Week 2: Multiple Linear Regression and Feature Engineering

Table of Contents

---

1. Multiple Linear Regression {#multiple-regression}

Overview

Multiple linear regression extends simple linear regression to handle multiple input features simultaneously, enabling more complex and realistic models.

Mathematical Notation

- $x_1, x_2, ..., x_n$ = input features
- $w_1, w_2, ..., w_n$ = weights/parameters for each feature
- b = bias term
- n = number of features
- m = number of training examples

Multiple Linear Regression Model

$$f(x) = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

Vector Notation:

$$f(x) = w⃗ \cdot x⃗ + b$$

Where:

- $w\vec{} = [w_1, w_2, ..., w_n]$ (weight vector)
- $x\vec{} = [x_1, x_2, ..., x_n]$ (feature vector)

Example: House Price Prediction

Features might include:

- Size (square feet)
- Number of bedrooms
- Number of bathrooms
- Age of house
- Location score

Implementation Example

```python
import numpy as np

# Multiple features example
def multiple_linear_regression(X, w, b):
    """
    X: (m, n) matrix of features
    w: (n,) weight vector
    b: scalar bias
    """
    return np.dot(X, w) + b

# Example usage
X = np.array([[2104, 5, 1, 45],   # house 1
              [1416, 3, 2, 40],   # house 2
              [1534, 3, 2, 30]])  # house 3

w = np.array([0.39, 18.75, -53.36, -26.42])
b = 785.18

predictions = multiple_linear_regression(X, w, b)
print(f"Predictions: {predictions}")
```

---

2. Vectorization {#vectorization}

What is Vectorization?

Vectorization is the process of replacing explicit loops with array operations, making code more efficient and readable.

Benefits of Vectorization

1. **Performance**: Faster execution using optimized libraries
2. **Readability**: Cleaner, more concise code
3. **Scalability**: Better handling of large datasets
4. **Parallelization**: Automatic use of multiple CPU cores

Vectorization Examples

Without Vectorization (Loops)

```python
# Dot product using loops
def dot_product_loop(w, x):
    result = 0
    for i in range(len(w)):
        result += w[i] * x[i]
    return result

# Example
w = np.array([1, 2, 3])
x = np.array([4, 5, 6])
result = dot_product_loop(w, x)  # Slow
```

With Vectorization

```python
# Vectorized dot product
def dot_product_vectorized(w, x):
    return np.dot(w, x)

# Example
w = np.array([1, 2, 3])
x = np.array([4, 5, 6])
result = dot_product_vectorized(w, x)  # Fast
```

Vectorized Multiple Linear Regression

```python
# Vectorized prediction for multiple examples
def vectorized_prediction(X, w, b):
    """
    X: (m, n) - m examples, n features
    w: (n,) - weight vector
    b: scalar - bias
    Returns: (m,) - predictions for all examples
    """
    return X.dot(w) + b

# Example with multiple houses
X = np.array([[2104, 5, 1, 45],
              [1416, 3, 2, 40],
              [1534, 3, 2, 30]])

w = np.array([0.39, 18.75, -53.36, -26.42])
b = 785.18

predictions = vectorized_prediction(X, w, b)
```

---

3. Gradient Descent for Multiple Features {#gradient-descent-multiple}

Cost Function for Multiple Features

$$J(w_1,w_2,...,w_n,b) = (1/2m) * \Sigma(f(x□^{(i)}) - y^{(i)})^2$$

Gradient Descent Update Rules

Simultaneous updates for all parameters:

$$w_j = w_j - \alpha * \partial J(w□,b)/\partial w_j \quad \text{for } j = 1,2,...,n$$
$$b = b - \alpha * \partial J(w□,b)/\partial b$$

Partial Derivatives

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

Vectorized Gradient Descent Implementation

```python
def gradient_descent_multiple(X, y, w_init, b_init, alpha, iterations):
    m, n = X.shape
    w = w_init.copy()
    b = b_init
    cost_history = []

    for i in range(iterations):
        # Forward pass (vectorized)
        predictions = X.dot(w) + b

        # Compute cost
        cost = np.sum((predictions - y) ** 2) / (2 * m)
        cost_history.append(cost)

        # Compute gradients (vectorized)
        dw = X.T.dot(predictions - y) / m
        db = np.sum(predictions - y) / m

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

    return w, b, cost_history

# Example usage
X = np.random.randn(100, 4)  # 100 examples, 4 features
y = np.random.randn(100)     # 100 targets
w_init = np.zeros(4)
b_init = 0

w, b, costs = gradient_descent_multiple(X, y, w_init, b_init, 0.01, 1000)
```

---

4. Feature Scaling {#feature-scaling}

Why Feature Scaling?

When features have different scales, gradient descent can be slow and inefficient. Feature scaling ensures all features contribute equally to the learning process.

Problems Without Feature Scaling

- **Slow convergence**: Algorithm takes longer to reach minimum
- **Oscillation**: Parameters may oscillate before converging
- **Numerical instability**: Large numbers can cause precision issues

Feature Scaling Methods

1. Z-score Normalization (Standardization)

$$x_i = (x_i - \mu) / \sigma$$

Where:

- μ = mean of feature
- σ = standard deviation of feature

2. Min-Max Normalization

$$x_i = (x_i - \min) / (\max - \min)$$

3. Mean Normalization

$$x_i = (x_i - \mu) / (\max - \min)$$

Implementation Examples

```python
```

```python
# Z-score normalization
def z_score_normalize(X):
    mu = np.mean(X, axis=0)
    sigma = np.std(X, axis=0)
    X_norm = (X - mu) / sigma
    return X_norm, mu, sigma

# Min-max normalization
def min_max_normalize(X):
    X_min = np.min(X, axis=0)
    X_max = np.max(X, axis=0)
    X_norm = (X - X_min) / (X_max - X_min)
    return X_norm, X_min, X_max

# Example usage
X = np.array([[2104, 5, 1, 45],
              [1416, 3, 2, 40],
              [1534, 3, 2, 30]])

X_norm, mu, sigma = z_score_normalize(X)
print(f"Normalized features:\n{X_norm}")
```

When to Apply Feature Scaling

- **Always recommended** for gradient descent
- **Essential** when features have vastly different scales
- **Not needed** for tree-based algorithms
- **Apply to training and test sets** using training statistics

---

5. Convergence and Learning Rate {#convergence}

Checking for Convergence

Monitor the cost function to ensure gradient descent is working properly.

Convergence Indicators

1. **Decreasing cost**: Cost should decrease with each iteration
2. **Plateau**: Cost levels off when minimum is reached
3. **Automatic convergence test**: Stop when cost decrease $< \varepsilon$ (e.g., 0.001)

Learning Rate Selection

Learning Rate Too Small

- **Symptoms**: Very slow convergence
- **Solution**: Increase learning rate

Learning Rate Too Large

- **Symptoms**: Cost increases or oscillates
- **Solution**: Decrease learning rate

Debugging and Monitoring

```python
def plot_cost_function(cost_history):
    plt.figure(figsize=(10, 6))
    plt.plot(cost_history)
    plt.title('Cost Function Over Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show()

# Convergence checking
def check_convergence(cost_history, tolerance=1e-6):
    if len(cost_history) < 2:
        return False

    recent_change = abs(cost_history[-1] - cost_history[-2])
    return recent_change < tolerance

# Example usage
cost_history = [100, 95, 90, 85, 80, 79.99, 79.98]
converged = check_convergence(cost_history)
```

Learning Rate Recommendations

- **Start with**: $\alpha = 0.01$
- **Try**: 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1.0
- **Rule of thumb**: Roughly 3x increase/decrease between trials

---

6. Feature Engineering {#feature-engineering}

What is Feature Engineering?

The process of creating new features from existing ones to improve model performance.

Common Feature Engineering Techniques

1. Polynomial Features

Create higher-order terms:

```python
# Original features: $x_1$, $x_2$
# Polynomial features: $x_1$, $x_2$, $x_1^2$, $x_2^2$, $x_1x_2$
```

2. Feature Combinations

Combine existing features:

```python
# House price example
# New feature: price_per_sqft = price / size
# New feature: bedroom_to_bathroom_ratio = bedrooms / bathrooms
```

3. Transformations

Apply mathematical functions:

```python
# Logarithmic transformation
# Square root transformation
# Exponential transformation
```

Feature Engineering Examples

```python
```

```python
# Creating polynomial features
def create_polynomial_features(X, degree=2):
    m, n = X.shape
    # Start with original features
    X_poly = X.copy()

    # Add squared terms
    for i in range(n):
        X_poly = np.column_stack([X_poly, X[:, i] ** 2])

    # Add interaction terms (for degree 2)
    for i in range(n):
        for j in range(i+1, n):
            X_poly = np.column_stack([X_poly, X[:, i] * X[:, j]])

    return X_poly

# Example: House price features
def engineer_house_features(size, bedrooms, bathrooms, age):
    # Original features
    X = np.column_stack([size, bedrooms, bathrooms, age])

    # Engineered features
    price_per_sqft = size / 1000  # Normalize size
    bed_bath_ratio = bedrooms / bathrooms
    age_factor = np.log(age + 1)  # Log transform age

    # Combine all features
    X_engineered = np.column_stack([X, price_per_sqft, bed_bath_ratio, age_factor])
    return X_engineered
```

---

7. Polynomial Regression {#polynomial-regression}

Overview

Polynomial regression extends linear regression by adding polynomial terms, allowing the model to fit curved relationships.

Mathematical Representation

For single feature with degree n:

$$f(x) = w_0 + w_1x + w_2x^2 + w_3x^3 + ... + w_nx^n$$

Why Polynomial Regression?

- **Captures non-linear relationships**
- **Still uses linear regression techniques**
- **Flexible model complexity**

Implementation Example

python

```python
# Polynomial regression implementation
def polynomial_features(X, degree):
    """
    Create polynomial features up to specified degree
    """
    m = X.shape[0]
    X_poly = np.ones((m, 1))  # Start with bias column

    for d in range(1, degree + 1):
        X_poly = np.column_stack([X_poly, X ** d])

    return X_poly

# Example usage
def fit_polynomial_regression(X, y, degree=2, alpha=0.01, iterations=1000):
    # Create polynomial features
    X_poly = polynomial_features(X, degree)

    # Initialize parameters
    w = np.zeros(X_poly.shape[1])

    # Gradient descent
    for i in range(iterations):
        predictions = X_poly.dot(w)
        error = predictions - y
        gradient = X_poly.T.dot(error) / len(X)
        w = w - alpha * gradient

    return w

# Example: Fitting curved data
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([1, 4, 9, 16, 25])  # Quadratic relationship

w = fit_polynomial_regression(X.flatten(), y, degree=2)
print(f"Polynomial coefficients: {w}")
```

Overfitting Considerations

- **Higher degree**: More complex curves, risk of overfitting
- **Lower degree**: Simpler curves, risk of underfitting
- **Solution**: Use regularization (covered in later weeks)

8. Implementation Examples {#implementation}

Complete Multiple Linear Regression Class

python

```
class MultipleLinearRegression:
    def __init__(self, learning_rate=0.01, max_iterations=1000):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.w = None
        self.b = None
        self.cost_history = []
        self.feature_stats = {}

    def _normalize_features(self, X, fit=True):
        if fit:
            self.feature_stats['mean'] = np.mean(X, axis=0)
            self.feature_stats['std'] = np.std(X, axis=0)

        return (X - self.feature_stats['mean']) / self.feature_stats['std']

    def fit(self, X, y):
        # Normalize features
        X_norm = self._normalize_features(X, fit=True)

        # Initialize parameters
        m, n = X_norm.shape
        self.w = np.zeros(n)
        self.b = 0

        # Gradient descent
        for i in range(self.max_iterations):
            # Forward pass
            predictions = X_norm.dot(self.w) + self.b

            # Compute cost
            cost = np.sum((predictions - y) ** 2) / (2 * m)
            self.cost_history.append(cost)

            # Compute gradients
            dw = X_norm.T.dot(predictions - y) / m
            db = np.sum(predictions - y) / m

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

    def predict(self, X):
```

```
        X_norm = self._normalize_features(X, fit=False)
        return X_norm.dot(self.w) + self.b

# Example usage
X = np.random.randn(100, 3)
y = X.dot([1, 2, 3]) + np.random.randn(100) * 0.1

model = MultipleLinearRegression()
model.fit(X, y)
predictions = model.predict(X)
```

Polynomial Regression with Feature Scaling

python

```python
class PolynomialRegression:
    def __init__(self, degree=2, learning_rate=0.01, max_iterations=1000):
        self.degree = degree
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.w = None

    def _create_polynomial_features(self, X):
        m = X.shape[0]
        X_poly = np.ones((m, 1))

        for d in range(1, self.degree + 1):
            X_poly = np.column_stack([X_poly, X ** d])

        return X_poly

    def fit(self, X, y):
        X_poly = self._create_polynomial_features(X)

        # Feature scaling
        self.X_mean = np.mean(X_poly, axis=0)
        self.X_std = np.std(X_poly, axis=0)
        X_poly_norm = (X_poly - self.X_mean) / self.X_std

        # Initialize parameters
        self.w = np.zeros(X_poly_norm.shape[1])

        # Gradient descent
        m = len(X)
        for i in range(self.max_iterations):
            predictions = X_poly_norm.dot(self.w)
            error = predictions - y
            gradient = X_poly_norm.T.dot(error) / m
            self.w = self.w - self.learning_rate * gradient

    def predict(self, X):
        X_poly = self._create_polynomial_features(X)
        X_poly_norm = (X_poly - self.X_mean) / self.X_std
        return X_poly_norm.dot(self.w)
```

Key Takeaways

1. **Multiple Linear Regression**: Extends simple regression to handle multiple features using vectorization
2. **Vectorization**: Critical for efficient computation with large datasets
3. **Feature Scaling**: Essential for gradient descent convergence when features have different scales
4. **Convergence Monitoring**: Track cost function to ensure proper algorithm behavior
5. **Feature Engineering**: Create new features to improve model performance
6. **Polynomial Regression**: Captures non-linear relationships while maintaining linear regression framework
7. **Learning Rate**: Critical hyperparameter requiring careful tuning

---

Next Steps

- Week 3: Classification algorithms (Logistic Regression)
- Regularization techniques to prevent overfitting
- Advanced optimization algorithms
- Model evaluation and validation techniques