Optimizing Multivariable Linear Regression with Vectorization Techniques

Steven Owen Liauw - 13523103
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523103@itb.ac.id, *owenliauw05@gmail.com*

***Abstract*— This paper explores the optimization of multivariable linear regression through the use of vectorization methods to improve computational efficiency. Conventional methods for linear regression, like iterative gradient descent, frequently need several iterations and can be resource-intensive for extensive datasets. Through the use of vectorization, these repetitive tasks are replaced with effective matrix and vector calculations, decreasing processing duration while keeping the precision. Furthermore, the closed-form solution for linear regression, derived from the normal equation, is considered as an alternative to iterative techniques. A comparative study will show the benefits and drawbacks of vectorization methods and the closed-form technique, stressing their effectiveness in various situations. This paper shows how vectorized calculations simplify linear regression tasks and provide efficient implementations.**

***Keywords*—Vector, Linear Regression, Matrix, Optimization.**

I. Introduction

Linear regression is a core algorithm in statistics and machine learning, commonly used for predictive analysis and exploring the connections between variables. It serves as the foundation for more complex models and is frequently the initial step in examining and interpreting data. Although the principles of linear regression are quite straightforward, efficiently applying it, particularly for extensive datasets or high-dimensional information, presents considerable computational difficulties.

Historically, the iterative technique using gradient descent has been the conventional approach for addressing linear regression issues. Gradient descent is an optimization technique that progressively refines the model parameters to reduce the cost function, reflecting the difference between predicted and actual outcomes. Although gradient descent is effective, it can be costly in terms of computation since it frequently requires several passes through the dataset, and its performance is greatly influenced by hyperparameters like the learning rate and iteration count. This repetitive process turns into a limitation when handling large datasets or complex features, resulting in longer computational times and greater resource use.

Vectorization methods provide an effective answer to these issues by substituting iterative calculations with matrix and vector operations. Utilizing the advantages of linear algebra, vectorized calculations handle complete datasets simultaneously, greatly minimizing the expenses associated with loops. For instance, rather than adjusting model parameters individually for each sample, vectorization enables concurrent updates through the use of matrix multiplication. This method not only accelerates calculations but also leverages modern hardware designs, like GPUs and enhanced linear algebra libraries, specifically built to perform matrix operations effectively.

A different option to iterative techniques is the closed-form solution for linear regression, obtained from the normal equation. The closed-form solution offers a precise mathematical equation to determine the optimal model parameters without requiring iterative adjustments.

This study explores the effectiveness of vectorization methods and the closed-form solution for enhancing multivariable linear regression. Through a comparison of these methods against the conventional iterative approach, the study aim to highlight their advantages and disadvantages, offering insights into their appropriateness for different situations.

II. Theoretical Framework

A. *Vector*

In mathematics and computing domains, vectors are essential because they offer an organized method of representing data and carrying out computations quickly. Vectors allow features, parameters, and target values to be represented in machine learning and linear regression, serving as the foundation for complex calculations. The notion of vectors and the fundamental procedures that enable their application in resolving linear regression issues are examined in this framework.

1. Definition of vector

A vector is an ordered set of elements, often written as:

$$v = [v_1, v_2, \dots, v_n]$$

Where v_i represents the i_{th} component of the vector, and n is the dimension of the vector. Vectors can be represented geometrically as directed line segments in space or algebraically as one-dimensional arrays of scalars.

There are two primary types of vectors:

- Row Vectors: Represented as $v = [v_1, v_2, \dots, v_n]$
- Column Vectors: Represented as $v = \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix}$

Depending on the application or challenge, vectors can exist in any dimension, from higher-dimensional spaces to one-dimensional arrays.

2. Core Vector Operations

- Addition and Subtraction

The addition and substraction of two vectors $a = [a_1, a_2, \dots, a_n]$ and $b = [b_1, b_2, \dots, b_n]$ is performed element wise:

$$c = a + b, \quad c_i = a_i + b_i$$
$$c = a - b, \quad c_i = a_i - b_i$$

- Scalar Multiplication

A vector can be scaled by multiplying each element by a scalar α :

$$\mathrm{d} = \alpha \mathrm{a}, \mathrm{d}_\mathrm{i} = \alpha \mathrm{a}_\mathrm{i}$$

- Dot Product

The dot product (or scalar product) of two vectors a b is a scalar defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

The dot product is essential for computing projections and is frequently used to gauge how well two vectors align.

- Cross Product (for 3D Vectors)

For three-dimensional vectors a and b the cross product produces a new vector orthogonal to both:

$$c = a \times b$$

- Norm (Magnitude)

The norm or magnitude of a vector represents its length and is computed as:

$$|v| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

- Unit Vector

A unit vector is obtained by normalizing a vector a, giving it a magnitude of 1:

$$u = \frac{v}{|v|}$$

3. Vector Transformation

Vectors can be transformed by applying functions or interacting with matrices:

- Matrix-Vector Multiplication

Given a matrix A of size $m\ x\ m$ and vector x of size n, the product is is a vector b of size m:

$$\mathrm{b} = \mathrm{Ax}$$

- Projection

A vector b can be projected onto another vector a as:

$$\mathrm{Proj_{ab}} = \frac{\mathrm{a} \cdot \mathrm{b}}{|\mathrm{a}|^2} \mathrm{a}$$

- Cosine Similarity (Cos Product)

measures the cosine of the angle between two vectors, providing a similarity score between −1 (opposite) and 1 (identical direction).

$$\cos \theta = \frac{a \cdot b}{|a||b|}$$

4. Vector Representation

- Parametric Representation

allows a vector to change along a line, plane, or higher-dimensional space by expressing it in terms of a parameter. For a line passing through a point p and extending in the direction of d, the parametric form is :

$$r(t) = p + td, \quad t \in R$$

For a plane, the parametric representation becomes:

$$r(s,t) = p + sd_1 + td_2, \quad s,t \in R$$

- Linear Combination of Vectors

A linear combination of vectors represents a vector as weighted sum of other vectors. Given a set of vectors $v_1, v_2, \dots, v_n$, a linear combination is:

$$v = c_1 v_1 + c_2 v_2 + \dots + c_n v_n$$

B. Matrix

1. Definition of a Matrix

Matrices are two-dimensional arrays of numbers arranged in rows and columns. A matrix A of size $m\ x\ n$ can be viewed as a collection of m row vectors or n column vectors.

$$A = [a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Figure 1. Matrix
Source: [3]

2. Types of Matrices

Matrices can be classified based on their structure and properties:

- Row Matrix: A matrix with only one row, e.g., $1\ x\ n$.
- Column Matrix: A matrix with only one column, e.g., $m\ x\ 1$.
- Square Matrix: A matrix with the same number of rows and columns, e.g., $n\ x\ n$.
- Diagonal Matrix: A square matrix where all elements outside the main diagonal are zero.

- Identity Matrix: A diagonal matrix where all diagonal elements are 1.

3. Matrix Operations
- Addition and Subtraction

Two matrices A and B of the same dimensions can be added or subtracted element-wise:

$$C = A + B, \quad c_{ij} = a_{ij} + b_{ij}$$

- Scalar Multiplication

A matrix can be multiplied by a scalar K, scaling each element:

$$C = kA, \quad c_{ij} = k \cdot a_{ij}$$

- Matrix Multiplication

If A is $m\ x\ n$ and B is $n\ x\ p$, their product $C = AB$ is an $m\ x\ p$ matrix:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

- Transpose

The transpose of a matrix A flips its rows and columns

- Determinant

For a square matrix A the determinant provides important information about the matrix, such as whether it is invertible and the scaling factor of the transformation represented by the matrix.

- Inverse

The inverse of matrix A is A^{-1} and the relation between them is expressed as:

$$AA^{-1} = A^{-1}A = I$$

C. *Linear Regression*

Linear regression is a fundamental algorithm in machine learning used for depicting the connection between one or several independent variables (features) and a dependent variable (target). The algorithm assumes a linear connection between the features and the target, trying to identify the optimal line by reducing the error, typically assessed using a loss or cost function.

The process of optimizing linear regression can be achieved through iterative techniques such as gradient descent or by using a closed-form solution for direct computation. This framework examines both methods, emphasizing their mathematical foundations and computational efficiencies.

1. Definition of Linear Regression

In linear regression, the prediction ŷ for given input x is modeled as:

$$\hat{y} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

This can be written in vectorized form as:

$$\hat{y} = w^T x + b$$

Where:
- $x = [x_1, x_2, \ldots, x_n]$ is Feature vector.
- $w = [w_1, w_2, \ldots, w_n]$ is Weight vector (coefficients).
- b = Bias term (intercept).

2. Loss Function

The most commonly used loss function in linear regression is the Mean Squared Error (MSE), defined as:

$$L(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(\widehat{y^{(i)}} - y^{(i)}\right)^2$$

Where:
- $\widehat{y^{(i)}}$ is Predicted value for the i_{th} sample.
- $y^{(i)}$ is True target value for the i_{th} sample.
- m is Number of samples.

The goal of linear regression is to minimize this loss function by optimizing w and b.

3. Computing Linear Regression

Linear regression can be calculated through two primary methods: iterative looping (such as gradient descent) and direct vectorized techniques (closed form).

- Iterative Looping Approach

Gradient descent is an iterative optimization algorithm used to minimize the loss function. It updates the weights and bias using the gradients of the loss function with respect to w and b.

Gradient Descent Update Rules:

$$w_j \leftarrow w_j - \alpha \frac{\partial L}{\partial w_j}, \quad b \leftarrow b - \alpha \frac{\partial L}{\partial b}$$

Where α is the learning rate, and the partial derivatives are:

$$\frac{\partial L}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left(\widehat{y^{(i)}} - y^{(i)}\right) x_j^{(i)}$$

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \left(\widehat{y^{(i)}} - y^{(i)}\right)$$

- Vectorized Approach (Closed Form Solution)

The vectorized or closed-form solution directly computes the optimal values of w and b by solving the normal equation:

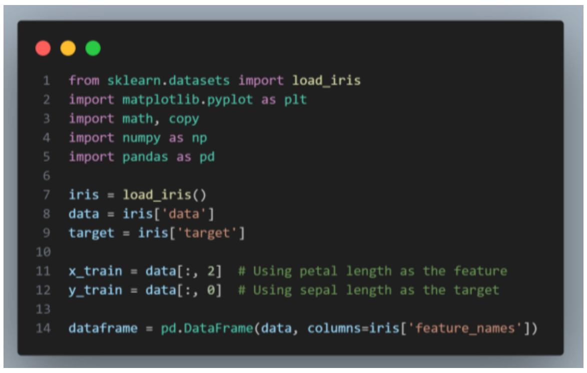
$$w = (X^T X)^{-1} X^T y$$

Where:
- X^T is Transpose of the feature matrix.

- $(X^T X)^{-1}$ is Inverse of Gram matrix.

III. Implementation

A. Initialization

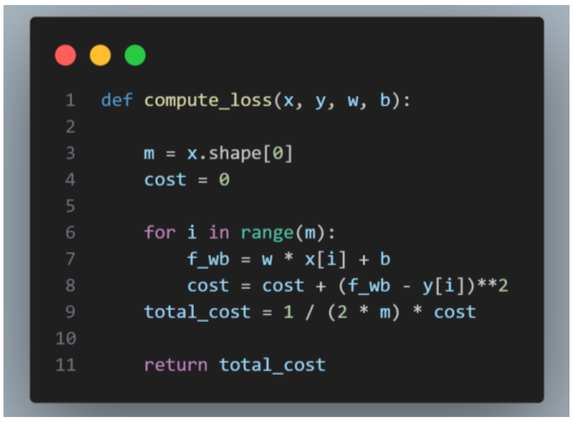


```python
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import math, copy
import numpy as np
import pandas as pd

iris = load_iris()
data = iris['data']
target = iris['target']

x_train = data[:, 2]  # Using petal length as the feature
y_train = data[:, 0]  # Using sepal length as the target

dataframe = pd.DataFrame(data, columns=iris['feature_names'])
```

Figure 2. Initialization code
Source: writer's archive

This code sets up and handles the Iris dataset, a popular dataset in machine learning. It loads the dataset with sklearn.datasets.load_iris, obtaining the feature matrix (data) and target labels (target). The particular feature petal length (data[:, 2]) is designated as x_train for the feature variable, whereas sepal length (data[:, 0]) is assigned as y_train for the target variable.

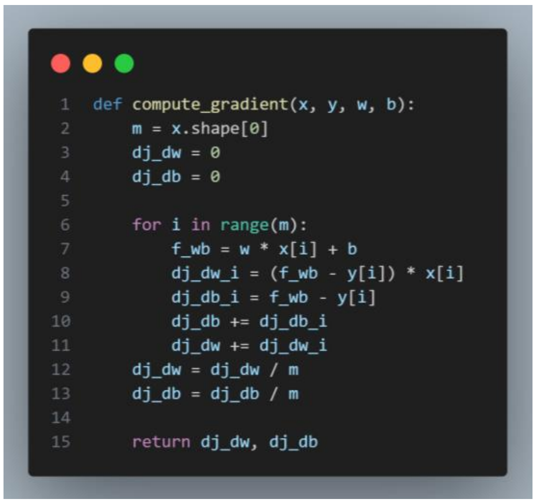
B. Linear Regression with Iterative Approach



```python
def compute_loss(x, y, w, b):

    m = x.shape[0]
    cost = 0

    for i in range(m):
        f_wb = w * x[i] + b
        cost = cost + (f_wb - y[i])**2
    total_cost = 1 / (2 * m) * cost

    return total_cost
```

Figure 3. compute_loss function
Source: writer's archive

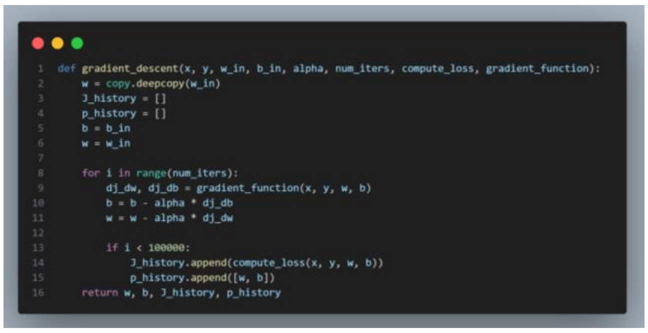
The compute_loss function calculates the Mean Squared Error (MSE) cost linked to linear regression. It accepts the input data x (features), y (responses), and the model parameters w (coefficients) and b (bias). The function goes through each data point, computes the predicted value based on the linear equation, and finds the square of the difference between the predicted value and the actual target value.



```python
def compute_gradient(x, y, w, b):
    m = x.shape[0]
    dj_dw = 0
    dj_db = 0

    for i in range(m):
        f_wb = w * x[i] + b
        dj_dw_i = (f_wb - y[i]) * x[i]
        dj_db_i = f_wb - y[i]
        dj_db += dj_db_i
        dj_dw += dj_dw_i
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_dw, dj_db
```

Figure 4. compute_gradient function
Source: writer's archive

The compute_gradient function calculates the gradients of the loss function with respect to the model parameters w (weight) b (bias) for a linear regression model. The function takes the feature array x, target array y, and the current model parameters w and b as inputs. It iterates through all data points to compute the predicted value, accumulates the gradients for each parameter, and normalizes them by dividing by the total number of samples. These gradients, dj_dw and dj_db, are used in gradient descent to update the model parameters and minimize the loss.

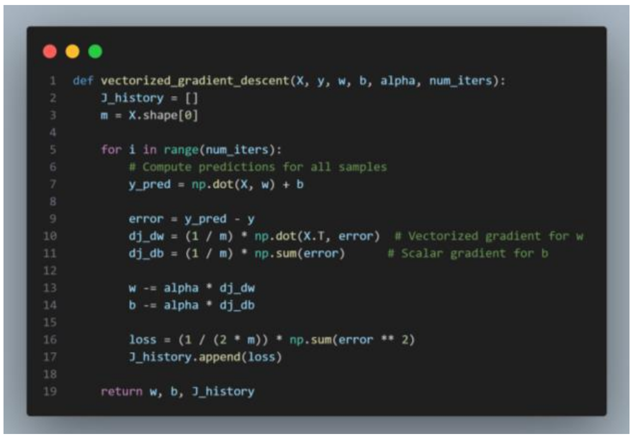


```python
def gradient_descent(x, y, w_in, b_in, alpha, num_iters, compute_loss, gradient_function):
    w = copy.deepcopy(w_in)
    J_history = []
    p_history = []
    b = b_in
    w = w_in

    for i in range(num_iters):
        dj_dw, dj_db = gradient_function(x, y, w, b)
        b = b - alpha * dj_db
        w = w - alpha * dj_dw

        if i < 100000:
            J_history.append(compute_loss(x, y, w, b))
            p_history.append([w, b])
    return w, b, J_history, p_history
```

Figure 5. gradient_descent function
Source: writer's archive

The gradient_descent function optimizes a linear regression model by repeatedly adjusting the model parameters w (weight) and b (bias) to reduce the loss function. The function receives the feature array x, the target array y, initial parameters w_in and b_in, the learning rate alpha, the number of iterations num_iters, along with the compute_loss and gradient_function as inputs.

C. Linear Regression with Vectorized Approach



```python
def vectorized_gradient_descent(X, y, w, b, alpha, num_iters):
    J_history = []
    m = X.shape[0]

    for i in range(num_iters):
        # Compute predictions for all samples
        y_pred = np.dot(X, w) + b

        error = y_pred - y
        dj_dw = (1 / m) * np.dot(X.T, error)  # Vectorized gradient for w
        dj_db = (1 / m) * np.sum(error)      # Scalar gradient for b

        w -= alpha * dj_dw
        b -= alpha * dj_db

        loss = (1 / (2 * m)) * np.sum(error ** 2)
        J_history.append(loss)

    return w, b, J_history
```

Figure 6. vectorized_gradient_decent function
Source: writer's archive

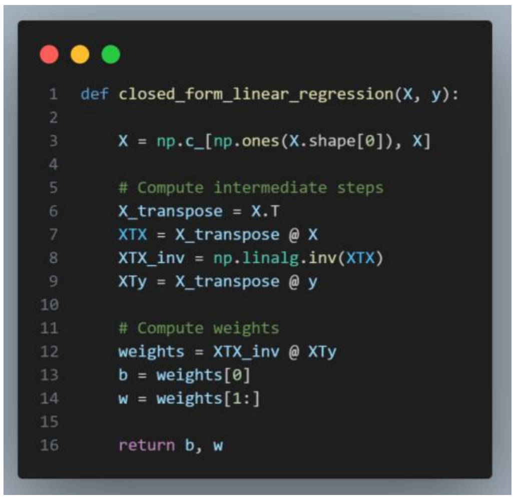
The vectorized_gradient_descent function performs gradient descent for linear regression using a fully vectorized approach. Predictions for all samples are computed simultaneously as $\hat{y} = X.w + b$, and the gradients are calculated in a vectorized form:

$$\frac{\partial L}{\partial w} = \frac{1}{m} X^T \cdot (\hat{y} - y)$$

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum (\hat{y} - y)$$

These gradients enable effective modifications of the parameters without going through each individual sample. This vectorized method enhances the algorithm's speed and is more suitable for larger datasets. The function returns the optimized parameters along with the loss history.

D. Linear Regression with Closed Form

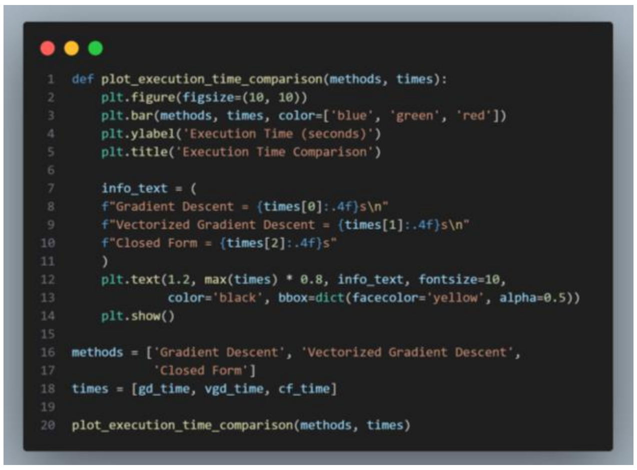


```python
def closed_form_linear_regression(X, y):

    X = np.c_[np.ones(X.shape[0]), X]

    # Compute intermediate steps
    X_transpose = X.T
    XTX = X_transpose @ X
    XTX_inv = np.linalg.inv(XTX)
    XTy = X_transpose @ y

    # Compute weights
    weights = XTX_inv @ XTy
    b = weights[0]
    w = weights[1:]

    return b, w
```

Figure 7. closed_form_linear_regression function
Source: writer's archive

The closed_form_linear_regression function computes the optimal weights w and bias b for linear regression using the normal equation. It augments the feature matrix X with a column of ones to account for bias term and perform matrix operations to calculate $w = (X^T X)^{-1} X^T Y$. The function then extracts b from the first element of the result and the remaining elements as w.

E. Time Comparison Visualization



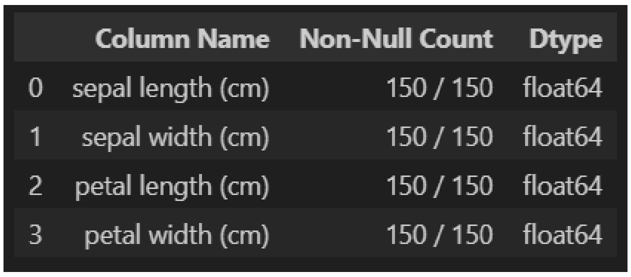
```python
def plot_execution_time_comparison(methods, times):
    plt.figure(figsize=(10, 10))
    plt.bar(methods, times, color=['blue', 'green', 'red'])
    plt.ylabel('Execution Time (seconds)')
    plt.title('Execution Time Comparison')

    info_text = (
    f"Gradient Descent = {times[0]:.4f}s\n"
    f"Vectorized Gradient Descent = {times[1]:.4f}s\n"
    f"Closed Form = {times[2]:.4f}s"
    )
    plt.text(1.2, max(times) * 0.8, info_text, fontsize=10,
             color='black', bbox=dict(facecolor='yellow', alpha=0.5))
    plt.show()

methods = ['Gradient Descent', 'Vectorized Gradient Descent',
           'Closed Form']
times = [gd_time, vgd_time, cf_time]

plot_execution_time_comparison(methods, times)
```

Fig 8. Plot_execution_time_comparison
Source: writer's archive

The plot_execution_time_comparison function uses a bar chart to show the execution timings of several techniques (such as Gradient Descent, Vectorized Gradient Descent, and Closed Form). It takes in times (the corresponding execution times in seconds) and methods (a list of method names). The function shows a title, plots the bars, and labels the y-axis with "Execution Time (seconds)". Additionally, it overlays a text box on the chart that is dynamically positioned in relation to the bar heights and contains specific execution timings for each method.

IV. Results and Analysis

A. Initial Dataset

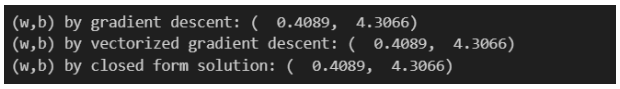


	Column Name	Non-Null Count	Dtype
0	sepal length (cm)	150 / 150	float64
1	sepal width (cm)	150 / 150	float64
2	petal length (cm)	150 / 150	float64
3	petal width (cm)	150 / 150	float64

Figure 9. Initial Dataset Information
Source: writer's archive

The initially loaded dataset contains 4 features and 150 rows, with one feature (sepal length) designated as the target variable. All features are of type float64, and there are no null values in the dataset.

B. Final w and b



```
(w,b) by gradient descent: (  0.4089,  4.3066)
(w,b) by vectorized gradient descent: (  0.4089,  4.3066)
(w,b) by closed form solution: (  0.4089,  4.3066)
```

Figure 10. Final w and b values
Source: writer's archive

The final w (weights) and b (bias) values remain consistent throughout the three methods: Gradient Descent, Vectorized Gradient Descent, and the Closed-Form solution. This result emphasizes the accuracy of every method in addressing the linear regression issue, since they reach the same ideal parameters. Gradient Descent and its vectorized version progressively reduce the loss function over 10,000 iterations using a learning rate (α) of $1x10^{-2}$, ensuring a precise output. Whereas the Closed-Form solution directly calculates the optimal parameters through the normal equation. Although they use different computational methods, the ultimate outcomes confirm the mathematical equivalence of these techniques when applied properly.

C. Time Comparison

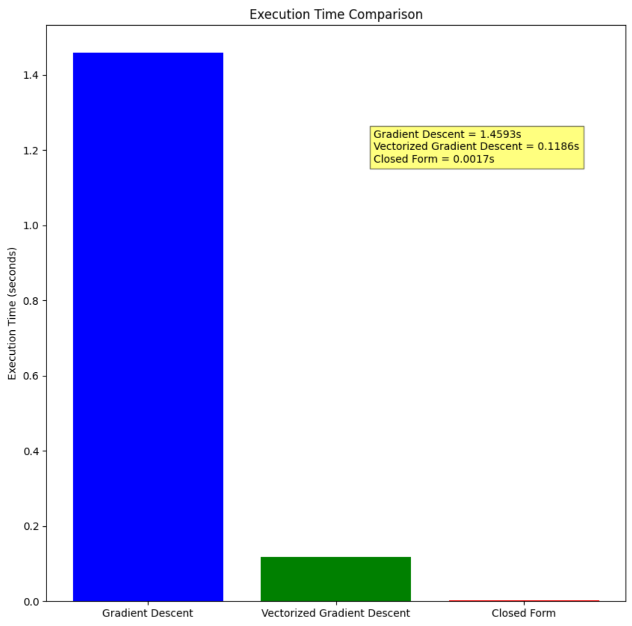


Fig 11. Time Comparison Chart
Source: writer's archive

The time comparison chart shows that the Closed-Form solution is the quickest of the three methods, followed by Vectorized Gradient Descent and lastly Gradient Descent. This outcome is expected because of the characteristics of the computational procedure for each method. The Closed-Form solution computes the ideal parameters directly through matrix operations, avoiding iterative updates and achieving convergence in one step, which makes it very efficient.

Vectorized Gradient Descent, although it remains iterative, greatly advantages from using optimized NumPy libraries. These libraries are created to efficiently manage extensive matrix and vector operations, allowing concurrent updates for all data points and minimizing computation time relative to the traditional Gradient Descent method. This vectorized approach reduces overhead and fully utilizes modern hardware acceleration. Gradient Descent, by contrast, executes these updates for every data point iteratively, making it more computationally intensive, particularly as the size of the dataset increases.

These findings highlight the necessity of choosing the right approach according to the dataset size and available computational resources. The Closed-Form solution works best for small to medium datasets, whereas Vectorized Gradient Descent provides a good mix of efficiency and scalability for larger datasets. Standard Gradient Descent, although less speedy, remains beneficial in scenarios where a straightforward approach or incremental monitoring is needed.

V. Conclusion

This study compared three approaches for solving linear regression problems: Gradient Descent, Vectorized Gradient Descent, and the Closed-Form solution. The results showed that although all techniques yielded identical final values for the model parameters (w and b), their computational efficiency differed notably.

The Closed-Form solution proved to be the quickest, thanks to its straightforward calculation of parameters via the normal equation without requiring iterative updates. Nonetheless, the computational expense may become excessive for extremely large datasets because of the matrix inversion process. Vectorized Gradient Descent provides a mix of speed and scalability, greatly enhanced by optimized NumPy libraries for matrix and vector computations, making it an ideal option for extensive datasets. Gradient Descent is the slowest method because of its non-vectorized iterative updates, making it less ideal for modern applications with large-scale data.

This study highlights the significance of choosing the right technique according to dataset size, available computational resources, and the particular needs of the issue. For datasets that are small to medium in size, the Closed-Form solution is the best choice. For bigger datasets, Vectorized Gradient Descent offers a more effective and scalable option.

VI. APPENDIX

Video Link :
https://www.canva.com/design/DAGbCzc9EMM/PtwpeA7qbBHVizJL0-mldQ/view?utm_content=DAGbCzc9EMM&utm_campaign=designshare&utm_medium=link&utm_source=recording_view

VII. Acknowledgment

The completion of this paper would not have been possible without the support and encouragement of many people. First, I thank God for providing the strength, wisdom, and determination to finish this work.

I sincerely thank Ir. Rila Mandala, M.Eng, Ph.D the lecturer for Linear Algebra and Geometrics (K-01), whose insightful lectures and helpful feedback were crucial in developing the concepts examined in this paper.

I am also deeply grateful to my parents for their constant

support, love, and encouragement throughout my studies. Finally, I extend my thanks to my friends, whose guidance and companionship made this work more enjoyable and meaningful.

REFERENCES

[1] Alshammari, Ahmad. (2024). Implementation of Linear Regression using Least Squares and Gradient Descent in Python. International Journal of Computer Applications. 186. 52-57. 10.5120/ijca2024923446.

[2] Geng, Yu & Li, Qin & Yang, Geng & Qiu, Wan. (2024). Linear Regression. 10.1007/978-981-97-3954-7_3.

[3] "Homepage Rinaldi Munir." https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-08-Determinan-bagian1-2023.pdf (accessed Dec. 28, 2024).

[4] H. Anton and C. Rorres, Elementary linear algebra: With Supplemental Applications. International student version. 2015.

[5] Kumari, Khushbu & Yadav, Suniti. (2018). Linear regression analysis study. Journal of the Practice of Cardiovascular Sciences. 4. 33. 10.4103/jpcs.jpcs_8_18.

[6] Maindonald, John & Braun, W. & Andrews, Jeffrey. (2024). Multiple Linear Regression. 10.1017/9781009282284.004.

[7] Maulud, Dastan & Abdulazeez, Adnan. (2020). A Review on Linear Regression Comprehensive in Machine Learning. Journal of Applied Science and Technology Trends. 1. 140-147. 10.38094/jastt1457.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Januari 2025

[illegible]

Steven Owen Liauw - 13523103