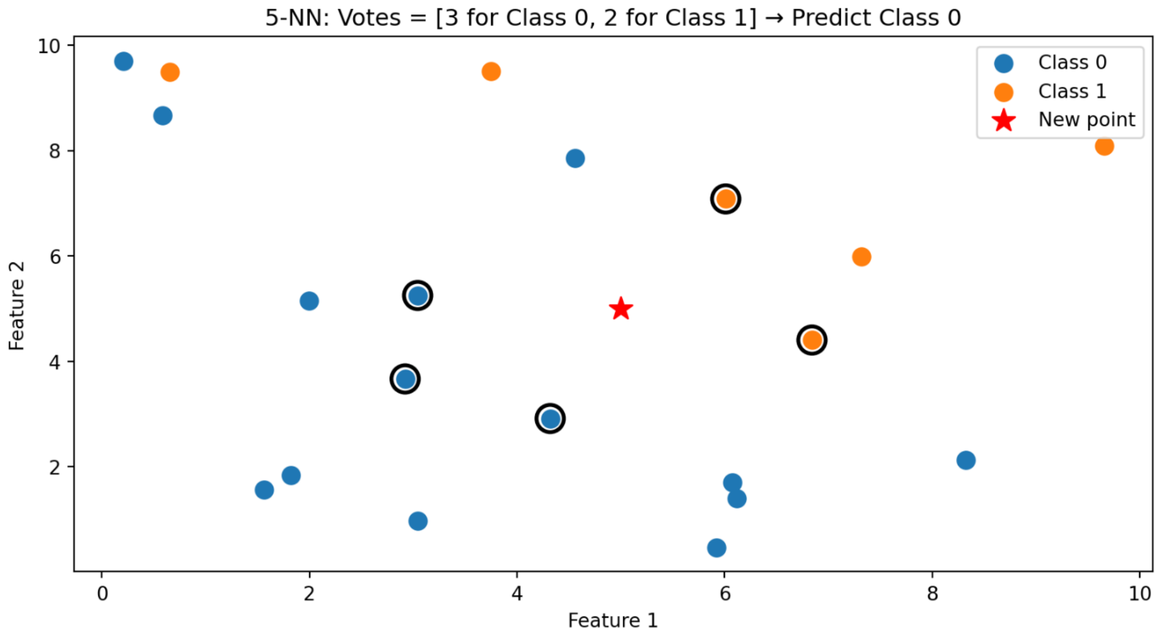
The notation $\mathbb{1}_{\{y_i = c\}}$ is the **indicator function**: it equals 1 if $y_i = c$ and 0 otherwise. So we're just counting votes.

k-NN in Action: k = 1



With $k = 1$, we classify based on the single closest training point. The new point (star) is assigned the class of its nearest neighbor (circled).

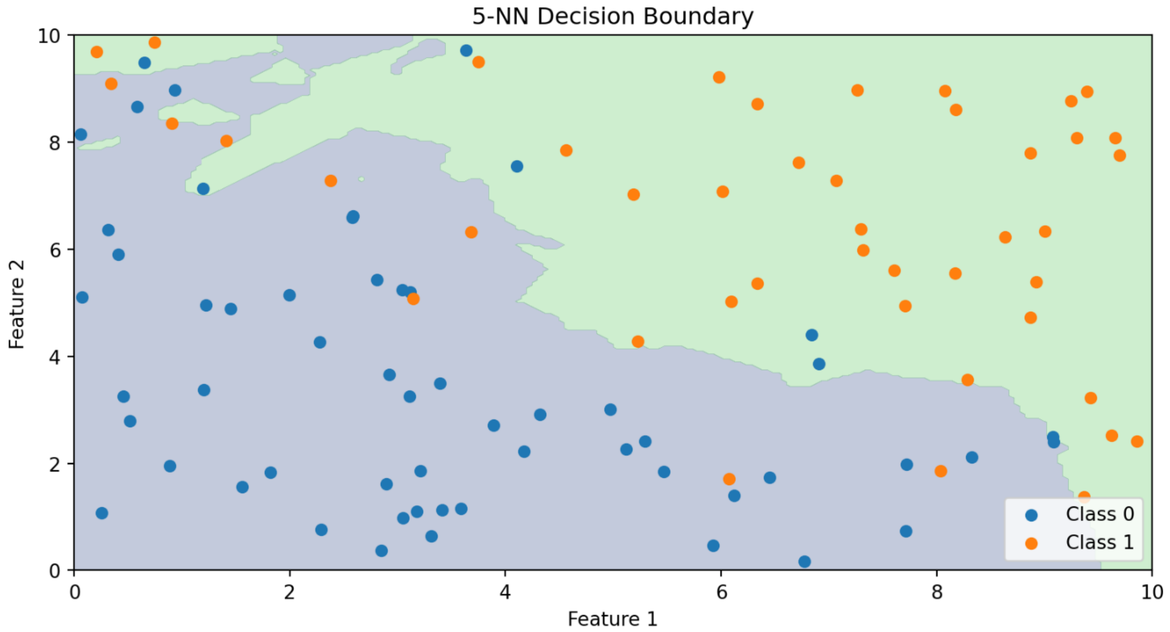
k-NN in Action: k = 5



With $k = 5$, we take a majority vote among the 5 nearest neighbors (circled). This is more robust than using just one neighbor.

The Decision Boundary of k-NN

Unlike linear classifiers, k-NN doesn't explicitly compute a decision boundary. But we can visualize what the boundary looks like by classifying every point in the feature space.



The k-NN decision boundary is **nonlinear** and **adapts to the local density of data**. It naturally forms complex shapes without us specifying any functional form.

The Role of k: Bias-Variance Tradeoff

The choice of k is crucial:

Small k (e.g., k = 1):

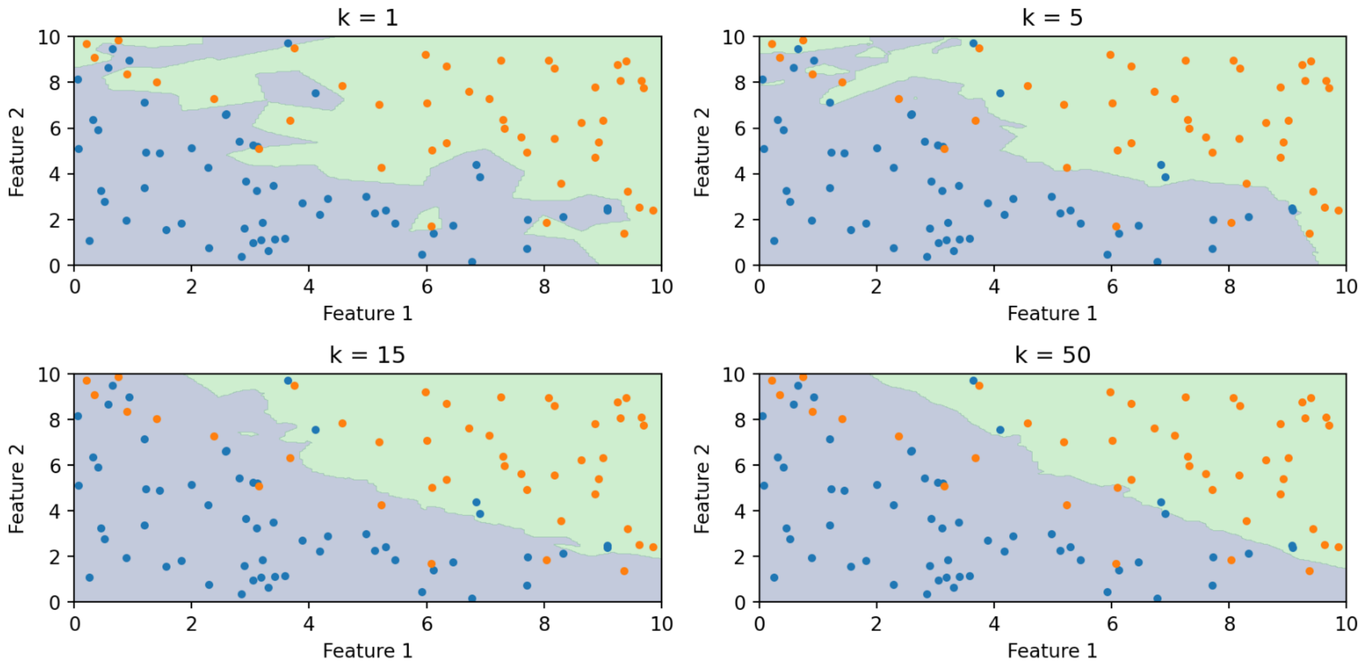
- Boundary closely follows the training data
- Very flexible—can capture complex patterns
- High variance, low bias
- Risk of **overfitting** (sensitive to noise)

Large k (e.g., k = 100):

- Boundary is smoother
- Less flexible—averages over many neighbors
- Low variance, high bias
- Risk of **underfitting** (misses local patterns)

This is the **bias-variance tradeoff** we've seen before. We need to choose k that balances these concerns.

Effect of k on the Decision Boundary



As k increases, the boundary becomes smoother. With $k = 1$, every training point gets its own region. With large k, the boundary approaches the overall majority class.

Choosing k with Cross-Validation

How do we choose k? Use **cross-validation** (from Lecture 5):

1. Split training data into folds
2. For each candidate value of k:
   - Fit k-NN on training folds
   - Evaluate accuracy on validation fold
3. Choose k that maximizes cross-validated accuracy

A common rule of thumb: $k < \sqrt{n}$ where n is the sample size. But cross-validation is more reliable.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
import numpy as np

# Try different values of k
k_values = range(1, 31)
cv_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=5)
    cv_scores.append(scores.mean())
```

Rotman Commerce

1. Split training data into folds
2. For each candidate value of k:
   - Fit k-NN on training folds
   - Evaluate accuracy on validation fold
3. Choose k that maximizes cross-validated accuracy

A common rule of thumb: $k < \sqrt{n}$ where n is the sample size. But cross-validation is more reliable.

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
import numpy as np

# Try different values of k
k_values = range(1, 31)
cv_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=5)
    cv_scores.append(scores.mean())

best_k = k_values[np.argmax(cv_scores)]
print(f"Best k: {best_k} with CV accuracy: {max(cv_scores):.3f}")
```

```
Best k: 19 with CV accuracy: 0.850
```

k-NN: Advantages and Disadvantages

Advantages:

- Simple to understand and implement
- No training phase (just store the data)
- Naturally handles multi-class problems
- Can capture complex, nonlinear boundaries
- No assumptions about the data distribution

Disadvantages:

- Slow at prediction time—must compute distances to all training points
- Doesn’t work well in high dimensions (“curse of dimensionality”)
- Sensitive to irrelevant features (all features contribute to distance)
- Requires feature scaling

For large datasets, approximate nearest neighbor methods can speed up k-NN, but it remains computationally intensive.

Rotman Commerce

The Curse of Dimensionality

k-NN relies on distance, and distance breaks down in high dimensions. Three related problems:

1. **The space becomes sparse.** In 1D, 100 points cover the range well. In 2D, the same 100 points are scattered across a plane. In 50D, they're lost in a vast empty space. The amount of data you need to "fill" the space grows exponentially with p.
2. **You need more data to have local neighbours.** If the space is mostly empty, the k "nearest" neighbours may be far away — and far-away neighbours aren't informative about the local structure.
3. **Distances become less informative.** Euclidean distance sums p squared differences. As p grows, all these sums converge to roughly the same value (law of large numbers). The nearest and farthest neighbours end up almost the same distance away, so "nearest" stops meaning much.

Part V: Decision Trees

The Intuition Behind Decision Trees

Decision trees mimic how humans make decisions: a series of yes/no questions.

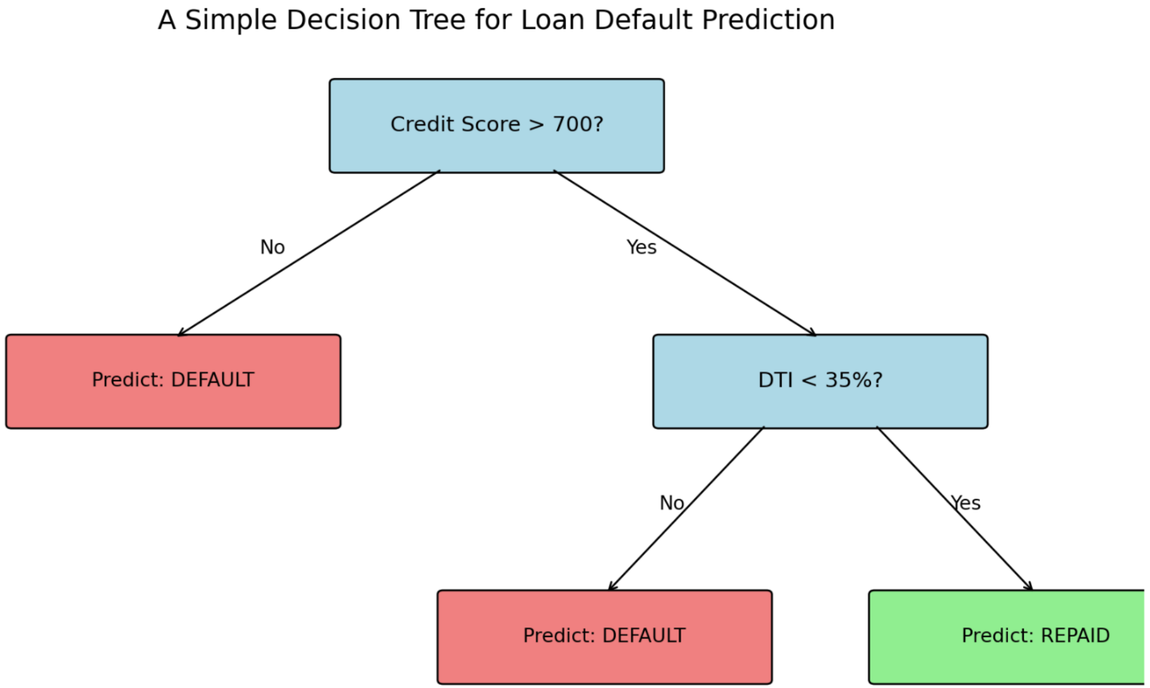
Consider a loan officer evaluating an application:

1. Is the credit score above 700?
   - If no → High risk, deny
   - If yes → Continue…
2. Is the debt-to-income ratio below 35%?
   - If no → Medium risk, deny
   - If yes → Low risk, approve

Each question splits the population into subgroups, and we make predictions based on which group an observation falls into.

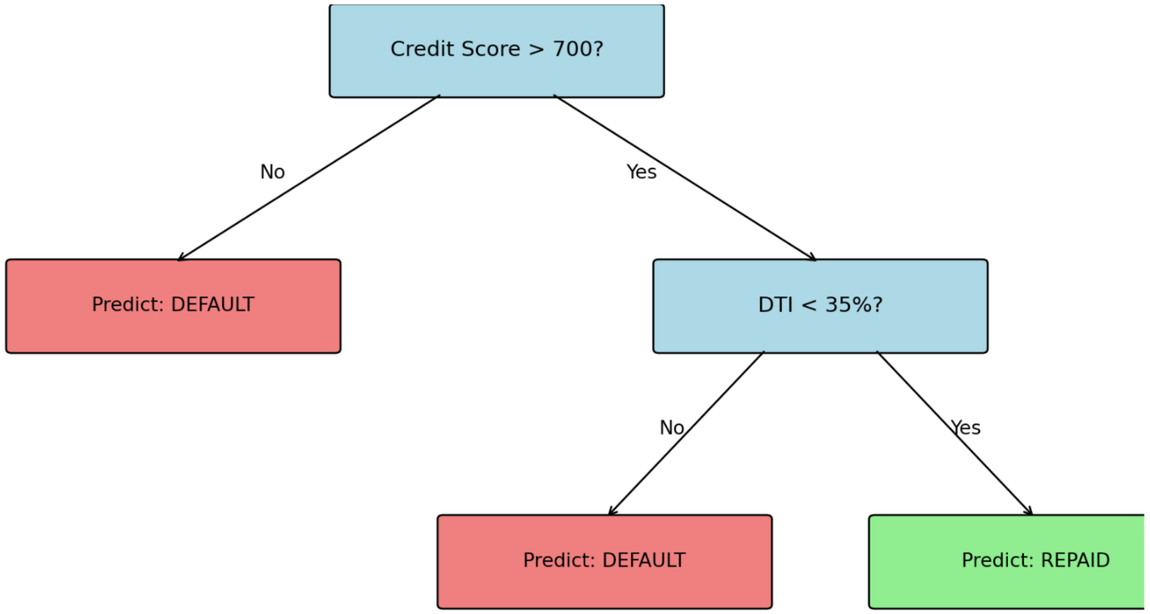
Decision trees automate this process: they learn which questions to ask and in what order.

Anatomy of a Decision Tree



Terminology:

- **Root node:** The first split (top of the tree)

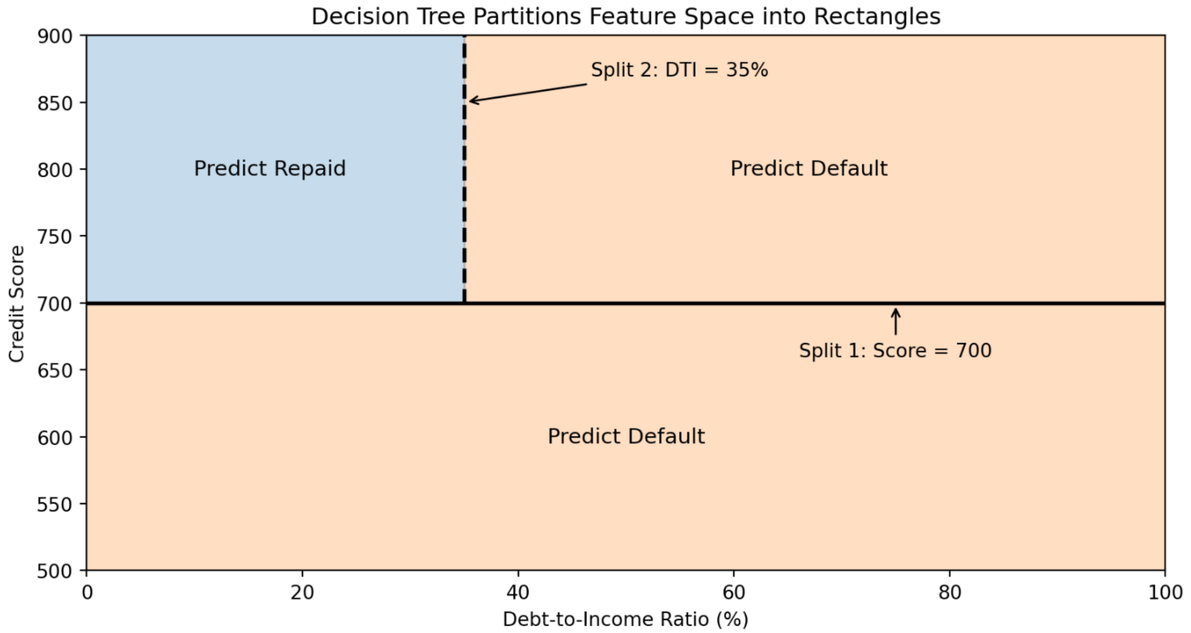


Terminology:

- **Root node:** The first split (top of the tree)
- **Internal nodes:** Decision points that split the data
- **Leaf nodes:** Terminal nodes that make predictions
- **Depth:** The number of splits from root to leaf

How Trees Partition the Feature Space

Each split in a decision tree divides the feature space with an **axis-aligned** boundary (parallel to one axis).



The tree creates **rectangular regions**. Each leaf corresponds to one region, and all observations in that region get the same prediction.

The Decision Tree Algorithm: Recursive Partitioning

The Goal: Build a tree that makes good predictions.

The Approach: Greedy, recursive partitioning.

1. Start with all training data at the root
2. Find the best split—the feature and threshold that best separates the classes
3. Split the data into two groups based on this rule
4. Recursively apply steps 2-3 to each group
5. Stop when a stopping criterion is met (e.g., minimum samples per leaf, maximum depth)

The key question: **How do we define “best” split?**

Measuring Split Quality: Impurity

A good split should create child nodes that are more “pure” than the parent—ideally, each child contains only one class.

We measure **impurity**—how mixed the classes are in a node. A pure node (all one class) has impurity = 0.

For a node with n observations where p_c is the proportion belonging to class c:

Gini impurity:

$$\text{Gini} = 1 - \sum_c p_c^2$$

Entropy:

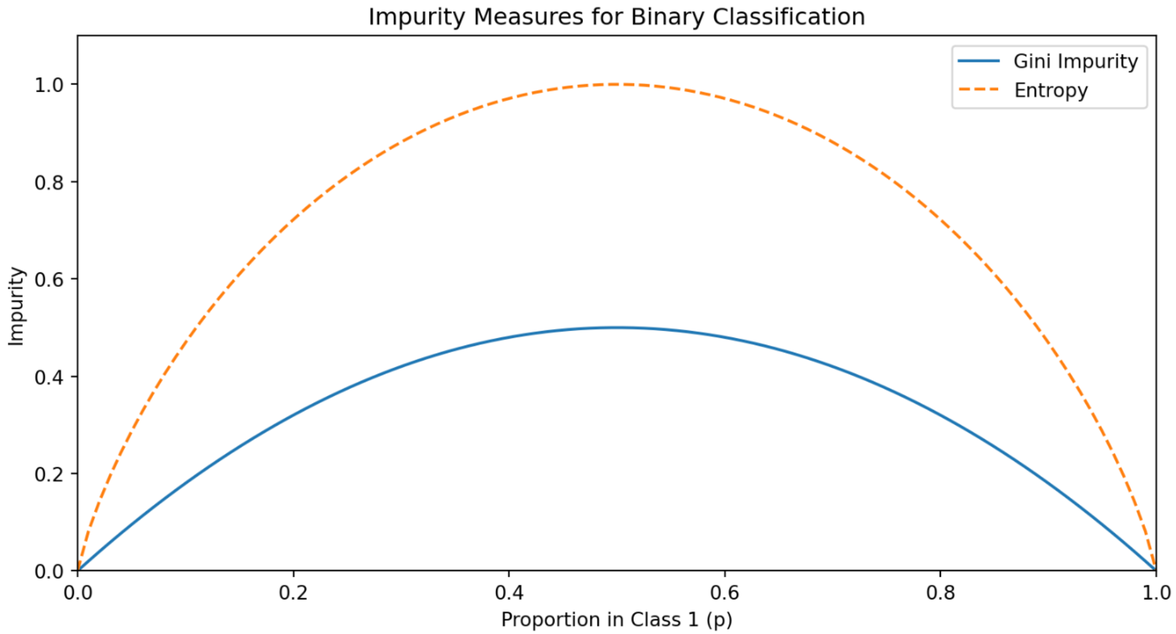
$$\text{Entropy} = -\sum_c p_c \log_2(p_c)$$

Both measures equal 0 for a pure node and are maximized when classes are equally mixed.

Understanding Gini Impurity

For binary classification with p being the proportion in class 1:

$$\text{Gini} = 1 - p^2 - (1-p)^2 = 2p(1-p)$$



Both measures are minimized (= 0) when $p = 0$ or $p = 1$ (pure node) and maximized when $p = 0.5$ (maximum uncertainty).

Computing Information Gain

Information gain measures how much a split reduces impurity.

If a parent node P is split into children L (left) and R (right):

$$\text{Information Gain} = \text{Impurity}(P) - \left[\frac{n_L}{n_P} \cdot \text{Impurity}(L) + \frac{n_R}{n_P} \cdot \text{Impurity}(R)\right]$$

where n_P , n_L , n_R are the number of observations in the parent, left child, and right child.

The weighted average accounts for the sizes of the child nodes. The best split is the one that **maximizes information gain**.

Example: Computing Information Gain

Suppose we have 100 loan applicants: 60 repaid, 40 defaulted.

Parent impurity (Gini):

$$\text{Gini}_P = 1 - (0.6)^2 - (0.4)^2 = 1 - 0.36 - 0.16 = 0.48$$

Option A: Split on Credit Score > 700

- Left (below 700): 30 observations (10 repaid, 20 default) → Gini = $1 - (1/3)^2 - (2/3)^2 = 0.444$
- Right (above 700): 70 observations (50 repaid, 20 default) → Gini = $1 - (5/7)^2 - (2/7)^2 = 0.408$

$$\text{Gain}_A = 0.48 - \left[\frac{30}{100}(0.444) + \frac{70}{100}(0.408)\right] = 0.48 - 0.419 = 0.061$$

We'd compute this for all possible features and thresholds, then choose the split with highest gain.

For Continuous Features: Finding the Best Threshold

For a continuous feature (like credit score), we need to find the best **threshold** for splitting.

Algorithm:

1. Sort the observations by the feature value
2. Consider each unique value as a potential threshold
3. For each threshold, compute the information gain
4. Choose the threshold with the highest gain

If there are n unique values, we evaluate up to $n - 1$ possible splits for that feature. This is computationally tractable because we can update class counts incrementally as we move through sorted values.

Building a Tree in Python

```python
from sklearn.tree import DecisionTreeClassifier
import numpy as np

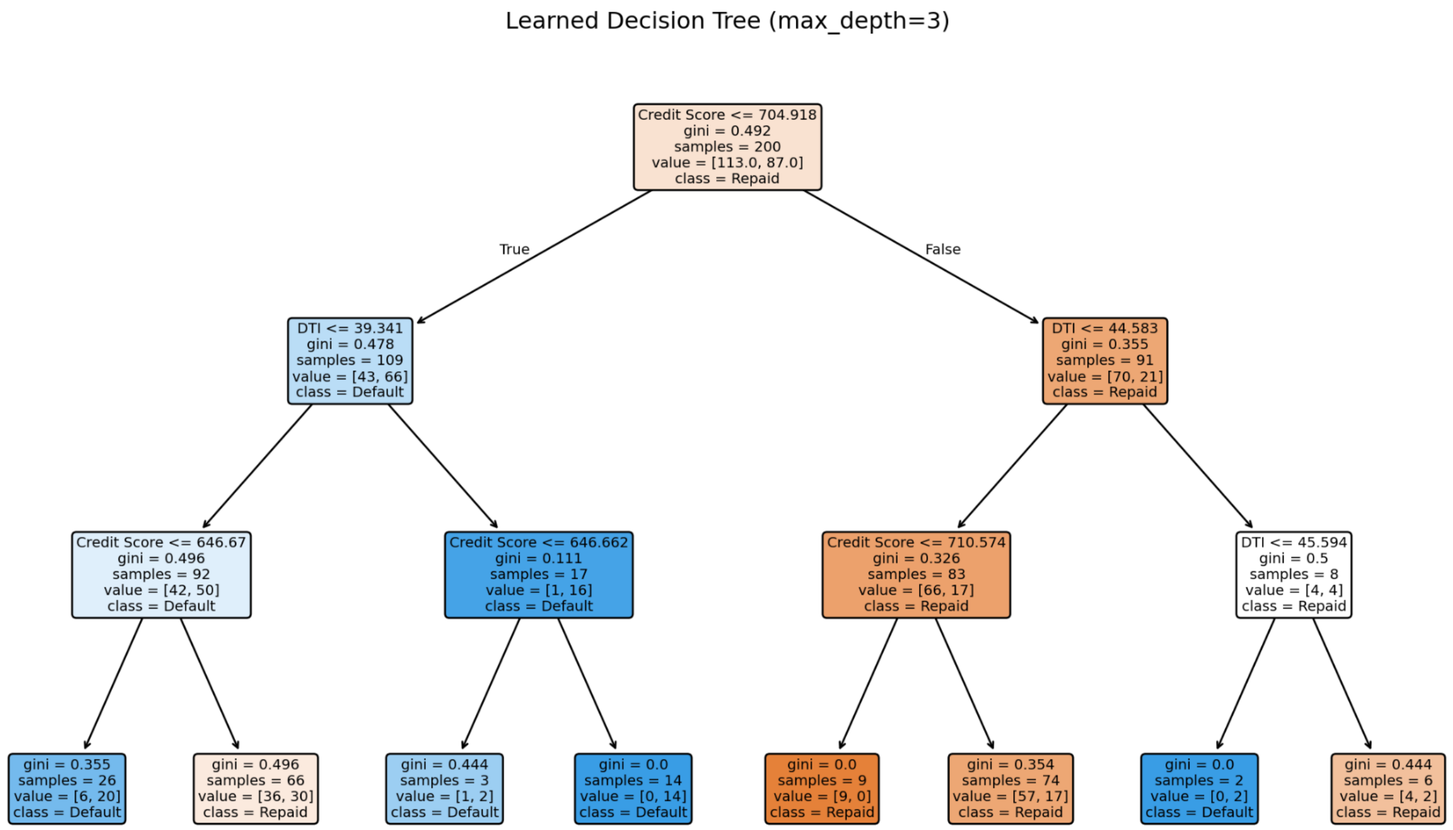
# Generate sample data
np.random.seed(42)
n = 200
credit_score = np.random.normal(700, 50, n)
dti = np.random.normal(30, 10, n)
X_tree = np.column_stack([credit_score, dti])

# Default probability depends on both features
prob_default = 1 / (1 + np.exp(0.02 * (credit_score - 680) - 0.05 * (dti - 35)))
y_tree = (np.random.random(n) < prob_default).astype(int)

# Fit decision tree
tree = DecisionTreeClassifier(max_depth=3, random_state=42)
tree.fit(X_tree, y_tree)
```

```
Tree depth: 3
Number of leaves: 8
Training accuracy: 0.720
```

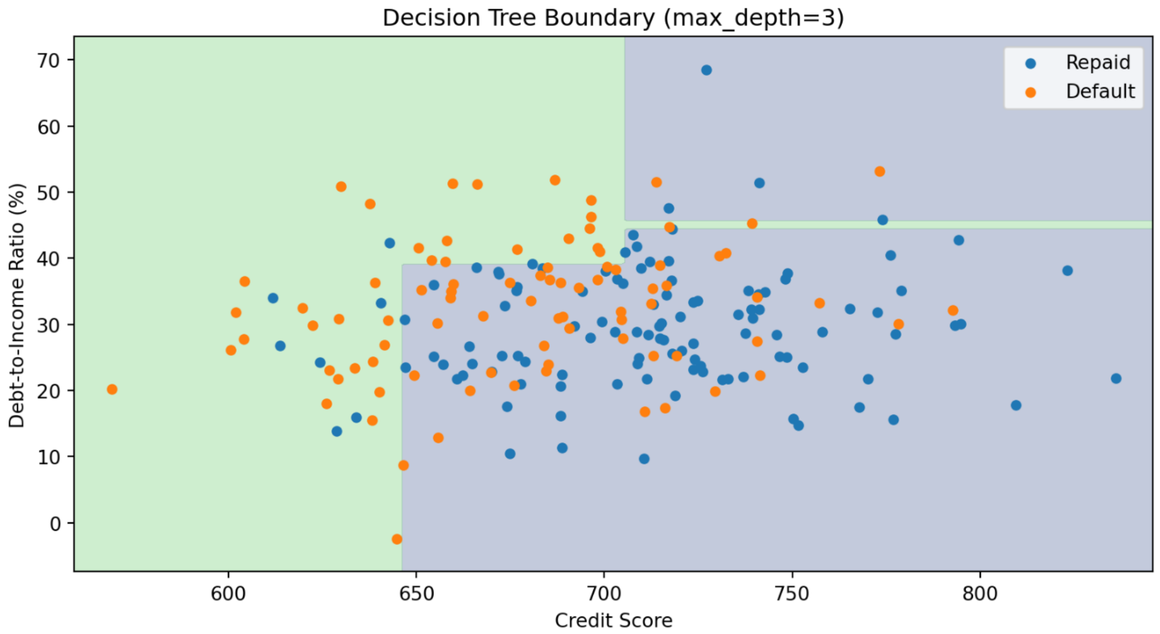
Visualizing the Learned Tree



The tree learns splits automatically from the data. Each node shows the split condition, impurity, sample count, and class distribution.

Rotman Commerce

The Decision Boundary of a Tree



Decision tree boundaries are always **axis-aligned rectangles**—combinations of horizontal and vertical lines. This is a limitation compared to k-NN's curved boundaries.

Controlling Tree Complexity

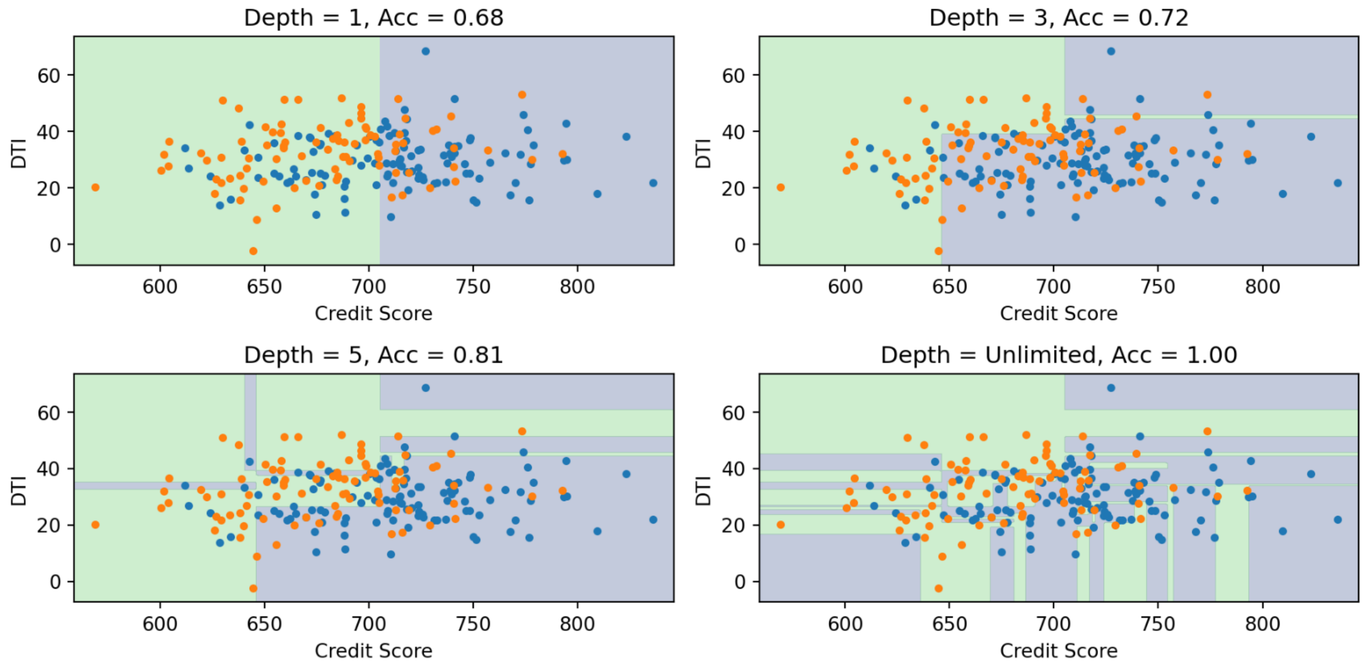
Deep trees can overfit—they memorize the training data perfectly but fail on new data.

Strategies to prevent overfitting:

1. **Pre-pruning:** Stop growing before the tree becomes too complex
   - `max_depth`: Maximum tree depth
   - `min_samples_split`: Minimum samples required to split a node
   - `min_samples_leaf`: Minimum samples required in a leaf
2. **Post-pruning:** Grow a full tree, then remove branches that don't help
   - `ccp_alpha`: Cost-complexity pruning parameter

These hyperparameters are chosen via cross-validation.

Effect of Tree Depth



Deeper trees create more complex boundaries. With unlimited depth, the tree can achieve 100% training accuracy but likely overfits.

Decision Trees: Advantages and Disadvantages

Advantages:

- Easy to interpret and explain (white-box model)
- Handles both numeric and categorical features
- Requires little data preprocessing (no scaling needed)
- Can capture interactions between features
- Fast prediction

Disadvantages:

- Axis-aligned boundaries only (can't capture diagonal boundaries efficiently)
- High variance—small changes in data can produce very different trees
- Prone to overfitting without regularization
- Greedy algorithm may not find globally optimal tree

The high variance problem is addressed by **ensemble methods** (Random Forests, Gradient Boosting)—we'll revisit decision trees as building blocks for these in the next lecture.

Rotman Commerce

Part VI: Evaluating Classification Models

Beyond Accuracy

For regression, we use MSE or R^2 to measure performance.

For classification, **accuracy** (% correct) is the obvious metric:

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

But accuracy can be misleading, especially with **imbalanced classes**.

Example: Predicting credit card fraud (1% fraud rate)

- A model that predicts “not fraud” for everyone achieves 99% accuracy!
- But it catches zero fraud cases—useless for the actual goal.

We need metrics that account for different types of errors.

The Confusion Matrix

A confusion matrix summarizes all prediction outcomes:

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

- **True Positive (TP):** Correctly predicted positive
- **False Positive (FP):** Incorrectly predicted positive (Type I error)
- **True Negative (TN):** Correctly predicted negative
- **False Negative (FN):** Incorrectly predicted positive as negative (Type II error)

Different applications care about different cells of this matrix.

Key Metrics from the Confusion Matrix

Accuracy: Overall correct rate

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

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

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

Recall (Sensitivity): Of the actual positives, how many did we catch?

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

Specificity: Of the actual negatives, how many did we correctly identify?

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

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

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

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

Recall (Sensitivity): Of the actual positives, how many did we catch?

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

Specificity: Of the actual negatives, how many did we correctly identify?

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

False Positive Rate: Of actual negatives, how many did we wrongly call positive?

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

Credit Default: Confusion Matrix

```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}")
```

```
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%}")
```

```python

# 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}")
```

```
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%}")
```

```

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

Rotman Commerce

The Class Imbalance Problem

With 97% non-defaulters and 3% defaulters, the model is biased toward predicting “no default.”

Using threshold = 0.5, we have high accuracy (97%+) but low recall—we miss most actual defaults.

For a credit card company, missing defaults is costly! They’d rather:

- Catch more actual defaults (higher recall)
- Even if it means more false alarms (lower precision)

The 0.5 threshold isn’t sacred—we can adjust it based on business needs.

Adjusting the Classification Threshold

Instead of predicting "default" when $P(\text{default}) > 0.5$, we can use a lower threshold:

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

Lower threshold τ:

- More predictions of "default"
- Higher recall (catch more true defaults)
- Lower precision (more false alarms)

Higher threshold τ:

- Fewer predictions of "default"
- Lower recall (miss more true defaults)
- Higher precision (fewer false alarms)

Effect of Threshold Choice

```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)
```

```
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%
```

Rotman Commerce

```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)
```

```
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.

Rotman Commerce

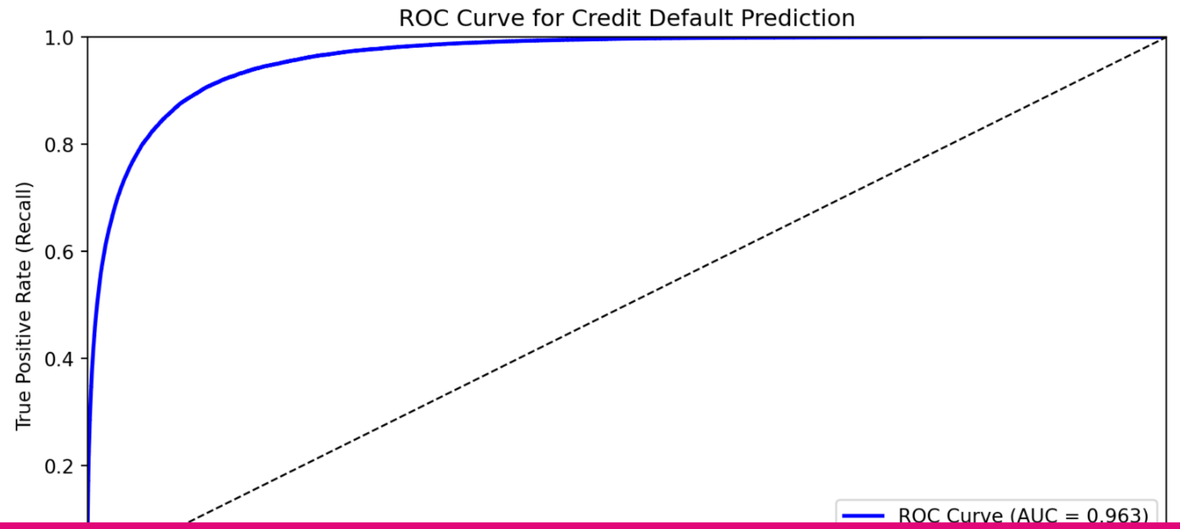
The ROC Curve

The **Receiver Operating Characteristic (ROC) curve** shows the trade-off between true positive rate (recall) and false positive rate across all thresholds.

- X-axis: False Positive Rate (FPR)
- Y-axis: True Positive Rate (TPR = Recall)

As we lower the threshold:

- We move from bottom-left (predict nothing positive) toward top-right (predict everything positive)
- Good classifiers hug the top-left corner

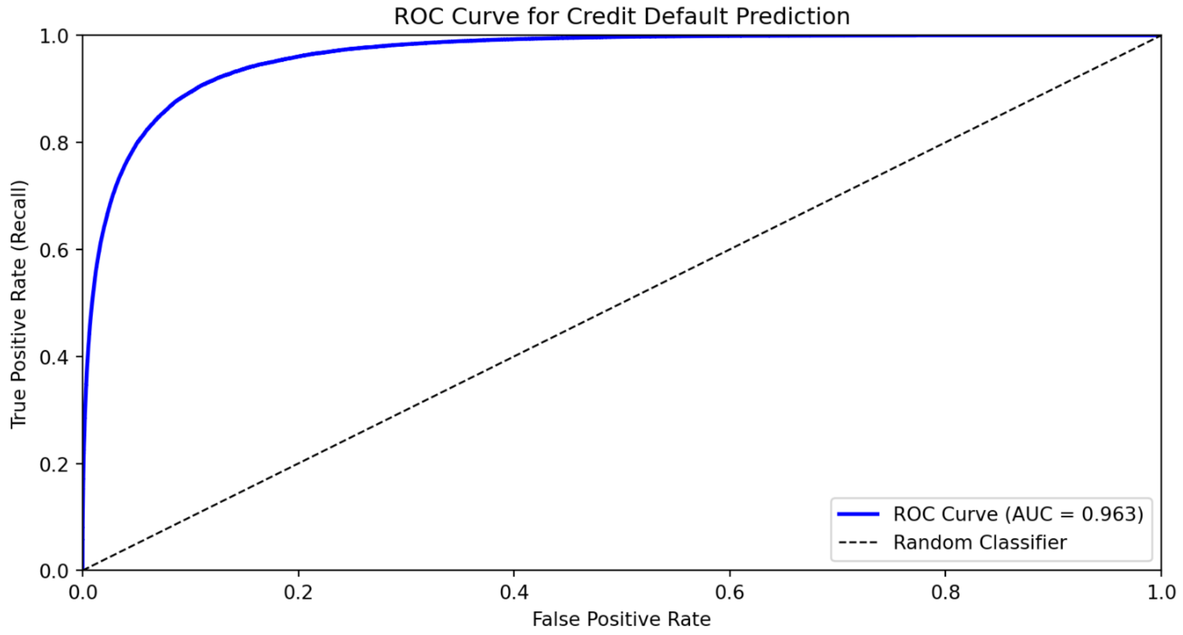


The **Receiver Operating Characteristic (ROC) curve** shows the trade-off between true positive rate (recall) and false positive rate across all thresholds.

- X-axis: False Positive Rate (FPR)
- Y-axis: True Positive Rate (TPR = Recall)

As we lower the threshold:

- We move from bottom-left (predict nothing positive) toward top-right (predict everything positive)
- Good classifiers hug the top-left corner



Area Under the ROC Curve (AUC)

The **Area Under the Curve (AUC)** summarizes the ROC curve in a single number:

- AUC = 1.0: Perfect classifier
- AUC = 0.5: Random guessing (diagonal line)
- AUC < 0.5: Worse than random (predictions inverted)

Interpretation: AUC is the probability that a randomly chosen positive example is ranked higher than a randomly chosen negative example.

```
1 from sklearn.metrics import roc_auc_score
2
3 auc = roc_auc_score(default, prob_pred)
4 print(f"AUC for credit default model: {auc:.3f}")
```

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

AUC is useful for comparing models because it's threshold-independent—it measures the model's ability to rank observations correctly.

Choosing the Optimal Threshold

The “best” threshold depends on the **costs** of different errors:

- Cost of false negative (missing a default): c_{FN}
- Cost of false positive (false alarm): c_{FP}

If missing defaults is very costly (e.g., the bank loses the loan amount), we want a lower threshold to maximize recall.

Objective: 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.

Summary and Preview

What We Learned Today

Classification predicts categorical outcomes—a core supervised learning task.

Logistic regression uses the sigmoid function to model probabilities:

- Outputs are always in (0, 1)
- Coefficients measure effect on log-odds
- Decision boundary is linear; can be extended with feature engineering
- Can be regularized (Lasso) for variable selection

k-Nearest Neighbors classifies based on majority vote among nearby training points:

- Flexible, curved decision boundaries without specifying functional form
- Requires feature scaling; struggles in high dimensions
- No training phase, but slow at prediction time

Decision trees recursively partition feature space with axis-aligned splits:

- Highly interpretable; handles mixed feature types
- Prone to overfitting and high variance
- Building blocks for ensemble methods (next lecture)

Evaluation Metrics

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.

AUC summarizes discriminative ability in a single number.

The optimal **threshold** depends on the costs of different types of errors.

Next Lecture

Lecture 8: Ensemble Methods

Decision trees have high variance—small changes in data can produce very different trees. next lecture we'll see how to fix this.

- **Random Forests:** Average many trees, each trained on random subsets
- **Gradient Boosting:** Build trees sequentially, each correcting previous errors
- **XGBoost:** Industrial-strength boosting used in finance and competitions

Ensemble methods combine many weak learners into a strong learner, dramatically reducing variance while maintaining flexibility.

References

- Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1), 21-27.
- Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). *Classification and Regression Trees*. CRC Press.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning* (2nd ed.). Springer. Chapters 4, 9, 13.
- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An Introduction to Statistical Learning* (2nd ed.). Springer. Chapter 4.
- Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1), 81-106.

Appendix: Linear Discriminant Analysis

Why Another Classifier?

	Clustering (Lecture 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 $P(y \mid \mathbf{x})$	Model $P(\mathbf{x} \mid y)$ per class, then apply Bayes' theorem

LDA is the supervised version of the distributional thinking you used in clustering: instead of discovering groups, you already know them and want to learn what makes each group different.

In practice, LDA and logistic regression often give similar answers — the value is in understanding both ways of thinking about classification.

A Different Approach: Bayes' Theorem

Logistic regression directly models $P(y|\mathbf{x})$—the probability of the class given the features.

Discriminant analysis takes a different approach using Bayes' theorem:

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

In words:

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

Instead of modeling $P(y|\mathbf{x})$ directly, we model:

- $P(y = k)$ — the **prior probability** of each class (how common is each class?)
- $P(\mathbf{x}|y = k)$ — the **likelihood** (what do features look like within each class?)

Setting Up the Model

Let's establish notation:

- K classes labeled $1, 2, \ldots, K$
- $\pi_k = P(y = k)$ — prior probability of class k
- $f_k(\mathbf{x}) = P(\mathbf{x}|y = k)$ — 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 is the same for all classes—it just ensures probabilities sum to 1.

We classify to the class with the highest posterior probability.

The Normality Assumption

Linear Discriminant Analysis (LDA) assumes that within each class, the features follow a **multivariate normal distribution**:

$$\mathbf{X} \mid y = k \sim \square(\boldsymbol{\mu}_k, \boldsymbol{\Sigma})$$

The density function is:

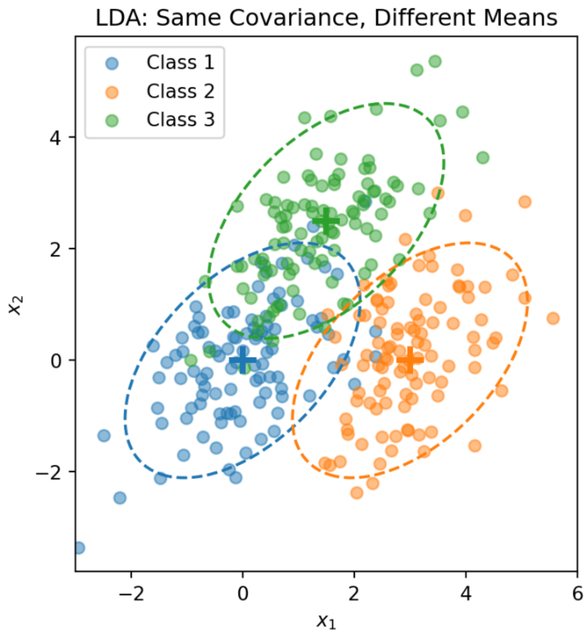
$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)' \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)\right)$$

where:

- $\boldsymbol{\mu}_k$ is the mean of class k (different for each class)
- $\boldsymbol{\Sigma}$ is the covariance matrix (**same for all classes** — this is the key assumption!)

Each class is a normal "blob" centered at $\boldsymbol{\mu}_k$, but all classes share the same shape (covariance).

Visualizing the LDA Assumption



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

The LDA Discriminant Function

Start with the posterior from Bayes' theorem:

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

Taking the log and plugging in the normal density:

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

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

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**:

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

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

Taking the log and plugging in the normal density:

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

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

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**:

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

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.

The LDA Decision Boundary

The decision boundary between classes k and ℓ is where:

$$\delta_k(\mathbf{x}) = \delta_\ell(\mathbf{x})$$

This simplifies to:

$$\mathbf{x}'\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_k - \boldsymbol{\mu}_\ell) = \frac{1}{2}(\boldsymbol{\mu}_k'\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_k - \boldsymbol{\mu}_\ell'\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_\ell) + \ln\frac{\pi_\ell}{\pi_k}$$

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.

Estimating LDA Parameters

In practice, we estimate the parameters from training data:

Prior probabilities:

$$\hat{\pi_k} = \frac{n_k}{n}$$

where n_k is the number of training observations in class k.

Class means:

$$\hat{\boldsymbol{\mu}_k} = \frac{1}{n_k} \sum_{i:y_i=k} \mathbf{x}_i$$

Pooled covariance matrix:

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

The LDA Recipe

	LDA
Model	Each class k is a multivariate normal: $\mathbf{x} \mid y = k \sim \square(\boldsymbol{\mu}_k, \boldsymbol{\Sigma})$
Parameters	Priors $\hat{\pi_k} = n_k/n$, means $\hat{\boldsymbol{\mu}_k}$, pooled covariance $\hat{\boldsymbol{\Sigma}}$
"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.

Rotman Commerce

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_}")
```

```
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]
```

```python
# Visualize LDA decision boundaries
fig, ax = plt.subplots()

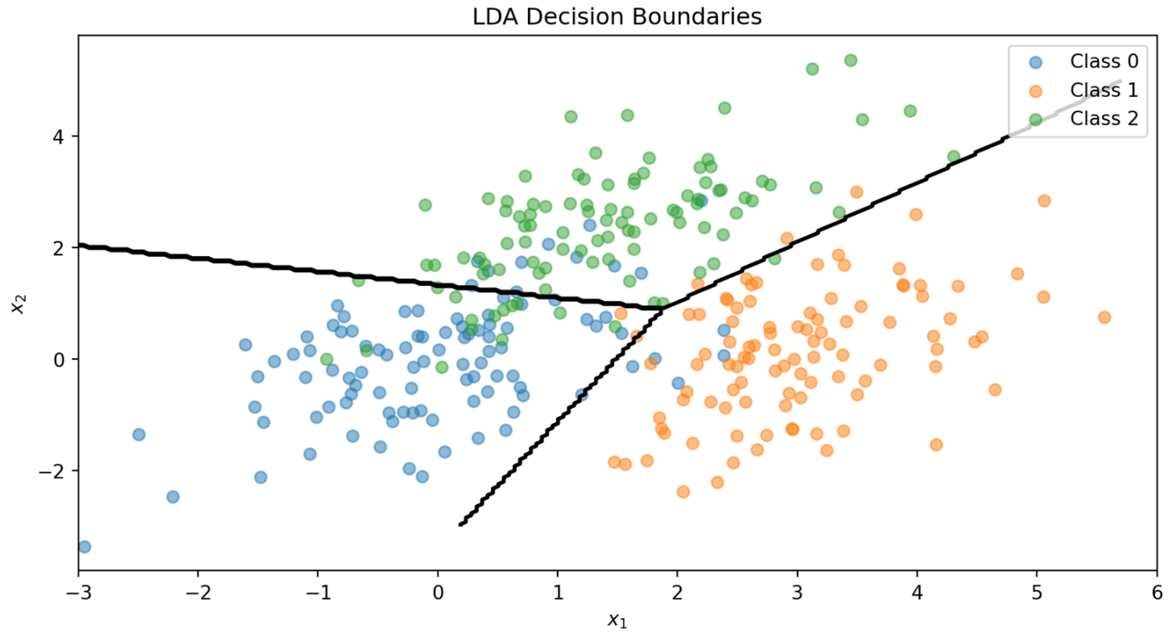
ax.scatter(X1[:, 0], X1[:, 1], alpha=0.5, label='Class 0')
ax.scatter(X2[:, 0], X2[:, 1], alpha=0.5, label='Class 1')
```

Rotman Commerce

```
ax.scatter(X1[:, 0], X1[:, 1], alpha=0.5, label='Class 0')
ax.scatter(X2[:, 0], X2[:, 1], alpha=0.5, label='Class 1')
ax.scatter(X3[:, 0], X3[:, 1], alpha=0.5, label='Class 2')

# Decision boundaries
x_range = np.linspace(-3, 6, 200)
y_range = np.linspace(-3, 5, 200)
X_grid, Y_grid = np.meshgrid(x_range, y_range)
grid_points = np.column_stack([X_grid.ravel(), Y_grid.ravel()])
Z = lda.predict(grid_points).reshape(X_grid.shape)

ax.contour(X_grid, Y_grid, Z, levels=[0.5, 1.5], colors='black', linewidths=2)
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
```



Quadratic Discriminant Analysis (QDA)

LDA assumes all classes share the same covariance matrix $\boldsymbol{\Sigma}$.

Quadratic Discriminant Analysis (QDA) relaxes this: each class has its own covariance $\boldsymbol{\Sigma}_k$.

The discriminant function becomes:

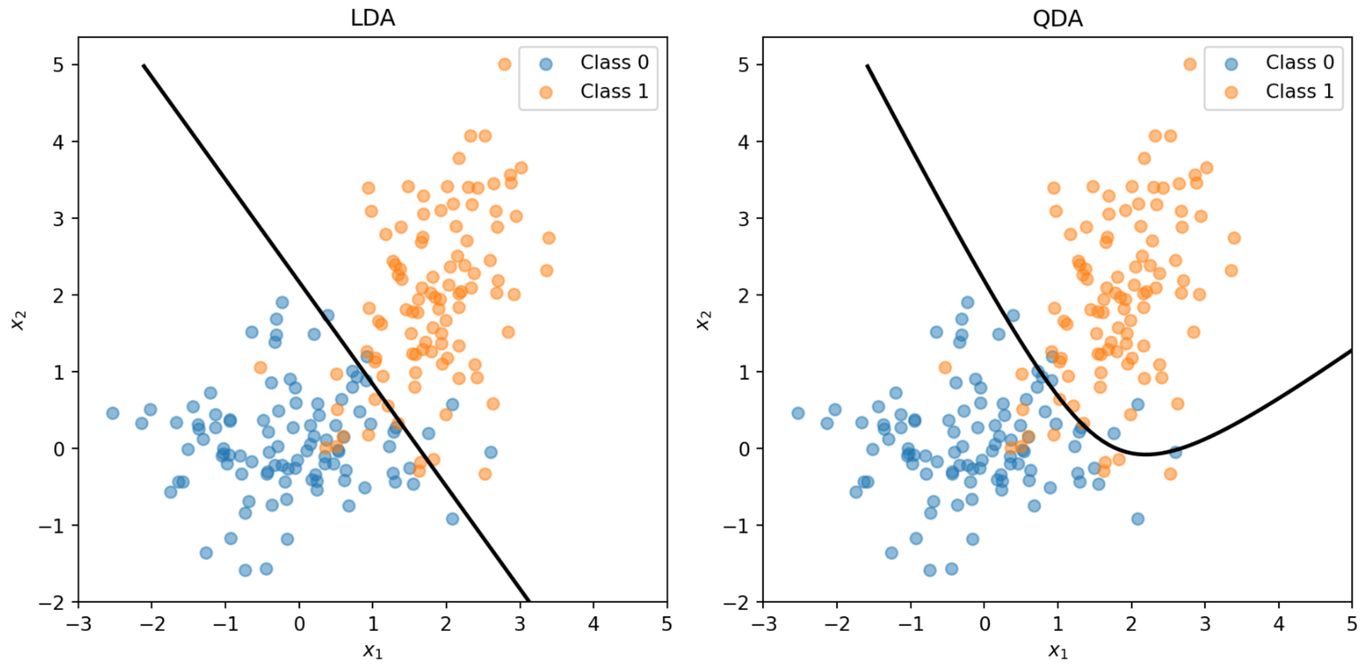
$$\delta_k(\mathbf{x}) = -\frac{1}{2}\ln|\boldsymbol{\Sigma}_k| - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)'\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}-\boldsymbol{\mu}_k) + \ln\pi_k$$

This is **quadratic** in $\mathbf{x}$, giving curved decision boundaries.

Trade-off:

- LDA: More restrictive assumptions, fewer parameters to estimate, more stable
- QDA: More flexible, more parameters, can overfit with small samples

LDA vs. QDA



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

Why LDA Works Well

LDA and QDA have good track records as classifiers, not necessarily because the normality assumption is correct, but because:

1. **Stability:** Estimating fewer parameters (shared Σ) reduces variance
2. **Robustness:** The decision boundary often works well even when normality is violated
3. **Closed-form solution:** No iterative optimization needed

When to use which:

- **LDA:** When you have limited data or classes are well-separated
- **QDA:** When you have more data and suspect different class shapes
- **Regularized DA:** Blend between LDA and QDA to control flexibility

LDA vs. Logistic Regression

Both produce linear decision boundaries. When to use which?

Logistic Regression:

- Makes no assumption about the distribution of $\mathbf{x}$
- More robust when the normality assumption is violated
- Preferred when you have binary features or mixed feature types

LDA:

- More efficient when normality holds (uses information about class distributions)
- Can be more stable with small samples
- Naturally handles multi-class problems

In practice, they often give similar results. Try both and compare via cross-validation.