Matthew Oyeniran on Multiple Linear Regression

September 14, 2024

1 Multiple Linear Regression

<img src="https://github.com/user-attachments/assets/b0ecbf61-83bf-4e46-93fd-78f72a3b9919" alt=

Multiple Linear Regression (MLR) is a powerful statistical technique used to model the relationship between a single dependent variable and multiple independent variables. Multiple Linear Regression (MLR) extends the concept of Simple Linear Regression (SLR) to model the relationship between a single dependent variable and multiple independent variables. While SLR examines how one independent variable influences the dependent variable, MLR allows us to explore how several factors interact to affect the outcome.

In Simple Linear Regression, the model is expressed as:

$$y = \beta_0 + \beta_1 X + \epsilon$$

where: - y is the dependent variable. - X is the single independent variable. - β_0 is the intercept. - β_1 is the coefficient for X - ϵ is the error term.

This model captures the linear relationship between y and X, providing a straightforward way to predict y based on X.

Multiple Linear Regression, on the other hand, generalizes this concept to accommodate multiple independent variables. The MLR model is given by:

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n$$

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i X_i$$

where: - y is the dependent variable. - β_0 is the intercept. - β_i are the coefficients for these variables. - X_i are multiple independent variables. - ϵ is the error term.

MLR aims to find the best-fitting linear equation that minimizes the sum of squared differences between the observed and predicted values of the dependent variable. By doing so, it provides insights into the strength and direction of relationships between variables, making it an invaluable tool for statistical analysis and decision-making.

There are two primary approaches to solve for the coefficients in multiple linear regression: the - Normal Equation - Gradient Descent.

1.1 Normal Equation

The Normal Equation is a direct method for finding the coefficients that minimize the sum of squared errors in a multiple linear regression model.

Given the data below;

Column 1 (X_1)	Column 2 (X_2)	Column 3 (X_3)	...	Column m (X_m)	Target (y)
Row 1, Col 1 (x_{11})	Row 1, Col 2 (x_{12})	Row 1, Col 3 (x_{13})	...	Row 1, Col m (x_{m1})	Row 1, Target
Row 2, Col 1 (x_{21})	Row 2, Col 2 (x_{22})	Row 2, Col 3 (x_{23})	...	Row 2, Col m (x_{m2})	Row 2, Target

$$\hat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_m X_m$$

Let n be the number of rows and m be the number of columns

$$\hat{y} = \begin{pmatrix} \hat{y_1} \\ \hat{y_2} \\ \vdots \\ \hat{y_n} \end{pmatrix} = \begin{pmatrix} \beta_0 & \beta_1 x_{11} & \beta_2 x_{12} & \cdots & \beta_m x_{1m} \\ \beta_0 & \beta_1 x_{21} & \beta_2 x_{22} & \cdots & \beta_m x_{2m} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \beta_0 & \beta_1 x_{n1} & \beta_2 x_{n2} & \cdots & \beta_m x_{nm} \end{pmatrix}$$

$$\hat{y} = \begin{pmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{pmatrix}$$

$$\hat{y} = X\beta$$

Objective: Minimize the Sum of Squared Residuals

We want to find β that minimizes the sum of squared residuals:

$$\text{SSE} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

In matrix form:

$$\text{SSE} = (\mathbf{y} - \hat{\mathbf{y}})^T (\mathbf{y} - \hat{\mathbf{y}})$$

Substitute $\hat{\mathbf{y}}$ from the model equation:

$$\text{SSE} = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta)$$

Expand the expression for SSE:

$$\text{SSE} = (\mathbf{y}^T - \beta^T \mathbf{X}^T)(\mathbf{y} - \mathbf{X}\beta)$$

$$\text{SSE} = \mathbf{y}^T \mathbf{y} - 2\beta^T \mathbf{X}^T \mathbf{y} + \beta^T \mathbf{X}^T \mathbf{X}\beta$$

To find the coefficients β that minimize SSE, we take the gradient of the SSE with respect to β and set it to zero:

$$\frac{\partial \text{SSE}}{\partial \beta} = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\beta$$

Set the gradient to zero:

$$-2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\beta = 0$$

Solve for β:

$$\mathbf{X}^T\mathbf{X}\beta = \mathbf{X}^T\mathbf{y}$$
$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

The formula for the coefficients β that minimizes the sum of squared errors in a multiple linear regression model is:

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Advantages	**Disadvantages**
1. Provides an exact solution 2. Easy to implement with linear algebra libraries	Computationally expensive for large datasets Memory intensive as it requires storing X^TX in memory

Advantages & Disadvantages of Normal Equation Let's write out the python computation

[10]:
```
import numpy as np
```

[11]:
```
class multiple_linear_regression:
    def __init__ (self):
        self.coef_ = None
        self.intercept_ = None

    def fit(self, X, y):
        X = np.insert(X, 0, 1, axis=1)

        # Calculate the coefficient
        betas =  np.linalg.inv(np.dot(X.T, X)).dot(X.T).dot(y)
        self.intercept_ = betas[0]
        self.coef_ = betas[1: ]

    def predict(self, X):
        y_pred = np.dot(X, self.coef_) + self.intercept_
        return y_pred
```

Let's import a data

[13]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_diabetes
```

```
from sklearn.metrics import r2_score
```

[ ]:

[14]:
```
# load the diabetes dataset
diabetes = load_diabetes()

x = diabetes.data
y = diabetes.target
```

[15]:
```
mlr = multiple_linear_regression()
```

[16]:
```
mlr.fit(x, y)
```

[17]:
```
y_pred = mlr.predict(x)
```

[18]:
```
mlr.coef_
```

[18]:
```
array([ -10.0098663 , -239.81564367,  519.84592005,  324.3846455 ,
       -792.17563855,  476.73902101,  101.04326794,  177.06323767,
        751.27369956,   67.62669218])
```

[19]:
```
mlr.intercept_
```

[19]:
```
152.13348416289597
```

[20]:
```
r2_score(y, y_pred)
```

[20]:
```
0.5177484222203498
```

[ ]:

1.1.1 Implementation with Sklearn Linear Model

[22]:
```
from sklearn.linear_model import LinearRegression
```

[23]:
```
model = LinearRegression()
```

[24]:
```
model.fit(x, y)
```

[24]:
```
LinearRegression()
```

[25]:
```
model.coef_
```

[25]:
```
array([ -10.0098663 , -239.81564367,  519.84592005,  324.3846455 ,
       -792.17563855,  476.73902101,  101.04326794,  177.06323767,
        751.27369956,   67.62669218])
```

[26]:
```
mlr.intercept_
```

[26]:
```
152.13348416289597
```

Scikit-learn's LinearRegression internally uses the Normal Equation (via a method called the Ordinary Least Squares, or OLS) to compute the best-fitting parameters for linear regression. It solves the following equation:

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

This is why the results (i.e., the coefficients and intercepts) from Scikit-learn's LinearRegression match exactly with those from the Normal Equation when implemented manually.

[ ]:

2 Gradient Descent Methods

Gradient Descent is a first-order optimization algorithm used to minimize a function by iteratively moving toward the point of minimum value. It is especially useful in situations where finding an analytical solution is difficult or infeasible. The core idea is to iteratively update the parameters in the direction opposite to the gradient of the cost function, leading to a minimum value.

The goal of gradient descent is to minimize the cost function L by updating the model parameters θ_j iteratively. The update rule for each parameter θ_j is:

$$\theta_j^{new} = \theta_j^{old} - \alpha\frac{\partial L}{\partial \theta_j}$$

where: - α is the learning rate, which controls the step size in the parameter updates. - $\frac{\partial L}{\partial \theta_j}$ is the gradient of the cost function with respect to θ_j.

The process begins with random initialization of $ $, followed by gradual improvements, with each step aimed at decreasing the cost function (the MSE) until the algorithm converges to a minimum.

An important aspect of Gradient Descent is the learning rate $ $, which controls the size of the steps taken during optimization. If the learning rate is too small, the algorithm will converge slowly, requiring many iterations, which can be time-consuming.

Conversely, if the learning rate is too large, the algorithm may overshoot the minimum, potentially causing divergence, where the parameter values grow larger without converging to a good solution.

It is also important to note that not all cost functions have a smooth, convex shape like a bowl. Some may have irregular terrains with holes, ridges, or plateaus, which can make convergence challenging. If the algorithm starts in a region with a poor local minimum or a flat plateau, it may take a long time to converge, or it may never reach the global minimum.

In linear regression, the cost function L is the Mean Squared Error (MSE). This function measures the average squared difference between the observed actual outcomes and the predictions made by the model. The formula for the cost function is:

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where: - n is the number of examples in the dataset. - y_i is the actual value for the i-th example. - $\hat{y}_i$ is the predicted value for the i-th example.

For linear regression, the predicted value $\hat{y}_i$ is given by:

$$\hat{y}_i = \theta_0 + \theta_1 x_{i1} + \theta_2 x_{i2} + \cdots + \theta_m x_{im}$$

where: - θ_0 is the intercept. - $\theta_1, \theta_2, \ldots, \theta_m$ are the coefficients of the features $x_{i1}, x_{i2}, \ldots, x_{im}$.

2.1 Gradient Descent Algorithm

The goal of gradient descent is to minimize the cost function L by updating the model parameters θ_j iteratively. The update rule for each parameter θ_j is:

$$\theta_j^{new} = \theta_j^{old} - \theta \frac{\partial L}{\partial \theta_j}$$

where: - θ is the learning rate, which controls the step size in the parameter updates. - $\frac{\partial L}{\partial \theta_j}$ is the gradient of the cost function with respect to θ_j.

2.1.1 Computing the Gradient

To use gradient descent, we need to compute the gradient of the cost function with respect to each parameter θ_j. Let's derive this gradient step-by-step.

1. **Cost Function Expression:**

The cost function L can be expressed as:

$$L = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

$$L = \frac{1}{n} \sum_{i=1}^{n} (y_i - (\theta_0 + \theta_1 x_{i1} + \theta_2 x_{i2} + \cdots + \theta_m x_{im}))^2$$

2. **Differentiate with Respect to θ_j:**

To find $\frac{\partial L}{\partial \theta_j}$, we apply the chain rule:

$$\frac{\partial L}{\partial \theta_j} = \frac{\partial}{\partial \theta_j} \left[\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \right]$$

Let $e_i = y_i - \hat{y}_i$. Thus:

$$\frac{\partial L}{\partial \theta_j} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial \theta_j} (e_i^2)$$

Using the chain rule:

$$\frac{\partial}{\partial \theta_j} e_i^2 = 2e_i \cdot \frac{\partial e_i}{\partial \theta_j}$$

Since $e_i = y_i - \hat{y}_i$:

$$\frac{\partial e_i}{\partial \theta_j} = -\frac{\partial \hat{y}_i}{\partial \theta_j}$$

3. **Differentiate $\hat{y}_i$:**

The partial derivative of $\hat{y}_i$ with respect to θ_j is:

$$\frac{\partial \hat{y}_i}{\partial \theta_j} = x_{ij}$$

Thus:

$$\frac{\partial e_i}{\partial \theta_j} = -x_{ij}$$

4. **Substitute and Simplify:**

Substitute back into the derivative:

$$\frac{\partial L}{\partial \theta_j} = \frac{1}{n} \sum_{i=1}^{n} \left[2(y_i - \hat{y}_i) \cdot (-x_{ij}) \right]$$

$$\frac{\partial L}{\partial \theta_j} = -\frac{2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i) x_{ij}$$

Thus:

$$\frac{\partial L}{\partial \theta_j} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_{ij}$$

2.1.2 Gradient Descent Update Rule

Using the computed gradient, the update rule for θ_j in gradient descent is:

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_{ij}$$

For a linear regression model, the update rules for the intercept θ_0 and the coefficients θ_j (where $j \neq 0$) follow from the same gradient calculation process. Specifically:

- **Intercept θ_0:**

$$\frac{\partial L}{\partial \theta_0} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) \cdot 1$$

Here, x_{i0} (the feature associated with θ_0) is always 1.

The update rule for the intercept θ_0 is:

$$\theta_0^{new} = \theta_0^{old} - \alpha \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)$$

- **Coefficient θ_j for $j \neq 0$:**

The update rule remains:

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_{ij}$$

This update rule is applied iteratively to adjust each parameter θ_j until the cost function L converges to a minimum value or a stopping criterion is met.

2.2 Algorithm for Gradient Descent

Input: - A dataset with `m` examples: $X = \{x_1, x_2, ..., x_m\}$ - Target values: $y = \{y_1, y_2, ..., y_m\}$ - Learning rate: α - Number of iterations: T - Initial parameters: θ (weights)

2.2.1 Steps:

1. **Initialize** the parameters θ (usually with small random values or zeros).
2. **For each iteration (t = 1, 2, ..., T)**:
    1. **Compute the predicted values for all examples**:

    $$\hat{y}_{(i)} = \theta_0 + \theta_1 x_{i1} + \theta_2 x_{i2} + \cdots + \theta_m x_{im}$$

    where $\hat{y}_{(i)}$ is the predicted output for the i-th training example.
    2. **Compute the cost function (Mean Squared Error)**:

    $$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$$

    3. **Compute the gradients** of the cost function with respect to each parameter θ_j:

    $$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) x_j^{(i)}$$

    for $j = 0, 1, ..., n$.

4. **Update the parameters** using the gradients:

$$\theta_j = \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

for each parameter θ_j.

3. **Repeat the above steps** until convergence (i.e., when the cost function decreases very little between iterations) or until the maximum number of iterations T is reached.
4. **Return the optimized parameters** θ.

Output: - The optimized parameters θ (intercept and coefficients) after T iterations or convergence.

[31]:
```python
import numpy as np

class GradientDescent:
    def __init__(self, learning_rate=0.01, iterations=100):
        self.coef_ = None
        self.intercept_ = None
        self.lr = learning_rate
        self.iterations = iterations
        self.cost_history = []

    def fit(self, X, y):
        # Initialize the coefficients
        self.intercept_ = 0
        self.coef_ = np.zeros(X.shape[1])  # Initialize to zero

        # Gradient Descent
        for i in range(self.iterations):
            y_hat = np.dot(X, self.coef_) + self.intercept_
            intercept_der = -2 * np.mean(y - y_hat)
            coef_der = -2 * np.dot((y - y_hat), X) / X.shape[0]

            # Update parameters
            self.intercept_ -= self.lr * intercept_der
            self.coef_ -= self.lr * coef_der

            # Compute cost
            cost = (1 / X.shape[0]) * np.sum((y - y_hat) ** 2)
            self.cost_history.append(cost)

        print(f'Intercept: {self.intercept_}')
        print(f'Coefficients: {self.coef_}')

    def predict(self, X):
        return np.dot(X, self.coef_) + self.intercept_
```

```
    def get_cost_history(self):
        return self.cost_history

    def plot_cost_history(self):
        plt.figure(figsize=(10, 6))
        plt.plot(range(self.iterations), self.cost_history, color='blue')
        plt.xlabel('Iterations')
        plt.ylabel('Cost')
        plt.title('Cost Function History')
        plt.grid(True)
        plt.show()
```

[32]:
```
gd = GradientDescent(learning_rate=0.35, iterations=250000)

# Fit the model
gd.fit(x, y)
```

```
Intercept: 152.13348416289597
Coefficients: [  -9.89157119 -239.68292052  520.14495864  324.26781081
-766.44014036
  456.31832592   89.5331217   173.77834342  741.68450862   67.72134932]
```

[33]:
```
# Make predictions
y_pred = bgd.predict(x)
r2_score(y, y_pred)
```

[33]:
```
0.5177441281821082
```

Grid Search for Best Learning Rate and Iterations

[71]:
```
learning_rates = [0.1, 0.25, 0.35]
iterations = [50000, 100000, 250000]

results = []

for lr in learning_rates:
    for iter in iterations:
        bgd = GradientDescent(learning_rate=lr, iterations=iter)
        bgd.fit(x, y)
        cost_history = bgd.get_cost_history()
        final_cost = cost_history[-1]
        results.append({
            'Learning Rate': lr,
            'Iterations': iter,
            'Cost History':  cost_history,
            'Final Cost': final_cost
        })
```

```
# convert results into dataframe
result_df = pd.DataFrame(results)

# best learning_rate and iteration
best_comb = result_df.loc[result_df['Final Cost'].idxmin()]
```

```
Intercept: 152.13348416289597
Coefficients: [  -7.03131212 -236.59821657  527.61413706  321.83012908
-163.94484411
  -26.05367838 -174.86635523  105.29574395  514.94130093   69.77288695]
Intercept: 152.13348416289597
Coefficients: [  -7.613792   -237.14817775  525.94375675  322.08370419
-274.26213975
   65.05177889 -129.73108409  112.38776129  557.91416458   69.50345617]
Intercept: 152.13348416289597
Coefficients: [  -8.67807511 -238.32152097  523.21275966  323.06961801
-502.45650646
  246.84805561  -28.52875943  140.09034747  643.3208494    68.69216308]
Intercept: 152.13348416289602
Coefficients: [  -7.84242667 -237.39244418  525.3418148   322.2709884
-322.02726994
  103.38249285 -108.87448217  117.64328961  575.93627346   69.34432415]
Intercept: 152.133484162896
Coefficients: [  -8.67807887 -238.32152517  523.2127501   323.06962165
-502.45732149
  246.84870316  -28.52839591  140.09044984  643.32115352   68.69216011]
Intercept: 152.13348416289597
Coefficients: [  -9.69830315 -239.46608025  520.63352154  324.07692876
-724.39403006
  422.95542806   70.72808928  168.41155299  726.01789214   67.87599815]
Intercept: 152.133484162896
Coefficients: [  -8.22814051 -237.81806791  524.35280878  322.62950937
-404.79069136
  169.30417933  -72.15346104  127.71748714  606.90512444   69.04955048]
Intercept: 152.133484162896
Coefficients: [  -9.1058421  -238.80136174  522.13121151  323.49179125
-595.50261729
  320.68210545   13.08180426  151.95997973  677.99217478   68.35006668]
Intercept: 152.13348416289597
Coefficients: [  -9.89157119 -239.68292052  520.14495864  324.26781081
-766.44014036
  456.31832592   89.5331217   173.77834342  741.68450862   67.72134932]
```

[72]:
```
# Display the results summary and best combination
print("\nResults Summary:")
print(result_df[['Learning Rate', 'Iterations', 'Final Cost']])
print(f"\nBest Combination: Learning Rate: {best_comb['Learning Rate']}, "
```

```
    f"Iterations: {best_comb['Iterations']} with Final Cost:␣
↪{best_comb['Final Cost']}")
```

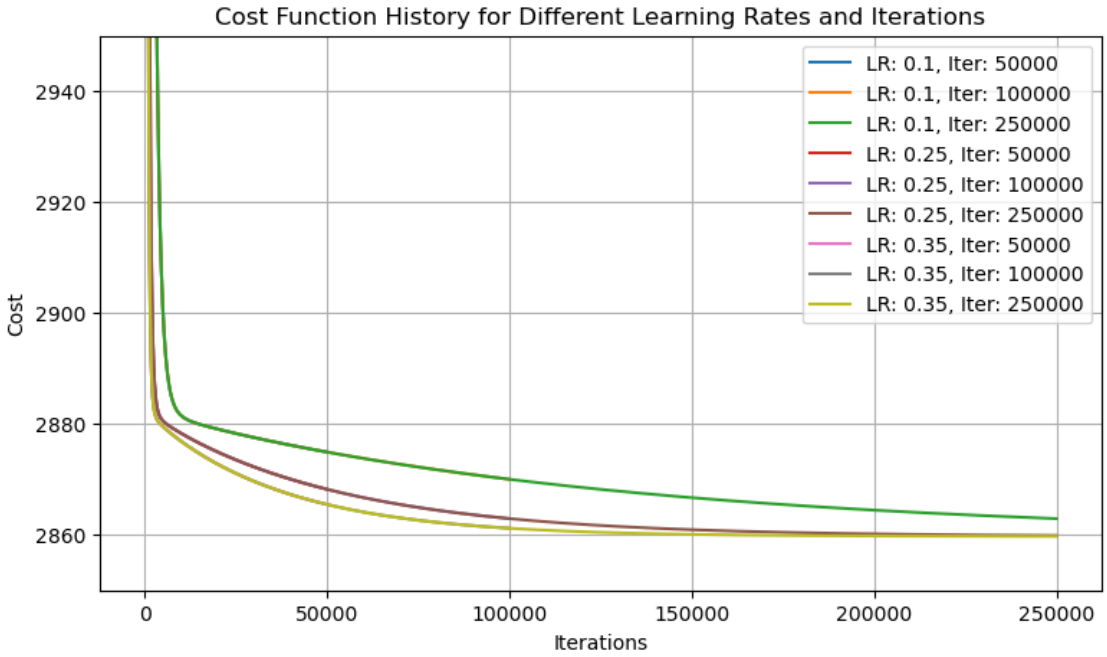
```
Results Summary:
   Learning Rate  Iterations   Final Cost
0           0.10       50000  2874.913514
1           0.10      100000  2870.012602
2           0.10      250000  2862.923399
3           0.25       50000  2868.195787
4           0.25      100000  2862.923418
5           0.25      250000  2859.872984
6           0.35       50000  2865.466095
7           0.35      100000  2861.183476
8           0.35      250000  2859.721811

Best Combination: Learning Rate: 0.35, Iterations: 250000 with Final Cost:
2859.7218114297257
```

[101]:
```
# Plot cost function history
fig, ax = plt.subplots(figsize=(9, 5))

# Plot cost history for each combination
for result in results:
    ax.plot(result['Cost History'], label=f"LR: {result['Learning Rate']}, Iter:
↪ {result['Iterations']}")

ax.set_xlabel('Iterations')
ax.set_ylabel('Cost')
ax.set_ylim(ymin=2850, ymax=2950)
ax.set_title('Cost Function History for Different Learning Rates and␣
↪Iterations')
ax.legend()
plt.grid(True)
plt.show()
```



2.3 Conclusion

In this project, I implemented multiple linear regression using three different approaches: the Normal Equation, Scikit-learn's `LinearRegression` module, and Gradient Descent. Both the Normal Equation and Scikit-learn yielded identical coefficient values (`coef_`) and intercepts (`intercept_`). However, Gradient Descent required careful tuning of the learning rate ( ) and a large number of iterations to converge to similar values.

For Gradient Descent, a moderate learning rate and a high number of iterations were essential to achieve comparable coefficients and intercepts. The learning rate controls how large the parameter updates are, and too large a rate can cause divergence, while too small a rate slows convergence significantly. The number of iterations determines how many times the parameters are updated to minimize the cost function, and insufficient iterations will prevent reaching optimal values.

In terms of practical use:

- The Normal Equation is efficient for small to medium-sized datasets, as it involves matrix inversion, which is computationally expensive for large datasets.
- Gradient Descent, on the other hand, can handle much larger datasets since it does not require matrix inversion, but it needs tuning of the learning rate and can take time to converge.

2.3.1 Advantages and Disadvantages of Gradient Descent vs. Normal Equation

Method	Advantages	Disadvantages
Gradient Descent	- Can be used for large datasets	- Requires tuning of learning rate

Method	Advantages	Disadvantages
Normal Equation	- Memory efficient (does not require storing entire dataset) - Works with any differentiable cost function - Fast for small to medium-sized datasets - No need to choose learning rate or set iterations - Deterministic (direct solution)	- Slow convergence without proper learning rate and iterations - May get stuck in local minima in non-convex problems - Computationally expensive for large datasets (due to matrix inversion) - Memory intensive for large datasets - Cannot be used with non-invertible matrices

2.3.2 Choosing the Right Approach

1. **Normal Equation** is preferable when the dataset is small to medium-sized and you need an exact solution without iterating or tuning any hyperparameters. It is computationally more efficient when the number of features is small.
2. **Gradient Descent** is better suited for very large datasets where matrix inversion becomes infeasible. It also provides flexibility in terms of different optimization objectives, though it requires careful tuning of the learning rate and the number of iterations.

In conclusion, the choice between the Normal Equation and Gradient Descent depends on the size of the dataset, the computational resources available, and whether you prioritize exact solutions or scalability.

[ ]: