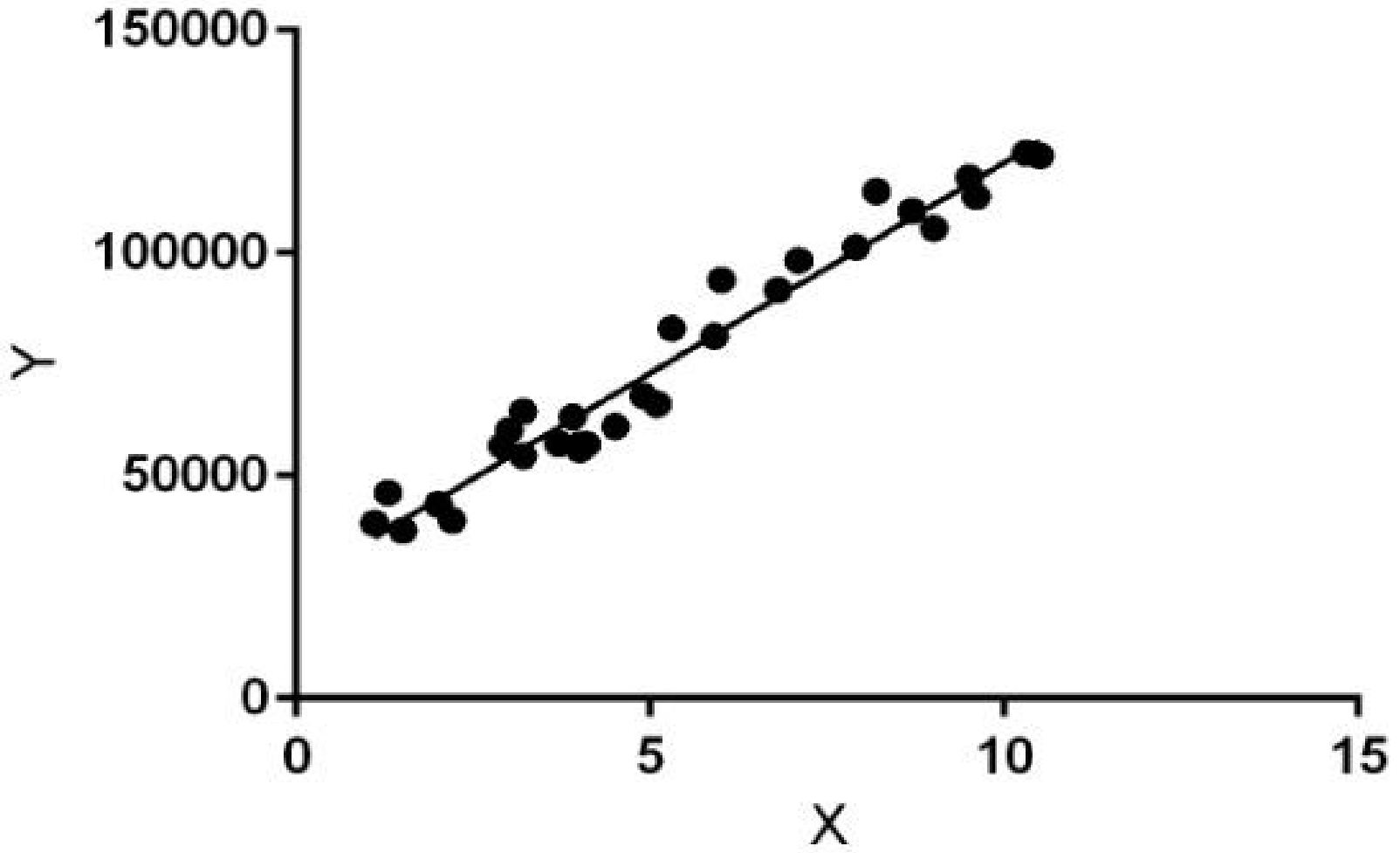
Supervised:Linear Regession

Linear Regression



Linear Regression

Training set of housing prices (Portland, OR)

Size in feet^2 (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
...	...

m = 47

Notation:

- **m** = Number of training examples
- **x**'s = "input" variable / features
- **y**'s = "output" variable / "target" variable

$x^{(1)} = 2104$

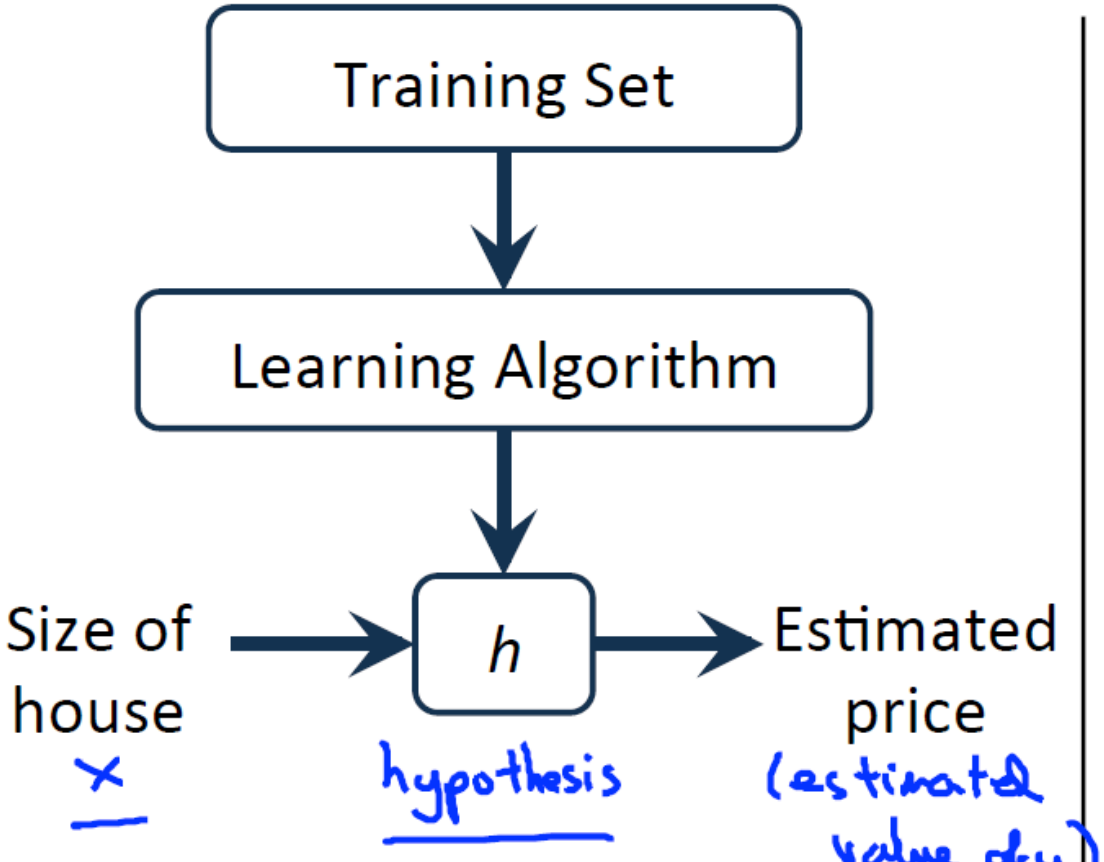
$x^{(2)} = 1416$

$y^{(1)} = 460$

(x, y) - one training example

$(x^{(i)}, y^{(i)})$ - i^{th} training example

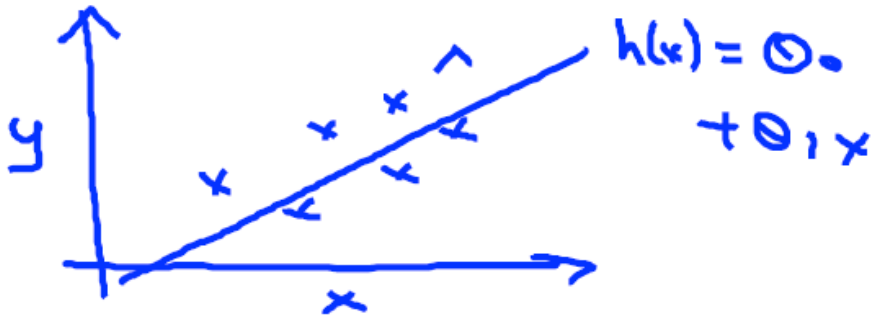
Linear Regression



How do we represent *h* ?

$h_\theta(x) = \theta_0 + \theta_1 x$

Shorthand: $h(x)$



Linear regression with one variable. (x)

Univariate linear regression.

└ one variable

Linear Regression:Cost Function

Training Set

Size in feet² (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
…	…

m = 47

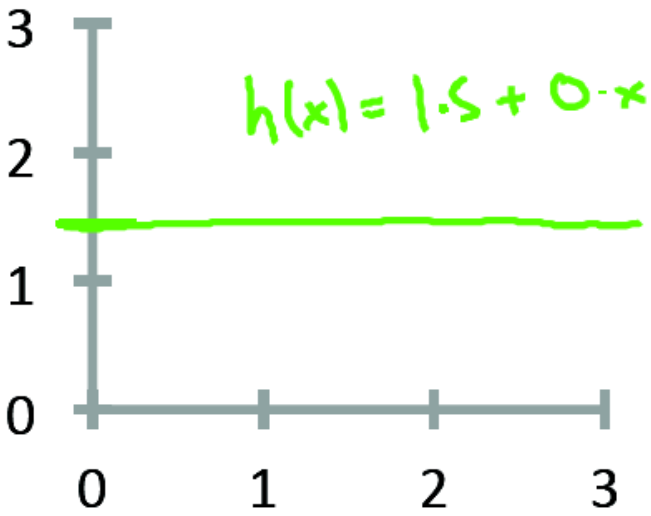
Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

θ_i's: Parameters

How to choose θ_i's ?

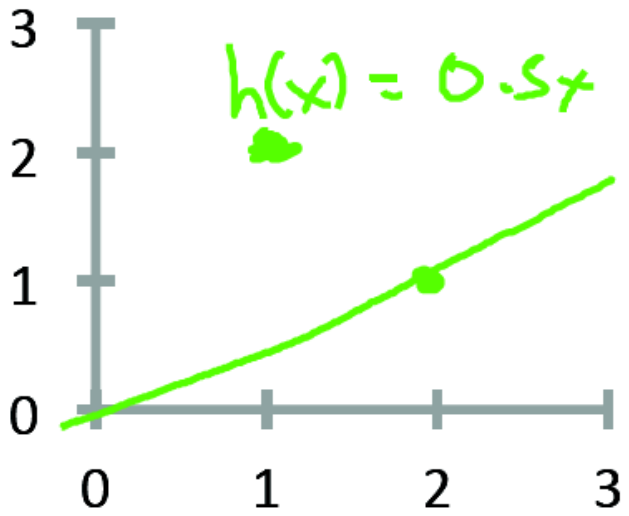
Linear Regression:Cost Function

$h_\theta(x) = \theta_0 + \theta_1 x$



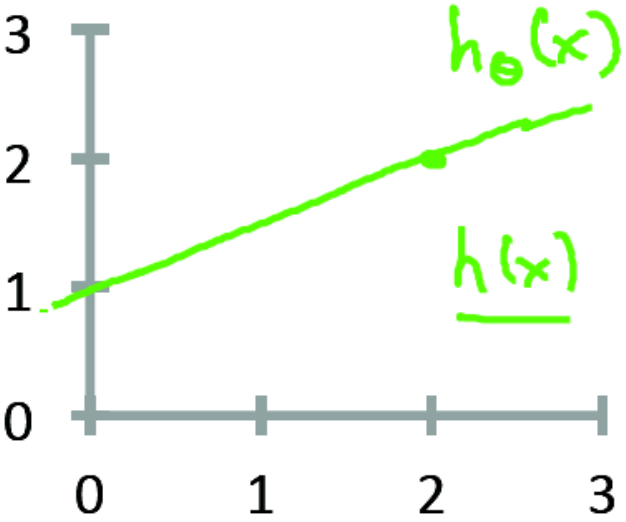
→ $\theta_0 = 1.5$

→ $\theta_1 = 0$



→ $\theta_0 = 0$

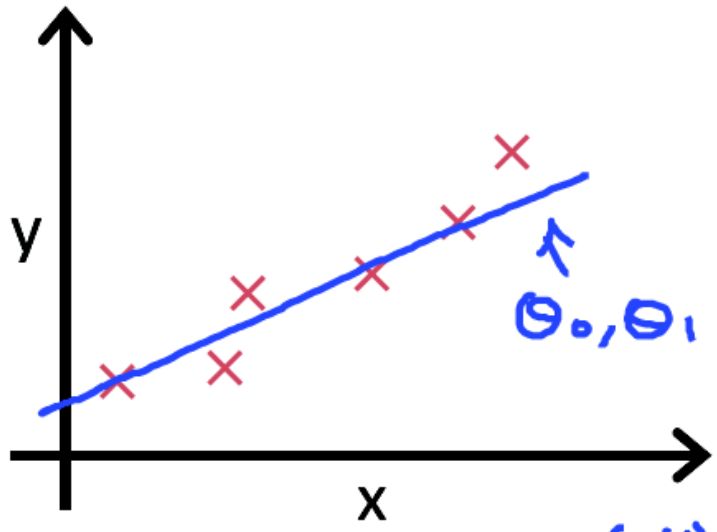
→ $\theta_1 = 0.5$



→ $\theta_0 = 1$

→ $\theta_1 = 0.5$

Linear Regression:Cost Function



#training examples

minimize θ_0, θ_1 $\frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

$h_\theta(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$

$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

Idea: Choose θ_0, θ_1 so that $h_\theta(x)$ is close to y for our training examples (x, y)

x, y

Minimize θ_0, θ_1 $J(\theta_0, \theta_1)$

Cost function

Squared error function

Linear Regression:Cost Function:intuitively

Hypothesis:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Parameters:

$$\theta_0, \theta_1$$

Cost Function:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

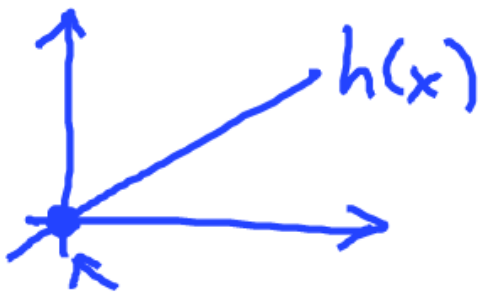
Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

Simplified

$$h_\theta(x) = \theta_1 x$$

$\theta_0 = 0$

θ_1



$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

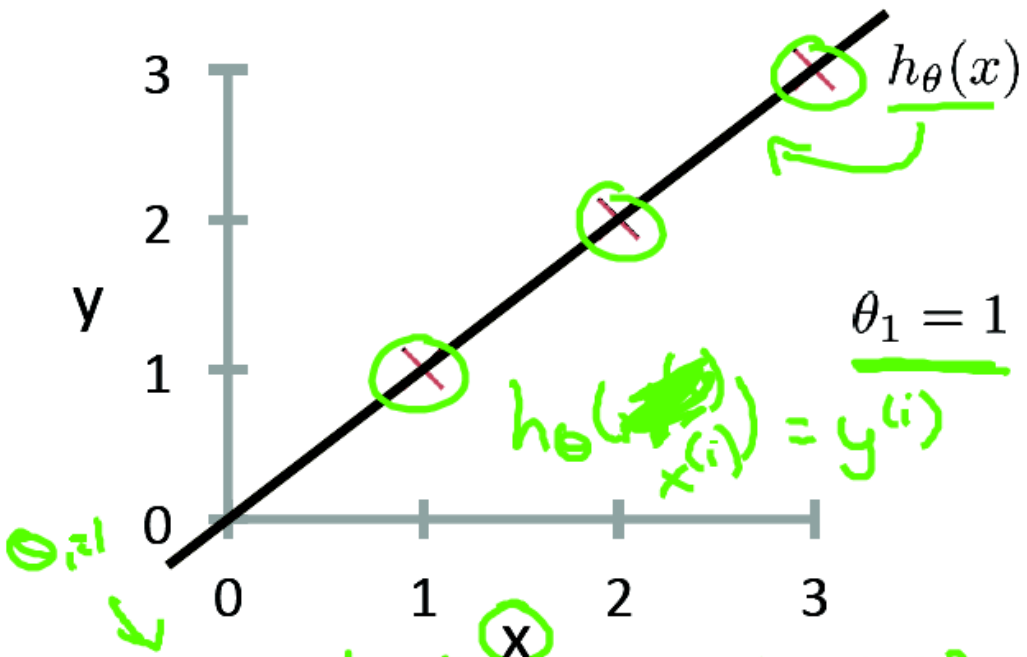
$\theta_1, x^{(i)}$

$$\underset{\theta_1}{\text{minimize}}\ J(\theta_1)$$

Linear Regression:Cost Function:intuitively

→ $h_\theta(x)$

(for fixed θ_1, this is a function of x)



$h_\theta(x^{(i)}) = y^{(i)}$

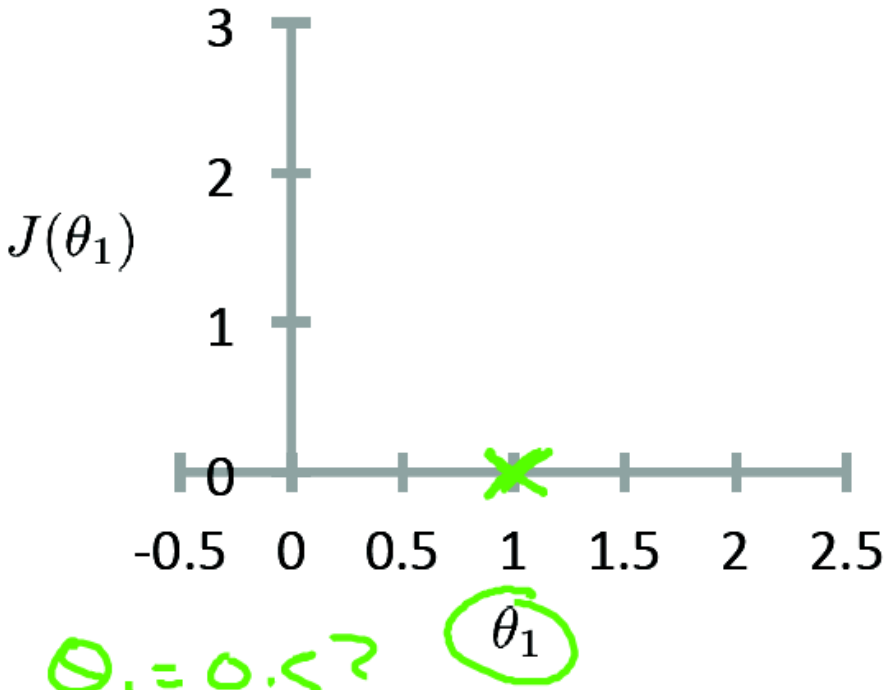
$\theta_1 = 1$

$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

$= \frac{1}{2m} \sum_{i=1}^{m} (\theta_1 x^{(i)} - y^{(i)})^2 = \frac{1}{2m}(0^2 + 0^2 + 0^2) = 0$

→ $J(\theta_1)$

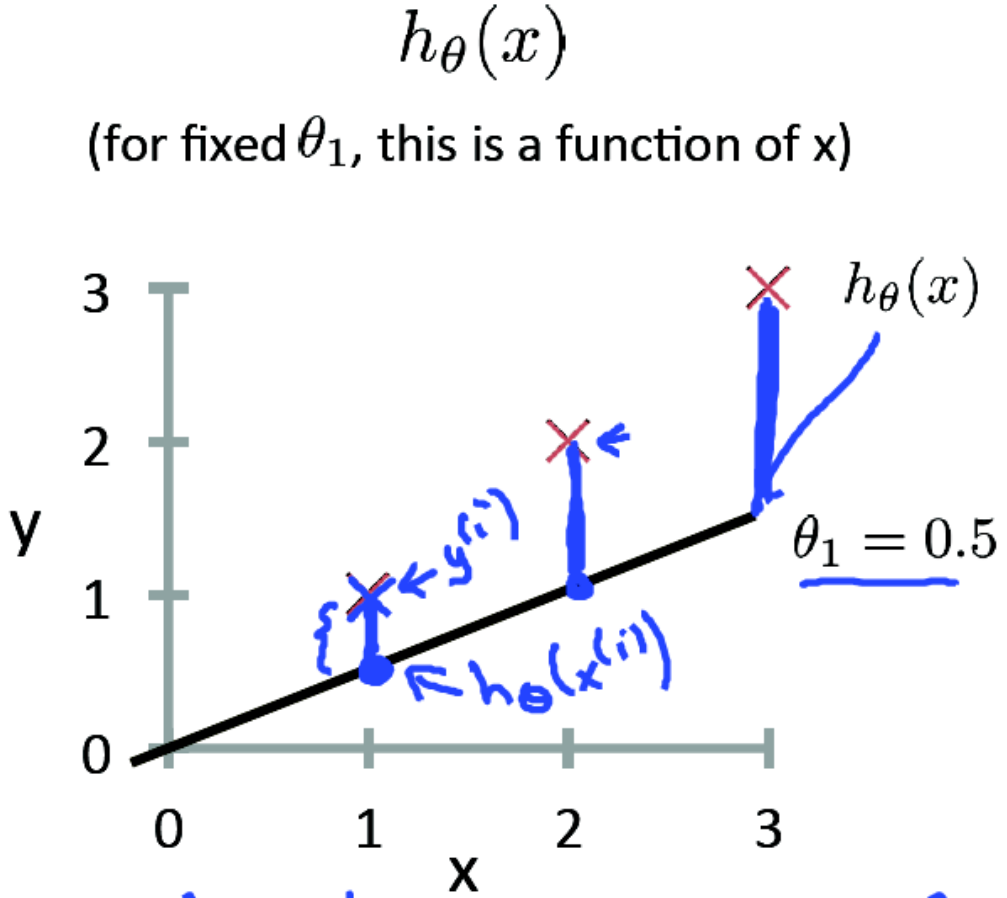
(function of the parameter θ_1)

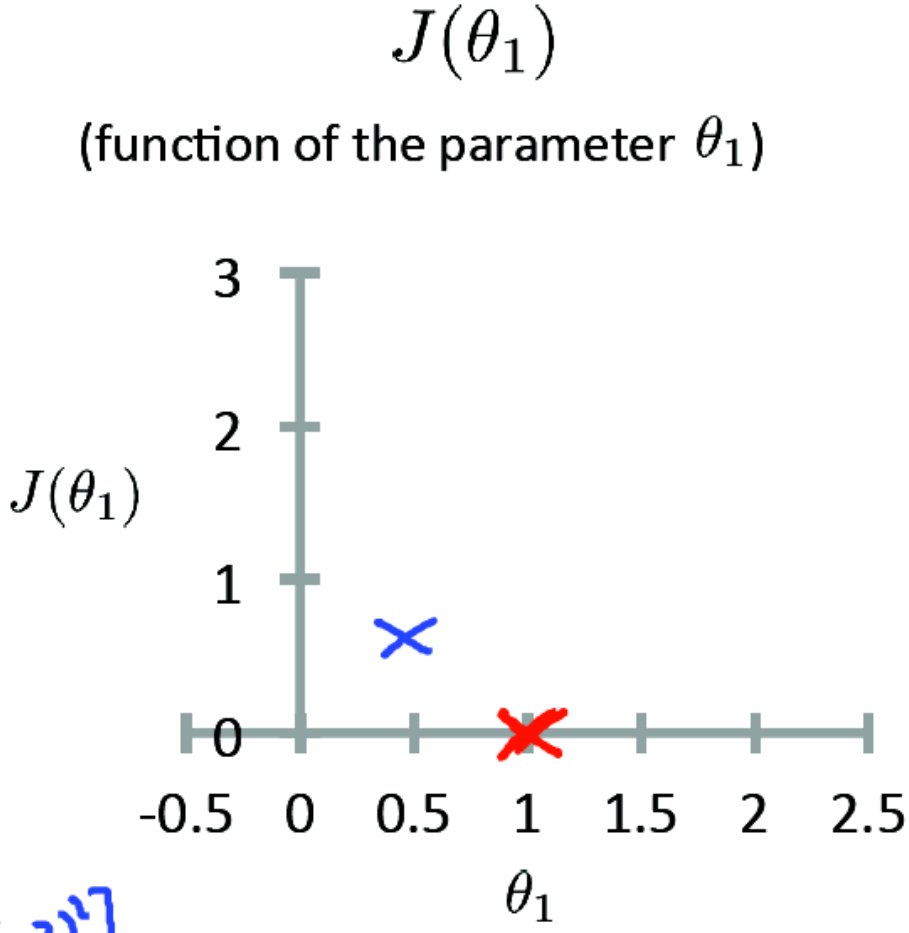


$\theta_1 = 0.5$?

$J(1) = 0$

Linear Regression:Cost Function:intuitively





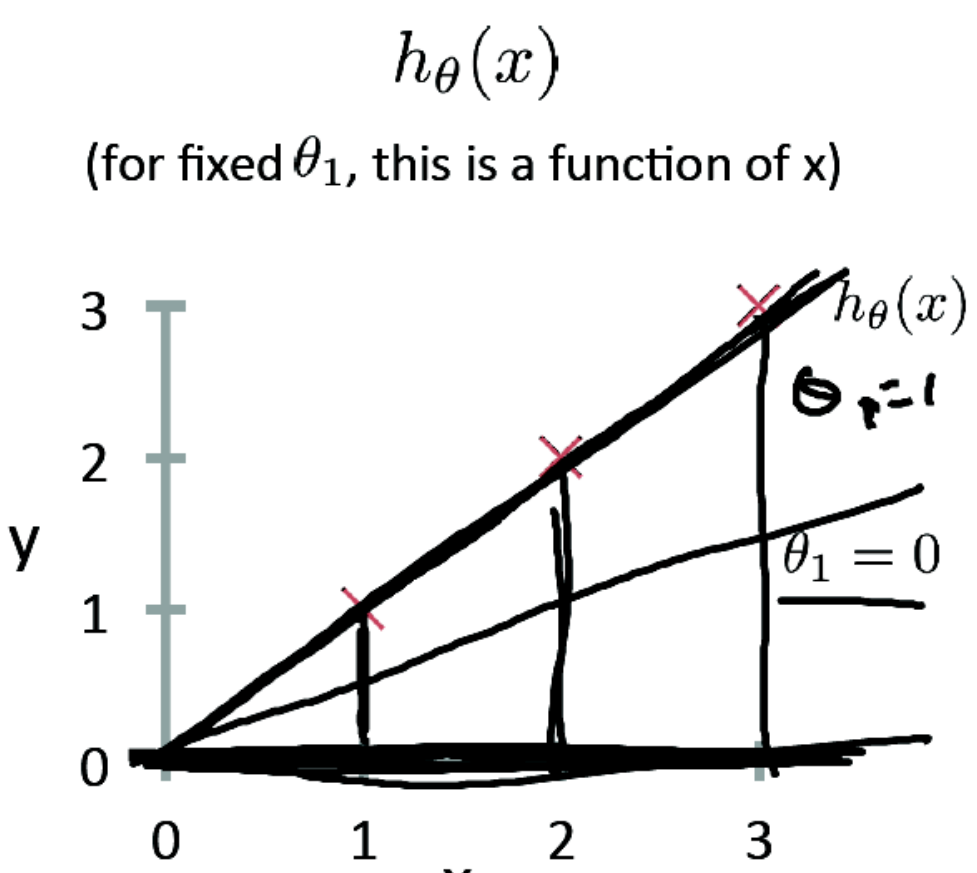
$J(0.5) = \frac{1}{2m}\left[(0.5-1)^2 + (1-2)^2 + (1.5-3)^2\right]$

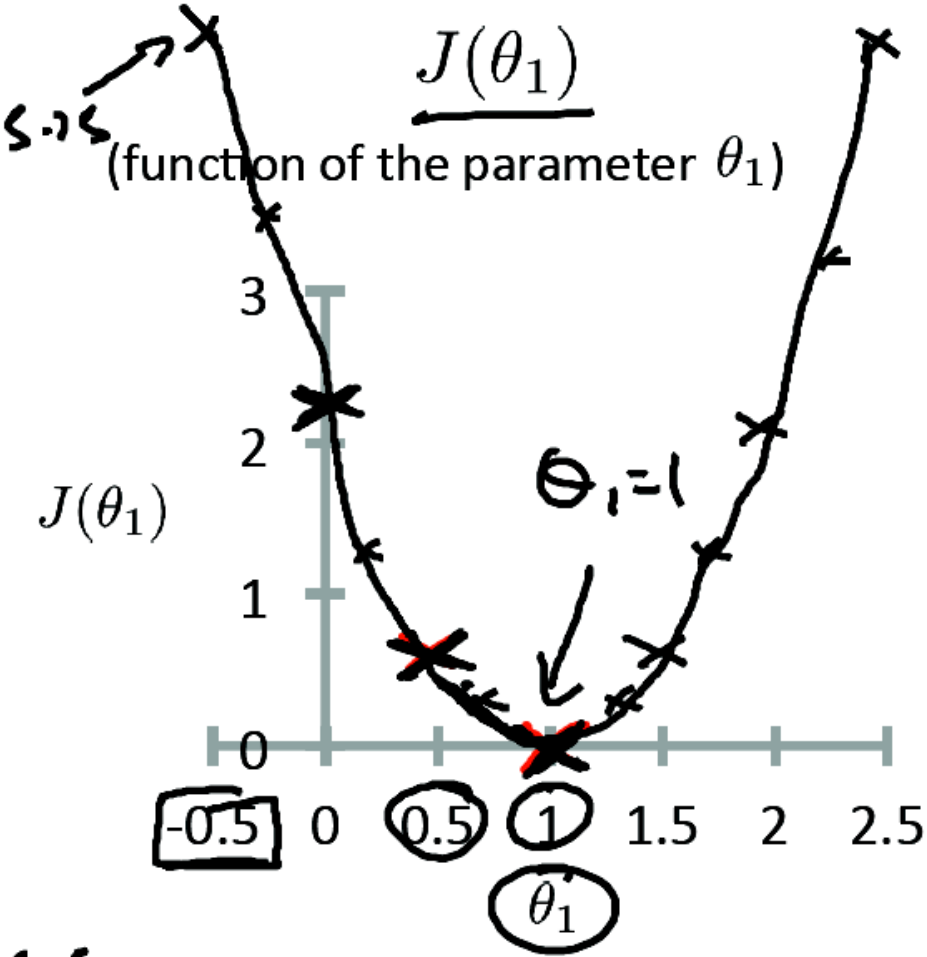
$= \frac{1}{2\times 3}(3.5) = \frac{3.5}{6} \approx 0.58$

$\theta_1 = 0?$

$J(0) = ?$

Linear Regression:Cost Function:intuitively





$$J(0) = \frac{1}{2m}(1^2 + 2^2 + 3^2)$$
$$= \frac{1}{6} \cdot 14 \approx 2.3$$

$h(x) = -0.5x$

Minimize $J(\theta_1)$
θ_1

Linear Regression:Gradient Descent

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

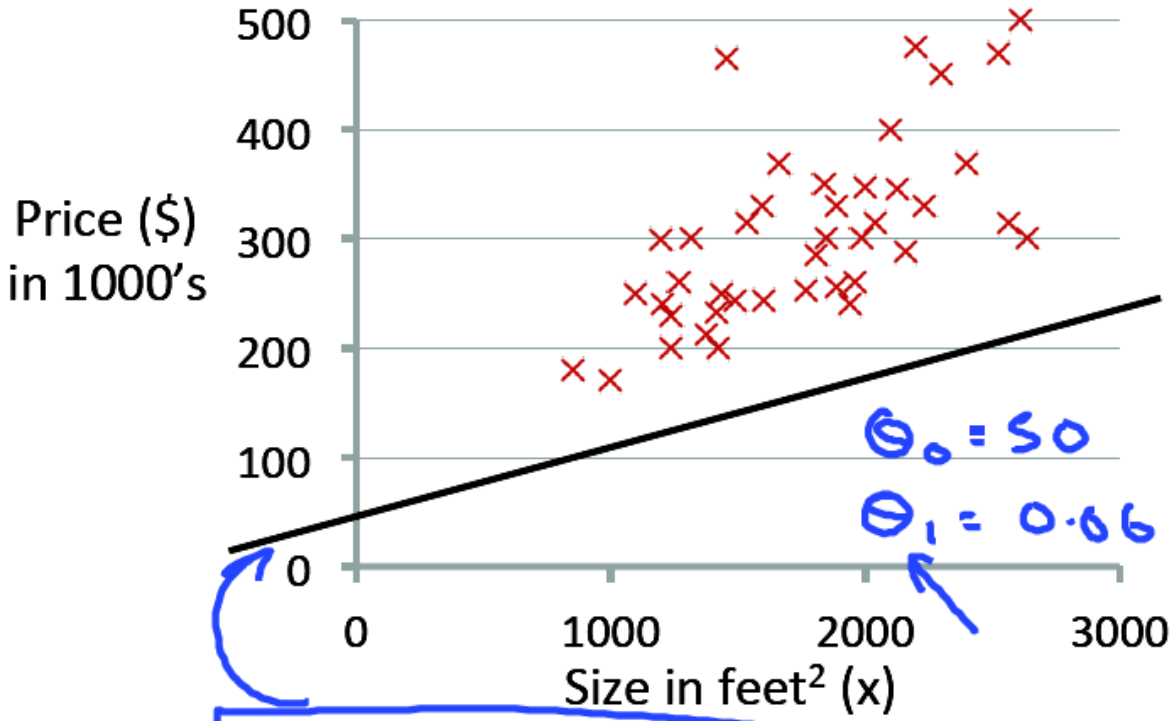
Cost Function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

Linear Regression:Gradient Descent

$h_\theta(x)$

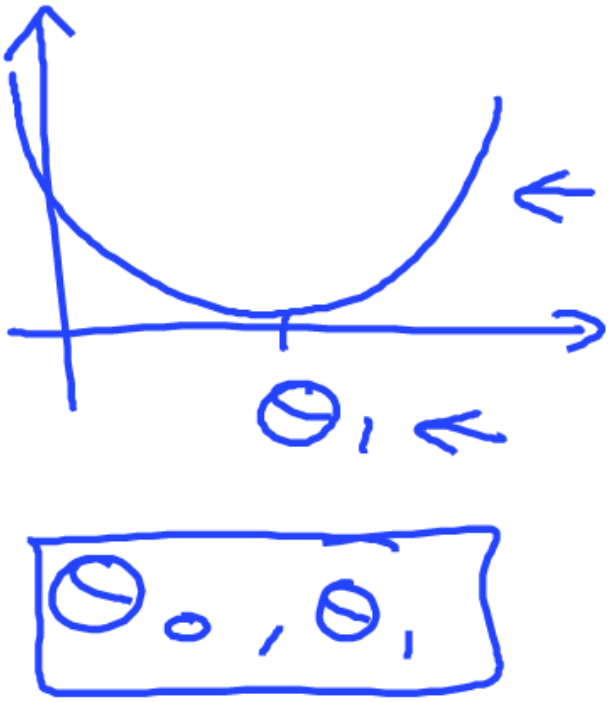
(for fixed θ_0, θ_1, this is a function of x)



$h_\theta(x) = 50 + 0.06x$

$J(\theta_0, \theta_1)$

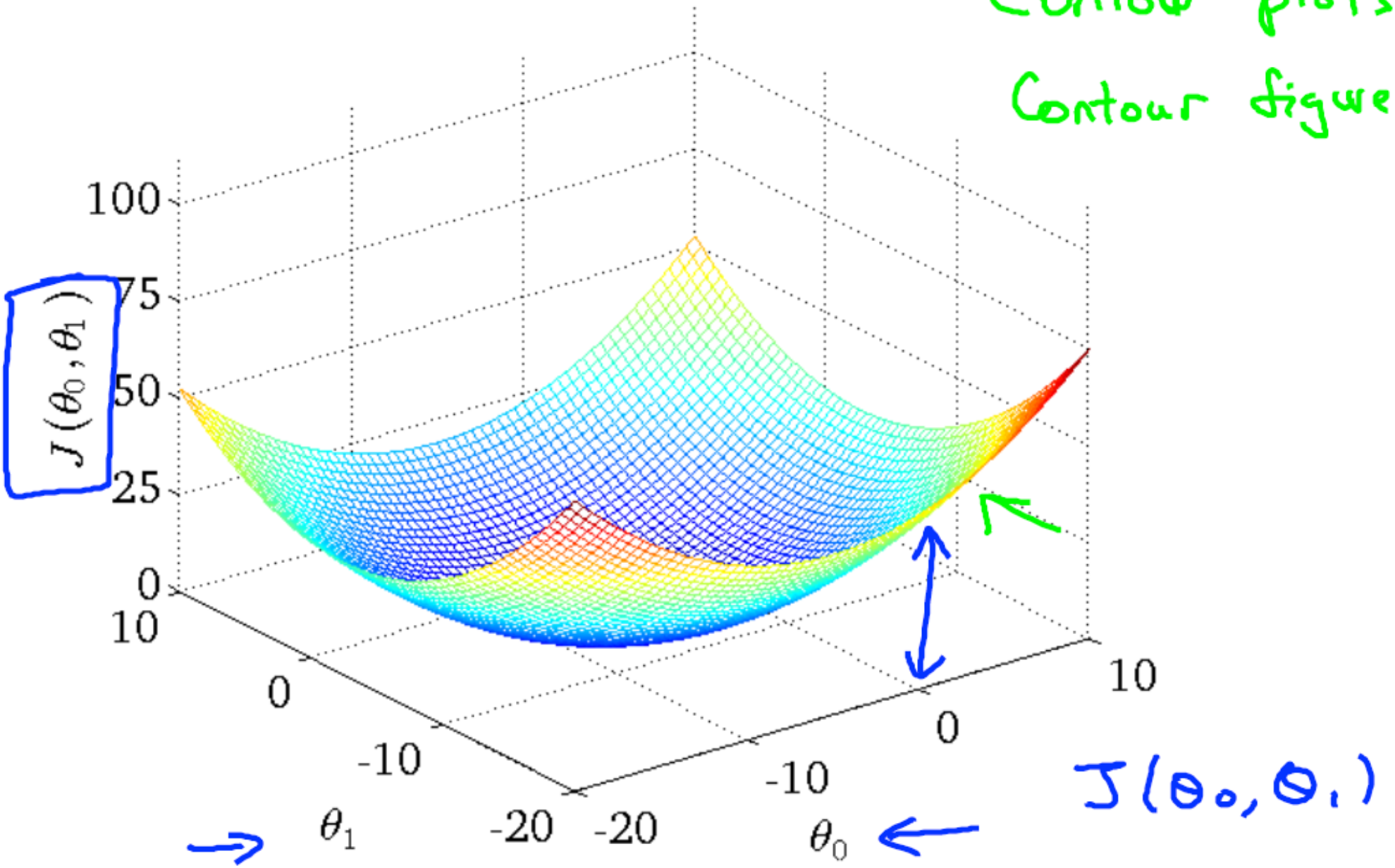
(function of the parameters θ_0, θ_1)



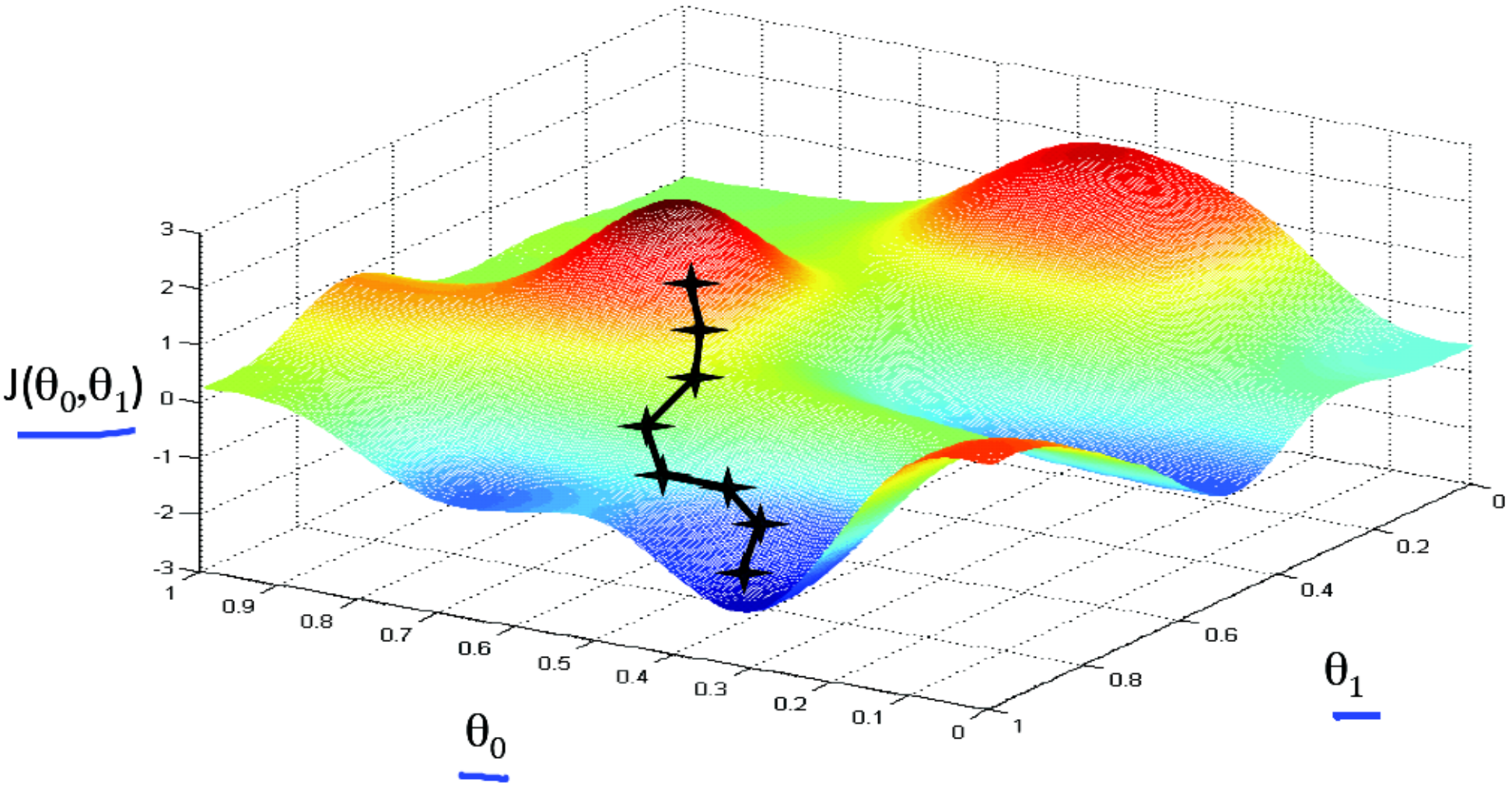
Linear Regression:Gradient Descent

Contour plots

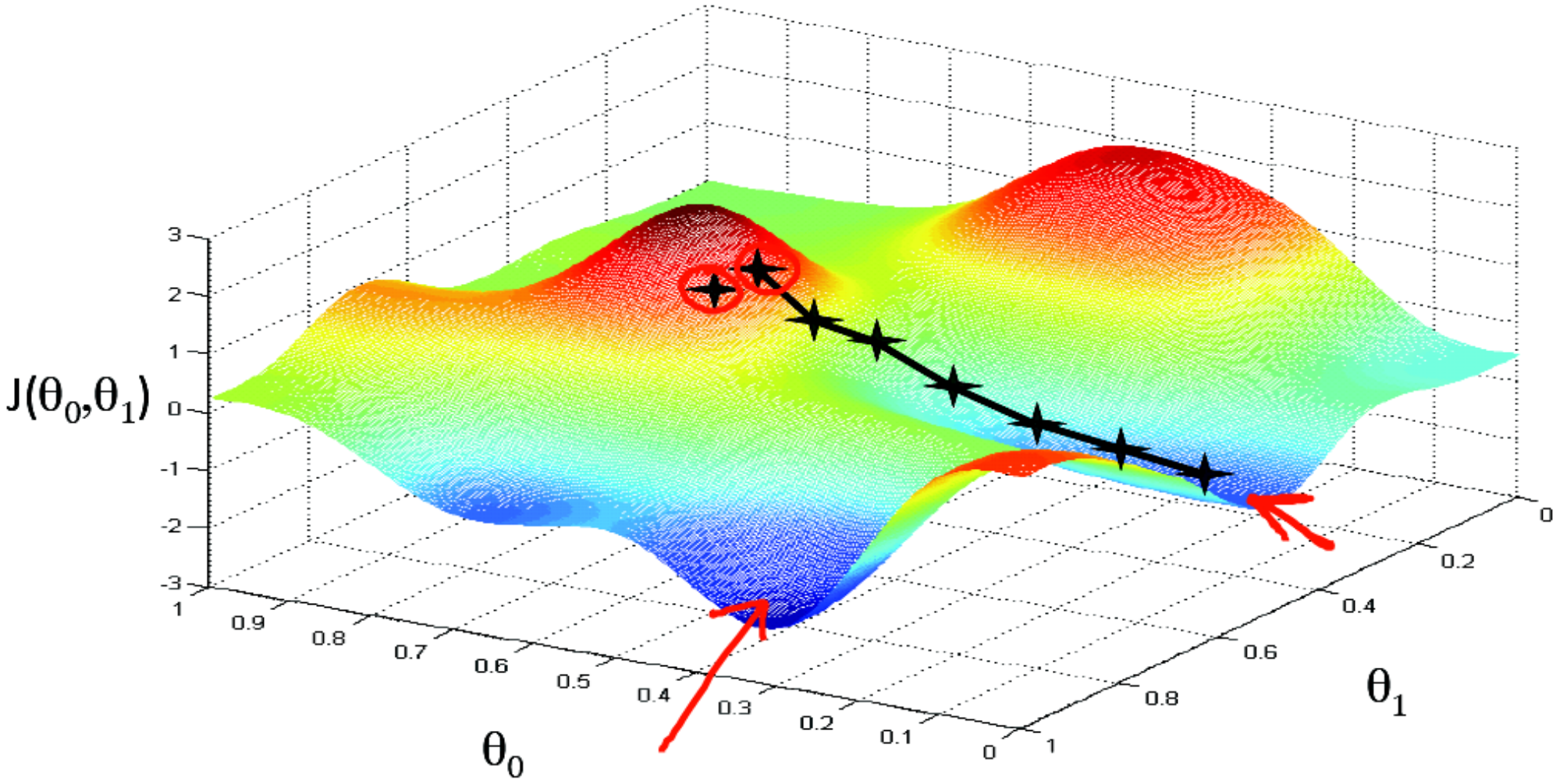
Contour figures -



Linear Regression:Gradient Descent



Linear Regression:Gradient Descent



Linear Regression:Gradient Descent Algorithm

hypothesis $\Rightarrow$ $h(x) = \theta_0 + \theta_1 x$

cost $\Rightarrow$ $J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2$

Gradient Descent $\Rightarrow$ repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

for $j = 1$, $j = 0$

}

Linear Regression:Gradient Descent Algorithm

1. $J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h(x)^{(i)} - y^{(i)})^2 \Rightarrow \frac{1}{m} \sum_{i=1}^{m} (f(\theta_0, \theta_1)^{(i)})^2$

2. $f(\theta_0, \theta_1)^{(i)} = \theta_0 + \theta_1 x - y$

3. $J(f(\theta_0, \theta_1)^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x - y^{i})^2$

4. $\frac{\partial}{\partial \theta_0} (J\theta_0 \theta_1) = \frac{\partial}{\partial_0} \frac{1}{m} \sum_{i=1}^{m} (f(\theta_0, \theta_1)^{i})^2 = 2/m \sum_{i=1}^{m} (f(\theta_0, \theta_1)^{(i)})^{(2-1)}$

- treating function f as opaque

- It is like finding a chain of derivatives. The first one is the cost function and the second one is the hypothesis.

Linear Regression:Gradient Descent Algorithm

5. $$\frac{\partial}{\partial\theta_0} f(\theta_0,\theta_1)^i = \frac{\partial}{\partial\theta_0}(\theta_0+\theta_1 x^{(i)}-y^{(i)})$$

$$= \frac{\partial}{\partial\theta_0}(\theta_0 + [a\ num][a\ num] - [a\ num]$$

$$= \theta_0' = 1\times\theta^{1-1} = 1$$

6 $$\frac{\partial}{\partial\theta_0} g(f(\theta_0\theta_1)^{(i)}) = \frac{\partial}{\partial\theta_0} g(\theta_0\theta_1) f(\theta_0\theta_1)$$

$$= 2/m \sum_{i=1}^{m} (\theta_0+\theta_1 x^{(i)}-y^{(i)}) \times 1$$

Linear Regression:Gradient Descent Algorithm

With respect to θ_1

7. $$\frac{\partial}{\partial \theta_1} f(\theta_0 \theta_1)^i = \frac{\partial}{\partial \theta_1} (\theta_0 + \theta_1 x - y)$$

$$= \frac{\partial}{\partial \theta_1} ([\text{a num}] + \theta_1 [\text{a num}] - [\text{a num}])$$

$$= 0 + \theta_1 x^i - 0$$

$$= 0 + 1 \times \theta_1^{i-1} \times x^i - 0$$

$$= x^i$$

8. $$\frac{\partial}{\partial \theta_1} g(f(\theta_0 \theta_1)^i) = \frac{\partial}{\partial \theta_1} g(\theta_0 \theta_1) \frac{\partial}{\partial \theta_1} f(\theta_0 \theta_1)^i$$

$$= \boxed{2/m \sum_{i=1}^{m} (\theta_0 + \theta_1 x^i - y^i)} \times \boxed{x^i}$$

Linear Regression:BatchGradientDescent

$$J(\theta_0, \theta_1) = 1/m \sum_{i=1}^{m} (h(x)^i - y^i)^2$$

$$= 1/m \sum_{i=0}^{m} (\theta_0 + \theta_1 x^i - y^i)^2$$

$$\partial/\partial_{\theta_1} J(\theta_0, \theta_1) = 2/m \sum_{i=1}^{m} (\theta_0 + \theta_1 x^i - y) \cdot x^i$$

$$\theta_j := \theta_j - \alpha \, \partial/\partial_{\theta_j} J(\theta_0, \theta_1)$$

```
def costfunc(X,theta):
    m = len(X)
    y_predict = X.dot(theta)
    return np.sum(((y-y_predict)**2)/m)

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

def gradient(X,theta):
    m = len(X)
    return 2/m * X_b.T.dot(X_b.dot(theta) - y)

def plot_gradient_descent(theta, eta, theta_path=None):
    m = len(X_b)
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < iterations_to_track:
            y_predict=thetaTransposeX(X_b,theta)
            cost=costfunc(X_b,theta)
            if theta_path_bgd_10 is not None:
                theta_path_bgd_10.append(theta[0][0])
            if cost_path_bgd_10 is not None:
                cost_path_bgd_10.append(cost)
            if y_predict_bgd_10 is not None:
                y_predict_bgd_10.append(y_predict)
        gradients = gradient(X_b,theta)
        theta = theta - eta * gradients
    print("theta best:",theta)

rnd.seed(42)
theta = rnd.randn(1,1)  # random initialization
plot_gradient_descent(theta, eta=0.1)
minx=0.0
maxx=10.0
```

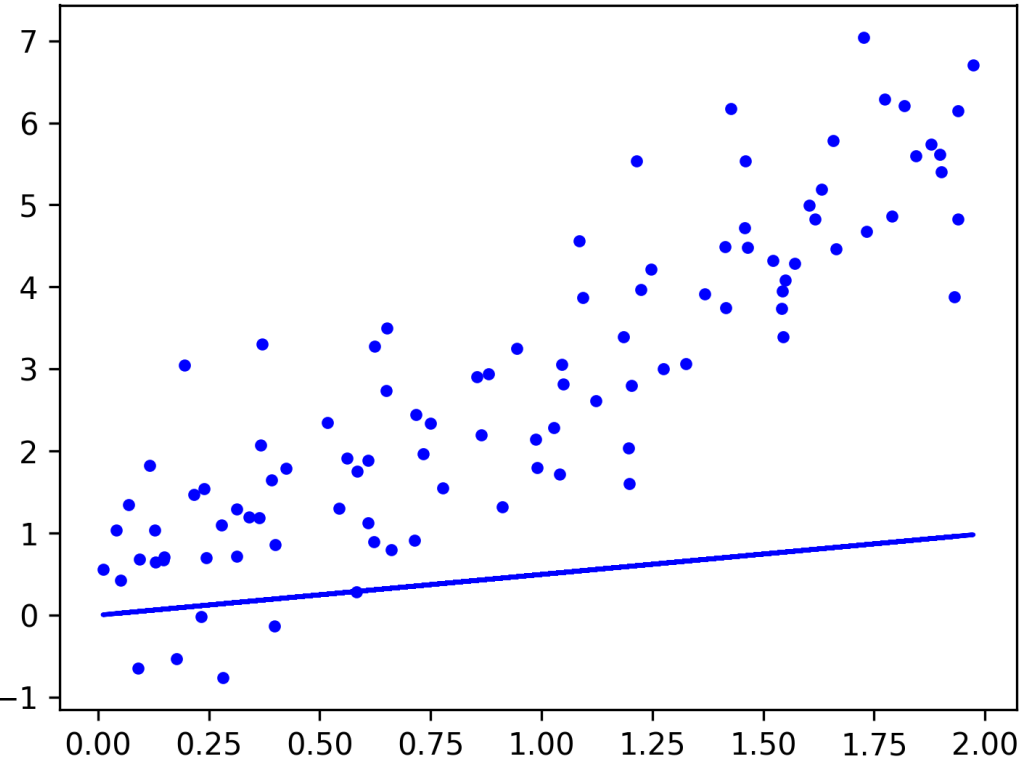
Linear Regression:BatchGradientDescent

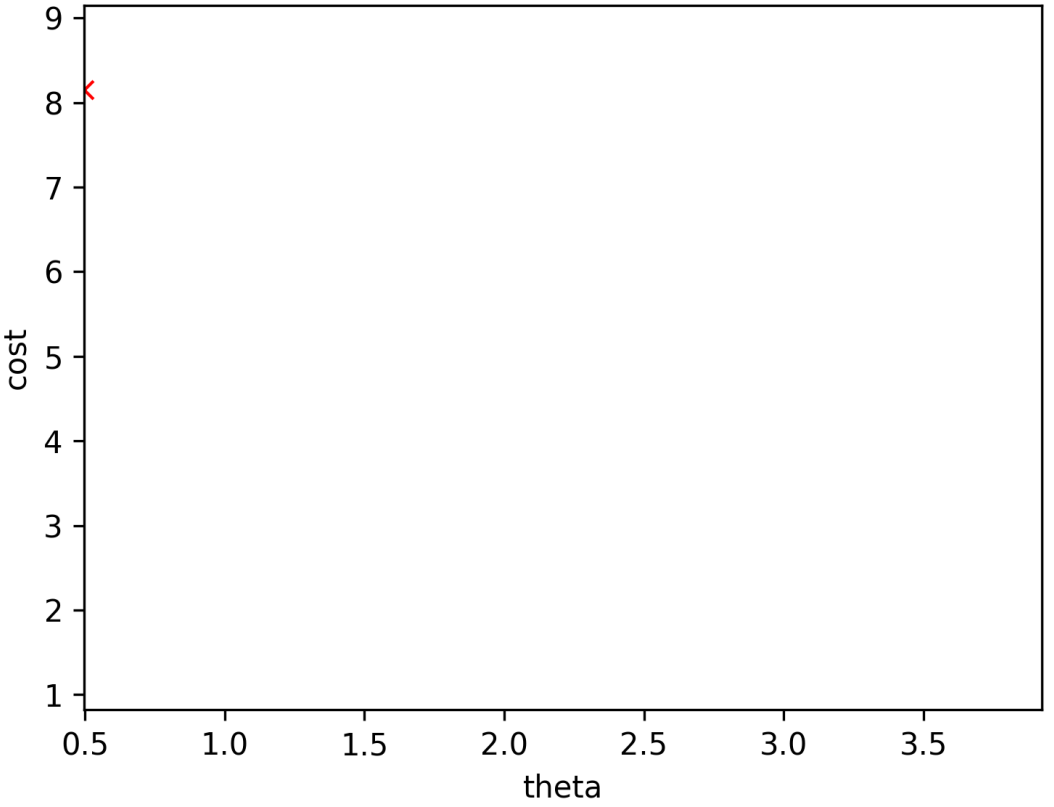
```
def plot_progress(xvec,ttxvec,theta,cost):
    fig = plt.figure(figsize=(10, 4))
    sub1 = plt.subplot(1, 2, 1)
    sub2 = plt.subplot(1, 2, 2)
    style = "b-"
    sub1.plot(X, y, "b.")
    xmin=np.amin(theta)
    xmax=np.amax(theta)+1
    ymin=np.amin(cost)
    ymax=np.amax(cost)+1
    for i,ttx in enumerate(ttxvec):
        sub1.plot(X, ttx.T[0], style)
        plt.xlabel("theta")
        plt.ylabel("cost")
        axes = plt.gca()
        axes.set_xlim([xmin,xmax])
        axes.set_ylim([ymin,ymax])
        sub2.plot(theta[i], cost[i], "rx")
        fl.save_fig("progress"+str(i))
```

plots figure-1
1. x against y
2. and overlays predicted y.

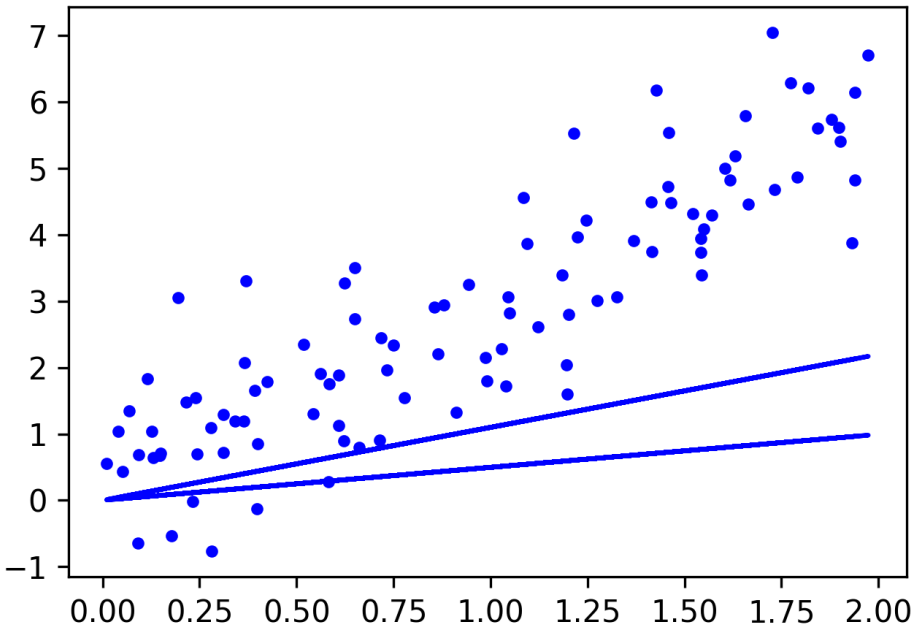
plots figure-2
1. theta against cost.
2. easier to plot because θ is scalar.

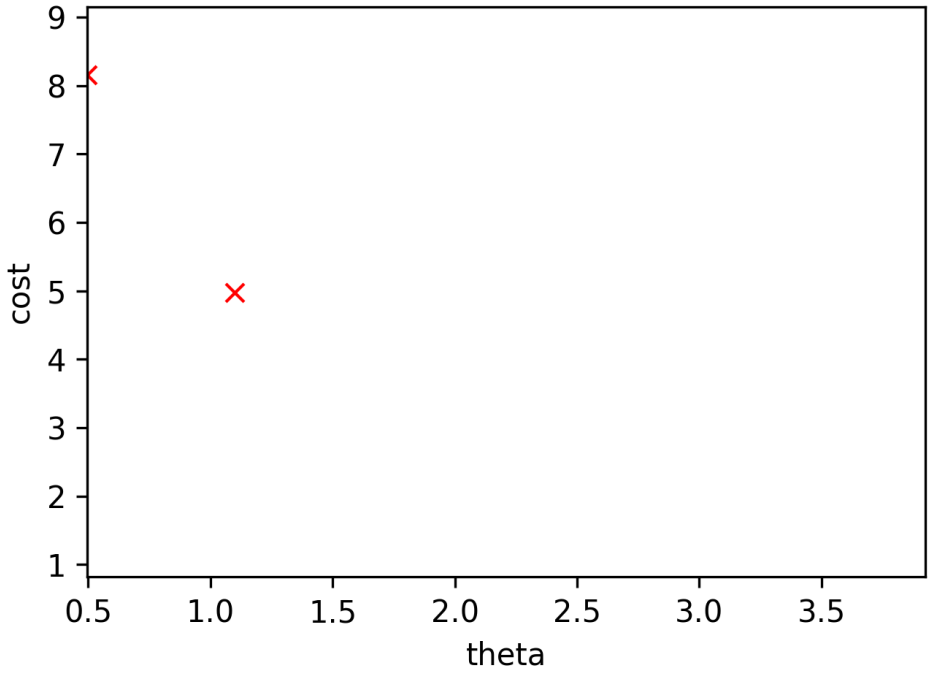
Linear Regression:BatchGradientDescent



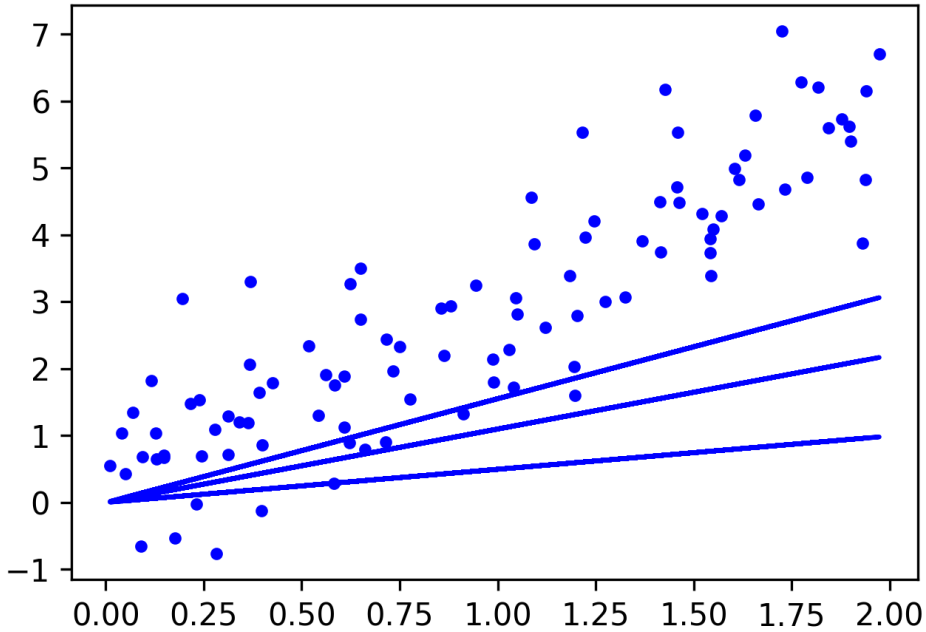


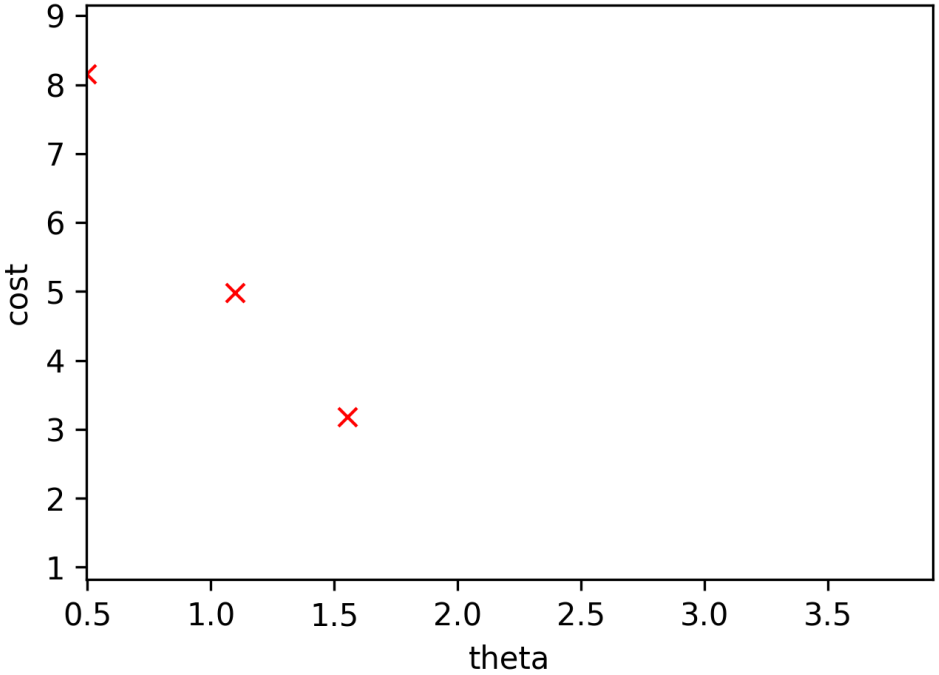
Linear Regression:BatchGradientDescent



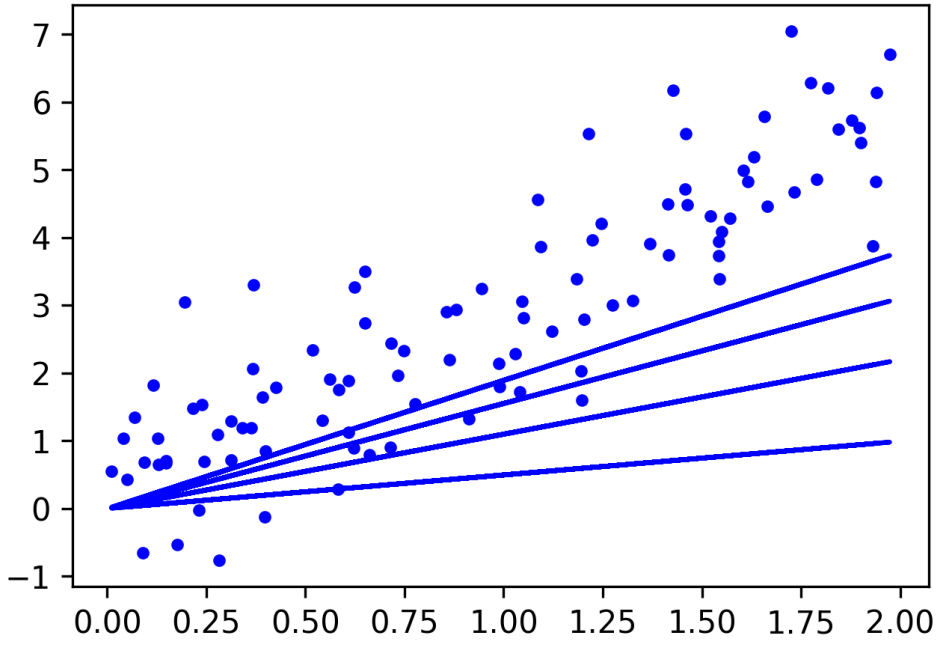


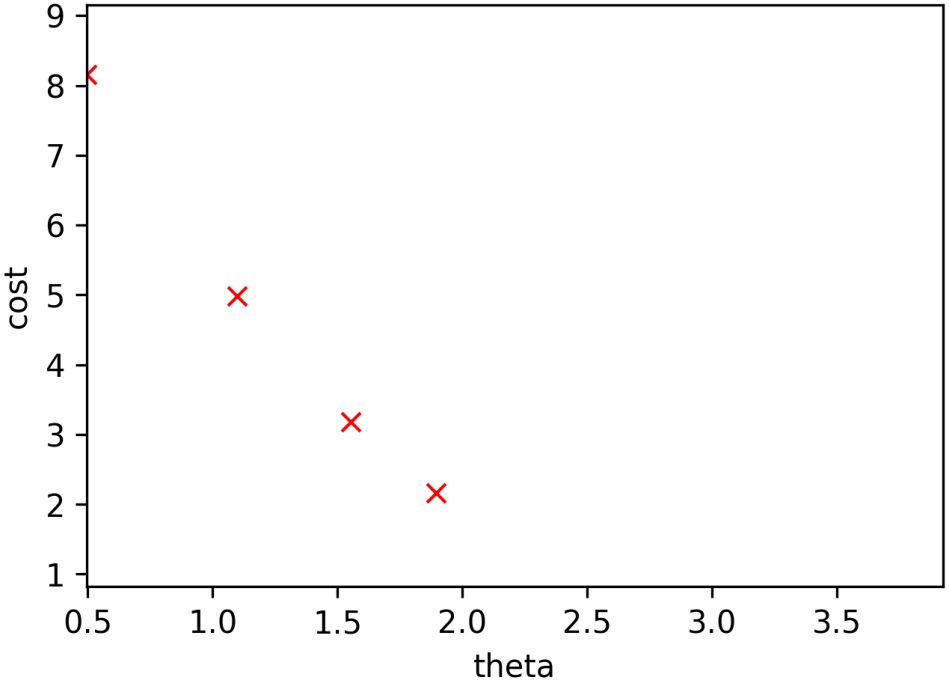
Linear Regression:BatchGradientDescent



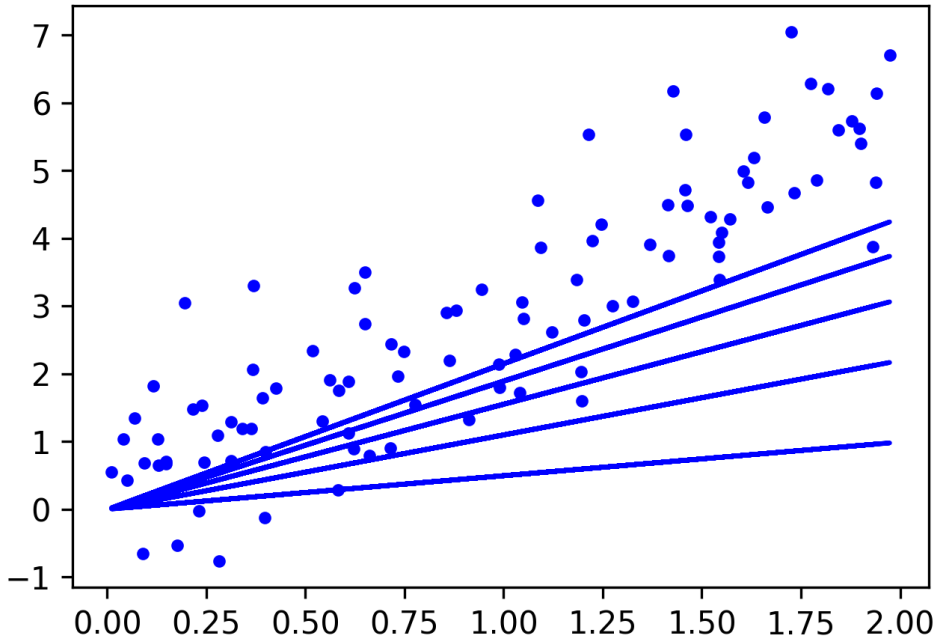


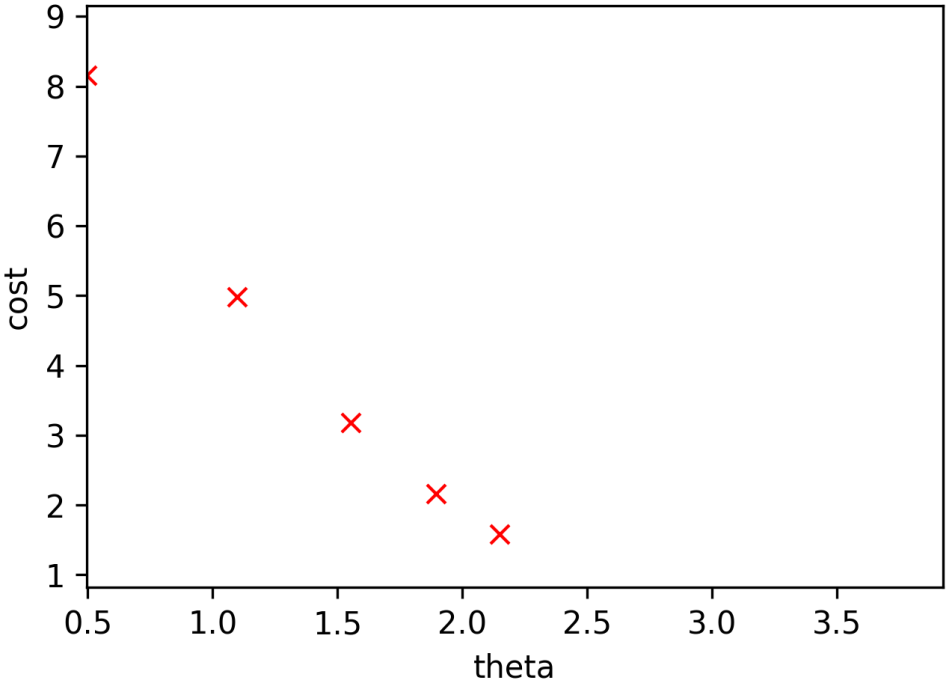
Linear Regression:BatchGradientDescent



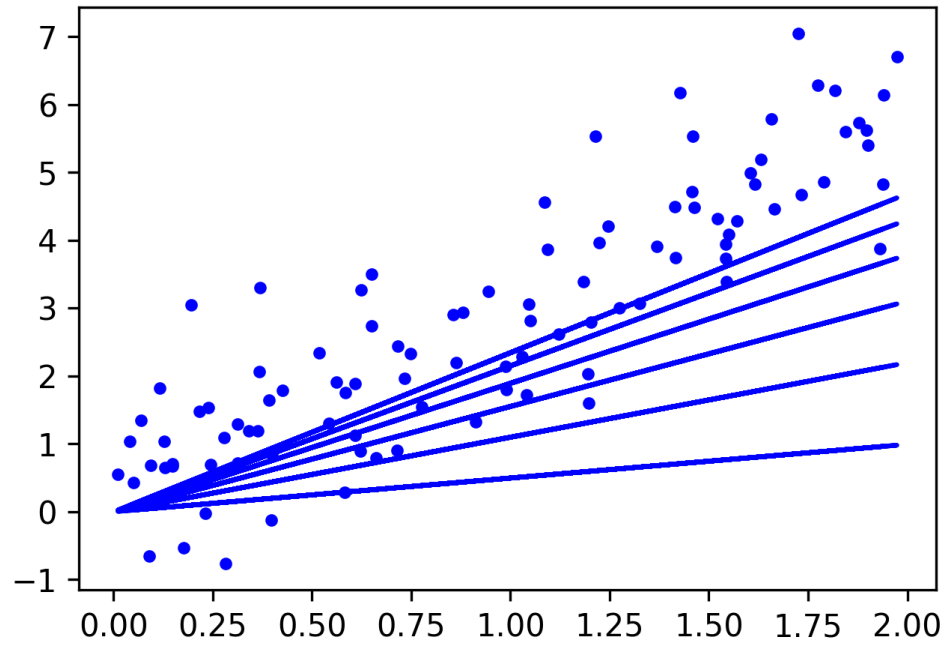


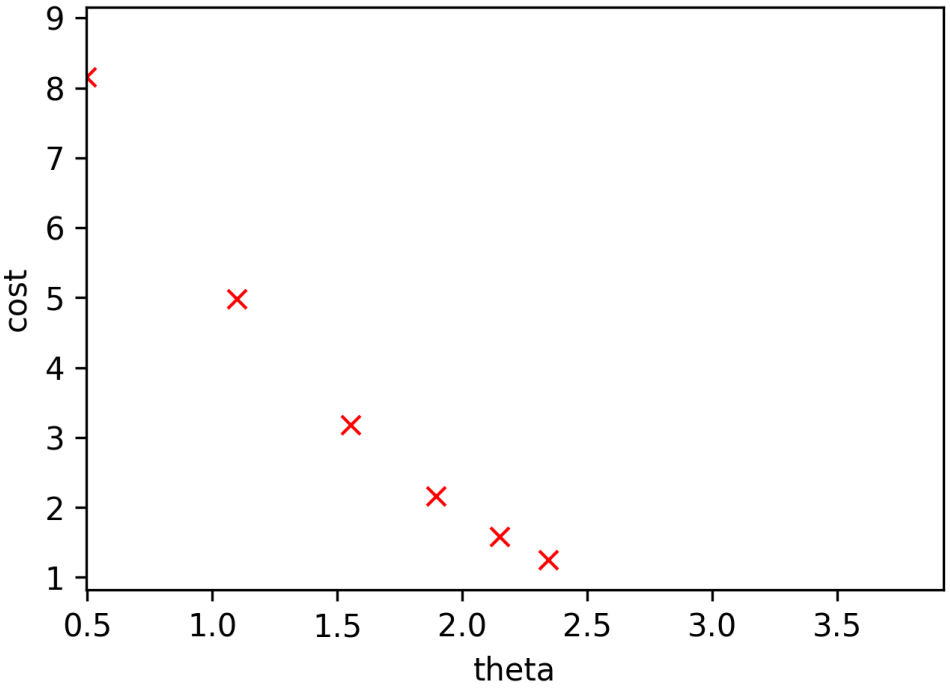
Linear Regression:BatchGradientDescent



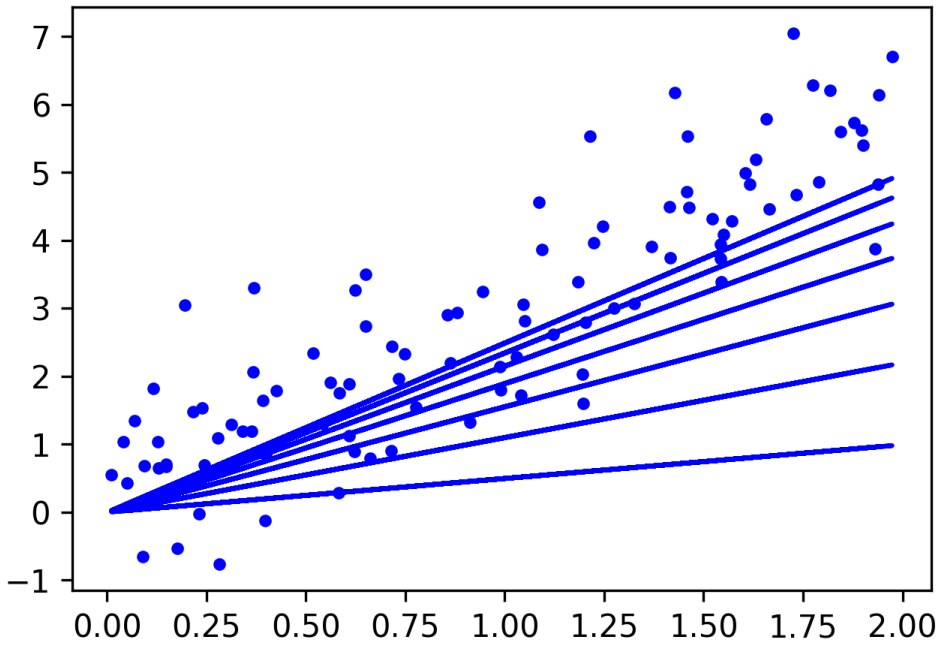


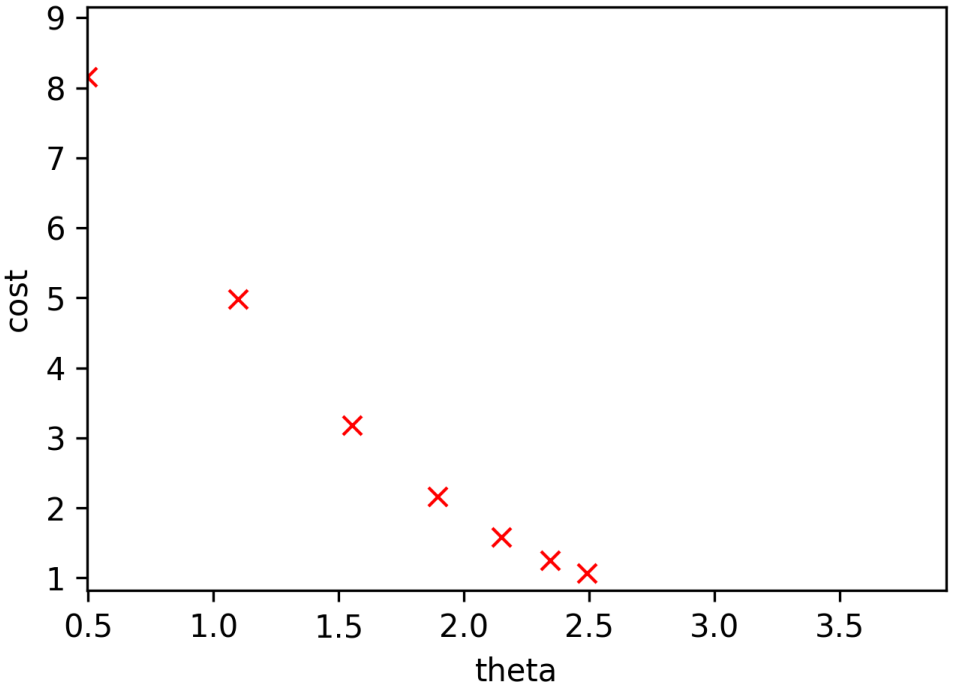
Linear Regression:BatchGradientDescent



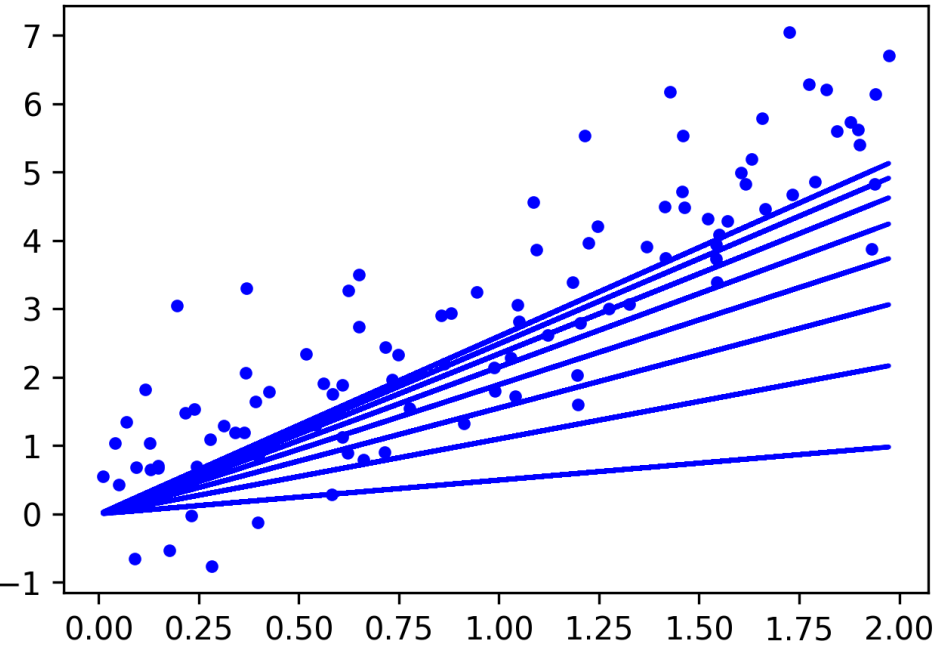


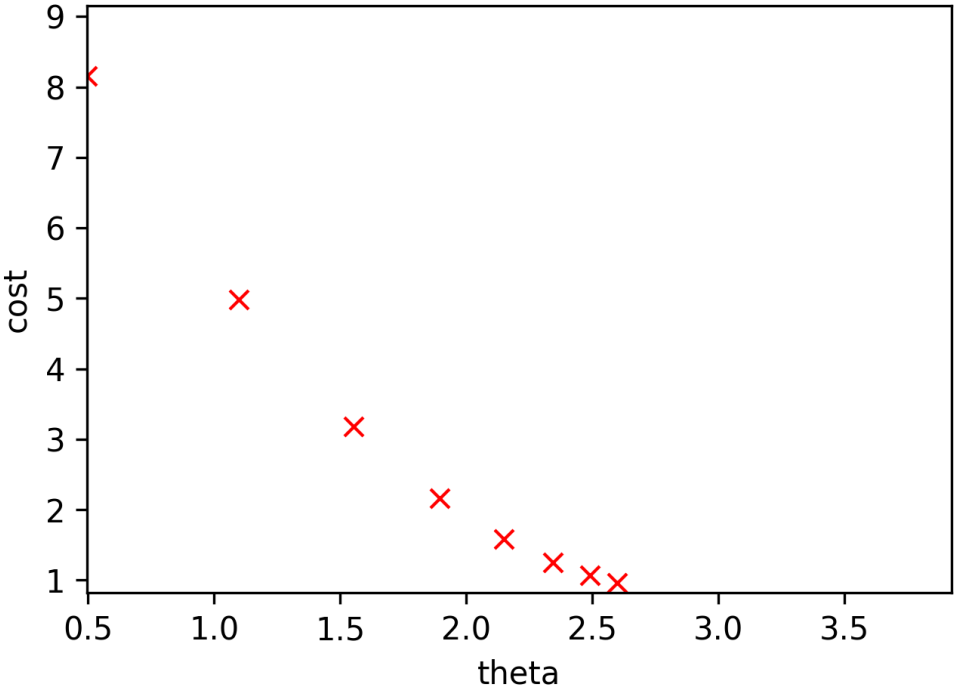
Linear Regression:BatchGradientDescent



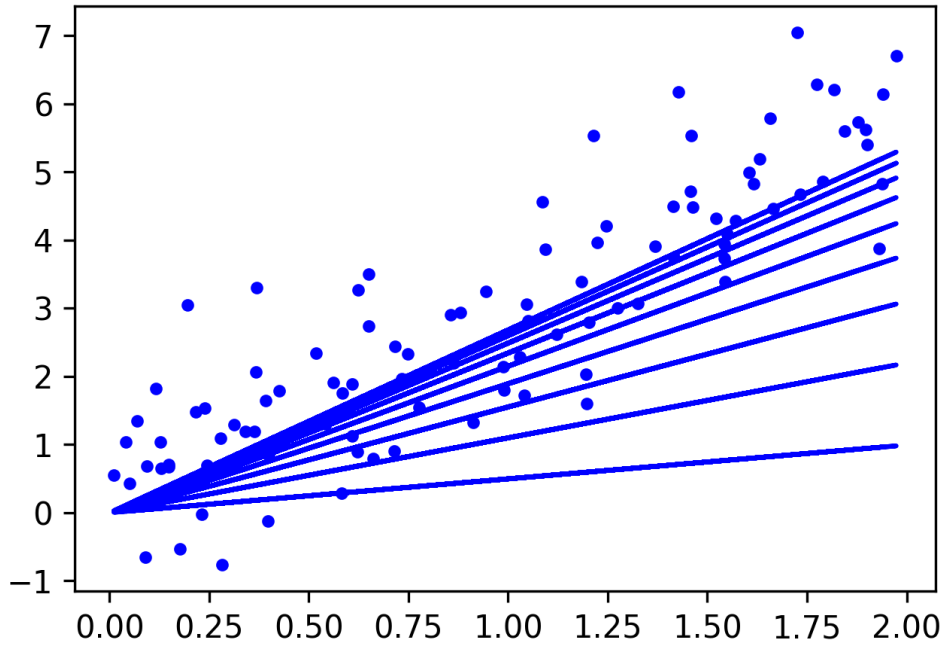


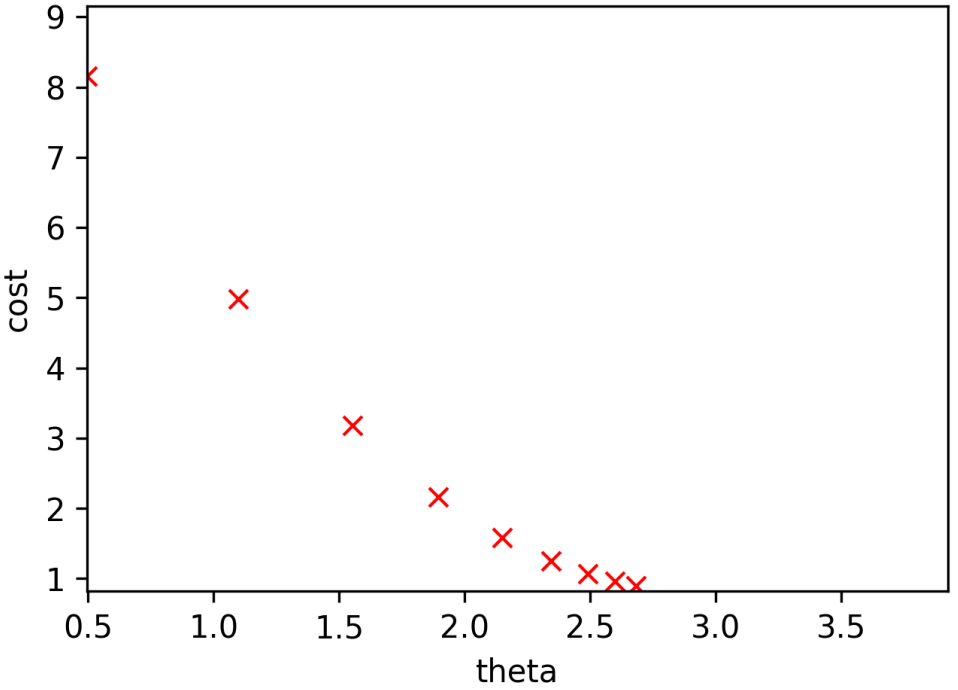
Linear Regression:BatchGradientDescent



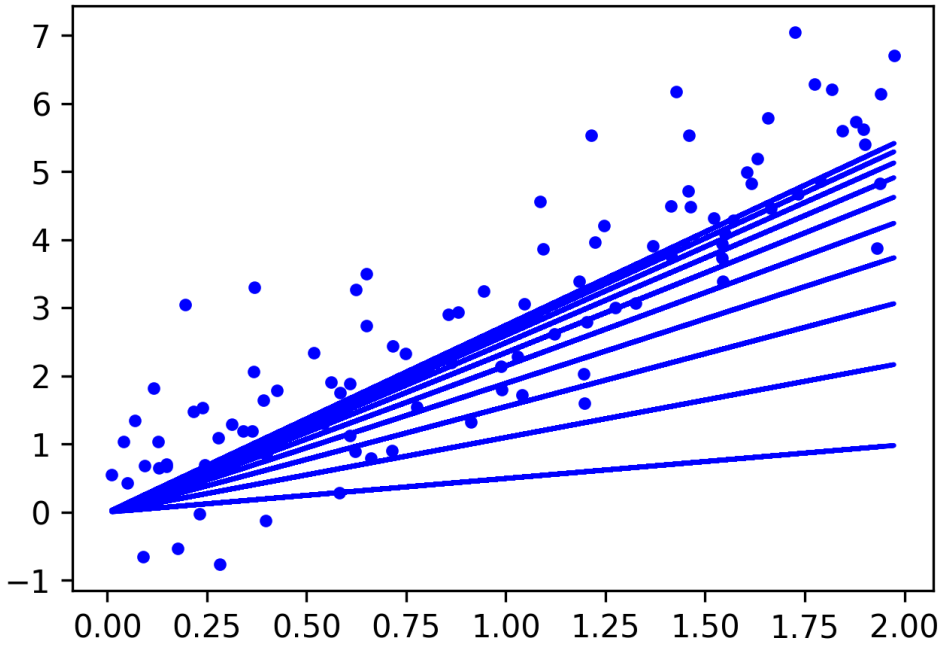


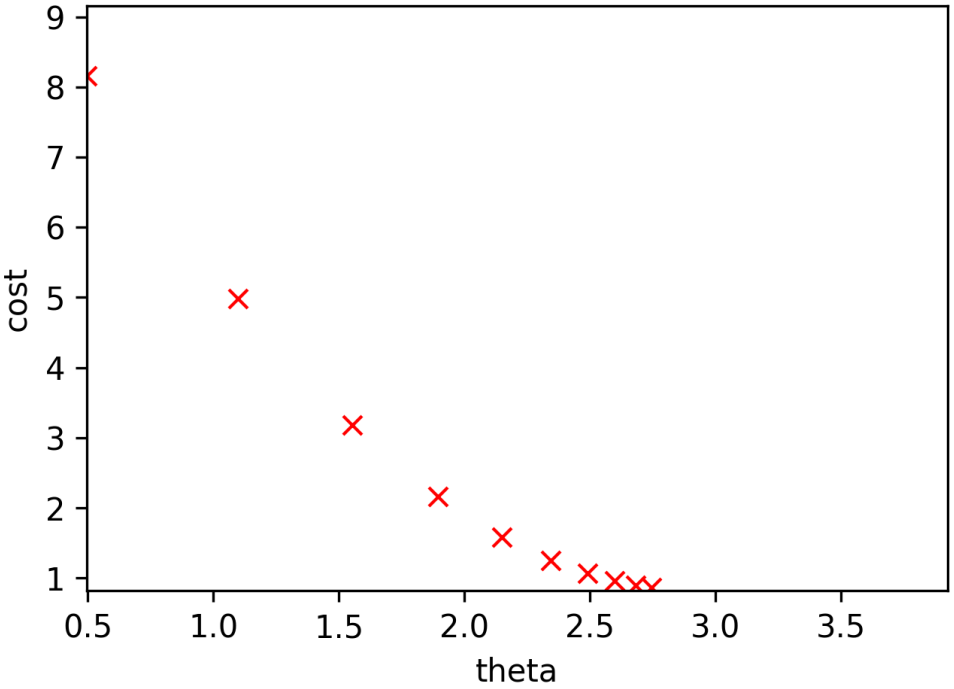
Linear Regression:BatchGradientDescent





Linear Regression:BatchGradientDescent





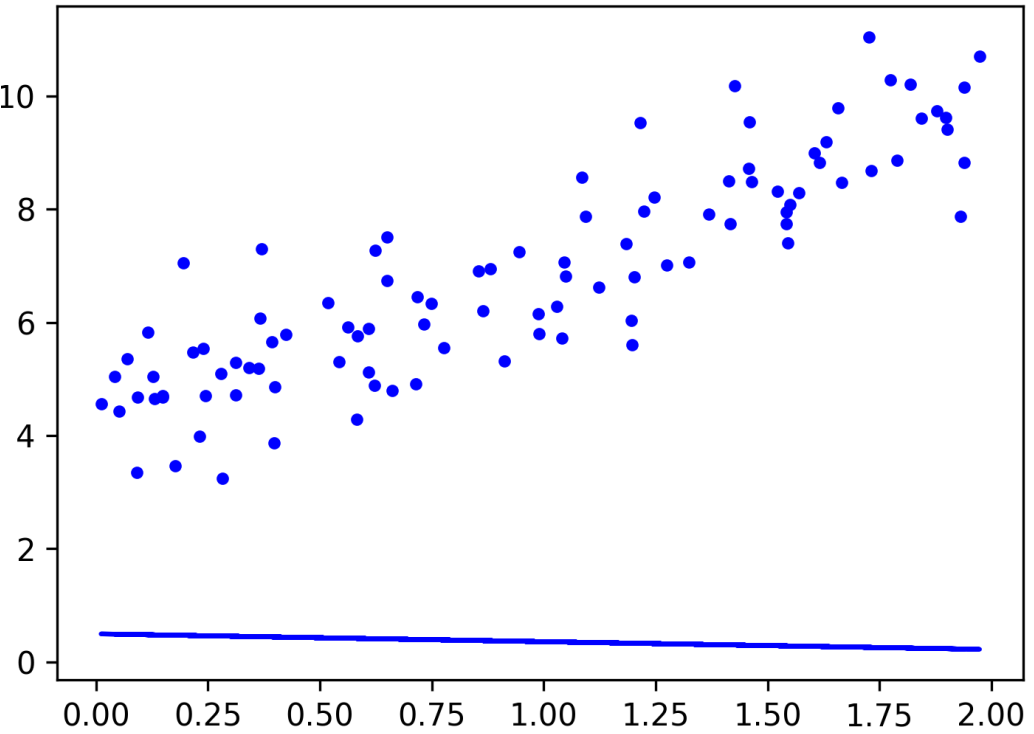
Linear Regression:BatchGradientDescent:ContourPlot

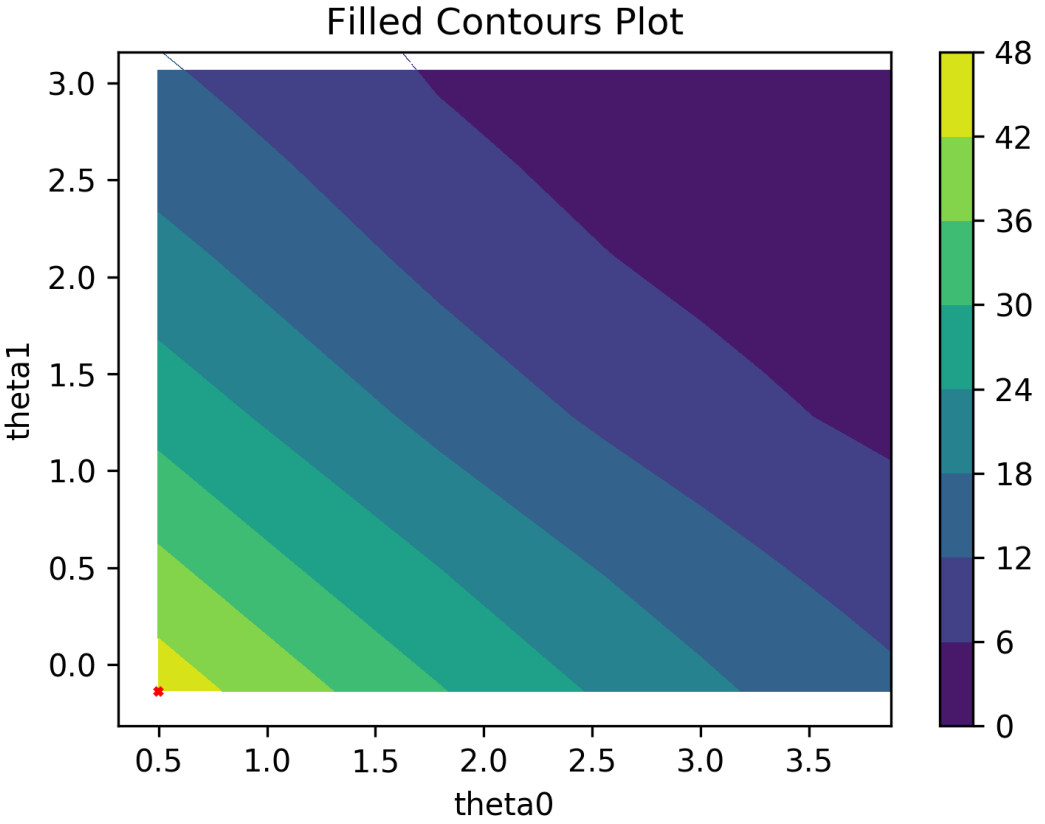
```
def plot_progress(xvec,ttxvec,theta,cost):
    m = len(X_b)
    fig = plt.figure(figsize=(10, 8))
    sub1 = plt.subplot(2, 2, 1)
    sub2 = plt.subplot(2, 2, 2)
    style = "b-"
    sub1.plot(X, y, "b.")
    xmin=np.amin(theta)
    xmax=np.amax(theta)+1
    ymin=np.amin(cost)
    ymax=np.amax(cost)+1
    thetaar=np.array(theta)
    print("thetaar:",thetaar)
    theta0 = thetaar[:,0]
    theta1 = thetaar[:,1]
    print("theta0:",theta0)
    print("theta1:",theta1)
    r,c=theta0.shape
    X1, Y1 = np.meshgrid(theta0, theta1)
    print("X1:",X1)
    print("Y1:",Y1)
    T = np.c_[X1.ravel(), Y1.ravel()]
    print("T:",T)
    Z=np.sum((((T.dot(X_b.T).T-y)**2)/m).T,axis=1).reshape(r,-1)
    print("Z:",Z)
    cp = plt.contourf(X1, Y1, Z)
    plt.colorbar(cp)
    plt.title('Filled Contours Plot')
    plt.xlabel('x (cm)')
    plt.ylabel('y (cm)')
    for i,ttx in enumerate(ttxvec):
        sub1.plot(X, ttx.T[0], style)
        thetai=theta[i]
        plt.scatter(thetai[0][0], thetai[1][0], marker='x', s=6, zorder=10,color='r')
        fl.save_fig("cpprogress"+str(i))
```

- is a vector
- θ_0, θ_1 together
- get all possible combinations of θ_0 & θ_1
- for all possible combinations calculate the cost Z.
- plot θ_0 & θ_1 on x & y axis and draw contours of Z.
- overlay a progressive scatter of 'theta'.

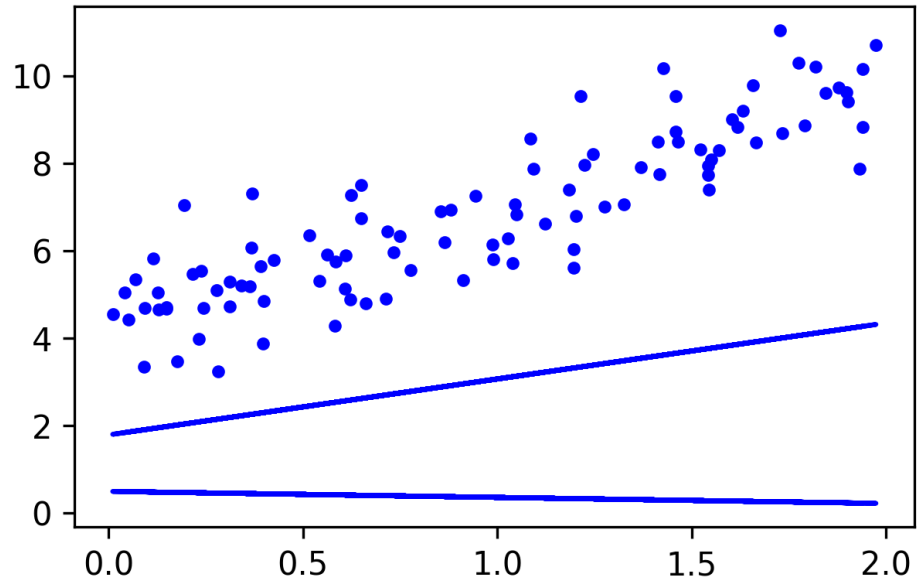
```
thetaar: [[[ 0.49671415]
  [-0.1382643 ]]
 [[ 1.78737583]
  [ 1.27927812]]]

theta0: [[ 0.49671415]
 [ 1.78737583]]

theta1: [[-0.1382643 ]
 [ 1.27927812]]

X1: [[ 0.49671415  1.78737583]
 [ 0.49671415  1.78737583]]

Y1: [[-0.1382643  -0.1382643 ]
 [ 1.27927812  1.27927812]]

T: [[ 0.49671415 -0.1382643 ]
 [ 1.78737583 -0.1382643 ]
 [ 0.49671415  1.27927812]
 [ 1.78737583  1.27927812]]

Z: [[ 45.41619406  30.42392605]
 [ 27.80304718  16.25168913]]
```

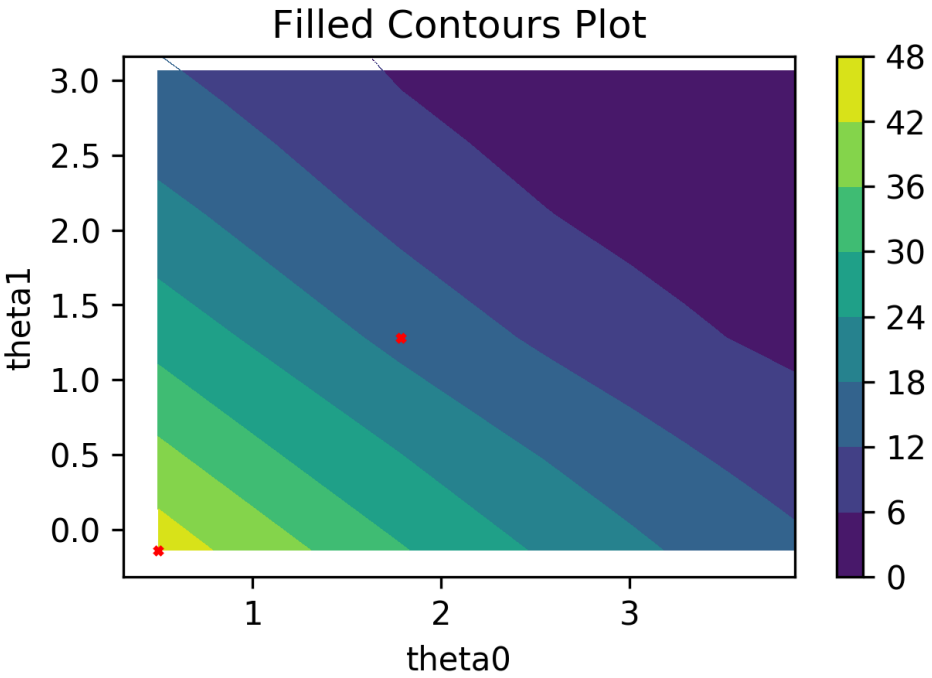
Linear Regression:BatchGradientDescent:ContourPlot



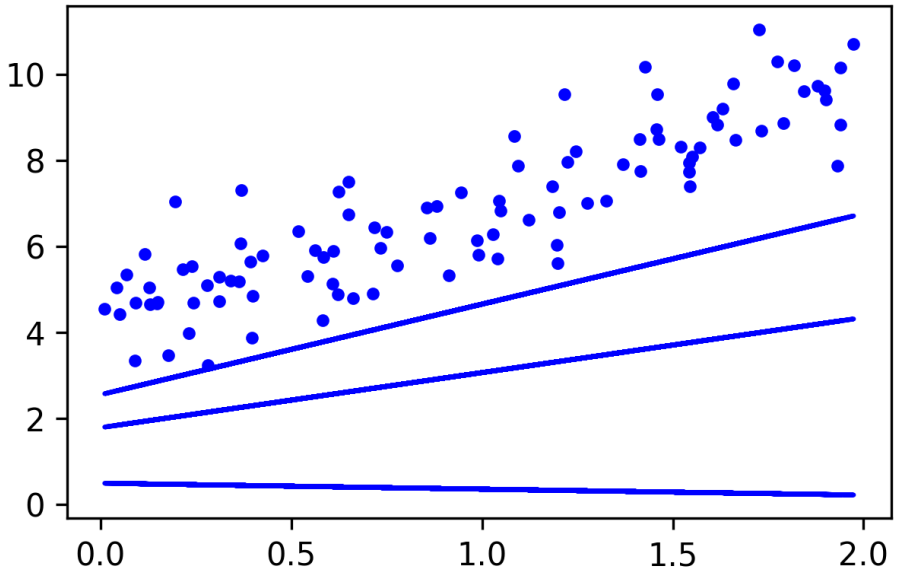


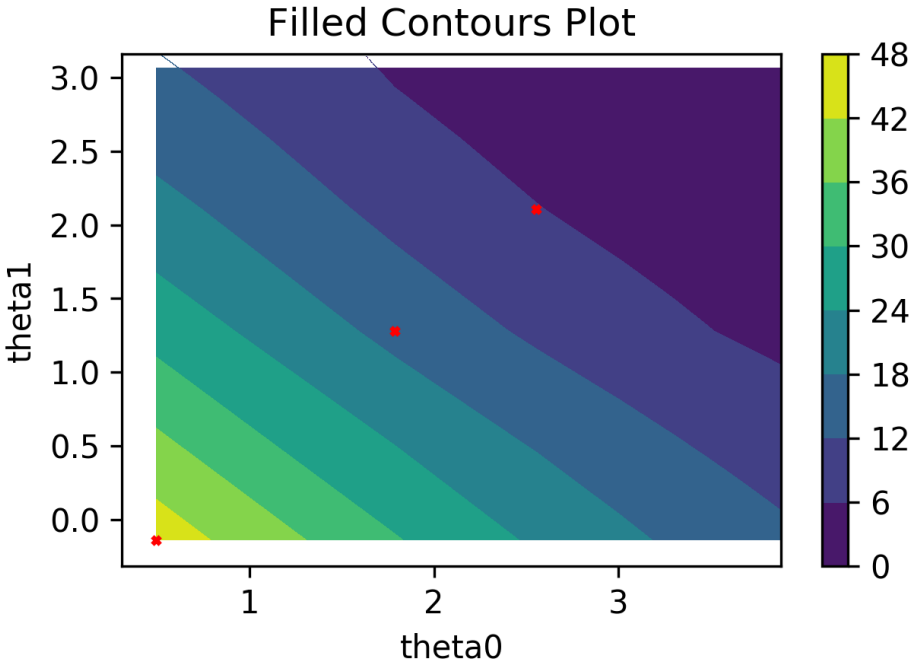
Linear Regression:BatchGradientDescent:ContourPlot



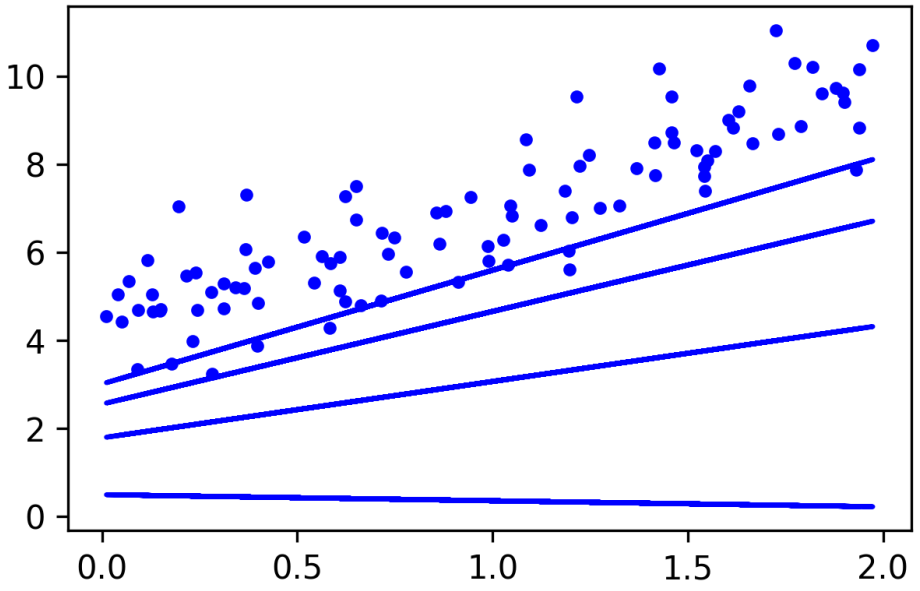


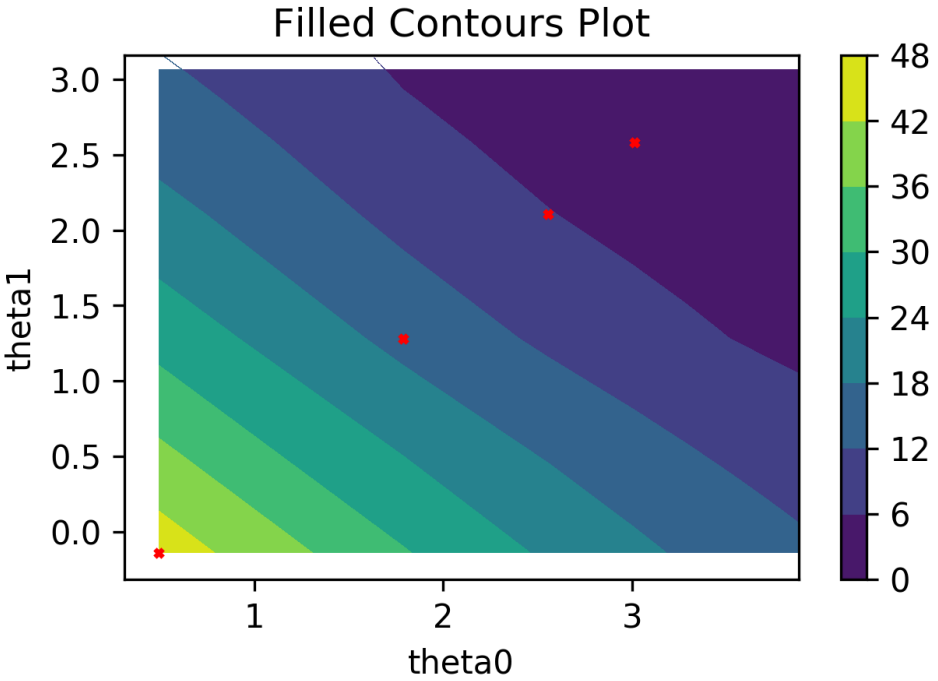
Linear Regression:BatchGradientDescent:ContourPlot



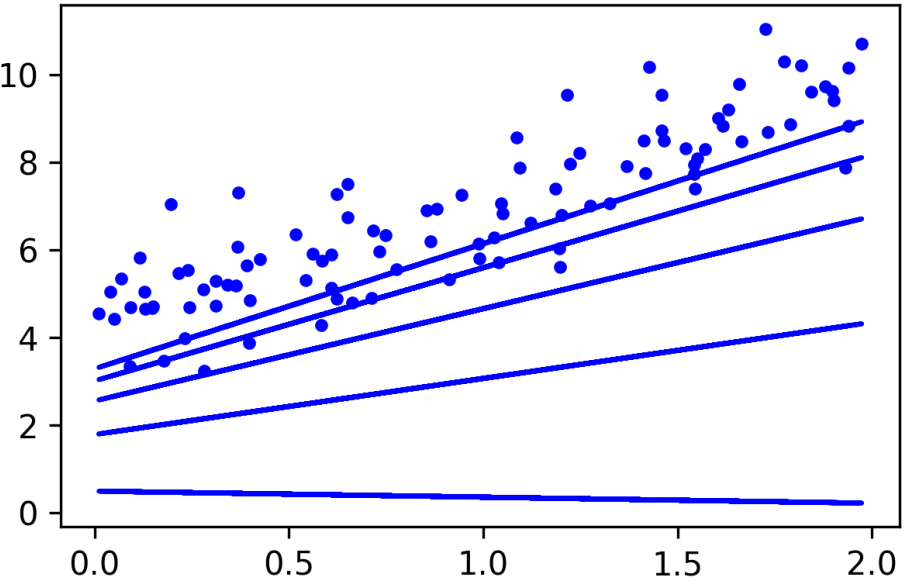


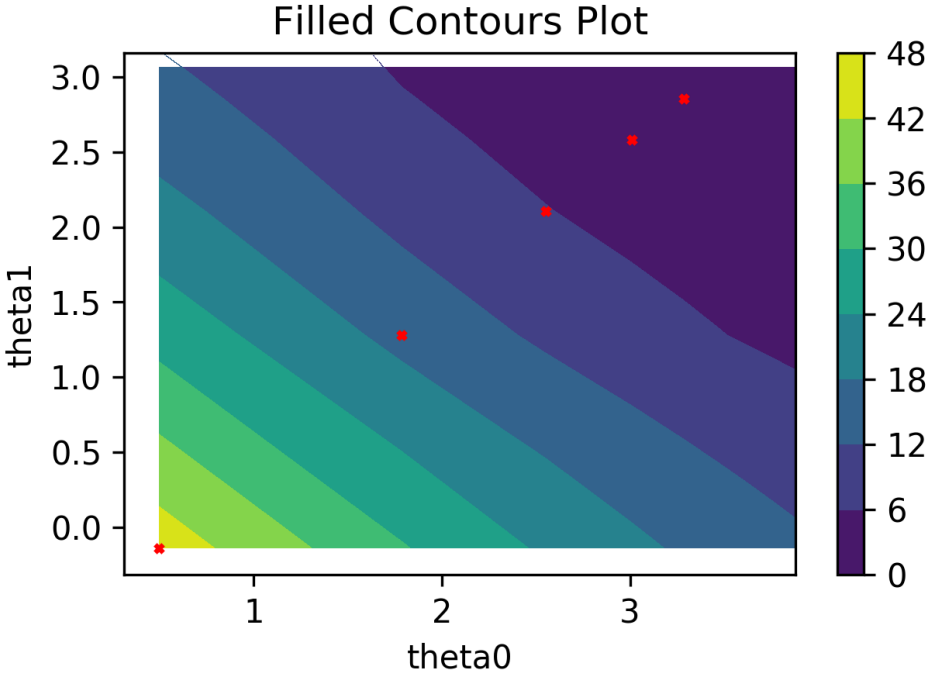
Linear Regression:BatchGradientDescent:ContourPlot



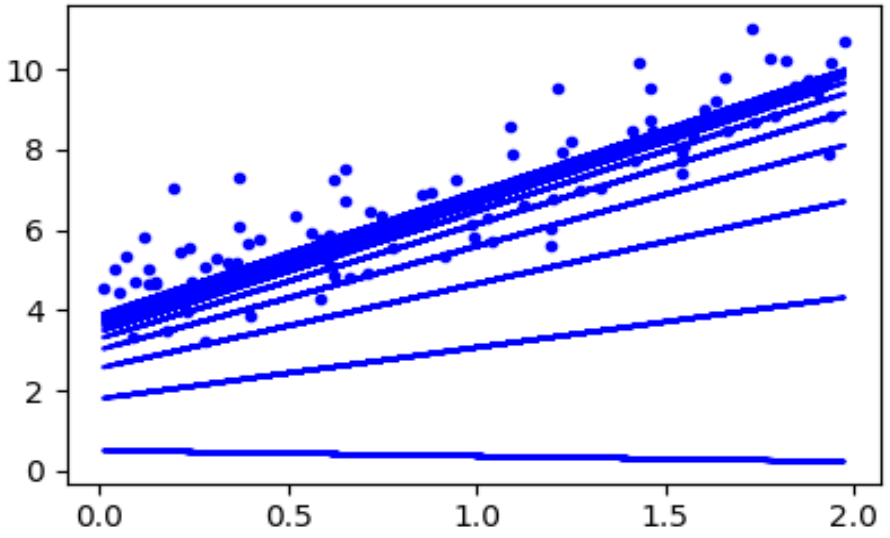


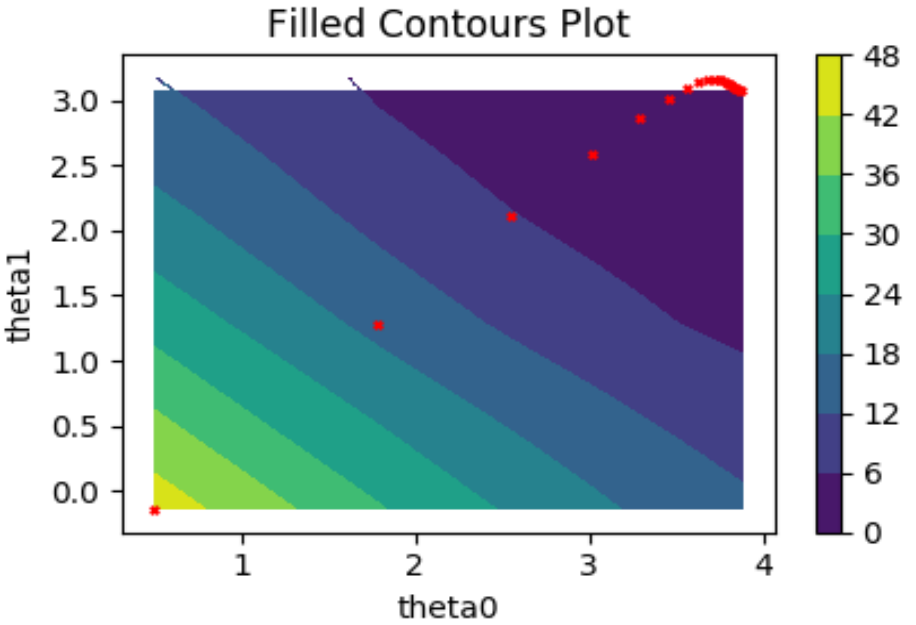
Linear Regression:BatchGradientDescent:ContourPlot



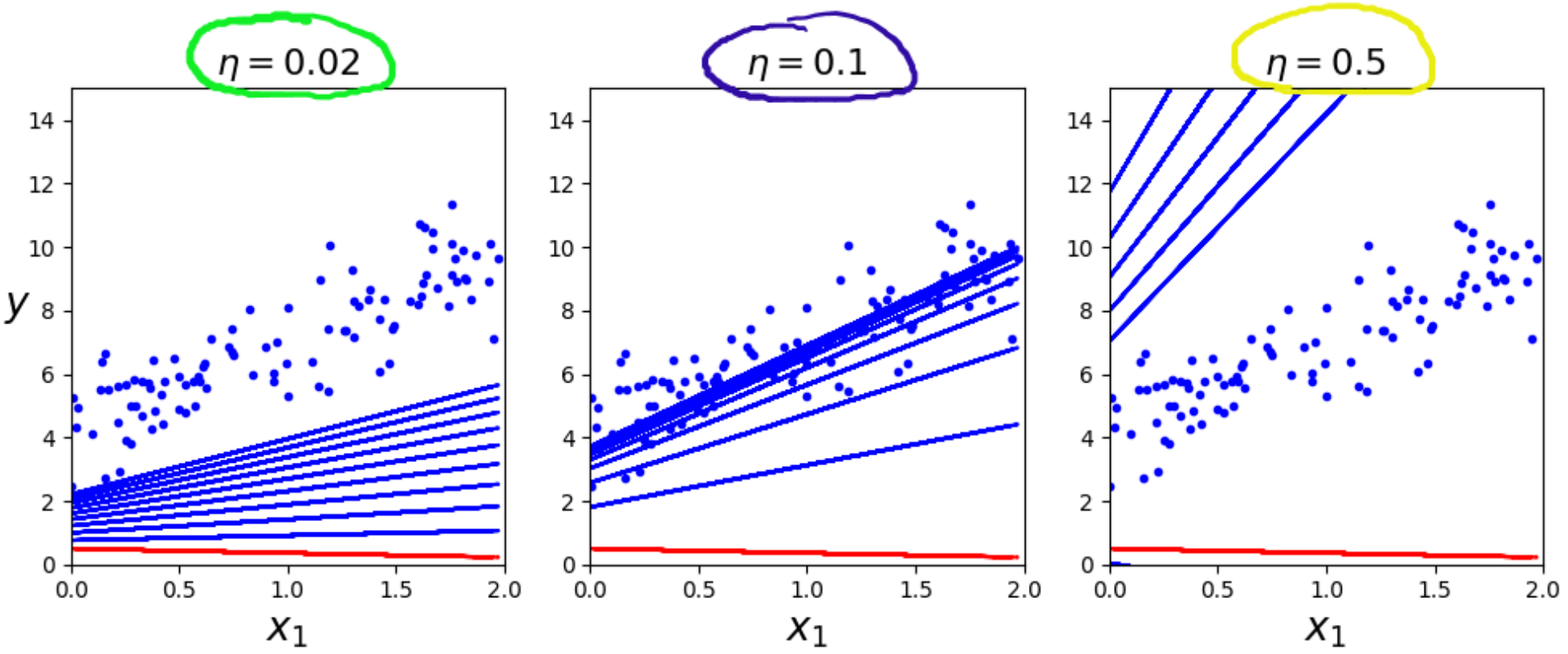


Linear Regression:BatchGradientDescent:ContourPlot:final





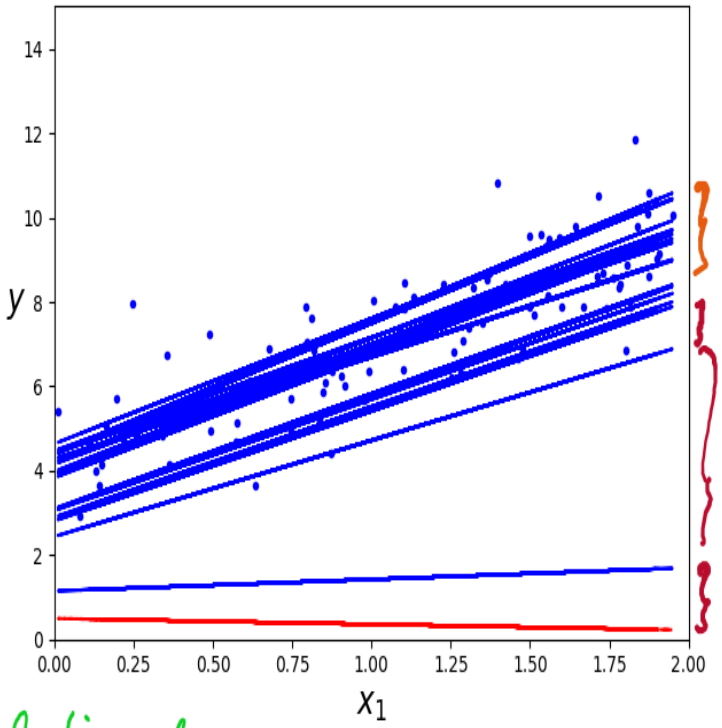
Linear Regression:BatchGradientDescent:learning rate




```
NE:theta_best: [[ 4.27138592]
                [ 2.74644938]]
BGD:theta best: [[ 4.27068108]
                 [ 2.74705865]]  for learning rate: 0.02
BGD:theta best: [[ 4.27138592]
                 [ 2.74644938]]  for learning rate: 0.1
BGD:theta best: [[ -3.22273656e+44]
                 [ -3.72825266e+44]]  for learning rate: 0.5
```

Linear Regression:StochasticGradientDescent

```
n_iterations = 5
t0, t1 = 5, 50  # learning schedule hyperparameters
rnd.seed(42)
theta = rnd.randn(2,1)  # random initialization

def learning_schedule(t):
    return t0 / (t + t1)

m = len(X_b)

def cost(X,theta):
    m = len(X_b)
    y_predict = X.dot(theta)
    return np.sum(((y-y_predict)**2)/m)

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

def gradient(X,theta):
    m = len(X)
    return 2/m * X_b.T.dot(X_b.dot(theta) - y)

for epoch in range(n_iterations):
    for i in range(m):
        if epoch == 0 and i < 20:
            y_predict = thetaTransposeX(X_b,theta)
            style = "b-" if i > 0 else "r--"
            plt.plot(X, y_predict, style)
        random_index = rnd.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)
```



smaller jumps

bigger jumps.

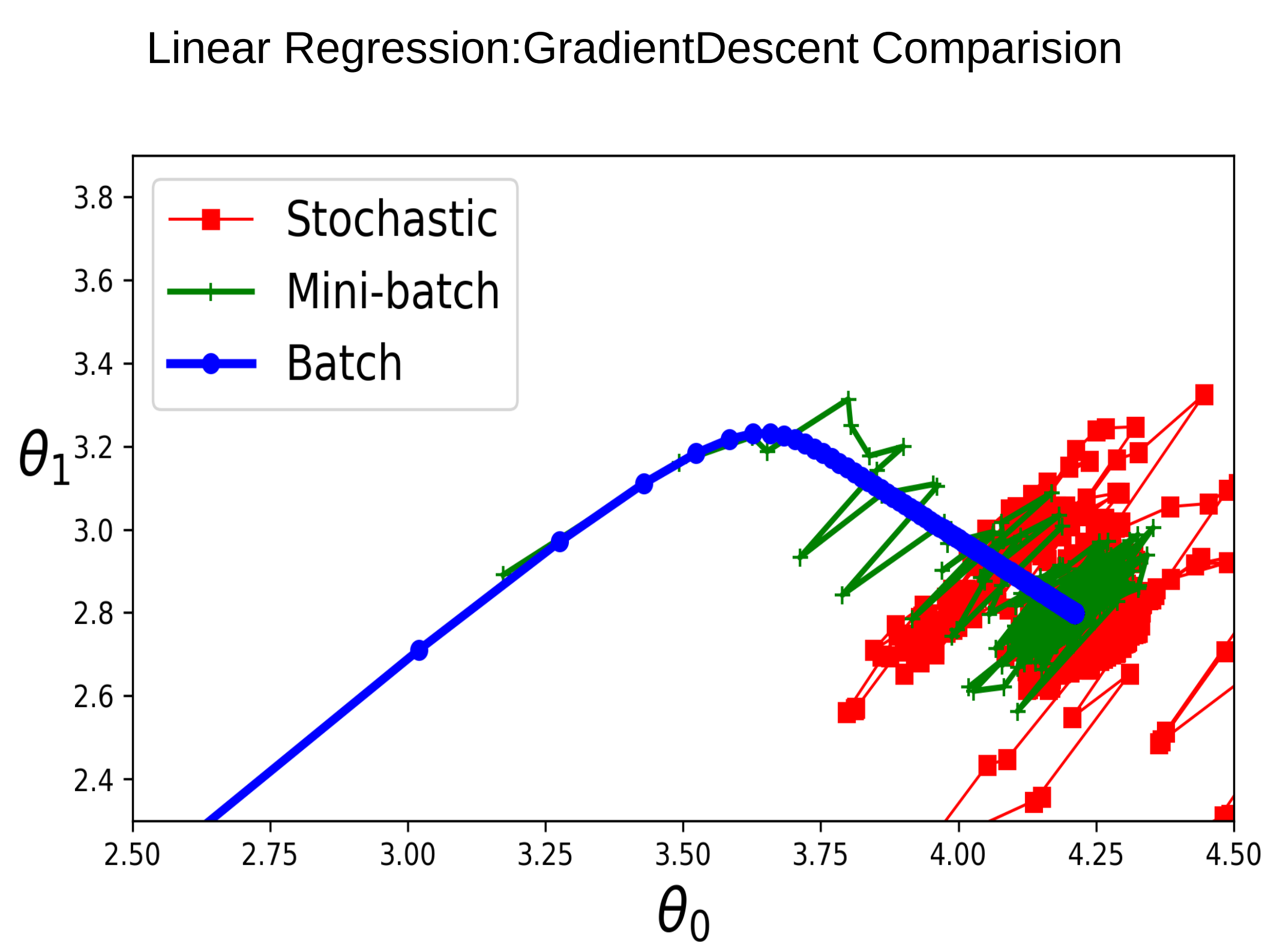
- gradient calculation done over a random selection of sample.
  - random is good to escape local optima but bad because the algorithm may never settle down.
- change in gradient.
- returns learning rates which reduce overtime.

Linear Regression:StochasticGradientDescent(scikit)

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import SGDRegressor

fl=fileloader("input")
X = 2 * rnd.rand(100, 1)
y = 4 + 3 * X + rnd.randn(100, 1)

sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
print("sgd_reg.intercept_, sgd_reg.coef_:",sgd_reg.intercept_, sgd_reg.coef_)
```

Linear Regression:MiniBatchGradientDescent

```
def learning_schedule(t):
    return t0 / (t + t1)
m = len(X_b)
theta_path_mgd = []
n_iterations = 50
minibatch_size = 20
rnd.seed(42)
theta = rnd.randn(2,1)  # random initialization

t0, t1 = 10, 1000
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    shuffled_indices = rnd.permutation(m)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, m, minibatch_size):
        t += 1
        xi = X_b_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(t)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)
```


Linear Regression:GradientDescent Comparision
Stochastic
Mini-batch
Batch
θ_1
θ_0
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.50
2.75
3.00
3.25
3.50
3.75
4.00
4.25
4.50

Linear Regression:Normal Equation

1. $Y = \theta x$, so if $x, y,$ & θ were scalar then $\theta = x/y$.
2. Because x, θ may be matrices and y may be a vector and matrix division is not defined.
3. a round about way of doing matric devision is by finding out the inverse. Which is what we use in Normal equation.
4. $\theta = (X^T . X)^{-1} . X^T . y \Rightarrow \theta = X^T . y \,/\!/ (X^T . X) = y/X$

Linear Regression:Normal Equation

$$X = \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}, \quad Y = \begin{bmatrix} 4 \\ 6 \end{bmatrix} \quad \theta = ?$$

$$X^T X = \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 5 \\ 5 & 13 \end{bmatrix}$$

$$(X^T X)^{-1} = \frac{1}{26 - 25} \begin{bmatrix} 13 & -5 \\ -5 & 2 \end{bmatrix} = \begin{bmatrix} 13 & -5 \\ -5 & 2 \end{bmatrix}$$

- Using normal equation

$$\theta = (X^T X)^{-1} X^T \cdot Y$$

$$= \begin{bmatrix} 13 & -5 \\ -5 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 13 & -5 \\ -5 & 2 \end{bmatrix} \cdot \begin{bmatrix} 10 \\ 26 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

Matrix inverse

$$A = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

$$A^{-1} = \frac{1}{(ad - bc)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

$$A \times A^{-1} = I$$

$$AA^{-1} = \begin{bmatrix} 2 & 5 \\ 5 & 13 \end{bmatrix} \begin{bmatrix} 13 & -5 \\ -5 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Linear Regression:Normal Equation

```
#prepare theta vector
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = LA.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
print("NE:theta_best:",theta_best)
```

```
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./ML/linearregression/batchgradientdescent3.py
NE:theta_best: [[ 3.92959891]
 [ 3.04278259]]
BGD:theta best: [[ 3.92864309]
 [ 3.04362033]]  for learning rate: 0.02
BGD:theta best: [[ 3.92959891]
 [ 3.04278259]]  for learning rate: 0.1
BGD:theta best: [[ -2.33831603e+45]
 [ -2.66787815e+45]]  for learning rate: 0.5
```

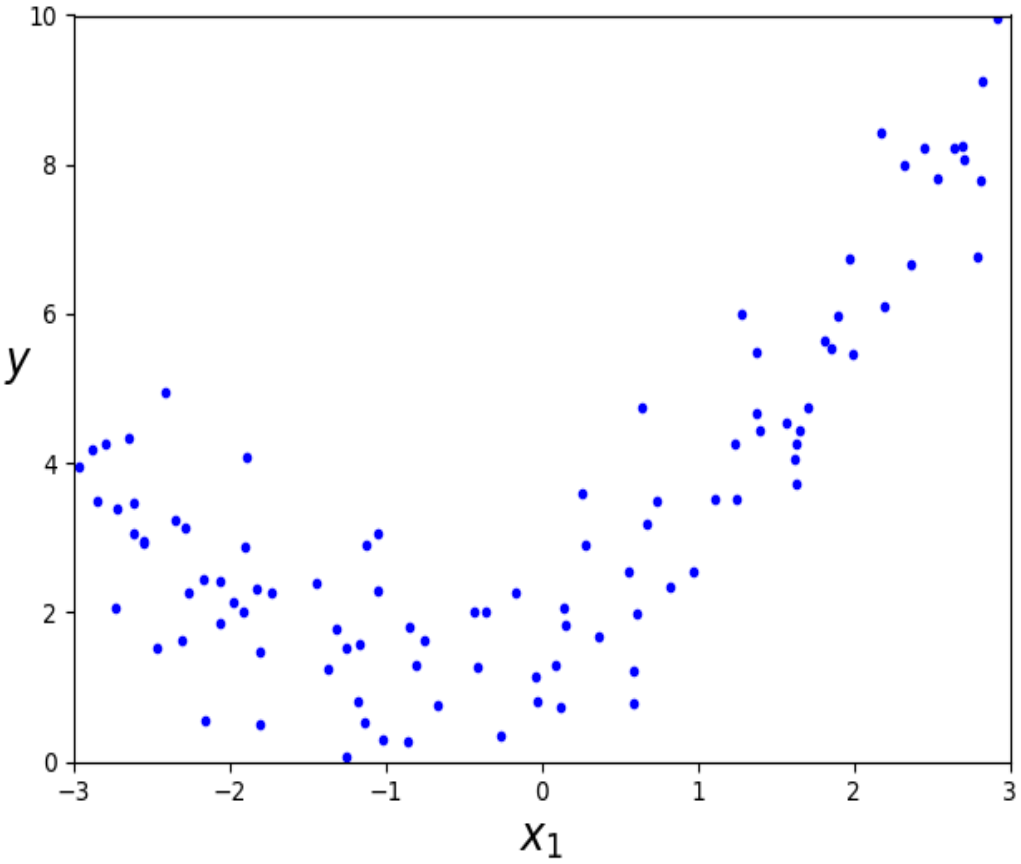
Linear Regression:Normal Equation(scikit)

```
from sklearn.linear_model import LinearRegression
fl=fileloader("input")
X = 2 * rnd.rand(100, 1)
y = 4 + 3 * X + rnd.randn(100, 1)

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
fl.save_fig("generated_data_plot")
plt.show()

#calculate theta vector
lin_reg = LinearRegression()
lin_reg.fit(X, y)
print(lin_reg.intercept_,lin_reg.coef_)

X_new = np.array([[0], [2]])
print("feature vector:",X_new)
y_predict = lin_reg.predict(X_new)
print("y_predict:",y_predict)
```

Polynomial Regression

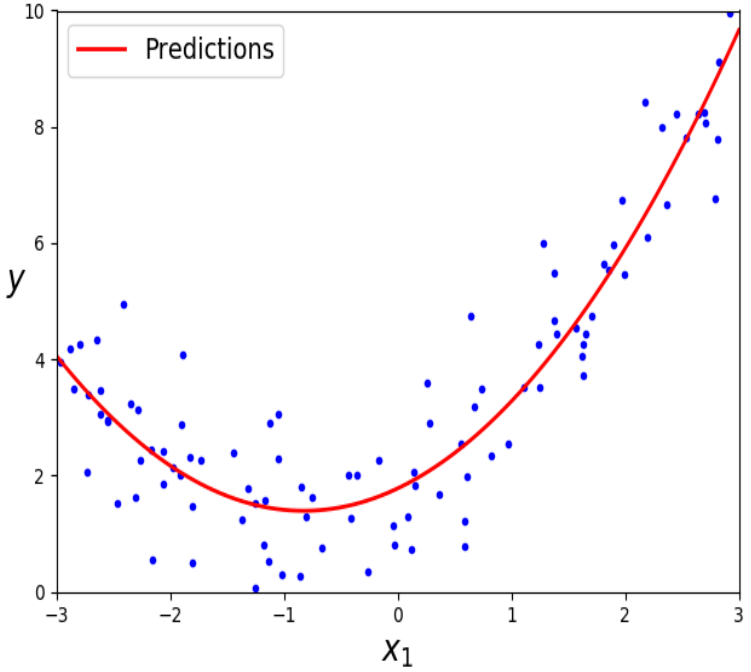
```
fl=fileloader("input")
rnd.seed(42)
m = 100
X = 6 * rnd.rand(m, 1) - 3
y = 2 + X + 0.5 * X**2 + rnd.randn(m, 1)
```

- Data generator
- A straight line will not fit this data properly.



Polynomial Regression

```
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
print("X[0] and its square:",X[0])
print("X[0] and its square:",X_poly[0])

#train
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
print("lin_reg.intercept_:",lin_reg.intercept_," lin_reg.coef_:",lin_reg.coef_)

#test data
X_new=np.linspace(-3, 3, 100).reshape(100, 1)
print("X_new[0]:",X_new[0])
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
print("predicted:y_new[0]:",y_new[0])
```




```
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./ML/linearregression/polynomialregression.py
X[0] and its square: [-0.75275929]
X[0] and its square: [-0.75275929  0.56664654]
lin_reg.intercept_: [ 1.78134581]  lin_reg.coef_: [[ 0.93366893  0.56456263]]
X_new[0]: [-3.]
predicted:y_new[0]: [ 4.06140272]
```

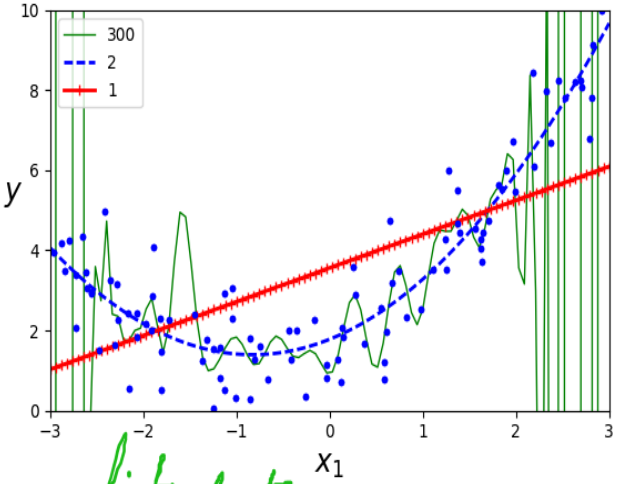
- X[0] and its square as its features
- $.56x^2 + .93x + 1.78$ as computed by training, and the original function was $.5x^2 + x + 2$. which is a descent function approximation.

Polynomial Regression

- Note that when there are multiple features, Polynomial Regression is capable of finding relationships between features (which is something a plain Linear Regression model cannot do).
- This is made possible by the fact that PolynomialFeatures also adds all combinations of features up to the given degree.
- For example, if there were two features a and b, PolynomialFeatures with degree=3 would not only add the features a2, a3, b2, and b3, but also the combinations ab, a2b, and ab2.

Learning Curve:Overfit and underfit

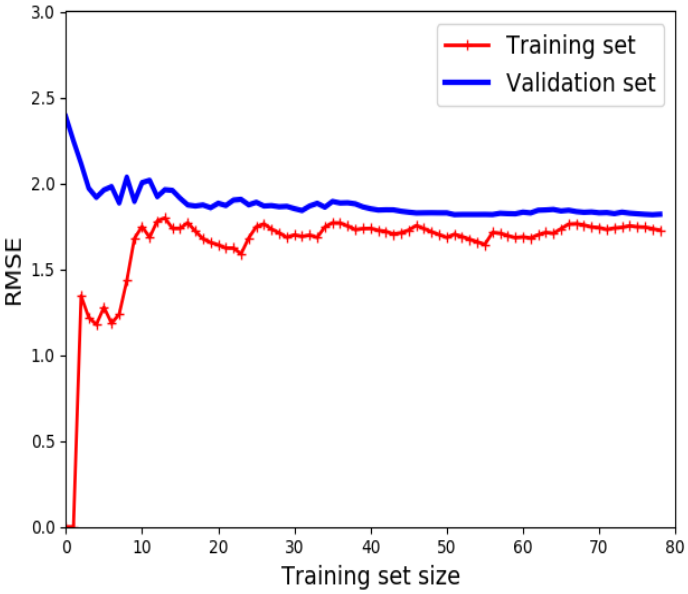
```
for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline((
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ))
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)
```



- high order polynomial trying to overfit data
- quadratic is a good fit for known quadratic data
  - the right degree of freedom must be chosen after visualizing the data
- linear model clearly an underfit for the data.
- but in general, you would not know the data generation function.
- Cross validation is one way. Learning curve is another.

Learning Curve:Overfit and underfit

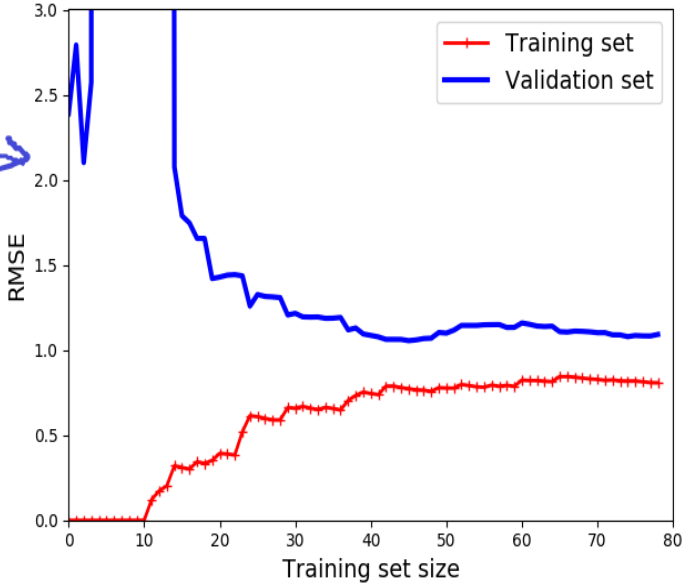
```
def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=10)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="Training set")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Validation set")
    plt.legend(loc="upper right", fontsize=14)
    plt.xlabel("Training set size", fontsize=14)
    plt.ylabel("RMSE", fontsize=14)

lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```



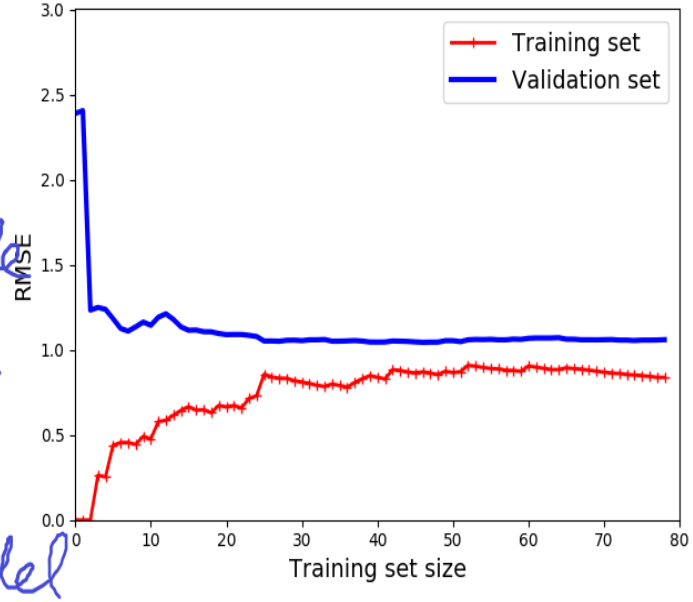
- This is clearly an underfit. The algo does well with very small data but is unable to generalize even on training data.
  - Hence the error goes up.
  - Adding more data would not make it better or worse.
- On the test set, with very few data it is unable to generalize.
  - As the model is shown more data, it averages close to the training set. Both these curves have a high RMSE.

Learning Curve:Overfit and underfit

```
polynomial_regression = Pipeline((
        ("10 deg-poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("sgd_reg", LinearRegression()),
    ))

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])
fl.save_fig("learning_curves_plot-10")
plt.show()

polynomial_regression = Pipeline((
        ("2 deg-poly_features", PolynomialFeatures(degree=2, include_bias=False)),
        ("sgd_reg", LinearRegression()),
    ))

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])
fl.save_fig("learning_curves_plot-2")
plt.show()
```

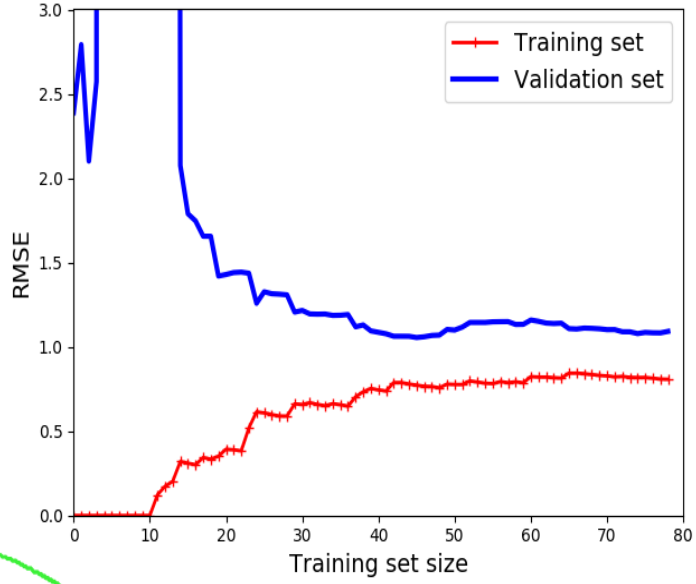


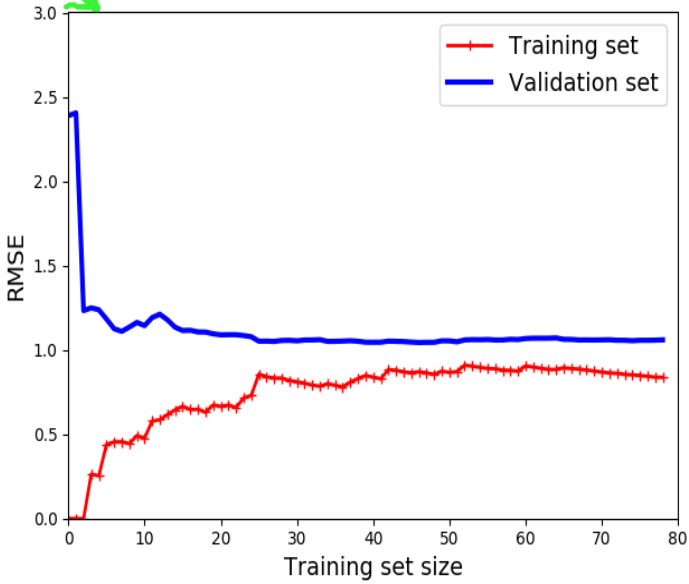
- Difference from the previous model.
  - error on training data is much less.
  - gap → unable to generalize. Hallmark of an overfitting model. If you provide more data the curves will continue to get closer.
- one way to improve overfitting model is to feed more data.



Learning Curve:Overfit and underfit

```
polynomial_regression = Pipeline((
        ("10 deg-poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("sgd_reg", LinearRegression()),
    ))

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])
fl.save_fig("learning_curves_plot-10")
plt.show()

polynomial_regression = Pipeline((
        ("2 deg-poly_features", PolynomialFeatures(degree=2, include_bias=False)),
        ("sgd_reg", LinearRegression()),
    ))

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])
fl.save_fig("learning_curves_plot-2")
plt.show()
```





- Models generalization can be expressed as 3 different errors.
  - Bias - High bias model is likely to underfit data.
    - generalization is due to wrong assumptions. (model is linear and data is quadratic.

Learning Curve:Overfit and underfit

- Variance - This is due to models extreme sensitivity to small variations. (high degree polynomial model is likely to have high variance and thus overfit the data.
- Irreducible errors - This is due to noisiness of data. The only way to reduce errors is by cleaning up the data.

- Increase model complexity → inc variance, dec bias
- Decrease model complexity → dec variance, inc bias

Learning Curve:Overfit and underfit

Accounting for overfitting:

(1) feature selection

- The problem being you also loose information
- The right tactic here would be to visualize and see the non important features

(2) Regularization

(3) In practice a combination of both techniques is used.

Regularization

In regularized linear regression, we choose θ to minimize

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

What if λ is set to an extremely large value (perhaps for too large for our problem, say $\lambda = 10^{10}$)?

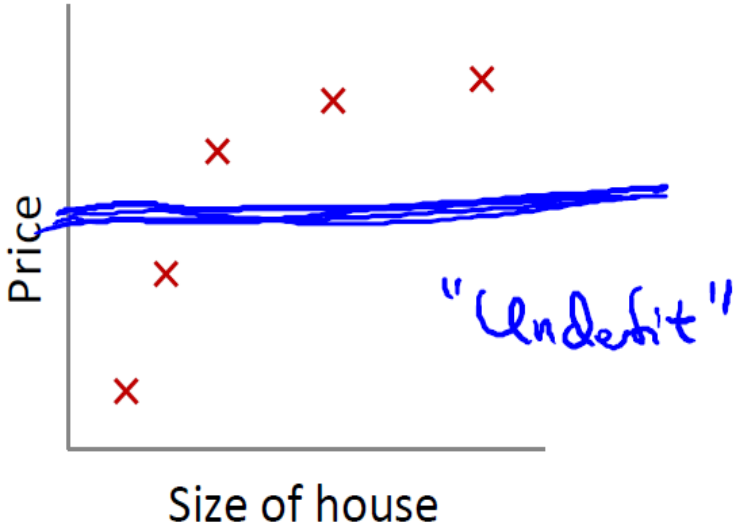
- Algorithm works fine; setting λ to be very large can't hurt it
- Algortihm fails to eliminate overfitting.
- Algorithm results in underfitting. (Fails to fit even training data well).
- Gradient descent will fail to converge.

Regularization

In regularized linear regression, we choose θ to minimize

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

What if λ is set to an extremely large value (perhaps for too large for our problem, say $\lambda = 10^{10}$)?



$h_\theta(x)$

$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$\theta_1, \theta_2, \theta_3, \theta_4$

$\theta_1 \approx 0, \theta_2 \approx 0$

$\theta_3 \approx 0, \theta_4 \approx 0$

$h_\theta(x) = \theta_0$

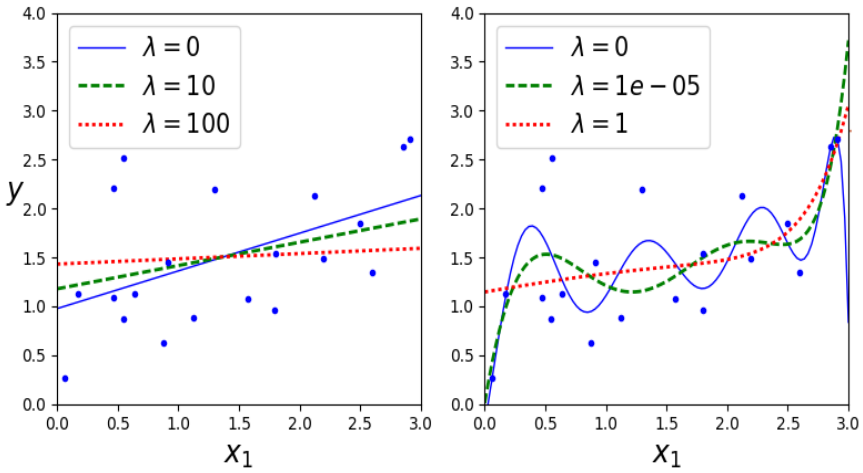
Regularization:Linear Regression

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n}\theta_j^2\right]$$

$$\min_{\theta} J(\theta)$$

Regularization:Linear Regression

```
def plot_model(model_class, polynomial, lambdas, **model_kargs):
    for lambdai, style in zip(lambdas, ("b-", "g--", "r:")):
        model = model_class(lambdai, **model_kargs) if lambdai > 0 else LinearRegression()
        if polynomial:
            model = Pipeline((
                    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ))
        model.fit(X, y)
        y_new_regul = model.predict(X_new)
        lw = 2 if lambdai > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\lambda = {}$".format(lambdai))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 4])

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Ridge, polynomial=False, lambdas=(0, 10, 100))
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Ridge, polynomial=True, lambdas=(0, 10**-5, 1))
```



Linear Regression

- Higher λ values reduce the effect of θ and x on the curve
- The idea is to choose λ in such a way that we get the best fit with the least possible θ.

Polynomial Regression

- It "flattens" out the curve.

As usual, the advise here is to visualize for various values of λ and then pick the best.

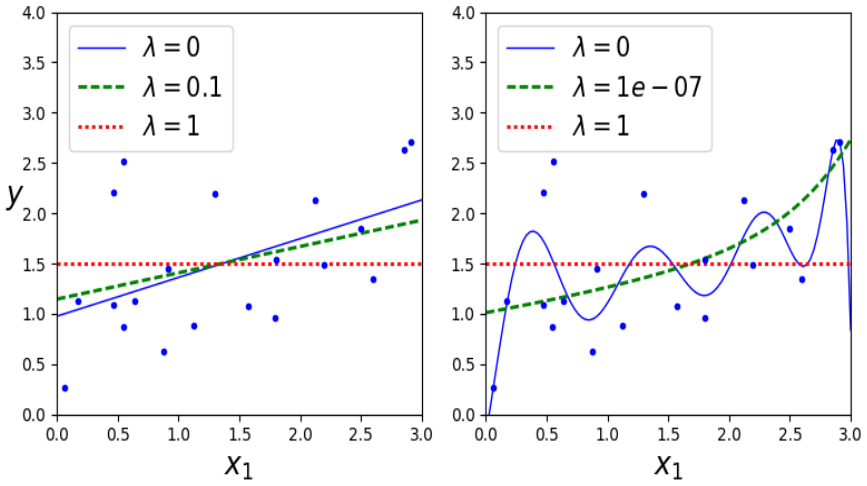
Ridge Regularization:Linear Regression

```
fl=fileloader("input")
rnd.seed(42)
m = 20
X = 3 * rnd.rand(m, 1)
y = 1 + 0.5 * X + rnd.randn(m, 1) / 1.5
X_new = np.linspace(0, 3, 100).reshape(100, 1)

ridge_reg = Ridge(lambdai=1, solver="cholesky")
ridge_reg.fit(X, y)
print(ridge_reg.predict([[1.5]]))

sgd_reg = SGDRegressor(penalty="l2", random_state=42)
sgd_reg.fit(X, y.ravel())
print(sgd_reg.predict([[1.5]]))

ridge_reg = Ridge(lambdai=1, solver="sag")
ridge_reg.fit(X, y)
print(ridge_reg.predict([[1.5]]))
```

- 3 ways of getting this done.
  - closed form solution or the normal equation
  - SGD - with l_2 representing the regularization term.
  - 1/2 the square of l_2 norm of the weight vector.
  - weight vector is $\hat{\theta}$
  - SAG: Stocastic average GD is a variant of SGD.

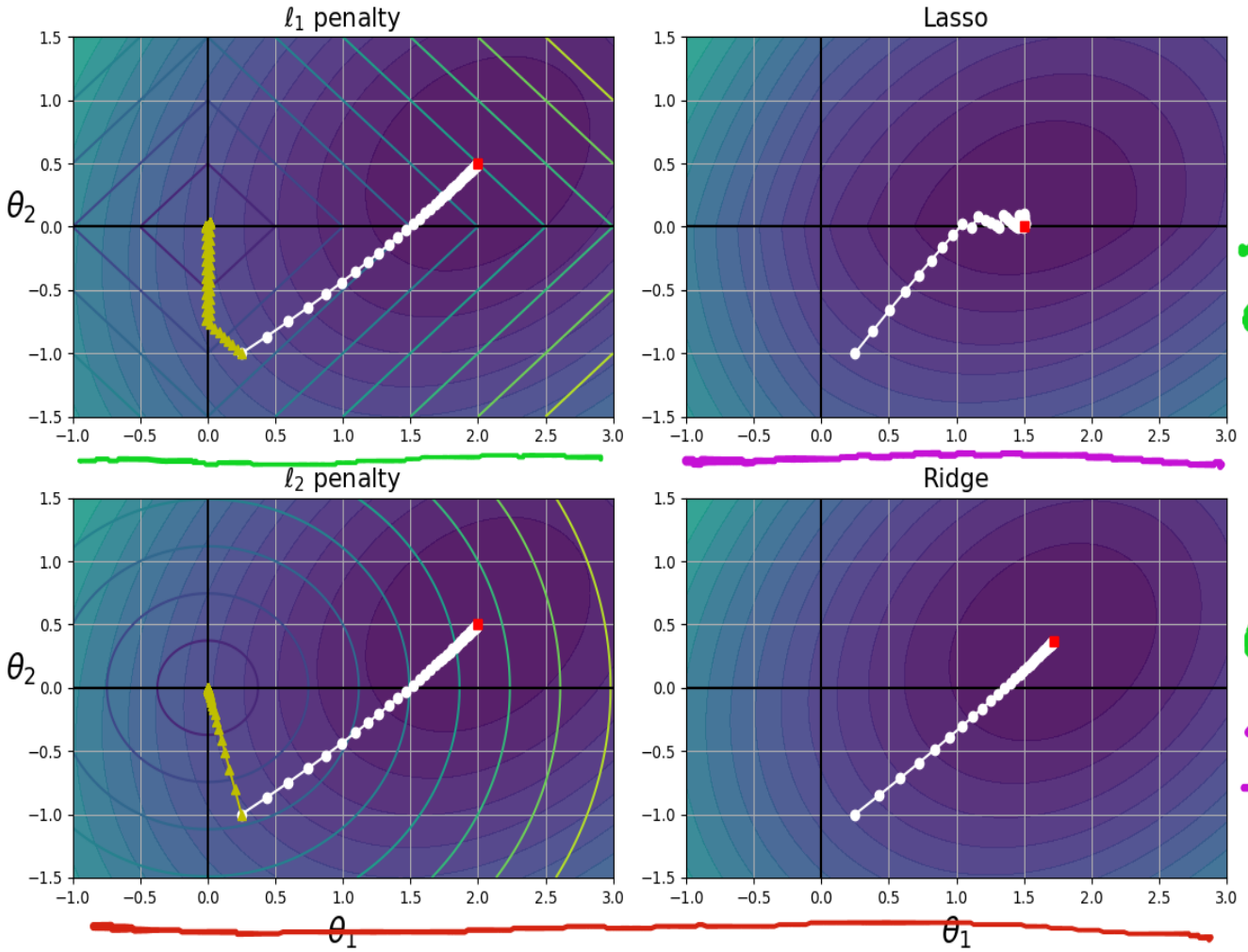
Lasso Regularization:Linear Regression

```
def plot_model(model_class, polynomial, lambdas, **model_kargs):
    for lambdai, style in zip(lambdas, ("b-", "g--", "r:")):
        model = model_class(lambdai, **model_kargs) if lambdai > 0 else LinearRegression()
        if polynomial:
            model = Pipeline((
                    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ))
        model.fit(X, y)
        y_new_regul = model.predict(X_new)
        lw = 2 if lambdai > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\lambda = {}$".format(lambdai))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 4])

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Lasso, polynomial=False, lambdas=(0, 0.1, 1))
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Lasso, polynomial=True, lambdas=(0, 10**-7, 1), tol=1)
```



- Lasso Regression.
  - Identical to Ridge except uses l_1 norm.
  - It completely eliminates the weights of least important features.
  - for $\alpha = 10^{-7}$ looks quadratic
  - In other words it performs feature selection and outputs a sparse model.

Lasso vs Ridge Regularization:Linear Regression



- Same thing for ridge regression.

TL

- Background elipses represent unregularized cost function. BGD denoted by white circles. ($\theta_1 = 2.0$, $\theta_2 = -.5$)
- ◇ represent L_1 penalty with $\lambda \to \alpha$. ($\theta_1 \approx 0$ $\theta_2 \approx 0$)

TR

- $J(\theta) = MSE(\theta) + \lambda \sum_{i=1}^{n} |\theta_i|$
- when $\lambda = 0.5$, it doesn't just minimize the cost; it does it at lowest possible θ. ($\theta_1 = 1.5$, $\theta_2 = 0$)

Lasso vs Ridge Regularization:Linear Regression

```
rnd.seed(42)
m = 20
X = 3 * rnd.rand(m, 1)
y = 1 + 0.5 * X + rnd.randn(m, 1) / 1.5
X_new = np.linspace(0, 3, 100).reshape(100, 1)

lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
print(lasso_reg.predict([[1.5]]))

elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X, y)
print(elastic_net.predict([[1.5]]))
```

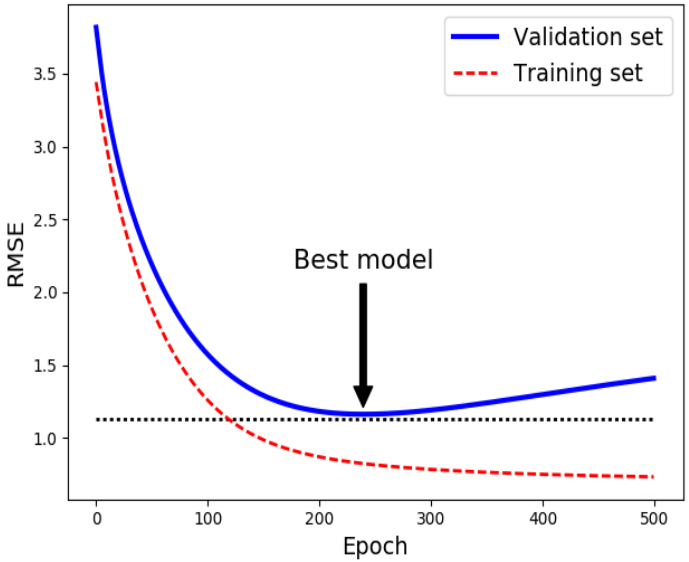
- Lasso Regression
- One could also use
  SGDRegressor(penalty='l1')
- ElasticNet is the middle ground between Ridge and lasso.

$$J(\theta) = MSE(\theta) + r\lambda \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\lambda \sum_{i=1}^{n} \theta_i^2$$

- r is the mix ratio. When r = 1 it is only lasso regression.

Lasso vs Ridge Regularization:Linear Regression

```
for epoch in range(n_epochs):
    sgd_reg.fit(X_train_poly_scaled, y_train)
    y_train_predict = sgd_reg.predict(X_train_poly_scaled)
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    train_errors.append(mean_squared_error(y_train_predict, y_train))
    val_errors.append(mean_squared_error(y_val_predict, y_val))

best_epoch = np.argmin(val_errors)
best_val_rmse = np.sqrt(val_errors[best_epoch])

print("best_epoch:",best_epoch,"best_val_rmse:",best_val_rmse)
plt.annotate('Best model',
             xy=(best_epoch, best_val_rmse),
             xytext=(best_epoch, best_val_rmse + 1),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.05),
             fontsize=16,
            )

best_val_rmse -= 0.03  # just to make the graph look better
plt.plot([0, n_epochs], [best_val_rmse, best_val_rmse], "k:", linewidth=2)
plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Validation set")
plt.plot(np.sqrt(train_errors), "r--", linewidth=2, label="Training set")
plt.legend(loc="upper right", fontsize=14)
```



- Hopelessly simple trick that works.