Hands-On Machine Learning
Chapter 3: Classification
Study Notes

Personal Learning Notes

December 25, 2025

Contents

1 Introduction to Classification

Classification is a supervised learning task where the goal is to predict discrete categories or classes. Unlike regression which predicts continuous values, classification assigns instances to predefined categories.

1.1 Binary vs Multiclass Classification

- **Binary Classification**: Two possible classes (e.g., spam vs not spam, 5 vs not-5)
- **Multiclass Classification**: More than two classes (e.g., digit 0-9 classification)
- **Multilabel Classification**: Multiple labels per instance
- **Multioutput Classification**: Multiple outputs, each with multiple classes

2 SGD Classifier - Deep Dive

2.1 What is SGD?

Stochastic Gradient Descent (SGD) is an optimization algorithm used to train linear models. The `SGDClassifier` in scikit-learn uses SGD to learn a linear decision boundary.

2.2 The Linear Model

The core prediction function is:

$$\hat{y} = \text{sign}(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \tag{1}$$

Where:

- $\mathbf{w}$ = weight vector (one weight per feature)
- $\mathbf{x}$ = input feature vector
- b = bias term
- sign() = returns +1 or -1 for binary classification

Goal: Find optimal weights $\mathbf{w}$ and bias b that best separate the classes.

2.3 Loss Function

The default loss function is the **hinge loss** (same as Support Vector Machines):

$$L = \max(0, 1 - y \times (\mathbf{w} \cdot \mathbf{x} + b)) \tag{2}$$

Where $y \in \{-1, +1\}$ is the true label.

Hinge Loss Interpretation

- If prediction is correct and confident: loss = 0
- If prediction is wrong or not confident enough: loss increases linearly
- Encourages a margin of safety around the decision boundary

2.4 Gradient Descent vs Stochastic Gradient Descent

2.4.1 Regular Gradient Descent

1. Calculate loss on **ALL** training examples
2. Compute average gradient
3. Update weights: $\mathbf{w} = \mathbf{w} - \eta \nabla L$
4. Repeat

Problem: Slow on large datasets (e.g., 60,000 MNIST images)!

2.4.2 Stochastic Gradient Descent

1. Pick **ONE** random training example
2. Calculate loss on just that example
3. Compute gradient from that single example
4. Update weights immediately
5. Repeat with next random example

Advantages:

- Much faster - updates after each example
- Scales well to large datasets
- Random noise helps escape local minima

2.5 The SGD Algorithm

SGD Algorithm Step-by-Step

Initialize: $\mathbf{w}$ = small random values, $b = 0$
For each epoch (pass through dataset):

1. Shuffle training data randomly
2. **For each** training example $(\mathbf{x}_i, y_i)$:
   (a) Make prediction: $\hat{y} = \mathbf{w} \cdot \mathbf{x}_i + b$
   (b) Calculate loss: $L = \max(0, 1 - y_i \times \hat{y})$
   (c) Calculate gradient:

$$\text{If } L > 0: \quad \frac{\partial L}{\partial \mathbf{w}} = -y_i \mathbf{x}_i, \quad \frac{\partial L}{\partial b} = -y_i$$
$$\text{Else}: \quad \frac{\partial L}{\partial \mathbf{w}} = 0, \quad \frac{\partial L}{\partial b} = 0$$

   (d) Update weights:

$$\mathbf{w} = \mathbf{w} - \eta \left(\frac{\partial L}{\partial \mathbf{w}} + \alpha \mathbf{w} \right)$$
$$b = b - \eta \frac{\partial L}{\partial b}$$

Where η is the learning rate and α is the regularization parameter.

2.6 Why "Stochastic"?

The term **stochastic** means "random". The randomness comes from:

- Randomly shuffling data each epoch
- Processing examples one at a time in random order
- Each weight update based on a single random sample

This introduces noise in the gradient, but provides benefits in speed and convergence.

2.7 Key Hyperparameters

`loss` Type of loss function (default: `'hinge'` for SVM)

- `'hinge'`: Linear SVM
- `'log_loss'`: Logistic regression
- `'perceptron'`: Perceptron algorithm

`learning_rate` Controls step size for weight updates

- Too high → overshoots, unstable
- Too low → slow convergence
- Scikit-learn uses adaptive learning rates by default

`max_iter` Number of epochs (passes through dataset)

- More iterations = more learning
- Risk of overfitting with too many

`alpha` Regularization strength

- Adds penalty: $\alpha||\mathbf{w}||^2$
- Prevents overfitting by keeping weights small
- Higher α = stronger regularization

`random_state` Controls shuffling randomness for reproducibility

2.8 SGD Variants

- **Batch Gradient Descent**: Uses all samples per update (slow but stable)
- **Stochastic Gradient Descent**: Uses 1 sample per update (fast but noisy)
- **Mini-batch Gradient Descent**: Uses small batches (e.g., 32 samples) per update (good balance)

Scikit-learn's `SGDClassifier` uses pure SGD (batch size = 1).

2.9 Why SGD for MNIST?

- 60,000 training images with 784 features each (28×28 pixels)
- Regular gradient descent would be computationally expensive
- SGD updates quickly and scales well to large datasets
- Linear model is fast to train and predict

3 Performance Measures

3.1 Cross-Validation

Cross-Validation is a technique to evaluate model performance by splitting the dataset into multiple folds and training/testing on different combinations.

3.1.1 K-Fold Cross-Validation

The dataset is split into k folds (e.g., $k = 3$):

1. Train on folds 1 & 2, test on fold 3
2. Train on folds 1 & 3, test on fold 2
3. Train on folds 2 & 3, test on fold 1

Final performance = average of all k test scores.

Implementation: Cross-Validation in Scikit-learn

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)

# Perform 3-fold cross-validation
scores = cross_val_score(sgd_clf, X_train, y_train_5,
                         cv=3, scoring="accuracy")

print(f"Accuracy scores: {scores}")
print(f"Mean accuracy: {scores.mean():.4f}")
# Output might be: [0.95, 0.96, 0.94] → Mean: 0.95
```

Benefits:

- More reliable than single train/test split
- Uses all data for both training and testing
- Reduces variance in performance estimates
- Helps detect overfitting

3.2 The Problem with Accuracy on Skewed Datasets

3.2.1 What is a Skewed Dataset?

A dataset where one class significantly outnumbers the other(s). In the MNIST "5-detector":

- **Class "5"**: 10% of images (≈6,000 out of 60,000)
- **Class "not-5"**: 90% of images (≈54,000 out of 60,000)

This is a **highly skewed** (imbalanced) dataset!

3.2.2 The Dummy Classifier Problem

WARNING: Accuracy Can Be Misleading!

Consider a "dummy" classifier that **always predicts "not-5"**:

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier(strategy="most_frequent")
dummy_clf.fit(X_train, y_train_5)

# Evaluate with cross-validation
scores = cross_val_score(dummy_clf, X_train, y_train_5,
                         cv=3, scoring="accuracy")
print(f"Dummy classifier accuracy: {scores.mean():.2f}")
# Output: 0.90 (90% accuracy!)
```

Why 90% accuracy?

- 90% of images are "not-5"
- Always predicting "not-5" gets 90% correct
- But it **never detects a single "5"!**

This classifier is **completely useless** despite 90% accuracy!

3.2.3 Why Accuracy Fails on Skewed Datasets

Accuracy is defined as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Predictions}} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

Where:

- TP = True Positives (correctly predicted "5")
- TN = True Negatives (correctly predicted "not-5")
- FP = False Positives (incorrectly predicted "5")
- FN = False Negatives (incorrectly predicted "not-5")

The Problem:

- When one class dominates (90% "not-5"), the TN term dominates the accuracy
- A classifier can achieve high accuracy by simply predicting the majority class
- It doesn't reflect the model's ability to detect the minority class (the "5"s we care about!)

Key Insight

Accuracy is NOT a good metric when:

- Classes are imbalanced (skewed)
- You care more about detecting the minority class
- The cost of different errors varies (e.g., missing a fraud vs false alarm)

Better alternatives: Precision, Recall, F1-Score, Confusion Matrix, ROC-AUC (covered in next sections)

3.2.4 Real-World Examples of Skewed Datasets

- **Fraud Detection**: 99.9% legitimate transactions, 0.1% fraud
- **Disease Diagnosis**: 95% healthy, 5% disease
- **Spam Detection**: 80% ham, 20% spam
- **Anomaly Detection**: 99% normal, 1% anomalies

In all these cases, a dummy classifier predicting the majority class would have high accuracy but be useless in practice!

3.3 Confusion Matrix

The **Confusion Matrix** is a table that visualizes the performance of a classification model by showing all possible outcomes of predictions.

3.3.1 The Four Categories

For binary classification, every prediction falls into one of four categories:

	Predicted Negative	**Predicted Positive**
Actually Negative	True Negative (TN)	False Positive (FP)
Actually Positive	False Negative (FN)	True Positive (TP)

True Positive (TP) Correctly predicted positive class (predicted "5", actually "5")

True Negative (TN) Correctly predicted negative class (predicted "not-5", actually "not-5")

False Positive (FP) Incorrectly predicted positive (predicted "5", actually "not-5") - **Type I Error**

False Negative (FN) Incorrectly predicted negative (predicted "not-5", actually "5") - **Type II Error**

Implementation: Creating Confusion Matrix

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix

# Get predictions for all training samples using CV
y_train_pred = cross_val_predict(sgd_clf, X_train,
                                 y_train_5, cv=3)

# Create confusion matrix
cm = confusion_matrix(y_train_5, y_train_pred)
print(cm)

# Output example:
# [[53272   1307]  ← Row 0: Actually "not-5"
#  [  1077  4344]] ← Row 1: Actually "5"
#    ↑TN    ↑FP      ↑FN    ↑TP

# Perfect classifier would be:
# [[54579     0]
#  [    0  5421]]
```

Key Point: Reading Confusion Matrix

Remember: Rows = Actual class, Columns = Predicted class
Matrix layout:

$$\begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix}$$

Quick checks:

- Diagonal elements (TN, TP) = **correct predictions**
- Off-diagonal elements (FP, FN) = **errors**
- Sum of row = total actual instances of that class
- Sum of column = total predicted instances of that class

3.4 Precision and Recall

These metrics focus on the **positive class** performance, which is often more important than overall accuracy in skewed datasets.

3.4.1 Precision

Precision answers: "Of all instances predicted as positive, how many were actually positive?"

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{\text{True Positives}}{\text{All Predicted Positives}} \tag{4}$$

Interpretation:

- High precision = few false positives
- "When the model says it's a 5, how often is it right?"

- Precision = 1.0 means no false alarms

Example: If classifier predicts 100 images as "5" but only 80 are actually "5":

$$\text{Precision} = \frac{80}{100} = 0.80 = 80\%$$

3.4.2 Recall (Sensitivity or True Positive Rate)

Recall answers: "Of all actual positive instances, how many did we correctly identify?"

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{\text{True Positives}}{\text{All Actual Positives}} \tag{5}$$

Interpretation:

- High recall = few false negatives
- "Of all the 5s in the dataset, how many did we catch?"
- Recall = 1.0 means we didn't miss any positives

Example: If there are 100 actual "5"s but we only detected 80 of them:

$$\text{Recall} = \frac{80}{100} = 0.80 = 80\%$$

Implementation: Calculating Precision and Recall

```
from sklearn.metrics import precision_score, recall_score

# Calculate precision
precision = precision_score(y_train_5, y_train_pred)
print(f"Precision: {precision:.4f}")
# Example: 0.7687

# Calculate recall
recall = recall_score(y_train_5, y_train_pred)
print(f"Recall: {recall:.4f}")
# Example: 0.8013

# Or get both from confusion matrix
tn, fp, fn, tp = cm.ravel()
precision = tp / (tp + fp)
recall = tp / (tp + fn)
```

3.4.3 The Precision-Recall Tradeoff

IMPORTANT: You Cannot Maximize Both Simultaneously!

Increasing precision typically **decreases** recall, and vice versa.
Why?

- **High precision** (strict classifier): Only predict "5" when very confident
  - Few false positives (FP ↓)
  - But miss many actual "5"s (FN ↑)
  - Result: High precision, Low recall
- **High recall** (lenient classifier): Predict "5" more liberally
  - Catch most actual "5"s (FN ↓)
  - But many false alarms (FP ↑)
  - Result: High recall, Low precision

3.4.4 When to Prefer Precision vs Recall

Prefer High Precision when false positives are costly

- Email spam filter: Don't want to mark important emails as spam
- Video recommendation: Don't want to show unsafe content to kids
- Stock trading: Don't want to buy bad stocks

Prefer High Recall when false negatives are costly

- Disease screening: Don't want to miss sick patients
- Fraud detection: Don't want to miss fraudulent transactions
- Security threat detection: Don't want to miss intrusions

Exam Key Points: Confusion Matrix and Metrics

Must Remember Formulas:

$$\text{Precision} = \frac{TP}{TP + FP} \quad \text{(of predicted positives, how many correct?)}$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad \text{(of actual positives, how many found?)}$$

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

Confusion Matrix Layout:

$$\begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix} \quad \text{(Rows = Actual, Columns = Predicted)}$$

Quick Mnemonics:

- **Precision**: "Positive Predictive Value" - both start with P
- **Recall**: "How many Reals did we Retrieve?" - both start with R
- **FP**: "False Alarm" (said positive, was wrong)
- **FN**: "Missed Positive" (said negative, was wrong)

Common Mistakes to Avoid:

- Don't confuse rows and columns in confusion matrix
- Precision uses predicted positives (TP + FP) in denominator
- Recall uses actual positives (TP + FN) in denominator
- High accuracy $\neq$ good model on imbalanced data

3.5 Understanding All Performance Metrics - Intuitive Guide

Let's clarify all the terminology with intuitive explanations:

Metric Intuitions: The Complete Picture

Start with the Confusion Matrix:

$$\begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix}$$

Now, let's understand each metric:

Precision "When I say YES, how often am I right?"

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{\text{Correct YES}}{\text{All my YES predictions}}$$

Focus: Quality of positive predictions

Recall (TPR) "Of all actual YES cases, how many did I catch?"

$$\text{Recall} = \text{TPR} = \frac{TP}{TP + FN} = \frac{\text{Correct YES}}{\text{All actual YES}}$$

Focus: Coverage of positive class

True Negative Rate (TNR) "Of all actual NO cases, how many did I correctly identify?"

$$\text{TNR (Specificity)} = \frac{TN}{TN + FP} = \frac{\text{Correct NO}}{\text{All actual NO}}$$

Focus: Coverage of negative class

False Positive Rate (FPR) "Of all actual NO cases, how many did I wrongly call YES?"

$$\text{FPR} = \frac{FP}{TN + FP} = \frac{\text{Wrong YES}}{\text{All actual NO}} = 1 - \text{TNR}$$

Focus: False alarm rate

Key Relationships:

- TPR (Recall) and FPR both increase as threshold decreases
- TNR and FPR are complementary: $\text{TNR} + \text{FPR} = 1$
- TPR = Recall = Sensitivity (all same thing!)

Memory Aid: Row vs Column Metrics

Think of the confusion matrix as a table:
Row-based metrics (use actual class totals):

- Recall (TPR) = TP / (TP + FN) - uses *actual positive* row
- TNR = TN / (TN + FP) - uses *actual negative* row

Column-based metrics (use predicted class totals):

- Precision = TP / (TP + FP) - uses *predicted positive* column

Simple story:

- Precision: "Of my YES answers, how many correct?"
- Recall: "Of actual YESes, how many did I find?"
- TNR: "Of actual NOs, how many did I correctly identify?"
- FPR: "Of actual NOs, how many did I mess up?"

3.6 Decision Function and Threshold

3.6.1 How SGDClassifier Makes Predictions

SGDClassifier doesn't directly output a class label. Instead, it:

1. Computes a **decision score**: score = $\mathbf{w} \cdot \mathbf{x} + b$
2. Compares score to a **threshold** (default = 0)
3. If score $\geq$ threshold $\rightarrow$ predict positive class
4. If score $<$ threshold $\rightarrow$ predict negative class

Implementation: Getting Decision Scores

```
# Train the classifier
sgd_clf.fit(X_train, y_train_5)

# Get decision scores (not predictions!)
y_scores = sgd_clf.decision_function(X_train)
print(y_scores[:5])
# Example: [2500.5, -1800.3, 3200.1, -500.2, 1200.7]

# Positive scores → likely "5"
# Negative scores → likely "not-5"

# Default threshold is 0
threshold = 0
y_pred = (y_scores >= threshold)
```

3.6.2 Adjusting the Threshold

By changing the threshold, you can control the precision-recall tradeoff:

- **Higher threshold** (e.g., 3000):
  - Only predict "5" when very confident
  - Fewer predictions → Higher precision, Lower recall
  - Fewer false positives (FP ↓), More false negatives (FN ↑)
- **Lower threshold** (e.g., -1000):
  - Predict "5" more liberally
  - More predictions → Lower precision, Higher recall
  - More false positives (FP ↑), Fewer false negatives (FN ↓)

Implementation: Custom Threshold Selection

```
from sklearn.metrics import precision_recall_curve

# Get decision scores using cross-validation
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5,
                             cv=3, method="decision_function")

# Calculate precision and recall for all thresholds
precisions, recalls, thresholds = precision_recall_curve(
    y_train_5, y_scores)

# Example: Find threshold for 90% precision
target_precision = 0.90
threshold_90_precision = thresholds[
    np.argmax(precisions >= target_precision)]

print(f"Threshold for 90% precision: {threshold_90_precision}")

# Use custom threshold
y_train_pred_90 = (y_scores >= threshold_90_precision)

# Verify
print(f"Precision: {precision_score(y_train_5, y_train_pred_90):.2f}")
print(f"Recall: {recall_score(y_train_5, y_train_pred_90):.2f}")
```

3.7 Precision-Recall Curve

The **Precision-Recall curve** plots precision vs recall for all possible thresholds.

Implementation: Plotting Precision-Recall Curve

```
import matplotlib.pyplot as plt

# Plot precision-recall curve
plt.plot(recalls, precisions, linewidth=2)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve")
plt.grid(True)
plt.show()

# Highlight your operating point (e.g., 90% precision)
plt.plot(recalls, precisions, linewidth=2, label="PR Curve")
idx_90 = np.argmax(precisions >= 0.90)
plt.plot(recalls[idx_90], precisions[idx_90], "ro",
         markersize=10, label="90% Precision")
plt.legend()
plt.show()
```

How to read the curve:

- **Top-right** = ideal (high precision AND high recall)
- **Bottom-right** = high recall, low precision (too many false alarms)
- **Top-left** = high precision, low recall (missing many positives)
- The curve shows the tradeoff: as recall increases, precision typically decreases

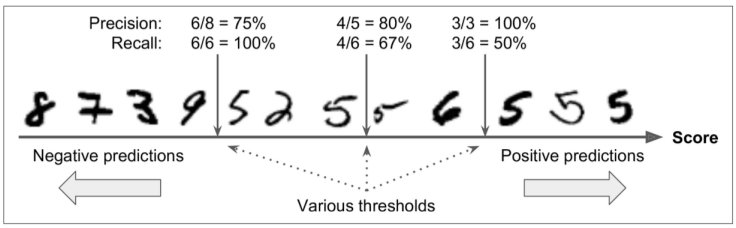

Figure 3-3. Decision threshold and precision/recall tradeoff

Figure 1: Precision-Recall Curve showing the tradeoff between precision and recall at different thresholds

3.8 ROC Curve (Receiver Operating Characteristic)

The **ROC curve** plots True Positive Rate (TPR = Recall) vs False Positive Rate (FPR) for all thresholds.

3.8.1 Understanding ROC Curve

$$\text{TPR (Recall)} = \frac{TP}{TP + FN} \quad \text{(y-axis)}$$

$$\text{FPR} = \frac{FP}{FP + TN} \quad \text{(x-axis)}$$

Intuition:

- **TPR**: "Of actual positives, how many did we catch?" (GOOD!)
- **FPR**: "Of actual negatives, how many did we falsely alarm?" (BAD!)
- Goal: High TPR (catch positives) with Low FPR (few false alarms)

Implementation: Creating ROC Curve

```
from sklearn.metrics import roc_curve, roc_auc_score

# Calculate TPR and FPR for all thresholds
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)

# Plot ROC curve
plt.plot(fpr, tpr, linewidth=2, label="SGD Classifier")
plt.plot([0, 1], [0, 1], 'k--', label="Random Classifier")
plt.xlabel("False Positive Rate (FPR)")
plt.ylabel("True Positive Rate (TPR/Recall)")
plt.title("ROC Curve")
plt.legend()
plt.grid(True)
plt.show()

# Calculate AUC (Area Under Curve)
auc_score = roc_auc_score(y_train_5, y_scores)
print(f"AUC Score: {auc_score:.4f}")
# Example: 0.9604 (closer to 1.0 = better)
```

3.8.2 Interpreting ROC Curve

Curve Position	Interpretation
Top-left corner	Perfect classifier (TPR=1, FPR=0)
Diagonal line	Random classifier (no better than guessing)
Above diagonal	Better than random
Below diagonal	Worse than random (flip predictions!)

AUC (Area Under Curve):

- AUC = 1.0 → Perfect classifier
- AUC = 0.5 → Random classifier
- AUC > 0.9 → Excellent
- AUC 0.8-0.9 → Good
- AUC 0.7-0.8 → Fair
- AUC < 0.7 → Poor

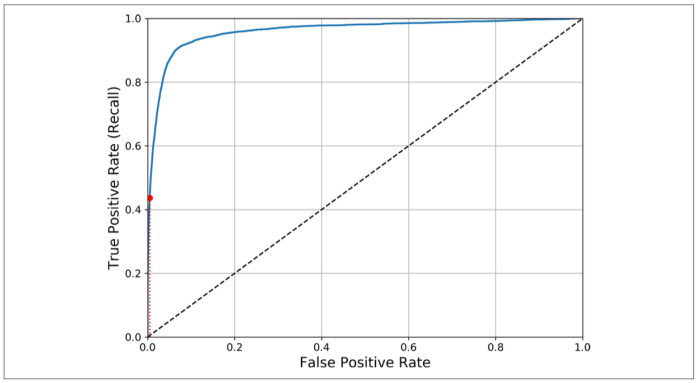

Figure 3-6. ROC curve

Figure 2: ROC Curve showing TPR vs FPR. The diagonal line represents a random classifier. The closer the curve is to the top-left corner, the better the classifier.

When to Use PR Curve vs ROC Curve

Use Precision-Recall Curve when:

- Dataset is highly imbalanced
- You care more about the positive class
- False positives are very costly
- Example: Rare disease detection, fraud detection

Use ROC Curve when:

- Dataset is relatively balanced
- You care about both classes equally
- Want to compare multiple classifiers
- Example: Balanced binary classification tasks

Why PR is better for imbalanced data: ROC curve can be overly optimistic on imbalanced datasets because a large number of true negatives (TN) can make FPR look artificially low. PR curve focuses on the positive class performance.

3.9 Creating Classifier with Target Precision

Here's a complete workflow to create a classifier with your desired precision:

Complete Workflow: 90% Precision Classifier

```
# Step 1: Train classifier
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)

# Step 2: Get decision scores
from sklearn.model_selection import cross_val_predict
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5,
                             cv=3, method="decision_function")

# Step 3: Compute precision-recall curve
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(
    y_train_5, y_scores)

# Step 4: Find threshold for 90% precision
target_precision = 0.90
idx = np.argmax(precisions >= target_precision)
threshold_90 = thresholds[idx]
print(f"Threshold: {threshold_90:.2f}")
print(f"At this threshold:")
print(f"  Precision: {precisions[idx]:.2%}")
print(f"  Recall: {recalls[idx]:.2%}")

# Step 5: Make predictions with custom threshold
y_train_pred_90 = (y_scores >= threshold_90)

# Step 6: Verify on test set
y_test_scores = sgd_clf.decision_function(X_test)
y_test_pred_90 = (y_test_scores >= threshold_90)

from sklearn.metrics import classification_report
print(classification_report(y_test_5, y_test_pred_90))
```

IMPORTANT: Exam Key Points

Decision Function & Threshold:

- Decision score = $\mathbf{w} \cdot \mathbf{x} + b$
- Default threshold = 0
- Higher threshold → Higher precision, Lower recall
- Lower threshold → Lower precision, Higher recall

Metric Formulas (MEMORIZE!):

$$\text{TPR (Recall)} = \frac{TP}{TP + FN}$$
$$\text{FPR} = \frac{FP}{FP + TN} = 1 - \text{TNR}$$
$$\text{Precision} = \frac{TP}{TP + FP}$$
$$\text{TNR (Specificity)} = \frac{TN}{TN + FP}$$

Curve Interpretations:

- **PR Curve**: Precision vs Recall - use for imbalanced data
- **ROC Curve**: TPR vs FPR - use for balanced data
- **AUC**: Area under ROC curve (1.0 = perfect, 0.5 = random)

Quick Decision Guide:

- Need high precision? → Increase threshold (fewer false alarms)
- Need high recall? → Decrease threshold (catch more positives)
- Imbalanced data? → Use PR curve and F1-score
- Balanced data? → Use ROC curve and AUC

4 Multiclass Classification

Multiclass classification involves distinguishing between more than two classes (e.g., digits 0-9 in MNIST).

4.1 Strategies for Multiclass Classification

Some algorithms (like Random Forest, Naive Bayes) naturally support multiple classes. Others (like SGD, SVM) are strictly binary classifiers. For binary classifiers, we use one of two strategies:

4.1.1 One-vs-All (OvA) or One-vs-Rest (OvR)

Strategy: Train N binary classifiers, one for each class.
How it works:

- Classifier 1: "Is it a 0?" (0 vs not-0)
- Classifier 2: "Is it a 1?" (1 vs not-1)
- Classifier 3: "Is it a 2?" (2 vs not-2)
- ... (10 classifiers total for digits 0-9)

Prediction: Run all N classifiers, choose the one with the highest decision score.
Advantages:

- Only need N classifiers
- Efficient for most cases

Disadvantages:

- Each classifier sees imbalanced data (1 class vs all others)

4.1.2 One-vs-One (OvO) or All-vs-All (AvA)

Strategy: Train one binary classifier for every pair of classes.
How it works:

- Classifier 1: "0 vs 1"
- Classifier 2: "0 vs 2"
- Classifier 3: "0 vs 3"
- ... (45 classifiers total for 10 classes: $\frac{N(N-1)}{2} = \frac{10\times 9}{2} = 45$)

Prediction: Run all classifiers, each votes for a class. Class with most votes wins.
Advantages:

- Each classifier only needs to distinguish between two classes
- Training data for each classifier is smaller (only examples of two classes)
- Useful for algorithms that scale poorly with dataset size (like SVM)

Disadvantages:

- Need $\frac{N(N-1)}{2}$ classifiers (many more!)
- Slower prediction (must run all classifiers)

Implementation: Multiclass Classification with SGD

```
from sklearn.linear_model import SGDClassifier
from sklearn.multiclass import OneVsRestClassifier, OneVsOneClassifier

# Automatic: Scikit-learn detects multiclass and uses OvR by default
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train)  # y_train has values 0-9

# Prediction
y_pred = sgd_clf.predict(X_test)
print(y_pred[:5])  # [3, 5, 1, 7, 4]

# Check decision scores for all classes
decision_scores = sgd_clf.decision_function(X_test[:1])
print(decision_scores)  # Array of 10 scores (one per class)
print(f"Predicted class: {decision_scores.argmax()}")

# Force OvO strategy
ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
ovo_clf.fit(X_train, y_train)
print(f"Number of estimators: {len(ovo_clf.estimators_)}")  # 45

# Force OvR strategy
ovr_clf = OneVsRestClassifier(SGDClassifier(random_state=42))
ovr_clf.fit(X_train, y_train)
print(f"Number of estimators: {len(ovr_clf.estimators_)}")  # 10
```

4.2 Random Forest for Multiclass Classification

Random Forest naturally supports multiclass classification without needing OvR or OvO strategies.

Implementation: Random Forest Multiclass

```
from sklearn.ensemble import RandomForestClassifier

# Random Forest handles multiclass directly
forest_clf = RandomForestClassifier(random_state=42)
forest_clf.fit(X_train, y_train)

# Prediction
y_pred = forest_clf.predict(X_test)

# Get probability estimates for each class
y_proba = forest_clf.predict_proba(X_test[:1])
print(y_proba)  # Array of 10 probabilities (one per class)
# Example: [[0.1, 0.0, 0.0, 0.7, 0.0, 0.2, ...]]

# Evaluate
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")
```

4.3 Evaluating Multiclass Classifiers

Use cross-validation and standard metrics:

Implementation: Multiclass Evaluation

```
from sklearn.model_selection import cross_val_score

# Cross-validation
scores = cross_val_score(sgd_clf, X_train, y_train,
                         cv=3, scoring="accuracy")
print(f"CV Accuracy: {scores.mean():.4f}")

# Scale input for better SGD performance
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))

sgd_clf.fit(X_train_scaled, y_train)
scores_scaled = cross_val_score(sgd_clf, X_train_scaled, y_train,
                                cv=3, scoring="accuracy")
print(f"CV Accuracy (scaled): {scores_scaled.mean():.4f}")
# Scaling often improves SGD performance significantly!
```

5 Error Analysis

After building a classifier, analyzing where it makes mistakes helps improve performance.

5.1 Confusion Matrix for Multiclass

For multiclass problems, the confusion matrix is N×N (where N = number of classes).

Implementation: Multiclass Confusion Matrix

```
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_val_predict
import matplotlib.pyplot as plt

# Get predictions using cross-validation
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled,
                                 y_train, cv=3)

# Create confusion matrix
conf_mx = confusion_matrix(y_train, y_train_pred)
print(conf_mx)

# Visualize confusion matrix
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.title("Confusion Matrix")
plt.colorbar()
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.show()
```

5.2 Reading the Confusion Matrix

Matrix structure: Row = actual class, Column = predicted class

Example for digits 0-9:

$$\begin{bmatrix} 5923 & 0 & 0 & 0 & \dots \\ 0 & 6742 & 3 & 1 & \dots \\ 0 & 0 & 5958 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

How to read:

- **Diagonal elements**: Correct predictions (darker = better)
- **Off-diagonal elements**: Errors (lighter = fewer errors)
- **Row 3, Column 5**: Number of "3"s misclassified as "5"
- **Row 5, Column 3**: Number of "5"s misclassified as "3"

5.3 Normalizing the Confusion Matrix

To better visualize error patterns, normalize by row (divide by total instances per class):

Implementation: Normalized Confusion Matrix

```
# Normalize by row (show error rates, not counts)
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums

# Fill diagonal with zeros to highlight errors
np.fill_diagonal(norm_conf_mx, 0)

# Visualize errors only
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.title("Confusion Matrix - Errors Only (Normalized)")
plt.colorbar()
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.show()

# Now bright spots show where classifier gets confused
```

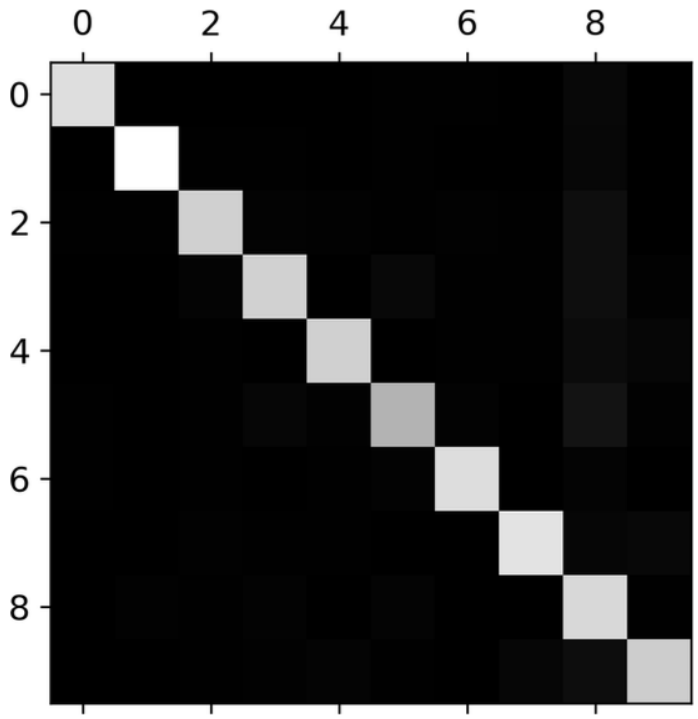

Figure 3: Confusion Matrix for MNIST digit classification. Darker cells on diagonal indicate correct predictions, lighter off-diagonal cells show misclassifications.

5.4 Why Does the Classifier Confuse 3 and 5?

Common confusion patterns in MNIST:

- **3 vs 5**: Similar shapes, especially if 3 is written with rounded top

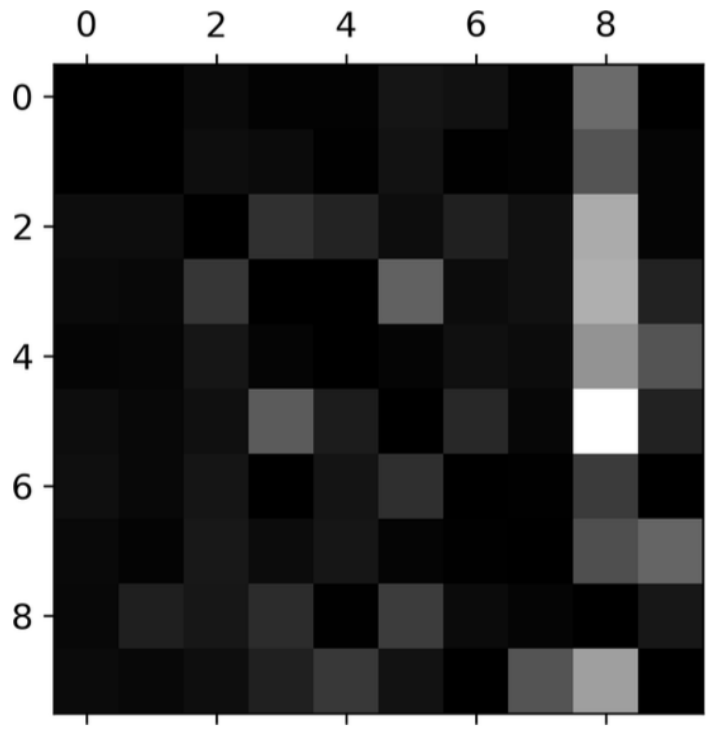

Figure 4: Normalized Confusion Matrix with diagonal zeroed out to highlight error patterns. Bright spots indicate common misclassifications.

- **8 vs 9**: 9 can look like 8 with top loop not fully closed
- **4 vs 9**: Some handwritten 4s resemble 9s
- **7 vs 9**: Similar stroke patterns

Understanding Confusion Between 3 and 5

Visual similarity:

- Both have horizontal lines at top and middle
- Both have curved segments
- Pixel-level differences can be subtle

How to investigate:

1. Find examples where 3 was predicted as 5
2. Visualize these images
3. Look for patterns (handwriting style, image quality)

Possible improvements:

- More training data for confused classes
- Feature engineering (extract stroke features)
- Data augmentation (rotate, shift images)
- Try different algorithms (CNN works better for images)

Implementation: Analyzing Specific Errors

```
# Find where 3s were classified as 5s
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

# Visualize examples
fig, axes = plt.subplots(2, 5, figsize=(10, 4))
for i, ax in enumerate(axes[0]):
    ax.imshow(X_ab[i].reshape(28, 28), cmap='binary')
    ax.set_title(f"3→5")
    ax.axis('off')

for i, ax in enumerate(axes[1]):
    ax.imshow(X_ba[i].reshape(28, 28), cmap='binary')
    ax.set_title(f"5→3")
    ax.axis('off')

plt.tight_layout()
plt.show()
```

Figure 5: Image denoising example: (left) noisy input image, (center) denoised output from KNN classifier, (right) original clean image.

6 Multilabel Classification

Multilabel classification assigns multiple labels to each instance (not mutually exclusive).

6.1 Example: Face Recognition

An image might contain:

- Alice: Yes
- Bob: Yes
- Charlie: No

Output: [True, True, False] - multiple labels per instance!

6.2 MNIST Multilabel Example

Create a multilabel target: "is it large?" (≥ 7) and "is it odd?"

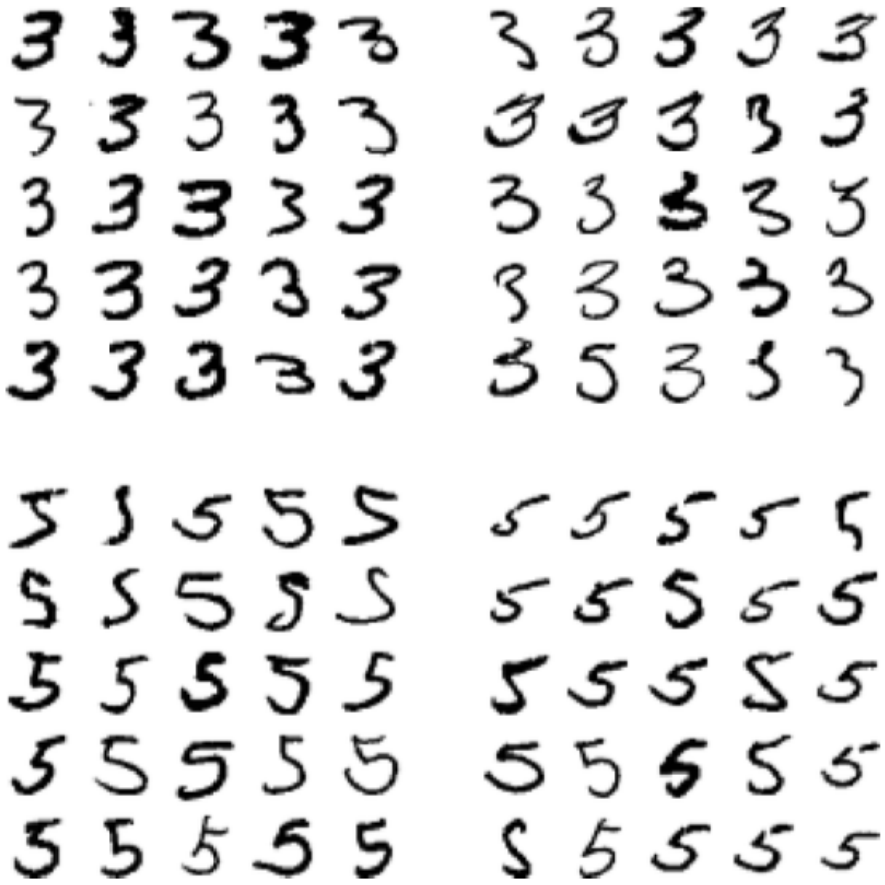

Figure 6: Examples of misclassified digits: 3s predicted as 5s (top row) and 5s predicted as 3s (bottom row). Notice the visual similarities that confuse the classifier.

Implementation: Multilabel Classification

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

# Create multilabel target
y_train_large = (y_train >= 7)  # True if digit >= 7
y_train_odd = (y_train % 2 == 1)  # True if odd
y_multilabel = np.c_[y_train_large, y_train_odd]

# Train KNN classifier
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)

# Predict
print(knn_clf.predict([some_digit]))
# Output example: [[False, True]]
# Meaning: not large (< 7) but odd

# Explanation: If digit is 5
# - Is it large? No (5 < 7) → False
# - Is it odd? Yes (5 % 2 = 1) → True
# - Result: [False, True]
```

6.3 Evaluating Multilabel Classifiers

Many evaluation metrics available. Common approach: average F1-score across all labels.

Implementation: Multilabel Evaluation

```
from sklearn.metrics import f1_score

# Get predictions
y_train_knn_pred = cross_val_predict(knn_clf, X_train,
                                     y_multilabel, cv=3)

# Calculate F1 score (averaged across all labels)
f1 = f1_score(y_multilabel, y_train_knn_pred, average="macro")
print(f"F1 Score: {f1:.4f}")

# Per-label F1 scores
f1_per_label = f1_score(y_multilabel, y_train_knn_pred,
                        average=None)
print(f"F1 for 'large': {f1_per_label[0]:.4f}")
print(f"F1 for 'odd': {f1_per_label[1]:.4f}")
```

Key difference from multiclass:

- **Multiclass**: Each instance belongs to exactly ONE class (digit is 0 OR 1 OR 2...)
- **Multilabel**: Each instance can have MULTIPLE labels (large AND odd)

7 Multioutput Classification

Multioutput classification (or multioutput-multiclass) is a generalization where each label can be multiclass (more than binary).

7.1 Example: Image Denoising

Task: Remove noise from MNIST images

- **Input**: Noisy image (784 pixels)
- **Output**: Clean image (784 pixel values, each 0-255)
- Each pixel is a multiclass classification (256 possible values)
- 784 outputs, each with 256 classes!

Implementation: Image Denoising with KNN

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

# Add noise to training images
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_noisy = X_train + noise

# Add noise to test images
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_noisy = X_test + noise

# Target: clean images
y_train_clean = X_train
y_test_clean = X_test

# Train KNN to denoise
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_noisy, y_train_clean)

# Denoise a test image
clean_digit = knn_clf.predict([X_test_noisy[0]])

# Visualize
import matplotlib.pyplot as plt
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].imshow(X_test_noisy[0].reshape(28, 28),
               cmap='binary', interpolation='nearest')
axes[0].set_title("Noisy Image")
axes[0].axis('off')

axes[1].imshow(clean_digit.reshape(28, 28),
               cmap='binary', interpolation='nearest')
axes[1].set_title("Denoised Image")
axes[1].axis('off')

axes[2].imshow(y_test_clean[0].reshape(28, 28),
               cmap='binary', interpolation='nearest')
axes[2].set_title("Original Clean Image")
axes[2].axis('off')

plt.tight_layout()
plt.show()
```

7.2 Classification Type Summary

Type	Description	Example
Binary	2 classes, mutually exclusive	Spam vs Ham
Multiclass	N classes, mutually exclusive	Digits 0-9
Multilabel	Multiple binary labels	[Large, Odd]
Multioutput	Multiple multiclass outputs	Image denoising

Understanding the Hierarchy

Classification hierarchy (from simple to complex):

1. **Binary**: One output, two classes
2. **Multiclass**: One output, N classes (N > 2)
3. **Multilabel**: M outputs, each binary
4. **Multioutput**: M outputs, each with N classes

Output shapes:

- Binary/Multiclass: Single value (0, 1, 2, ...)
- Multilabel: Vector [0, 1, 0, 1, ...] (multiple binary)
- Multioutput: Vector [5, 128, 200, ...] (multiple multiclass)

8 Chapter Exercises - Notes and Solutions

8.1 Exercise 1: Achieve 97% Accuracy on MNIST

Goal: Build classifier with > 97% accuracy on test set.

Hint: KNeighborsClassifier works well. Use GridSearchCV to find optimal hyperparameters.

Solution Approach

Key points:

- `weights='uniform'`: All neighbors weighted equally
- `weights='distance'`: Closer neighbors have more influence
- `n_neighbors`: Number of neighbors to consider
- Usually `weights='distance'` performs better
- Data scaling can further improve results

Code implementation:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = [
```

```
    {
        'weights': ['uniform', 'distance'],
        'n_neighbors': [3, 4, 5, 6, 7]
    }
]

# Create KNN classifier
knn_clf = KNeighborsClassifier()

# Grid search with cross-validation
grid_search = GridSearchCV(knn_clf, param_grid, cv=3,
                           scoring='accuracy', verbose=2,
                           n_jobs=-1)

grid_search.fit(X_train, y_train)

# Best parameters
print("Best params:", grid_search.best_params_)
print("Best CV score:", grid_search.best_score_)

# Evaluate on test set
best_knn = grid_search.best_estimator_
from sklearn.metrics import accuracy_score
y_pred = best_knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Test accuracy:", accuracy)
# Expected: > 0.97
```

8.2 Exercise 2: Data Augmentation with Shifted Images

Goal: Augment training set by shifting images in 4 directions, improving model accuracy.

Why Data Augmentation Works

Benefits:

- Increases training set size (5x in this case)
- Makes model robust to small translations
- Reduces overfitting
- Simulates real-world variations in digit position

Implementation:

```
from scipy.ndimage import shift

def shift_image(image, dx, dy):
    """Shift image by dx, dy pixels."""
    image_2d = image.reshape(28, 28)
    shifted = shift(image_2d, [dy, dx], cval=0, mode='constant')
    return shifted.reshape(-1)

# Create augmented dataset
X_train_augmented = [X_train]
y_train_augmented = [y_train]
```

```
# Add shifted versions
for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
    X_shifted = np.array([shift_image(img, dx, dy)
                          for img in X_train])
    X_train_augmented.append(X_shifted)
    y_train_augmented.append(y_train)

# Concatenate all
X_train_aug = np.concatenate(X_train_augmented)
y_train_aug = np.concatenate(y_train_augmented)

print(f"Original size: {X_train.shape[0]}")
print(f"Augmented size: {X_train_aug.shape[0]}")
# Output: 60000 -> 300000 (5x larger!)

# Train on augmented data
knn_clf.fit(X_train_aug, y_train_aug)
y_pred = knn_clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy with augmentation: {accuracy:.4f}")
# Should be even higher than 97%!
```

8.3 Exercise 3: Titanic Dataset

Task: Binary classification - predict survival (0 = died, 1 = survived)

Features: Age, sex, passenger class, fare, number of siblings/spouses, etc.

Key Steps for Titanic

Approach:

1. **Data exploration**: Check missing values, distributions
2. **Feature engineering**:
   - Handle missing age values (median imputation)
   - Convert categorical variables (sex, embarked)
   - Create new features (family size = siblings + parents)
   - Extract title from name (Mr., Mrs., Miss.)
3. **Try multiple classifiers**:
   - Logistic Regression
   - Random Forest
   - Gradient Boosting
4. **Evaluate**: Use cross-validation, check precision/recall

Example pipeline:

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier

pipeline = Pipeline([
```

```
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(random_state=42))
])

pipeline.fit(X_train, y_train)
```

8.4 Exercise 4: Spam Classifier

Task: Binary classification - spam vs ham (legitimate email)
Challenge: Convert text emails to feature vectors

Spam Classifier Pipeline

Data preparation steps:

1. **Load data**: Parse email files from SpamAssassin dataset
2. **Preprocessing**:
   - Strip headers (optional)
   - Convert to lowercase
   - Remove punctuation
   - Replace URLs with "URL" token
   - Replace numbers with "NUMBER" token
   - Stemming (e.g., "running" to "run")
3. **Feature extraction**:
   - Bag of words (word presence/absence)
   - Or word counts
   - Or TF-IDF scores
4. **Classification**:
   - Try Logistic Regression, Naive Bayes, SVM
   - Optimize for both precision and recall

Why this is challenging:

- Email format varies widely
- Need careful text preprocessing
- Balance precision (don't mark ham as spam) vs recall (catch all spam)
- Large vocabulary (sparse, high-dimensional features)

Example feature extraction:

```
from sklearn.feature_extraction.text import CountVectorizer

# Bag of words
vectorizer = CountVectorizer(binary=True)  # Presence only
X_train_counts = vectorizer.fit_transform(emails_train)
```

```
# Or with counts
vectorizer = CountVectorizer()
X_train_counts = vectorizer.fit_transform(emails_train)

# Or TF-IDF
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer()
X_train_tfidf = tfidf.fit_transform(emails_train)
```

Exercise Key Takeaways

Common themes across exercises:

1. **Hyperparameter tuning**: Use GridSearchCV or RandomizedSearchCV
2. **Data augmentation**: Artificially expand training set when possible
3. **Feature engineering**: Domain knowledge improves performance
4. **Pipeline approach**: Combine preprocessing and modeling
5. **Multiple metrics**: Don't rely only on accuracy
6. **Cross-validation**: Essential for reliable evaluation

Practical tips:

- Start simple, then add complexity
- Visualize errors to understand model behavior
- Try multiple algorithms before settling on one
- Feature engineering often beats fancy algorithms
- Use pipelines to avoid data leakage