The Complete Machine Learning Guide: From Mathematical Foundations to Mastery

Table of Contents

Introduction

Machine learning is the science of getting computers to learn and act like humans do, improving their learning over time in autonomous fashion. This comprehensive guide will take you from the mathematical foundations through to advanced implementations, ensuring you understand not just how to use algorithms, but why they work.

Prerequisites

- Basic programming knowledge (Python preferred)
- High school mathematics
- Curiosity and persistence

How to Use This Book

1. **Complete beginners**: Start from Chapter 1 and work through sequentially
2. **Those with math background**: Can skip to Chapter 6
3. **Experienced practitioners**: Use as reference for specific topics

---

Part I: Mathematical Foundations

Chapter 1: Linear Algebra Fundamentals

Linear algebra is the backbone of machine learning. Nearly every algorithm relies on vector and matrix operations.

1.1 Vectors

A vector is an ordered list of numbers. In ML, vectors represent data points or features.

Mathematical Definition: A vector $\mathbf{v} \in \mathbb{R}^n$ is an n-tuple of real numbers: $\mathbf{v} = [v_1, v_2, ..., v_n]^T$

Python Implementation:

python

```python
import numpy as np

# Creating vectors
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])

# Vector operations
# Addition
v_sum = v1 + v2  # [5, 7, 9]

# Scalar multiplication
v_scaled = 3 * v1  # [3, 6, 9]

# Dot product (inner product)
dot_product = np.dot(v1, v2)  # 1*4 + 2*5 + 3*6 = 32

# Magnitude (norm)
magnitude = np.linalg.norm(v1)  # √(1² + 2² + 3²) = √14
```

Geometric Interpretation:

- Vectors represent points or directions in space
- Dot product measures similarity (cosine of angle between vectors)
- Magnitude represents the length of the vector

1.2 Matrices

A matrix is a 2D array of numbers. In ML, matrices represent datasets or transformations.

Mathematical Definition: A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ has m rows and n columns:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Python Implementation:

python

```python
# Creating matrices
A = np.array([[1, 2, 3],
              [4, 5, 6]])

B = np.array([[7, 8],
              [9, 10],
              [11, 12]])

# Matrix multiplication
C = np.dot(A, B)  # Result is 2x2 matrix
# C = [[1*7+2*9+3*11, 1*8+2*10+3*12],
#      [4*7+5*9+6*11, 4*8+5*10+6*12]]
# C = [[58, 64],
#      [139, 154]]

# Transpose
A_transpose = A.T  # Shape changes from (2,3) to (3,2)

# Identity matrix
I = np.eye(3)  # 3x3 identity matrix
```

1.3 Eigenvalues and Eigenvectors

Eigenvectors and eigenvalues reveal the fundamental properties of linear transformations.

Mathematical Definition: For a square matrix **A**, if **Av** = λ**v** for some scalar λ and non-zero vector **v**, then:

- λ is an eigenvalue
- **v** is an eigenvector

Intuition: Eigenvectors are directions that don't change under the transformation, only scaled by eigenvalues.

Python Implementation:

python

```python
# Example: Principal Component Analysis foundation
A = np.array([[3, 1],
              [1, 3]])

eigenvalues, eigenvectors = np.linalg.eig(A)
print(f"Eigenvalues: {eigenvalues}")  # [4, 2]
print(f"Eigenvectors:\n{eigenvectors}")

# Verify: Av = λv
v1 = eigenvectors[:, 0]
λ1 = eigenvalues[0]
print(np.allclose(A @ v1, λ1 * v1))  # True
```

1.4 Matrix Decompositions

Singular Value Decomposition (SVD): Any matrix **A** can be decomposed as: **A** = **UΣV**ᵀ

python

```python
# SVD example - used in dimensionality reduction
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

U, s, Vt = np.linalg.svd(A)

# Reconstruct with reduced rank (compression)
k = 2  # Keep top 2 singular values
A_approx = U[:, :k] @ np.diag(s[:k]) @ Vt[:k, :]
```

1.5 Key Concepts for ML

1. **Rank**: Maximum number of linearly independent columns/rows
2. **Determinant**: Measures how much a matrix scales volumes
3. **Inverse**: Matrix that "undoes" the transformation
4. **Orthogonality**: Perpendicular vectors (dot product = 0)

Chapter 2: Calculus for Machine Learning

Calculus enables us to optimize ML models by finding minima and maxima of functions.

2.1 Derivatives

The derivative measures the rate of change of a function.

Mathematical Definition: For a function f(x), the derivative is: $f'(x) = \lim_{h \to 0} (f(x+h) - f(x))/h$

Common Derivatives:

- $d/dx(x^n) = nx^{n-1}$
- $d/dx(e^x) = e^x$
- $d/dx(\ln x) = 1/x$
- $d/dx(\sin x) = \cos x$

Python Implementation:

python

```python
import numpy as np
import matplotlib.pyplot as plt

# Numerical differentiation
def numerical_derivative(f, x, h=1e-5):
    return (f(x + h) - f(x - h)) / (2 * h)

# Example function: f(x) = x²
def f(x):
    return x**2

# Analytical derivative: f'(x) = 2x
def f_prime(x):
    return 2*x

# Compare numerical and analytical
x = 3.0
print(f"Numerical: {numerical_derivative(f, x)}")  # ≈ 6.0
print(f"Analytical: {f_prime(x)}")  # 6.0

# Visualize
x_vals = np.linspace(-5, 5, 100)
y_vals = f(x_vals)
y_prime = f_prime(x_vals)

plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.plot(x_vals, y_vals)
plt.title("Function f(x) = x²")
plt.subplot(1, 2, 2)
plt.plot(x_vals, y_prime)
plt.title("Derivative f'(x) = 2x")
plt.show()
```

2.2 Partial Derivatives

For functions of multiple variables, we take derivatives with respect to each variable.

Mathematical Definition: For f(x, y), the partial derivatives are:

- ∂f/∂x: derivative with respect to x (treating y as constant)
- ∂f/∂y: derivative with respect to y (treating x as constant)

Example - Loss Function:

python

```python
# Mean Squared Error loss function
def mse_loss(y_pred, y_true):
    return np.mean((y_pred - y_true)**2)

# Partial derivative with respect to predictions
def mse_gradient(y_pred, y_true):
    return 2 * (y_pred - y_true) / len(y_true)

# Example
y_true = np.array([1, 2, 3, 4, 5])
y_pred = np.array([1.1, 2.2, 2.9, 4.1, 5.2])

loss = mse_loss(y_pred, y_true)
gradient = mse_gradient(y_pred, y_true)
print(f"Loss: {loss}")
print(f"Gradient: {gradient}")
```

2.3 The Chain Rule

The chain rule is crucial for backpropagation in neural networks.

Mathematical Definition: If y = f(g(x)), then dy/dx = (dy/dg) × (dg/dx)

Neural Network Example:

python

```python
# Simple neural network forward pass
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

# Forward pass: y = sigmoid(wx + b)
w = 2.0
b = 1.0
x = 3.0

# Intermediate computation
z = w * x + b  # Linear combination
y = sigmoid(z)  # Activation

# Backward pass using chain rule
dy_dz = sigmoid_derivative(z)  # Derivative of sigmoid
dz_dw = x  # Derivative of z with respect to w
dy_dw = dy_dz * dz_dw  # Chain rule

print(f"Output: {y}")
print(f"Gradient w.r.t. weight: {dy_dw}")
```

2.4 Gradient and Directional Derivatives

The gradient points in the direction of steepest increase.

Mathematical Definition: For $f(x_1, x_2, ..., x_n)$, the gradient is: $\nabla f = [\partial f/\partial x_1, \partial f/\partial x_2, ..., \partial f/\partial x_n]^T$

Visualization:

python

```python
# 2D function and its gradient
def f(x, y):
    return x**2 + y**2

def gradient_f(x, y):
    return np.array([2*x, 2*y])

# Create meshgrid
x = np.linspace(-5, 5, 20)
y = np.linspace(-5, 5, 20)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

# Compute gradient field
U = 2 * X  # ∂f/∂x
V = 2 * Y  # ∂f/∂y

# Plot
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.contour(X, Y, Z, levels=20)
plt.title("Function Contours")
plt.subplot(1, 2, 2)
plt.quiver(X, Y, -U, -V, alpha=0.5)  # Negative for descent
plt.title("Gradient Field (Descent Direction)")
plt.show()
```

2.5 Taylor Series and Approximations

Taylor series help us approximate complex functions locally.

Mathematical Definition: $f(x) \approx f(a) + f'(a)(x-a) + f''(a)(x-a)^2/2! + ...$

Application in Optimization:

python

```python
# Second-order Taylor approximation for optimization
def quadratic_approximation(f, f_prime, f_double_prime, x0, x):
    """Taylor expansion up to second order"""
    return (f(x0) +
            f_prime(x0) * (x - x0) +
            0.5 * f_double_prime(x0) * (x - x0)**2)

# Example: approximate f(x) = e^x around x=0
f = np.exp
f_prime = np.exp  # derivative of e^x is e^x
f_double_prime = np.exp

x0 = 0
x_range = np.linspace(-2, 2, 100)
actual = f(x_range)
approx = quadratic_approximation(f, f_prime, f_double_prime, x0, x_range)

plt.plot(x_range, actual, label='Actual: e^x')
plt.plot(x_range, approx, '--', label='Taylor Approximation')
plt.legend()
plt.title('Taylor Series Approximation')
plt.show()
```

Chapter 3: Probability Theory

Probability theory provides the framework for dealing with uncertainty in ML.

3.1 Basic Probability Concepts

Sample Space (Ω): Set of all possible outcomes **Event**: Subset of the sample space **Probability**: Measure of likelihood, P(A) ∈ [0, 1]

Axioms of Probability:

1. P(A) ≥ 0 for any event A
2. P(Ω) = 1
3. For disjoint events: P(A ∪ B) = P(A) + P(B)

Python Implementation:

python

```python
import numpy as np
from collections import Counter

# Simulating dice rolls
def roll_dice(n_rolls=1000):
    return np.random.randint(1, 7, size=n_rolls)

rolls = roll_dice(10000)
counts = Counter(rolls)

# Empirical probabilities
for outcome in range(1, 7):
    prob = counts[outcome] / len(rolls)
    print(f"P({outcome}) = {prob:.3f}")  # Should be ≈ 0.167
```

3.2 Conditional Probability and Bayes' Theorem

Conditional Probability: $P(A|B) = P(A \cap B) / P(B)$

Bayes' Theorem: $P(A|B) = P(B|A) \times P(A) / P(B)$

Application - Naive Bayes Classifier:

python

```
# Spam classification example
class NaiveBayesClassifier:
    def __init__(self):
        self.word_probs_spam = {}
        self.word_probs_ham = {}
        self.p_spam = 0.5

    def train(self, messages, labels):
        spam_messages = [m for m, l in zip(messages, labels) if l == 1]
        ham_messages = [m for m, l in zip(messages, labels) if l == 0]

        self.p_spam = len(spam_messages) / len(messages)

        # Calculate word probabilities
        spam_words = ' '.join(spam_messages).split()
        ham_words = ' '.join(ham_messages).split()

        spam_word_counts = Counter(spam_words)
        ham_word_counts = Counter(ham_words)

        # Laplace smoothing
        vocab = set(spam_words + ham_words)
        for word in vocab:
            self.word_probs_spam[word] = (spam_word_counts[word] + 1) / (len(spam_words) + len(
            self.word_probs_ham[word] = (ham_word_counts[word] + 1) / (len(ham_words) + len(voc

    def predict_proba(self, message):
        words = message.split()

        # Calculate log probabilities (to avoid underflow)
        log_prob_spam = np.log(self.p_spam)
        log_prob_ham = np.log(1 - self.p_spam)

        for word in words:
            if word in self.word_probs_spam:
                log_prob_spam += np.log(self.word_probs_spam[word])
                log_prob_ham += np.log(self.word_probs_ham[word])

        # Convert back to probabilities
        prob_spam = np.exp(log_prob_spam)
        prob_ham = np.exp(log_prob_ham)

        # Normalize
        total = prob_spam + prob_ham
        return prob_spam / total
```

```python
# Example usage
messages = ["buy viagra now", "meeting tomorrow", "cheap pills", "project deadline"]
labels = [1, 0, 1, 0]  # 1 = spam, 0 = ham

nb = NaiveBayesClassifier()
nb.train(messages, labels)
print(nb.predict_proba("buy cheap"))  # Should indicate spam
```

3.3 Random Variables and Distributions

Discrete Random Variables:

python

```python
# Binomial distribution - number of successes in n trials
from scipy import stats

n, p = 10, 0.3  # 10 trials, 30% success probability
binomial_rv = stats.binom(n, p)

# PMF (Probability Mass Function)
x = np.arange(0, 11)
pmf = binomial_rv.pmf(x)

plt.bar(x, pmf)
plt.xlabel('Number of Successes')
plt.ylabel('Probability')
plt.title(f'Binomial Distribution (n={n}, p={p})')
plt.show()

# Expected value and variance
print(f"Expected value: {binomial_rv.mean()}")  # n*p = 3
print(f"Variance: {binomial_rv.var()}")  # n*p*(1-p) = 2.1
```

Continuous Random Variables:

python

```python
# Normal (Gaussian) distribution
mu, sigma = 0, 1  # Mean and standard deviation
normal_rv = stats.norm(mu, sigma)

# PDF (Probability Density Function)
x = np.linspace(-4, 4, 100)
pdf = normal_rv.pdf(x)

plt.plot(x, pdf)
plt.fill_between(x, pdf, alpha=0.3)
plt.xlabel('Value')
plt.ylabel('Density')
plt.title(f'Normal Distribution (μ={mu}, σ={sigma})')
plt.show()

# Cumulative Distribution Function (CDF)
# P(X ≤ x)
print(f"P(X ≤ 0) = {normal_rv.cdf(0)}")  # 0.5
print(f"P(-1 ≤ X ≤ 1) = {normal_rv.cdf(1) - normal_rv.cdf(-1)}")  # ≈ 0.68
```

3.4 Central Limit Theorem

The CLT states that the sum of many independent random variables tends toward a normal distribution.

python

```python
# Demonstration of Central Limit Theorem
def demonstrate_clt(dist_func, n_samples=1000, sample_sizes=[1, 5, 30, 100]):
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))
    axes = axes.ravel()

    for idx, sample_size in enumerate(sample_sizes):
        # Generate many sample means
        sample_means = []
        for _ in range(n_samples):
            sample = dist_func(size=sample_size)
            sample_means.append(np.mean(sample))

        # Plot histogram
        axes[idx].hist(sample_means, bins=30, density=True, alpha=0.7)
        axes[idx].set_title(f'Sample Size = {sample_size}')

        # Overlay normal distribution
        mean = np.mean(sample_means)
        std = np.std(sample_means)
        x = np.linspace(mean - 4*std, mean + 4*std, 100)
        axes[idx].plot(x, stats.norm.pdf(x, mean, std), 'r-', linewidth=2)

    plt.suptitle('Central Limit Theorem Demonstration')
    plt.tight_layout()
    plt.show()

# Apply to uniform distribution
demonstrate_clt(lambda size: np.random.uniform(0, 1, size))
```

3.5 Important Probability Concepts for ML

1. Maximum Likelihood Estimation (MLE):

python

```python
# MLE for Gaussian distribution
def gaussian_mle(data):
    """Estimate parameters of Gaussian using MLE"""
    mu_mle = np.mean(data)
    sigma_mle = np.std(data, ddof=0)  # Population std
    return mu_mle, sigma_mle

# Generate data and estimate
true_mu, true_sigma = 5, 2
data = np.random.normal(true_mu, true_sigma, 1000)
mu_est, sigma_est = gaussian_mle(data)

print(f"True parameters: μ={true_mu}, σ={true_sigma}")
print(f"MLE estimates: μ={mu_est:.3f}, σ={sigma_est:.3f}")
```

2. Jensen's Inequality: For convex function f and random variable X: E[f(X)] ≥ f(E[X])

3. Information Theory Basics:

python

```python
# Entropy - measure of uncertainty
def entropy(probs):
    """Calculate Shannon entropy"""
    return -np.sum(probs * np.log2(probs + 1e-10))

# Cross-entropy - used in classification
def cross_entropy(y_true, y_pred):
    """Binary cross-entropy loss"""
    return -np.mean(y_true * np.log(y_pred + 1e-10) +
                    (1 - y_true) * np.log(1 - y_pred + 1e-10))

# KL Divergence - measure difference between distributions
def kl_divergence(p, q):
    """KL(P||Q)"""
    return np.sum(p * np.log(p / (q + 1e-10) + 1e-10))
```

Chapter 4: Statistics and Statistical Learning

Statistics provides tools for understanding data and making inferences.

4.1 Descriptive Statistics

Measures of Central Tendency:

python

```python
import numpy as np
import pandas as pd
from scipy import stats

# Generate sample data
np.random.seed(42)
data = np.random.normal(100, 15, 1000)

# Central tendency
mean = np.mean(data)
median = np.median(data)
mode = stats.mode(data, keepdims=True).mode[0]

print(f"Mean: {mean:.2f}")
print(f"Median: {median:.2f}")
print(f"Mode: {mode:.2f}")

# Measures of spread
variance = np.var(data)
std_dev = np.std(data)
iqr = np.percentile(data, 75) - np.percentile(data, 25)

print(f"\nVariance: {variance:.2f}")
print(f"Standard Deviation: {std_dev:.2f}")
print(f"IQR: {iqr:.2f}")

# Skewness and Kurtosis
skewness = stats.skew(data)
kurtosis = stats.kurtosis(data)

print(f"\nSkewness: {skewness:.3f}")  # 0 = symmetric
print(f"Kurtosis: {kurtosis:.3f}")  # 0 = normal tails
```

4.2 Statistical Inference

Hypothesis Testing:

python

```python
# T-test example
def perform_t_test(group1, group2, alpha=0.05):
    """Perform independent samples t-test"""
    t_stat, p_value = stats.ttest_ind(group1, group2)

    print(f"T-statistic: {t_stat:.3f}")
    print(f"P-value: {p_value:.3f}")

    if p_value < alpha:
        print(f"Reject null hypothesis (p < {alpha})")
        print("Groups are significantly different")
    else:
        print(f"Fail to reject null hypothesis (p >= {alpha})")
        print("No significant difference between groups")

    return t_stat, p_value

# Example: A/B testing
control_group = np.random.normal(100, 15, 100)
treatment_group = np.random.normal(105, 15, 100)  # Slightly higher mean

perform_t_test(control_group, treatment_group)
```

Confidence Intervals:

python

```python
def confidence_interval(data, confidence=0.95):
    """Calculate confidence interval for the mean"""
    n = len(data)
    mean = np.mean(data)
    sem = stats.sem(data)  # Standard error of mean

    # T-distribution critical value
    alpha = 1 - confidence
    df = n - 1
    t_critical = stats.t.ppf(1 - alpha/2, df)

    # Margin of error
    margin = t_critical * sem

    ci_lower = mean - margin
    ci_upper = mean + margin

    return ci_lower, ci_upper, mean

# Example
data = np.random.normal(100, 15, 50)
ci_lower, ci_upper, mean = confidence_interval(data)
print(f"95% CI: [{ci_lower:.2f}, {ci_upper:.2f}]")
print(f"Mean: {mean:.2f}")
```

4.3 Correlation and Dependence

python

```python
# Different types of correlation
def analyze_correlation(x, y):
    """Analyze different correlation measures"""
    # Pearson correlation (linear)
    pearson_r, pearson_p = stats.pearsonr(x, y)

    # Spearman correlation (monotonic)
    spearman_r, spearman_p = stats.spearmanr(x, y)

    # Kendall's tau (ordinal)
    kendall_tau, kendall_p = stats.kendalltau(x, y)

    print(f"Pearson r: {pearson_r:.3f} (p={pearson_p:.3f})")
    print(f"Spearman ρ: {spearman_r:.3f} (p={spearman_p:.3f})")
    print(f"Kendall τ: {kendall_tau:.3f} (p={kendall_p:.3f})")

    return pearson_r, spearman_r, kendall_tau

# Example with different relationships
n = 100
x = np.linspace(0, 10, n)

# Linear relationship
y_linear = 2 * x + np.random.normal(0, 2, n)
print("Linear relationship:")
analyze_correlation(x, y_linear)

# Non-linear monotonic
y_exp = np.exp(0.5 * x) + np.random.normal(0, 10, n)
print("\nExponential relationship:")
analyze_correlation(x, y_exp)
```

4.4 Resampling Methods

Bootstrap:

python

```python
def bootstrap_confidence_interval(data, statistic, n_bootstrap=1000, confidence=0.95):
    """Calculate bootstrap confidence interval"""
    n = len(data)
    bootstrap_stats = []

    for _ in range(n_bootstrap):
        # Resample with replacement
        sample = np.random.choice(data, size=n, replace=True)
        bootstrap_stats.append(statistic(sample))

    # Calculate percentiles
    alpha = 1 - confidence
    lower = np.percentile(bootstrap_stats, 100 * alpha/2)
    upper = np.percentile(bootstrap_stats, 100 * (1 - alpha/2))

    return lower, upper, bootstrap_stats

# Example: Bootstrap for median
data = np.random.exponential(2, 100)  # Skewed distribution
lower, upper, bootstrap_dist = bootstrap_confidence_interval(data, np.median)

plt.hist(bootstrap_dist, bins=30, density=True, alpha=0.7)
plt.axvline(lower, color='r', linestyle='--', label=f'CI: [{lower:.2f}, {upper:.2f}]')
plt.axvline(upper, color='r', linestyle='--')
plt.xlabel('Median')
plt.ylabel('Density')
plt.title('Bootstrap Distribution of Median')
plt.legend()
plt.show()
```

Cross-Validation:

python

```python
from sklearn.model_selection import KFold, cross_val_score
from sklearn.linear_model import LinearRegression

def demonstrate_cross_validation(X, y, model, k=5):
    """Demonstrate k-fold cross-validation"""
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    scores = []
    fold = 1

    for train_idx, val_idx in kf.split(X):
        X_train, X_val = X[train_idx], X[val_idx]
        y_train, y_val = y[train_idx], y[val_idx]

        # Train model
        model.fit(X_train, y_train)

        # Evaluate
        score = model.score(X_val, y_val)
        scores.append(score)
        print(f"Fold {fold}: R² = {score:.3f}")
        fold += 1

    print(f"\nMean R²: {np.mean(scores):.3f} (+/- {np.std(scores):.3f})")
    return scores

# Example
X = np.random.randn(100, 5)
y = X @ np.array([1.5, -2.0, 0.5, 1.0, -0.5]) + np.random.randn(100) * 0.5

model = LinearRegression()
scores = demonstrate_cross_validation(X, y, model)
```

Chapter 5: Optimization Theory

Optimization is at the heart of training ML models.

5.1 Convexity and Optimization

Convex Functions: A function f is convex if: $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$ for all $\lambda \in [0,1]$

python

```python
def is_convex_numerically(f, n_tests=100):
    """Test if function is convex numerically"""
    x_range = np.linspace(-10, 10, 20)

    for _ in range(n_tests):
        # Random points and lambda
        x, y = np.random.choice(x_range, 2)
        lam = np.random.uniform(0, 1)

        # Convexity condition
        lhs = f(lam * x + (1 - lam) * y)
        rhs = lam * f(x) + (1 - lam) * f(y)

        if lhs > rhs + 1e-6:  # Numerical tolerance
            return False

    return True

# Test functions
f_convex = lambda x: x**2
f_nonconvex = lambda x: np.sin(x)

print(f"x² is convex: {is_convex_numerically(f_convex)}")
print(f"sin(x) is convex: {is_convex_numerically(f_nonconvex)}")
```

5.2 Gradient Descent

Basic Gradient Descent:

python

```
class GradientDescent:
    def __init__(self, learning_rate=0.01, max_iterations=1000, tolerance=1e-6):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.tolerance = tolerance
        self.history = []

    def optimize(self, f, grad_f, x0):
        """Minimize function f with gradient grad_f starting from x0"""
        x = x0.copy()

        for i in range(self.max_iterations):
            # Compute gradient
            grad = grad_f(x)

            # Update parameters
            x_new = x - self.learning_rate * grad

            # Store history
            self.history.append({
                'iteration': i,
                'x': x.copy(),
                'f': f(x),
                'grad_norm': np.linalg.norm(grad)
            })

            # Check convergence
            if np.linalg.norm(x_new - x) < self.tolerance:
                print(f"Converged at iteration {i}")
                break

            x = x_new

        return x

# Example: Minimize quadratic function
def f(x):
    return x[0]**2 + 4*x[1]**2

def grad_f(x):
    return np.array([2*x[0], 8*x[1]])

# Optimize
optimizer = GradientDescent(learning_rate=0.1)
x0 = np.array([5.0, 5.0])
x_opt = optimizer.optimize(f, grad_f, x0)
```

```
print(f"Optimal point: {x_opt}")
print(f"Optimal value: {f(x_opt)}")

# Visualize optimization path
history = pd.DataFrame(optimizer.history)
plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
plt.plot(history['iteration'], history['f'])
plt.xlabel('Iteration')
plt.ylabel('Function Value')
plt.title('Convergence')

plt.subplot(1, 3, 2)
plt.plot(history['iteration'], history['grad_norm'])
plt.xlabel('Iteration')
plt.ylabel('Gradient Norm')
plt.title('Gradient Magnitude')

plt.subplot(1, 3, 3)
x_path = np.array([h['x'] for h in optimizer.history])
x_range = np.linspace(-6, 6, 100)
y_range = np.linspace(-6, 6, 100)
X, Y = np.meshgrid(x_range, y_range)
Z = X**2 + 4*Y**2

plt.contour(X, Y, Z, levels=20, alpha=0.5)
plt.plot(x_path[:, 0], x_path[:, 1], 'ro-', markersize=5)
plt.xlabel('x₁')
plt.ylabel('x₂')
plt.title('Optimization Path')
plt.tight_layout()
plt.show()
```

5.3 Advanced Optimization Algorithms

1. Momentum:

python

```python
class MomentumGD:
    def __init__(self, learning_rate=0.01, momentum=0.9):
        self.lr = learning_rate
        self.momentum = momentum

    def optimize(self, grad_f, x0, n_iterations=100):
        x = x0.copy()
        velocity = np.zeros_like(x)
        history = []

        for i in range(n_iterations):
            grad = grad_f(x)
            velocity = self.momentum * velocity - self.lr * grad
            x = x + velocity
            history.append(x.copy())

        return x, history
```

2. Adam Optimizer:

python

```
class Adam:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.lr = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon

    def optimize(self, grad_f, x0, n_iterations=1000):
        x = x0.copy()
        m = np.zeros_like(x)  # First moment
        v = np.zeros_like(x)  # Second moment
        history = []

        for t in range(1, n_iterations + 1):
            grad = grad_f(x)

            # Update moments
            m = self.beta1 * m + (1 - self.beta1) * grad
            v = self.beta2 * v + (1 - self.beta2) * grad**2

            # Bias correction
            m_hat = m / (1 - self.beta1**t)
            v_hat = v / (1 - self.beta2**t)

            # Update parameters
            x = x - self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)
            history.append(x.copy())

        return x, history

# Compare optimizers
def rosenbrock(x):
    """Rosenbrock function - challenging for optimization"""
    return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2

def grad_rosenbrock(x):
    dx = -2 * (1 - x[0]) - 400 * x[0] * (x[1] - x[0]**2)
    dy = 200 * (x[1] - x[0]**2)
    return np.array([dx, dy])

# Initial point
x0 = np.array([-1.0, 1.0])

# Run different optimizers
gd = GradientDescent(learning_rate=0.001)
momentum = MomentumGD(learning_rate=0.001)
```

```
adam = Adam(learning_rate=0.01)

# Get paths
_, gd_path = gd.optimize(rosenbrock, grad_rosenbrock, x0)
_, momentum_path = momentum.optimize(grad_rosenbrock, x0, 1000)
_, adam_path = adam.optimize(grad_rosenbrock, x0, 1000)
```

5.4 Constrained Optimization

Lagrange Multipliers:

python

```
# Example: Maximize f(x,y) = xy subject to x + y = 10
from scipy.optimize import minimize

def objective(vars):
    x, y = vars
    return -x * y  # Negative because we minimize

def constraint(vars):
    x, y = vars
    return x + y - 10

# Solve using scipy
constraints = {'type': 'eq', 'fun': constraint}
x0 = [1, 1]
result = minimize(objective, x0, constraints=constraints)

print(f"Optimal point: x={result.x[0]:.2f}, y={result.x[1]:.2f}")
print(f"Maximum value: {-result.fun:.2f}")
```

5.5 Stochastic Optimization

Stochastic Gradient Descent (SGD):

python

```python
class StochasticGD:
    def __init__(self, learning_rate=0.01, batch_size=32):
        self.lr = learning_rate
        self.batch_size = batch_size

    def fit(self, X, y, n_epochs=100):
        n_samples, n_features = X.shape

        # Initialize weights
        self.w = np.random.randn(n_features)
        self.b = 0
        self.losses = []

        for epoch in range(n_epochs):
            # Shuffle data
            indices = np.random.permutation(n_samples)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            epoch_loss = 0
            n_batches = n_samples // self.batch_size

            for i in range(n_batches):
                # Get batch
                start = i * self.batch_size
                end = start + self.batch_size
                X_batch = X_shuffled[start:end]
                y_batch = y_shuffled[start:end]

                # Forward pass
                y_pred = X_batch @ self.w + self.b
                loss = np.mean((y_pred - y_batch)**2)
                epoch_loss += loss

                # Backward pass
                grad_w = 2 * X_batch.T @ (y_pred - y_batch) / self.batch_size
                grad_b = 2 * np.mean(y_pred - y_batch)

                # Update
                self.w -= self.lr * grad_w
                self.b -= self.lr * grad_b

            self.losses.append(epoch_loss / n_batches)

            if epoch % 10 == 0:
                print(f"Epoch {epoch}, Loss: {self.losses[-1]:.4f}")
```

```
    def predict(self, X):
        return X @ self.w + self.b

# Example usage
np.random.seed(42)
X = np.random.randn(1000, 10)
true_w = np.random.randn(10)
y = X @ true_w + np.random.randn(1000) * 0.1

sgd = StochasticGD(learning_rate=0.01, batch_size=32)
sgd.fit(X, y, n_epochs=50)

plt.plot(sgd.losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('SGD Training Loss')
plt.show()
```

Part II: Core Machine Learning

Chapter 6: Introduction to Machine Learning

6.1 What is Machine Learning?

Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed.

Types of Learning:

1. **Supervised Learning**: Learning from labeled examples
2. **Unsupervised Learning**: Finding patterns in unlabeled data
3. **Reinforcement Learning**: Learning through interaction and rewards

6.2 The Machine Learning Pipeline

python

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

class MLPipeline:
    """Complete machine learning pipeline"""

    def __init__(self):
        self.scaler = StandardScaler()
        self.model = None
        self.feature_names = None

    def load_data(self, filepath):
        """Load and explore data"""
        self.data = pd.read_csv(filepath)
        print(f"Data shape: {self.data.shape}")
        print(f"\nData info:")
        print(self.data.info())
        print(f"\nFirst few rows:")
        print(self.data.head())
        return self.data

    def explore_data(self):
        """Exploratory Data Analysis"""
        # Statistical summary
        print("\nStatistical Summary:")
        print(self.data.describe())

        # Missing values
        print("\nMissing values:")
        print(self.data.isnull().sum())

        # Correlations
        numeric_cols = self.data.select_dtypes(include=[np.number]).columns
        if len(numeric_cols) > 1:
            plt.figure(figsize=(10, 8))
            sns.heatmap(self.data[numeric_cols].corr(),
                        annot=True, cmap='coolwarm', center=0)
            plt.title('Feature Correlations')
            plt.show()
```

```
def preprocess_data(self, target_column):
    """Prepare data for training"""
    # Separate features and target
    X = self.data.drop(columns=[target_column])
    y = self.data[target_column]

    # Handle categorical variables
    X = pd.get_dummies(X, drop_first=True)
    self.feature_names = X.columns.tolist()

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    # Scale features
    X_train_scaled = self.scaler.fit_transform(X_train)
    X_test_scaled = self.scaler.transform(X_test)

    return X_train_scaled, X_test_scaled, y_train, y_test

def train_model(self, X_train, y_train, model_type='logistic'):
    """Train machine learning model"""
    if model_type == 'logistic':
        self.model = LogisticRegression(random_state=42)
    # Add other models as needed

    self.model.fit(X_train, y_train)
    return self.model

def evaluate_model(self, X_test, y_test):
    """Evaluate model performance"""
    y_pred = self.model.predict(X_test)

    # Classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()
```

```
        return y_pred

    def feature_importance(self):
        """Analyze feature importance"""
        if hasattr(self.model, 'coef_'):
            importance = abs(self.model.coef_[0])
            feature_importance = pd.DataFrame({
                'feature': self.feature_names,
                'importance': importance
            }).sort_values('importance', ascending=False)

            plt.figure(figsize=(10, 6))
            plt.barh(feature_importance['feature'][:10],
                    feature_importance['importance'][:10])
            plt.xlabel('Importance')
            plt.title('Top 10 Feature Importances')
            plt.show()

            return feature_importance

# Example usage
# pipeline = MLPipeline()
# data = pipeline.load_data('data.csv')
# pipeline.explore_data()
# X_train, X_test, y_train, y_test = pipeline.preprocess_data('target')
# model = pipeline.train_model(X_train, y_train)
# predictions = pipeline.evaluate_model(X_test, y_test)
# importance = pipeline.feature_importance()
```

6.3 Bias-Variance Tradeoff

python

```python
def demonstrate_bias_variance(n_samples=100, n_datasets=100, noise_level=0.3):
    """Demonstrate bias-variance tradeoff"""
    np.random.seed(42)

    # True function
    def true_function(x):
        return np.sin(2 * np.pi * x)

    # Generate multiple datasets
    x_test = np.linspace(0, 1, 100)
    y_true = true_function(x_test)

    predictions = {'simple': [], 'complex': []}

    for _ in range(n_datasets):
        # Generate training data
        x_train = np.random.uniform(0, 1, n_samples)
        y_train = true_function(x_train) + np.random.normal(0, noise_level, n_samples)

        # Simple model (high bias, low variance)
        simple_model = np.poly1d(np.polyfit(x_train, y_train, 1))
        predictions['simple'].append(simple_model(x_test))

        # Complex model (low bias, high variance)
        complex_model = np.poly1d(np.polyfit(x_train, y_train, 15))
        predictions['complex'].append(complex_model(x_test))

    # Calculate bias and variance
    for model_name, preds in predictions.items():
        preds = np.array(preds)

        # Bias: average prediction - true function
        avg_pred = np.mean(preds, axis=0)
        bias = np.mean((avg_pred - y_true)**2)

        # Variance: spread of predictions
        variance = np.mean(np.var(preds, axis=0))

        # Total error
        mse = bias + variance

        print(f"\n{model_name.capitalize()} Model:")
        print(f"Bias²: {bias:.4f}")
        print(f"Variance: {variance:.4f}")
        print(f"Total Error: {mse:.4f}")
```

```
    # Visualization
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 3, 1)
    plt.plot(x_test, y_true, 'k-', label='True Function', linewidth=2)
    for i in range(min(10, n_datasets)):
        plt.plot(x_test, preds[i], alpha=0.3)
    plt.title(f'{model_name.capitalize()} Model - Multiple Fits')
    plt.legend()

    plt.subplot(1, 3, 2)
    plt.plot(x_test, y_true, 'k-', label='True Function', linewidth=2)
    plt.plot(x_test, avg_pred, 'r-', label='Average Prediction', linewidth=2)
    plt.fill_between(x_test,
                     avg_pred - np.std(preds, axis=0),
                     avg_pred + np.std(preds, axis=0),
                     alpha=0.3)
    plt.title('Mean Prediction ± Std')
    plt.legend()

    plt.subplot(1, 3, 3)
    plt.plot(x_test, (avg_pred - y_true)**2, label='Bias²')
    plt.plot(x_test, np.var(preds, axis=0), label='Variance')
    plt.title('Bias² and Variance')
    plt.legend()

    plt.tight_layout()
    plt.show()

demonstrate_bias_variance()
```

6.4 Overfitting and Regularization

python

```
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline

def demonstrate_regularization():
    """Show effect of regularization on overfitting"""
    np.random.seed(42)

    # Generate data
    n_samples = 30
    X = np.sort(np.random.uniform(0, 1, n_samples))
    y = np.sin(2 * np.pi * X) + np.random.normal(0, 0.1, n_samples)

    X_test = np.linspace(0, 1, 300)

    # Create polynomial features
    degrees = [1, 4, 15]
    alphas = [0, 0.01, 0.1, 1.0]

    fig, axes = plt.subplots(len(degrees), len(alphas),
                             figsize=(15, 10), sharex=True, sharey=True)

    for i, degree in enumerate(degrees):
        for j, alpha in enumerate(alphas):
            # Create pipeline
            model = Pipeline([
                ('poly', PolynomialFeatures(degree=degree)),
                ('ridge', Ridge(alpha=alpha))
            ])

            # Fit model
            model.fit(X.reshape(-1, 1), y)
            y_pred = model.predict(X_test.reshape(-1, 1))

            # Plot
            ax = axes[i, j]
            ax.scatter(X, y, s=30, alpha=0.5)
            ax.plot(X_test, y_pred, 'r-', linewidth=2)
            ax.plot(X_test, np.sin(2 * np.pi * X_test), 'k--',
                    alpha=0.5, linewidth=1)

            if i == 0:
                ax.set_title(f'α = {alpha}')
            if j == 0:
                ax.set_ylabel(f'Degree {degree}')
```

```
            # Calculate and display MSE
            y_true_test = np.sin(2 * np.pi * X_test)
            mse = np.mean((y_pred - y_true_test)**2)
            ax.text(0.05, 0.95, f'MSE: {mse:.3f}',
                    transform=ax.transAxes, verticalalignment='top')

    plt.suptitle('Effect of Regularization on Polynomial Regression')
    plt.tight_layout()
    plt.show()

demonstrate_regularization()
```

Chapter 7: Supervised Learning - Regression

7.1 Linear Regression

Mathematical Foundation: Linear regression assumes: $y = X\beta + \varepsilon$

Where:

- y: target variable
- X: feature matrix
- β: coefficients
- ε: error term

Closed-form solution: $\beta = (X^T X)^{-1} X^T y$

python

```python
class LinearRegressionFromScratch:
    """Linear regression implementation from scratch"""

    def __init__(self, fit_intercept=True):
        self.fit_intercept = fit_intercept
        self.coef_ = None
        self.intercept_ = None

    def fit(self, X, y):
        """Fit linear regression using normal equation"""
        X = np.array(X)
        y = np.array(y)

        # Add intercept term
        if self.fit_intercept:
            X = np.c_[np.ones(X.shape[0]), X]

        # Normal equation: θ = (X'X)^(-1)X'y
        XtX = X.T @ X
        Xty = X.T @ y

        # Solve using more stable method than direct inverse
        beta = np.linalg.solve(XtX, Xty)

        # Extract coefficients
        if self.fit_intercept:
            self.intercept_ = beta[0]
            self.coef_ = beta[1:]
        else:
            self.intercept_ = 0
            self.coef_ = beta

        # Calculate R-squared
        y_pred = self.predict(X[:, 1:] if self.fit_intercept else X)
        ss_res = np.sum((y - y_pred)**2)
        ss_tot = np.sum((y - np.mean(y))**2)
        self.r2_score_ = 1 - (ss_res / ss_tot)

        return self

    def predict(self, X):
        """Make predictions"""
        X = np.array(X)
        return X @ self.coef_ + self.intercept_

    def get_diagnostics(self, X, y):
```

```
        """Calculate regression diagnostics"""
        n = len(y)
        p = X.shape[1] + (1 if self.fit_intercept else 0)

        # Predictions and residuals
        y_pred = self.predict(X)
        residuals = y - y_pred

        # Standard error of residuals
        mse = np.sum(residuals**2) / (n - p)
        se_residuals = np.sqrt(mse)

        # Standardized residuals
        standardized_residuals = residuals / se_residuals

        # Calculate leverage (hat values)
        if self.fit_intercept:
            X_with_intercept = np.c_[np.ones(X.shape[0]), X]
        else:
            X_with_intercept = X

        hat_matrix = X_with_intercept @ np.linalg.inv(
            X_with_intercept.T @ X_with_intercept) @ X_with_intercept.T
        leverage = np.diag(hat_matrix)

        # Cook's distance
        cooks_d = (standardized_residuals**2 / p) * (leverage / (1 - leverage)**2)

        return {
            'residuals': residuals,
            'standardized_residuals': standardized_residuals,
            'leverage': leverage,
            'cooks_distance': cooks_d,
            'mse': mse,
            'r_squared': self.r2_score_
        }

# Example with diagnostic plots
def regression_diagnostic_plots(X, y):
    """Create diagnostic plots for linear regression"""
    model = LinearRegressionFromScratch()
    model.fit(X, y)
    diagnostics = model.get_diagnostics(X, y)

    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # Residuals vs Fitted
```

```
    ax1 = axes[0, 0]
    y_pred = model.predict(X)
    ax1.scatter(y_pred, diagnostics['residuals'], alpha=0.5)
    ax1.axhline(y=0, color='r', linestyle='--')
    ax1.set_xlabel('Fitted Values')
    ax1.set_ylabel('Residuals')
    ax1.set_title('Residuals vs Fitted')

    # Q-Q plot
    ax2 = axes[0, 1]
    stats.probplot(diagnostics['standardized_residuals'], dist="norm", plot=ax2)
    ax2.set_title('Normal Q-Q Plot')

    # Scale-Location
    ax3 = axes[1, 0]
    ax3.scatter(y_pred, np.sqrt(np.abs(diagnostics['standardized_residuals'])), alpha=0.5)
    ax3.set_xlabel('Fitted Values')
    ax3.set_ylabel('√|Standardized Residuals|')
    ax3.set_title('Scale-Location')

    # Residuals vs Leverage
    ax4 = axes[1, 1]
    ax4.scatter(diagnostics['leverage'], diagnostics['standardized_residuals'], alpha=0.5)
    ax4.set_xlabel('Leverage')
    ax4.set_ylabel('Standardized Residuals')
    ax4.set_title('Residuals vs Leverage')

    # Add Cook's distance contours
    leverage_range = np.linspace(0.001, 0.2, 50)
    for cooks_val in [0.5, 1.0]:
        std_res = np.sqrt(cooks_val * len(X) / leverage_range *
                          (1 - leverage_range)**2)
        ax4.plot(leverage_range, std_res, 'r--', alpha=0.5)
        ax4.plot(leverage_range, -std_res, 'r--', alpha=0.5)

    plt.suptitle(f"Regression Diagnostics (R² = {diagnostics['r_squared']:.3f})")
    plt.tight_layout()
    plt.show()

    return model, diagnostics

# Generate example data
np.random.seed(42)
X = np.random.randn(100, 3)
true_coef = np.array([2, -1, 0.5])
y = X @ true_coef + 3 + np.random.randn(100) * 0.5
```

```
model, diagnostics = regression_diagnostic_plots(X, y)
print(f"True coefficients: {true_coef}")
print(f"Estimated coefficients: {model.coef_}")
```

7.2 Polynomial Regression

python

```
def polynomial_regression_analysis():
    """Comprehensive polynomial regression analysis"""

    # Generate non-linear data
    np.random.seed(42)
    n_samples = 100
    X = np.sort(np.random.uniform(-3, 3, n_samples))
    y = 0.5 * X**3 - X**2 + 2 * X + np.random.normal(0, 3, n_samples)

    # Test different polynomial degrees
    degrees = range(1, 10)
    train_errors = []
    val_errors = []
    models = []

    # Split data
    X_train, X_val, y_train, y_val = train_test_split(
        X.reshape(-1, 1), y, test_size=0.3, random_state=42
    )

    for degree in degrees:
        # Create polynomial features
        poly = PolynomialFeatures(degree=degree)
        X_train_poly = poly.fit_transform(X_train)
        X_val_poly = poly.transform(X_val)

        # Fit model
        model = LinearRegression()
        model.fit(X_train_poly, y_train)
        models.append((poly, model))

        # Calculate errors
        train_pred = model.predict(X_train_poly)
        val_pred = model.predict(X_val_poly)

        train_errors.append(np.mean((train_pred - y_train)**2))
        val_errors.append(np.mean((val_pred - y_val)**2))

    # Plot results
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    # Training vs Validation Error
    ax1 = axes[0]
    ax1.plot(degrees, train_errors, 'b-o', label='Training Error')
    ax1.plot(degrees, val_errors, 'r-o', label='Validation Error')
    ax1.set_xlabel('Polynomial Degree')
```

```
    ax1.set_ylabel('Mean Squared Error')
    ax1.set_title('Model Selection')
    ax1.legend()
    ax1.set_yscale('log')

    # Best model fit
    best_degree = np.argmin(val_errors) + 1
    poly_best, model_best = models[best_degree - 1]

    ax2 = axes[1]
    X_plot = np.linspace(-3, 3, 300).reshape(-1, 1)
    X_plot_poly = poly_best.transform(X_plot)
    y_plot = model_best.predict(X_plot_poly)

    ax2.scatter(X, y, alpha=0.5, label='Data')
    ax2.plot(X_plot, y_plot, 'r-', linewidth=2,
             label=f'Best Fit (degree={best_degree})')
    ax2.set_xlabel('X')
    ax2.set_ylabel('y')
    ax2.set_title('Best Polynomial Fit')
    ax2.legend()

    # Coefficient magnitude
    ax3 = axes[2]
    for i, (poly, model) in enumerate(models[:6]):
        coef_magnitude = np.abs(model.coef_)
        ax3.semilogy(coef_magnitude, 'o-', label=f'Degree {i+1}')
    ax3.set_xlabel('Coefficient Index')
    ax3.set_ylabel('|Coefficient|')
    ax3.set_title('Coefficient Magnitudes')
    ax3.legend()

    plt.tight_layout()
    plt.show()

    return models, best_degree

models, best_degree = polynomial_regression_analysis()
```

7.3 Regularized Regression

python

```
class RegularizedRegression:
    """Implementation of Ridge, Lasso, and Elastic Net"""

    def __init__(self, alpha=1.0, l1_ratio=0.5, max_iter=1000, tol=1e-4):
        self.alpha = alpha
        self.l1_ratio = l1_ratio  # For elastic net
        self.max_iter = max_iter
        self.tol = tol
        self.coef_ = None
        self.intercept_ = None
        self.history_ = []

    def soft_threshold(self, x, lambda_):
        """Soft thresholding operator for Lasso"""
        return np.sign(x) * np.maximum(np.abs(x) - lambda_, 0)

    def fit_ridge(self, X, y):
        """Analytical solution for Ridge regression"""
        n_samples, n_features = X.shape

        # Add regularization to diagonal
        XtX = X.T @ X
        XtX_reg = XtX + self.alpha * np.eye(n_features)

        # Solve
        Xty = X.T @ y
        self.coef_ = np.linalg.solve(XtX_reg, Xty)

    def fit_lasso(self, X, y):
        """Coordinate descent for Lasso"""
        n_samples, n_features = X.shape
        self.coef_ = np.zeros(n_features)

        for iteration in range(self.max_iter):
            coef_old = self.coef_.copy()

            # Coordinate descent
            for j in range(n_features):
                # Compute residual without j-th feature
                residual = y - X @ self.coef_ + X[:, j] * self.coef_[j]

                # Update j-th coefficient
                rho = X[:, j] @ residual
                self.coef_[j] = self.soft_threshold(rho, self.alpha) / (X[:, j] @ X[:, j])

            # Check convergence
```

```
            if np.linalg.norm(self.coef_ - coef_old) < self.tol:
                break

            # Store history
            loss = 0.5 * np.mean((y - X @ self.coef_)**2) + self.alpha * np.sum(np.abs(self.coe
            self.history_.append(loss)

    def fit_elastic_net(self, X, y):
        """Coordinate descent for Elastic Net"""
        n_samples, n_features = X.shape
        self.coef_ = np.zeros(n_features)

        for iteration in range(self.max_iter):
            coef_old = self.coef_.copy()

            for j in range(n_features):
                residual = y - X @ self.coef_ + X[:, j] * self.coef_[j]
                rho = X[:, j] @ residual

                # Elastic net update
                l1_penalty = self.alpha * self.l1_ratio
                l2_penalty = self.alpha * (1 - self.l1_ratio)

                self.coef_[j] = self.soft_threshold(rho, l1_penalty) / (X[:, j] @ X[:, j] + l2_

            if np.linalg.norm(self.coef_ - coef_old) < self.tol:
                break

    def fit(self, X, y, method='ridge'):
        """Fit regularized regression model"""
        # Standardize features
        self.mean_ = X.mean(axis=0)
        self.std_ = X.std(axis=0)
        X_scaled = (X - self.mean_) / self.std_

        # Center target
        self.y_mean_ = y.mean()
        y_centered = y - self.y_mean_

        # Fit model
        if method == 'ridge':
            self.fit_ridge(X_scaled, y_centered)
        elif method == 'lasso':
            self.fit_lasso(X_scaled, y_centered)
        elif method == 'elastic_net':
            self.fit_elastic_net(X_scaled, y_centered)
```

```
        # Adjust coefficients for original scale
        self.coef_ = self.coef_ / self.std_
        self.intercept_ = self.y_mean_ - np.sum(self.coef_ * self.mean_)

    def predict(self, X):
        """Make predictions"""
        return X @ self.coef_ + self.intercept_

# Demonstration of regularization paths
def plot_regularization_paths():
    """Plot coefficient paths for different regularization strengths"""
    # Generate correlated features
    np.random.seed(42)
    n_samples, n_features = 100, 20

    # Create correlation structure
    correlation_matrix = 0.9 ** np.abs(np.arange(n_features)[:, np.newaxis] - np.arange(n_featu
    X = np.random.multivariate_normal(np.zeros(n_features), correlation_matrix, n_samples)

    # True coefficients (sparse)
    true_coef = np.zeros(n_features)
    true_coef[[0, 5, 10]] = [3, -2, 1.5]
    y = X @ true_coef + np.random.normal(0, 0.5, n_samples)

    # Range of regularization parameters
    alphas = np.logspace(-2, 2, 50)

    # Store coefficients
    ridge_coefs = []
    lasso_coefs = []
    elastic_coefs = []

    for alpha in alphas:
        # Ridge
        model_ridge = RegularizedRegression(alpha=alpha)
        model_ridge.fit(X, y, method='ridge')
        ridge_coefs.append(model_ridge.coef_)

        # Lasso
        model_lasso = RegularizedRegression(alpha=alpha)
        model_lasso.fit(X, y, method='lasso')
        lasso_coefs.append(model_lasso.coef_)

        # Elastic Net
        model_elastic = RegularizedRegression(alpha=alpha, l1_ratio=0.5)
        model_elastic.fit(X, y, method='elastic_net')
        elastic_coefs.append(model_elastic.coef_)
```

```
    # Plot
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    for ax, coefs, title in zip(axes,
                                [ridge_coefs, lasso_coefs, elastic_coefs],
                                ['Ridge', 'Lasso', 'Elastic Net']):
        coefs = np.array(coefs).T
        for i in range(n_features):
            ax.plot(alphas, coefs[i], label=f'β_{i}' if i in [0, 5, 10] else '')

        ax.set_xscale('log')
        ax.set_xlabel('Regularization Parameter (α)')
        ax.set_ylabel('Coefficient Value')
        ax.set_title(f'{title} Coefficient Paths')
        ax.axhline(y=0, color='k', linestyle='--', alpha=0.3)
        if title == 'Ridge':
            ax.legend()

    plt.tight_layout()
    plt.show()

plot_regularization_paths()
```

7.4 Advanced Regression Techniques

Robust Regression:

python

```python
from sklearn.linear_model import HuberRegressor, RANSACRegressor

def demonstrate_robust_regression():
    """Compare robust regression methods with outliers"""
    np.random.seed(42)

    # Generate data with outliers
    n_samples = 100
    n_outliers = 20

    X = np.random.randn(n_samples, 1)
    y = 2 * X.squeeze() + 1 + np.random.randn(n_samples) * 0.5

    # Add outliers
    outlier_indices = np.random.choice(n_samples, n_outliers, replace=False)
    y[outlier_indices] += np.random.randn(n_outliers) * 10

    # Fit different models
    models = {
        'OLS': LinearRegression(),
        'Huber': HuberRegressor(),
        'RANSAC': RANSACRegressor(),
        'Ridge': Ridge(alpha=1.0)
    }

    plt.figure(figsize=(12, 8))

    for i, (name, model) in enumerate(models.items()):
        model.fit(X, y)

        plt.subplot(2, 2, i+1)
        plt.scatter(X, y, alpha=0.5)

        # Highlight outliers
        plt.scatter(X[outlier_indices], y[outlier_indices],
                    color='red', s=50, label='Outliers')

        # Plot fit
        X_plot = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
        y_plot = model.predict(X_plot)
        plt.plot(X_plot, y_plot, 'g-', linewidth=2, label=f'{name} fit')

        # True line
        plt.plot(X_plot, 2 * X_plot + 1, 'k--', alpha=0.5, label='True line')

        plt.xlabel('X')
```

```
        plt.ylabel('y')
        plt.title(f'{name} Regression')
        plt.legend()

    plt.tight_layout()
    plt.show()

demonstrate_robust_regression()
```

Chapter 8: Supervised Learning - Classification

8.1 Logistic Regression

Mathematical Foundation: Logistic regression models the probability of class membership: P(y=1|x) = 1 / (1 + e^(-x^T β))

python

```python
class LogisticRegressionFromScratch:
    """Logistic regression implementation with gradient descent"""

    def __init__(self, learning_rate=0.01, n_iterations=1000, regularization=None, lambda_=0.01
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.regularization = regularization
        self.lambda_ = lambda_
        self.losses = []

    def sigmoid(self, z):
        """Sigmoid activation function"""
        return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

    def cost_function(self, X, y, theta):
        """Binary cross-entropy loss with optional regularization"""
        m = len(y)
        z = X @ theta
        predictions = self.sigmoid(z)

        # Binary cross-entropy
        cost = -np.mean(y * np.log(predictions + 1e-10) +
                        (1 - y) * np.log(1 - predictions + 1e-10))

        # Add regularization
        if self.regularization == 'l2':
            cost += self.lambda_ / (2 * m) * np.sum(theta[1:]**2)
        elif self.regularization == 'l1':
            cost += self.lambda_ / m * np.sum(np.abs(theta[1:]))

        return cost

    def gradient(self, X, y, theta):
        """Compute gradient of cost function"""
        m = len(y)
        predictions = self.sigmoid(X @ theta)

        # Basic gradient
        grad = X.T @ (predictions - y) / m

        # Add regularization gradient
        if self.regularization == 'l2':
            grad[1:] += self.lambda_ / m * theta[1:]
        elif self.regularization == 'l1':
            grad[1:] += self.lambda_ / m * np.sign(theta[1:])
```

```
        return grad

    def fit(self, X, y):
        """Train logistic regression model"""
        # Add intercept term
        X = np.c_[np.ones(X.shape[0]), X]

        # Initialize parameters
        self.theta = np.zeros(X.shape[1])

        # Gradient descent
        for i in range(self.n_iterations):
            # Compute cost
            cost = self.cost_function(X, y, self.theta)
            self.losses.append(cost)

            # Update parameters
            grad = self.gradient(X, y, self.theta)
            self.theta -= self.learning_rate * grad

            if i % 100 == 0:
                print(f"Iteration {i}, Cost: {cost:.4f}")

    def predict_proba(self, X):
        """Predict probabilities"""
        X = np.c_[np.ones(X.shape[0]), X]
        return self.sigmoid(X @ self.theta)

    def predict(self, X, threshold=0.5):
        """Predict classes"""
        return (self.predict_proba(X) >= threshold).astype(int)

    def decision_boundary(self, X):
        """Calculate decision boundary for 2D features"""
        # For 2D: theta0 + theta1*x1 + theta2*x2 = 0
        # x2 = -(theta0 + theta1*x1) / theta2
        if X.shape[1] == 2:
            x1 = np.array([X[:, 0].min(), X[:, 0].max()])
            x2 = -(self.theta[0] + self.theta[1] * x1) / self.theta[2]
            return x1, x2
        else:
            raise ValueError("Decision boundary plotting only supports 2D features")

# Demonstration with visualization
def visualize_logistic_regression():
    """Visualize logistic regression decision boundary"""
    from sklearn.datasets import make_classification
```

```
# Generate 2D classification data
X, y = make_classification(n_samples=200, n_features=2, n_redundant=0,
                           n_informative=2, n_clusters_per_class=1,
                           flip_y=0.1, random_state=42)

# Train model
model = LogisticRegressionFromScratch(learning_rate=0.1, n_iterations=1000)
model.fit(X, y)

# Create visualization
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Decision boundary
ax1 = axes[0]
ax1.scatter(X[y==0, 0], X[y==0, 1], c='blue', label='Class 0', alpha=0.5)
ax1.scatter(X[y==1, 0], X[y==1, 1], c='red', label='Class 1', alpha=0.5)

# Plot decision boundary
x1_boundary, x2_boundary = model.decision_boundary(X)
ax1.plot(x1_boundary, x2_boundary, 'k-', linewidth=2, label='Decision Boundary')

ax1.set_xlabel('Feature 1')
ax1.set_ylabel('Feature 2')
ax1.set_title('Decision Boundary')
ax1.legend()

# Loss curve
ax2 = axes[1]
ax2.plot(model.losses)
ax2.set_xlabel('Iteration')
ax2.set_ylabel('Loss')
ax2.set_title('Training Loss')

# Probability contours
ax3 = axes[2]
xx, yy = np.meshgrid(np.linspace(X[:, 0].min()-1, X[:, 0].max()+1, 100),
                     np.linspace(X[:, 1].min()-1, X[:, 1].max()+1, 100))
Z = model.predict_proba(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

contour = ax3.contourf(xx, yy, Z, levels=20, cmap='RdBu', alpha=0.8)
ax3.scatter(X[y==0, 0], X[y==0, 1], c='blue', edgecolors='k')
ax3.scatter(X[y==1, 0], X[y==1, 1], c='red', edgecolors='k')
plt.colorbar(contour, ax=ax3)
ax3.set_xlabel('Feature 1')
ax3.set_ylabel('Feature 2')
```

```
    ax3.set_title('Probability Contours')

    plt.tight_layout()
    plt.show()

visualize_logistic_regression()
```

8.2 Decision Trees

python

```python
class DecisionTreeNode:
    """Node in a decision tree"""
    def __init__(self, feature=None, threshold=None, left=None, right=None, value=None):
        self.feature = feature  # Feature index for splitting
        self.threshold = threshold  # Threshold value
        self.left = left  # Left subtree
        self.right = right  # Right subtree
        self.value = value  # Leaf node value

class DecisionTreeClassifier:
    """Decision tree classifier implementation"""

    def __init__(self, max_depth=None, min_samples_split=2, criterion='gini'):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.criterion = criterion
        self.tree = None

    def gini_impurity(self, y):
        """Calculate Gini impurity"""
        proportions = np.bincount(y) / len(y)
        return 1 - np.sum(proportions**2)

    def entropy(self, y):
        """Calculate entropy"""
        proportions = np.bincount(y) / len(y)
        return -np.sum(proportions * np.log2(proportions + 1e-10))

    def information_gain(self, X, y, feature, threshold):
        """Calculate information gain from split"""
        # Parent impurity
        if self.criterion == 'gini':
            parent_impurity = self.gini_impurity(y)
        else:
            parent_impurity = self.entropy(y)

        # Split data
        left_mask = X[:, feature] <= threshold
        right_mask = ~left_mask

        if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
            return 0

        # Children impurity
        left_impurity = self.gini_impurity(y[left_mask]) if self.criterion == 'gini' else self.
        right_impurity = self.gini_impurity(y[right_mask]) if self.criterion == 'gini' else sel
```

```
        # Weighted average
        n_left = np.sum(left_mask)
        n_right = np.sum(right_mask)
        n_total = n_left + n_right

        weighted_impurity = (n_left / n_total) * left_impurity + (n_right / n_total) * right_im

        return parent_impurity - weighted_impurity

    def find_best_split(self, X, y):
        """Find best feature and threshold for splitting"""
        best_gain = -np.inf
        best_feature = None
        best_threshold = None

        n_features = X.shape[1]

        for feature in range(n_features):
            thresholds = np.unique(X[:, feature])

            for threshold in thresholds:
                gain = self.information_gain(X, y, feature, threshold)

                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature
                    best_threshold = threshold

        return best_feature, best_threshold, best_gain

    def build_tree(self, X, y, depth=0):
        """Recursively build decision tree"""
        n_samples = len(y)
        n_classes = len(np.unique(y))

        # Stopping criteria
        if (self.max_depth is not None and depth >= self.max_depth) or \
           n_samples < self.min_samples_split or \
           n_classes == 1:
            # Leaf node
            leaf_value = np.bincount(y).argmax()
            return DecisionTreeNode(value=leaf_value)

        # Find best split
        feature, threshold, gain = self.find_best_split(X, y)
```

```
        if gain <= 0:
            leaf_value = np.bincount(y).argmax()
            return DecisionTreeNode(value=leaf_value)

        # Split data
        left_mask = X[:, feature] <= threshold
        right_mask = ~left_mask

        # Recursive build
        left_subtree = self.build_tree(X[left_mask], y[left_mask], depth + 1)
        right_subtree = self.build_tree(X[right_mask], y[right_mask], depth + 1)

        return DecisionTreeNode(feature=feature, threshold=threshold,
                                left=left_subtree, right=right_subtree)

    def fit(self, X, y):
        """Train decision tree"""
        self.tree = self.build_tree(X, y)

    def predict_sample(self, x, node):
        """Predict single sample"""
        if node.value is not None:
            return node.value

        if x[node.feature] <= node.threshold:
            return self.predict_sample(x, node.left)
        else:
            return self.predict_sample(x, node.right)

    def predict(self, X):
        """Predict multiple samples"""
        return np.array([self.predict_sample(x, self.tree) for x in X])

    def print_tree(self, node=None, depth=0):
        """Print tree structure"""
        if node is None:
            node = self.tree

        if node.value is not None:
            print(f"{'  ' * depth}Predict: {node.value}")
        else:
            print(f"{'  ' * depth}Feature {node.feature} <= {node.threshold:.2f}")
            self.print_tree(node.left, depth + 1)
            print(f"{'  ' * depth}Feature {node.feature} > {node.threshold:.2f}")
            self.print_tree(node.right, depth + 1)

# Visualization of decision tree
```

```python
def visualize_decision_tree():
    """Visualize decision tree boundaries"""
    from sklearn.datasets import make_moons

    # Generate data
    X, y = make_moons(n_samples=200, noise=0.3, random_state=42)

    # Train trees with different depths
    depths = [1, 2, 3, 5]
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))
    axes = axes.ravel()

    for i, max_depth in enumerate(depths):
        # Train tree
        tree = DecisionTreeClassifier(max_depth=max_depth)
        tree.fit(X, y)

        # Create mesh
        xx, yy = np.meshgrid(np.linspace(X[:, 0].min()-0.5, X[:, 0].max()+0.5, 100),
                             np.linspace(X[:, 1].min()-0.5, X[:, 1].max()+0.5, 100))
        Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)

        # Plot
        ax = axes[i]
        ax.contourf(xx, yy, Z, alpha=0.3, cmap='RdBu')
        ax.scatter(X[y==0, 0], X[y==0, 1], c='blue', s=50)
        ax.scatter(X[y==1, 0], X[y==1, 1], c='red', s=50)
        ax.set_title(f'Max Depth = {max_depth}')
        ax.set_xlabel('Feature 1')
        ax.set_ylabel('Feature 2')

    plt.tight_layout()
    plt.show()

    # Print tree structure for depth=3
    print("Tree structure (depth=3):")
    tree = DecisionTreeClassifier(max_depth=3)
    tree.fit(X, y)
    tree.print_tree()

visualize_decision_tree()
```

8.3 Support Vector Machines (Preview)

python

```python
# Basic SVM concepts
def svm_concepts_visualization():
    """Visualize SVM concepts"""
    from sklearn.svm import SVC
    from sklearn.datasets import make_blobs

    # Generate linearly separable data
    X, y = make_blobs(n_samples=100, centers=2, n_features=2,
                      cluster_std=0.5, random_state=42)

    # Train SVM
    svm = SVC(kernel='linear', C=1000)
    svm.fit(X, y)

    # Get support vectors
    support_vectors = svm.support_vectors_

    # Create mesh for decision boundary
    xx, yy = np.meshgrid(np.linspace(X[:, 0].min()-1, X[:, 0].max()+1, 100),
                         np.linspace(X[:, 1].min()-1, X[:, 1].max()+1, 100))
    Z = svm.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(10, 8))

    # Plot decision boundary and margins
    plt.contour(xx, yy, Z, levels=[-1, 0, 1], linewidths=[1, 2, 1],
                linestyles=['--', '-', '--'], colors='k')

    # Plot data points
    plt.scatter(X[y==0, 0], X[y==0, 1], c='blue', s=50, label='Class 0')
    plt.scatter(X[y==1, 0], X[y==1, 1], c='red', s=50, label='Class 1')

    # Highlight support vectors
    plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
                s=200, facecolors='none', edgecolors='green', linewidths=2,
                label='Support Vectors')

    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('SVM with Linear Kernel')
    plt.legend()
    plt.show()

svm_concepts_visualization()
```

Chapter 9: Unsupervised Learning

9.1 K-Means Clustering

python

```python
class KMeans:
    """K-Means clustering implementation"""

    def __init__(self, n_clusters=3, max_iters=100, random_state=None):
        self.n_clusters = n_clusters
        self.max_iters = max_iters
        self.random_state = random_state
        self.centroids = None
        self.labels = None
        self.inertia_ = None

    def initialize_centroids(self, X):
        """Initialize centroids using k-means++"""
        np.random.seed(self.random_state)
        n_samples = X.shape[0]

        # Choose first centroid randomly
        centroids = [X[np.random.choice(n_samples)]]

        for _ in range(1, self.n_clusters):
            # Calculate distances to nearest centroid
            distances = np.array([min([np.linalg.norm(x - c) for c in centroids]) for x in X])

            # Choose next centroid with probability proportional to squared distance
            probabilities = distances**2 / np.sum(distances**2)
            cumulative_probs = np.cumsum(probabilities)
            r = np.random.rand()

            for i, p in enumerate(cumulative_probs):
                if r < p:
                    centroids.append(X[i])
                    break

        return np.array(centroids)

    def assign_clusters(self, X):
        """Assign points to nearest centroid"""
        distances = np.array([[np.linalg.norm(x - c) for c in self.centroids] for x in X])
        return np.argmin(distances, axis=1)

    def update_centroids(self, X, labels):
        """Update centroid positions"""
        new_centroids = []
        for i in range(self.n_clusters):
            cluster_points = X[labels == i]
            if len(cluster_points) > 0:
```

```
            new_centroids.append(np.mean(cluster_points, axis=0))
        else:
            # Keep old centroid if cluster is empty
            new_centroids.append(self.centroids[i])
    return np.array(new_centroids)

def fit(self, X):
    """Fit K-means model"""
    # Initialize centroids
    self.centroids = self.initialize_centroids(X)

    for iteration in range(self.max_iters):
        # Assign clusters
        labels = self.assign_clusters(X)

        # Update centroids
        new_centroids = self.update_centroids(X, labels)

        # Check convergence
        if np.allclose(self.centroids, new_centroids):
            print(f"Converged at iteration {iteration}")
            break

        self.centroids = new_centroids

    self.labels = labels

    # Calculate inertia (sum of squared distances to nearest centroid)
    self.inertia_ = sum([np.linalg.norm(X[i] - self.centroids[labels[i]])**2
                         for i in range(len(X))])

    return self

def predict(self, X):
    """Predict cluster for new points"""
    return self.assign_clusters(X)

def elbow_method(self, X, max_k=10):
    """Find optimal k using elbow method"""
    inertias = []
    K = range(1, max_k + 1)

    for k in K:
        kmeans = KMeans(n_clusters=k, random_state=42)
        kmeans.fit(X)
        inertias.append(kmeans.inertia_)
```

```
        # Plot elbow curve
        plt.figure(figsize=(10, 6))
        plt.plot(K, inertias, 'bo-')
        plt.xlabel('Number of clusters (k)')
        plt.ylabel('Inertia')
        plt.title('Elbow Method For Optimal k')
        plt.show()

        return inertias

# Comprehensive K-means demonstration
def demonstrate_kmeans():
    """Demonstrate K-means clustering"""
    from sklearn.datasets import make_blobs

    # Generate data
    X, true_labels = make_blobs(n_samples=300, centers=4, n_features=2,
                                cluster_std=0.5, random_state=42)

    # Fit K-means
    kmeans = KMeans(n_clusters=4, random_state=42)
    kmeans.fit(X)

    # Visualizations
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # Original data with true labels
    axes[0, 0].scatter(X[:, 0], X[:, 1], c=true_labels, cmap='viridis', s=50)
    axes[0, 0].set_title('True Clusters')

    # K-means results
    axes[0, 1].scatter(X[:, 0], X[:, 1], c=kmeans.labels, cmap='viridis', s=50)
    axes[0, 1].scatter(kmeans.centroids[:, 0], kmeans.centroids[:, 1],
                       c='red', s=200, marker='*', edgecolors='black', linewidths=2)
    axes[0, 1].set_title('K-means Clusters')

    # Elbow method
    ax_elbow = plt.subplot(2, 2, 3)
    inertias = []
    K = range(1, 10)
    for k in K:
        km = KMeans(n_clusters=k, random_state=42)
        km.fit(X)
        inertias.append(km.inertia_)

    ax_elbow.plot(K, inertias, 'bo-')
    ax_elbow.set_xlabel('k')
```

```
    ax_elbow.set_ylabel('Inertia')
    ax_elbow.set_title('Elbow Method')

    # Silhouette analysis
    from sklearn.metrics import silhouette_samples, silhouette_score

    ax_sil = plt.subplot(2, 2, 4)
    silhouette_vals = silhouette_samples(X, kmeans.labels)

    y_lower = 10
    for i in range(4):
        cluster_silhouette_vals = silhouette_vals[kmeans.labels == i]
        cluster_silhouette_vals.sort()

        size_cluster_i = cluster_silhouette_vals.shape[0]
        y_upper = y_lower + size_cluster_i

        ax_sil.fill_betweenx(np.arange(y_lower, y_upper), 0,
                             cluster_silhouette_vals, alpha=0.7)
        ax_sil.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

        y_lower = y_upper + 10

    ax_sil.set_xlabel('Silhouette Coefficient')
    ax_sil.set_ylabel('Cluster')
    ax_sil.set_title('Silhouette Analysis')
    ax_sil.axvline(x=silhouette_score(X, kmeans.labels), color='red', linestyle='--')

    plt.tight_layout()
    plt.show()

demonstrate_kmeans()
```

9.2 Hierarchical Clustering

python

```
class AgglomerativeClustering:
    """Agglomerative hierarchical clustering"""

    def __init__(self, n_clusters=2, linkage='single'):
        self.n_clusters = n_clusters
        self.linkage = linkage
        self.labels_ = None
        self.linkage_matrix_ = None

    def compute_distance_matrix(self, X):
        """Compute pairwise distance matrix"""
        n_samples = X.shape[0]
        distances = np.zeros((n_samples, n_samples))

        for i in range(n_samples):
            for j in range(i+1, n_samples):
                dist = np.linalg.norm(X[i] - X[j])
                distances[i, j] = dist
                distances[j, i] = dist

        return distances

    def update_distance_matrix(self, distances, i, j, linkage):
        """Update distance matrix after merging clusters i and j"""
        n = distances.shape[0]

        # Create new distance matrix
        new_distances = np.zeros((n-1, n-1))

        # Indices for new matrix
        indices = [k for k in range(n) if k != j]

        # Copy distances
        for idx_new, idx_old in enumerate(indices[:-1]):
            for jdx_new, jdx_old in enumerate(indices[:-1]):
                if idx_new != jdx_new:
                    new_distances[idx_new, jdx_new] = distances[idx_old, jdx_old]

        # Calculate new distances for merged cluster
        for idx_new, idx_old in enumerate(indices[:-1]):
            if linkage == 'single':
                # Minimum distance
                new_dist = min(distances[i, idx_old], distances[j, idx_old])
            elif linkage == 'complete':
                # Maximum distance
                new_dist = max(distances[i, idx_old], distances[j, idx_old])
```

```
            elif linkage == 'average':
                # Average distance
                new_dist = (distances[i, idx_old] + distances[j, idx_old]) / 2

            new_distances[idx_new, -1] = new_dist
            new_distances[-1, idx_new] = new_dist

        return new_distances, indices

    def fit(self, X):
        """Perform hierarchical clustering"""
        n_samples = X.shape[0]

        # Initialize clusters
        clusters = [[i] for i in range(n_samples)]
        distances = self.compute_distance_matrix(X)

        # Linkage matrix for dendrogram
        linkage_matrix = []

        # Merge clusters until desired number
        while len(clusters) > self.n_clusters:
            # Find closest clusters
            min_dist = np.inf
            merge_i, merge_j = 0, 1

            for i in range(len(clusters)):
                for j in range(i+1, len(clusters)):
                    # Get minimum distance between clusters
                    cluster_distances = []
                    for idx_i in clusters[i]:
                        for idx_j in clusters[j]:
                            if idx_i < n_samples and idx_j < n_samples:
                                cluster_distances.append(distances[idx_i, idx_j])

                    if cluster_distances:
                        if self.linkage == 'single':
                            dist = min(cluster_distances)
                        elif self.linkage == 'complete':
                            dist = max(cluster_distances)
                        elif self.linkage == 'average':
                            dist = np.mean(cluster_distances)

                        if dist < min_dist:
                            min_dist = dist
                            merge_i, merge_j = i, j
```

```
            # Record merge
            linkage_matrix.append([
                min(clusters[merge_i][0], clusters[merge_j][0]),
                max(clusters[merge_i][0], clusters[merge_j][0]),
                min_dist,
                len(clusters[merge_i]) + len(clusters[merge_j])
            ])

            # Merge clusters
            clusters[merge_i].extend(clusters[merge_j])
            clusters.pop(merge_j)

        # Assign labels
        self.labels_ = np.zeros(n_samples, dtype=int)
        for cluster_idx, cluster in enumerate(clusters):
            for sample_idx in cluster:
                if sample_idx < n_samples:
                    self.labels_[sample_idx] = cluster_idx

        self.linkage_matrix_ = np.array(linkage_matrix)
        return self

# Dendrogram visualization
def plot_dendrogram(X, linkage='complete'):
    """Plot hierarchical clustering dendrogram"""
    from scipy.cluster.hierarchy import dendrogram, linkage as scipy_linkage

    # Compute linkage matrix
    linkage_matrix = scipy_linkage(X, method=linkage)

    # Plot dendrogram
    plt.figure(figsize=(12, 8))
    dendrogram(linkage_matrix)
    plt.xlabel('Sample Index')
    plt.ylabel('Distance')
    plt.title(f'Hierarchical Clustering Dendrogram ({linkage} linkage)')
    plt.show()

    return linkage_matrix

# Demonstration
def demonstrate_hierarchical_clustering():
    """Demonstrate hierarchical clustering"""
    from sklearn.datasets import make_moons

    # Generate data
    X, _ = make_moons(n_samples=100, noise=0.1, random_state=42)
```

```
    # Different linkage methods
    linkages = ['single', 'complete', 'average']

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    for i, linkage in enumerate(linkages):
        # Custom implementation
        hc = AgglomerativeClustering(n_clusters=2, linkage=linkage)
        hc.fit(X)

        # Plot results
        axes[0, i].scatter(X[:, 0], X[:, 1], c=hc.labels_, cmap='viridis', s=50)
        axes[0, i].set_title(f'{linkage.capitalize()} Linkage')

        # Sklearn implementation for dendrogram
        from sklearn.cluster import AgglomerativeClustering as SklearnAgglo
        sklearn_hc = SklearnAgglo(n_clusters=2, linkage=linkage)
        labels = sklearn_hc.fit_predict(X)

        axes[1, i].scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50)
        axes[1, i].set_title(f'Sklearn - {linkage.capitalize()}')

    plt.tight_layout()
    plt.show()

    # Plot dendrogram
    plot_dendrogram(X, linkage='complete')

demonstrate_hierarchical_clustering()
```

9.3 DBSCAN - Density-Based Clustering

python

```
class DBSCAN:
    """Density-Based Spatial Clustering of Applications with Noise"""

    def __init__(self, eps=0.5, min_samples=5):
        self.eps = eps
        self.min_samples = min_samples
        self.labels_ = None
        self.core_points_ = None

    def find_neighbors(self, X, point_idx):
        """Find all points within eps distance"""
        distances = np.linalg.norm(X - X[point_idx], axis=1)
        return np.where(distances <= self.eps)[0]

    def fit(self, X):
        """Perform DBSCAN clustering"""
        n_samples = X.shape[0]
        labels = -np.ones(n_samples)  # -1 indicates noise

        # Find core points
        core_points = []
        for i in range(n_samples):
            neighbors = self.find_neighbors(X, i)
            if len(neighbors) >= self.min_samples:
                core_points.append(i)

        self.core_points_ = np.array(core_points)

        # Cluster core points
        cluster_id = 0
        visited = np.zeros(n_samples, dtype=bool)

        for point_idx in core_points:
            if visited[point_idx]:
                continue

            # Start new cluster
            visited[point_idx] = True
            labels[point_idx] = cluster_id

            # Find all reachable points
            neighbors = self.find_neighbors(X, point_idx).tolist()

            while neighbors:
                neighbor_idx = neighbors.pop(0)
```

```
                if not visited[neighbor_idx]:
                    visited[neighbor_idx] = True
                    neighbor_neighbors = self.find_neighbors(X, neighbor_idx)

                    if len(neighbor_neighbors) >= self.min_samples:
                        neighbors.extend(neighbor_neighbors.tolist())

                if labels[neighbor_idx] == -1:
                    labels[neighbor_idx] = cluster_id

            cluster_id += 1

        self.labels_ = labels
        return self

# DBSCAN demonstration
def demonstrate_dbscan():
    """Demonstrate DBSCAN clustering"""
    from sklearn.datasets import make_moons, make_circles

    # Different datasets
    datasets = [
        make_moons(n_samples=200, noise=0.1, random_state=42),
        make_circles(n_samples=200, noise=0.1, factor=0.5, random_state=42)
    ]

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    for i, (X, _) in enumerate(datasets):
        # True clusters (for comparison)
        axes[i, 0].scatter(X[:, 0], X[:, 1], c=_, cmap='viridis', s=50)
        axes[i, 0].set_title('True Clusters')

        # K-means (fails on non-convex clusters)
        kmeans = KMeans(n_clusters=2, random_state=42)
        kmeans.fit(X)
        axes[i, 1].scatter(X[:, 0], X[:, 1], c=kmeans.labels, cmap='viridis', s=50)
        axes[i, 1].set_title('K-means')

        # DBSCAN
        dbscan = DBSCAN(eps=0.3, min_samples=5)
        dbscan.fit(X)

        # Plot with noise points in black
        colors = dbscan.labels_
        axes[i, 2].scatter(X[colors >= 0, 0], X[colors >= 0, 1],
                           c=colors[colors >= 0], cmap='viridis', s=50)
```

```
        axes[i, 2].scatter(X[colors == -1, 0], X[colors == -1, 1],
                           c='black', s=50, marker='x', label='Noise')
        axes[i, 2].set_title('DBSCAN')

        # Highlight core points
        if len(dbscan.core_points_) > 0:
            axes[i, 2].scatter(X[dbscan.core_points_, 0], X[dbscan.core_points_, 1],
                               s=200, facecolors='none', edgecolors='red', linewidths=2)

    plt.tight_layout()
    plt.show()

demonstrate_dbscan()
```

Chapter 10: Model Evaluation and Improvement

10.1 Cross-Validation Strategies

python

```python
class CrossValidation:
    """Implementation of various cross-validation strategies"""

    @staticmethod
    def k_fold_cv(X, y, model, k=5, shuffle=True, random_state=None):
        """K-fold cross-validation"""
        n_samples = len(y)
        indices = np.arange(n_samples)

        if shuffle:
            np.random.seed(random_state)
            np.random.shuffle(indices)

        fold_size = n_samples // k
        scores = []

        for i in range(k):
            # Create train/validation split
            val_start = i * fold_size
            val_end = val_start + fold_size if i < k - 1 else n_samples

            val_indices = indices[val_start:val_end]
            train_indices = np.concatenate([indices[:val_start], indices[val_end:]])

            # Train and evaluate
            X_train, y_train = X[train_indices], y[train_indices]
            X_val, y_val = X[val_indices], y[val_indices]

            model.fit(X_train, y_train)
            score = model.score(X_val, y_val)
            scores.append(score)

            print(f"Fold {i+1}: {score:.4f}")

        return np.array(scores)

    @staticmethod
    def stratified_k_fold_cv(X, y, model, k=5):
        """Stratified K-fold for classification"""
        from sklearn.model_selection import StratifiedKFold

        skf = StratifiedKFold(n_splits=k, shuffle=True, random_state=42)
        scores = []

        for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
            X_train, y_train = X[train_idx], y[train_idx]
```

```
            X_val, y_val = X[val_idx], y[val_idx]

            model.fit(X_train, y_train)
            score = model.score(X_val, y_val)
            scores.append(score)

            print(f"Fold {fold+1}: {score:.4f}")

        return np.array(scores)

    @staticmethod
    def time_series_cv(X, y, model, n_splits=5):
        """Time series cross-validation"""
        n_samples = len(y)
        scores = []

        for i in range(2, n_splits + 2):
            split_point = n_samples * i // (n_splits + 2)

            # Train on all data up to split point
            X_train, y_train = X[:split_point], y[:split_point]

            # Validate on next chunk
            val_start = split_point
            val_end = min(split_point + n_samples // (n_splits + 2), n_samples)
            X_val, y_val = X[val_start:val_end], y[val_start:val_end]

            model.fit(X_train, y_train)
            score = model.score(X_val, y_val)
            scores.append(score)

            print(f"Split {i-1}: Train size={len(y_train)}, Val size={len(y_val)}, Score={score

        return np.array(scores)

# Visualization of CV strategies
def visualize_cv_strategies():
    """Visualize different cross-validation strategies"""
    n_samples = 100
    n_splits = 5

    fig, axes = plt.subplots(3, 1, figsize=(12, 8))

    # K-Fold
    ax1 = axes[0]
    for i in range(n_splits):
        fold_size = n_samples // n_splits
```

```
    val_start = i * fold_size
    val_end = val_start + fold_size if i < n_splits - 1 else n_samples

    # Plot train/val split
    train_mask = np.ones(n_samples)
    train_mask[val_start:val_end] = 0

    ax1.scatter(range(n_samples), [i] * n_samples,
                c=train_mask, cmap='RdBu', s=10, vmin=0, vmax=1)

ax1.set_ylabel('Fold')
ax1.set_title('K-Fold Cross-Validation')
ax1.set_yticks(range(n_splits))

# Time Series Split
ax2 = axes[1]
for i in range(2, n_splits + 2):
    split_point = n_samples * i // (n_splits + 2)

    mask = np.zeros(n_samples)
    mask[:split_point] = 1  # Training
    mask[split_point:min(split_point + n_samples // (n_splits + 2), n_samples)] = 0.5  # Va

    ax2.scatter(range(n_samples), [i-2] * n_samples,
                c=mask, cmap='RdYlBu', s=10, vmin=0, vmax=1)

ax2.set_ylabel('Split')
ax2.set_title('Time Series Cross-Validation')
ax2.set_yticks(range(n_splits))

# Leave-One-Out
ax3 = axes[2]
# Show only first 20 for visibility
for i in range(min(20, n_samples)):
    mask = np.ones(min(20, n_samples))
    mask[i] = 0
    ax3.scatter(range(min(20, n_samples)), [i] * min(20, n_samples),
                c=mask, cmap='RdBu', s=20, vmin=0, vmax=1)

ax3.set_ylabel('Iteration')
ax3.set_xlabel('Sample Index')
ax3.set_title('Leave-One-Out Cross-Validation (first 20 samples)')

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

```
visualize_cv_strategies()
```

10.2 Evaluation Metrics

python

```python
class EvaluationMetrics:
    """Comprehensive evaluation metrics for ML"""

    @staticmethod
    def classification_metrics(y_true, y_pred, y_proba=None):
        """Calculate all classification metrics"""
        from sklearn.metrics import confusion_matrix

        # Basic metrics
        tp = np.sum((y_true == 1) & (y_pred == 1))
        tn = np.sum((y_true == 0) & (y_pred == 0))
        fp = np.sum((y_true == 0) & (y_pred == 1))
        fn = np.sum((y_true == 1) & (y_pred == 0))

        accuracy = (tp + tn) / (tp + tn + fp + fn)
        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0
        f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

        # Matthews Correlation Coefficient
        mcc_num = (tp * tn) - (fp * fn)
        mcc_den = np.sqrt((tp + fp) * (tp + fn) * (tn + fp) * (tn + fn))
        mcc = mcc_num / mcc_den if mcc_den > 0 else 0

        metrics = {
            'accuracy': accuracy,
            'precision': precision,
            'recall': recall,
            'f1_score': f1,
            'mcc': mcc,
            'confusion_matrix': confusion_matrix(y_true, y_pred)
        }

        # ROC and PR curves if probabilities available
        if y_proba is not None:
            from sklearn.metrics import roc_curve, auc, precision_recall_curve

            fpr, tpr, _ = roc_curve(y_true, y_proba)
            roc_auc = auc(fpr, tpr)

            precision_curve, recall_curve, _ = precision_recall_curve(y_true, y_proba)
            pr_auc = auc(recall_curve, precision_curve)

            metrics.update({
                'roc_curve': (fpr, tpr),
                'roc_auc': roc_auc,
```

```
            'pr_curve': (precision_curve, recall_curve),
            'pr_auc': pr_auc
        })

    return metrics

@staticmethod
def regression_metrics(y_true, y_pred):
    """Calculate regression metrics"""
    n = len(y_true)

    # Basic metrics
    mse = np.mean((y_true - y_pred)**2)
    rmse = np.sqrt(mse)
    mae = np.mean(np.abs(y_true - y_pred))

    # R-squared
    ss_res = np.sum((y_true - y_pred)**2)
    ss_tot = np.sum((y_true - np.mean(y_true))**2)
    r2 = 1 - (ss_res / ss_tot)

    # Adjusted R-squared (assuming p features)
    # adj_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)

    # Mean Absolute Percentage Error
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100 if np.all(y_true != 0) else np

    return {
        'mse': mse,
        'rmse': rmse,
        'mae': mae,
        'r2': r2,
        'mape': mape
    }

@staticmethod
def plot_evaluation_results(y_true, y_pred, y_proba=None, model_name='Model'):
    """Comprehensive evaluation plots"""
    fig = plt.figure(figsize=(15, 10))

    if len(np.unique(y_true)) == 2:  # Binary classification
        # Confusion Matrix
        ax1 = plt.subplot(2, 3, 1)
        cm = confusion_matrix(y_true, y_pred)
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax1)
        ax1.set_title('Confusion Matrix')
        ax1.set_xlabel('Predicted')
```

```
    ax1.set_ylabel('True')

    if y_proba is not None:
        # ROC Curve
        ax2 = plt.subplot(2, 3, 2)
        fpr, tpr, _ = roc_curve(y_true, y_proba)
        roc_auc = auc(fpr, tpr)
        ax2.plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})')
        ax2.plot([0, 1], [0, 1], 'k--')
        ax2.set_xlabel('False Positive Rate')
        ax2.set_ylabel('True Positive Rate')
        ax2.set_title('ROC Curve')
        ax2.legend()

        # Precision-Recall Curve
        ax3 = plt.subplot(2, 3, 3)
        precision, recall, _ = precision_recall_curve(y_true, y_proba)
        pr_auc = auc(recall, precision)
        ax3.plot(recall, precision, label=f'PR (AUC = {pr_auc:.3f})')
        ax3.set_xlabel('Recall')
        ax3.set_ylabel('Precision')
        ax3.set_title('Precision-Recall Curve')
        ax3.legend()

        # Probability Distribution
        ax4 = plt.subplot(2, 3, 4)
        ax4.hist(y_proba[y_true == 0], bins=30, alpha=0.5, label='Class 0', density=Tru
        ax4.hist(y_proba[y_true == 1], bins=30, alpha=0.5, label='Class 1', density=Tru
        ax4.set_xlabel('Predicted Probability')
        ax4.set_ylabel('Density')
        ax4.set_title('Probability Distributions')
        ax4.legend()

        # Calibration Plot
        ax5 = plt.subplot(2, 3, 5)
        from sklearn.calibration import calibration_curve
        fraction_pos, mean_pred = calibration_curve(y_true, y_proba, n_bins=10)
        ax5.plot(mean_pred, fraction_pos, 'o-', label=model_name)
        ax5.plot([0, 1], [0, 1], 'k--', label='Perfect')
        ax5.set_xlabel('Mean Predicted Probability')
        ax5.set_ylabel('Fraction of Positives')
        ax5.set_title('Calibration Plot')
        ax5.legend()

else:  # Regression
    # Predicted vs Actual
    ax1 = plt.subplot(2, 2, 1)
```

```
            ax1.scatter(y_true, y_pred, alpha=0.5)
            ax1.plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], 'r--')
            ax1.set_xlabel('True Values')
            ax1.set_ylabel('Predictions')
            ax1.set_title('Predicted vs Actual')

            # Residuals
            ax2 = plt.subplot(2, 2, 2)
            residuals = y_true - y_pred
            ax2.scatter(y_pred, residuals, alpha=0.5)
            ax2.axhline(y=0, color='r', linestyle='--')
            ax2.set_xlabel('Predicted Values')
            ax2.set_ylabel('Residuals')
            ax2.set_title('Residual Plot')

            # Residual Distribution
            ax3 = plt.subplot(2, 2, 3)
            ax3.hist(residuals, bins=30, edgecolor='black')
            ax3.set_xlabel('Residuals')
            ax3.set_ylabel('Frequency')
            ax3.set_title('Residual Distribution')

            # Q-Q Plot
            ax4 = plt.subplot(2, 2, 4)
            stats.probplot(residuals, dist="norm", plot=ax4)
            ax4.set_title('Normal Q-Q Plot')

        plt.suptitle(f'{model_name} Evaluation')
        plt.tight_layout()
        plt.show()

# Example usage
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Generate data
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
                           n_redundant=5, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

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

# Predictions
y_pred = model.predict(X_test)
```

```
y_proba = model.predict_proba(X_test)[:, 1]

# Evaluate
metrics = EvaluationMetrics.classification_metrics(y_test, y_pred, y_proba)
print("Classification Metrics:")
for metric, value in metrics.items():
    if not isinstance(value, tuple) and not isinstance(value, np.ndarray):
        print(f"{metric}: {value:.4f}")

# Plot results
EvaluationMetrics.plot_evaluation_results(y_test, y_pred, y_proba, 'Logistic Regression')
```

10.3 Hyperparameter Tuning

python

```python
class HyperparameterTuning:
    """Various hyperparameter tuning strategies"""

    @staticmethod
    def grid_search(model_class, param_grid, X_train, y_train, X_val, y_val):
        """Exhaustive grid search"""
        from itertools import product

        # Generate all parameter combinations
        param_names = list(param_grid.keys())
        param_values = [param_grid[name] for name in param_names]
        param_combinations = list(product(*param_values))

        best_score = -np.inf
        best_params = None
        results = []

        for params in param_combinations:
            # Create parameter dictionary
            param_dict = dict(zip(param_names, params))

            # Train model
            model = model_class(**param_dict)
            model.fit(X_train, y_train)

            # Evaluate
            score = model.score(X_val, y_val)
            results.append({
                'params': param_dict,
                'score': score
            })

            if score > best_score:
                best_score = score
                best_params = param_dict

        return best_params, best_score, results

    @staticmethod
    def random_search(model_class, param_distributions, X_train, y_train, X_val, y_val, n_iter=
        """Random search with distributions"""
        best_score = -np.inf
        best_params = None
        results = []

        for _ in range(n_iter):
```

```
            # Sample parameters
            param_dict = {}
            for param_name, distribution in param_distributions.items():
                if hasattr(distribution, 'rvs'):
                    # Scipy distribution
                    param_dict[param_name] = distribution.rvs()
                elif isinstance(distribution, list):
                    # List of values
                    param_dict[param_name] = np.random.choice(distribution)
                else:
                    # Assume it's a constant
                    param_dict[param_name] = distribution

            # Train and evaluate
            model = model_class(**param_dict)
            model.fit(X_train, y_train)
            score = model.score(X_val, y_val)

            results.append({
                'params': param_dict,
                'score': score
            })

            if score > best_score:
                best_score = score
                best_params = param_dict

        return best_params, best_score, results

    @staticmethod
    def bayesian_optimization(model_class, param_bounds, X_train, y_train, X_val, y_val, n_iter
        """Simple Bayesian optimization using Gaussian Process"""
        from sklearn.gaussian_process import GaussianProcessRegressor
        from sklearn.gaussian_process.kernels import Matern

        # Initialize
        gp = GaussianProcessRegressor(
            kernel=Matern(nu=2.5),
            n_restarts_optimizer=10,
            normalize_y=True
        )

        # Random initial points
        n_random = 5
        param_names = list(param_bounds.keys())
        X_sampled = []
        y_sampled = []
```

```
    for _ in range(n_random):
        params = []
        param_dict = {}
        for param_name in param_names:
            low, high = param_bounds[param_name]
            value = np.random.uniform(low, high)
            params.append(value)
            param_dict[param_name] = int(value) if param_name.endswith('_int') else value

        X_sampled.append(params)

        # Evaluate
        model = model_class(**param_dict)
        model.fit(X_train, y_train)
        score = model.score(X_val, y_val)
        y_sampled.append(score)

    # Bayesian optimization loop
    for i in range(n_iter - n_random):
        # Fit GP
        gp.fit(X_sampled, y_sampled)

        # Acquisition function (Upper Confidence Bound)
        def acquisition(x):
            mean, std = gp.predict(x.reshape(1, -1), return_std=True)
            return mean + 2 * std  # UCB with κ=2

        # Find next point to sample
        best_acq = -np.inf
        best_x = None

        for _ in range(100):  # Random search for acquisition max
            x = []
            for param_name in param_names:
                low, high = param_bounds[param_name]
                x.append(np.random.uniform(low, high))

            acq = acquisition(np.array(x))
            if acq > best_acq:
                best_acq = acq
                best_x = x

        # Evaluate new point
        param_dict = {}
        for j, param_name in enumerate(param_names):
            value = best_x[j]
```

```
                param_dict[param_name] = int(value) if param_name.endswith('_int') else value

            model = model_class(**param_dict)
            model.fit(X_train, y_train)
            score = model.score(X_val, y_val)

            X_sampled.append(best_x)
            y_sampled.append(score)

        # Return best found
        best_idx = np.argmax(y_sampled)
        best_params = {}
        for j, param_name in enumerate(param_names):
            value = X_sampled[best_idx][j]
            best_params[param_name] = int(value) if param_name.endswith('_int') else value

        return best_params, y_sampled[best_idx], list(zip(X_sampled, y_sampled))

# Demonstration
def demonstrate_hyperparameter_tuning():
    """Compare different hyperparameter tuning methods"""
    from sklearn.datasets import make_classification
    from sklearn.ensemble import RandomForestClassifier
    from scipy.stats import randint, uniform

    # Generate data
    X, y = make_classification(n_samples=500, n_features=20, n_informative=15,
                               n_redundant=5, random_state=42)
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, random_state=42)

    # Parameter spaces
    param_grid = {
        'n_estimators': [10, 50, 100],
        'max_depth': [3, 5, 10],
        'min_samples_split': [2, 5, 10]
    }

    param_distributions = {
        'n_estimators': randint(10, 200),
        'max_depth': randint(3, 20),
        'min_samples_split': randint(2, 20)
    }

    param_bounds = {
        'n_estimators_int': (10, 200),
        'max_depth_int': (3, 20),
        'min_samples_split_int': (2, 20)
```

```
}

# Grid Search
print("Grid Search:")
best_params_grid, best_score_grid, results_grid = HyperparameterTuning.grid_search(
    RandomForestClassifier, param_grid, X_train, y_train, X_val, y_val
)
print(f"Best params: {best_params_grid}")
print(f"Best score: {best_score_grid:.4f}\n")

# Random Search
print("Random Search:")
best_params_random, best_score_random, results_random = HyperparameterTuning.random_search(
    RandomForestClassifier, param_distributions, X_train, y_train, X_val, y_val, n_iter=20
)
print(f"Best params: {best_params_random}")
print(f"Best score: {best_score_random:.4f}\n")

# Bayesian Optimization
print("Bayesian Optimization:")
best_params_bayes, best_score_bayes, results_bayes = HyperparameterTuning.bayesian_optimiza
    RandomForestClassifier, param_bounds, X_train, y_train, X_val, y_val, n_iter=20
)
print(f"Best params: {best_params_bayes}")
print(f"Best score: {best_score_bayes:.4f}")

# Visualize results
plt.figure(figsize=(15, 5))

# Grid search results
plt.subplot(1, 3, 1)
scores_grid = [r['score'] for r in results_grid]
plt.bar(range(len(scores_grid)), scores_grid)
plt.xlabel('Configuration')
plt.ylabel('Score')
plt.title(f'Grid Search (Best: {best_score_grid:.4f})')

# Random search convergence
plt.subplot(1, 3, 2)
scores_random = [r['score'] for r in results_random]
best_so_far = np.maximum.accumulate(scores_random)
plt.plot(best_so_far)
plt.xlabel('Iteration')
plt.ylabel('Best Score')
plt.title(f'Random Search (Best: {best_score_random:.4f})')

# Bayesian optimization convergence
```

```
    plt.subplot(1, 3, 3)
    scores_bayes = [r[1] for r in results_bayes]
    best_so_far_bayes = np.maximum.accumulate(scores_bayes)
    plt.plot(best_so_far_bayes)
    plt.xlabel('Iteration')
    plt.ylabel('Best Score')
    plt.title(f'Bayesian Optimization (Best: {best_score_bayes:.4f})')

    plt.tight_layout()
    plt.show()

demonstrate_hyperparameter_tuning()
```

Part III: Advanced Algorithms

Chapter 11: Ensemble Methods

11.1 Bagging and Random Forests

python

```python
class RandomForestFromScratch:
    """Random Forest implementation"""

    def __init__(self, n_estimators=100, max_depth=None, min_samples_split=2,
                 max_features='sqrt', bootstrap=True):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.max_features = max_features
        self.bootstrap = bootstrap
        self.trees = []
        self.feature_indices = []

    def bootstrap_sample(self, X, y):
        """Create bootstrap sample"""
        n_samples = X.shape[0]
        indices = np.random.choice(n_samples, size=n_samples, replace=True)
        return X[indices], y[indices], indices

    def get_random_features(self, n_features):
        """Select random subset of features"""
        if self.max_features == 'sqrt':
            n_selected = int(np.sqrt(n_features))
        elif self.max_features == 'log2':
            n_selected = int(np.log2(n_features))
        elif isinstance(self.max_features, int):
            n_selected = self.max_features
        else:
            n_selected = n_features

        return np.random.choice(n_features, size=n_selected, replace=False)

    def fit(self, X, y):
        """Train Random Forest"""
        n_samples, n_features = X.shape

        for i in range(self.n_estimators):
            # Bootstrap sampling
            if self.bootstrap:
                X_sample, y_sample, _ = self.bootstrap_sample(X, y)
            else:
                X_sample, y_sample = X, y

            # Random feature selection
            feature_indices = self.get_random_features(n_features)
            X_sample = X_sample[:, feature_indices]
```

```
# Train decision tree
tree = DecisionTree
```