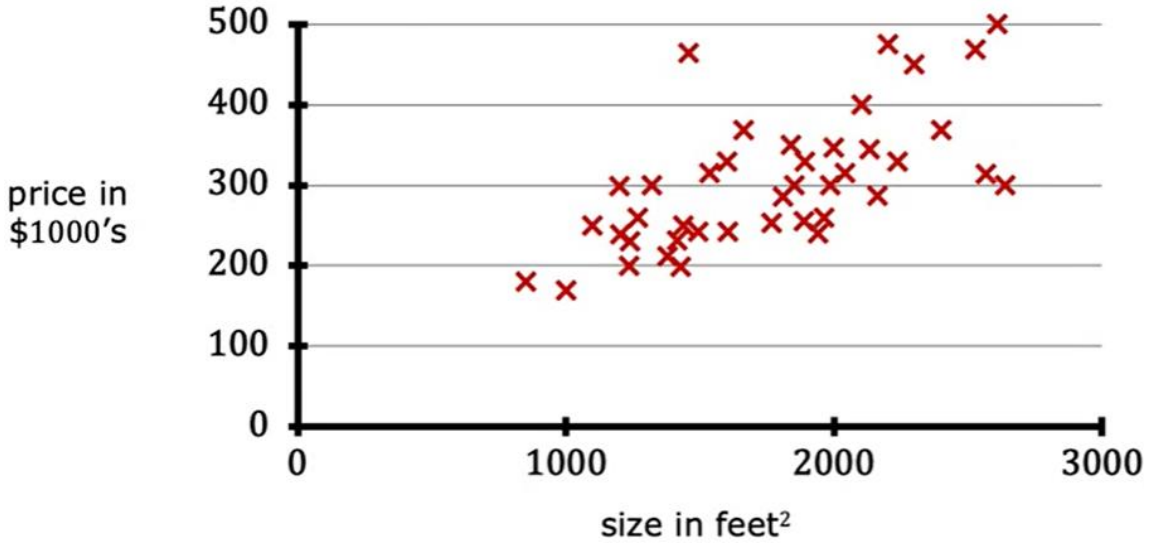
Linear Regression

Some materials are taken from course of Andrew Ng and online materials

Living area and price of houses in a city:

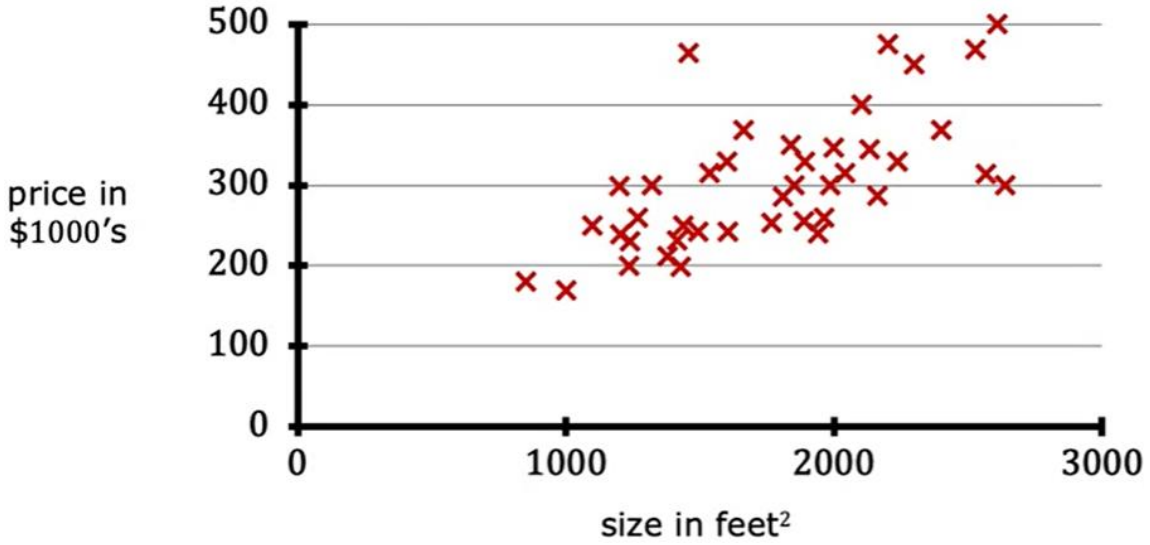
Living area (feet^2)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮



- Given this dataset, we want to predict the price of houses of other sizes in the city.
- In other words, we want the price represented as a function of the size.

Living area and price of houses in a city:

Living area ($feet^2$)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

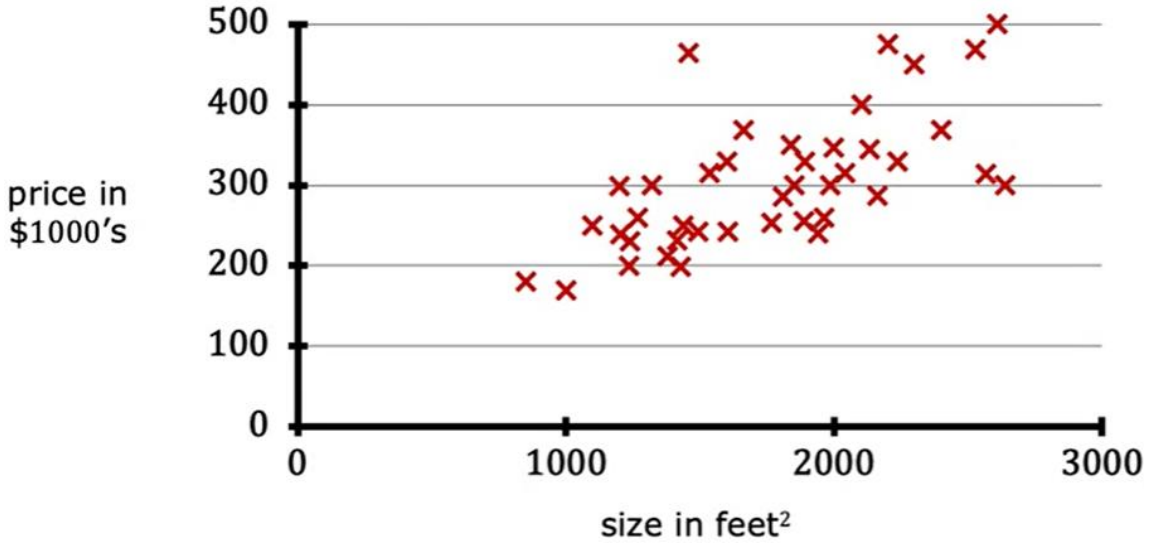


- We will consider this as a training set composed of feature (Living area size) and target/label variable (Price) -> Supervised Learning Problem.

- The target variable we are trying to predict is continuous -> Regression Problem

Living area and price of houses in a city:

Living area ($feet^2$)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮



- N = number of training examples
- (x, y) - single training example
- (x^i, y^i) - specific example (i^{th} training example)
- We want to learn a linear function $\hat{y}$ which can be expressed as:

$$\hat{y} = wx + b$$

- Univariate linear regression. How to learn the values of the parameters w and b?

Multiple input features

Living area (feet^2)	#bedrooms	Price (1000$s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
⋮	⋮	⋮

- Multivariate Linear Regression

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

Ordinary Linear Regression

- The goal of linear regression is to find a linear function that fits the training data with minimal error.
- For N-training examples (X, y), where $X = [x_D, x_{D-1}, \dots, x_2, x_1]$ is a D-dimension feature vector and corresponding target value y, the predicted linear function $\hat{y}(X, W)$ can be expressed as:

$$\hat{y} = w_D x_D + w_{D-1} x_{D-1} + \dots + w_1 x_1 + b$$

- where $W^T = [w_D, w_{D-1}, \dots, w_2, w_1]$ is a D-dimension vector and b is the bias (intercept) represent the model parameters. The goal is to find the model parameters that minimize the error between the predicted and target values as.

$$\min_{w,b} \frac{1}{2N} \sum_{n=1}^{N} (\hat{y}^{(n)} - y^{(n)})^2 = \min_{W,b} \frac{1}{2N} \sum_{n=1}^{N} (w_D x_D^{(n)} + w_{D-1} x_{D-1}^{(n)} + \dots + w_1 x_1^{(n)} + b - y^{(n)})^2$$

$$= \min_{W,b} \frac{1}{2N} \sum_{n=1}^{N} (\sum_{d=1}^{D} w_d x_d^{(n)} + b - y^{(n)})^2 = \min_{w,b} J(w, b)$$

- Where $J(w, b)$ is the cost/error function.

Ordinary Linear Regression

- To solve the optimization problem, we need to define gradient of J(w , b) with respect to w and b as:

$$\frac{\partial J(W,b)}{\partial w_d} = \frac{1}{N}\sum_{n=1}^{N}(\sum_{d=1}^{D} w_d x_d^{(n)} + b - y^{(n)})x_d^{(n)}$$

$$\frac{\partial J(W,b)}{\partial b} = \frac{1}{N}\sum_{n=1}^{N}(\sum_{d=1}^{D} w_d x_d^{(n)} + b - y^{(n)})$$

for one dimensional case, D = 1, (i.e. a line) we have:

$$\frac{\partial J(W,b)}{\partial w_1} = \frac{1}{N}\sum_{n=1}^{N}(w_1 x_1^{(n)} + b - y^{(n)})x_1^{(n)}$$

$$\frac{\partial J(W,b)}{\partial b} = \frac{1}{N}\sum_{n=1}^{N}(w_1 x_1^{(n)} + b - y^{(n)})$$

Closed form Solution

- In closed form, we can find the solution for the 1-d case when $\frac{\partial J(W,b)}{\partial b} = 0$ as:

$$0 = \frac{1}{N}\sum_{n=1}^{N}(w_1 x_1^{(n)} + b - y^{(n)})$$

$$b = \frac{1}{N}\sum_{n=1}^{N} y^{(n)} - \frac{1}{N}\sum_{n=1}^{N} w_1 x_1^{(n)}$$

$$\boxed{b = \bar{y} - w_1 \bar{x}_1}$$

and $\frac{\partial J(W,b)}{\partial w_1} = 0$

$$0 = \frac{1}{N}\sum_{n=1}^{N}(w_1 x_1^{(n)} + b - y^{(n)})x_1^{(n)}$$

$$0 = \frac{1}{N}\sum_{n=1}^{N}(w_1 x_1^{(n)} + \bar{y} - w_1 \bar{x}_1 - y^{(n)})x_1^{(n)}$$

$$0 = \frac{1}{N}\sum_{n=1}^{N} w_1 [x_1^{(n)}]^2 + \frac{1}{N}\sum_{n=1}^{N} x_1^{(n)} \bar{y} - \frac{1}{N}\sum_{n=1}^{N} w_1 \bar{x}_1 x_1^{(n)} - \frac{1}{N}\sum_{n=1}^{N} x_1^{(n)} y^{(n)} = w_1 \overline{x_1^2} + \bar{x}_1 \bar{y} - w_1 \bar{x}_1{}^2 - \frac{1}{N}\sum_{n=1}^{N} x_1^{(n)} y^{(n)}$$

$$\boxed{w_1 = \frac{(\bar{x}_1 \bar{y} - \frac{1}{N}\sum_{n=1}^{N} x_1^{(n)} y^{(n)})}{(\bar{x}_1{}^2 - \overline{x_1^2})}}$$

Example

For input data $(x_1^{(n)}, y^{(n)}), n = 1, 2, 3, 4$ where:

$$(x_1^{(1)}, y^{(1)}) = (0, 1.5), (x_1^{(2)}, y^{(2)}) = (2, 2),$$
$$(x_1^{(3)}, y^{(3)}) = (3, 3.5), (x_1^{(4)}, y^{(4)}) = (4, 4)$$

Find the line $\hat{y} = w_1 x_1 + b$ that best-fit the above data

$$w_1 = \frac{(\bar{x}_1 \bar{y} - \frac{1}{N}\sum_{n=1}^{N} x_1^{(n)} y^{(n)})}{(\bar{x}_1{}^2 - \overline{x_1^2})} \qquad b = \bar{y} - w_1 \bar{x}_1$$

$\bar{x}_1 = (0 + 2 + 3 + 4)/4 = 2.25$

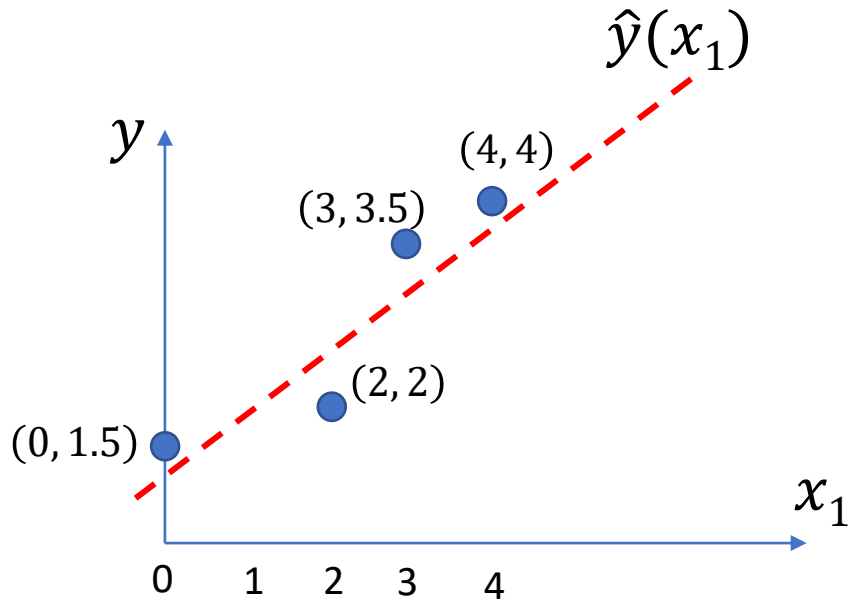
$\bar{y} = (1.5 + 2 + 3.5 + 4)/4 = 2.75$

$\overline{x_1^2} = (0 + 4 + 9 + 16)/4 = 7.25$

$\frac{1}{N}\sum_{n=1}^{N} x_1^{(n)} y^{(n)} = (0 + 4 + 10.5 + 16)/4 = 7.625$

$w_1 = \frac{(6.1875 - 7.625)}{(5.0625 - 7.25)} = 1.4375 / 2.1875 = 0.657$ $\qquad b = 2.75 - 0.657 * 2.25 = 1.272$

$$\hat{y} = 0.657 x_1 + 1.272$$



Predictions:

$x_1 = 1, \hat{y}(1) = 1.929$

$x_1 = 5, \hat{y}(5) = 4.557$

Normal Equations

We can write the training examples $(X^{(n)}, y^{(n)})$ as a set of N linear equations with unknowns W and b. For example n, the equation can be written as:

$$\hat{y}^{(n)} = w_D x_D^{(n)} + w_{D-1} x_{D-1}^{(n)} + \ldots + w_1 x_1^{(n)} + b$$

Or in a matrix form: $\hat{Y} = \acute{\mathbf{X}}\acute{W}$

where $\acute{\mathbf{X}} = [\mathbf{X} \quad 1]$ $\acute{W} = \begin{bmatrix} W \\ b \end{bmatrix}$

$$\begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(n)} \\ \vdots \\ \hat{y}^{(N-1)} \\ \hat{y}^{(N)} \end{bmatrix} = \begin{bmatrix} x_D^{(1)} & x_{D-1}^{(1)} & \cdots & x_d^{(1)} & \cdots & x_1^{(1)} & 1 \\ x_D^{(2)} & x_{D-1}^{(2)} & \cdots & x_d^{(2)} & \cdots & x_1^{(2)} & 1 \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots & \vdots \\ x_D^{(n)} & \vdots & \cdots & \vdots & \cdots & x_1^{(n)} & 1 \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots & \vdots \\ x_D^{(N-1)} & x_{D-1}^{(N-1)} & \cdots & \vdots & \cdots & x_1^{(N-1)} & 1 \\ x_D^{(N)} & x_{D-1}^{(N)} & \cdots & \vdots & \cdots & x_1^{(N)} & 1 \end{bmatrix} \begin{bmatrix} w_D \\ w_{D-1} \\ \vdots \\ w_d \\ \vdots \\ w_1 \\ b \end{bmatrix}$$

Normal Equations

We can minimize:

$$\mathrm{J}(\acute{W}) = \frac{1}{2}\left(Y - \hat{Y}\right)^{T}(Y - \hat{Y}) = \frac{1}{2}\left(Y - \acute{\mathbf{X}}\acute{W}\right)^{T}(Y - \acute{\mathbf{X}}\acute{W})$$

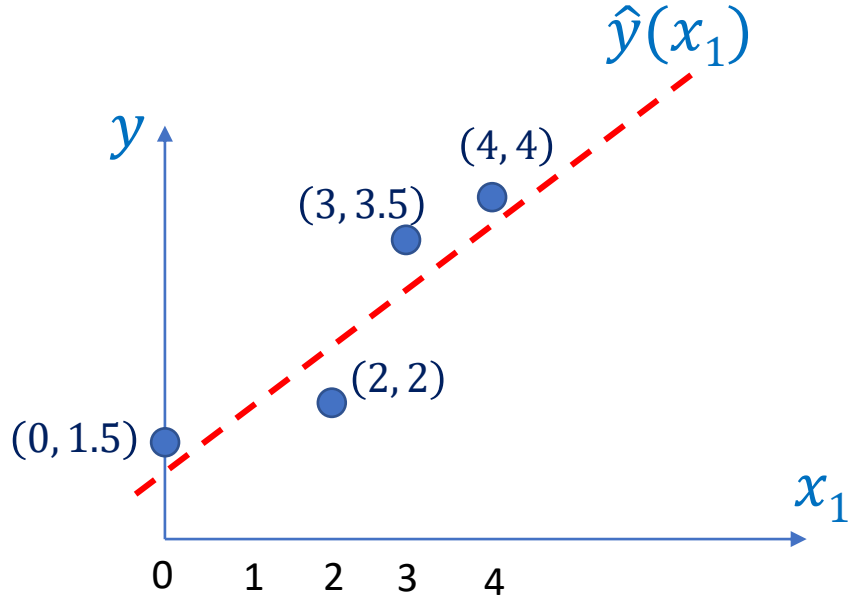
By taking the first derivative of $\mathrm{J}(\acute{W})$ With respect to $\acute{W}$ as:

$$\acute{\mathbf{X}}^{T}\,\acute{\mathbf{X}}\acute{W} - \acute{\mathbf{X}}^{T}Y = 0 \quad \text{Or} \quad \acute{\mathbf{X}}^{\mathbf{T}}\,\acute{\mathbf{X}}\acute{W} = \acute{\mathbf{X}}^{\mathbf{T}}Y, \quad \text{then,} \quad \boxed{\acute{W} = (\acute{\mathbf{X}}^{\mathbf{T}}\acute{\mathbf{X}})^{-1}\acute{\mathbf{X}}^{\mathbf{T}}Y}$$

Example

For input data $(x_1^{(n)}, y^{(n)}), n = 1, 2, 3, 4$ where:

$$(x_1^{(1)}, y^{(1)}) = (0, 1.5), (x_1^{(2)}, y^{(2)}) = (2, 2),$$
$$(x_1^{(3)}, y^{(3)}) = (3, 3.5), (x_1^{(4)}, y^{(4)}) = (4, 4)$$



So,

$$Y = \begin{bmatrix} 1.5 \\ 2 \\ 3.5 \\ 4 \end{bmatrix} \qquad \acute{\mathbf{X}} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix} \qquad \acute{W} = \begin{bmatrix} w_1 \\ b \end{bmatrix}$$

Since,

$$\acute{W} = (\acute{\mathbf{X}}^{\mathbf{T}}\acute{\mathbf{X}})^{-1}\acute{\mathbf{X}}^{T}Y$$

Then,

$$\begin{bmatrix} w_1 \\ b \end{bmatrix} = \left(\begin{bmatrix} 0 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 0 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1.5 \\ 2 \\ 3.5 \\ 4 \end{bmatrix}$$

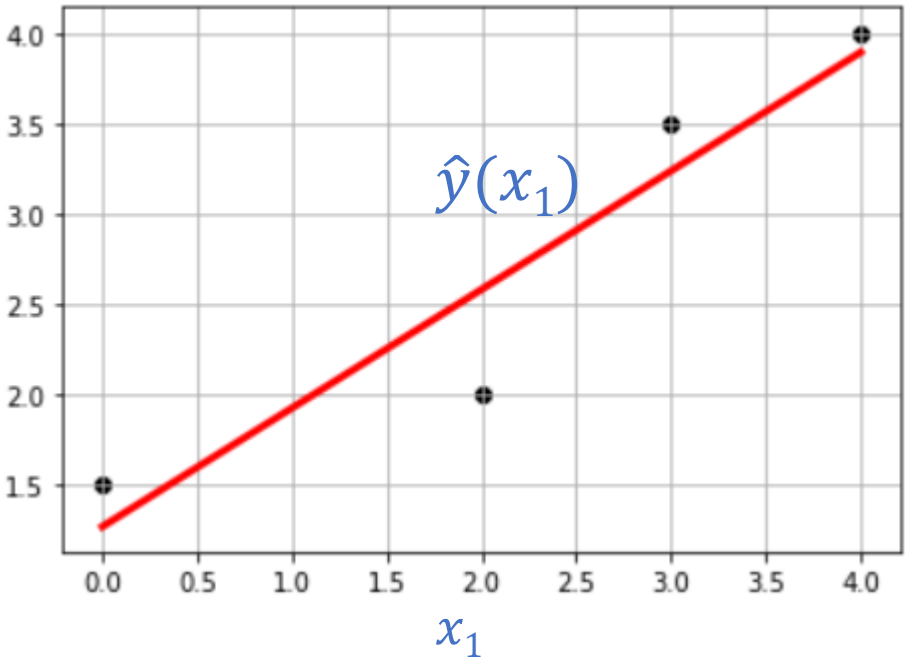
Example

$$\begin{bmatrix} w_1 \\ b \end{bmatrix} = \left(\begin{bmatrix} 0 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 0 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1.5 \\ 2 \\ 3.5 \\ 4 \end{bmatrix}$$

Which can be coded as:

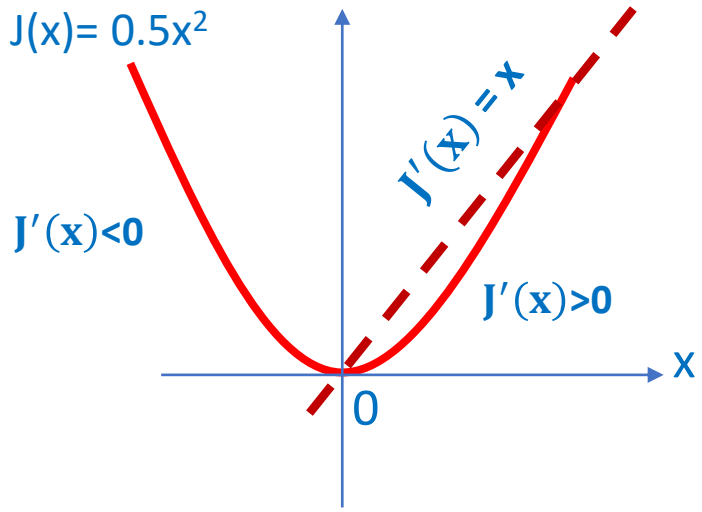
In [3]:
```
# Use the normal Equations
# construct X_prime by adding 1's to X as the last dimension
X_prime = np.array([[0,1],[2,1],[3,1],[4,1]])
Y = np.array([[1.5],[2],[3.5],[4]])
XT_X = np.matmul(X_prime.T,X_prime) # you can use np.dot() as well
W_prime = np.matmul(np.dot(np.linalg.inv(XT_X),X_prime.T),Y)
#XT_X = np.dot(X_prime.T,X_prime)
#W_prime = np.dot(np.dot(np.linalg.inv(XT_X),X_prime.T),Y)
print('w1 = %.3f\n b = %.3f'%(W_prime[0],W_prime[1])) # [[W], [b]]
plt.scatter(X_prime[:,0], Y,  color='black')
plt.plot(X_prime[:,0], np.dot(X_prime,W_prime), color='red', linewidth=3)
plt.grid(True)
plt.show()
```

```
w1 = 0.657
 b = 1.271
```



Gradient Based Optimization

- We usually use optimization techniques to minimize an objective function J(x). To maximize the same function you can minimize –J(x) instead.
- The first derivative of a function J(x) is denoted as $J'(x) = \frac{dJ(x)}{dx}$
- The gradient can be approximated as $J'(x) \approx \frac{J(x+\epsilon) - J(x)}{\epsilon}$, where ϵ is a small change in the input.
- The information we get from the gradient can help us to minimize a function (to find values of the input where the function attains minimum value(s)).



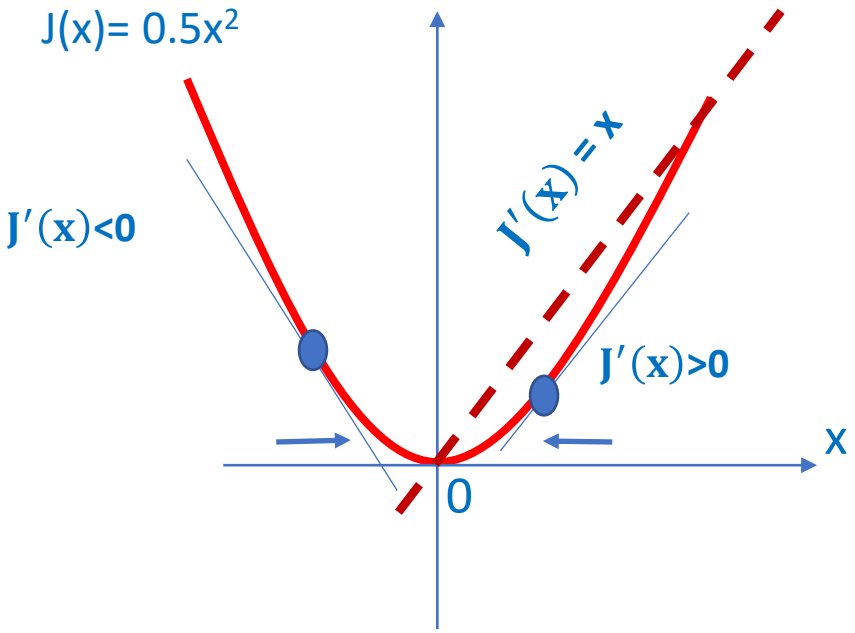
Gradient Based Optimization

- gradient descent technique uses the first derivative iteratively as:

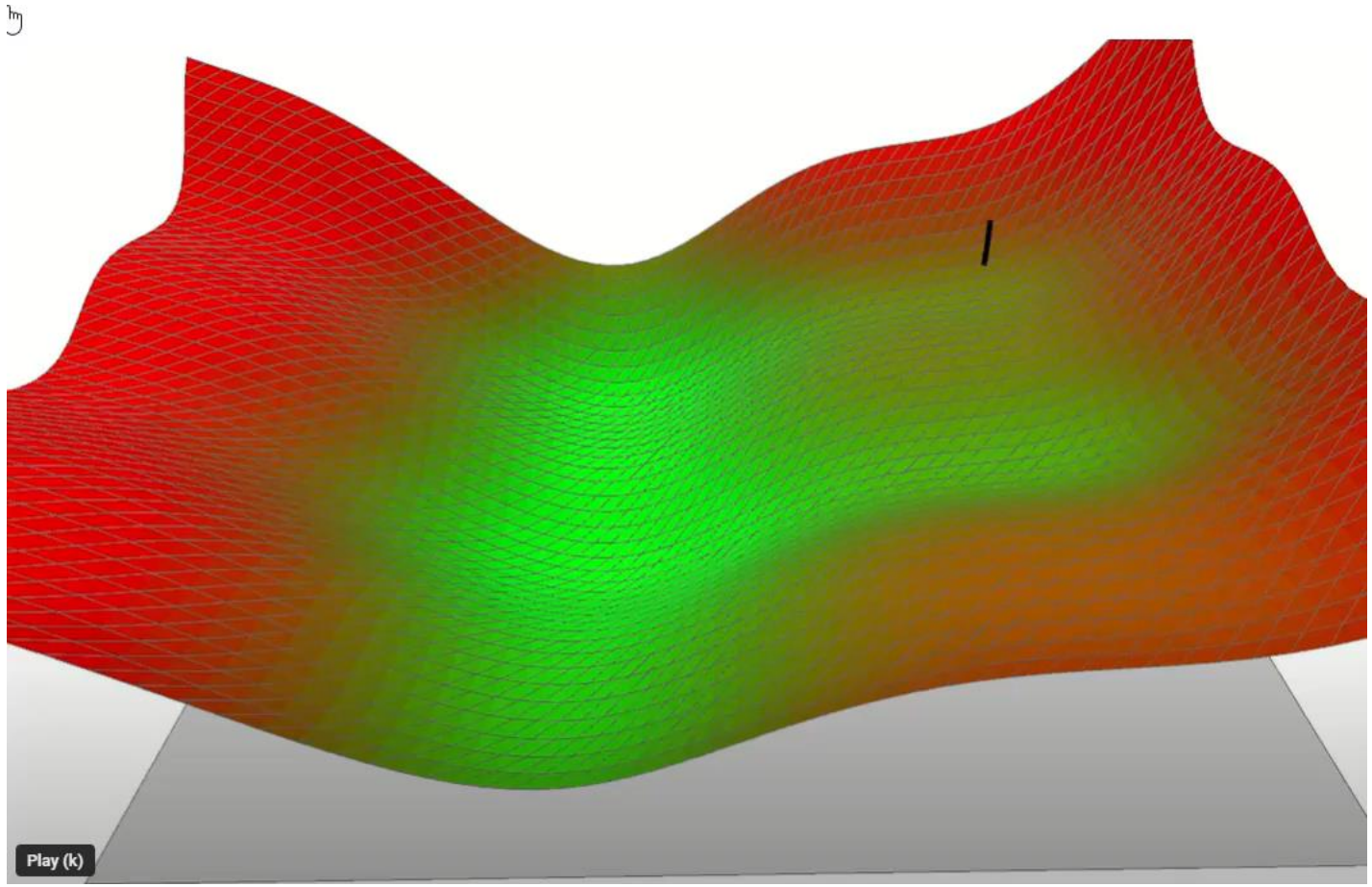
$$x^{(t+1)} = x^{(t)} - \alpha J'(x)$$

(where α is learning rate; a small positive number to control the rate of updating the optimized parameters)

- If we start with $x^{(t)} < 0$, the slope (first derivative $J'(x)$) is negative, then $-J'(x)$ is positive, then $x^{(t+1)}$ is moving to the right toward the minimum $(x = 0)$
- If we start with $x^{(t)} > 0$, the slope is positive, then $-J'(x)$ is negative, then $x^{(t+1)}$ is moving to the left toward the minimum $(x = 0)$
- If $J'(x) = 0$, the no change since we are already reached to the minimum



Visualization



Source: https://www.youtube.com/watch?v=kJgx2RcJKZY

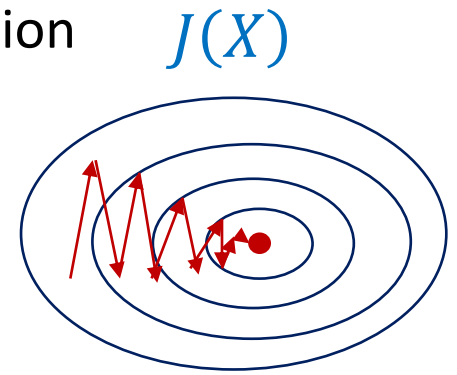
Gradient Descent with Momentum

- The gradient descent (GD) approach could experience oscillations in one direction of the parameters and slowness in the direction of other parameters.
- A momentum term is added to the normal gradient descent update term. This momentum term is the exponential weighted average of the gradient, dv, over each iteration $(dV^{(0)} = 0)$:

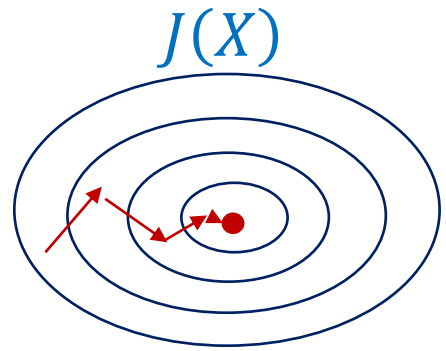
$$dV^{(t)} = \beta dV^{(t-1)} + (1-\beta)J'(X), \qquad usually\ \beta = 0.9$$

Then the parameters update as:

$$X^{(t)} = X^{(t-1)} - \alpha\, dV^{(t)}$$



GD



GD with momentum

GD with momentum shows less oscillations and takes fewer steps toward convergence

Gradient Descent for Linear Regression

Using the partial derivatives of the cost function with respect to W and b as:

$$\frac{\partial J(W,b)}{\partial w_d} = \frac{1}{N}\sum_{n=1}^{N}(\sum_{d=1}^{D} w_d x_d^{(n)} + b - y^{(n)})x_d^{(n)}$$

$$\frac{\partial J(W,b)}{\partial b} = \frac{1}{N}\sum_{n=1}^{N}(\sum_{d=1}^{D} w_d x_d^{(n)} + b - y^{(n)})$$

And by initializing W and b with random values, or zeros, then we can update the W and b at each iteration t update equation as to get new values as:

$$w_d^{[t+1]} = w_d^{[t]} - \alpha \frac{\partial J(W,b)}{\partial w_d}$$

$$b^{[t+1]} = b^{[t]} - \alpha \frac{\partial J(W,b)}{\partial b}$$

Where α is the learning rate of the gradient descent approach. Large values for α can cause the algorithm to diverge, while very small values can cause slower convergence.

The algorithm converges when there is no significant changes in $J(W,b)$ in consecutive iterations.

Example

For input data $(x_1^{(n)}, y^{(n)}), n = 1, 2, 3, 4$ where:

$$(x_1^{(1)}, y^{(1)}) = (0, 1.5), (x_1^{(2)}, y^{(2)}) = (2, 2),$$
$$(x_1^{(3)}, y^{(3)}) = (3, 3.5), (x_1^{(4)}, y^{(4)}) = (4, 4)$$

$$\frac{\partial J(W,b)}{\partial w_1} = \frac{1}{N}\sum_{n=1}^{N}(w_1 x_1^{(n)} + b - y^{(n)})x_1^{(n)}$$

$$\frac{\partial J(W,b)}{\partial b} = \frac{1}{N}\sum_{n=1}^{N}(w_1 x_1^{(n)} + b - y^{(n)})$$

$$w_i^{[t+1]} = w_1^{[t]} - \alpha\frac{\partial J(W,b)}{\partial w_1}$$

$$b^{[t+1]} = b^{[t]} - \alpha\frac{\partial J(W,b)}{\partial b}$$

$$w_1^{[0]} = 0, b^{[0]} = 0, \alpha = 0.1$$

$$\frac{\partial J(W,b)}{\partial w_1} = \frac{1}{4}(-0 * 1.5 - 2 * 2 - 3 * 3.5 - 4 * 4) = -7.625$$

$$\frac{\partial J(W,b)}{\partial b} = \frac{1}{4}(-1.5 - 2 - 3.5 - 4) = -2.75$$

$$w_1^{[1]} = 0 - 0.1 * -7.625 = 0.7625$$
$$b^{[1]} = 0.275$$

$$\frac{\partial J(W,b)}{\partial w_1} = \frac{1}{4}\begin{pmatrix}(0.7625 * 0 + 0.275 - 1.5) * 0 + \\ (0.7625 * 2 + 0.275 - 2) * 2 + \\ (0.7625 * 3 + 0.275 - 3.5) * 3 + \\ (0.7625 * 4 + 0.275 - 4) * 4\end{pmatrix} = -1.48$$

$$\frac{\partial J(W,b)}{\partial b} = \frac{1}{4}\begin{pmatrix}(0.7625 * 0 + 0.275 - 1.5) + \\ (0.7625 * 2 + 0.275 - 2) + \\ (0.7625 * 3 + 0.275 - 3.5) + \\ (0.7625 * 4 + 0.275 - 4)\end{pmatrix} = -0.76$$

$$w_1^{[2]} = 0.7625 - 0.1 * -1.48 = 0.91$$
$$b^{[2]} = 0.351$$

then after 200 iterations you get $w_1^{[200]} = 0.65833175$
$b^{[200]} = 1.267742610841535$

Example

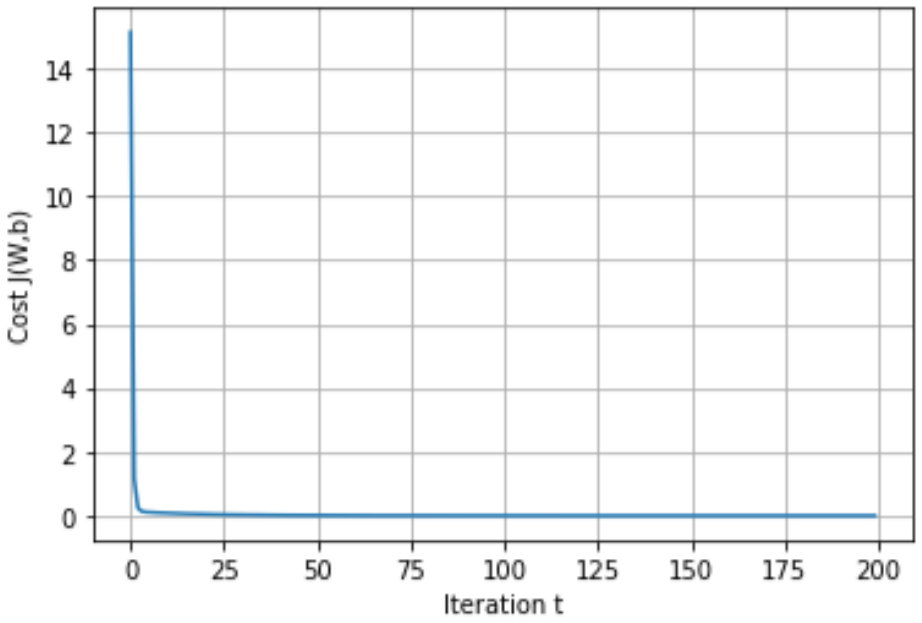
```
import numpy as np
import matplotlib.pyplot as plt
# use the input training data and the learning rate alpha, with number of iterations T
def lin_regression(X,y, alpha, T):
    # initialize W and b by zeros. You can use random values as well
    W = np.zeros((np.shape(X)[1], 1))
    b = 0
    # intialize the cost. Cost is not needed to find the optimal solution since we use
    # the gradient of it. But it can be used as stopping criterion and visulaize the error
    cost = np.zeros(T)
    for t in range(T):
        cost[t] = 0.5 * (np.sum((np.dot(X, W) + b) - y)**2) / len(y)
        # find the gradients gW and gb
        gW = np.dot(X.T,((np.dot(X, W) + b) - y)) / len(y)
        gb = (np.sum((np.dot(X, W) + b) - y)) / len(y)
        # update model parameters
        W = W - alpha * gW
        b = b - alpha * gb
        #print(W,b)
    # the cost at the final solution
    cost[-1] = 0.5 * (np.sum((np.dot(X, W) + b) - y)**2) / len(y)
    return W, b, cost
```

Effect of the learning rate on convergence

alpha = 0. 1

```
X = np.array([[0],[2],[3],[4]])
y = np.array([[1.5],[2],[3.5],[4]])
T = 200
alpha = 0.1
W,b,cost = lin_regression(X, y, alpha, T)
print('No. Iterations = %d, alpha = %.3f'%(T, alpha))
print('W = %.3f, b = %.3f'%(W, b))
print('Final Cost J(W,b) = %f'%cost[-1])
plt.plot(range(T), cost)
plt.xlabel('Iteration t')
plt.ylabel('Cost J(W,b)')
plt.grid(True)
plt.show()
```

```
No. Iterations = 200, alpha = 0.100
W = 0.658, b = 1.268
Final Cost J(W,b) = 0.000002
```

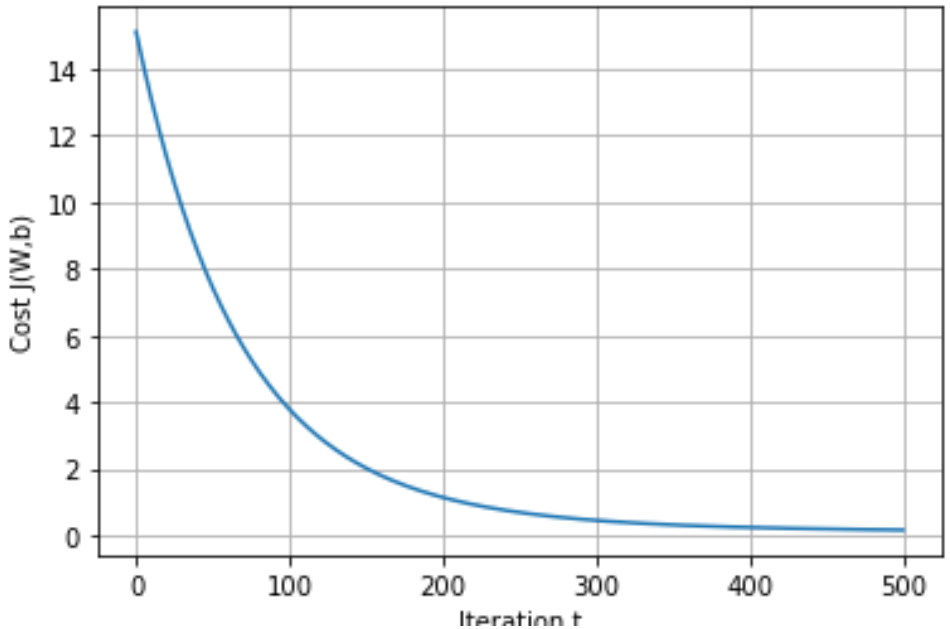


We obtained a good solution with small error

Effect of the learning rate on convergence

alpha = 0.001

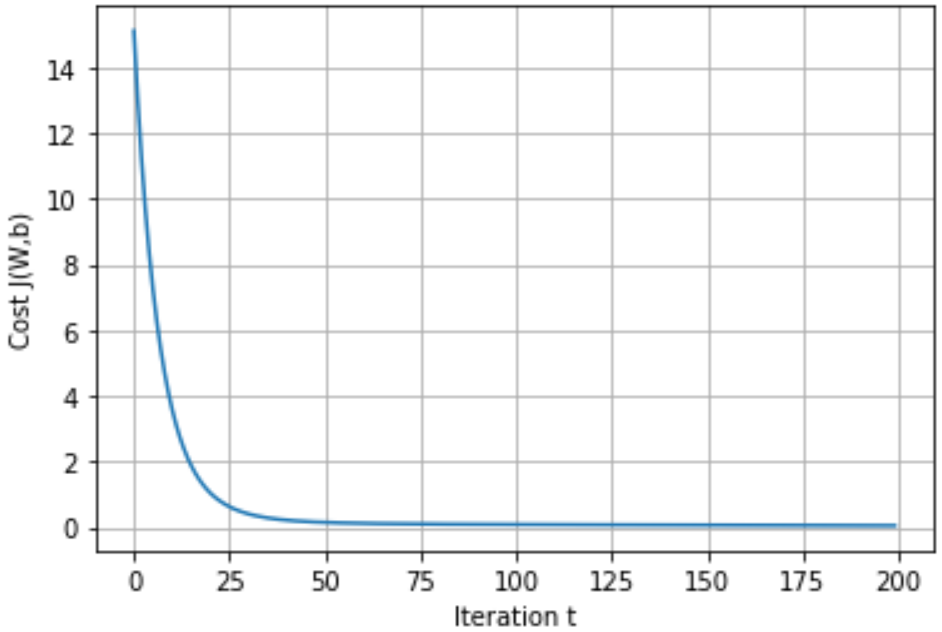
```
No. Iterations = 500, alpha = 0.001
W = 0.909, b = 0.429
Final Cost J(W,b) = 0.151072
```



Very small alpha, very slow convergence

alpha = 0.01

```
No. Iterations = 200, alpha = 0.010
W = 0.836, b = 0.717
Final Cost J(W,b) = 0.046183
```

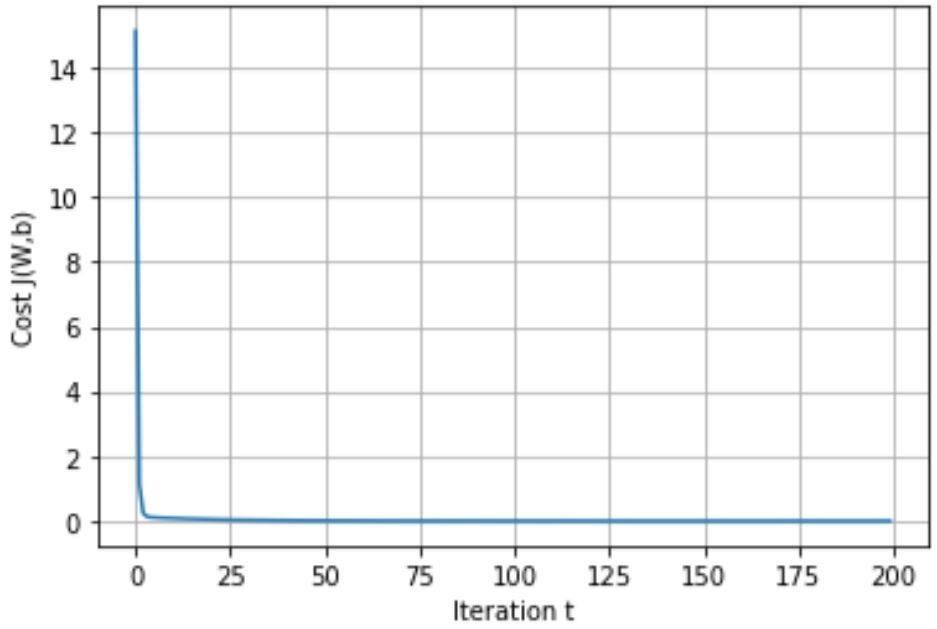


small alpha, slow convergence

Effect of the learning rate on convergence

alpha = 0. 1

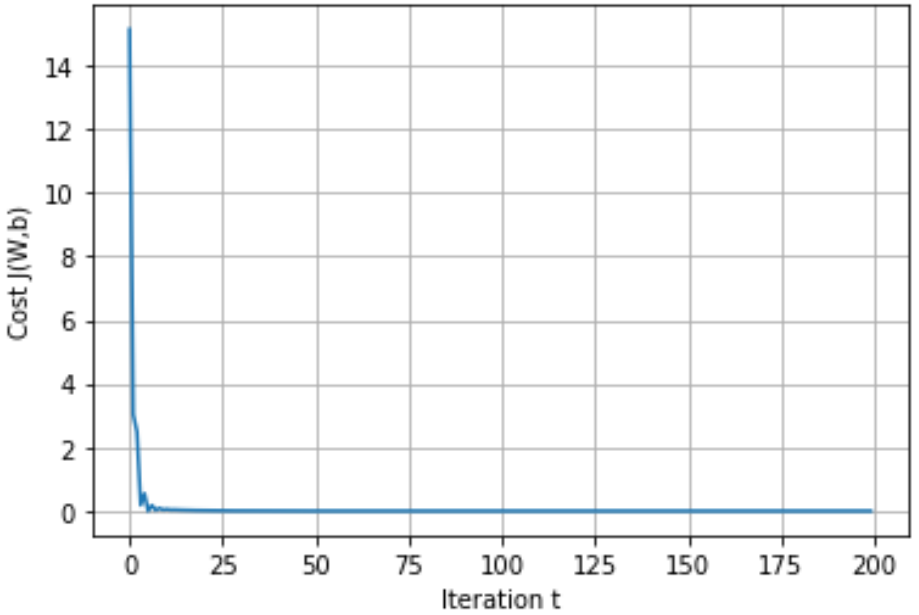
```
No. Iterations = 200, alpha = 0.100
W = 0.658, b = 1.268
Final Cost J(W,b) = 0.000002
```



Reasonable alpha, better convergence

alpha = 0.2

```
No. Iterations = 200, alpha = 0.200
W = 0.657, b = 1.271
Final Cost J(W,b) = 0.000000
```

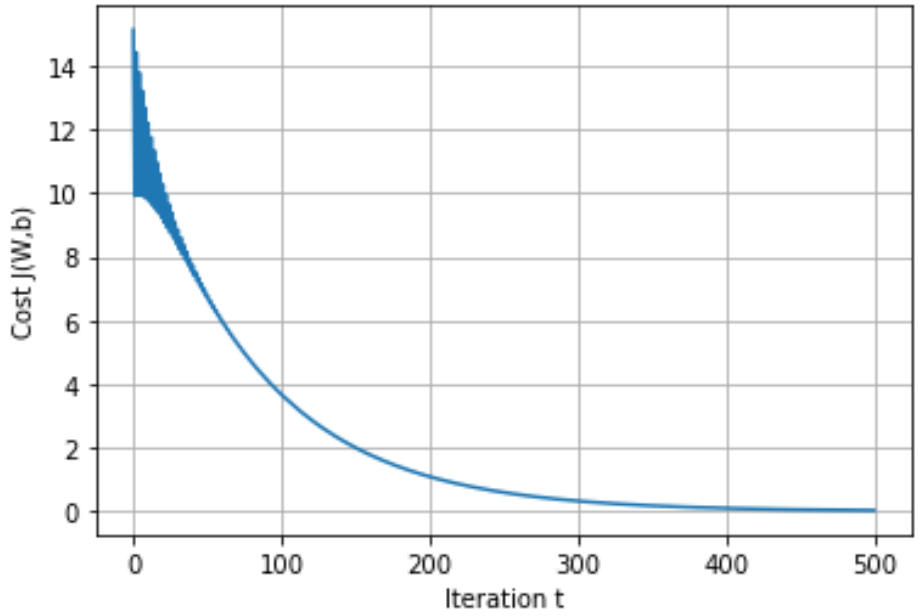


A little larger alpha, faster convergence with some oscillations

Effect of the learning rate on convergence

alpha = 0. 25

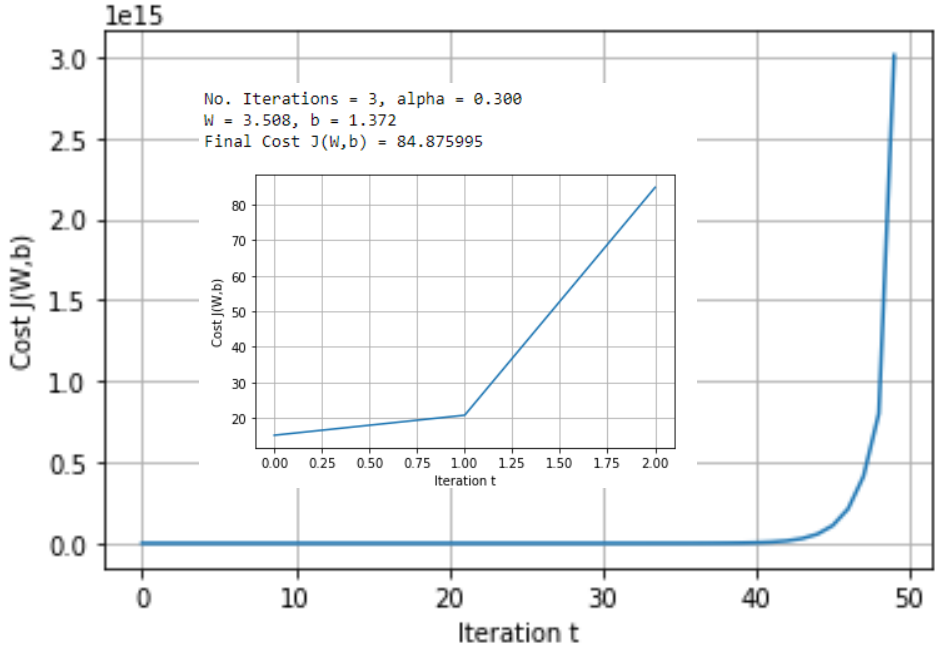
```
No. Iterations = 500, alpha = 0.250
W = 0.611, b = 1.257
Final Cost J(W,b) = 0.028135
```



Larger alpha, higher oscillations, but it converges

alpha = 0.3

```
No. Iterations = 50, alpha = 0.300
W = -15081864.330, b = -4864607.039
Final Cost J(W,b) = 3010694466016436.000000
```



Very large alpha, diverges and can cause overflow

Normal Equations versus Gradient Descent

Normal Equations	Gradient descent
+ Non-iterative	- Iterative and depends on the initial guess
- Needs matrix inversion • Computationally expensive for large matrices (size >1000 x 1000) • Some matrices are not invertible in some cases such as features are not independent	+ No matrix inversion, computationally attractive
+ No controlling Parameters	- Convergence is affected by the value of the learning rate
+ No need for stopping criterion	- Needs stopping criterion, affects the convergence
- Is not extendable to other machine learning approaches	+ Can be extended to other machines learning approaches and it is a commonly used approach

scikit-learn *linear_model.LinearRegression()*

Use linear_model.LinearRegression() to initialize an instance of the linear model with parameters:

fit_intercept : *bool, default=True*

If True, find the intercept, bias term, b of the line. Otherwise, data assumed to be centered (i.e. b = 0)

Attribute coef_

Returns the estimated coefficients *W*

Attribute intercept_

Returns the estimated intercept *b*

For more description and examples: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

scikit-learn *linear_model.LinearRegression()*

Method fit(X_train, Y_train)

finds the linear model

Method predict(X_test)

Use the model to predict the output values

Method Score(Y_test, Y_pridect)

Returns the coefficient of determination R^2 score as:

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

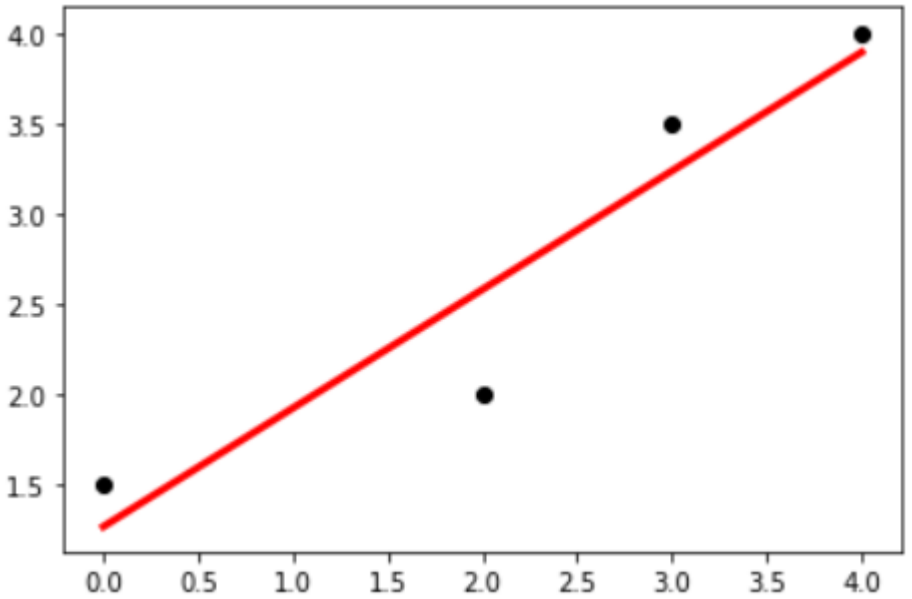
best value is 1 when $y = \hat{y}$, and it is zero when the model always predict $\hat{y}$ as $\bar{y}$ regardless of the input. It assesses the goodness of fit of a linear regression model. It quantifies the proportion of the variance in the dependent variable (the variable you're trying to predict) that is explained by the independent variable(s) in the model.

Code

In [10]:
```
from sklearn import linear_model
def linear_regression(X_train, y_train, X_test):
    line = linear_model.LinearRegression()
    line.fit(X_train, y_train)
    y_pred = line.predict(X_train)
    plt.scatter(X_train[:, 0], y_train,  color = 'black')
    plt.plot(X_train[:, 0], y_pred, color = 'red', linewidth = 3)
    print('W: ',line.coef_)
    print('b: ',line.intercept_)
    print('Test:', X_test,'\nPredicted:', line.predict(X_test))
    print('R^2: ', line.score(y_train, y_pred))
```

In [11]:
```
linear_regression(X, y, [[1],[5]])
```

```
W:  [[0.65714286]]
b:  [1.27142857]
Test: [[1], [5]]
Predicted: [[1.92857143]
 [4.55714286]]
R^2:  0.7142857142857144
```



Thank You!