Advanced Machine Learning: Complete Mathematical & Practical Guide

A Comprehensive Deep-Dive into Modern ML Algorithms with Implementation

By Siddharth Vidyarthi

Table of Contents

Chapter 1: Linear Algebra for Machine Learning

1.1 Vector Spaces and Linear Transformations

Linear algebra forms the backbone of machine learning. Every ML algorithm manipulates vectors and matrices to extract patterns from data.

Vector Operations in Detail

A vector $\mathbf{x} \in \mathbb{R}^n$ represents a point in n-dimensional space:
$\mathbf{x} = [x_1, x_2, ..., x_n]^T$

Dot Product Mathematical Properties:
$\mathbf{u} \cdot \mathbf{v} = ||\mathbf{u}|| \, ||\mathbf{v}|| \cos(\theta) = \Sigma_{i=1}^{n} u_i v_i$

Geometric Interpretation: The dot product measures how much two vectors point in the same direction. In ML:

- Similarity between data points
- Feature correlation analysis
- Distance metrics for clustering

Practical Example - Document Similarity:

```
import numpy as np

def cosine_similarity(doc1_vector, doc2_vector):
    """
    Calculate cosine similarity between two document vectors
    Used in text mining and information retrieval
    """
    dot_product = np.dot(doc1_vector, doc2_vector)
    norm_doc1 = np.linalg.norm(doc1_vector)
    norm_doc2 = np.linalg.norm(doc2_vector)
    return dot_product / (norm_doc1 * norm_doc2)

# Example: TF-IDF vectors for two documents
doc1 = np.array([0.5, 0.3, 0.8, 0.1, 0.0])  # TF-IDF weights
doc2 = np.array([0.4, 0.2, 0.7, 0.0, 0.1])
similarity = cosine_similarity(doc1, doc2)
print(f"Document similarity: {similarity:.3f}")
```

Matrix Operations Deep Dive

Matrix Multiplication Computational Complexity:
For matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$:

- Standard algorithm: O(mkn)
- Strassen's algorithm: O(n^2.807)
- Current best: O(n^2.373)

Matrix Multiplication in Neural Networks:

```
import numpy as np

class LinearLayer:
    """
    Fully connected neural network layer implementation
    Demonstrates matrix operations in deep learning
    """
    def __init__(self, input_dim, output_dim):
        self.W = np.random.randn(output_dim, input_dim) * 0.01
        self.b = np.zeros((output_dim, 1))
```

```
    def forward(self, X):
        """
        Forward pass: Y = WX + b
        X: (input_dim, batch_size)
        Y: (output_dim, batch_size)
        """
        self.X = X  # Store for backpropagation
        Z = np.dot(self.W, X) + self.b
        return Z

    def backward(self, dZ):
        """
        Backward pass: compute gradients
        dZ: gradient from next layer
        """
        m = self.X.shape[1]  # batch size

        # Gradients
        self.dW = (1/m) * np.dot(dZ, self.X.T)
        self.db = (1/m) * np.sum(dZ, axis=1, keepdims=True)
        dX = np.dot(self.W.T, dZ)

        return dX

# Usage example
layer = LinearLayer(784, 128)  # 784 input features, 128 output neurons
X = np.random.randn(784, 32)   # Batch of 32 samples
output = layer.forward(X)
print(f"Output shape: {output.shape}")  # (128, 32)
```

Eigenvalue Decomposition - Mathematical Foundation

For square matrix $A \in \mathbb{R}^{n \times n}$, eigenvalues λ and eigenvectors $\mathbf{v}$ satisfy:
$A\mathbf{v} = \lambda\mathbf{v}$

Characteristic Polynomial:
$\det(A - \lambda I) = 0$

Eigendecomposition:
$A = Q\Lambda Q^{-1}$

where:

- $Q = [\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n]$ (eigenvector matrix)
- $\Lambda = \text{diag}(\lambda_1, \lambda_2, ..., \lambda_n)$ (eigenvalue matrix)

Principal Component Analysis Implementation:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
```

```
class PCAFromScratch:
    """
    Principal Component Analysis implementation from scratch
    Demonstrates eigenvalue decomposition in dimensionality reduction
    """
    def __init__(self, n_components):
        self.n_components = n_components
        self.components = None
        self.mean = None

    def fit(self, X):
        """
        Fit PCA on training data
        X: (n_samples, n_features)
        """
        # Center the data
        self.mean = np.mean(X, axis=0)
        X_centered = X - self.mean

        # Compute covariance matrix
        cov_matrix = np.cov(X_centered.T)

        # Eigenvalue decomposition
        eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

        # Sort eigenvalues and eigenvectors in descending order
        idx = np.argsort(eigenvalues)[::-1]
        eigenvalues = eigenvalues[idx]
        eigenvectors = eigenvectors[:, idx]

        # Store first n_components
        self.components = eigenvectors[:, :self.n_components]
        self.explained_variance_ratio = eigenvalues[:self.n_components] / np.sum(eigenva]

    def transform(self, X):
        """
        Transform data to lower dimensional space
        """
        X_centered = X - self.mean
        return np.dot(X_centered, self.components)

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)

# Practical example with Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

pca = PCAFromScratch(n_components=2)
X_pca = pca.fit_transform(X)

print(f"Original shape: {X.shape}")
print(f"Reduced shape: {X_pca.shape}")
print(f"Explained variance ratio: {pca.explained_variance_ratio}")
```

Singular Value Decomposition (SVD) - Complete Analysis

SVD Mathematical Framework:
For any matrix $A \in \mathbb{R}^{m \times n}$:
$A = U\Sigma V^T$

where:

- $U \in \mathbb{R}^{m \times m}$: left singular vectors (orthogonal)
- $\Sigma \in \mathbb{R}^{m \times n}$: singular values (diagonal)
- $V \in \mathbb{R}^{n \times n}$: right singular vectors (orthogonal)

Properties:

- $U^TU = I$, $V^TV = I$ (orthogonality)
- $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq 0$ (singular values in descending order)
- rank(A) = number of non-zero singular values

SVD Applications in ML:

1. **Matrix Completion (Recommendation Systems):**

```
import numpy as np
from scipy.sparse import random
from scipy.linalg import svd

class MatrixFactorization:
    """
    Matrix factorization using SVD for recommendation systems
    Handles missing values through iterative approach
    """
    def __init__(self, n_factors=50, n_iterations=100, learning_rate=0.01, regularizatior
        self.n_factors = n_factors
        self.n_iterations = n_iterations
        self.learning_rate = learning_rate
        self.regularization = regularization

    def fit(self, R):
        """
        R: user-item rating matrix with NaN for missing values
        """
        self.n_users, self.n_items = R.shape

        # Initialize factor matrices
        self.P = np.random.normal(scale=1./self.n_factors, size=(self.n_users, self.n_fac
        self.Q = np.random.normal(scale=1./self.n_factors, size=(self.n_items, self.n_fac

        # Get indices of non-missing values
        self.valid_indices = ~np.isnan(R)

        for iteration in range(self.n_iterations):
            for i in range(self.n_users):
                for j in range(self.n_items):
                    if self.valid_indices[i, j]:
```

```
                        error = R[i, j] - np.dot(self.P[i, :], self.Q[j, :].T)

                        # Update factors using gradient descent
                        P_temp = self.P[i, :].copy()
                        self.P[i, :] += self.learning_rate * (error * self.Q[j, :] - self
                        self.Q[j, :] += self.learning_rate * (error * P_temp - self.regul

        return self

    def predict(self):
        """
        Predict the complete rating matrix
        """
        return np.dot(self.P, self.Q.T)

# Example usage
# Create sample rating matrix (users × items)
n_users, n_items = 100, 50
rating_matrix = np.random.choice([1, 2, 3, 4, 5], size=(n_users, n_items))

# Introduce missing values (80% sparsity typical in recommender systems)
mask = np.random.random((n_users, n_items)) &gt; 0.2
rating_matrix = rating_matrix.astype(float)
rating_matrix[mask] = np.nan

# Fit matrix factorization
mf = MatrixFactorization(n_factors=10)
mf.fit(rating_matrix)
predicted_ratings = mf.predict()

print(f"Original matrix shape: {rating_matrix.shape}")
print(f"Sparsity: {np.isnan(rating_matrix).sum() / rating_matrix.size * 100:.1f}%")
```

2. **Image Compression using SVD:**

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def compress_image_svd(image_path, n_components):
    """
    Compress image using SVD by keeping only top n_components
    """
    # Load and convert image to grayscale
    img = Image.open(image_path).convert('L')
    img_array = np.array(img)

    # Perform SVD
    U, s, Vt = svd(img_array, full_matrices=False)

    # Reconstruct with only n_components
    U_reduced = U[:, :n_components]
    s_reduced = s[:n_components]
    Vt_reduced = Vt[:n_components, :]
```

```
    # Reconstruct image
    compressed_img = np.dot(U_reduced, np.dot(np.diag(s_reduced), Vt_reduced))

    # Calculate compression ratio
    original_size = img_array.shape[0] * img_array.shape[1]
    compressed_size = n_components * (img_array.shape[0] + img_array.shape[1] + 1)
    compression_ratio = original_size / compressed_size

    return compressed_img, compression_ratio

# Example usage (replace with actual image path)
# compressed, ratio = compress_image_svd('sample_image.jpg', 50)
# print(f"Compression ratio: {ratio:.2f}")
```

1.2 Norms and Distance Metrics

Vector Norms - Mathematical Framework

L^p Norm Family:
$||\mathbf{x}||_p = (\Sigma_i |x_i|^p)^{(1/p)}$

Special Cases:

- L_1 norm (Manhattan): $||\mathbf{x}||_1 = \Sigma_i |x_i|$
- L_2 norm (Euclidean): $||\mathbf{x}||_2 = \sqrt{(\Sigma_i x_i^2)}$
- L_∞ norm (Max): $||\mathbf{x}||_\infty = \max_i |x_i|$

Applications in Machine Learning:

1. **Regularization:**

```
import numpy as np

class RegularizedLinearRegression:
    """
    Linear regression with L1 (Lasso) and L2 (Ridge) regularization
    Demonstrates the effect of different norms in regularization
    """
    def __init__(self, regularization='ridge', alpha=1.0):
        self.regularization = regularization
        self.alpha = alpha
        self.weights = None

    def fit(self, X, y):
        """
        Fit regularized linear regression
        """
        # Add bias term
        X_with_bias = np.c_[np.ones(X.shape[0]), X]

        if self.regularization == 'ridge':
            # Ridge regression: (X^T X + αI)^(-1) X^T y
            A = X_with_bias.T @ X_with_bias + self.alpha * np.eye(X_with_bias.shape[1])
```

```
            b = X_with_bias.T @ y
            self.weights = np.linalg.solve(A, b)

        elif self.regularization == 'lasso':
            # Lasso regression using coordinate descent
            self.weights = self._coordinate_descent(X_with_bias, y)

    def _coordinate_descent(self, X, y, max_iter=1000, tol=1e-6):
        """
        Coordinate descent for Lasso regression
        """
        n_features = X.shape[1]
        weights = np.zeros(n_features)

        for iteration in range(max_iter):
            weights_old = weights.copy()

            for j in range(n_features):
                # Compute partial residual
                residual = y - X @ weights + weights[j] * X[:, j]
                rho = X[:, j] @ residual

                # Soft thresholding
                if rho &gt; self.alpha:
                    weights[j] = (rho - self.alpha) / (X[:, j] @ X[:, j])
                elif rho &lt; -self.alpha:
                    weights[j] = (rho + self.alpha) / (X[:, j] @ X[:, j])
                else:
                    weights[j] = 0

            # Check convergence
            if np.linalg.norm(weights - weights_old) &lt; tol:
                break

        return weights

    def predict(self, X):
        """
        Make predictions
        """
        X_with_bias = np.c_[np.ones(X.shape[0]), X]
        return X_with_bias @ self.weights

# Comparison example
np.random.seed(42)
X = np.random.randn(100, 20)  # 100 samples, 20 features
true_weights = np.zeros(20)
true_weights[:5] = [2, -1.5, 3, -2, 1]  # Only 5 features are relevant
y = X @ true_weights + 0.1 * np.random.randn(100)

# Ridge regression
ridge = RegularizedLinearRegression('ridge', alpha=1.0)
ridge.fit(X, y)

# Lasso regression
lasso = RegularizedLinearRegression('lasso', alpha=0.1)
```

```
lasso.fit(X, y)

print("True weights (first 10):", true_weights[:10])
print("Ridge weights (first 10):", ridge.weights[1:11])  # Skip bias term
print("Lasso weights (first 10):", lasso.weights[1:11])  # Skip bias term
print("Lasso sparsity:", np.sum(np.abs(lasso.weights[1:]) &lt; 1e-6))
```

Distance Metrics in ML

Minkowski Distance Family:
$d(\mathbf{x}, \mathbf{y}) = (\Sigma_i |x_i - y_i|^p)^{(1/p)}$

Special Cases and Applications:

1. **K-Means Clustering with Different Distance Metrics:**

```
import numpy as np
from sklearn.datasets import make_blobs

class KMeansCustomDistance:
    """
    K-Means clustering with custom distance metrics
    Demonstrates impact of distance choice on clustering
    """
    def __init__(self, k=3, max_iters=100, distance_metric='euclidean'):
        self.k = k
        self.max_iters = max_iters
        self.distance_metric = distance_metric

    def _distance(self, x1, x2):
        """
        Compute distance based on chosen metric
        """
        if self.distance_metric == 'euclidean':
            return np.sqrt(np.sum((x1 - x2) ** 2))
        elif self.distance_metric == 'manhattan':
            return np.sum(np.abs(x1 - x2))
        elif self.distance_metric == 'cosine':
            return 1 - np.dot(x1, x2) / (np.linalg.norm(x1) * np.linalg.norm(x2))

    def fit(self, X):
        """
        Fit K-means clustering
        """
        n_samples, n_features = X.shape

        # Initialize centroids randomly
        self.centroids = X[np.random.choice(n_samples, self.k, replace=False)]

        for _ in range(self.max_iters):
            # Assign points to closest centroid
            clusters = []
            for _ in range(self.k):
                clusters.append([])
```

```
            for point in X:
                distances = [self._distance(point, centroid) for centroid in self.centroi
                cluster_idx = np.argmin(distances)
                clusters[cluster_idx].append(point)

            # Update centroids
            prev_centroids = self.centroids.copy()
            for i in range(self.k):
                if clusters[i]:  # Avoid empty clusters
                    self.centroids[i] = np.mean(clusters[i], axis=0)

            # Check for convergence
            if np.allclose(prev_centroids, self.centroids):
                break

        self.clusters = clusters
        return self

    def predict(self, X):
        """
        Predict cluster assignments for new data
        """
        predictions = []
        for point in X:
            distances = [self._distance(point, centroid) for centroid in self.centroids]
            predictions.append(np.argmin(distances))
        return np.array(predictions)

# Generate sample data
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Compare different distance metrics
metrics = ['euclidean', 'manhattan', 'cosine']
results = {}

for metric in metrics:
    kmeans = KMeansCustomDistance(k=4, distance_metric=metric)
    kmeans.fit(X)
    predictions = kmeans.predict(X)
    results[metric] = {
        'centroids': kmeans.centroids,
        'predictions': predictions
    }
    print(f"{metric.capitalize()} clustering completed")
```

Chapter 2: Calculus & Optimization Theory

2.1 Derivatives and Gradients in Machine Learning

Partial Derivatives - Foundation of ML Optimization

For multivariate function $f(x_1, x_2, ..., x_n)$, the partial derivative with respect to x_i:

$\partial f/\partial x_i = \lim_{h \to 0} [f(x_1,...,x_{i+h},...,x_n) - f(x_1,...,x_i,...,x_n)] / h$

Gradient Vector:
$\nabla f = [\partial f/\partial x_1, \partial f/\partial x_2, ..., \partial f/\partial x_n]^T$

Geometric Interpretation: The gradient points in the direction of steepest increase of the function.

Chain Rule - Backbone of Backpropagation

For composite function $f(g(x))$:
$df/dx = (df/dg)(dg/dx)$

Multivariate Chain Rule:
For $f(u_1(x_1,...,x_n), u_2(x_1,...,x_n), ..., u_m(x_1,...,x_n))$:

$\partial f/\partial x_i = \Sigma_j (\partial f/\partial u_j)(\partial u_j/\partial x_i)$

Neural Network Application:

```
import numpy as np

class NeuralNetworkGradients:
    """
    Neural network implementation with detailed gradient computation
    Demonstrates chain rule application in backpropagation
    """
    def __init__(self, layers):
        """
        Initialize network with specified layer sizes
        layers: list of integers representing neurons in each layer
        """
        self.layers = layers
        self.weights = []
        self.biases = []

        # Initialize weights and biases
        for i in range(len(layers) - 1):
            W = np.random.randn(layers[i+1], layers[i]) * np.sqrt(2.0 / layers[i])
            b = np.zeros((layers[i+1], 1))
            self.weights.append(W)
            self.biases.append(b)

    def sigmoid(self, z):
        """Sigmoid activation function"""
        return 1 / (1 + np.exp(-np.clip(z, -250, 250)))
```

```
    def sigmoid_derivative(self, z):
        """Derivative of sigmoid function"""
        s = self.sigmoid(z)
        return s * (1 - s)

    def relu(self, z):
        """ReLU activation function"""
        return np.maximum(0, z)

    def relu_derivative(self, z):
        """Derivative of ReLU function"""
        return (z &gt; 0).astype(float)

    def forward_pass(self, X):
        """
        Forward pass with detailed intermediate storage
        X: input data (features × samples)
        """
        self.activations = [X]  # Store all activations
        self.z_values = []      # Store all pre-activation values

        A = X
        for i in range(len(self.weights)):
            Z = np.dot(self.weights[i], A) + self.biases[i]
            self.z_values.append(Z)

            # Apply activation function (ReLU for hidden, sigmoid for output)
            if i == len(self.weights) - 1:  # Output layer
                A = self.sigmoid(Z)
            else:  # Hidden layers
                A = self.relu(Z)

            self.activations.append(A)

        return A

    def backward_pass(self, X, y, learning_rate=0.01):
        """
        Backward pass with detailed gradient computation
        """
        m = X.shape[1]  # Number of samples

        # Initialize gradient storage
        dW = [np.zeros_like(w) for w in self.weights]
        db = [np.zeros_like(b) for b in self.biases]

        # Output layer error (derivative of cost w.r.t. output)
        dA = -(y / self.activations[-1] - (1 - y) / (1 - self.activations[-1]))

        # Backpropagate through all layers
        for l in reversed(range(len(self.weights))):
            # Current layer pre-activation
            Z = self.z_values[l]

            # Gradient of cost w.r.t. pre-activation
            if l == len(self.weights) - 1:  # Output layer
```

```
                dZ = dA * self.sigmoid_derivative(Z)
            else:  # Hidden layers
                dZ = dA * self.relu_derivative(Z)

            # Gradients w.r.t. weights and biases
            dW[l] = (1/m) * np.dot(dZ, self.activations[l].T)
            db[l] = (1/m) * np.sum(dZ, axis=1, keepdims=True)

            # Gradient w.r.t. previous layer activation
            if l &gt; 0:
                dA = np.dot(self.weights[l].T, dZ)

        # Update parameters
        for i in range(len(self.weights)):
            self.weights[i] -= learning_rate * dW[i]
            self.biases[i] -= learning_rate * db[i]

        return dW, db

    def compute_cost(self, predictions, y):
        """
        Binary cross-entropy cost function
        """
        m = y.shape[1]
        cost = -(1/m) * np.sum(y * np.log(predictions + 1e-8) +
                               (1 - y) * np.log(1 - predictions + 1e-8))
        return cost

# Example usage and gradient verification
def numerical_gradient(f, x, h=1e-5):
    """
    Compute numerical gradient for gradient checking
    """
    grad = np.zeros_like(x)
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])

    while not it.finished:
        idx = it.multi_index
        old_value = x[idx]

        x[idx] = old_value + h
        fxh_pos = f(x)

        x[idx] = old_value - h
        fxh_neg = f(x)

        grad[idx] = (fxh_pos - fxh_neg) / (2 * h)
        x[idx] = old_value
        it.iternext()

    return grad

# Create sample dataset
np.random.seed(42)
X = np.random.randn(2, 100)  # 2 features, 100 samples
y = ((X[0] + X[1]) &gt; 0).astype(int).reshape(1, -1)  # XOR-like problem
```

```
# Initialize and train network
nn = NeuralNetworkGradients([2, 4, 1])

print("Training neural network with gradient computation...")
for epoch in range(1000):
    predictions = nn.forward_pass(X)
    dW, db = nn.backward_pass(X, y, learning_rate=0.1)

    if epoch % 200 == 0:
        cost = nn.compute_cost(predictions, y)
        accuracy = np.mean((predictions &gt; 0.5) == y) * 100
        print(f"Epoch {epoch}: Cost = {cost:.4f}, Accuracy = {accuracy:.1f}%")
```

2.2 Optimization Theory Deep Dive

Convex Optimization Fundamentals

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

First-Order Condition:
f is convex $\iff f(y) \geq f(x) + \nabla f(x)^T(y - x)$ for all x, y

Second-Order Condition:
f is convex $\iff \nabla^2 f(x) \succeq 0$ (Hessian is positive semidefinite)

Gradient Descent Variants - Mathematical Analysis

1. Batch Gradient Descent:
$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$

Convergence Analysis:
For convex function with Lipschitz gradient (L-smooth):
$J(\theta_t) - J(\theta^*) \leq (||\theta_0 - \theta*||^2)/(2\alpha t)$ for $\alpha \leq 1/L$

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

class OptimizationComparison:
    """
    Comprehensive comparison of optimization algorithms
    with convergence analysis and visualization
    """

    def __init__(self, objective_func, gradient_func, hessian_func=None):
        self.objective_func = objective_func
        self.gradient_func = gradient_func
        self.hessian_func = hessian_func

    def batch_gradient_descent(self, x0, learning_rate=0.01, max_iter=1000, tol=1e-6):
```

```
        """Standard batch gradient descent"""
        x = x0.copy()
        history = [x.copy()]
        costs = [self.objective_func(x)]

        for i in range(max_iter):
            grad = self.gradient_func(x)
            x = x - learning_rate * grad

            history.append(x.copy())
            costs.append(self.objective_func(x))

            if np.linalg.norm(grad) &lt; tol:
                break

        return np.array(history), np.array(costs)

    def momentum_gradient_descent(self, x0, learning_rate=0.01, momentum=0.9,
                                  max_iter=1000, tol=1e-6):
        """Gradient descent with momentum"""
        x = x0.copy()
        v = np.zeros_like(x)  # velocity
        history = [x.copy()]
        costs = [self.objective_func(x)]

        for i in range(max_iter):
            grad = self.gradient_func(x)
            v = momentum * v + learning_rate * grad
            x = x - v

            history.append(x.copy())
            costs.append(self.objective_func(x))

            if np.linalg.norm(grad) &lt; tol:
                break

        return np.array(history), np.array(costs)

    def adagrad(self, x0, learning_rate=0.1, eps=1e-8, max_iter=1000, tol=1e-6):
        """AdaGrad optimizer"""
        x = x0.copy()
        G = np.zeros_like(x)  # Accumulated squared gradients
        history = [x.copy()]
        costs = [self.objective_func(x)]

        for i in range(max_iter):
            grad = self.gradient_func(x)
            G += grad ** 2
            x = x - learning_rate * grad / (np.sqrt(G) + eps)

            history.append(x.copy())
            costs.append(self.objective_func(x))

            if np.linalg.norm(grad) &lt; tol:
                break
```

```
        return np.array(history), np.array(costs)

    def adam(self, x0, learning_rate=0.001, beta1=0.9, beta2=0.999,
             eps=1e-8, max_iter=1000, tol=1e-6):
        """Adam optimizer"""
        x = x0.copy()
        m = np.zeros_like(x)  # First moment
        v = np.zeros_like(x)  # Second moment
        history = [x.copy()]
        costs = [self.objective_func(x)]

        for t in range(1, max_iter + 1):
            grad = self.gradient_func(x)

            # Update biased first moment estimate
            m = beta1 * m + (1 - beta1) * grad
            # Update biased second raw moment estimate
            v = beta2 * v + (1 - beta2) * grad ** 2

            # Compute bias-corrected first moment estimate
            m_hat = m / (1 - beta1 ** t)
            # Compute bias-corrected second raw moment estimate
            v_hat = v / (1 - beta2 ** t)

            # Update parameters
            x = x - learning_rate * m_hat / (np.sqrt(v_hat) + eps)

            history.append(x.copy())
            costs.append(self.objective_func(x))

            if np.linalg.norm(grad) &lt; tol:
                break

        return np.array(history), np.array(costs)

    def newton_method(self, x0, max_iter=1000, tol=1e-6):
        """Newton's method (requires Hessian)"""
        if self.hessian_func is None:
            raise ValueError("Hessian function required for Newton's method")

        x = x0.copy()
        history = [x.copy()]
        costs = [self.objective_func(x)]

        for i in range(max_iter):
            grad = self.gradient_func(x)
            hess = self.hessian_func(x)

            # Solve Hx = -g
            try:
                direction = np.linalg.solve(hess, -grad)
                x = x + direction
            except np.linalg.LinAlgError:
                print("Singular Hessian matrix")
                break
```

```
            history.append(x.copy())
            costs.append(self.objective_func(x))

            if np.linalg.norm(grad) &lt; tol:
                break

        return np.array(history), np.array(costs)

# Example: Optimize Rosenbrock function
def rosenbrock(x):
    """Rosenbrock function: f(x,y) = (a-x)² + b(y-x²)²"""
    a, b = 1, 100
    return (a - x[0])**2 + b * (x[1] - x[0]**2)**2

def rosenbrock_gradient(x):
    """Gradient of Rosenbrock function"""
    a, b = 1, 100
    dx = -2*(a - x[0]) - 4*b*x[0]*(x[1] - x[0]**2)
    dy = 2*b*(x[1] - x[0]**2)
    return np.array([dx, dy])

def rosenbrock_hessian(x):
    """Hessian of Rosenbrock function"""
    a, b = 1, 100
    h11 = 2 - 4*b*(x[1] - x[0]**2) + 8*b*x[0]**2
    h12 = -4*b*x[0]
    h21 = -4*b*x[0]
    h22 = 2*b
    return np.array([[h11, h12], [h21, h22]])

# Initialize optimization comparison
optimizer = OptimizationComparison(rosenbrock, rosenbrock_gradient, rosenbrock_hessian)

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

# Run different optimizers
methods = {
    'Gradient Descent': lambda: optimizer.batch_gradient_descent(x0, 0.001, 5000),
    'Momentum': lambda: optimizer.momentum_gradient_descent(x0, 0.001, 0.9, 5000),
    'AdaGrad': lambda: optimizer.adagrad(x0, 0.1, max_iter=5000),
    'Adam': lambda: optimizer.adam(x0, 0.01, max_iter=5000),
    'Newton': lambda: optimizer.newton_method(x0, 100)
}

results = {}
for name, method in methods.items():
    try:
        history, costs = method()
        results[name] = {'history': history, 'costs': costs}
        final_point = history[-1]
        final_cost = costs[-1]
        iterations = len(costs) - 1
        print(f"{name}: Final point = [{final_point[0]:.4f}, {final_point[1]:.4f}], "
              f"Final cost = {final_cost:.6f}, Iterations = {iterations}")
```

```
    except Exception as e:
        print(f"{name} failed: {e}")
```

Line Search Methods

Wolfe Conditions:
For step size α, require:

1. Sufficient decrease: $f(x + \alpha p) \leq f(x) + c_1 \alpha \nabla f(x)^T p$
2. Curvature condition: $\nabla f(x + \alpha p)^T p \geq c_2 \nabla f(x)^T p$

where $0 < c_1 < c_2 < 1$

```
def backtracking_line_search(objective_func, gradient_func, x, direction,
                             alpha_init=1.0, rho=0.8, c=1e-4):
    """
    Backtracking line search satisfying Armijo condition
    """
    alpha = alpha_init
    fx = objective_func(x)
    grad_x = gradient_func(x)
    descent_condition = c * np.dot(grad_x, direction)

    while objective_func(x + alpha * direction) &gt; fx + alpha * descent_condition:
        alpha *= rho

        if alpha &lt; 1e-10:  # Prevent infinite loop
            break

    return alpha

# Integration with gradient descent
class GradientDescentWithLineSearch:
    """
    Gradient descent with adaptive step size using line search
    """
    def __init__(self, objective_func, gradient_func):
        self.objective_func = objective_func
        self.gradient_func = gradient_func

    def optimize(self, x0, max_iter=1000, tol=1e-6):
        x = x0.copy()
        history = [x.copy()]
        costs = [self.objective_func(x)]

        for i in range(max_iter):
            grad = self.gradient_func(x)

            if np.linalg.norm(grad) &lt; tol:
                break

            # Search direction (negative gradient)
            direction = -grad
```

```
            # Line search for optimal step size
            alpha = backtracking_line_search(
                self.objective_func, self.gradient_func, x, direction
            )

            # Update position
            x = x + alpha * direction

            history.append(x.copy())
            costs.append(self.objective_func(x))

        return np.array(history), np.array(costs)

# Example usage
line_search_optimizer = GradientDescentWithLineSearch(rosenbrock, rosenbrock_gradient)
history_ls, costs_ls = line_search_optimizer.optimize(np.array([-1.0, 1.0]))

print(f"Line Search GD: Final point = [{history_ls[-1][0]:.4f}, {history_ls[-1][1]:.4f}],
      f"Final cost = {costs_ls[-1]:.6f}, Iterations = {len(costs_ls)-1}")
```

Chapter 5: Linear & Logistic Regression (Complete Guide)

5.1 Linear Regression - Mathematical Foundation

Linear regression models the relationship between dependent variable y and independent variables X:

Simple Linear Regression:
$y = \beta_0 + \beta_1 x + \varepsilon$

Multiple Linear Regression:
$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \varepsilon$

Matrix Form:
$\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon}$

where:

- $\mathbf{y} \in \mathbb{R}^n$: response vector
- $X \in \mathbb{R}^{n \times p}$: design matrix
- $\boldsymbol{\beta} \in \mathbb{R}^p$: parameter vector
- $\boldsymbol{\varepsilon} \in \mathbb{R}^n$: error vector

Ordinary Least Squares (OLS) Derivation

Objective Function:
Minimize: $RSS(\boldsymbol{\beta}) = ||\mathbf{y} - X\boldsymbol{\beta}||^2$

Normal Equations:
$\nabla RSS(\boldsymbol{\beta}) = -2X^T(\mathbf{y} - X\boldsymbol{\beta}) = 0$

Closed-Form Solution:
$\hat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T \mathbf{y}$

Geometric Interpretation:
The OLS solution projects **y** onto the column space of X.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
import seaborn as sns

class LinearRegressionFromScratch:
    """
    Complete implementation of linear regression with statistical analysis
    """
    def __init__(self, fit_intercept=True):
        self.fit_intercept = fit_intercept
        self.coefficients = None
        self.intercept = None
        self.X_train = None
        self.y_train = None

    def _add_intercept(self, X):
        """Add intercept column to design matrix"""
        return np.column_stack([np.ones(X.shape[0]), X])

    def fit(self, X, y):
        """
        Fit linear regression using normal equations
        """
        self.X_train = X.copy()
        self.y_train = y.copy()

        if self.fit_intercept:
            X_with_intercept = self._add_intercept(X)
        else:
            X_with_intercept = X

        # Normal equations: β = (X^T X)^(-1) X^T y
        try:
            XtX = X_with_intercept.T @ X_with_intercept
            Xty = X_with_intercept.T @ y
            params = np.linalg.solve(XtX, Xty)

            if self.fit_intercept:
                self.intercept = params[0]
                self.coefficients = params[1:]
            else:
                self.intercept = 0
                self.coefficients = params

        except np.linalg.LinAlgError:
            # Use pseudoinverse if X^T X is singular
            params = np.linalg.pinv(X_with_intercept) @ y
```

```
            if self.fit_intercept:
                self.intercept = params[0]
                self.coefficients = params[1:]
            else:
                self.intercept = 0
                self.coefficients = params

        # Compute residuals and statistics
        self._compute_statistics(X, y)

    def predict(self, X):
        """Make predictions"""
        return X @ self.coefficients + self.intercept

    def _compute_statistics(self, X, y):
        """Compute regression statistics"""
        n, p = X.shape
        y_pred = self.predict(X)
        residuals = y - y_pred

        # Sum of squares
        self.tss = np.sum((y - np.mean(y))**2)  # Total sum of squares
        self.rss = np.sum(residuals**2)         # Residual sum of squares
        self.ess = self.tss - self.rss          # Explained sum of squares

        # R-squared
        self.r_squared = 1 - self.rss / self.tss
        self.adj_r_squared = 1 - (self.rss / (n - p - 1)) / (self.tss / (n - 1))

        # Standard errors
        self.mse = self.rss / (n - p - 1)  # Mean squared error

        if self.fit_intercept:
            X_with_intercept = self._add_intercept(X)
        else:
            X_with_intercept = X

        try:
            # Covariance matrix of coefficients
            XtX_inv = np.linalg.inv(X_with_intercept.T @ X_with_intercept)
            self.covariance_matrix = self.mse * XtX_inv
            self.standard_errors = np.sqrt(np.diag(self.covariance_matrix))
        except np.linalg.LinAlgError:
            self.covariance_matrix = None
            self.standard_errors = None

        # t-statistics and p-values
        if self.standard_errors is not None:
            if self.fit_intercept:
                all_params = np.concatenate([[self.intercept], self.coefficients])
            else:
                all_params = self.coefficients

            self.t_statistics = all_params / self.standard_errors
            # p-values (two-tailed t-test)
            from scipy.stats import t
```

```
            self.p_values = 2 * (1 - t.cdf(np.abs(self.t_statistics), df=n-p-1))

        self.residuals = residuals

    def summary(self):
        """Print regression summary"""
        print("Linear Regression Summary")
        print("=" * 50)
        print(f"R-squared: {self.r_squared:.4f}")
        print(f"Adjusted R-squared: {self.adj_r_squared:.4f}")
        print(f"Mean Squared Error: {self.mse:.4f}")
        print(f"Residual Standard Error: {np.sqrt(self.mse):.4f}")
        print("\nCoefficients:")

        if self.standard_errors is not None:
            print(f"{'Variable':&lt;12} {'Coefficient':&lt;12} {'Std Error':&lt;12} {'t-s
            print("-" * 65)

            if self.fit_intercept:
                print(f"{'Intercept':&lt;12} {self.intercept:&lt;12.4f} {self.standard_er
                      f"{self.t_statistics[0]:&lt;10.4f} {self.p_values[0]:&lt;10.4f}")

            for i, coef in enumerate(self.coefficients):
                idx = i + 1 if self.fit_intercept else i
                print(f"{'X' + str(i+1):&lt;12} {coef:&lt;12.4f} {self.standard_errors[id
                      f"{self.t_statistics[idx]:&lt;10.4f} {self.p_values[idx]:&lt;10.4f}
        else:
            print(f"{'Intercept':&lt;12} {self.intercept:.4f}")
            for i, coef in enumerate(self.coefficients):
                print(f"{'X' + str(i+1):&lt;12} {coef:.4f}")

    def plot_diagnostics(self):
        """Plot regression diagnostics"""
        fig, axes = plt.subplots(2, 2, figsize=(12, 10))

        y_pred = self.predict(self.X_train)

        # 1. Residuals vs Fitted
        axes[0, 0].scatter(y_pred, self.residuals, alpha=0.6)
        axes[0, 0].axhline(y=0, color='r', linestyle='--')
        axes[0, 0].set_xlabel('Fitted Values')
        axes[0, 0].set_ylabel('Residuals')
        axes[0, 0].set_title('Residuals vs Fitted')

        # 2. QQ plot of residuals
        from scipy.stats import probplot
        probplot(self.residuals, dist="norm", plot=axes[0, 1])
        axes[0, 1].set_title('Normal Q-Q Plot')

        # 3. Scale-Location plot
        standardized_residuals = self.residuals / np.sqrt(self.mse)
        axes[1, 0].scatter(y_pred, np.sqrt(np.abs(standardized_residuals)), alpha=0.6)
        axes[1, 0].set_xlabel('Fitted Values')
        axes[1, 0].set_ylabel('√|Standardized Residuals|')
        axes[1, 0].set_title('Scale-Location Plot')
```

```
        # 4. Residuals vs Leverage
        if self.X_train.shape[1] == 1:  # Only for simple regression
            axes[1, 1].hist(self.residuals, bins=20, alpha=0.7)
            axes[1, 1].set_xlabel('Residuals')
            axes[1, 1].set_ylabel('Frequency')
            axes[1, 1].set_title('Histogram of Residuals')
        else:
            axes[1, 1].text(0.5, 0.5, 'Leverage plot\nrequires additional\ncomputation',
                            transform=axes[1, 1].transAxes, ha='center', va='center')
            axes[1, 1].set_title('Residuals Distribution')

        plt.tight_layout()
        plt.show()

# Example with synthetic data
np.random.seed(42)
X, y = make_regression(n_samples=100, n_features=3, noise=10, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Fit custom linear regression
lr = LinearRegressionFromScratch()
lr.fit(X_train, y_train)

# Display results
lr.summary()

# Make predictions
y_pred_train = lr.predict(X_train)
y_pred_test = lr.predict(X_test)

# Evaluate performance
train_mse = np.mean((y_train - y_pred_train)**2)
test_mse = np.mean((y_test - y_pred_test)**2)
train_r2 = 1 - np.sum((y_train - y_pred_train)**2) / np.sum((y_train - np.mean(y_train))**
test_r2 = 1 - np.sum((y_test - y_pred_test)**2) / np.sum((y_test - np.mean(y_test))**2)

print(f"\nPerformance Metrics:")
print(f"Train MSE: {train_mse:.4f}, Train R²: {train_r2:.4f}")
print(f"Test MSE: {test_mse:.4f}, Test R²: {test_r2:.4f}")

# Plot diagnostics
lr.plot_diagnostics()
```

Regularized Linear Regression

Ridge Regression (L2 Regularization):
Minimize: $||\mathbf{y} - X\boldsymbol{\beta}||^2 + \lambda||\boldsymbol{\beta}||^2$

Closed-form solution:
$\hat{\boldsymbol{\beta}}_{ridge} = (X^TX + \lambda I)^{-1}X^T\mathbf{y}$

Lasso Regression (L1 Regularization):
Minimize: $||\mathbf{y} - X\boldsymbol{\beta}||^2 + \lambda||\boldsymbol{\beta}||_1$

Elastic Net:
Minimize: $||\mathbf{y} - X\boldsymbol{\beta}||^2 + \lambda_1||\boldsymbol{\beta}||_1 + \lambda_2||\boldsymbol{\beta}||^2$

```
class RegularizedRegression:
    """
    Implementation of Ridge, Lasso, and Elastic Net regression
    with cross-validation for hyperparameter selection
    """
    def __init__(self, reg_type='ridge', alpha=1.0, l1_ratio=0.5):
        self.reg_type = reg_type
        self.alpha = alpha
        self.l1_ratio = l1_ratio  # For Elastic Net
        self.coefficients = None
        self.intercept = None

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

    def fit_ridge(self, X, y):
        """Ridge regression using closed-form solution"""
        X_centered = X - np.mean(X, axis=0)
        y_centered = y - np.mean(y)

        # Ridge solution: (X^T X + αI)^(-1) X^T y
        XtX_reg = X_centered.T @ X_centered + self.alpha * np.eye(X.shape[1])
        self.coefficients = np.linalg.solve(XtX_reg, X_centered.T @ y_centered)
        self.intercept = np.mean(y) - np.sum(self.coefficients * np.mean(X, axis=0))

    def fit_lasso(self, X, y, max_iter=1000, tol=1e-6):
        """Lasso regression using coordinate descent"""
        X_centered = X - np.mean(X, axis=0)
        y_centered = y - np.mean(y)

        n, p = X_centered.shape
        self.coefficients = np.zeros(p)

        # Precompute X^T X diagonal
        XtX_diag = np.sum(X_centered**2, axis=0)

        for iteration in range(max_iter):
            coeffs_old = self.coefficients.copy()

            for j in range(p):
                # Partial residual
                residual = y_centered - X_centered @ self.coefficients + self.coefficient

                # Coordinate update
                rho = X_centered[:, j] @ residual
                self.coefficients[j] = self._soft_threshold(rho, self.alpha) / XtX_diag[

            # Check convergence
            if np.linalg.norm(self.coefficients - coeffs_old) &lt; tol:
                break
```

```
        self.intercept = np.mean(y) - np.sum(self.coefficients * np.mean(X, axis=0))

    def fit_elastic_net(self, X, y, max_iter=1000, tol=1e-6):
        """Elastic Net using coordinate descent"""
        X_centered = X - np.mean(X, axis=0)
        y_centered = y - np.mean(y)

        n, p = X_centered.shape
        self.coefficients = np.zeros(p)

        alpha_l1 = self.alpha * self.l1_ratio
        alpha_l2 = self.alpha * (1 - self.l1_ratio)

        XtX_diag = np.sum(X_centered**2, axis=0) + alpha_l2

        for iteration in range(max_iter):
            coeffs_old = self.coefficients.copy()

            for j in range(p):
                residual = y_centered - X_centered @ self.coefficients + self.coefficient
                rho = X_centered[:, j] @ residual
                self.coefficients[j] = self._soft_threshold(rho, alpha_l1) / XtX_diag[j]

            if np.linalg.norm(self.coefficients - coeffs_old) &lt; tol:
                break

        self.intercept = np.mean(y)

    def fit(self, X, y):
        """Fit the specified regression type"""
        if self.reg_type == 'ridge':
            self.fit_ridge(X, y)
        elif self.reg_type == 'lasso':
            self.fit_lasso(X, y)
        elif self.reg_type == 'elastic_net':
            self.fit_elastic_net(X, y)
        else:
            raise ValueError("reg_type must be 'ridge', 'lasso', or 'elastic_net'")

    def predict(self, X):
        """Make predictions"""
        return X @ self.coefficients + self.intercept

# Cross-validation for hyperparameter selection
def cross_validate_regression(X, y, reg_type='ridge', alphas=None, cv_folds=5):
    """
    Cross-validation for regularized regression hyperparameter selection
    """
    if alphas is None:
        alphas = np.logspace(-3, 3, 50)

    n_samples = X.shape[0]
    fold_size = n_samples // cv_folds
    cv_scores = []

    for alpha in alphas:
```

```
        fold_scores = []

        for fold in range(cv_folds):
            # Split data
            start_idx = fold * fold_size
            end_idx = start_idx + fold_size if fold &lt; cv_folds - 1 else n_samples

            test_indices = list(range(start_idx, end_idx))
            train_indices = [i for i in range(n_samples) if i not in test_indices]

            X_train_fold = X[train_indices]
            y_train_fold = y[train_indices]
            X_test_fold = X[test_indices]
            y_test_fold = y[test_indices]

            # Fit model
            model = RegularizedRegression(reg_type=reg_type, alpha=alpha)
            model.fit(X_train_fold, y_train_fold)

            # Evaluate
            y_pred = model.predict(X_test_fold)
            mse = np.mean((y_test_fold - y_pred)**2)
            fold_scores.append(mse)

        cv_scores.append(np.mean(fold_scores))

    # Find best alpha
    best_idx = np.argmin(cv_scores)
    best_alpha = alphas[best_idx]
    best_score = cv_scores[best_idx]

    return best_alpha, best_score, cv_scores

# Example: Compare regularization methods
X, y = make_regression(n_samples=100, n_features=20, noise=0.1, random_state=42)

# Add some irrelevant features to demonstrate feature selection
X_irrelevant = np.random.randn(100, 30)
X_combined = np.column_stack([X, X_irrelevant])

# Cross-validation for each method
methods = ['ridge', 'lasso', 'elastic_net']
results = {}

for method in methods:
    best_alpha, best_score, cv_scores = cross_validate_regression(X_combined, y, method)
    results[method] = {
        'best_alpha': best_alpha,
        'best_score': best_score,
        'cv_scores': cv_scores
    }

    # Fit final model with best alpha
    model = RegularizedRegression(reg_type=method, alpha=best_alpha)
    model.fit(X_combined, y)
```

```
    print(f"\n{method.upper()} Regression:")
    print(f"Best alpha: {best_alpha:.4f}")
    print(f"Best CV score (MSE): {best_score:.4f}")
    print(f"Number of non-zero coefficients: {np.sum(np.abs(model.coefficients) &gt; 1e-6
    print(f"Coefficient norm: L1={np.sum(np.abs(model.coefficients)):.4f}, L2={np.linalg.
```

5.2 Logistic Regression - Complete Mathematical Treatment

Binary Logistic Regression

Logistic Function (Sigmoid):
$p(y=1|\mathbf{x}) = \sigma(\mathbf{x}^T\boldsymbol{\beta}) = 1/(1 + e^{(-\mathbf{x}^T\boldsymbol{\beta})})$

Odds and Log-Odds:
Odds = $p/(1-p)$
Log-Odds (Logit) = $\log(p/(1-p)) = \mathbf{x}^T\boldsymbol{\beta}$

Likelihood Function:
$L(\boldsymbol{\beta}) = \prod_{i=1}^{n} [p_i^{y_i} \times (1-p_i)^{(1-y_i)}]$

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

Gradient:
$\nabla\ell(\boldsymbol{\beta}) = X^T(\mathbf{y} - \mathbf{p})$

Hessian:
$\nabla^2\ell(\boldsymbol{\beta}) = -X^TWX$

where $W = \text{diag}(p_1(1-p_1), \ldots, p_n(1-p_n))$

```
import numpy as np
from scipy.optimize import minimize
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

class LogisticRegressionFromScratch:
    """
    Complete implementation of logistic regression with detailed mathematics
    """
    def __init__(self, fit_intercept=True, regularization=None, C=1.0, max_iter=1000, tol
        self.fit_intercept = fit_intercept
        self.regularization = regularization  # None, 'l1', 'l2', 'elastic_net'
        self.C = C  # Inverse of regularization strength
        self.max_iter = max_iter
        self.tol = tol
        self.coefficients = None
        self.intercept = None
        self.n_iter_ = None

    def _add_intercept(self, X):
```

```
        """Add intercept column"""
        return np.column_stack([np.ones(X.shape[0]), X])

    def _sigmoid(self, z):
        """Sigmoid function with numerical stability"""
        z = np.clip(z, -250, 250)  # Prevent overflow
        return 1 / (1 + np.exp(-z))

    def _log_likelihood(self, beta, X, y):
        """Compute log-likelihood"""
        z = X @ beta
        p = self._sigmoid(z)

        # Add small epsilon to prevent log(0)
        eps = 1e-15
        p = np.clip(p, eps, 1 - eps)

        log_likelihood = np.sum(y * np.log(p) + (1 - y) * np.log(1 - p))

        # Add regularization
        if self.regularization == 'l2':
            penalty = 0.5 / self.C * np.sum(beta[1 if self.fit_intercept else 0:]**2)
            log_likelihood -= penalty
        elif self.regularization == 'l1':
            penalty = 1 / self.C * np.sum(np.abs(beta[1 if self.fit_intercept else 0:]))
            log_likelihood -= penalty

        return -log_likelihood  # Return negative for minimization

    def _gradient(self, beta, X, y):
        """Compute gradient of log-likelihood"""
        z = X @ beta
        p = self._sigmoid(z)
        gradient = X.T @ (p - y)

        # Add regularization
        if self.regularization == 'l2':
            reg_gradient = beta / self.C
            if self.fit_intercept:
                reg_gradient[0] = 0  # Don't regularize intercept
            gradient += reg_gradient
        elif self.regularization == 'l1':
            reg_gradient = np.sign(beta) / self.C
            if self.fit_intercept:
                reg_gradient[0] = 0
            gradient += reg_gradient

        return gradient

    def _hessian(self, beta, X, y):
        """Compute Hessian matrix"""
        z = X @ beta
        p = self._sigmoid(z)
        W = np.diag(p * (1 - p))
        hessian = X.T @ W @ X
```

```
        # Add regularization
        if self.regularization == 'l2':
            reg_hessian = np.eye(len(beta)) / self.C
            if self.fit_intercept:
                reg_hessian[0, 0] = 0
            hessian += reg_hessian

        return hessian

    def _newton_raphson(self, X, y):
        """Newton-Raphson optimization"""
        # Initialize parameters
        n_features = X.shape[1]
        beta = np.zeros(n_features)

        for i in range(self.max_iter):
            gradient = self._gradient(beta, X, y)
            hessian = self._hessian(beta, X, y)

            # Newton-Raphson update
            try:
                delta = np.linalg.solve(hessian, gradient)
                beta_new = beta - delta

                # Check convergence
                if np.linalg.norm(delta) &lt; self.tol:
                    self.n_iter_ = i + 1
                    break

                beta = beta_new

            except np.linalg.LinAlgError:
                # Fall back to gradient descent if Hessian is singular
                learning_rate = 0.01
                beta = beta - learning_rate * gradient

        return beta

    def _irls(self, X, y):
        """Iteratively Reweighted Least Squares"""
        n_samples, n_features = X.shape
        beta = np.zeros(n_features)

        for i in range(self.max_iter):
            z = X @ beta
            p = self._sigmoid(z)

            # Weights matrix
            W = np.diag(p * (1 - p) + 1e-8)  # Add small epsilon for stability

            # Working response
            eta = z + np.linalg.solve(W, y - p)

            # Weighted least squares update
            try:
                XtWX = X.T @ W @ X
```

```
                if self.regularization == 'l2':
                    XtWX += np.eye(n_features) / self.C
                    if self.fit_intercept:
                        XtWX[0, 0] -= 1 / self.C

                XtWeta = X.T @ W @ eta
                beta_new = np.linalg.solve(XtWX, XtWeta)

                # Check convergence
                if np.linalg.norm(beta_new - beta) &lt; self.tol:
                    self.n_iter_ = i + 1
                    break

                beta = beta_new

            except np.linalg.LinAlgError:
                print("IRLS failed, using gradient descent")
                gradient = self._gradient(beta, X, y)
                beta = beta - 0.01 * gradient

        return beta

    def fit(self, X, y, method='newton_raphson'):
        """
        Fit logistic regression

        Methods:
        - 'newton_raphson': Newton-Raphson method
        - 'irls': Iteratively Reweighted Least Squares
        - 'scipy': Use scipy optimization
        """
        # Prepare data
        if self.fit_intercept:
            X_with_intercept = self._add_intercept(X)
        else:
            X_with_intercept = X.copy()

        # Fit model using specified method
        if method == 'newton_raphson':
            beta = self._newton_raphson(X_with_intercept, y)
        elif method == 'irls':
            beta = self._irls(X_with_intercept, y)
        elif method == 'scipy':
            # Use scipy's optimization
            beta_init = np.zeros(X_with_intercept.shape[1])
            result = minimize(
                fun=self._log_likelihood,
                x0=beta_init,
                args=(X_with_intercept, y),
                jac=self._gradient,
                method='BFGS',
                options={'maxiter': self.max_iter}
            )
            beta = result.x
            self.n_iter_ = result.nit
```

```
        # Store parameters
        if self.fit_intercept:
            self.intercept = beta[0]
            self.coefficients = beta[1:]
        else:
            self.intercept = 0
            self.coefficients = beta

    def predict_proba(self, X):
        """Predict class probabilities"""
        z = X @ self.coefficients + self.intercept
        p = self._sigmoid(z)
        return np.column_stack([1 - p, p])

    def predict(self, X):
        """Predict binary classes"""
        return (self.predict_proba(X)[:, 1] &gt;= 0.5).astype(int)

    def score(self, X, y):
        """Compute accuracy score"""
        y_pred = self.predict(X)
        return np.mean(y_pred == y)

    def summary(self, X, y, feature_names=None):
        """Print model summary with statistical tests"""
        print("Logistic Regression Summary")
        print("=" * 50)

        # Compute statistics
        if self.fit_intercept:
            X_with_intercept = self._add_intercept(X)
            all_params = np.concatenate([[self.intercept], self.coefficients])
            param_names = ['Intercept'] + (feature_names if feature_names else [f'X{i}' f
        else:
            X_with_intercept = X
            all_params = self.coefficients
            param_names = feature_names if feature_names else [f'X{i}' for i in range(len

        # Standard errors from Hessian
        try:
            hessian = self._hessian(all_params, X_with_intercept, y)
            covariance_matrix = np.linalg.inv(hessian)
            standard_errors = np.sqrt(np.diag(covariance_matrix))

            # Wald statistics
            z_scores = all_params / standard_errors

            # p-values (two-tailed)
            from scipy.stats import norm
            p_values = 2 * (1 - norm.cdf(np.abs(z_scores)))

            print(f"{'Variable':&lt;12} {'Coefficient':&lt;12} {'Std Error':&lt;12} {'z-s
            print("-" * 85)

            for i, (name, coef, se, z, p) in enumerate(zip(param_names, all_params, stand
                ci_lower = coef - 1.96 * se
```

```
                ci_upper = coef + 1.96 * se
                ci_str = f"[{ci_lower:.3f}, {ci_upper:.3f}]"
                print(f"{name:&lt;12} {coef:&lt;12.4f} {se:&lt;12.4f} {z:&lt;10.4f} {p:&]

        except np.linalg.LinAlgError:
            print("Could not compute standard errors (singular Hessian)")
            for name, coef in zip(param_names, all_params):
                print(f"{name:&lt;12} {coef:.4f}")

        # Model fit statistics
        y_pred_proba = self.predict_proba(X)[:, 1]
        log_likelihood = -self._log_likelihood(all_params, X_with_intercept, y)

        # Null model (intercept only)
        p_null = np.mean(y)
        null_log_likelihood = np.sum(y * np.log(p_null + 1e-15) + (1 - y) * np.log(1 - p_

        # Pseudo R-squared measures
        mcfadden_r2 = 1 - log_likelihood / null_log_likelihood

        print(f"\nModel Fit Statistics:")
        print(f"Log-Likelihood: {log_likelihood:.4f}")
        print(f"Null Log-Likelihood: {null_log_likelihood:.4f}")
        print(f"McFadden's R²: {mcfadden_r2:.4f}")
        print(f"Number of iterations: {self.n_iter_}")

# Example usage with detailed analysis
print("Generating classification dataset...")
X, y = make_classification(n_samples=1000, n_features=5, n_informative=3,
                           n_redundant=1, n_clusters_per_class=1, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Compare different optimization methods
methods = ['newton_raphson', 'irls', 'scipy']
feature_names = [f'Feature_{i+1}' for i in range(X.shape[1])]

for method in methods:
    print(f"\n{'='*60}")
    print(f"METHOD: {method.upper()}")
    print('='*60)

    # Fit model
    lr = LogisticRegressionFromScratch(regularization='l2', C=1.0)
    lr.fit(X_train, y_train, method=method)

    # Display summary
    lr.summary(X_train, y_train, feature_names)

    # Evaluate performance
    train_accuracy = lr.score(X_train, y_train)
    test_accuracy = lr.score(X_test, y_test)

    print(f"\nPerformance:")
    print(f"Training Accuracy: {train_accuracy:.4f}")
    print(f"Test Accuracy: {test_accuracy:.4f}")
```

Multinomial Logistic Regression

For K classes, multinomial logistic regression uses:

Softmax Function:
$P(y=k|\mathbf{x}) = \exp(\mathbf{x}^T\boldsymbol{\beta}_k) / \Sigma_{j=1}^{K} \exp(\mathbf{x}^T\boldsymbol{\beta}_j)$

Log-Likelihood:
$\ell(\boldsymbol{\beta}) = \Sigma_i \Sigma_k \, I(y_i=k) \log P(y_i=k|\mathbf{x}_i)$

```
class MultinomialLogisticRegression:
    """
    Multinomial (multi-class) logistic regression implementation
    """
    def __init__(self, max_iter=1000, learning_rate=0.01, regularization=None, C=1.0):
        self.max_iter = max_iter
        self.learning_rate = learning_rate
        self.regularization = regularization
        self.C = C
        self.weights = None
        self.classes_ = None

    def _softmax(self, z):
        """Softmax function with numerical stability"""
        exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
        return exp_z / np.sum(exp_z, axis=1, keepdims=True)

    def _one_hot_encode(self, y):
        """Convert labels to one-hot encoding"""
        n_samples = len(y)
        n_classes = len(self.classes_)
        y_encoded = np.zeros((n_samples, n_classes))

        for i, class_label in enumerate(y):
            class_idx = np.where(self.classes_ == class_label)[0][0]
            y_encoded[i, class_idx] = 1

        return y_encoded

    def fit(self, X, y):
        """Fit multinomial logistic regression"""
        n_samples, n_features = X.shape
        self.classes_ = np.unique(y)
        n_classes = len(self.classes_)

        # Add intercept term
        X_with_intercept = np.column_stack([np.ones(n_samples), X])

        # Initialize weights
        self.weights = np.random.normal(0, 0.01, (X_with_intercept.shape[1], n_classes))

        # One-hot encode labels
        y_encoded = self._one_hot_encode(y)

        # Gradient descent
```

```
        for iteration in range(self.max_iter):
            # Forward pass
            z = X_with_intercept @ self.weights
            predictions = self._softmax(z)

            # Compute gradients
            gradient = X_with_intercept.T @ (predictions - y_encoded) / n_samples

            # Add regularization
            if self.regularization == 'l2':
                reg_gradient = self.weights / self.C
                reg_gradient[0, :] = 0  # Don't regularize intercept
                gradient += reg_gradient

            # Update weights
            self.weights -= self.learning_rate * gradient

    def predict_proba(self, X):
        """Predict class probabilities"""
        n_samples = X.shape[0]
        X_with_intercept = np.column_stack([np.ones(n_samples), X])
        z = X_with_intercept @ self.weights
        return self._softmax(z)

    def predict(self, X):
        """Predict classes"""
        probabilities = self.predict_proba(X)
        class_indices = np.argmax(probabilities, axis=1)
        return self.classes_[class_indices]

    def score(self, X, y):
        """Compute accuracy"""
        y_pred = self.predict(X)
        return np.mean(y_pred == y)

# Example with multi-class dataset
from sklearn.datasets import make_classification

X_multi, y_multi = make_classification(n_samples=1000, n_features=4, n_informative=3,
                                       n_classes=3, n_clusters_per_class=1, random_state=4

X_train_multi, X_test_multi, y_train_multi, y_test_multi = train_test_split(
    X_multi, y_multi, test_size=0.2, random_state=42
)

# Fit multinomial logistic regression
mlr = MultinomialLogisticRegression(max_iter=2000, learning_rate=0.1, regularization='l2'
mlr.fit(X_train_multi, y_train_multi)

# Evaluate
train_acc_multi = mlr.score(X_train_multi, y_train_multi)
test_acc_multi = mlr.score(X_test_multi, y_test_multi)

print(f"\nMultinomial Logistic Regression Results:")
print(f"Number of classes: {len(mlr.classes_)}")
print(f"Classes: {mlr.classes_}")
```

```
print(f"Training Accuracy: {train_acc_multi:.4f}")
print(f"Test Accuracy: {test_acc_multi:.4f}")

# Show class probabilities for first 5 test samples
probabilities = mlr.predict_proba(X_test_multi[:5])
print(f"\nSample Predictions (First 5 test samples):")
print(f"{'Sample':&lt;8} {'True':&lt;6} {'Pred':&lt;6} {'Prob Class 0':&lt;12} {'Prob Cla
print("-" * 70)
predictions = mlr.predict(X_test_multi[:5])
for i in range(5):
    true_label = y_test_multi[i]
    pred_label = predictions[i]
    probs = probabilities[i]
    print(f"{i+1:&lt;8} {true_label:&lt;6} {pred_label:&lt;6} {probs[0]:&lt;12.4f} {probs
```

5.3 Case Study: Predicting Customer Churn

```
# Generate realistic customer churn dataset
np.random.seed(42)

def generate_churn_dataset(n_customers=5000):
    """
    Generate realistic customer churn dataset
    """
    # Customer features
    tenure = np.random.exponential(24, n_customers)  # months
    monthly_charges = np.random.normal(65, 20, n_customers)
    total_charges = tenure * monthly_charges + np.random.normal(0, 100, n_customers)

    # Contract type (0: month-to-month, 1: one year, 2: two year)
    contract = np.random.choice([0, 1, 2], n_customers, p=[0.5, 0.3, 0.2])

    # Services
    internet_service = np.random.choice([0, 1], n_customers, p=[0.2, 0.8])
    streaming_tv = np.random.choice([0, 1], n_customers, p=[0.4, 0.6])
    tech_support = np.random.choice([0, 1], n_customers, p=[0.6, 0.4])

    # Customer satisfaction (1-5 scale)
    satisfaction = np.random.choice([1, 2, 3, 4, 5], n_customers, p=[0.1, 0.15, 0.3, 0.3,

    # Create churn probability based on features
    churn_prob = (
        0.1 +  # Base probability
        0.3 * (tenure &lt; 6) / 100 +  # New customers more likely to churn
        0.2 * (monthly_charges &gt; 80) / 100 +  # High charges increase churn
        0.15 * (contract == 0) +  # Month-to-month contracts
        0.1 * (satisfaction &lt;= 2) +  # Low satisfaction
        -0.05 * tech_support  # Tech support reduces churn
    )

    # Generate churn labels
    churn = np.random.binomial(1, np.clip(churn_prob, 0, 1), n_customers)

    # Combine features
    features = np.column_stack([
```

```
        tenure, monthly_charges, total_charges, contract,
        internet_service, streaming_tv, tech_support, satisfaction
    ])

    feature_names = [
        'tenure', 'monthly_charges', 'total_charges', 'contract_type',
        'internet_service', 'streaming_tv', 'tech_support', 'satisfaction'
    ]

    return features, churn, feature_names

# Generate dataset
X_churn, y_churn, feature_names_churn = generate_churn_dataset(5000)

# Split data
X_train_churn, X_test_churn, y_train_churn, y_test_churn = train_test_split(
    X_churn, y_churn, test_size=0.2, stratify=y_churn, random_state=42
)

print("Customer Churn Prediction Case Study")
print("=" * 50)
print(f"Dataset shape: {X_churn.shape}")
print(f"Churn rate: {np.mean(y_churn):.2%}")
print(f"Features: {feature_names_churn}")

# Standardize features
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_churn)
X_test_scaled = scaler.transform(X_test_churn)

# Fit logistic regression
lr_churn = LogisticRegressionFromScratch(regularization='l2', C=1.0)
lr_churn.fit(X_train_scaled, y_train_churn, method='scipy')

# Detailed analysis
print(f"\nModel Summary:")
lr_churn.summary(X_train_scaled, y_train_churn, feature_names_churn)

# Performance evaluation
from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score, roc_c

y_pred_churn = lr_churn.predict(X_test_scaled)
y_pred_proba_churn = lr_churn.predict_proba(X_test_scaled)[:, 1]

print(f"\nPerformance Metrics:")
print(f"Accuracy: {lr_churn.score(X_test_scaled, y_test_churn):.4f}")
print(f"AUC-ROC: {roc_auc_score(y_test_churn, y_pred_proba_churn):.4f}")

print(f"\nConfusion Matrix:")
cm = confusion_matrix(y_test_churn, y_pred_churn)
print(cm)

print(f"\nClassification Report:")
print(classification_report(y_test_churn, y_pred_churn, target_names=['No Churn', 'Churn'
```

```
# Feature importance (coefficient magnitudes)
print(f"\nFeature Importance (Coefficient Magnitudes):")
feature_importance = np.abs(lr_churn.coefficients)
sorted_idx = np.argsort(feature_importance)[::-1]

for i in sorted_idx:
    direction = "increases" if lr_churn.coefficients[i] &gt; 0 else "decreases"
    print(f"{feature_names_churn[i]:&lt;20}: {feature_importance[i]:.4f} ({direction} chu

# Business insights
print(f"\nBusiness Insights:")
print(f"1. Contract type has major impact - month-to-month customers much more likely to
print(f"2. Customer satisfaction is crucial - low scores strongly predict churn")
print(f"3. Tech support reduces churn probability")
print(f"4. New customers (low tenure) are at higher risk")
print(f"5. High monthly charges increase churn risk")
```

This comprehensive implementation demonstrates:

1. **Mathematical rigor** - Complete derivations and formulations
2. **Multiple algorithms** - Various optimization approaches
3. **Statistical analysis** - Standard errors, confidence intervals, hypothesis tests
4. **Practical implementation** - Real-world case study with business insights
5. **Performance evaluation** - Multiple metrics and diagnostic tools
6. **Code explanations** - Detailed comments and mathematical foundations

The implementations show both the theoretical foundations and practical applications of linear and logistic regression with extensive statistical analysis and real-world case studies.