Building Multi-Linear Regression from Scratch

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler
```

Importing the libraries

```
# Load dataset
data = fetch_california_housing()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['Target'] = data.target
print(df)
# Split dataset
x = df.drop('Target', axis=1)
y = df['Target']
```

```
       MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  Longitude  Target
0      8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88    -122.23   4.526
1      8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86    -122.22   3.585
2      7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85    -122.24   3.521
3      5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85    -122.25   3.413
4      3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85    -122.25   3.422
...       ...       ...       ...        ...         ...       ...       ...        ...     ...
20635  1.5603      25.0  5.045455   1.133333       845.0  2.560606     39.48    -121.09   0.781
20636  2.5568      18.0  6.114035   1.315789       356.0  3.122807     39.49    -121.21   0.771
20637  1.7000      17.0  5.205543   1.120092      1007.0  2.325635     39.43    -121.22   0.923
20638  1.8672      18.0  5.329513   1.171920       741.0  2.123209     39.43    -121.32   0.847
20639  2.3886      16.0  5.254717   1.162264      1387.0  2.616981     39.37    -121.24   0.894

[20640 rows x 9 columns]
```

20640 Samples

8 Features,
x_size: 20640 x 8

Target,
y_size: 20640 x 1

```
train_split = 0.8
train_split_index = int(train_split * len(x))

x_train = x.iloc[:train_split_index]
x_test = x.iloc[train_split_index:]
y_train = y.iloc[:train_split_index]
y_test = y.iloc[train_split_index:]

# Scale data
scaler = StandardScaler()
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)
print(' x_train_shape:',x_train.shape,'\n','x_test_shape:',x_test.shape,'\n','y_train_shape:',y_train.shape,
      '\n','y_test_shape:',y_test.shape)
```

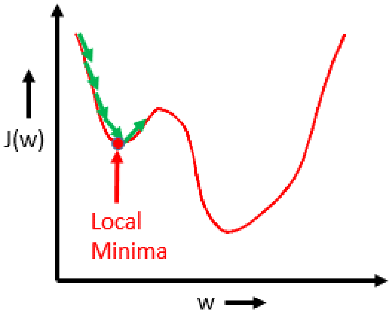
Note: Without standardization, the gradient descent updates can become unstable, leading to exploding or vanishing gradients. This happens because features with large magnitudes dominate the weight updates, causing inefficient learning and convergence issues.

```
 x_train_shape: (16512, 8)
 x_test_shape: (4128, 8)
 y_train_shape: (16512,)
 y_test_shape: (4128,)
```

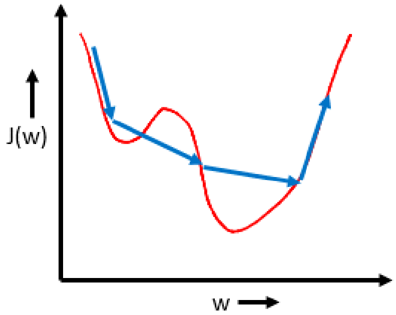
x_train_size: (0.8 x 20640)x 8
x_train_size: 16512 x 8
x_test_size: (0.2 x 20640)x 8
x_test_size: 4128x 8

Since y represents the target variable, it has a single column
y_train_size: 16512 x 1
y_test_size: 4128x 1

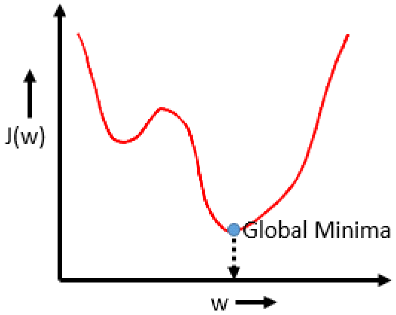
Selecting Learning Rates



It represents a scenario where the learning rate is low, allowing the optimization process to carefully explore the loss landscape and potentially converge to a local minimum.



It shows a higher learning rate, which can cause the optimizer to take large steps, potentially skipping over global minima and moving toward another region.



Adjusting the learning rate properly ensures the optimizer reaches the weight corresponding to the minimum cost function (global minimum)

```python
class Multi_Linear_Regression:
    def __init__(self, lr=0.01, n_iter=1000):
        self.lr = lr
        self.n_iter = n_iter

    def fit(self, X_train, y_train):
        n_samples, n_features = X_train.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iter):
            y_pred = np.dot(X_train, self.weights) + self.bias
            dw = (1/n_samples) * np.dot(X_train.T, (y_pred - y_train))
            db = (1/n_samples) * np.sum(y_pred - y_train)
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X_test):
        return np.dot(X_test, self.weights) + self.bias

    def evaluate(self, X_test, y_test):
        y_pred = self.predict(X_test)
        r_square = 1 - (np.sum((y_test - y_pred) ** 2) / np.sum((y_test - np.mean(y_test)) ** 2))
        print(f"R² Score: {r_square:.4f}")
        return r_square
```

Prediction Equation: $\hat{y} = X_{\text{train}} \cdot w + b$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{16512} \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,8} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ x_{16512,1} & x_{16512,2} & \cdots & x_{16512,8} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_8 \end{bmatrix} + b$$

w and b are initialized as 0, and is updated using gradient descent.

Gradient Descent Rule:

$$w = w - lr \cdot dw$$

$$b = b - lr \cdot db$$

Cost function (Mean Square Error):

$$J(w,b) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Substituting $\hat{y}$ into the Cost Function

$$J(w,b) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - (X_i \cdot w + b))^2$$

Differentiate J (w,b) w.r.t. w:

$$\frac{dJ}{dw} = \frac{1}{2n} \cdot 2\sum_{i=1}^{n}(y_i - (X_i \cdot w + b)) \cdot (-X_i)$$

$$\frac{dJ}{dw} = \frac{1}{n}\sum_{i=1}^{n} X_i(-(y_i - (X_i \cdot w + b)))$$

$$\frac{dJ}{dw} = \frac{1}{n}\sum_{i=1}^{n} X_i((X_i \cdot w + b) - y_i)$$

$$dw = \frac{1}{n}X^T(\hat{y} - y)$$

Differentiate J (w,b) w.r.t. b:

$$\frac{dJ}{db} = \frac{1}{2n} \cdot 2\sum_{i=1}^{n}(y_i - (X_i \cdot w + b)) \cdot (-1)$$

$$\frac{dJ}{db} = \frac{1}{n}\sum_{i=1}^{n}((X_i \cdot w + b) - y_i)$$

$$db = \frac{1}{n}\sum_{i=1}^{n}(\hat{y} - y)$$

Equation for R^2

$$R^2 = 1 - \frac{\sum(y_{\text{test}} - \hat{y})^2}{\sum(y_{\text{test}} - \bar{y})^2}$$

```python
# Train Multi-Linear Regression Model
model = Multi_Linear_Regression(lr=0.01, n_iter=1000)
model.fit(x_train_scaled, y_train.values)

# Evaluate the model
model.evaluate(x_test_scaled, y_test.values)
```

Multi-Linear Regression Model using gradient descent with a learning rate of 0.01 and 1000 iterations is initialized. After training on the scaled dataset, the model is evaluated on the test data to compute the R^2 score, measuring its performance.