AERO 489/689 Intro ML for AERO – Classification

AERO 689: Machine Learning for Aerospace Engineers

Raktim Bhattacharya

Texas A&M University

Overview

Topics Covered

1. Classification Problem
2. Evaluation Metrics
3. Logistic Regression
4. Additional Classifiers
5. Imbalanced Data

Classification

Definition

Classification: Learn $f : X \to Y$ where $Y = \{0, 1, \ldots, K-1\}$

Regression: Learn $f : X \to \mathbb{R}$

Classification error (0-1 loss):

$$\text{Error}(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}[f(x_i) \neq y_i]$$

Example: $n = 5$ samples, $y = [0, 1, 1, 0, 1]$, $\hat{y} = [0, 1, 0, 0, 1]$

$$\text{Error} = \frac{1}{5}(0 + 0 + 1 + 0 + 0) = 0.20$$

Classes and Labels

Class: A category that a data sample can belong to

Label: The value y that indicates which class a sample belongs to

Label Representations:

Integer encoding: $y \in \{0, 1, 2, 3, 4\}$

- Example: Classes $=$ {Taxi, Takeoff, Cruise, Descent, Landing}
- Sample label: $y = 2$ (means "Cruise")

One-hot encoding: $y \in \{0, 1\}^K$ (binary vector)

- Same classes: $y = [0, 0, 1, 0, 0]$ (also means "Cruise")
- Position indicates class, value indicates presence

Different algorithms use different representations

Classification Types

Binary ($K = 2$): $y \in \{0, 1\}$

- Example: Sensor fault detection (faulty/healthy)

Multiclass ($K > 2$): One label from K classes

- Output: $y \in \{0, 1, \ldots, K-1\}$ (single integer)
- Example: Flight phase $\rightarrow y = 2$ means "cruise"
- Classes: {0=taxi, 1=takeoff, 2=cruise, 3=descent, 4=landing}

Multilabel: Multiple independent binary decisions

- Output: $y \in \{0, 1\}^K$ (binary vector of length K)
- Example: System faults $\rightarrow y = [1, 0, 1, 0]$
- Means: engine=faulty, hydraulic=ok, electrical=faulty, fuel=ok

Multiclass vs Multilabel

Core Difference: "Pick one" vs "Pick any combination"

Multiclass: Choose exactly 1 from K options

- Think: Radio buttons (only one can be selected)
- Example: Aircraft in cruise phase (can't be in taxi AND cruise simultaneously)

Multilabel: Independent yes/no for each of K options

- Think: Checkboxes (select any combination)
- Example: Multiple systems can fail at the same time

Multiclass vs Multilabel (contd.)

Aspect	Multiclass	Multilabel
Output	Single integer	Binary vector
Notation	$y = 2$	$y = [1, 0, 1, 0]$
Constraint	Exactly one active	Any subset active

Probabilistic Formulation

Goal: Predict class probabilities, not just hard labels

Instead of $f(x) = k$, model $P[Y = k|X = x]$ for all classes

Benefits:

- Quantify prediction confidence
- Handle uncertainty in aerospace safety applications
- Enable threshold tuning for cost-sensitive decisions

Bayes Optimal Classifier

Model class posterior: $P(Y = k|X = x)$

Optimal decision rule:

$$f^*(x) = \arg\max_k P(Y = k|X = x)$$

Minimizes expected 0-1 loss (classification error)

Interpretation: Choose the most probable class

Classifier Reliability & Accuracy

How Do We Measure Success?

Question: How do we know if our classifier is good?

Challenges:

- Different applications have different priorities
- Not all errors are equally costly
- Class distributions may be imbalanced (one class much more frequent than others)

Aerospace Context:

- Fault detection: Missing a fault is catastrophic
- False alarms: Costly but less critical
- Need metrics that reflect real-world consequences

Goal: Choose metrics that align with application requirements

Why Not Just Accuracy?

Accuracy: Fraction of correct predictions

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

Problem: Misleading for imbalanced classes

- 10,000 flights, 100 have faults (1%)
- Classifier predicts "no fault" for all flights
- Accuracy = 99% but detects zero faults!

Need: Better metrics to evaluate classification performance

Confusion Matrix

Classification outcomes organized by actual vs predicted:

	Pred: Negative	Pred: Positive
Actual: Negative	TN	FP
Actual: Positive	FN	TP

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

Computing Confusion Matrix Elements

The Methodology: Compare actual vs predicted for each sample

Decision Rules:

- **TP** (True Positive): $y = 1$ AND $\hat{y} = 1$ (correct)
- **TN** (True Negative): $y = 0$ AND $\hat{y} = 0$ (correct)
- **FP** (False Positive): $y = 0$ BUT $\hat{y} = 1$ (error)
- **FN** (False Negative): $y = 1$ BUT $\hat{y} = 0$ (error)

Computing Confusion Matrix Elements (contd.)

Example: 10 samples, binary classification (0 = No Fault, 1 = Fault)

Sample	Actual (y)	Predicted ($\hat{y}$)	Result
1	0	0	TN
2	0	0	TN
3	0	1	FP
4	1	1	TP
5	1	0	FN
6	0	0	TN
7	1	1	TP
8	0	1	FP
9	1	1	TP
10	0	0	TN

Counts:

Count samples where:

- TP = 3 ($y = 1, \hat{y} = 1$)
- TN = 4 ($y = 0, \hat{y} = 0$)
- FP = 2 ($y = 0, \hat{y} = 1$)
- FN = 1 ($y = 1, \hat{y} = 0$)

Check: $3 + 4 + 2 + 1 = 10$

Accuracy Revisited

Now we can express accuracy formally:

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

The 99% accuracy example:

- TN = 9,900 (correctly predicted no fault)
- FN = 100 (missed all faults)
- TP = 0, FP = 0

$$\text{Accuracy} = \frac{0 + 9900}{0 + 9900 + 0 + 100} = 0.99$$

Looks good, but FN = 100 is catastrophic!

Key Metrics

Precision: $P = \frac{TP}{TP+FP}$ — positive predictive value

Recall: $R = \frac{TP}{TP+FN}$ — sensitivity, true positive rate

Specificity: $\frac{TN}{TN+FP}$ — true negative rate

F1 Score: $F_1 = \frac{2PR}{P+R}$ — harmonic mean

How to Use These Metrics

When to prioritize Recall: Minimize false negatives

- Fault detection: Can't miss critical failures
- Medical diagnosis: Better safe than sorry

When to prioritize Precision: Minimize false positives

- Spam detection: Don't block important emails
- Expensive interventions: Avoid unnecessary maintenance

When to use F_1: Balance precision and recall

- General classification when both errors matter equally

When to use Specificity: Important for majority class performance

- Verify healthy systems correctly classified

Metric Selection Example

Scenario: Aircraft engine fault detection

- **100 flights**: 2 have actual faults (2% positive class)
- **Classifier A**: High recall (100%), low precision (10%)
  - Detects both faults but 18 false alarms
- **Classifier B**: High precision (50%), low recall (50%)
  - Detects 1 fault, only 1 false alarm

Choice depends on cost:

- Missing a fault $\rightarrow$ engine failure (catastrophic)
- False alarm $\rightarrow$ unnecessary inspection (costly but safe)
- **Choose Classifier A** (high recall) for safety-critical applications

Logistic Regression

Probabilistic Classification Framework

Supervised Learning: We have labeled training data $\{(x_i, y_i)\}_{i=1}^n$

- Features x_i (sensor readings, measurements, etc.)
- Labels y_i (known outcomes: faulty/healthy, phase labels, etc.)
- Goal: Learn $f : X \to Y$ that predicts labels for new, unseen data

We want a classifier with quantified uncertainty:

$$f(x) = \arg\max_k P(Y = k|X = x)$$

Challenge: We don't know the true class probabilities $P(Y = k|X = x)$

Solution: Use labeled training data to learn a model that estimates these probabilities

Probabilistic Classification Framework (contd.)

Logistic Regression: Supervised parametric approach

- Learns from labeled examples
- Assumes specific functional form (sigmoid)
- Uses maximum likelihood to find optimal parameters
- Provides calibrated probability estimates

Why Logistic Regression?

Goal: Estimate $P(Y = 1|X = x)$ for binary classification

Why not linear regression? $y = \beta^T x$

- Predictions can be < 0 or > 1 (invalid probabilities!)
- Doesn't model probability constraints

Solution: Transform linear model through sigmoid function

$$P(Y = 1|X = x) = \sigma(\beta^T x) = \frac{1}{1 + e^{-\beta^T x}}$$

- Output always in $[0, 1]$
- Monotonic: larger $\beta^T x \rightarrow$ higher probability
- Smooth and differentiable (easy to optimize)

Sigmoid Function

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

Properties:

- $\sigma(0) = 0.5$
- $\sigma(-z) = 1 - \sigma(z)$
- $\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$

Understanding Odds and Log-Odds

From probabilities to odds:

$$\text{odds} = \frac{P(y=1|x)}{P(y=0|x)} = \frac{P(y=1|x)}{1-P(y=1|x)} \in [0, \infty)$$

- Probability $p = 0.5 \rightarrow$ odds $= 1$ (even chance)
- Probability $p = 0.8 \rightarrow$ odds $= 4$ ($4\times$ more likely positive)
- Probability $p = 0.2 \rightarrow$ odds $= 0.25$ ($4\times$ more likely negative)

Why log-odds?

$$\text{log-odds} = \log \frac{P(y=1|x)}{P(y=0|x)}$$

- Range: $(-\infty, +\infty)$ instead of $(0, \infty)$
- Symmetric: $\log(\text{odds}) = -\log(1/\text{odds})$
- Linear relationship with features: $\log(\text{odds}) = \beta^T x$

Logistic Regression Model

Binary classification: $y \in \{0, 1\}$

Linear model in log-odds space:

$$\log \frac{P(y=1|x)}{P(y=0|x)} = \beta^T x = \beta_0 + \beta_1 x_1 + \cdots + \beta_d x_d$$

Interpretation of coefficients:

- $\beta_j > 0$: Feature x_j increases probability of class 1
- $\beta_j < 0$: Feature x_j decreases probability of class 1
- $\beta_j = 0$: Feature x_j has no effect

Model Summary

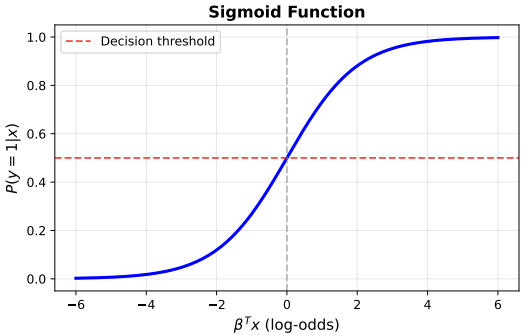
$$P(y = 1|x) = \sigma(\beta^T x) = \frac{1}{1 + e^{-\beta^T x}}$$

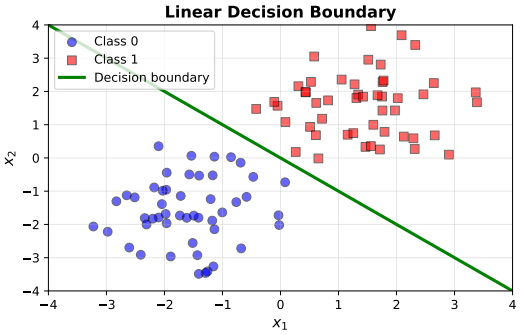
Decision rule: $\hat{y} = \mathbf{1}[P(y = 1|x) > \tau]$ (typically $\tau = 0.5$)

Decision boundary: Hyperplane $\{x : \beta^T x = 0\}$

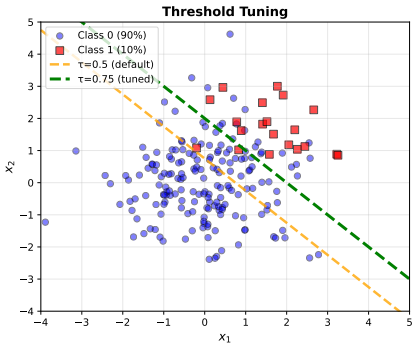
- Points on one side $\rightarrow P(y = 1|x) > 0.5 \rightarrow$ predict class 1
- Points on other side $\rightarrow P(y = 1|x) < 0.5 \rightarrow$ predict class 0

Model Visualization





Threshold Tuning: Imbalanced Case



Problem with $\tau = 0.5$:
Many Class 0 points would be misclassified as Class 1 (high false positives)

How to choose threshold? (Discussed later)

1. **Validation set**: Try different , maximize target metric (F_β, recall, precision)
2. **ROC/PR curve**: Plot performance across all thresholds, choose optimal point
3. **Cost-sensitive**: Minimize $L = C_{FN} \cdot FN + C_{FP} \cdot FP$

For this case: $\tau = 0.75$ reduces false positives while maintaining recall on critical minority class

Training Logistic Regression

The Question: How do we find the optimal parameters β?

What we have:

- Model form: $P(y = 1|x) = \sigma(\beta^T x)$ binary classifier, for simplicity
- Training data: $\{(x_i, y_i)\}_{i=1}^n$

What we need: A principled method to learn β from data

Approach: Find β that makes observed data most likely

This leads to **Maximum Likelihood Estimation** (MLE)

Maximum Likelihood Estimation: The Setup

The Problem: We have a model form but need to find the best β

Our model: $P(y = 1|x) = \sigma(\beta^T x)$

Training data: $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$

Question: Which β makes our model "best explain" the observed data?

MLE Principle: Choose β that maximizes the probability of observing our actual training data

Understanding Likelihood

Likelihood = Probability of observing the data, given parameters β

For a single sample (x_i, y_i):

$$P(\text{observing } y_i | x_i, \beta) = \begin{cases} p_i & \text{if } y_i = 1 \\ 1 - p_i & \text{if } y_i = 0 \end{cases}$$

where $p_i = \sigma(\beta^T x_i)$ is our model's predicted probability

Compact form: $P(y_i | x_i, \beta) = p_i^{y_i}(1 - p_i)^{1-y_i}$ Bernoulli distribution

Check the formula:

- If $y_i = 1$: $p_i^1(1 - p_i)^0 = p_i$
- If $y_i = 0$: $p_i^0(1 - p_i)^1 = 1 - p_i$

Bernoulli in Logistic Regression

Each binary label follows a Bernoulli distribution:

- Each training sample: (x_i, y_i) where $y_i \in \{0, 1\}$
- Our model predicts: $p_i = \sigma(\beta^T x_i)$
- Each y_i is a Bernoulli trial with parameter p_i
- Training data: n independent Bernoulli trials

Why this matters: The Bernoulli PMF gives us the formula to compute likelihood!

Building the Likelihood Function

From single sample to all data:

$$\text{Single: } P(y_i|x_i,\beta) = p_i^{y_i}(1-p_i)^{1-y_i}$$

$$\text{All data: } L(\beta) = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i}$$

MLE objective: Find β^* that maximizes $L(\beta)$

$$\beta^* = \arg\max_{\beta} \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i}$$

Different $\beta \rightarrow$ different $p_i \rightarrow$ different likelihood

The Complete Chain: $\beta \rightarrow$ Likelihood

Step 1: Parameters $\rightarrow$ Linear combination

$$z_i = \beta^T x_i$$

Step 2: Linear combination $\rightarrow$ Probability

$$p_i = \sigma(z_i) = \frac{1}{1 + e^{-\beta^T x_i}}$$

Step 3: Probability $\rightarrow$ Likelihood

$$P(y_i | x_i, \beta) = p_i^{y_i} (1 - p_i)^{1 - y_i}$$

MLE: Concrete Numerical Example

Recall the chain: $\beta \to \beta^T x_i \to \sigma(\beta^T x_i) = p_i \to \text{Bernoulli}(p_i)$

Our 3-sample dataset:

Sample	Features x_i	Actual label y_i
1	[2, 1]	1 (positive)
2	[-1, 3]	0 (negative)
3	[0, 2]	1 (positive)

What we'll do:

1. Pick a candidate β (e.g., $\beta = [0.5, 0.3]$)
2. Compute predicted probabilities p_i for each sample
3. Calculate likelihood using Bernoulli formula: $L(\beta) = \prod p_i^{y_i}(1-p_i)^{1-y_i}$
4. Compare different β values to find which maximizes $L(\beta)$

MLE: How β Determines Probabilities

Our model: $p_i = P(y_i = 1|x_i, \beta) = \sigma(\beta^T x_i) = \frac{1}{1+e^{-\beta^T x_i}}$

Key insight: Different β values give different probability predictions

Example: Let's try $\beta = [0.5, 0.3]$

For each sample, compute $\beta^T x_i$, then apply sigmoid:

Sample	x_i	$\beta^T x_i$	$p_i = \sigma(\beta^T x_i)$	Actual y_i
1	[2,1]	$0.5(2) + 0.3(1) = 1.3$	$\sigma(1.3) = 0.79$	1
2	[-1,3]	$0.5(-1) + 0.3(3) = 0.4$	$\sigma(0.4) = 0.60$	0
3	[0,2]	$0.5(0) + 0.3(2) = 0.6$	$\sigma(0.6) = 0.65$	1

MLE: Computing Likelihood

For this β, how well do predictions match actual labels?

Use Bernoulli formula: $P(y_i|x_i, \beta) = p_i^{y_i}(1-p_i)^{1-y_i}$

Sample	y_i	p_i	$1-p_i$	Likelihood contribution
1	1	0.79	0.21	$0.79^1 \times 0.21^0 = 0.79$
2	0	0.60	0.40	$0.60^0 \times 0.40^1 = 0.40$
3	1	0.65	0.35	$0.65^1 \times 0.35^0 = 0.65$

Total likelihood: $L(\beta) = 0.79 \times 0.40 \times 0.65 = 0.21$

Interpretation: With this β, there's a 21% probability of observing our exact training data

Is this good? We don't know yet — need to try other β values and compare!

MLE: Finding Optimal β

Key idea: Try many different β values, pick the one with highest $L(\beta)$

Example comparison:

β	p_1	p_2	p_3	$L(\beta)$	Better?
[0.5, 0.3]	0.79	0.60	0.65	0.21	Baseline
[0.8, 0.4]	0.90	0.30	0.80	0.50	Yes! Much better
[1.0, 0.2]	0.88	0.40	0.73	0.26	Worse than [0.8, 0.4]

Observations:

- $\beta = [0.8, 0.4]$ gives higher p_i when $y_i = 1$ and lower p_i when $y_i = 0$
- This makes the observed data more likely: $L = 0.50 > 0.21$
- **MLE objective**: $\beta^* = \arg\max_\beta L(\beta)$
- Use gradient descent to efficiently search for optimal β

From Likelihood to Log-Likelihood

Problem with products: For n samples, $L(\beta) = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i}$

Issues:

- Products of many small numbers $\rightarrow$ numerical underflow
- Hard to differentiate products

Solution: Take logarithm (log is monotonic, so max of L = max of $\log L$)

$$\ell(\beta) = \log L(\beta) = \log \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i},$$
$$= \sum_{i=1}^{n} [y_i \log p_i + (1-y_i)\log(1-p_i)]$$

Maximum Likelihood Formulation

Log-likelihood:

$$\ell(\beta) = \sum_{i=1}^{n} [y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

where $p_i = \sigma(\beta^T x_i)$

MLE objective: Find $\beta^* = \arg\max_\beta \ell(\beta)$

In practice: Minimize negative log-likelihood (turn maximization into minimization)

$$\beta^* = \arg\min_\beta \left[-\frac{1}{n} \sum_{i=1}^{n} [y_i \log p_i + (1 - y_i) \log(1 - p_i)] \right]$$

This is our **loss function** $J(\beta)$

Training

Loss (negative log-likelihood):

$$J(\beta) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log p_i + (1-y_i)\log(1-p_i)\right]$$

Gradient:

$$\nabla J = \frac{1}{n} X^T (p - y)$$

Update: $\beta \leftarrow \beta - \alpha \nabla J$

Regularization

L_2 (Ridge): $J(\beta) + \lambda \|\beta\|_2^2$

L_1 (Lasso): $J(\beta) + \lambda \|\beta\|_1$

Prevents overfitting, controls model complexity

Code Example: Sensor Fault Detection

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report

# Simulated sensor data: [temperature, pressure, vibration]
X_train = np.array([[25, 100, 0.5], [27, 102, 0.6], [80, 95, 2.1],
                    [26, 101, 0.55], [85, 90, 2.3], [24, 103, 0.45]])
y_train = np.array([0, 0, 1, 0, 1, 0])  # 0=healthy, 1=faulty

# Train logistic regression
model = LogisticRegression()
model.fit(X_train, y_train)
```

Code Example: Sensor Fault Detection (contd.)

```
# New sensor readings
X_test = np.array([[26, 100, 0.5], [82, 92, 2.0]])
y_test = np.array([0, 1])

# Predictions and probabilities
y_pred = model.predict(X_test)
probs = model.predict_proba(X_test)[:, 1]  # P(fault)

print(f"Predictions: {y_pred}")
print(f"Probabilities: {probs}")
print(classification_report(y_test, y_pred,
      target_names=['Healthy', 'Faulty']))
```

Multiclass Extension: The Problem

Binary logistic regression: 2 classes, 1 probability

- Sigmoid: $P(y = 1|x) = \sigma(\beta^T x)$
- Automatically: $P(y = 0|x) = 1 - P(y = 1|x)$

Multiclass problem: $K > 2$ classes, need K probabilities

- Flight phase: {taxi, takeoff, cruise, descent, landing} $\rightarrow K = 5$
- Need: $P(y = 0|x), P(y = 1|x), \ldots, P(y = 4|x)$
- Constraint: $\sum_{k=0}^{K-1} P(y = k|x) = 1$ (probabilities must sum to 1)

Challenge: How do we ensure all probabilities are valid and sum to 1?

Softmax Function: The Solution

Idea: Create K separate linear models, then normalize

Step 1: Compute K "scores" (one per class)

$$z_k = \beta_k^T x \quad \text{for } k = 0, 1, \ldots, K-1$$

Step 2: Convert scores to probabilities using **softmax**

$$P(y = k|x) = \frac{e^{z_k}}{\sum_{j=0}^{K-1} e^{z_j}} = \frac{e^{\beta_k^T x}}{\sum_{j=0}^{K-1} e^{\beta_j^T x}}$$

Why exponentials?

- $e^{z_k} > 0$ ensures positive values
- Division by sum ensures $\sum_k P(y = k|x) = 1$
- Larger $z_k \rightarrow$ higher probability for class k

Softmax Generalizes Sigmoid

For binary case ($K = 2$): Softmax reduces to sigmoid!

$$P(y = 1|x) = \frac{e^{\beta_1^T x}}{e^{\beta_0^T x} + e^{\beta_1^T x}} = \frac{1}{1 + e^{-(\beta_1 - \beta_0)^T x}}$$

This is exactly sigmoid with $\beta = \beta_1 - \beta_0$

Key insight: Softmax is the natural generalization of logistic regression to multiple classes

Training: Cross-Entropy Loss

For each sample: Maximize probability of correct class

- Sample i has true label $y_i \in \{0, 1, \ldots, K-1\}$
- Model predicts: $P(y = k|x_i)$ for all k
- Want: High probability for $k = y_i$

Cross-entropy loss:

$$J = -\frac{1}{n} \sum_{i=1}^{n} \log P(y = y_i|x_i)$$

Equivalently (using indicator function):

$$J = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=0}^{K-1} \mathbf{1}[y_i = k] \log P(y = k|x_i)$$

Connection to binary: When $K = 2$, this becomes negative log-likelihood!

Multiclass Example: Bearing Fault Classification

Application Context:

- **Problem**: Bearings are critical components in aircraft engines, turbines, landing gear
- **Failure modes**: Different fault types have distinct vibration signatures
- **Goal**: Early detection of specific fault type for targeted maintenance

Classes ($K = 5$):

- Class 0: Normal (healthy bearing)
- Class 1: Inner race fault

  Fault in the inner ring of the bearing
- Class 2: Outer race fault

  Fault in the outer ring of the bearing
- Class 3: Ball (rolling element) fault
- Class 4: Cage fault

  Fault in the retainer that holds the rolling elements

Why multiclass? Each fault requires different maintenance action

Feature Extraction from Vibration Signals

Raw data: Vibration time series from accelerometer

Feature engineering: Extract statistical and spectral features

Feature	Description	Physical Meaning
RMS	Root mean square	Overall vibration energy
Kurtosis	Fourth moment (peakedness)	Impulsiveness (spikes)
Peak freq	Dominant frequency	Fault characteristic frequency
Spectral entropy	Frequency distribution	Signal randomness

Example values for one sample:

- RMS $= 0.85$, Kurtosis $= 6.2$, Peak freq $= 120$ Hz, Spectral entropy $= 0.42$

Softmax Model: Architecture

For each class k, we have separate linear model:

$$z_k = \beta_k^T x + b_k$$

For $K = 5$ **classes with** $d = 4$ **features**:

$$\begin{aligned}
z_0 &= \beta_{0,1} \cdot \text{RMS} + \beta_{0,2} \cdot \text{Kurtosis} + \beta_{0,3} \cdot \text{Peak freq} + \beta_{0,4} \cdot \text{Entropy} + b_0 \\
z_1 &= \beta_{1,1} \cdot \text{RMS} + \beta_{1,2} \cdot \text{Kurtosis} + \beta_{1,3} \cdot \text{Peak freq} + \beta_{1,4} \cdot \text{Entropy} + b_1 \\
&\vdots \\
z_4 &= \beta_{4,1} \cdot \text{RMS} + \beta_{4,2} \cdot \text{Kurtosis} + \beta_{4,3} \cdot \text{Peak freq} + \beta_{4,4} \cdot \text{Entropy} + b_4
\end{aligned}$$

Total parameters: $5 \times 4 = 20$ weights $\beta + 5$ biases $b = 25$ parameters

Computing Class Scores

Sample features: $x = [0.85, 6.2, 120, 0.42]$

Learned parameters (example after training):

$$\beta_0 = [0.5, -0.3, 0.01, -1.2], \quad b_0 = 0.2$$
$$\beta_1 = [0.8, 0.4, 0.02, 0.5], \quad b_1 = -0.5$$
$$\beta_2 = [1.2, 0.6, 0.03, 0.8], \quad b_2 = -0.3$$
$$\beta_3 = [0.6, 0.2, 0.015, 0.3], \quad b_3 = -0.8$$
$$\beta_4 = [0.4, -0.1, 0.005, -0.5], \quad b_4 = -1.0$$

Computing Class Scores (contd.)

Compute score for each class:

$$z_0 = 0.5(0.85) + (-0.3)(6.2) + 0.01(120) + (-1.2)(0.42) + 0.2 = -1.2$$

$$z_1 = 0.8(0.85) + 0.4(6.2) + 0.02(120) + 0.5(0.42) + (-0.5) = 0.5$$

$$z_2 = 1.2(0.85) + 0.6(6.2) + 0.03(120) + 0.8(0.42) + (-0.3) = 3.8$$

$$z_3 = 0.6(0.85) + 0.2(6.2) + 0.015(120) + 0.3(0.42) + (-0.8) = 1.0$$

$$z_4 = 0.4(0.85) + (-0.1)(6.2) + 0.005(120) + (-0.5)(0.42) + (-1.0) = -0.6$$

Observation: Class 2 (outer race fault) has highest score $z_2 = 3.8$

Softmax: Scores to Probabilities

Apply softmax to convert scores to probabilities:

$$P(y = k|x) = \frac{e^{z_k}}{\sum_{j=0}^{4} e^{z_j}}$$

Step 1: Compute exponentials

$$e^{z_0} = e^{-1.2} = 0.30, \quad e^{z_1} = e^{0.5} = 1.65, \quad e^{z_2} = e^{3.8} = 44.70$$

$$e^{z_3} = e^{1.0} = 2.72, \quad e^{z_4} = e^{-0.6} = 0.55$$

Step 2: Sum of exponentials

$$\text{Sum} = 0.30 + 1.65 + 44.70 + 2.72 + 0.55 = 49.92$$

Softmax: Scores to Probabilities (contd.)

Step 3: Normalize to get probabilities

$$P(y=0|x) = \frac{0.30}{49.92} = 0.006 \approx 0.01$$

$$P(y=1|x) = \frac{1.65}{49.92} = 0.033 \approx 0.03$$

$$P(y=2|x) = \frac{44.70}{49.92} = 0.895 \approx 0.90$$

$$P(y=3|x) = \frac{2.72}{49.92} = 0.054 \approx 0.05$$

$$P(y=4|x) = \frac{0.55}{49.92} = 0.011 \approx 0.01$$

Check: $0.01 + 0.03 + 0.90 + 0.05 + 0.01 = 1.00$

Model Output Summary

Class	Score z_k	Probability $P(y = k\|x)$	Confidence
Normal (0)	-1.2	0.01	Very low
Inner race (1)	0.5	0.03	Low
Outer race (2)	**3.8**	**0.90**	**Very high**
Ball (3)	1.0	0.05	Low
Cage (4)	-0.6	0.01	Very low

Prediction: $\hat{y} = \arg\max_k P(y = k|x) = 2$ (outer race fault)

Confidence: 90% probability $\rightarrow$ High confidence prediction

Decision: Schedule maintenance for outer race replacement

Implementation Demo

Dataset: Case Western Reserve University (CWRU) bearing dataset

- Freely available benchmark dataset
- Vibration data for normal and faulty bearings
- Multiple fault types, severities, and load conditions

Complete implementation: See Jupyter notebook
`bearing_fault_classification.ipynb`

Logistic Regression: Advantages

1. Probabilistic outputs

- Not just predictions, but confidence: $P(y = 1|x) = 0.95$ vs 0.51
- Critical for aerospace: "How certain are we this component will fail?"
- Enables cost-sensitive decision making

2. Interpretable coefficients

- $\beta_j > 0$: increasing feature x_j increases probability of positive class
- Example: $\beta_{\text{vibration}} = 2.3 \rightarrow$ strong positive indicator of fault
- Explainable to engineers and regulators

3. Computationally efficient

- Training: $O(nd)$ with gradient descent
- Prediction: $O(d)$ per sample (just matrix-vector multiply + sigmoid)
- Scales well to large datasets

Logistic Regression: Advantages (contd.)

4. Convex optimization

- Single global optimum (no local minima)
- Guaranteed convergence with gradient descent
- Reliable, reproducible results

5. Well-calibrated probabilities

- $P(y = 1|x) = 0.7 \rightarrow$ roughly 70% of such predictions are correct
- Important for safety-critical aerospace applications
- Can be further calibrated if needed (Platt scaling)

6. Works well with regularization

- L_1 (Lasso): Feature selection, sparse models
- L_2 (Ridge): Handles correlated features
- Both prevent overfitting

Logistic Regression: Limitations

1. **Linear decision boundary**
   - Can only separate classes with hyperplane: $\beta^T x = 0$
   - Cannot learn XOR, circular boundaries, etc.
   - Example: Cannot separate two overlapping circular clusters
2. **Requires feature engineering for complex patterns**
   - Need to manually create: x^2, $x_1 x_2$, $\sin(x)$, etc.
   - Time-consuming and domain knowledge intensive
   - May miss important nonlinear relationships
3. **Sensitive to outliers** (without regularization)
   - Extreme feature values can dominate
   - May need preprocessing: standardization, outlier removal

Logistic Regression: Limitations (contd.)

4. **Assumes feature independence** (naive interpretation)

- Correlated features can make coefficients unstable
- Multicollinearity affects interpretability
- Solution: Use L_2 regularization or remove correlated features

5. **May underfit complex data**

- Real-world aerospace problems often highly nonlinear
- Temperature-pressure interactions, aerodynamic effects
- May need more expressive models (neural networks, trees)

When to use logistic regression?

- Baseline model (always try first!)
- When interpretability is critical
- Linear or near-linear decision boundaries
- Limited training data (less overfitting risk)

Additional Algorithms

Overview of Classification Methods

Beyond Logistic Regression:

- **K-Nearest Neighbors (KNN)**: Instance-based, non-parametric
- **Decision Trees**: Rule-based, interpretable
- **Random Forest**: Ensemble of trees, robust
- **Support Vector Machines (SVM)**: Maximum margin classification
- **Neural Networks**: Universal approximators, black-box

Each has different strengths for aerospace applications

K-Nearest Neighbors (KNN)

Idea: Classify based on k nearest training samples

Algorithm:

1. Given new point x, find k nearest neighbors in training set
2. Majority vote: $\hat{y} = \text{mode}(\{y_{i_1}, \ldots, y_{i_k}\})$
3. For probabilities: $P(y = c|x) = \frac{1}{k} \sum_{j=1}^{k} \mathbf{1}[y_{i_j} = c]$

Distance metric (typically Euclidean):

$$d(x, x') = \sqrt{\sum_{j=1}^{d} (x_j - x'_j)^2}$$

Hyperparameter: k (number of neighbors)

- Small k: Flexible but noisy (high variance)
- Large k: Smooth but biased (high bias)

KNN: Aerospace Example

Flight phase classification: New flight data point

- Features: altitude = 35,000 ft, speed = 450 knots, vertical rate = 0 ft/min
- Find $k = 5$ nearest flights in training data
- Neighbors: 4 cruise, 1 descent
- Prediction: cruise (majority vote)
- Probability: $P(\text{cruise}) = 4/5 = 0.80$

Pros:

- No training phase (just store data)
- Naturally handles multiclass problems
- Intuitive and simple

Cons:

- Slow at prediction time (search through all training data)
- Sensitive to feature scaling (need standardization)
- Curse of dimensionality

Decision Trees

Idea: Recursively split feature space into regions

Structure: Binary tree where each node tests a feature

- **Internal node**: $x_j < \theta$ (feature j, threshold θ)
- **Leaf node**: Class prediction (majority class in region)

Example rule:

```
IF altitude > 30,000 ft THEN
  IF vertical_rate > 100 ft/min THEN
    class = ascent
  ELSE
    class = cruise
ELSE
  class = low_altitude
```

Decision Trees: Training

Goal: Find splits that maximize class separation

Gini impurity (measure of class mixing):

$$G = 1 - \sum_{k=1}^{K} p_k^2$$

where p_k = fraction of samples in class k

Algorithm (greedy, recursive):

1. For each feature j and threshold θ:
   - Split data into left ($x_j < \theta$) and right ($x_j \geq \theta$)
   - Compute weighted Gini: $G_{\text{split}} = \frac{n_L}{n} G_L + \frac{n_R}{n} G_R$
2. Choose split with lowest G_{split}
3. Recurse on left and right subsets
4. Stop when: max depth reached, min samples, or pure node

Decision Trees: Pros and Cons

Advantages:

- **Interpretable**: Can visualize and explain rules
- **Non-linear**: Captures complex decision boundaries
- **Mixed features**: Handles categorical and numerical
- **Feature importance**: Identifies key sensors
- **No scaling needed**: Invariant to monotonic transformations

Limitations:

- **Overfitting**: Deep trees memorize training data
- **Unstable**: Small data changes $\rightarrow$ completely different tree
- **Greedy**: Locally optimal splits, not globally optimal
- **Biased**: Favors features with many values

Solution to instability: Use ensemble methods (Random Forest)

Random Forest

Idea: Train many decision trees, average their predictions

Algorithm:

1. For $b = 1, \ldots, B$ (e.g., $B = 100$ trees):
   - **Bootstrap sample**: Randomly sample n points with replacement
   - **Random features**: At each split, consider only $m < d$ random features
   - Train decision tree on this sample
2. **Prediction**:
   - Classification: Majority vote across all trees
   - Probabilities: Average probabilities from all trees

$$P(y = k|x) = \frac{1}{B} \sum_{b=1}^{B} P_b(y = k|x)$$

Random Forest: Key Properties

Why it works:

- **Bagging** (bootstrap aggregating): Reduces variance by averaging
- **Random features**: Decorrelates trees (prevents all trees being similar)
- **Law of large numbers**: Average of many noisy predictors $\rightarrow$ stable

Feature importance:

$$\text{Importance}_j = \frac{1}{B}\sum_{b=1}^{B}\sum_{\text{nodes using } x_j} \Delta G$$

Measures total Gini reduction from feature j across all trees

Aerospace use:

- Identify most critical sensors for fault detection
- Rank features: temperature, pressure, vibration, etc.

Random Forest: Pros and Cons

Advantages:

- **Accurate**: Often best off-the-shelf performance
- **Robust**: Handles outliers, missing data, noisy features
- **No overfitting**: Adding more trees never hurts (just plateaus)
- **Feature importance**: Quantifies sensor relevance
- **Parallel**: Each tree trained independently

Limitations:

- **Less interpretable**: Ensemble of 100+ trees is a black box
- **Memory**: Must store all trees
- **Slower prediction**: Query many trees vs one logistic regression

Aerospace: Excellent for fault detection with many sensors when interpretability is secondary to accuracy

Support Vector Machines (SVM)

Idea: Find hyperplane that maximally separates classes

Linear SVM: Find β, β_0 such that:

$$y_i(\beta^T x_i + \beta_0) \geq 1 \quad \forall i$$

Objective: Maximize margin $M = \frac{2}{\|\beta\|}$

Equivalently, minimize:

$$\min_{\beta,\beta_0} \frac{1}{2}\|\beta\|^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i(\beta^T x_i + \beta_0))$$

C: Regularization (tradeoff between margin and misclassifications)

- Large C: Hard margin (fewer errors, may overfit)
- Small C: Soft margin (allows errors, more robust)

SVM: The Kernel Trick

Problem: Linear boundary insufficient for complex data

Solution: Map to higher-dimensional space where data is separable

$$x \to \phi(x) \quad \text{(e.g., } [x_1, x_2] \to [x_1, x_2, x_1^2, x_2^2, x_1 x_2])$$

Kernel trick: Compute $K(x, x') = \phi(x)^T \phi(x')$ without explicit ϕ

Common kernels:

- **Linear**: $K(x, x') = x^T x'$
- **Polynomial**: $K(x, x') = (x^T x' + 1)^p$
- **RBF (Gaussian)**: $K(x, x') = \exp(-\gamma \|x - x'\|^2)$

RBF most popular: can approximate any boundary with proper γ

SVM: Aerospace Applications

Pros:

- **High-dimensional**: Excellent when $d \gg n$ (many sensors, few samples)
- **Robust**: Only depends on support vectors (points near boundary)
- **Theoretically principled**: Maximum margin has good generalization
- **Kernel flexibility**: Adapts to data complexity

Cons:

- **Slow training**: $O(n^2)$ to $O(n^3)$ for kernel SVM
- **Hyperparameter tuning**: Need to select C, kernel type, kernel parameters
- **No direct probabilities**: Need Platt scaling for calibrated probabilities
- **Binary focus**: Multiclass requires one-vs-one or one-vs-all

Use case: Anomaly detection with limited labeled data

Neural Networks (Brief Overview)

Feedforward Neural Network: Layers of weighted transformations

$$
\begin{aligned}
h^{(1)} &= \sigma(W^{(1)}x + b^{(1)}) \\
h^{(2)} &= \sigma(W^{(2)}h^{(1)} + b^{(2)}) \\
&\vdots \\
\hat{y} &= \text{softmax}(W^{(L)}h^{(L-1)} + b^{(L)})
\end{aligned}
$$

Pros: Universal approximators, learn complex patterns, handle high-dimensional data

Cons: Black box, requires large data, computationally expensive, many hyperparameters

Aerospace: Image-based classification (damage detection, satellite imagery), time-series (LSTMs for flight trajectories)

Note: Neural networks covered in depth in later lectures

Algorithm Selection Guide

Algorithm	Interpretability	Training Time	Prediction Time	Non-linear	Data Required
Logistic Regression	High	Fast	Fast	No	Low
KNN	Medium	None	Slow	Yes	Medium
Decision Tree	High	Fast	Fast	Yes	Medium
Random Forest	Low	Medium	Medium	Yes	Medium
SVM	Low	Slow	Fast	Yes (kernel)	Low-Medium
Neural Network	Very Low	Slow	Fast	Yes	High

Current Trends in Aerospace Classification

Deep Learning Classifiers:

- CNNs for aircraft damage classification (crack/no-crack detection)
- LSTMs/Transformers for flight phase classification from trajectories
- Vision Transformers for multi-class defect classification

Explainable AI (XAI) for Classification:

- SHAP values for fault classification (which sensor drove the fault prediction?)
- LIME for explaining binary failure/healthy classifications
- Critical for FAA/EASA certification of classification systems

Current Trends in Aerospace Classification (contd.)

Automated Machine Learning (AutoML):

- H2O.ai, Auto-sklearn for automated classifier selection and tuning
- Neural Architecture Search (NAS) for optimal classification architectures
- Reduces ML expertise needed for building fault classifiers

Advanced Imbalanced Classification Methods:

- Aerospace faults are rare (1-5% of data) $\rightarrow$ severe class imbalance
- Focal Loss for neural network classifiers (focuses on hard-to-classify examples)
- One-Class SVM and Isolation Forest for binary anomaly classification

Ensemble Methods:

- Stacking multiple classifiers (e.g., Random Forest + SVM + Neural Network)
- Voting classifiers for robust fault detection
- Improves classification accuracy and reduces false negatives

DRAFT Slides

Evaluation Metrics for Imbalanced Data

Imbalanced Data Challenge: Standard accuracy can be misleading when classes are imbalanced. For example, if 99% of samples are negative, a classifier that always predicts negative achieves 99% accuracy but is useless.

Key Metrics:

- **Precision**: How many predicted positives are actually positive?
- **Recall (Sensitivity)**: How many actual positives are correctly predicted?
- **F1 Score**: Harmonic mean of Precision and Recall
- **ROC AUC**: Area under the ROC curve
- **PR AUC**: Area under the Precision-Recall curve

Definitions

- **True Positive (TP)**: Correctly classified positive samples
  - Actual = Positive, Predicted = Positive
- **True Negative (TN)**: Correctly classified negative samples
  - Actual = Negative, Predicted = Negative
- **False Positive (FP)**: Negative samples incorrectly classified as positive
  - Actual = Negative, Predicted = Positive (Type I error)
- **False Negative (FN)**: Positive samples incorrectly classified as negative
  - Actual = Positive, Predicted = Negative (Type II error)

Confusion Matrix:

	Predicted: Neg	Predicted: Pos
Actual: Neg	TN	FP
Actual: Pos	FN	TP

ROC Curve: Evaluation Across All Thresholds

ROC (Receiver Operating Characteristic): Visualizes classifier performance at every possible threshold

Key Metrics:

$$\text{TPR (True Positive Rate)} = \frac{TP}{TP + FN} = \text{Recall (Sensitivity)}$$

$$\text{FPR (False Positive Rate)} = \frac{FP}{FP + TN}$$

How to construct ROC:

1. **Vary threshold** τ from 0 to 1 (0 = predict all positive, 1 = predict all negative)
2. **At each** τ: Compute TPR and FPR
3. **Plot**: (FPR, TPR) pairs $\rightarrow$ creates ROC curve

ROC Curve: Interpretation

AUC (Area Under ROC Curve):

- **AUC = 0.5**: Random classifier (diagonal line from (0,0) to (1,1))
- **AUC = 1.0**: Perfect classifier (goes through point (0,1) = 100% TPR, 0% FPR)
- **AUC (0.5, 1.0)**: Better than random, higher = better class separation

Probabilistic Interpretation:

AUC = Probability that classifier ranks a randomly chosen positive sample higher than a randomly chosen negative sample

When to use ROC:

- Balanced classes (roughly equal number of positives and negatives)
- Both false positives and false negatives are equally costly
- Want to see performance across all possible operating points

Precision-Recall Curve: Definition

PR Curve: Plot Precision vs Recall at all thresholds

$$\text{Precision} = \frac{TP}{TP + FP} \quad \text{(How many predicted positives are correct?)}$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad \text{(How many actual positives were found?)}$$

AP (Average Precision): Area under PR curve (summary metric)

How to construct PR curve:

1. **Vary threshold** τ from 0 to 1
2. **At each** τ: Compute Precision and Recall
3. **Plot**: (Recall, Precision) pairs $\rightarrow$ creates PR curve

Key difference from ROC: PR focuses entirely on positive class performance (ignores TN)

When to Use ROC vs PR Curve?

ROC vs PR Comparison:

Scenario	Prefer	Reason
Balanced classes (50/50 split)	ROC	Both metrics equally informative
Imbalanced (rare positives)	**PR**	Focuses on minority class performance
False positives not critical	ROC	TN dominates, inflates specificity
False positives are costly	PR	Precision directly penalizes FP
Need to see all operating points	ROC	Standard in ML literature
Aerospace fault detection	**PR**	Faults are rare (1-5% of data)

Why PR for Aerospace Faults?

Aerospace Context: Faults are rare events (1-5% of data)

Example: 1000 samples, 10 actual faults (1% positive class)

Classifier predicts 20 positives: 8 TP, 12 FP, 2 FN

- **Precision** $= 8/20 =$ **40%** (only 40% of alarms are real faults)
- **Recall** $= 8/10 =$ **80%** (caught 80% of actual faults)
- **FPR** $= 12/990 =$ **1.2%** (looks excellent! Low false positive rate)

Problem with ROC: Large TN (978) makes FPR look great, hides poor precision

PR Curve reveals: 40% precision means 60% of maintenance actions are unnecessary

Conclusion: For imbalanced aerospace data, PR curve gives realistic view of classifier performance

Precision-Recall Curve

PR Curve: Plot Precision vs Recall at all thresholds

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

AP (Average Precision): Area under PR curve

When to use which?

Scenario	Use	Reason
Balanced classes	ROC	Both classes equally important
Imbalanced classes	PR	Focuses on minority (positive) class
Aerospace faults (rare)	**PR**	Don't want FP to inflate metrics

Example: 1000 samples, 10 faults (1%)

Imbalanced Data

The Imbalanced Classification Problem

Definition: One class has significantly more samples than others

Class imbalance ratio: $\pi_+ = \frac{n_+}{n}$ where $n_+ =$ minority class samples

Aerospace examples:

- Fault detection: 1-2% faulty, 98-99% normal ($\pi_+ \approx 0.01$)
- Engine failure: 0.1% failures, 99.9% normal ($\pi_+ \approx 0.001$)
- Anomaly detection: 0.5-5% anomalies ($\pi_+ \approx 0.01$)

Why it matters: Standard classifiers optimize overall accuracy $\rightarrow$ ignore minority class

Why Standard Classifiers Fail

The accuracy trap: Predicting majority class gives high accuracy

Example: 1000 flights, 10 faults (1% faults)

Naive classifier: Always predict "no fault"

- Accuracy = 990/1000 = 99% (looks great!)
- But recall = 0% (detected zero faults!)

Problem: Gradient descent minimizes overall error

- Majority class errors dominate the loss
- Model learns to predict majority class
- Minority class ignored

Result: Effective decision threshold shifts away from minority class

Approaches to Handle Imbalanced Data

Three main strategies:

1. **Resampling**: Change the class distribution
   - Oversample minority class (SMOTE)
   - Undersample majority class
2. **Algorithm-level**: Modify the learning algorithm
   - Class weights in loss function
   - Cost-sensitive learning
3. **Threshold tuning**: Adjust decision boundary
   - Find optimal threshold on validation set
   - Use cost-sensitive threshold

We'll cover each in detail

SMOTE: Synthetic Minority Over-sampling

Problem: Not enough minority samples → oversample

Naive approach: Duplicate minority samples

- Creates exact copies → overfitting
- No new information added

SMOTE (Synthetic Minority Over-sampling Technique): Create synthetic samples

Algorithm: For each minority sample x_i:

1. Find k nearest minority neighbors (typically $k = 5$)
2. Randomly select one neighbor x_j
3. Create synthetic sample: $x_{new} = x_i + \lambda(x_j - x_i)$

where $\lambda \sim U(0, 1)$ (random value between 0 and 1)

SMOTE: Geometric Interpretation

What SMOTE does: Interpolates between minority samples

Example: Two minority samples

- $x_i = [2.0, 3.5]$ (fault sample 1)
- $x_j = [3.0, 4.0]$ (fault sample 2, nearest neighbor)
- $\lambda = 0.6$ (random)

Synthetic sample:

$$\begin{aligned} x_{new} &= [2.0, 3.5] + 0.6([3.0, 4.0] - [2.0, 3.5]) \\ &= [2.0, 3.5] + 0.6([1.0, 0.5]) \\ &= [2.0, 3.5] + [0.6, 0.3] = [2.6, 3.8] \end{aligned}$$

Result: New sample lies on line segment between x_i and x_j

Benefit: Creates diverse samples, expands minority class region

SMOTE: When to Use

Advantages:

- Generates new information (not duplicates)
- Works well for many aerospace applications
- Easy to implement: `imblearn.over_sampling.SMOTE`

Limitations:

- Can create noisy samples in overlapping regions
- Computational cost increases with dataset size
- May not work well in very high dimensions

When to use:

- Small minority class ($n_+ < 100$ samples)
- Clear separation between classes
- Low-to-medium dimensional features

Implementation:

Class Weights: Algorithm-Level Solution

Idea: Make minority errors more expensive during training

Standard loss (equal weight):

$$J = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log p_i + (1-y_i)\log(1-p_i)\right]$$

Weighted loss:

$$J = -\frac{1}{n}\sum_{i=1}^{n} w_{y_i}\left[y_i \log p_i + (1-y_i)\log(1-p_i)\right]$$

Class weight w_k for class k:

- Higher weight $\rightarrow$ larger contribution to loss
- Minority class gets higher weight
- Forces model to pay attention to rare class

Class Weights: Balanced Weighting

Balanced weights formula:

$$w_k = \frac{n}{K \cdot n_k}$$

where: - n = total samples - K = number of classes - n_k = samples in class k

Example: 1000 samples, 2 classes, 10 faults, 990 normal

- $w_{\text{fault}} = \frac{1000}{2 \times 10} = 50$
- $w_{\text{normal}} = \frac{1000}{2 \times 990} = 0.505$

Effect: Fault misclassification costs 100× more than normal misclassification

Class Weights: Implementation

Scikit-learn integration:

```
from sklearn.linear_model import LogisticRegression

# Automatic balanced weights
model = LogisticRegression(class_weight='balanced')
model.fit(X_train, y_train)
```

Custom weights:

```
# Manual control over weights
weights = {0: 1.0, 1: 50.0}  # Class 0: normal, Class 1: fault
model = LogisticRegression(class_weight=weights)
model.fit(X_train, y_train)
```

Class Weights: When to Use

When to use: Always for imbalanced classification (simplest solution)

Works with:

- Logistic Regression: `class_weight='balanced'`
- SVM: `class_weight='balanced'`
- Random Forest: `class_weight='balanced'`
- Neural Networks: Custom loss weighting

Advantages:

- No data modification required
- Works during training
- Easy to implement

Tip: Start with balanced weights, then fine-tune if needed

Threshold Tuning: Post-Training Solution

Recall: Logistic regression predicts $\hat{y} = 1$ if $P(y = 1|x) > \tau$

Default: $\tau = 0.5$ (neutral threshold)

Problem with imbalance: Optimal threshold is NOT 0.5

Solution: Find optimal τ on validation set

Approach 1 - Maximize F_β score:

$$\tau^* = \arg\max_\tau F_\beta(\tau)$$

Approach 2 - Cost-sensitive:

$$\tau^* = \arg\min_\tau \left[C_{FN} \cdot FN(\tau) + C_{FP} \cdot FP(\tau)\right]$$

where C_{FN} = cost of false negative, C_{FP} = cost of false positive

Threshold Tuning: Example

Scenario: Bearing fault detection

- 1000 test samples, 20 actual faults
- Cost of missing fault: $C_{FN} = \$100{,}000$ (engine failure)
- Cost of false alarm: $C_{FP} = \$1{,}000$ (unnecessary inspection)

Test different thresholds:

τ	TP	FP	FN	Cost
0.3	18	150	2	\$200k + \$150k = \$350k
0.4	17	80	3	\$300k + \$80k = \$380k
0.5	15	30	5	\$500k + \$30k = \$530k

Optimal: $\tau^* = 0.3$ (lowest cost)

Trade-off: More false alarms, but fewer missed faults

Threshold Tuning: Process

Step-by-step approach:

1. Train classifier on training set
2. Get predicted probabilities on validation set
3. Try thresholds from 0 to 1 (e.g., 0.1, 0.2, ..., 0.9)
4. For each threshold, compute metric (F_1, precision, recall, cost)
5. Select threshold that maximizes/minimizes objective

Key insight: Separate training from threshold selection

Validation set: Essential for finding optimal threshold

Threshold Tuning: Code Implementation

Code example:

```
from sklearn.metrics import f1_score

best_f1, best_tau = 0, 0.5
for tau in np.arange(0.1, 1.0, 0.05):
    y_pred = (y_proba >= tau).astype(int)
    f1 = f1_score(y_val, y_pred)
    if f1 > best_f1:
        best_f1, best_tau = f1, tau

# Use best_tau for final predictions
y_test_pred = (y_test_proba >= best_tau).astype(int)
```

Result: Threshold optimized for validation performance

Comparing Imbalanced Data Solutions

Method	Pros	Cons	When to Use
SMOTE	Creates diverse samples	Noisy in overlap regions	Small minority class
Class Weights	Simple, no data change	May not help extreme imbalance	First try, works universally
Threshold Tuning	Fine control, post-hoc	Needs validation set	Cost-sensitive problems

Key insight: Each method addresses imbalance differently

- **SMOTE**: Changes the data distribution
- **Class Weights**: Changes the learning objective
- **Threshold Tuning**: Changes the decision rule

Best Practices for Imbalanced Data

General approach: Combine multiple methods

1. **Start**: Use class weights during training
2. **If needed**: Apply SMOTE for very small minority class
3. **Final step**: Fine-tune threshold based on costs

Why combine?

- Addresses problem at multiple levels
- Robust to different imbalance ratios
- Maximizes performance

Aerospace-specific recommendations:

Safety-critical systems:

- Always use class weights
- Add threshold tuning based on failure costs
- Monitor on validation set continuously

Cost considerations:

- $C_{FN} \gg C_{FP}$: Lower threshold (favor recall)
- Use PR curves over ROC curves
- Report precision and recall separately

Validation

Why Do We Need Validation?

The Fundamental Problem: Training accuracy doesn't predict real-world performance

What we want to know: "How will my classifier perform on new, unseen data?"

What we have: Only our training dataset

The danger of overfitting:

- Complex models can memorize training data (100% training accuracy!)
- But fail on new data (poor generalization)
- Example: Decision tree with 1 sample per leaf $\rightarrow$ perfect training, terrible testing

Solution: Simulate "unseen data" by holding out part of our dataset

Goal of validation: Get honest estimate of generalization performance before deploying to production

Train-Test Split: The Simplest Approach

Idea: Split dataset into two disjoint sets

- **Training set** (typically 70-80%): Used to fit model parameters β
- **Test set** (20-30%): Held out, never seen during training

Example: 1000 flight samples $\rightarrow$ 800 train, 200 test

Process:

1. Train classifier on 800 samples
2. Evaluate on 200 test samples
3. Report test accuracy/precision/recall as performance estimate

Critical rule: NEVER use test set for any training decisions (hyperparameters, feature selection, etc.)

Train-Test Split: Best Practices

Random shuffling: Randomly assign samples to train/test

- Avoids bias from data collection order
- Use fixed random seed for reproducibility

Stratification: Maintain class proportions

- If 10% samples are faults in full dataset
- Keep 10% faults in both train and test
- Essential for imbalanced classification

When to use:

- Large datasets ($n > 10{,}000$): Single split is reliable
- Quick model prototyping: Fast, simple
- Final model evaluation: After all development is done

Temporal Splits for Time-Series Data

Problem with random split: Data leakage from future to past

Aerospace example: Flight trajectory classification

- Sample at 10:00 AM and 10:05 AM are highly correlated
- Random split might put them in train and test $\rightarrow$ overly optimistic results

Temporal split: Train on past, test on future

- Train: All flights before January 1, 2025
- Test: All flights after January 1, 2025
- Simulates real deployment: model trained on historical data, used on future data

Rule: For time-series, always split by time, never randomly

k-Fold Cross-Validation: Better Use of Data

Problem with single train-test split: Performance estimate depends on which samples ended up in test set

Solution: Average over multiple train-test splits

k-Fold CV Algorithm:

1. **Partition** dataset into k equal-sized folds (typically $k = 5$ or $k = 10$)
2. **For each fold** $i = 1, \ldots, k$:
   - Use fold i as test set
   - Use remaining $k - 1$ folds as training set
   - Train model and compute performance metric S_i (e.g., accuracy)
3. **Average** across all folds: CV score $= \frac{1}{k} \sum_{i=1}^{k} S_i$

Benefit: Every sample is used for testing exactly once $\rightarrow$ more reliable estimate

k-Fold Cross-Validation: Example

Dataset: 1000 samples, 5-fold CV

Fold structure:

Fold	Samples	Role in iteration
1	1-200	Test in iter 1, train in iters 2-5
2	201-400	Test in iter 2, train in iters 1,3-5
3	401-600	Test in iter 3, train in iters 1-2,4-5
4	601-800	Test in iter 4, train in iters 1-3,5
5	801-1000	Test in iter 5, train in iters 1-4

Results: Accuracies = [0.82, 0.85, 0.83, 0.84, 0.81]

CV score: $(0.82 + 0.85 + 0.83 + 0.84 + 0.81)/5 = 0.83$

Standard deviation: 0.015 $\rightarrow$ measure of stability across folds

Choosing k: Bias-Variance Tradeoff

Small k (e.g., $k = 3$):

- Each fold: train on 67% of data
- Fewer training samples $\rightarrow$ underestimates true performance (high bias)
- Faster computation (only 3 iterations)

Large k (e.g., $k = 10$ or $k = 20$):

- Each fold: train on 90% or 95% of data
- More representative of full-data performance (low bias)
- Higher variance (folds more correlated)
- Slower computation

Leave-One-Out CV ($k = n$): Train on $n - 1$ samples, test on 1

- Lowest bias, highest variance
- Extremely slow for large n

Stratified k-Fold: Essential for Imbalanced Data

Problem: Random fold assignment can create unbalanced folds

Example: 1000 samples, 50 faults (5%), $k = 5$ folds (200 samples each)

- Random assignment: [8, 12, 10, 11, 9] faults per fold
- Worst case: [2, 3, 15, 20, 10] $\rightarrow$ highly variable estimates

Stratified k-Fold: Maintain class proportions in every fold

- Each fold: exactly 10 faults + 190 normal (5% ratio)
- Formula: $\frac{n_+^{(i)}}{n^{(i)}} = \frac{n_+}{n}$ for all folds i

When to use: Always for imbalanced classification

Implementation: `sklearn.model_selection.StratifiedKFold`

Hyperparameter Selection: The Wrong Way

Common mistake: Use cross-validation to tune hyperparameters, then report CV score as performance

Example: Logistic regression with regularization λ

```
best_score = 0
for lambda in [0.001, 0.01, 0.1, 1.0, 10.0]:
    score = cross_val_score(LogisticRegression(C=1/lambda), X, y, cv=5).mea
    if score > best_score:
        best_score = score
        best_lambda = lambda

print(f"Performance: {best_score}")  # WRONG! This is overly optimistic
```

Why it's wrong: We "peeked" at test folds multiple times (once per λ)

- Test folds influenced hyperparameter choice

Hyperparameter Selection: The Right Way

Solution: Nested cross-validation (CV within CV)

Structure:

- **Outer loop** (evaluation): Unbiased performance estimate
- **Inner loop** (tuning): Hyperparameter selection

Why nested?

- Inner CV: Finds best hyperparameter for each training set
- Outer CV: Evaluates generalization of entire process
- Never test on same data used for hyperparameter selection

Key principle: Treat hyperparameter selection as part of the training process

Nested CV: Algorithm

Step-by-step process:

1. **Outer CV**: Split data into k_{outer} folds (e.g., 5)
2. **For each outer fold** i:
   - Hold out fold i as final test set
   - Use remaining folds for hyperparameter tuning
   - **Inner CV**: k_{inner}-fold CV (e.g., 3-fold) on remaining folds
   - Select best hyperparameter λ_i^* based on inner CV
   - Train model with λ_i^* on all remaining folds
   - Evaluate on outer fold $i \rightarrow$ score S_i
3. **Report**: Average outer CV score $\frac{1}{k_{\text{outer}}} \sum_i S_i$

Nested CV: Example Setup

Dataset: 1000 samples

Hyperparameter search: Regularization $\lambda \in \{0.01, 0.1, 1.0, 10.0\}$

Configuration:

- Outer CV: 5 folds (200 samples each)
- Inner CV: 3 folds (on 800 samples)

What we'll do: Walk through outer fold 1

Nested CV: Iteration 1 Walkthrough

Step 1 - Outer split:

- Fold 1 (200 samples) $\rightarrow$ held out as final test
- Folds 2-5 (800 samples) $\rightarrow$ available for training

Step 2 - Inner CV for hyperparameter tuning:

- Run 3-fold CV on 800 samples
- Test each λ: [0.01 $\rightarrow$ 0.78, 0.1 $\rightarrow$ 0.82, 1.0 $\rightarrow$ 0.84, 10.0 $\rightarrow$ 0.79]
- Best: $\lambda_1^* = 1.0$ (highest inner CV score)

Step 3 - Final training and evaluation:

- Train with $\lambda_1^* = 1.0$ on all 800 samples (folds 2-5)
- Evaluate on fold 1 $\rightarrow S_1 = 0.83$

Nested CV: Complete Results

Repeat for all 5 outer folds:

Outer Fold	Best λ^* (from inner CV)	Outer Score S_i
1	1.0	0.83
2	0.1	0.81
3	1.0	0.85
4	1.0	0.82
5	0.1	0.84

Final performance: $\frac{1}{5}(0.83 + 0.81 + 0.85 + 0.82 + 0.84) = 0.83$

Computational cost: $5 \times 3 \times 4 = 60$ inner fits + 5 outer fits = 65 trainings

Validation Strategies: Summary

Method	Use Case	Pros	Cons
Train-Test Split	Large datasets, quick prototyping	Fast, simple	High variance, wastes data
Temporal Split	Time-series data	No data leakage	Single split, may not be representative
k-Fold CV	Medium datasets, reliable estimate	Low variance, uses all data	Slower than single split
Stratified k-Fold	Imbalanced classes	Maintains class balance	Slightly more complex
Nested CV	Hyperparameter tuning + evaluation	Unbiased estimate	Computationally expensive

Aerospace recommendation: Stratified k-fold for model selection, nested CV for final