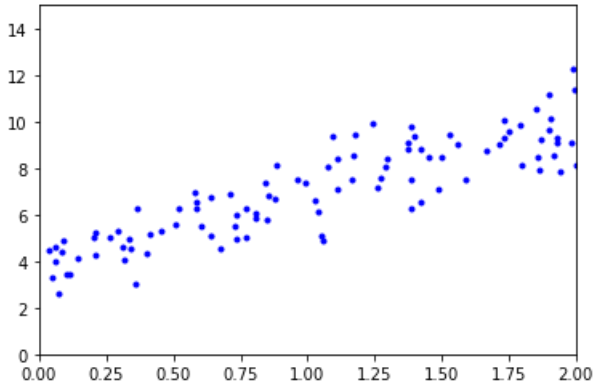
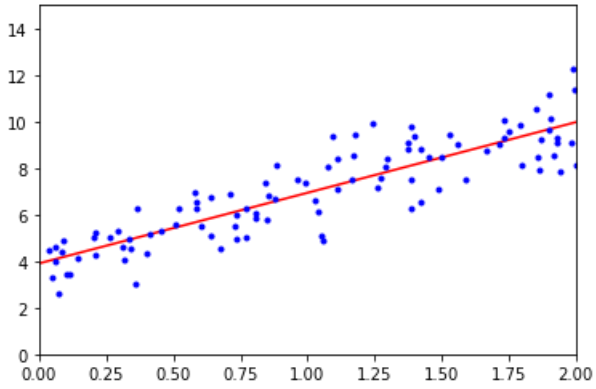
# Linear Regression

Linear Regression is a supervised learning algorithm, which works best for continuous data. Linear Regression is most common and one of the powerful algorithm.
In this lecture, we will talk about this in detail.

So below, is our scatter data,




We will fit the straight line and make predictions.




So, above we fitted the straight line on our inputs and we are predicting the output using x and y.

# How do we make this straight line?

For making this straight line we have a function "h(x)" as:-

$$\hat{Y} = h(x) = \Theta_0 * x_0 + \Theta_1 * x_1 + \Theta_2 * x_2 + \Theta_3 * x_3 + \Theta_n * x_n$$

In the above equation, $\Theta_0$ is a bias term often called the intercept and $\Theta_1, \Theta_2, \Theta_3, \ldots \Theta_n$ are the feature weights.

Above is the equation of Linear Regression with multiple variables often called **Multivariate Linear Regression**, There is also a univariate linear regression, which has only one feature means on the basis of only one feature we have to predict the output but in general you will see **Multivariate Linear Regression** a lot and In above equation we have multiple features and on the basic of multiple features , we are predicting the output.

More vectorized form of the above equation is

$$\hat{Y} = h(x) = \Theta * X$$

$\Theta$ is the parameter vector containing the bias term and feature weights often abbreviated as a giant vector theta.

X is the feature vector containing all x's and note $x_0$ is always equals to 1. theta and x is the dot product which is equals to the same equation above.

**Example**:- let's say you have one only one feature and basis on that you have to predict, so let's take $\Theta_0 = 2.5$ and $\Theta_1 = 3.6$

if, we make use of this bias term and feature weights, $\hat{Y} = h(x) = \Theta_0 + \Theta_1 * x_1$

$\hat{Y} = h(x) = 2.5 + 3.6 * x$

Now, you can put the values of x and then you will get your desired output.

## Cost Function J(Θ)

So, we have choose our $\Theta_0$ and $\Theta_1$ but who gaurantees that our parameters is the best parameter, so for checking the accuracy of our hypothesis or our parameters, we have cost function.

Cost Function J(Θ) is directly propostional to the error. Higher the cost function is higher the error is and vice versa

There are two algorithms which are mostly used in calculating the cost function,

- MSE ( Mean Squared Error )
- RMSE ( Root Mean Squared Error )

for now we will make use of MSE, so the equation of Mean Squared Error is:-

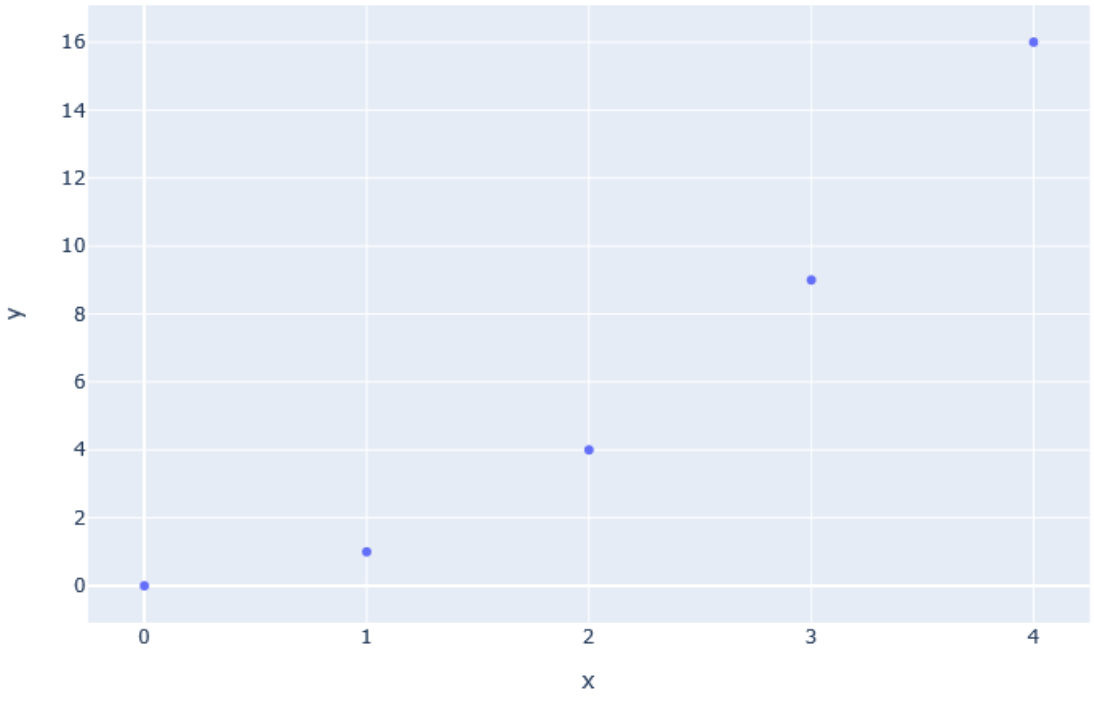
for now we will focus on MSE, so the equation of Mean Squared Error is:-

$$\text{MSE}(\theta_0, \theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \frac{1}{m} + \sum_{i=1}^{m} (\Theta^T x^i - y^i)^2$$

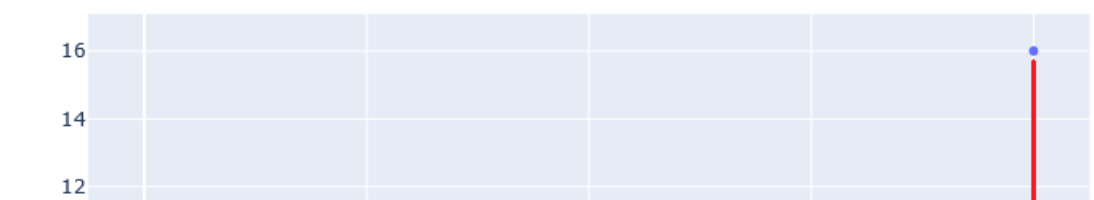
Understanding the above equation

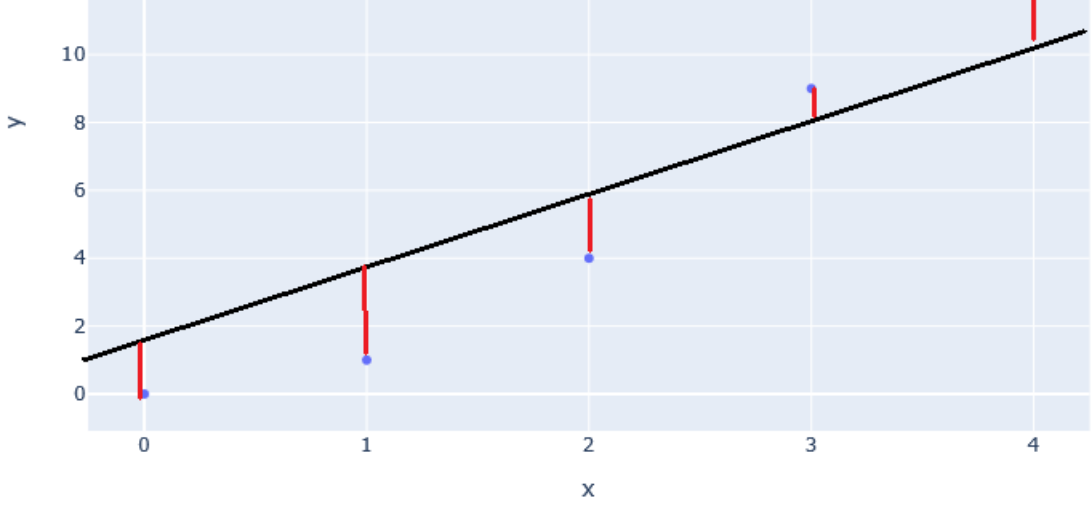
- m is the size of training examples, we can take out with the 'len(y)'.

Below we have a simple data




Now, we fit a line on this data






Cost function is taking out the distance between actual and predicted values by subtracting $h(x^i) - y^i$ and at last we are squaring them up.

So, If our cost function is relatively large then we haven't choose the best theta, we have to try to another theta.

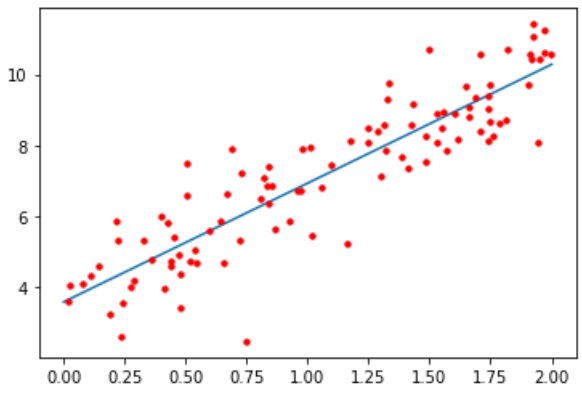
There is a direct method or we can say equation which is called the normal equation,which gives the best $\Theta$ parmeter. Below is the equation of of Normal equation.

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

## Implementation of Normal Equation

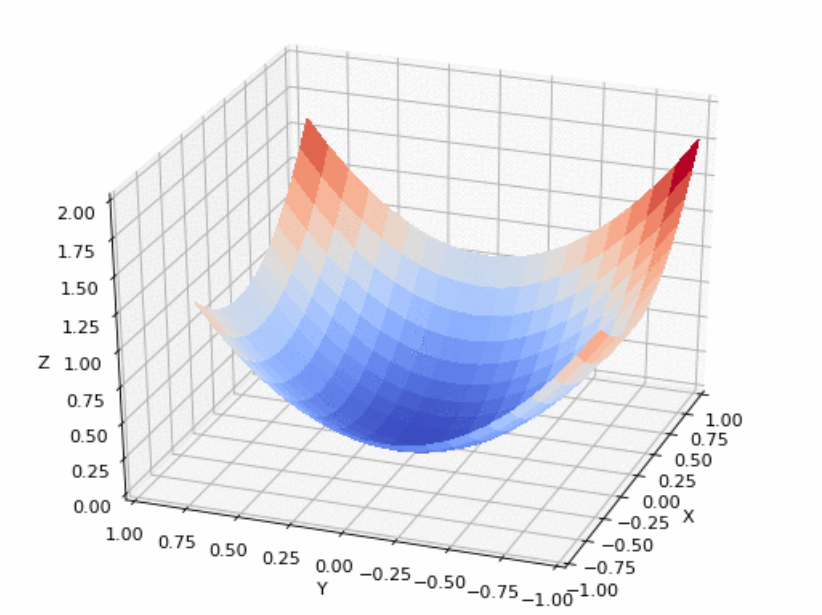
In [5]:

```
def generate_data():
    X = 2 * np.random.rand(100, 1)
    y = 4 + 3 * X + np.random.randn(100, 1)

    return X,y

def get_best_param(X, y):
    X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
    theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y) # normal equation formula

    return theta_best

def predict(X,theta):
    return X.dot(theta)

X,y = generate_data()
plt.plot(X,y,"r.")

theta = get_best_param(X,y)
theta

X_to_pred = np.array([[0],[2]])
X_new_b = np.c_[np.ones((2, 1)), X_to_pred]
predict= predict(X_new_b,theta)

plt.plot(X_to_pred, predict)
plt.plot(X,y,"r.")
plt.show()
```

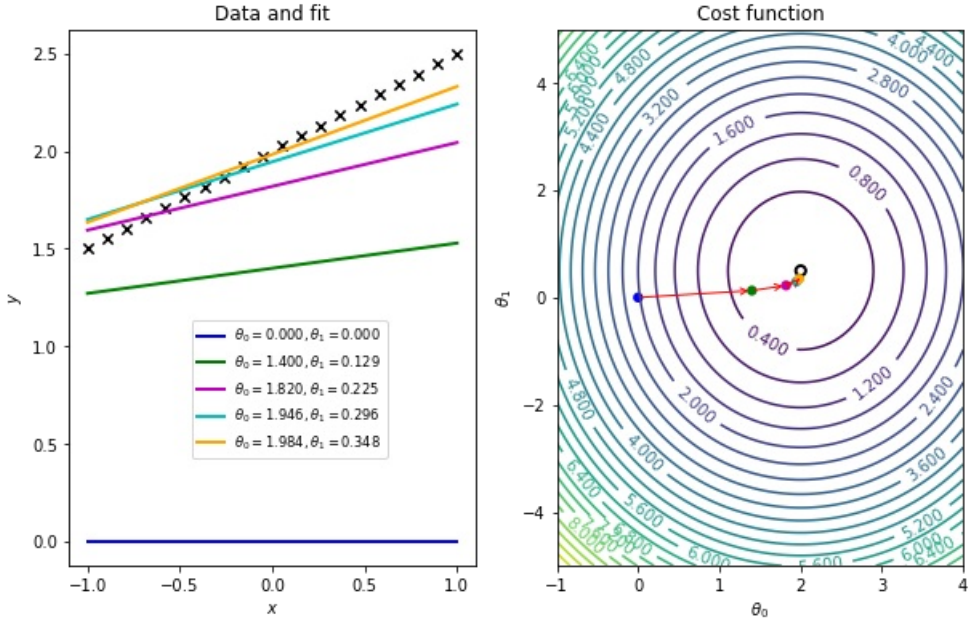


# Batch Gradient Descent

Gradient Descent is an Opitmization algorithm that help us to find the best theta parameter, it simple tweak the parameters to reduce our cost function, so if our cost function reduces it means that we are getting good theta parameters.




Let's take another example




# Implementation of Gradient Descent

To Implement gradient descent, you need to compute the gradient of the cost function with respect to each model parameter $\Theta_j$

You need to compute the **partial deriviative** of our cost function i.e how much cost function will change when if you change $\Theta$ little bit.

$$\frac{\partial}{\partial \Theta_j} J(\Theta) = \frac{2}{m} + \sum_{i=1}^{m} (\Theta^T x^i - y^i)^2$$

### What we are doing?

$$\frac{\partial}{\partial \Theta_0} J(\Theta) = \frac{2}{m} + \sum_{i=1}^{m} (\Theta^T x^i - y^i)^2$$

$$\frac{\partial}{\partial \Theta_1} J(\Theta) = \frac{2}{m} + \sum_{i=1}^{m} (\Theta^T x^i - y^i)^2$$

$$\frac{\partial}{\partial \Theta_2} J(\Theta) = \frac{2}{m} + \sum_{i=1}^{m} (\Theta^T x^i - y^i)^2$$

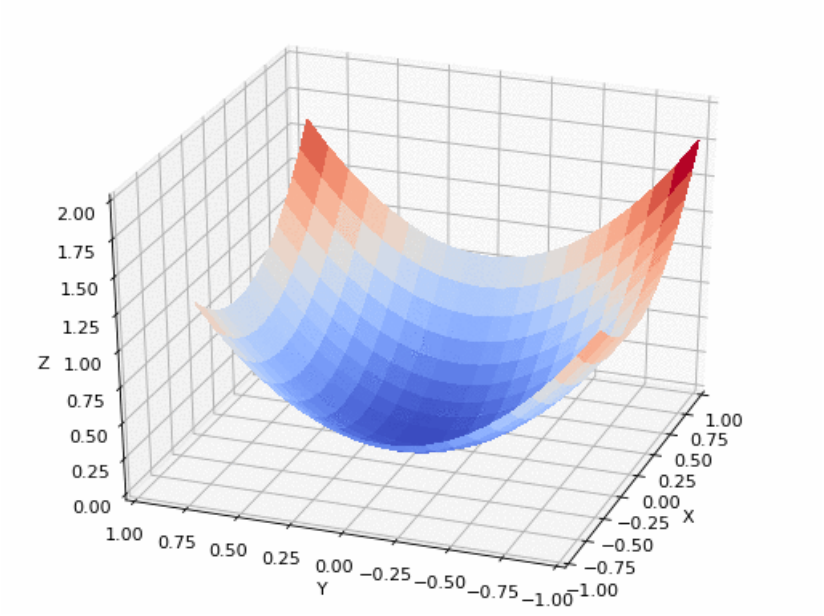
.
.
.

$$\frac{\partial}{\partial \Theta_j} J(\Theta) = \frac{2}{m} + \sum_{i=1}^{m} (\Theta^T x^i - y^i)^2$$

### Vectorized Form

$$\nabla J(\Theta) = \frac{2}{m} . X^T(X\Theta - y)$$

### Gradient Descent Step




$$\Theta_j := \Theta_j - \alpha \nabla J(\Theta)$$

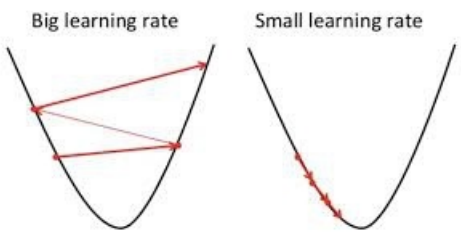
Here we are taking gradient step to reach to our global minimum, so that's why we are updating our previous theta parameter with the new one.

### How to Choose the learning rate?

You have to tune the learning rate means you have to try different different rates and see your model how it performs.

If you choose **very small learning rate**, it will be very very slow and it will never converge to the local minimum.

If you choose **very large learning rate**, your model might diverge and never converge to the local minimum.




### Implementation of Linear Regression From Scratch

In [ ]:

```
def model(X, Y, learning_rate, iteration):
    m = Y.size
    theta = np.zeros((2, 1))
    cost_list = []
    for i in range(iteration):
        y_pred = np.dot(X, theta)
        cost = (1/(2*m))*np.sum(np.square(y_pred - Y))
        d_theta = (1/m)*np.dot(X.T, y_pred - Y)
        theta = theta - learning_rate*d_theta
        cost_list.append(cost)

    return theta, cost_list
```

## Implementation of Linear Regression With Scikit Learn

In [ ]:

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression() # object
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
lin_reg.predict(X_new)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: