Univariate Linear Regression and Gradient Descent

Formatted by a Professional LaTeX Generator

October 10, 2025

Contents

Chapter 1

Introduction

1.1 Univariate Linear Regression

Univariate Linear Regression is a fundamental statistical and machine learning technique used to model the relationship between a single independent variable (also known as a feature or predictor) and a dependent variable (also known as a target or response). The goal is to find the best-fitting straight line through the data points that can be used to make predictions.

This relationship is represented by the simple linear equation:

$$y = mx + c \tag{1.1}$$

Where:

- **y** is the dependent variable.
- **x** is the independent variable.
- **m** is the slope of the line.
- **c** is the y-intercept.

In machine learning terminology, this is often written as:

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1.2}$$

Where:

- $h_\theta(x)$ is the hypothesis or predicted value.
- θ_0 is the bias term (intercept).
- θ_1 is the weight for the input feature x (slope).

Why it matters: Univariate linear regression is often the first algorithm taught in machine learning because of its simplicity and interpretability. It serves as a foundation for understanding more complex algorithms.

1.2 Gradient Descent

Gradient Descent is an iterative optimization algorithm used to find the minimum of a function. In the context of linear regression, it's used to find the optimal values for the parameters (θ_0 and θ_1) that minimize the cost function. The cost function measures how well the model is performing by quantifying the difference between the predicted values and the actual values.

This guide will explore how to implement univariate linear regression using gradient descent, both with and without vectorization, to understand the underlying mechanics and the benefits of vectorized computation.

Chapter 2

Deep Explanation

2.1 The Core Components

To understand univariate linear regression with gradient descent, we need to be familiar with three key components:

1. **Hypothesis Function:** This is the linear equation our model uses to make predictions.

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{2.1}$$

2. **Cost Function (Mean Squared Error - MSE):** The cost function measures the average squared difference between the predicted values and the actual values. Our goal is to minimize this function.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2.2}$$

   Where:

   - **m** is the number of training examples.
   - $x^{(i)}$ **and** $y^{(i)}$ are the i-th training example.

   The 1/2 is a convention to simplify the derivative calculation.

3. **Gradient Descent Algorithm:** This algorithm iteratively adjusts the parameters (θ_0 and θ_1) to minimize the cost function.

   - It works by taking steps in the direction of the steepest descent of the cost function.
   - The size of each step is determined by the learning rate (α).

2.2 Gradient Descent Algorithm: Step-by-Step

1. **Initialize Parameters:** Start with random or zero values for θ_0 and θ_1.

2. **Calculate the Gradient:** Compute the partial derivatives of the cost function with respect to each parameter. The gradient tells us the direction of the steepest ascent, so we move in the opposite direction.

   - For θ_0:

$$\frac{\partial J}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \tag{2.3}$$

- For θ_1:

$$\frac{\partial J}{\partial \theta_1} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)} \tag{2.4}$$

3. **Update Parameters:** Simultaneously update θ_0 and θ_1 using the gradients and the learning rate.

$$\theta_0 := \theta_0 - \alpha \frac{\partial J}{\partial \theta_0} \tag{2.5}$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial J}{\partial \theta_1} \tag{2.6}$$

4. **Repeat:** Repeat steps 2 and 3 for a fixed number of iterations or until the cost function converges (i.e., the change in cost is negligible).

2.3 Without Vectorization (Using Loops)

In a non-vectorized implementation, we use loops to iterate through each training example to calculate the sum of errors for the gradient.

2.3.1 The Process

1. Initialize θ_0 and θ_1.
2. Start a loop for the number of iterations.
3. Inside this loop, initialize variables for the sum of errors for θ_0 and θ_1 to zero.
4. Start another loop to iterate through each training example (i from 1 to m).
   (a) Calculate the predicted value: `prediction = ` θ_0 `+` θ_1 `*` $x^{(i)}$.
   (b) Calculate the error: `error = prediction -` $y^{(i)}$.
   (c) Add to the sum of errors for θ_0: `sum_error += error`.
   (d) Add to the sum of errors for θ_1: `sum_error += error *` $x^{(i)}$.
5. After the inner loop finishes, calculate the gradients:

$$\text{grad}_0 = \frac{1}{m} \cdot \text{sum\_error}_0$$

$$\text{grad}_1 = \frac{1}{m} \cdot \text{sum\_error}_1$$

6. Update the parameters:

$$\theta_0 = \theta_0 - \alpha \cdot \text{grad}_0$$

$$\theta_1 = \theta_1 - \alpha \cdot \text{grad}_1$$

7. Repeat for the specified number of iterations.

This approach is intuitive and easy to understand but can be computationally inefficient, especially with large datasets.

2.4 With Vectorization (Using Matrix Operations)

Vectorization allows us to perform calculations on entire arrays or matrices at once, eliminating the need for explicit loops. This is significantly faster due to optimized low-level library implementations (like NumPy in Python).

2.4.1 The Process

1. **Prepare the Data:**
    - Create a feature matrix X and add a column of ones to accommodate the bias term θ_0.
    - This makes the hypothesis function a dot product: $h_\theta(X) = X \cdot \theta$.
    - Create a target vector y.
    - Initialize a parameter vector θ with θ_0 and θ_1.
2. **Vectorized Gradient Descent:**
    (a) Initialize θ.
    (b) Start a loop for the number of iterations.
    (c) Calculate predictions for all examples at once: `predictions = X` $\cdot$ θ.
    (d) Calculate the error vector: `errors = predictions - y`.
    (e) Calculate the gradients using matrix multiplication: `gradient = (1/m) *` X^T $\cdot$ `errors`.
    (f) Update the parameter vector: θ `=` θ `-` α `* gradient`.
    (g) Repeat for the specified number of iterations.

2.4.2 Why Vectorization is Faster

- **Parallelism:** Matrix operations can be parallelized to take advantage of modern hardware (CPUs and GPUs).
- **Reduced Overhead:** Python loops have a higher overhead than optimized C or Fortran code used in libraries like NumPy.

Chapter 3

Examples

Let's consider a simple dataset where we want to predict a student's exam score based on the number of hours they studied.

Table 3.1: Student Study and Score Data

Hours Studied (x)	Exam Score (y)
1	2
2	4
3	5
4	4
5	5

3.1 Example 1: Without Vectorization

```python
import numpy as np

# Data
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

# Parameters
alpha = 0.01
iterations = 1000
m = len(y)
theta0 = 0
theta1 = 0

# Gradient Descent
for _ in range(iterations):
    sum_error0 = 0
    sum_error1 = 0
    for i in range(m):
        prediction = theta0 + theta1 * x[i]
        error = prediction - y[i]
        sum_error0 += error
        sum_error1 += error * x[i]

    grad0 = (1/m) * sum_error0
    grad1 = (1/m) * sum_error1

    theta0 = theta0 - alpha * grad0
    theta1 = theta1 - alpha * grad1

print(f"Theta0: {theta0}, Theta1: {theta1}")
```

Listing 3.1: Non-Vectorized Gradient Descent in Python

3.2 Example 2: With Vectorization

```python
import numpy as np

# Data
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

# Parameters
alpha = 0.01
iterations = 1000
m = len(y)

# Add a column of ones to x for the bias term
X = np.c_[np.ones(m), x]
theta = np.zeros(2)

# Gradient Descent
for _ in range(iterations):
    predictions = X.dot(theta)
    errors = predictions - y
    gradient = (1/m) * X.T.dot(errors)
    theta = theta - alpha * gradient

print(f"Theta: {theta}")
```

Listing 3.2: Vectorized Gradient Descent in Python

Both examples will converge to similar optimal values for θ_0 and θ_1.

Chapter 4

Related Concepts

- **Multivariate Linear Regression:** An extension of univariate linear regression with multiple independent variables.
- **Polynomial Regression:** A type of regression where the relationship between the independent and dependent variables is modeled as an n-th degree polynomial.
- **Regularization (L1 and L2):** Techniques used to prevent overfitting by adding a penalty term to the cost function.
- **Types of Gradient Descent:**
  - **Batch Gradient Descent:** Uses the entire training dataset to compute the gradient at each step.
  - **Stochastic Gradient Descent (SGD):** Uses a single training example to compute the gradient at each step.
  - **Mini-Batch Gradient Descent:** A compromise between batch and stochastic, using a small batch of training examples at each step.

Chapter 5

Assignments / Practice Questions

1. **MCQ:** What is the primary advantage of using vectorization in gradient descent?

   a) It is easier to write the code.

   b) It converges in fewer iterations.

   c) It is computationally more efficient.

   d) It always finds a better minimum.

2. **MCQ:** In the equation $h_\theta(x) = \theta_0 + \theta_1 x$, what does θ_1 represent?

   a) The y-intercept.

   b) The learning rate.

   c) The slope of the line.

   d) The number of training examples.

3. **Short Question:** Explain the role of the learning rate (α) in gradient descent. What happens if it's too large or too small?

4. **Problem-Solving:** Given the following data, perform two iterations of gradient descent *manually* (without code) for univariate linear regression. Use $\alpha = 0.1$, and initial $\theta_0 = 0$, $\theta_1 = 0$.

   - Data: $x = [1, 2]$, $y = [1, 3]$

5. **Coding Task:** Modify the provided vectorized Python code to also calculate and print the cost $J(\theta_0, \theta_1)$ at each iteration. Plot the cost over iterations to visualize convergence.

Chapter 6

Applications

Univariate linear regression is used in various fields for prediction and forecasting:

- **Finance:** Predicting stock prices based on a single economic indicator.
- **Economics:** Estimating the impact of price changes on product demand.
- **Sales and Marketing:** Forecasting sales based on advertising spending.
- **Healthcare:** Predicting blood pressure based on a patient's weight.
- **Environmental Science:** Modeling the relationship between temperature and ice cream sales.

Chapter 7

Related Study Resources

- **Coursera - Machine Learning by Andrew Ng:** A classic course that provides a thorough introduction to linear regression and gradient descent.
  - https://www.coursera.org/learn/machine-learning
- **MIT OpenCourseWare - Introduction to Machine Learning:** In-depth lectures and materials on machine learning fundamentals.
  - https://ocw.mit.edu/courses/6-036-introduction-to-machine-learning-fall-2020/
- **Google Developers - Linear Regression:** A concise and practical explanation of linear regression concepts.
  - https://developers.google.com/machine-learning/crash-course/descending-into-ml/linear-regression
- **NumPy Documentation:** The official documentation for the library used in the vectorized implementation.
  - https://numpy.org/doc/

Chapter 8

Summary / Key Takeaways

Table 8.1: Summary of Key Concepts

Concept	**Description**
Univariate Linear Regression	Models the relationship between one independent and one dependent variable using a straight line.
Hypothesis	$h_\theta(x) = \theta_0 + \theta_1 x$ (the predictive model).
Cost Function (MSE)	$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$ (measures model error).
Gradient Descent	An iterative algorithm to minimize the cost function by updating parameters in the opposite direction of the gradient.
Parameter Update Rule	$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j}$
Non-Vectorized Approach	Uses explicit 'for' loops to iterate over training examples. Slower but intuitive.
Vectorized Approach	Uses matrix operations to process all examples at once. Faster and more efficient.