Week 3: Classification and Logistic Regression

Table of Contents

---

1. Introduction to Classification {#introduction}

What is Classification?

Classification is a supervised learning task where the goal is to predict discrete categories or classes rather than continuous values.

Types of Classification

Binary Classification

- **Output**: Two possible classes (0 or 1, Yes or No, True or False)
- **Examples**:
  - Email spam detection (spam/not spam)
  - Medical diagnosis (disease/no disease)
  - Image recognition (cat/dog)

Multi-class Classification

- **Output**: More than two classes
- **Examples**:
  - Handwritten digit recognition (0-9)
  - Weather prediction (sunny/cloudy/rainy)
  - Sentiment analysis (positive/negative/neutral)

Why Not Linear Regression for Classification?

Linear regression problems with classification:

1. **Unbounded outputs**: Linear regression can output any value, but we need probabilities (0-1)
2. **Outlier sensitivity**: Extreme values can skew the decision boundary
3. **No probabilistic interpretation**: Cannot interpret outputs as probabilities

Motivation Example

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

# Email classification example
# Feature: Number of spam keywords
# Target: 0 = Not spam, 1 = Spam

# Why linear regression fails
X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).reshape(-1, 1)
y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

# Linear regression would give values outside [0,1]
# We need a function that maps to [0,1] range
```

---

2. Logistic Regression {#logistic-regression}

Sigmoid Function

The sigmoid function maps any real number to a value between 0 and 1, making it perfect for binary classification.

Mathematical Definition

```
g(z) = 1 / (1 + e^(-z))
```

Where z can be any real number.

Properties of Sigmoid

- **Range**: (0, 1) - never exactly 0 or 1

- **Shape**: S-shaped curve
- **Midpoint**: g(0) = 0.5
- **Asymptotes**: Approaches 0 as z → -∞, approaches 1 as z → +∞

Logistic Regression Model

Combines linear regression with sigmoid function:

```
f(x) = g(w⃗ · x⃗ + b) = 1 / (1 + e^(-(w⃗ · x⃗ + b)))
```

Where:

- $\vec{w}$ = weight vector
- $\vec{x}$ = feature vector
- b = bias term
- $g(z)$ = sigmoid function

Interpretation

- **Output**: Probability that y = 1 given input x
- **P(y=1|x)**: Probability of positive class
- **P(y=0|x)**: 1 - P(y=1|x)

Implementation Example

```python
```

```python
def sigmoid(z):
    """
    Sigmoid activation function
    """
    # Clip z to prevent overflow
    z = np.clip(z, -500, 500)
    return 1 / (1 + np.exp(-z))

def logistic_regression_predict(X, w, b):
    """
    Logistic regression prediction
    """
    z = np.dot(X, w) + b
    return sigmoid(z)

# Example usage
X = np.array([[1, 2], [2, 3], [3, 4]])
w = np.array([0.5, -0.3])
b = 0.1

probabilities = logistic_regression_predict(X, w, b)
print(f"Probabilities: {probabilities}")

# Convert to binary predictions
predictions = (probabilities >= 0.5).astype(int)
print(f"Binary predictions: {predictions}")
```

---

3. Decision Boundary {#decision-boundary}

Definition

The decision boundary is the line (or surface) that separates different classes in the feature space.

Mathematical Derivation

For binary classification, the decision boundary occurs where:

$$P(y=1|x) = 0.5$$

This happens when:

$$g(w\vec{} \cdot x\vec{} + b) = 0.5$$

Since g(0) = 0.5, the decision boundary is:

$$w\vec{} \cdot x\vec{} + b = 0$$

Linear Decision Boundary

For 2D case with features x_1 and x_2:

$$w_1x_1 + w_2x_2 + b = 0$$

This is a straight line with:

- **Slope**: $-w_1/w_2$
- **y-intercept**: $-b/w_2$

Non-linear Decision Boundaries

Can be created using polynomial features:

```python
```

```python
# Polynomial features for non-linear boundary
def create_polynomial_features(X1, X2, degree=2):
    """
    Create polynomial features for non-linear decision boundary
    """
    features = []
    for i in range(degree + 1):
        for j in range(degree + 1 - i):
            if i + j <= degree and i + j > 0:
                features.append((X1 ** i) * (X2 ** j))

    return np.column_stack(features)

# Example: Circular decision boundary
def circular_decision_boundary():
    # Generate circular data
    theta = np.linspace(0, 2*np.pi, 100)
    X1 = np.cos(theta)
    X2 = np.sin(theta)

    # Create polynomial features
    X_poly = create_polynomial_features(X1, X2, degree=2)

    # Weights for circular boundary: x₁² + x₂² - 1 = 0
    w = np.array([1, 1, 0])  # [x₁², x₂², x₁x₂]
    b = -1

    return X_poly, w, b

# Visualize decision boundary
def plot_decision_boundary(X, y, w, b, title="Decision Boundary"):
    plt.figure(figsize=(10, 8))

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

    # Create mesh for decision boundary
    h = 0.02
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
```

```
    # Calculate decision boundary
    mesh_points = np.c_[xx.ravel(), yy.ravel()]
    Z = logistic_regression_predict(mesh_points, w, b)
    Z = Z.reshape(xx.shape)

    # Plot decision boundary
    plt.contour(xx, yy, Z, levels=[0.5], colors='black', linestyles='--', linewidths=2)

    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()
```

---

4. Cost Function for Logistic Regression {#cost-function}

Why Not Squared Error?

Squared error cost function with logistic regression creates a non-convex optimization problem with many local minima, making gradient descent ineffective.

Logistic Loss Function

Uses logarithmic loss (cross-entropy) which is convex:

For Single Training Example

$$\text{Loss}(f(x^{(i)}), y^{(i)}) = -y^{(i)} \log(f(x^{(i)})) - (1-y^{(i)}) \log(1-f(x^{(i)}))$$

Intuition

- **When y = 1**: Loss = -log(f(x))
  - If f(x) → 1: Loss → 0 (good)
  - If f(x) → 0: Loss → ∞ (bad)
- **When y = 0**: Loss = -log(1-f(x))
  - If f(x) → 0: Loss → 0 (good)
  - If f(x) → 1: Loss → ∞ (bad)

Overall Cost Function

$$J(w□,b) = (1/m) * Σ[Loss(f(x^{(i)}), y^{(i)})]$$

Simplified form:

$$J(w□,b) = -(1/m) * Σ[y^{(i)} \log(f(x^{(i)})) + (1-y^{(i)}) \log(1-f(x^{(i)}))]$$

Implementation Example

```python
def logistic_cost_function(X, y, w, b):
    """
    Compute logistic regression cost function
    """
    m = len(y)

    # Forward pass
    z = np.dot(X, w) + b
    f = sigmoid(z)

    # Prevent log(0) by adding small epsilon
    epsilon = 1e-15
    f = np.clip(f, epsilon, 1 - epsilon)

    # Compute cost
    cost = -(1/m) * np.sum(y * np.log(f) + (1 - y) * np.log(1 - f))

    return cost

# Example usage
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y = np.array([0, 0, 1, 1])
w = np.array([0.5, -0.3])
b = 0.1

cost = logistic_cost_function(X, y, w, b)
print(f"Cost: {cost}")
```

---

5. Gradient Descent Implementation {#gradient-descent}

Gradient Computation

The partial derivatives for logistic regression are:

$$\partial J(w\square,b)/\partial w_j \equiv (1/m) * \Sigma(f(x^{(i)}) - y^{(i)}) * x_j^{(i)}$$
$$\partial J(w\square,b)/\partial b \equiv (1/m) * \Sigma(f(x^{(i)}) - y^{(i)})$$

Note: Same form as linear regression, but f(x) is now the sigmoid function!

Update Rules

$$w_j = w_j - \alpha * \partial J(w\square,b)/\partial w_j$$
$$b = b - \alpha * \partial J(w\square,b)/\partial b$$

Vectorized Implementation

python

```python
def logistic_gradient_descent(X, y, w_init, b_init, alpha, iterations):
    """
    Logistic regression gradient descent
    """
    m, n = X.shape
    w = w_init.copy()
    b = b_init
    cost_history = []

    for i in range(iterations):
        # Forward pass
        z = np.dot(X, w) + b
        f = sigmoid(z)

        # Compute cost
        cost = logistic_cost_function(X, y, w, b)
        cost_history.append(cost)

        # Compute gradients
        dw = (1/m) * np.dot(X.T, (f - y))
        db = (1/m) * np.sum(f - y)

        # Update parameters
        w = w - alpha * dw
        b = b - alpha * db

        # Print progress
        if i % 100 == 0:
            print(f"Iteration {i}: Cost = {cost:.6f}")

    return w, b, cost_history

# Example usage
np.random.seed(42)
X = np.random.randn(100, 2)
y = (X[:, 0] + X[:, 1] > 0).astype(int)

w_init = np.zeros(2)
b_init = 0
alpha = 0.1
iterations = 1000
```

```
w, b, costs = logistic_gradient_descent(X, y, w_init, b_init, alpha, iterations)
print(f"Final parameters: w = {w}, b = {b}")
```

---

6. The Problem of Overfitting {#overfitting}

Definition

Overfitting occurs when a model learns the training data too well, including noise and irrelevant patterns, leading to poor generalization on new data.

Symptoms of Overfitting

1. **High training accuracy, low test accuracy**
2. **Complex decision boundaries** that follow training data exactly
3. **High variance** in predictions
4. **Poor performance** on unseen data

Causes of Overfitting

1. **Too many features** relative to training examples
2. **Complex model** (high-degree polynomials)
3. **Insufficient training data**
4. **Noise in training data**

Underfitting vs. Overfitting vs. Just Right

Underfitting (High Bias)

- **Training error**: High
- **Test error**: High
- **Cause**: Model too simple
- **Solution**: Increase model complexity

Overfitting (High Variance)

- **Training error**: Low
- **Test error**: High
- **Cause**: Model too complex
- **Solution**: Reduce model complexity, regularization

Just Right

- **Training error**: Low
- **Test error**: Low
- **Generalization**: Good

Overfitting Example

python

```
# Generate polynomial overfitting example
def generate_overfitting_example():
    np.random.seed(42)

    # Generate training data
    X_train = np.linspace(-3, 3, 20).reshape(-1, 1)
    y_train = (X_train.ravel() > 0).astype(int) + 0.1 * np.random.randn(20)

    # Generate test data
    X_test = np.linspace(-3, 3, 100).reshape(-1, 1)
    y_test = (X_test.ravel() > 0).astype(int)

    return X_train, y_train, X_test, y_test

# Fit polynomial logistic regression
def fit_polynomial_logistic(X, y, degree):
    # Create polynomial features
    X_poly = np.column_stack([X**i for i in range(1, degree+1)])

    # Fit logistic regression
    w = np.random.randn(degree) * 0.01
    b = 0

    # Simple gradient descent
    alpha = 0.01
    for _ in range(1000):
        z = np.dot(X_poly, w) + b
        f = sigmoid(z)

        dw = np.dot(X_poly.T, (f - y)) / len(y)
        db = np.sum(f - y) / len(y)

        w -= alpha * dw
        b -= alpha * db

    return w, b

# Compare different polynomial degrees
degrees = [1, 5, 15]
X_train, y_train, X_test, y_test = generate_overfitting_example()

for degree in degrees:
    w, b = fit_polynomial_logistic(X_train, y_train, degree)
```

```
    # Evaluate on training and test sets
    X_train_poly = np.column_stack([X_train**i for i in range(1, degree+1)])
    X_test_poly = np.column_stack([X_test**i for i in range(1, degree+1)])

    train_pred = sigmoid(np.dot(X_train_poly, w) + b)
    test_pred = sigmoid(np.dot(X_test_poly, w) + b)

    print(f"Degree {degree}:")
    print(f"  Training accuracy: {np.mean((train_pred > 0.5) == y_train):.3f}")
    print(f"  Test accuracy: {np.mean((test_pred > 0.5) == y_test):.3f}")
```

---

7. Regularization {#regularization}

What is Regularization?

Regularization is a technique to prevent overfitting by adding a penalty term to the cost function that discourages large parameter values.

Types of Regularization

L2 Regularization (Ridge)

Adds squared magnitude of parameters:

$$J_{regularized}(w□,b) = J(w□,b) + (λ/2m) * Σw_j^2$$

L1 Regularization (Lasso)

Adds absolute magnitude of parameters:

$$J_{regularized}(w□,b) = J(w□,b) + (λ/m) * Σ|w_j|$$

Regularization Parameter (λ)

- **λ = 0**: No regularization (may overfit)
- **λ very large**: High regularization (may underfit)
- **λ optimal**: Balances fitting and regularization

Regularized Cost Functions

Regularized Linear Regression

$$J(w□,b) = (1/2m) * Σ(f(x^{(i)}) - y^{(i)})^2 + (λ/2m) * Σw_j^2$$

Regularized Logistic Regression

$$J(w□,b) = -(1/m) * Σ[y^{(i)} log(f(x^{(i)})) + (1-y^{(i)}) log(1-f(x^{(i)}))] + (λ/2m) * Σw_j^2$$

Note: Typically don't regularize the bias term b.

Regularized Gradient Descent

Linear Regression

$$w_j = w_j - α * [(1/m) * Σ(f(x^{(i)}) - y^{(i)}) * x_j^{(i)} + (λ/m) * w_j]$$
$$b = b - α * [(1/m) * Σ(f(x^{(i)}) - y^{(i)})]$$

Logistic Regression

$$w_j = w_j - α * [(1/m) * Σ(f(x^{(i)}) - y^{(i)}) * x_j^{(i)} + (λ/m) * w_j]$$
$$b = b - α * [(1/m) * Σ(f(x^{(i)}) - y^{(i)})]$$

Implementation Example

python

```python
def regularized_logistic_regression(X, y, lambda_reg, alpha=0.01, iterations=1000):
    """
    Regularized logistic regression with L2 regularization
    """
    m, n = X.shape
    w = np.zeros(n)
    b = 0
    cost_history = []

    for i in range(iterations):
        # Forward pass
        z = np.dot(X, w) + b
        f = sigmoid(z)

        # Compute regularized cost
        base_cost = -(1/m) * np.sum(y * np.log(f + 1e-15) +
                                    (1 - y) * np.log(1 - f + 1e-15))
        reg_cost = (lambda_reg / (2 * m)) * np.sum(w ** 2)
        total_cost = base_cost + reg_cost
        cost_history.append(total_cost)

        # Compute regularized gradients
        dw = (1/m) * np.dot(X.T, (f - y)) + (lambda_reg / m) * w
        db = (1/m) * np.sum(f - y)

        # Update parameters
        w = w - alpha * dw
        b = b - alpha * db

        if i % 100 == 0:
            print(f"Iteration {i}: Cost = {total_cost:.6f}")

    return w, b, cost_history

# Example: Compare regularized vs non-regularized
np.random.seed(42)
X = np.random.randn(50, 10)  # 50 samples, 10 features
y = (np.sum(X[:, :3], axis=1) > 0).astype(int)  # Only first 3 features matter

# No regularization
w_no_reg, b_no_reg, _ = regularized_logistic_regression(X, y, lambda_reg=0)

# With regularization
```

```
w_reg, b_reg, _ = regularized_logistic_regression(X, y, lambda_reg=1.0)

print(f"No regularization - weights: {w_no_reg}")
print(f"With regularization - weights: {w_reg}")
```

---

8. Implementation Examples {#implementation}

Complete Logistic Regression Class

python

```
class LogisticRegression:
    def __init__(self, learning_rate=0.01, max_iterations=1000,
                 regularization=None, lambda_reg=0.01):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.regularization = regularization
        self.lambda_reg = lambda_reg
        self.w = None
        self.b = None
        self.cost_history = []

    def _sigmoid(self, z):
        z = np.clip(z, -500, 500)
        return 1 / (1 + np.exp(-z))

    def _compute_cost(self, X, y):
        m = len(y)
        z = np.dot(X, self.w) + self.b
        f = self._sigmoid(z)

        # Prevent log(0)
        f = np.clip(f, 1e-15, 1 - 1e-15)

        # Base cost
        base_cost = -(1/m) * np.sum(y * np.log(f) + (1 - y) * np.log(1 - f))

        # Regularization
        if self.regularization == 'l2':
            reg_cost = (self.lambda_reg / (2 * m)) * np.sum(self.w ** 2)
        elif self.regularization == 'l1':
            reg_cost = (self.lambda_reg / m) * np.sum(np.abs(self.w))
        else:
            reg_cost = 0

        return base_cost + reg_cost

    def fit(self, X, y):
        m, n = X.shape
        self.w = np.zeros(n)
        self.b = 0

        for i in range(self.max_iterations):
            # Forward pass
```

```
            z = np.dot(X, self.w) + self.b
            f = self._sigmoid(z)

            # Compute cost
            cost = self._compute_cost(X, y)
            self.cost_history.append(cost)

            # Compute gradients
            dw = (1/m) * np.dot(X.T, (f - y))
            db = (1/m) * np.sum(f - y)

            # Add regularization to weight gradient
            if self.regularization == 'l2':
                dw += (self.lambda_reg / m) * self.w
            elif self.regularization == 'l1':
                dw += (self.lambda_reg / m) * np.sign(self.w)

            # Update parameters
            self.w -= self.learning_rate * dw
            self.b -= self.learning_rate * db

    def predict_proba(self, X):
        z = np.dot(X, self.w) + self.b
        return self._sigmoid(z)

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

    def score(self, X, y):
        predictions = self.predict(X)
        return np.mean(predictions == y)

# Example usage
np.random.seed(42)
X = np.random.randn(100, 2)
y = (X[:, 0] + X[:, 1] > 0).astype(int)

# Train models
model_no_reg = LogisticRegression(regularization=None)
model_l2 = LogisticRegression(regularization='l2', lambda_reg=0.1)

model_no_reg.fit(X, y)
model_l2.fit(X, y)
```

```
print(f"No regularization accuracy: {model_no_reg.score(X, y):.3f}")
print(f"L2 regularization accuracy: {model_l2.score(X, y):.3f}")
```

Multi-class Logistic Regression (One-vs-All)

python

```
class MultiClassLogisticRegression:
    def __init__(self, learning_rate=0.01, max_iterations=1000):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.classifiers = {}
        self.classes = None

    def fit(self, X, y):
        self.classes = np.unique(y)

        for class_label in self.classes:
            # Create binary labels (one-vs-all)
            binary_y = (y == class_label).astype(int)

            # Train binary classifier
            classifier = LogisticRegression(
                learning_rate=self.learning_rate,
                max_iterations=self.max_iterations
            )
            classifier.fit(X, binary_y)

            self.classifiers[class_label] = classifier

    def predict_proba(self, X):
        probabilities = np.zeros((len(X), len(self.classes)))

        for i, class_label in enumerate(self.classes):
            probabilities[:, i] = self.classifiers[class_label].predict_proba(X)

        return probabilities

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

# Example: Multi-class classification
np.random.seed(42)
X = np.random.randn(150, 2)
y = np.array([0] * 50 + [1] * 50 + [2] * 50)

model = MultiClassLogisticRegression()
model.fit(X, y)
```

```
predictions = model.predict(X)
accuracy = np.mean(predictions == y)
print(f"Multi-class accuracy: {accuracy:.3f}")
```

Key Takeaways

1. **Classification**: Predicts discrete categories using logistic regression with sigmoid function
2. **Sigmoid Function**: Maps any real number to (0,1) range, perfect for probability interpretation
3. **Decision Boundary**: Separates classes in feature space; linear for basic logistic regression
4. **Logistic Cost Function**: Uses cross-entropy loss, which is convex and suitable for gradient descent
5. **Overfitting**: Major problem when model is too complex; symptoms include high training accuracy but low test accuracy
6. **Regularization**: Prevents overfitting by adding penalty terms to cost function (L1 or L2)
7. **Lambda Parameter**: Controls regularization strength; requires tuning for optimal performance

---

Next Steps

- Week 4: Advanced classification techniques
- Decision trees and ensemble methods
- Model evaluation metrics (precision, recall, F1-score)
- Cross-validation techniques
- Feature selection methods