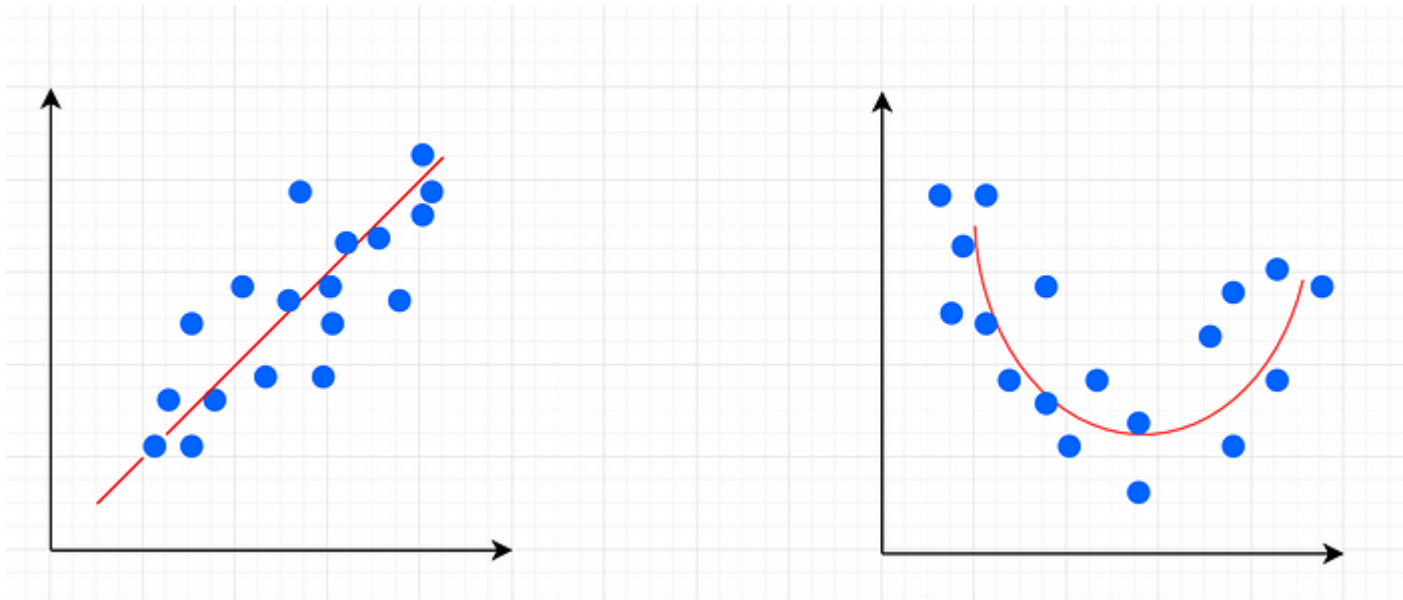


Simple Linear Regression

Shun-Chieh Hsieh, Professor & Head
Department of Land Management and Development
Chang Jung Christian University

Regression

Let's think there are some data points(x , y coordinates) in a given graph. You have to draw a line(the line can be straight or curved) that goes through all the data points or almost all the data points. So, you have to try to draw the best possible line. This technique is called regression.



Linear Model Assumptions

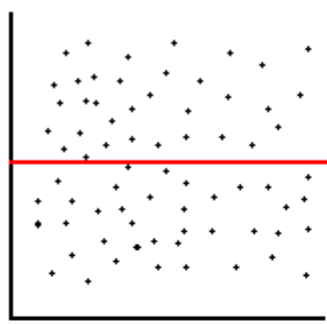
- * **Linearity:** The relationship between the independent variables and the dependent variable is linear. This means that a change in the independent variable has a constant effect on the dependent variable. You can assess this assumption by visually inspecting scatter plots and residual plots.
- * **Independence:** The observations in the data set must be independent of each other. This means that one observation should not be able to predict another observation. You can also formally test if this assumption is met using the Durbin-Watson test.
- * **No multicollinearity:** The independent variables should not be highly correlated with each other. Multicollinearity can occur when two or more independent variables are measuring the same thing. Multicollinearity can make it difficult to interpret the individual effects of predictors.
- * **Homoscedasticity:** The variance of the errors should be constant across all levels of the independent variables. This assumption implies that the spread of the residuals should be roughly the same throughout the range of independent variables.

Residual plot

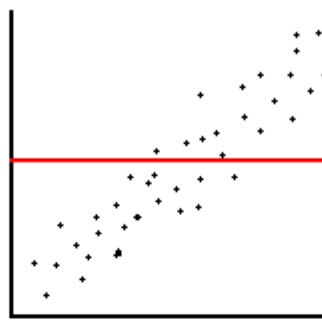
A residual plot plots the residuals on the y -axis vs. the predicted values of the dependent variable on the x -axis. We would like the residuals to be

Unbiased: have an average value of zero in any thin vertical strip, and

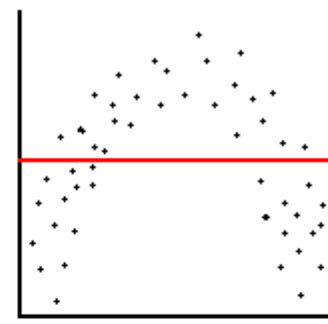
Homoscedastic, which means “same stretch”: the spread of the residuals should be the same in any thin vertical strip.



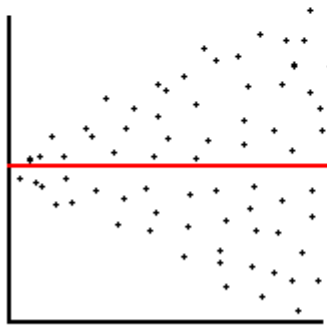
(a) Unbiased and Homoscedastic



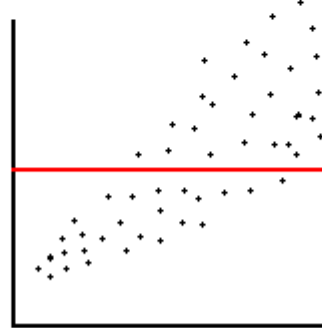
(b) Biased and Homoscedastic



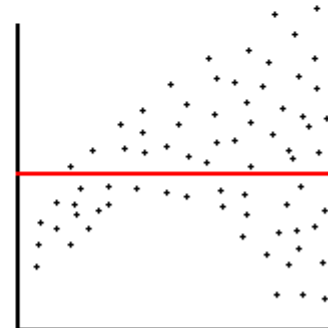
(c) Biased and Homoscedastic



(d) Unbiased and Heteroscedastic



(e) Biased and Heteroscedastic

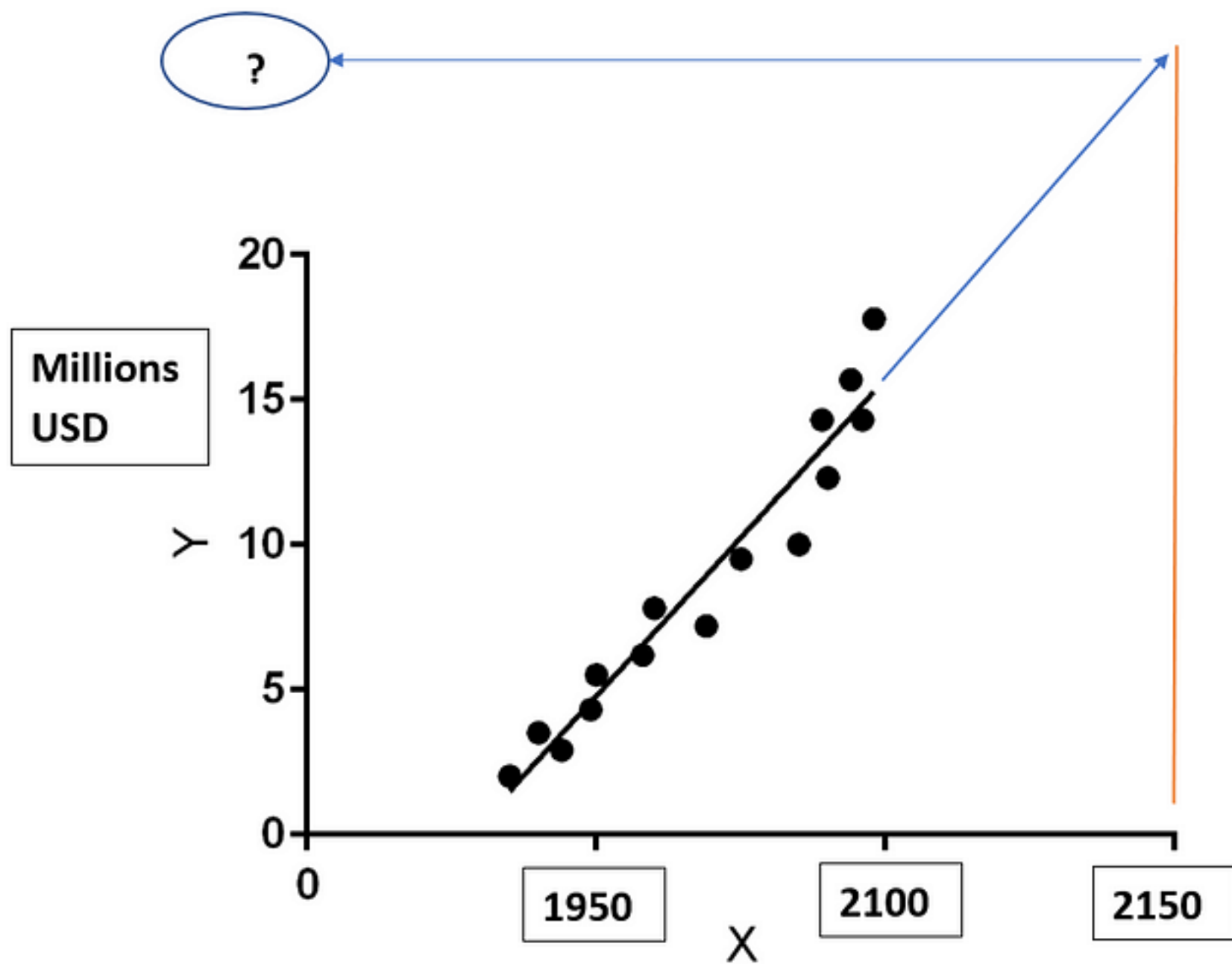


(f) Biased and Heteroscedastic

- (a) Unbiased and homoscedastic. The residuals average to zero in each thin vertical strip and the SD is the same all across the plot.
- (b) Biased and homoscedastic. The residuals show a linear pattern, probably due to a lurking variable not included in the experiment.
- (c) Biased and homoscedastic. The residuals show a quadratic pattern, possibly because of a nonlinear relationship. Sometimes a variable transform will eliminate the bias.
- (d) Unbiased, but heteroscedastic. The SD is small to the left of the plot and large to the right: the residuals are heteroscedastic.
- (e) Biased and heteroscedastic. The pattern is linear.
- (f) Biased and heteroscedastic. The pattern is quadratic.

Scenario

You have been observing the price of an antique painting. You have the price values for several previous years. Now, you want to estimate the price of the same antique painting after several years. What you can do is find the correlation between years and the price. That means you have to build a function that includes the relationship between years and prices. You can think of years as x and prices as y (the correlation between x and y can be linear or nonlinear (then the function would be a polynomial)).



Simple linear regression

Simple linear regression is a basic statistical technique used to model the relationship between two variables: one independent variable (often referred to as the “feature”) and one dependent variable (often referred to as the “target” or “response”).

In simple linear regression, the relationship between the independent variable (x) and the dependent variable (y) is assumed to be of the form:

$$y = b + wx$$

The intercept term (b) in a regression model accounts for the baseline value of the dependent variable when all independent variables are set to zero. Without an intercept, the model might not accurately represent the data. It's essential for a meaningful interpretation.

Cost Function

A cost function (also known as a loss function or objective function) is defined to measure the error between predicted values and actual values. In the context of linear regression, the most common cost function is the Mean Squared Error (MSE), which calculates the average squared difference between predicted and actual values.

```
# Cost function MSE
def cost_function(Y, b, w, X):
    m = len(Y)
    sse = 0

    for i in range(0, m):
        y_hat = b + w * X[i]
        y = Y[i]
        sse += (y_hat - y) ** 2

    mse = sse / m
    return mse
```

```
# Cost function MSE
def cost_function(Y, b, w, X):
    m = len(Y)
    sse = 0

    for i in range(0, m):
        y_hat = b + w * X[i]
        y = Y[i]
        sse += (y_hat - y) ** 2

    mse = sse / m
    return mse
```

Mean Squared Error (MSE)

It measures the average of the squared difference between the actual and predicted values.

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

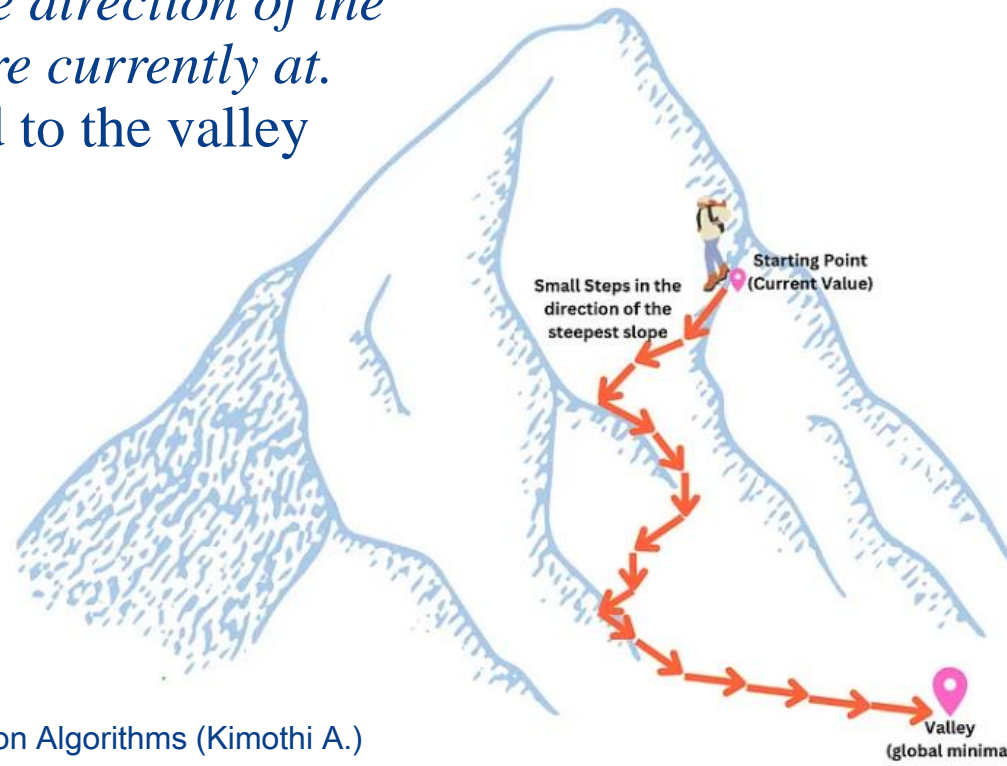
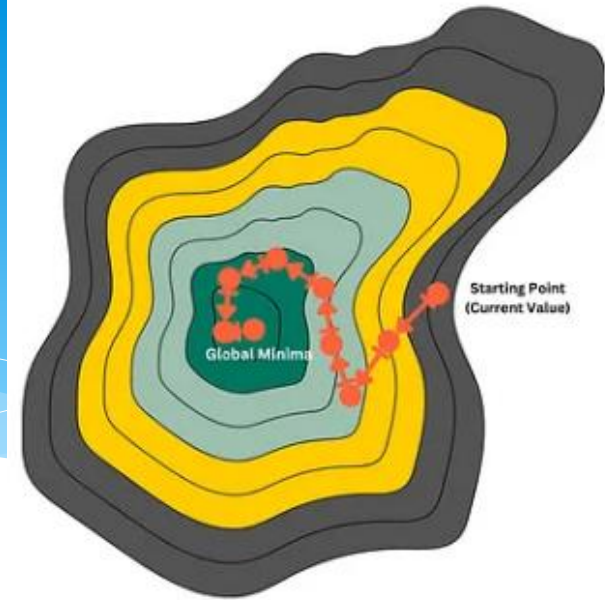
n = number of data points

y_i = actual value for the i th data point

\hat{y}_i = predicted value for the i th data point

Gradient Descent

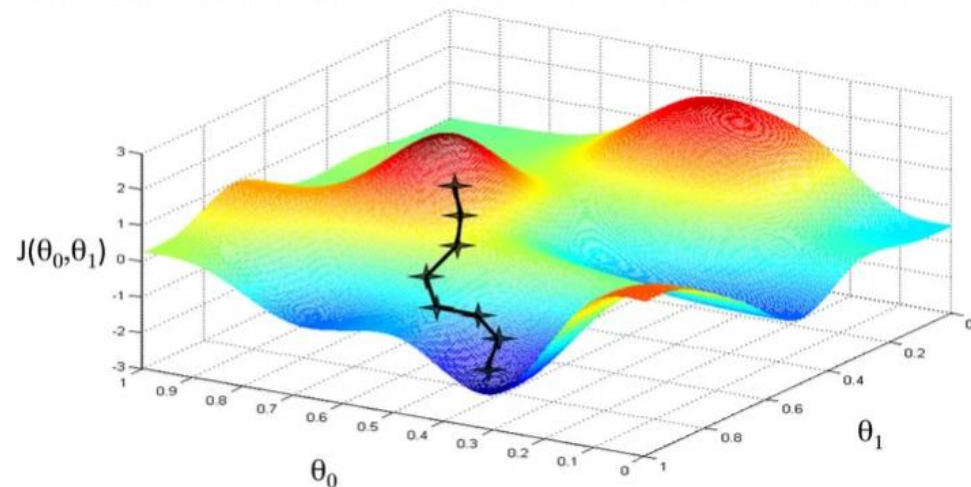
The basic intuition of gradient descent can be understood by imagining *a person stuck in a mountain on a foggy evening, trying to get down to the valley (the global minima)*. The visibility is extremely low and the path down the mountain is not visible. They must rely only on the very short distance that they can see. They can use the method of gradient descent by *moving in the direction of the steepest slope from the point they are currently at*. Taking multiple such steps will lead to the valley (lowest point).



Gradient Descent

Loss optimization techniques in the machine learning

- Gradient descent is an optimization technique used to find the optimal coefficients (slope, w , and y-intercept, b) of a linear regression model by iteratively adjusting them in the direction that reduces the cost function the most.
- Gradient Descent is an optimization algorithm that minimizes the error function by iteratively moving towards the minimum. It is the backbone of many ML algorithms, guiding them towards the best set of parameters.



The gradient of the cost function with respect to w :

$$\frac{\partial \text{MSE}}{\partial w} = \frac{2}{n} \sum_{i=1}^n -x_i (y_i - \hat{y}_i)$$

The gradient of the cost function with respect to b :

$$\frac{\partial \text{MSE}}{\partial b} = \frac{2}{n} \sum_{i=1}^n (y_i - \hat{y}_i)$$

Update rule:

$$w_{\text{new}} = w_{\text{old}} - \alpha \frac{\partial \text{MSE}}{\partial w}$$

$$b_{\text{new}} = b_{\text{old}} - \alpha \frac{\partial \text{MSE}}{\partial b}$$

α = learning rate, which determines the step size of each update

In a machine learning context, where l is the loss and w represents the model parameters, the update rule is given by:

$$w_{\text{new}} = w_{\text{old}} - \alpha \cdot \nabla l(w)$$

Here, α is the learning rate, and $\nabla l(w)$ is the gradient of the loss function l concerning the parameters.

```
import numpy as np

# Define the dataset
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

# Initialize parameters
alpha = 0.01 # Learning rate
epochs = 1000 # Number of iterations
w = 0 # Model parameter

# Perform Gradient Descent
for epoch in range(epochs):
    y_pred = w * X
    gradient = (-2/len(X)) * sum(X * (y - y_pred))
    w = w - alpha * gradient

print("Optimal parameter is: w =", w)
```

Optimal parameter is: w = 1.1999999999999995

```
import numpy as np

# Define the dataset
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

# Initialize parameters
alpha = 0.01 # Learning rate
epochs = 1000 # Number of iterations
w = 0 # Model parameter

# Perform Gradient Descent
for epoch in range(epochs):
    y_pred = w * X
    gradient = (-2/len(X)) * sum(X * (y - y_pred))
    w = w - alpha * gradient

print("Optimal parameter is: w =", w)
```

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Define the dataset
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 4, 5, 4, 5])

# Create a linear regression model
model = LinearRegression(fit_intercept = False)

# Fit the model to the data
model.fit(X, y)

# Get the slope (m) and y-intercept (b)
slope = model.coef_[0]
intercept = model.intercept_

# Print the regression line equation
print(f"Regression Line Equation: y = {slope:.2f}x + {intercept:.2f}")

# Plot the dataset and regression line
plt.scatter(X, y, color='blue', label='Data points')
plt.plot(X, model.predict(X), color='red', label='Regression line')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression')
plt.legend()
plt.show()
```



Regression Line Equation: $y = 1.20x + 0.00$

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Define the dataset
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 4, 5, 4, 5])

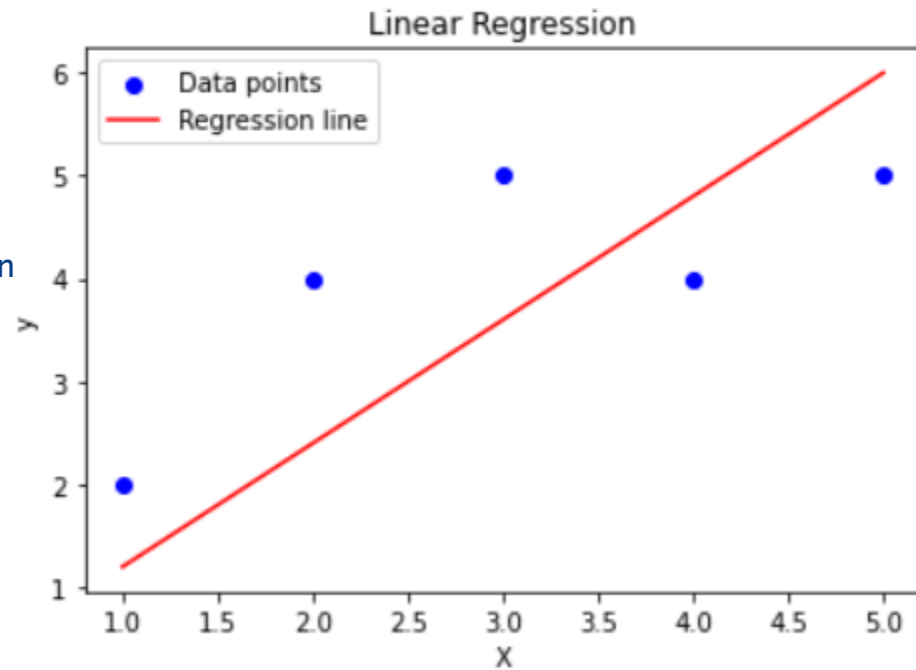
# Create a linear regression model
model = LinearRegression(fit_intercept = False)

# Fit the model to the data
model.fit(X, y)

# Get the slope (m) and y-intercept (b)
slope = model.coef_[0]
intercept = model.intercept_

# Print the regression line equation
print(f"Regression Line Equation:  $y = {slope:.2f}x + {intercept:.2f}$ ")

# Plot the dataset and regression line
plt.scatter(X, y, color='blue', label='Data points')
plt.plot(X, model.predict(X), color='red', label='Regression line')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression')
plt.legend()
plt.show()
```



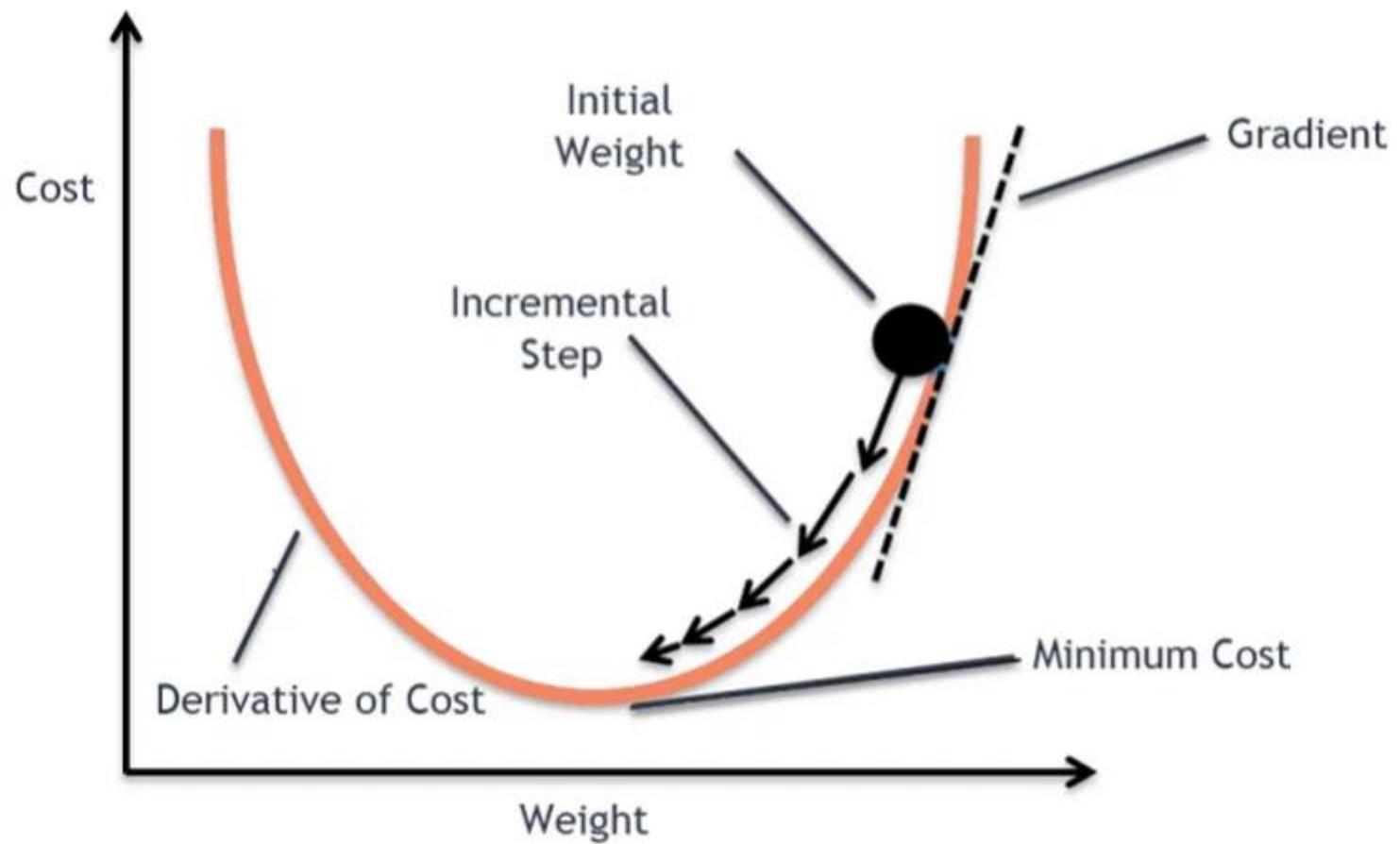
Linear regression with gradient descent

1. **Initial Coefficients:** Gradient descent starts with initial coefficients for the linear regression equation.
2. **Gradient Calculation:** The gradient of the cost function with respect to each coefficient is calculated. The gradient represents the direction and magnitude of the steepest increase in the cost function. It indicates how much the cost function would increase if a coefficient is increased or decreased slightly.
3. **Coefficient Update:** The coefficients are updated iteratively by subtracting the gradient of the cost function from the current coefficients. The learning rate, a hyperparameter, determines the step size in the direction of the gradient. The coefficients are adjusted in a way that reduces the cost function, leading to better predictions.
4. **Iteration:** Steps 2 and 3 are repeated iteratively until the algorithm converges to a point where further iterations do not significantly reduce the cost function or until a predetermined number of iterations is reached.

```
# Update weights
def update_weights(Y, b, w, X, learning_rate):
    m = len(Y)
    b_deriv_sum = 0
    w_deriv_sum = 0
    for i in range(0, m):
        y_hat = b + w * X[i]
        y = Y[i]
        b_deriv_sum += y_hat - y
        w_deriv_sum += (y_hat - y) * X[i]
    new_b = b - (learning_rate * 1 / m * b_deriv_sum)
    new_w = w - (learning_rate * 1 / m * w_deriv_sum)
    return new_b, new_w
```



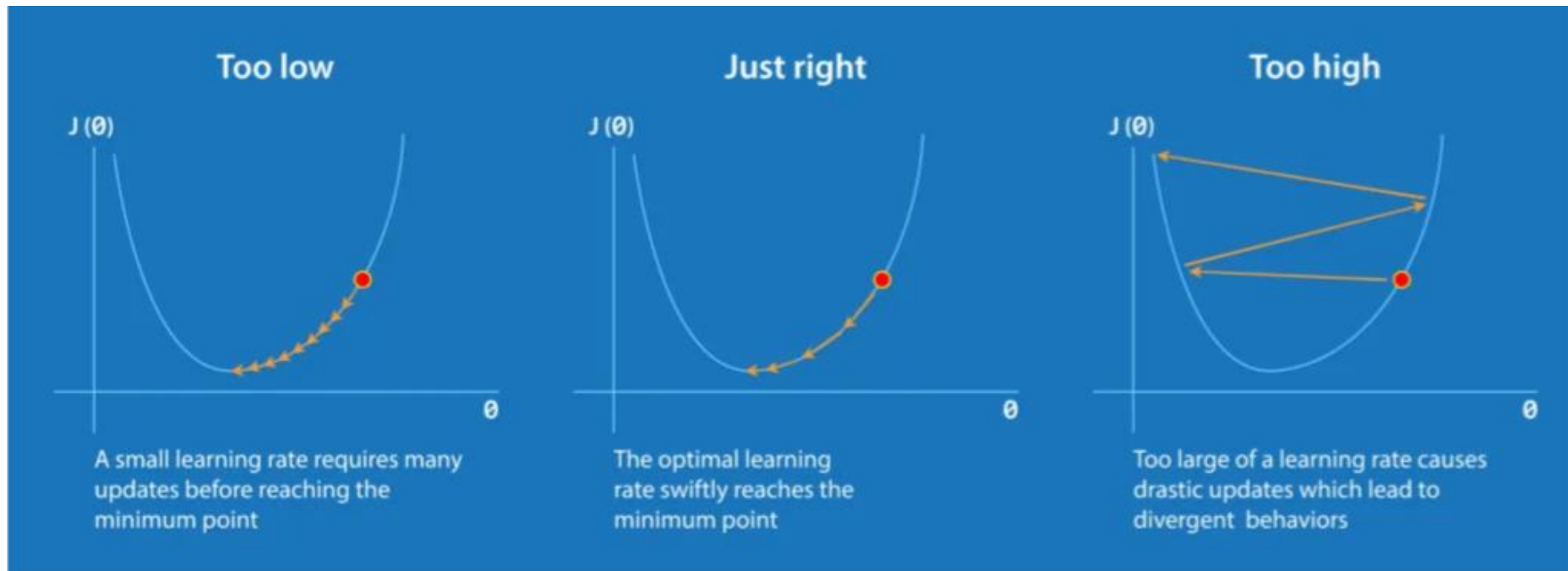
```
# Update weights
def update_weights(Y, b, w, X, learning_rate):
    m = len(Y)
    b_deriv_sum = 0
    w_deriv_sum = 0
    for i in range(0, m):
        y_hat = b + w * X[i]
        y = Y[i]
        b_deriv_sum += y_hat - y
        w_deriv_sum += (y_hat - y) * X[i]
    new_b = b - (learning_rate * 1 / m * b_deriv_sum)
    new_w = w - (learning_rate * 1 / m * w_deriv_sum)
    return new_b, new_w
```



Gradient descent gradually refines the coefficients to minimize the prediction error.

Learning Rate

The learning rate determines the step size at each iteration while moving toward a minimum of the loss function.



It's important to choose an appropriate learning rate to ensure convergence and avoid overshooting or slow convergence. If the learning rate is too high, the algorithm might not converge; if it's too low, convergence might be slow.

```

# Train function
def train(Y, initial_b, initial_w, X, learning_rate, num_iters, print_iter=False):
    print(
        "Starting gradient descent at b = {0}, w = {1}, train mse = {2}".format(
            initial_b, initial_w, cost_function(Y, initial_b, initial_w, X)
        )
    )

    b = initial_b
    w = initial_w
    cost_history = []

    for i in range(num_iters):
        b, w = update_weights(Y, b, w, X, learning_rate)
        mse = cost_function(Y, b, w, X)
        cost_history.append(mse)

        if (i % 100 == 0) & (print_iter==True):
            print("iter={:d}    b={:.2f}    w={:.4f}    train mse={:.4}".format(i, b, w, mse))

    print(
        "After {0} iterations b = {1}, w = {2}, train mse = {3}".format(
            num_iters, b, w, cost_function(Y, b, w, X)
        )
    )
    return cost_history, b, w

```

```
# Train function
def train(Y, initial_b, initial_w, X, learning_rate, num_iters, print_iter=False):
    print(
        "Starting gradient descent at b = {0}, w = {1}, train mse = {2}".format(
            initial_b, initial_w, cost_function(Y, initial_b, initial_w, X)
        )
    )

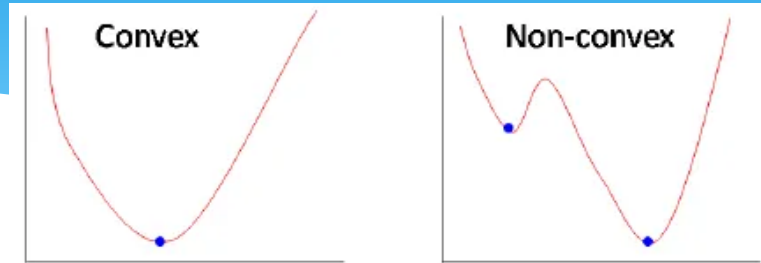
    b = initial_b
    w = initial_w
    cost_history = []

    for i in range(num_iters):
        b, w = update_weights(Y, b, w, X, learning_rate)
        mse = cost_function(Y, b, w, X)
        cost_history.append(mse)

        if (i % 100 == 0) & (print_iter==True):
            print("iter={:d}    b={:.2f}    w={:.4f}    train mse={:.4}".format(i, b, w, mse))

    print(
        "After {0} iterations b = {1}, w = {2}, train mse = {3}".format(
            num_iters, b, w, cost_function(Y, b, w, X)
        )
    )
    return cost_history, b, w
```

Convex vs. Non-Convex Cost Functions



- * The convexity of the cost function is vital because it guarantees that gradient-based optimization techniques, such as gradient descent, will converge to the global minimum. In simpler terms, it ensures that the algorithm finds the best set of parameters for the model.
- * When a cost function is non-convex, gradient-based optimization techniques might get stuck in local minima, preventing the model from reaching the best possible parameters. Researchers and practitioners often explore various strategies like using different optimization algorithms or initializing parameters differently to mitigate the problem.

A linear regression model assumes “a linear relationship between the input variables and the single output variable.” What’s the meaning of this assumption?

- (A) The output variable can’t be calculated from a linear combination of the input variables
- (B) The output variable can be calculated from a linear combination of the input variables
- (C) Input variables can be calculated from a linear combination of the output variables
- (D) Output variable = sum of the input variables

In a simple linear regression problem, a single input variable (x) and a single output variable (y), the linear equation would be $y = ax + b$; where a and b are _____ and _____ respectively. (select two)

- (A) bias coefficient, feature coefficient
- (B) feature coefficient, bias coefficient
- (C) slope, y-intercept
- (D) y-intercept, slope

For a regression line through the data, the vertical distance from each data point to the regression line is called residual. (i) Square the residual, and (ii) sum all of the squared errors together. This is the quantity that ordinary least squares seek to _____?

- (A) minimize
- (B) maximize
- (C) increase
- (D) None of these

For a linear regression model, start with random values for each coefficient. The sum of the squared errors is calculated for each pair of input and output values. A learning rate is used as a scale factor and the coefficients are updated in the direction of minimizing the error. The process is repeated until a minimum sum squared error is achieved or no further improvement is possible. This technique is called _____?

- (A) Gradient Descent
- (B) Ordinary Least Squares
- (C) Homoscedasticity
- (D) Regularization

Which parameter determines the size of the improvement step to take on each iteration of Gradient Descent?

- (A) learning rate
- (B) epoch
- (C) batch size
- (D) regularization parameter

One of the major assumptions of linear regression: when the variance around the regression line is the same for all values of the predictor variable is called _____?

- (A) L1 regularization
- (B) Lasso Regression
- (C) Homoscedasticity
- (D) Heteroscedasticity

For a Linear Regression model, we choose the coefficients and the bias term by minimizing the _____.

- (A) Loss function
- (B) Error function
- (C) Cost function
- (D) All of the above

Which one is the correct Linear regression assumption?

- (A) Linear regression assumes the input and output variables are not noisy
- (B) Linear regression will over-fit your data when you have highly correlated input variables
- (C) The residuals (true target value – predicted target value) of the data are normally distributed and independent from each other
- (D) All of the above

In a linear regression model, which technique can find the coefficients?

- (A) Ordinary Least Squares
- (B) Gradient Descent
- (C) Regularization
- (D) All of the above

Which one is the disadvantage of Linear Regression?

- (A) The assumption of linearity between the dependent variable and the independent variables. In the real world, the data is not always linearly separable
- (B) Linear regression is very sensitive to outliers
- (C) Before applying Linear regression, multicollinearity should be removed because it assumes that there is no relationship among independent variables.
- (D) All of the above

Boston dataset

The dataset includes 506 instances with 13 features each.

- * CRIM: Per capita crime rate by town.
- * ZN: Proportion of residential land zoned for lots over 25,000 sq. ft.
- * INDUS: Proportion of non-retail business acres per town.
- * CHAS: Charles River dummy variable (1 if tract bounds river; 0 otherwise).
- * NOX: Nitric oxides concentration (parts per 10 million).
- * RM: Average number of rooms per dwelling.
- * AGE: Proportion of owner-occupied units built before 1940.
- * DIS: Weighted distances to five Boston employment centers.
- * RAD: Index of accessibility to radial highways.
- * TAX: Full-value property tax rate per \$10,000.
- * PTRATIO: Pupil-teacher ratio by town.
- * B: Proportion of Black residents by town.
- * LSTAT: Percentage of lower status of the population.
- * The target variable, MEDV, is the median value of owner-occupied homes in \$1000s.

```
import pandas as pd
Location = "C:/Users/user/desktop/boston.csv"
data = pd.read_csv(Location)
data.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

```
import pandas as pd
Location = "C:/Users/user/desktop/boston.csv"
data = pd.read_csv(Location)
data.head()
```

Prepare training data

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+",
                      skiprows=22, header=None)
x_org = np.hstack([raw_df.values[::2, :],
                    raw_df.values[1::2, :2]])
yt = raw_df.values[1::2, 2]
feature_names = np.array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX',
                           'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT'])

print('Original data', x_org.shape, yt.shape)
print('Feature names: ', feature_names)
# Takes one feature RM
x_data = x_org[:, feature_names == 'RM']
print('After scaling down', x_data.shape)
# Add the dummy variable value 1 to the first position of the x array
x = np.insert(x_data, 0, 1.0, axis=1)
print('After adding dummy variables', x.shape)
```

Original data (506, 13) (506,)

Feature names: ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO' 'B' 'LSTAT']

After scaling down (506, 1)

After adding dummy variables (506, 2)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+",
                      skiprows=22, header=None)
x_orig = np.hstack([raw_df.values[::2, :],
                    raw_df.values[1::2, :2]])
yt = raw_df.values[1::2, 2]
feature_names = np.array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX',
                           'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT'])

print('Original data', x_orig.shape, yt.shape)
print('Feature names: ', feature_names)
# Takes one feature RM
x_data = x_orig[:, feature_names == 'RM']
print('After scaling down', x_data.shape)
# Add the dummy variable value 1 to the first position of the x array
x = np.insert(x_data, 0, 1.0, axis=1)
print('After adding dummy variables', x.shape)
```

```
print(x.shape)  
print(x[:5,:])
```

```
(506, 2)  
[[1.    6.575]  
 [1.    6.421]  
 [1.    7.185]  
 [1.    6.998]  
 [1.    7.147]]
```

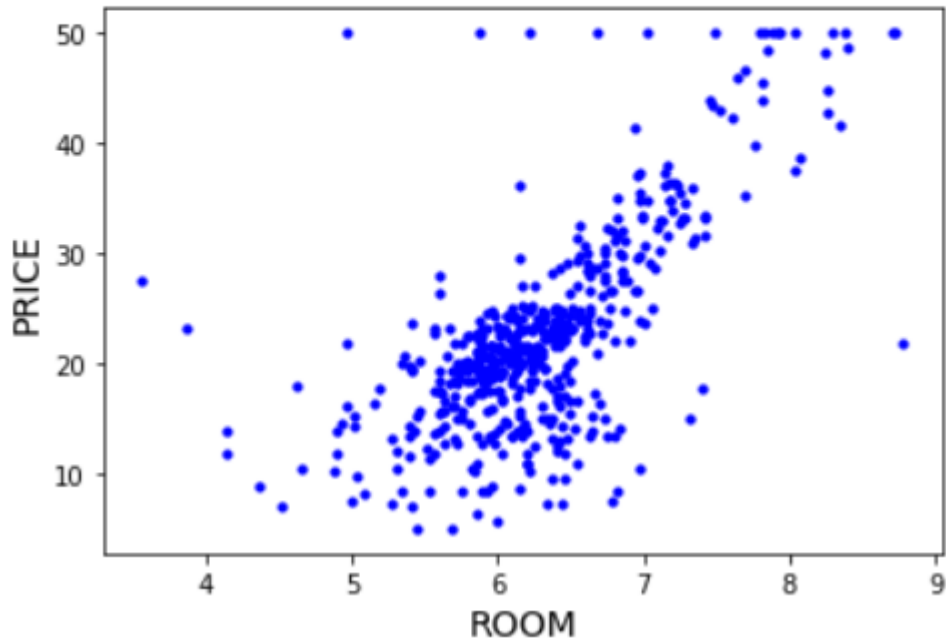
```
print(yt[:5])
```

```
[24.  21.6 34.7 33.4 36.2]
```

```
print(x.shape)  
print(x[:5,:])
```

```
print(yt[:5])
```

```
plt.scatter(x[:,1], yt, s=10, c='b')  
plt.xlabel('ROOM', fontsize=14)  
plt.ylabel('PRICE', fontsize=14)  
plt.show()
```



```
plt.scatter(x[:,1], yt, s=10, c='b')  
plt.xlabel('ROOM', fontsize=14)  
plt.ylabel('PRICE', fontsize=14)  
plt.show()
```

Define house price prediction function

```
def pred(x, w):  
    return(x @ w) # @ represents vector inner product  
# Initialization settings for gradient descent method  
M = x.shape[0] # The total number of data samples is 506  
D = x.shape[1] # The dimension of the input data is 2  
iters = 50000 # The number of iteration operations is 50000  
alpha = 0.01 # Learning rate  
# Set the initial value of the weight vector to 1  
w = np.ones(D)  
# Record the evaluation results (only the loss function value is recorded)  
history = np.zeros((0,2))
```

```
def pred(x, w):  
    return(x @ w) # @ represents vector inner product  
# Initialization settings for gradient descent method  
M = x.shape[0] # The total number of data samples is 506  
D = x.shape[1] # The dimension of the input data is 2  
iters = 50000 # The number of iteration operations is 50000  
alpha = 0.01 # Learning rate  
# Set the initial value of the weight vector to 1  
w = np.ones(D)  
# Record the evaluation results (only the loss function value is recorded)  
history = np.zeros((0,2))
```



```

for k in range(iters):
    yp = pred(x, w)
    yd = yp - yt
    w = w - alpha * (x.T @ yd) / M
    if (k % 100 == 0):
        loss = np.mean(yd ** 2) / 2
        history = np.vstack((history, np.array([k, loss])))
        print("iter = %d  loss = %f" % (k, loss))
print('Initial value of loss function: %f' % history[0,1])
print('Final value of loss function: %f' % history[-1,1])

```

```

iter = 0  loss = 154.224934
iter = 100  loss = 29.617518
iter = 200  loss = 29.431766
iter = 300  loss = 29.250428
iter = 400  loss = 29.073399
iter = 500  loss = 28.900577

```

```

iter = 49500  loss = 21.800330
iter = 49600  loss = 21.800329
iter = 49700  loss = 21.800327
iter = 49800  loss = 21.800326
iter = 49900  loss = 21.800325
Initial value of loss function: 154.224934
Final value of loss function: 21.800325

```

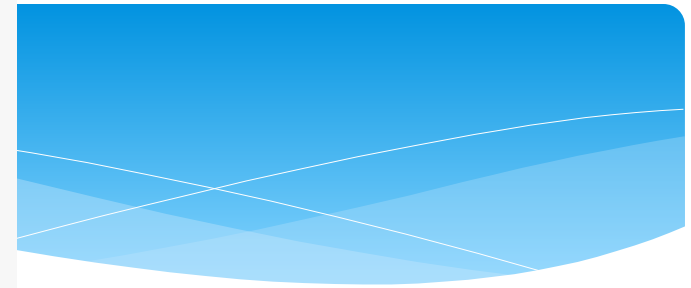
The loss function is the sum of squared errors divided by the total number of data (that is, the average) and then divided by 2 (to offset the differential, a coefficient of 2 will be generated)

```

for k in range(iters):
    yp = pred(x, w)
    yd = yp - yt
    w = w - alpha * (x.T @ yd) / M
    if (k % 100 == 0):
        loss = np.mean(yd ** 2) / 2
        history = np.vstack((history, np.array([k, loss])))
        print("iter = %d  loss = %f" % (k, loss))
print('Initial value of loss function: %f' % history[0,1])
print('Final value of loss function: %f' % history[-1,1])

```

The loss function value drops from 154.22 at the beginning to 21.80 (minimum value) at the end of the loop iteration calculation, which is the lowest point on the graph.



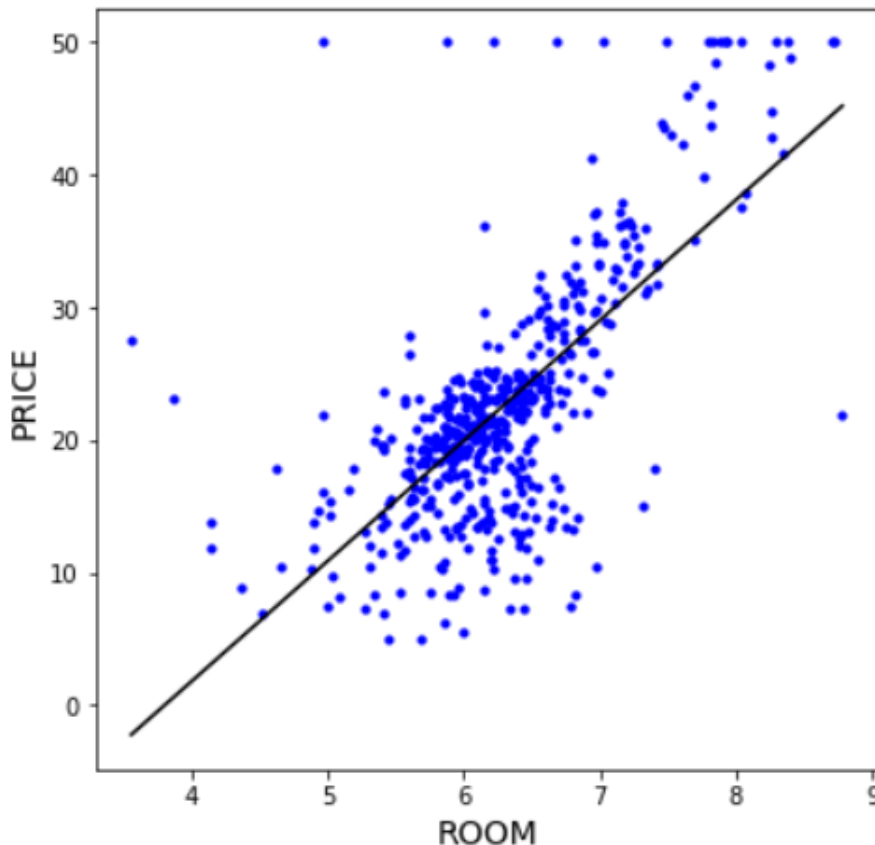
After obtaining the optimal parameter (weight) w value, the predicted value is then calculated to draw the regression line.

```
xall = x[:,1].ravel()
xl = np.array([[1, xall.min()], [1, xall.max()]])
yl = pred(xl, w)
plt.figure(figsize=(6,6))
plt.scatter(x[:,1], yt, s=10, c='b')
plt.xlabel('ROOM', fontsize=14)
plt.ylabel('PRICE', fontsize=14)
plt.plot(xl[:,1], yl, c='k')
plt.show()
```

```
print(w)
```

```
[-34.58233438  9.08822991]
```

$$\text{PRICE} = -34.58 + 9.09 \cdot \text{RM}$$

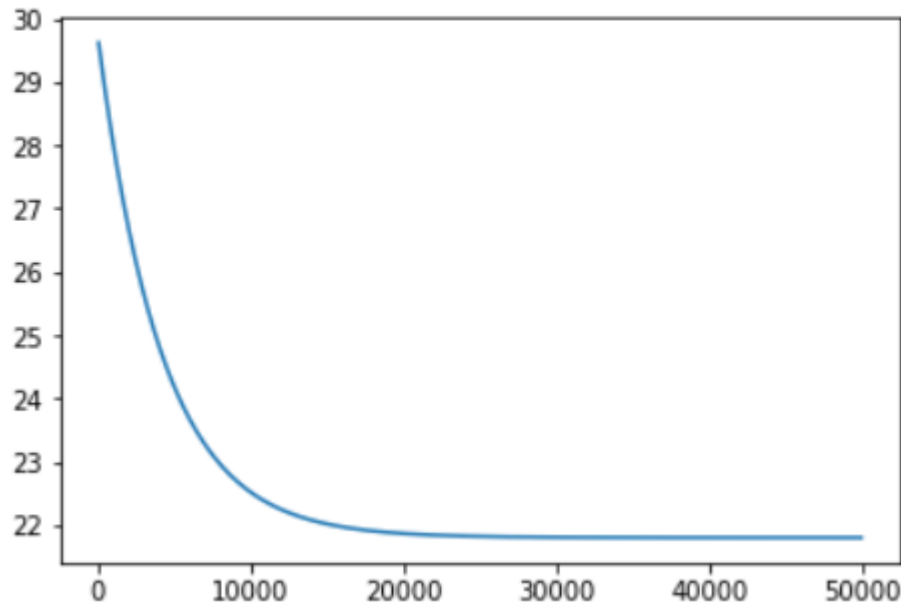


Use the min and max functions to find the minimum and maximum values of the input data x , respectively, and obtain the corresponding house price prediction values. Connect the two points to get the regression line that is most suitable for describing the input data.

```
xall = x[:,1].ravel()
xl = np.array([[1, xall.min()], [1, xall.max()]])
yl = pred(xl, w)
plt.figure(figsize=(6,6))
plt.scatter(x[:,1], yt, s=10, c='b')
plt.xlabel('ROOM', fontsize=14)
plt.ylabel('PRICE', fontsize=14)
plt.plot(xl[:,1], yl, c='k')
plt.show()
```

Learning curve

```
plt.plot(history[1:,0], history[1:,1])  
plt.show()
```

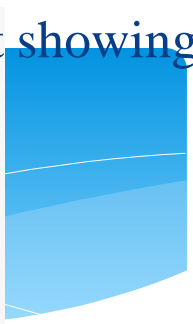


```
plt.plot(history[1:,0], history[1:,1])  
plt.show()
```

The horizontal axis of the learning curve is the number of iteration operations, and the vertical axis is the loss function value.

The learning curve is also called the history curve, which is the learning history of the model. The figure shows the convergence speed of the model.

Directly fitting LinearRegression model without showing the gradient descent method



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+",
                     skiprows=22, header=None)
x_org = np.hstack([raw_df.values[::2, :],
                   raw_df.values[1::2, :2]])
y = raw_df.values[1::2, 2]
feature_names = np.array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX',
                          'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT'])
# Takes one feature RM
X = x_org[:, feature_names == 'RM']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and fit the Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Calculate Mean Squared Error and R-squared
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared:", r2)
```

Mean Squared Error: 46.144775347317264
R-squared: 0.3707569232254778

37% of the variation in house prices can be explained by the number of rooms .

```
# print coefficients  
print('intercept', model.intercept_)  
print('slope', model.coef_)
```

```
intercept -36.24631889813792  
slope [9.34830141]
```

```
# print coefficients  
print('intercept', model.intercept_)  
print('slope', model.coef_)
```


```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+",
                     skiprows=22, header=None)
x_org = np.hstack([raw_df.values[::2, :],
                   raw_df.values[1::2, :2]])
y = raw_df.values[1::2, 2]
feature_names = np.array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX',
                          'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT'])
# Takes one feature RM
X = x_org[:, feature_names == 'RM']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and fit the Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Calculate Mean Squared Error and R-squared
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared:", r2)
```



```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline


# Create Polynomial Regression model
degree = 2 # Set the degree of the polynomial
poly_model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
poly_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_poly = poly_model.predict(X_test)

# Calculate Mean Squared Error and R-squared for Polynomial Regression
mse_poly = mean_squared_error(y_test, y_pred_poly)
r2_poly = r2_score(y_test, y_pred_poly)

print("Polynomial Regression (Degree {}):".format(degree))
print("Mean Squared Error:", mse_poly)
print("R-squared:", r2_poly)
```

```
Polynomial Regression (Degree 2):
Mean Squared Error: 35.36977373731789
R-squared: 0.5176878620868068
```



```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

# Create Polynomial Regression model
degree = 2 # Set the degree of the polynomial
poly_model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
poly_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_poly = poly_model.predict(X_test)

# Calculate Mean Squared Error and R-squared for Polynomial Regression
mse_poly = mean_squared_error(y_test, y_pred_poly)
r2_poly = r2_score(y_test, y_pred_poly)

print("Polynomial Regression (Degree {}):".format(degree))
print("Mean Squared Error:", mse_poly)
print("R-squared:", r2_poly)
```


advertising dataset

The dataset shows the sales amounts of a product (in thousands of units) and the budget allocated (in thousands of dollars) to TV, radio and newspaper for the advertisement of that product.

```
import pandas as pd
df = pd.read_csv("C:/Users/user/desktop/advertising.csv")
df.head()
```

	TV	radio	newspaper	sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	9.3
3	151.5	41.3	58.5	18.5
4	180.8	10.8	58.4	12.9

```
import pandas as pd
df = pd.read_csv("C:/Users/user/desktop/advertising.csv")
df.head()
```

```

import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, cross_val_score
X = df["TV"]
y = df["sales"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=1)
X_train = X_train.reset_index().iloc[:,1:].squeeze()
y_train = y_train.reset_index().iloc[:,1:].squeeze()
X_test = X_test.reset_index().iloc[:,1:].squeeze()
y_test = y_test.reset_index().iloc[:,1:].squeeze()
# Hyperparameters(you can experimentally change these values)
learning_rate = 0.00007
initial_b = 6.8
initial_w = 0.0493
num_iters = 10000
cost_history, b, w = train(y_train, initial_b, initial_w, X_train, learning_rate, num_iters)

```

Starting gradient descent at $b = 6.8$, $w = 0.0493$, train mse = 10.454355566898752

After 10000 iterations $b = 6.799960376189568$, $w = 0.04927414167961735$, train mse = 10.454336626738776

```

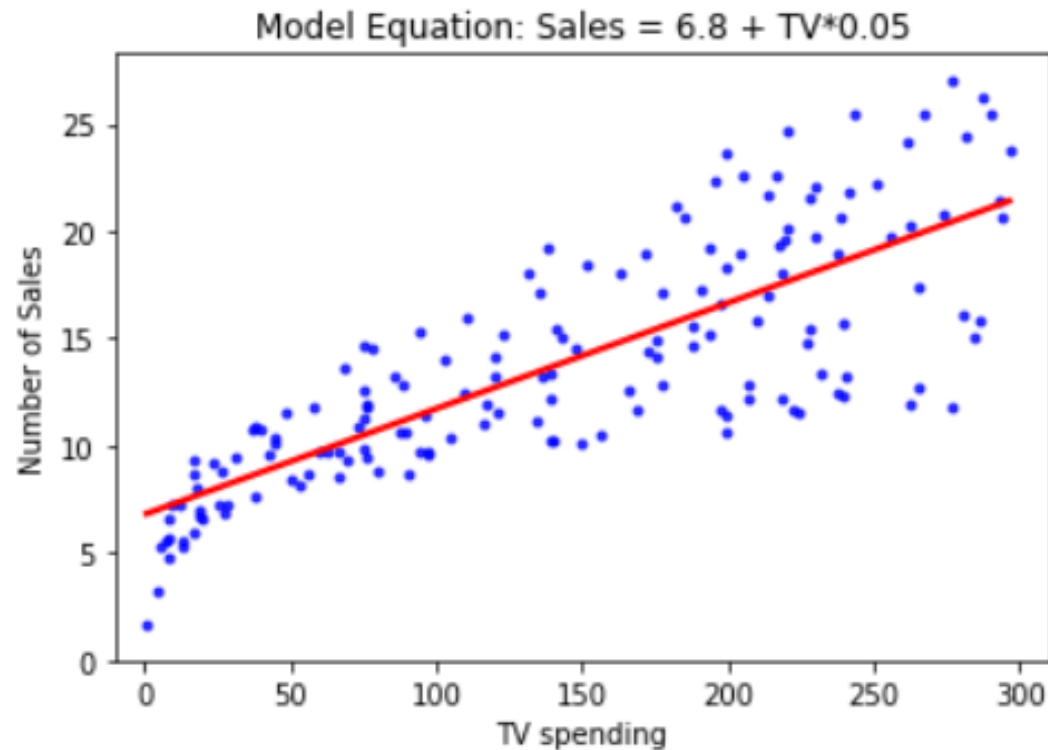
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, cross_val_score
X = df["TV"]
y = df["sales"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=1)
X_train = X_train.reset_index().iloc[:,1:].squeeze()
y_train = y_train.reset_index().iloc[:,1:].squeeze()
X_test = X_test.reset_index().iloc[:,1:].squeeze()
y_test = y_test.reset_index().iloc[:,1:].squeeze()
# Hyperparameters(you can experimentally change these values)
learning_rate = 0.00007
initial_b = 6.8
initial_w = 0.0493
num_iters = 10000
cost_history, b, w = train(y_train, initial_b, initial_w, X_train, learning_rate, num_iters)

```

```

import matplotlib.pyplot as plt
import seaborn as sns
# Model Visualization
g = sns.regplot(x=X_train, y=y_train, scatter_kws={"color": "b", "s": 9}, ci=False, color="r")
g.set_title(f"Model Equation: Sales = {round(b, 2)} + TV*{round(w, 2)}")
g.set_ylabel("Number of Sales")
g.set_xlabel("TV spending")
plt.xlim(-10, 310)
plt.ylim(bottom=0)
plt.show()

```



```

import matplotlib.pyplot as plt
import seaborn as sns
# Model visualization
g = sns.regplot(x=X_train, y=y_train, scatter_kws={"color": "b", "s": 9}, ci=False, color="r")
g.set_title(f"Model Equation: Sales = {round(b, 2)} + TV*{round(w, 2)}")
g.set_ylabel("Number of Sales")
g.set_xlabel("TV spending")
plt.xlim(-10, 310)
plt.ylim(bottom=0)
plt.show()

```

```
class LinearRegressionGD:
```

```
    def __init__(self):
```

```
        """
```

```
        The LinearRegressionGD class constructor.
```

```
        """
```

```
        # initialize learning rate lr and number of iteration iters
```

```
        self.lr = None
```

```
        self.iters = None
```

```
        # initialize the weights matrix
```

```
        self.weights = None
```

```
        # bins specifies how many iterations an MSE value will be saved to mse_history.
```

```
        self.bins = None
```

```
        # mse_history records MSE values in bins intervals.
```

```
        self.mse_history = []
```

```
        # keeps how many independent variables there are.
```

```
        self.n_features = "You should use fit() function first!"
```

```
        # keeps the MSE value of the optimal model.
```

```
        self.mse = "You should use performance() function first!"
```

```
        # keeps the RMSE value of the optimal model.
```

```
        self.rmse = "You should use performance() function first!"
```

```
        # keeps the MAE value of the optimal model.
```

```
        self.mae = "You should use performance() function first!"
```

```
        # keeps the R-squared value of the optimal model.
```

```
        self.r2 = "You should use performance() function first!"
```

```
        # keeps the adjusted R-squared value of the optimal model.
```

```
        self.ar2 = "You should use performance() function first!"
```

```
        # keeps the SSE value of the optimal model.
```

```
        self.sse = "You should use performance() function first!"
```

```
        # keeps the SSR value of the optimal model.
```

```
        self.ssr = "You should use performance() function first!"
```

```
        # keeps the SST value of the optimal model.
```

```
        self.sst = "You should use performance() function first!"
```

```

def performance(self, y_predicted, y, verbose=True):
    """
    This function calculates performance metrics such as
    RMSE, MSE, MAE, SSR, SSE, SST, R-squared and Adj. R-squared.

    Args:
        y_predicted (numpy.ndarray): predicted y values
        y (numpy.ndarray): true y values
        verbose (bool, optional): prints performance metrics. Defaults to True.
    """
    self.mse = np.mean(np.sum((y_predicted - y) ** 2))
    self.rmse = np.sqrt(self.mse)
    self.mae = np.mean(np.abs(y - y_predicted))
    self.ssr = np.sum((y_predicted - np.mean(y)) ** 2)
    self.sst = np.sum((y - np.mean(y)) ** 2)
    self.sse = np.sum((y - y_predicted) ** 2)
    self.r2 = 1 - self.sse / self.sst
    self.ar2 = 1 - (((1 - self.r2) * (len(y) - 1)) / (len(y) - self.n_features - 1))
    if verbose:
        print(f"RMSE = {self.rmse}")
        print(f"MSE = {self.mse}")
        print(f"MAE = {self.mae}")
        print(f"SSE = {self.sse}")
        print(f"SSR = {self.ssr}")
        print(f"SST = {self.sst}")
        print(f"R-squared = {self.r2}")
        print(f"Adjusted R-squared = {self.ar2}")

```

```
def predict(self, X):  
    """
```

This function takes one argument which is a numpy.array of predictor values, and returns predicted y values.

Note: You should use fit() function at least once before using predict() function, since the prediction is made with the optimal weights obtained by the fit() function.

Args:

X (numpy.ndarray): predictors(input)

Returns:

numpy.ndarray: predicted y values
"""

```
self.mse = "You should use performance() function first!"  
self.rmse = "You should use performance() function first!"  
self.mae = "You should use performance() function first!"  
self.r2 = "You should use performance() function first!"  
self.ar2 = "You should use performance() function first!"  
self.sse = "You should use performance() function first!"  
self.ssr = "You should use performance() function first!"  
self.sst = "You should use performance() function first!"  
# modify the features X by adding one column with value equal to 1  
ones = np.ones(len(X))  
features = np.c_[ones, X]  
# predict by multiplying the feature matrix with the weight matrix  
y_predicted = np.dot(features, self.weights.T)  
return y_predicted
```

```

def fit(
    self,
    X,
    y,
    init_weights: list = None,
    lr=0.00001,
    iters=1000,
    bins=100,
    verbose=False,
):
    """
    This function calculates optimal weights using X(predictors) and Y(true results).

    Args:
        X (numpy.ndarray): predictors
        y (numpy.ndarray): true results
        init_weights (list, optional): initial weights(including bias). Defaults to None.
        lr (float, optional): learning rate. Defaults to 0.00001.
        iters (int, optional): number of iterations. Defaults to 1000.
        bins (int, optional): specifies how many iterations an MSE value will be saved to mse_history. Defaults to 100.
        verbose (bool, optional): prints weights and MSE value in the current iteration. Defaults to False.

    Returns:
        numpy.ndarray: optimal weights(including bias)
    """
    n_samples = len(X)
    ones = np.ones(len(X))
    # modify x, add 1 column with value 1
    features = np.c_[ones, X]
    # initialize the weights matrix

```

```

if init_weights != None:
    if len(init_weights) != features.shape[1]:
        print(f"The length of 'init_weights' should be {features.shape[1]}")
        return
    else:
        self.weights = np.array(init_weights).reshape((1, len(init_weights)))
else:
    self.weights = np.zeros((1, features.shape[1]))
self.lr = lr
self.iters = iters
self.n_features = X.shape[1]
self.mse_history = []
self.bins = bins

for i in range(self.iters):
    # predicted labels
    y_predicted = np.dot(features, self.weights.T)
    # calculate the error
    error = y_predicted - y
    # compute the partial derivated of the cost function
    dw = (2 / n_samples) * np.dot(features.T, error)
    dw = np.sum(dw.T, axis=0).reshape(1, -1)
    # update the weights matrix
    self.weights -= self.lr * dw

    if i % self.bins == 0:
        self.mse_history.append(np.mean(np.sum(error**2)))
        if verbose:
            print(
                f"After {i} iterations: weights = {self.weights}, MSE = {np.mean(np.sum(error**2)):.6f}"
            )

if verbose:
    print(
        f"After {self.iters} iterations: weights = {self.weights}, MSE = {np.mean(np.sum(error**2)):.6f}"
    )
return self.weights

```



```
def visualize(self, size=(15, 6), bottom=0, top=None, left=-10, right=None):
```

```
    """
```

This function plots the cost and iteration graph.

Args:

size (tuple, optional): (width of plot, height of plot). Defaults to (15, 6).

bottom (int or float, optional): lowest value of y axis. Defaults to 0.

top (int or float, optional): highest value of y axis. Defaults to None.

left (int, optional): lowest value of x axis. Defaults to -10.

right (int, optional): highest value of x axis. Defaults to None.

```
    """
```

```
    if top == None:
```

```
        top = max(self.mse_history)
```

```
    if right == None:
```

```
        right = (self.iters // self.bins) * self.bins
```

```
    plt.figure(figsize=size)
```

```
    plt.title("Cost and Iteration", fontsize=20)
```

```
    plt.xlabel("Iterations")
```

```
    plt.ylabel("MSE")
```

```
    plt.plot(range(0, self.iters, self.bins), self.mse_history, color="b")
```

```
    plt.ylim(bottom=bottom, top=top)
```

```
    plt.xlim(left=left, right=right)
```

```
    plt.show(block=True)
```

```
def cross_validate(  
    self,  
    X,  
    y,  
    lr=0.00001,  
    iters=1000,  
    k=10,  
    scoring="r2",  
    init_weights: list = None,  
    verbose=True,  
):
```

```
    """
```

This function applies K-fold cross validation to the dataset and assess the performance of the model in a robust and reliable manner.

Args:

X (numpy.ndarray): predictors(input)

y (numpy.ndarray): true results

lr (float, optional): learning rate. Defaults to 0.00001.

iters (int, optional): number of iterations. Defaults to 1000.

k (int, optional): number of folds which the original dataset is divided. Defaults to 10.

scoring (str, optional): the performance metric to be calculated. Defaults to "r2".

init_weights (list, optional): initial weights(including bias). Defaults to None.

verbose (bool, optional): prints the average score and score list. Defaults to True.

Returns:

tuple: (average score, score list)

```
    """
```

```
if scoring not in ["r2", "ar2", "mse", "rmse", "mae"]:
    print(
        "The 'scoring' parameter is invalid. Available ones are the following:"
    )
    print(["r2", "ar2", "mse", "rmse", "mae"])
    return
scores = []
kf = KFold(n_splits=k, shuffle=True)
for train_index, test_index in kf.split(X):
    model = LinearRegressionGD()
    x_train, x_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    model.fit(x_train, y_train, lr=lr, iters=iters, init_weights=init_weights)
    y_pred = model.predict(x_test)
    model.performance(y_pred, y_test, verbose=False)
    if scoring == "r2":
        scores.append(model.r2)
    elif scoring == "ar2":
        scores.append(model.ar2)
    elif scoring == "mse":
        scores.append(model.mse)
    elif scoring == "rmse":
        scores.append(model.rmse)
    elif scoring == "mae":
        scores.append(model.mae)
if verbose:
    print(f"{scoring} scores : {scores}")
    print(f"Average {scoring} : {np.mean(scores)}")
return np.mean(scores), scores
```

```
class LinearRegressionGD:
    def __init__(self):
        """
        The LinearRegressionGD class constructor.
        """

        # initialize learning rate lr and number of iteration iters
        self.lr = None
        self.iters = None
        # initialize the weights matrix
        self.weights = None
        # bins specifies how many iterations an MSE value will be saved to mse_history.
        self.bins = None
        # mse_history records MSE values in bins intervals.
        self.mse_history = []
        # keeps how many independent variables there are.
        self.n_features = "You should use fit() function first!"
        # keeps the MSE value of the optimal model.
        self.mse = "You should use performance() function first!"
        # keeps the RMSE value of the optimal model.
        self.rmse = "You should use performance() function first!"
        # keeps the MAE value of the optimal model.
        self.mae = "You should use performance() function first!"
        # keeps the R-squared value of the optimal model.
        self.r2 = "You should use performance() function first!"
        # keeps the adjusted R-squared value of the optimal model.
        self.ar2 = "You should use performance() function first!"
        # keeps the SSE value of the optimal model.
        self.sse = "You should use performance() function first!"
        # keeps the SSR value of the optimal model.
        self.ssr = "You should use performance() function first!"
        # keeps the SST value of the optimal model.
        self.sst = "You should use performance() function first!"
```

```
def performance(self, y_predicted, y, verbose=True):
```

```
    """
```

This function calculates performance metrics such as
RMSE, MSE, MAE, SSR, SSE, SST, R-squared and Adj. R-squared.

Args:

y_predicted (numpy.ndarray): predicted y values

y (numpy.ndarray): true y values

verbose (bool, optional): prints performance metrics. Defaults to True.

```
    """
```

```
self.mse = np.mean(np.sum((y_predicted - y) ** 2))
```

```
self.rmse = np.sqrt(self.mse)
```

```
self.mae = np.mean(np.abs(y - y_predicted))
```

```
self.ssr = np.sum((y_predicted - np.mean(y)) ** 2)
```

```
self.sst = np.sum((y - np.mean(y)) ** 2)
```

```
self.sse = np.sum((y - y_predicted) ** 2)
```

```
self.r2 = 1 - self.sse / self.sst
```

```
self.ar2 = 1 - (((1 - self.r2) * (len(y) - 1)) / (len(y) - self.n_features - 1))
```

```
if verbose:
```

```
    print(f"RMSE = {self.rmse}")
```

```
    print(f"MSE = {self.mse}")
```

```
    print(f"MAE = {self.mae}")
```

```
    print(f"SSE = {self.sse}")
```

```
    print(f"SSR = {self.ssr}")
```

```
    print(f"SST = {self.sst}")
```

```
    print(f"R-squared = {self.r2}")
```

```
    print(f"Adjusted R-squared = {self.ar2}")
```

```
def predict(self, x):  
    """
```

This function takes one argument which is a numpy.array of predictor values, and returns predicted y values.

Note: You should use fit() function at least once before using predict() function, since the prediction is made with the optimal weights obtained by the fit() function.

Args:

X (numpy.ndarray): predictors(input)

Returns:

numpy.ndarray: predicted y values
"""

```
self.mse = "You should use performance() function first!"  
self.rmse = "You should use performance() function first!"  
self.mae = "You should use performance() function first!"  
self.r2 = "You should use performance() function first!"  
self.ar2 = "You should use performance() function first!"  
self.sse = "You should use performance() function first!"  
self.ssr = "You should use performance() function first!"  
self.sst = "You should use performance() function first!"  
# modify the features x by adding one column with value equal to 1  
ones = np.ones(len(X))  
features = np.c_[ones, X]  
# predict by multiplying the feature matrix with the weight matrix  
y_predicted = np.dot(features, self.weights.T)  
return y_predicted
```

```
def fit(
    self,
    X,
    y,
    init_weights: list = None,
    lr=0.00001,
    iters=1000,
    bins=100,
    verbose=False,
):
    """
```

This function calculates optimal weights using X(predictors) and Y(true results).

Args:

X (numpy.ndarray): predictors
 y (numpy.ndarray): true results
 init_weights (list, optional): initial weights(including bias). Defaults to None.
 lr (float, optional): learning rate. Defaults to 0.00001.
 iters (int, optional): number of iterations. Defaults to 1000.
 bins (int, optional): specifies how many iterations an MSE value will be saved to mse_history.
 Defaults to 100.
 verbose (bool, optional): prints weights and MSE value in the current iteration. Defaults to False.

Returns:

numpy.ndarray: optimal weights(including bias)
 """

```
n_samples = len(X)
ones = np.ones(len(X))
# modify x, add 1 column with value 1
features = np.c_[ones, X]
# initialize the weights matrix
```

```

if init_weights != None:
    if len(init_weights) != features.shape[1]:
        print(f"The length of 'init_weights' should be {features.shape[1]}")
        return
    else:
        self.weights = np.array(init_weights).reshape((1, len(init_weights)))
else:
    self.weights = np.zeros((1, features.shape[1]))
self.lr = lr
self.its = its
self.n_features = x.shape[1]
self.mse_history = []
self.bins = bins

for i in range(self.its):
    # predicted labels
    y_predicted = np.dot(features, self.weights.T)
    # calculate the error
    error = y_predicted - y
    # compute the partial derivated of the cost function
    dw = (2 / n_samples) * np.dot(features.T, error)
    dw = np.sum(dw.T, axis=0).reshape(1, -1)
    # update the weights matrix
    self.weights -= self.lr * dw

    if i % self.bins == 0:
        self.mse_history.append(np.mean(np.sum(error**2)))
        if verbose:
            print(
                f"After {i} iterations: weights = {self.weights}, MSE = {np.mean(np.sum(error**2)):.6f}"
            )

if verbose:
    print(
        f"After {self.its} iterations: weights = {self.weights}, MSE = {np.mean(np.sum(error**2)):.6f}"
    )
return self.weights

```



```
def visualize(self, size=(15, 6), bottom=0, top=None, left=-10, right=None):
    """
    This function plots the cost and iteration graph.

    Args:
        size (tuple, optional): (width of plot, height of plot). Defaults to (15, 6).
        bottom (int or float, optional): lowest value of y axis. Defaults to 0.
        top (int or float, optional): highest value of y axis. Defaults to None.
        left (int, optional): lowest value of x axis. Defaults to -10.
        right (int, optional): highest value of x axis. Defaults to None.
    """
    if top == None:
        top = max(self.mse_history)
    if right == None:
        right = (self.iters // self.bins) * self.bins
    plt.figure(figsize=size)
    plt.title("Cost and Iteration", fontsize=20)
    plt.xlabel("Iterations")
    plt.ylabel("MSE")
    plt.plot(range(0, self.iters, self.bins), self.mse_history, color="b")
    plt.ylim(bottom=bottom, top=top)
    plt.xlim(left=left, right=right)
    plt.show(block=True)
```



```
def cross_validate(
```

```
    self,
```

```
    X,
```

```
    y,
```

```
    lr=0.00001,
```

```
    iters=1000,
```

```
    k=10,
```

```
    scoring="r2",
```

```
    init_weights: list = None,
```

```
    verbose=True,
```

```
):
```

```
    """
```

This function applies K-fold cross validation to the dataset and assess the performance of the model in a robust and reliable manner.

Args:

X (numpy.ndarray): predictors(input)

y (numpy.ndarray): true results

lr (float, optional): learning rate. Defaults to 0.00001.

iters (int, optional): number of iterations. Defaults to 1000.

k (int, optional): number of folds which the original dataset is divided. Defaults to 10.

scoring (str, optional): the performance metric to be calculated. Defaults to "r2".

init_weights (list, optional): initial weights(including bias). Defaults to None.

verbose (bool, optional): prints the average score and score list. Defaults to True.

Returns:

tuple: (average score, score list)

```
    """
```

```
if scoring not in ["r2", "ar2", "mse", "rmse", "mae"]:
    print(
        "The 'scoring' parameter is invalid. Available ones are the following:"
    )
    print(["r2", "ar2", "mse", "rmse", "mae"])
    return
scores = []
kf = KFold(n_splits=k, shuffle=True)
for train_index, test_index in kf.split(X):
    model = LinearRegressionGD()
    x_train, x_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    model.fit(x_train, y_train, lr=lr, iters=iters, init_weights=init_weights)
    y_pred = model.predict(x_test)
    model.performance(y_pred, y_test, verbose=False)
    if scoring == "r2":
        scores.append(model.r2)
    elif scoring == "ar2":
        scores.append(model.ar2)
    elif scoring == "mse":
        scores.append(model.mse)
    elif scoring == "rmse":
        scores.append(model.rmse)
    elif scoring == "mae":
        scores.append(model.mae)
if verbose:
    print(f"{scoring} scores : {scores}")
    print(f"Average {scoring} : {np.mean(scores)}")
return np.mean(scores), scores
```

```

from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.model_selection import KFold
y = df[["sales"]]
X = df.drop("sales", axis=1)

# We need to convert X and y to numpy array!
X = np.array(X)
y = np.array(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=1)
# Let's try k=5 cross validation with 1.5 million iteration and 0.00003 learning rate.
model_01 = LinearRegressionGD()
model_01.cross_validate(X,y,lr=0.00003, k = 5, iters=1500000, scoring = "r2")

r2 scores : [0.9272448819192566, 0.9159523968877537, 0.9114713301826926, 0.8096015788983072, 0.9071467599479468]
Average r2 : 0.8942833895671913

```

```

(0.8942833895671913,
 [0.9272448819192566,
  0.9159523968877537,
  0.9114713301826926,
  0.8096015788983072,
  0.9071467599479468])

```

```

from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.model_selection import KFold
y = df[["sales"]]
X = df.drop("sales", axis=1)

```

```

# We need to convert X and y to numpy array!
X = np.array(X)
y = np.array(y)

```

```

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=1)
# Let's try k=5 cross validation with 1.5 million iteration and 0.00003 learning rate.
model_01 = LinearRegressionGD()
model_01.cross_validate(X,y,lr=0.00003, k = 5, iters=1500000, scoring = "r2")

```

```
model_01.fit(X_train,y_train, lr=0.00003, iters=1500000, bins = 10000, verbose=True)
```

Training Results

```
y_pred = model_01.predict(X_train)
model_01.performance(y_pred,y_train)
```

RMSE = 21.97027303484479

MSE = 482.69289722562814

MAE = 1.3288503051314866

SSE = 482.69289722562814

SSR = 4155.787835094638

SST = 4638.47975

R-squared = 0.8959372632325002

Adjusted R-squared = 0.8939360567562021

Test Results

```
y_pred = model_01.predict(X_test)
model_01.performance(y_pred,y_test)
```

RMSE = 8.926109222098756

MSE = 79.67542584483645

MAE = 1.0402155158461008

SSE = 79.67542584483645

SSR = 682.7488015856429

SST = 742.96775

R-squared = 0.8927605863850262

Adjusted R-squared = 0.8838239685837784

```
model_01.fit(X_train,y_train, lr=0.00003, iters=1500000, bins = 10000, verbose=True)
```

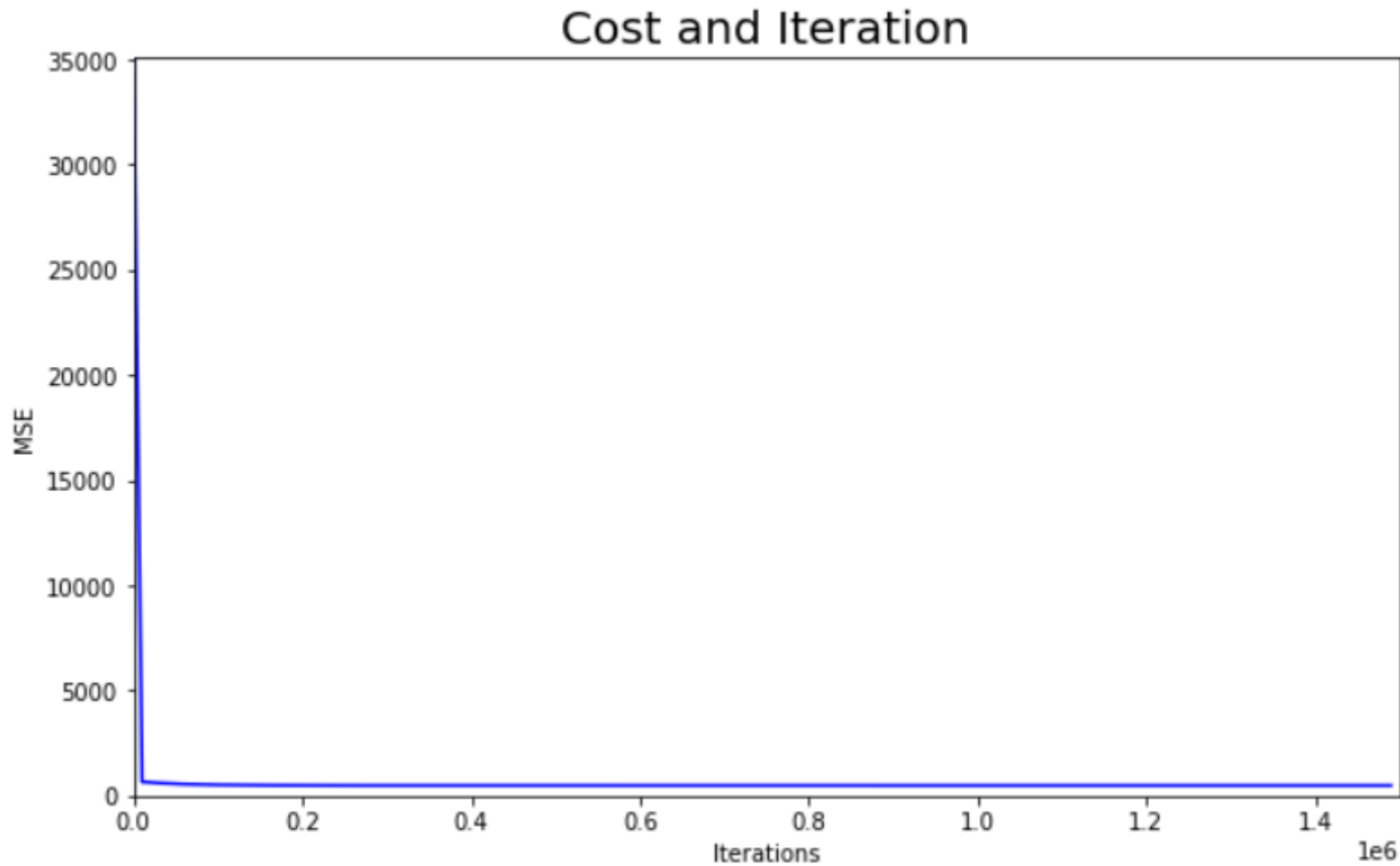
Training Results

```
y_pred = model_01.predict(X_train)
model_01.performance(y_pred,y_train)
```

Test Results

```
y_pred = model_01.predict(X_test)
model_01.performance(y_pred,y_test)
```

```
model_01.visualize(size = (10,6))
```



```
model_01.visualize(size = (10,6))
```

If you apply scaling you will reach optimal weights with less iterations and probably higher learning rate..

```
from sklearn.preprocessing import RobustScaler
# Scale X
rs = RobustScaler()
X_scaled = rs.fit_transform(X)

# Convert X and y to numpy array
X_scaled = np.array(X_scaled)
y = np.array(y)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.20, random_state=1)

# Learning rate = 0.9
# Number of iteration = 50
model_02 = LinearRegressionGD()
model_02.fit(X_train, y_train, lr=0.9, iters=50, bins = 5, verbose=True)
```

```
After 0 iterations: weights = [[24.86025      3.40410826  3.51735593  4.95713949]], MSE = 35158.580000
After 5 iterations: weights = [[ 8.6359129   7.1169975   4.15379772 -1.02471141]], MSE = 7438.581899
After 10 iterations: weights = [[16.66666278  6.59370776  4.90383833  0.70296709]], MSE = 2071.160556
After 15 iterations: weights = [[12.83927962  6.84929     4.65171459 -0.20460628]], MSE = 845.743302
After 20 iterations: weights = [[14.67005557  6.72692153  4.78179699  0.22217159]], MSE = 565.671808
After 25 iterations: weights = [[13.79488468  6.78540335  4.72045574  0.0175065 ]], MSE = 501.658601
After 30 iterations: weights = [[14.21329411  6.75744255  4.74985714  0.11529674]], MSE = 487.027709
After 35 iterations: weights = [[14.01326159  6.77080989  4.73580761  0.0685402 ]], MSE = 483.683664
After 40 iterations: weights = [[14.10889321  6.76441921  4.74252501  0.09089313]], MSE = 482.919347
After 45 iterations: weights = [[14.06317365  6.76747447  4.73931361  0.0802066 ]], MSE = 482.744655
After 50 iterations: weights = [[14.06976731  6.76703384  4.73977676  0.08174781]], MSE = 482.708789
```

```
array([[14.06976731,  6.76703384,  4.73977676,  0.08174781]])
```

Training Results

```
y_pred = model_02.predict(X_train)
model_02.performance(y_pred,y_train)
```

RMSE = 21.970542254675976

MSE = 482.7047269645026

MAE = 1.3305833102353888

SSE = 482.7047269645026

SSR = 4155.345351811815

SST = 4638.47975

R-squared = 0.8959347128842154

Adjusted R-squared = 0.8939334573627581

Test Results

```
y_pred = model_02.predict(X_test)
model_02.performance(y_pred,y_test)
```

RMSE = 8.92568912219261

MSE = 79.6679263060275

MAE = 1.0416834408256286

SSE = 79.6679263060275

SSR = 682.7324069350906

SST = 742.96775

R-squared = 0.8927706804150954

Adjusted R-squared = 0.8838349037830201

Training Results

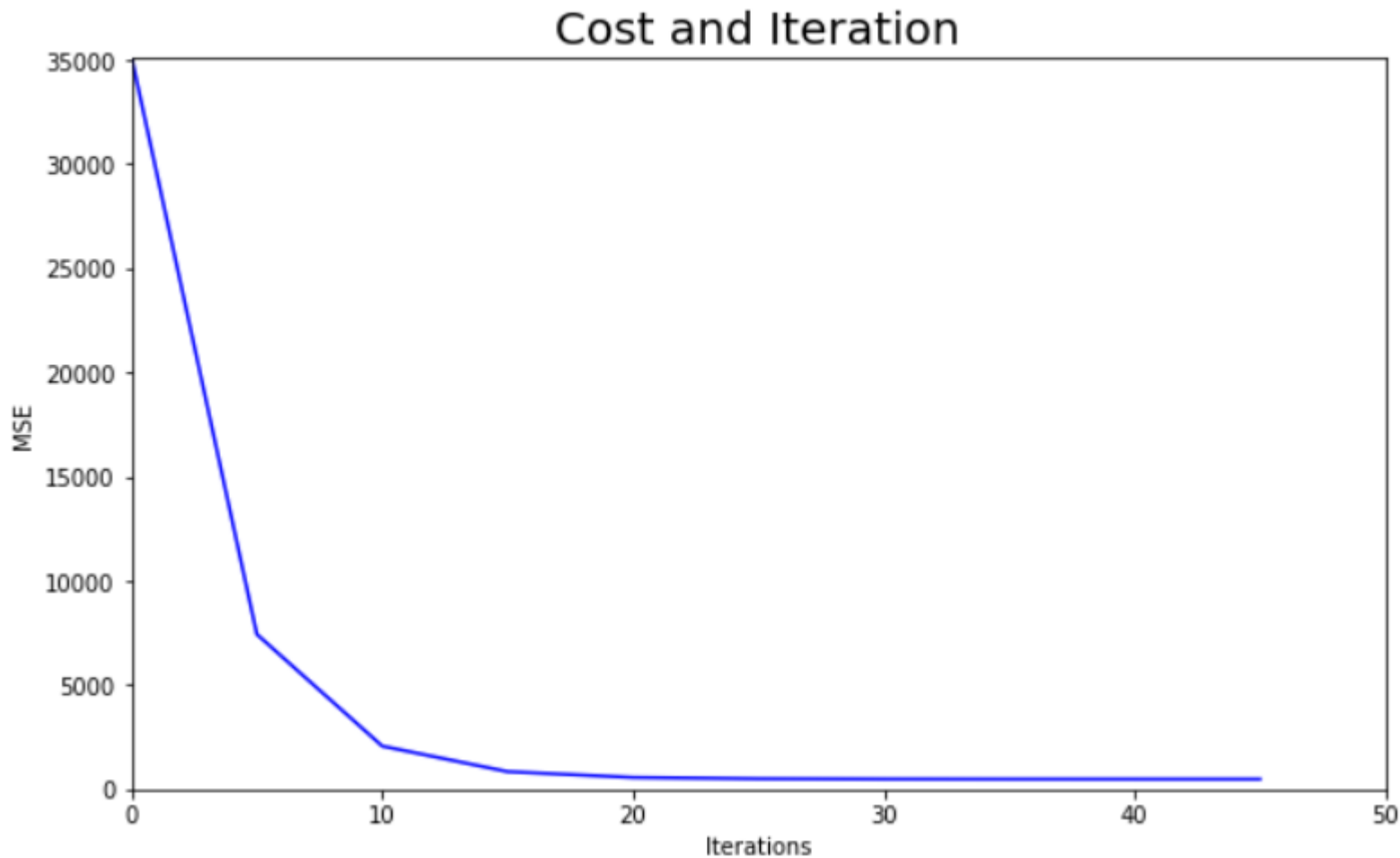
```
y_pred = model_02.predict(X_train)
model_02.performance(y_pred,y_train)
```

Test Results

```
y_pred = model_02.predict(X_test)
model_02.performance(y_pred,y_test)
```



```
model_02.visualize(size = (10,6), left = 0)
```



As can be seen, without applying feature scaling, we were able to reach the optimal point with 400,000 iterations and 0.00003 learning rate. However, after applying feature scaling, we quickly reached the optimal point with 50 iterations and 0.9 learning rate.

```
model_02.visualize(size = (10,6), left = 0)
```