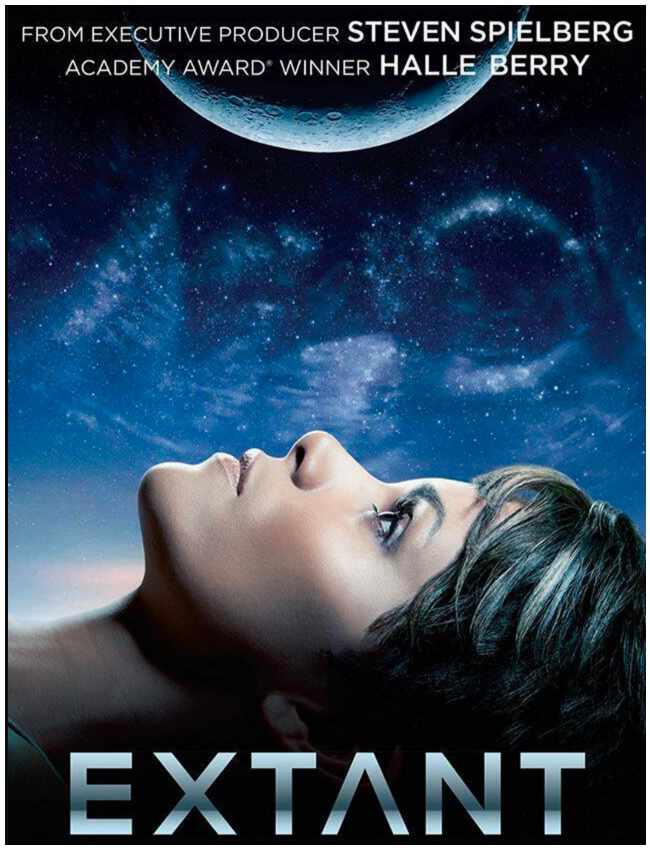


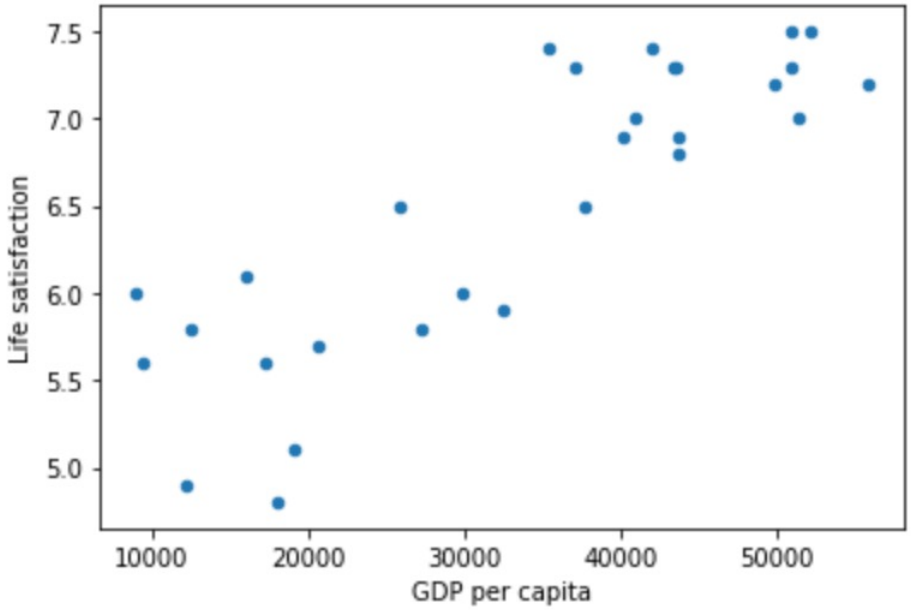
Machine Learning

Linear Regression

Country	GDP per capita	Life satisfaction
Brazil	8669.998	7.0
Mexico	9009.280	6.7
Russia	9054.914	6.0
Turkey	9437.372	5.6
Hungary	12239.894	4.9
Poland	12495.334	5.8
Chile	13340.905	6.7
Slovak Republic	15991.736	6.1
Czech Republic	17256.918	6.5
Estonia	17288.083	5.6
Greece	18064.288	4.8
Portugal	19121.592	5.1
Slovenia	20732.482	5.7
Spain	25864.721	6.5
Korea	27195.197	5.8
Italy	29866.581	6.0
Japan	32485.545	5.9
Israel	35343.336	7.4
New Zealand	37044.891	7.3

Regression Problem

Predict real-valued output



Country	GDP per capita	Life satisfaction
Brazil	8669.998	7.0
Mexico	9009.280	6.7
Russia	9054.914	6.0
Turkey	9437.372	5.6
Hungary	12239.894	4.9
Poland	12495.334	5.8
Chile	13340.905	6.7
Slovak Republic	15991.736	6.1
Czech Republic	17256.918	6.5
Estonia	17288.083	5.6
Greece	18064.288	4.8
Portugal	19121.592	5.1
Slovenia	20732.482	5.7
Spain	25864.721	6.5
Korea	27195.197	5.8
Italy	29866.581	6.0
Japan	32485.545	5.9
Israel	35343.336	7.4
New Zealand	37044.891	7.3

Regression Problem

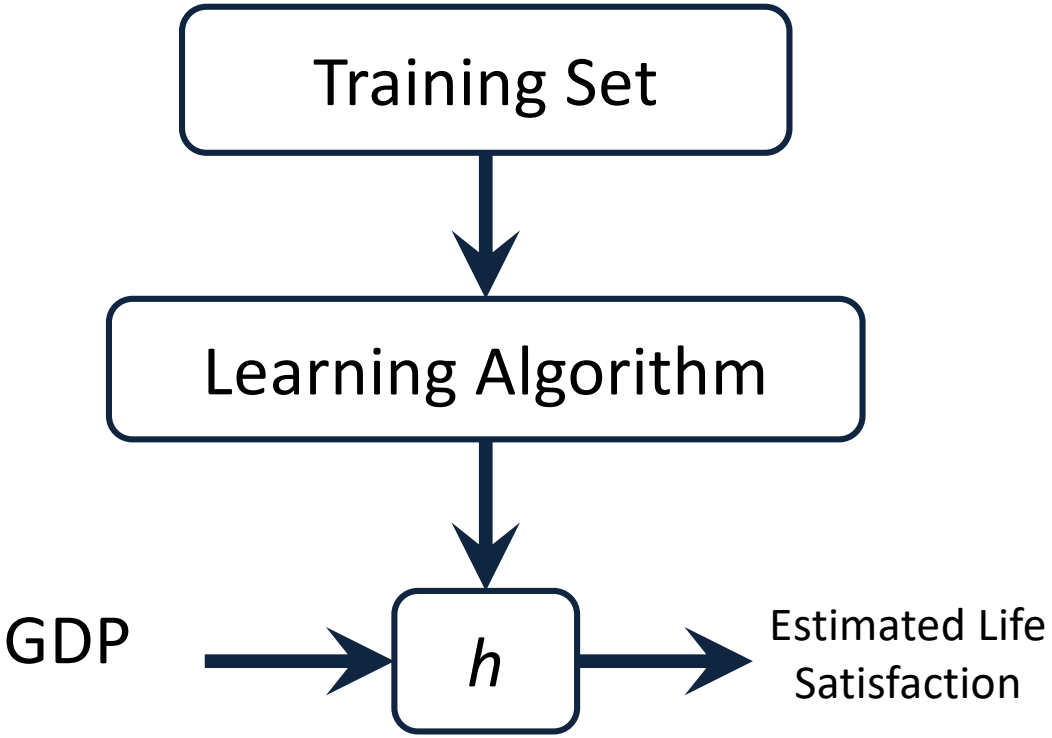
Predict real-valued output

Notation:

m = Number of training examples

x's = "input" variable / features

y's = "output" variable / "target" variable



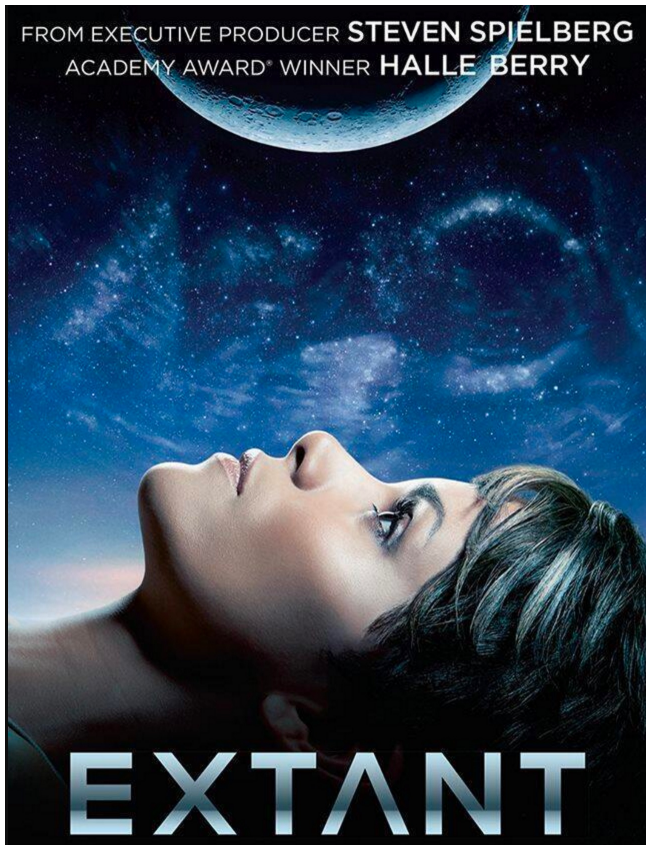
How do we represent *h* ?

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

θ_i's: Parameters

How to choose θ_i's ?

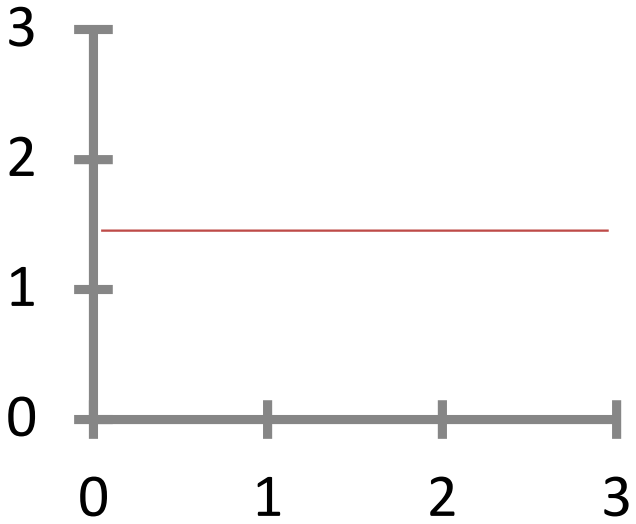
Linear regression with one variable.
Univariate linear regression.



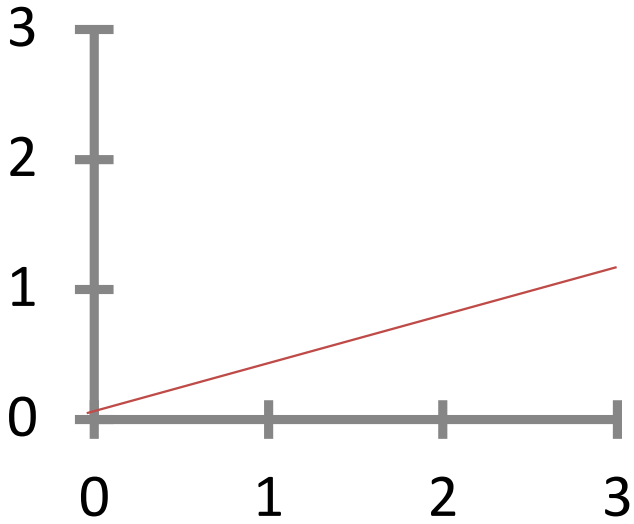
Machine Learning

Cost function

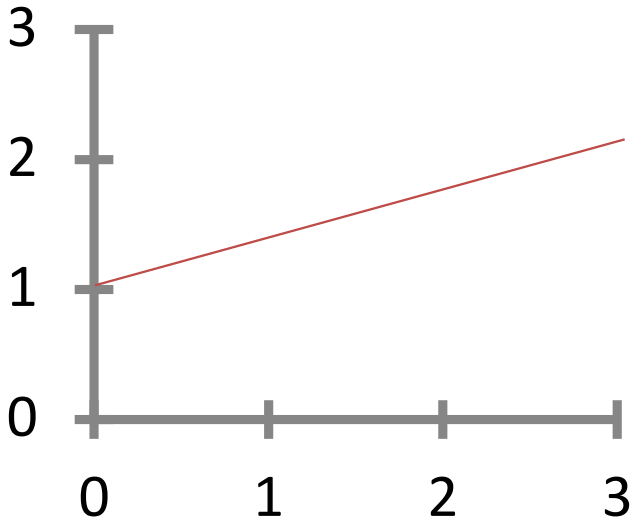
$$h_\theta(x) = \theta_0 + \theta_1 x$$



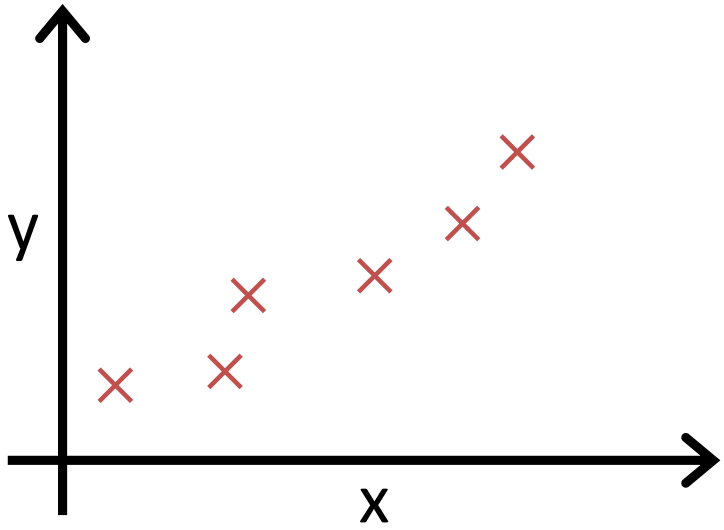
$\theta_0 = 1.5$
$\theta_1 = 0$



$\theta_0 = 0$
$\theta_1 = 0.5$



$\theta_0 = 1$
$\theta_1 = 0.5$



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

Simplified

Hypothesis:

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

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

Parameters:

$$\theta_0, \theta_1$$

$$\theta_1$$

Cost Function:

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

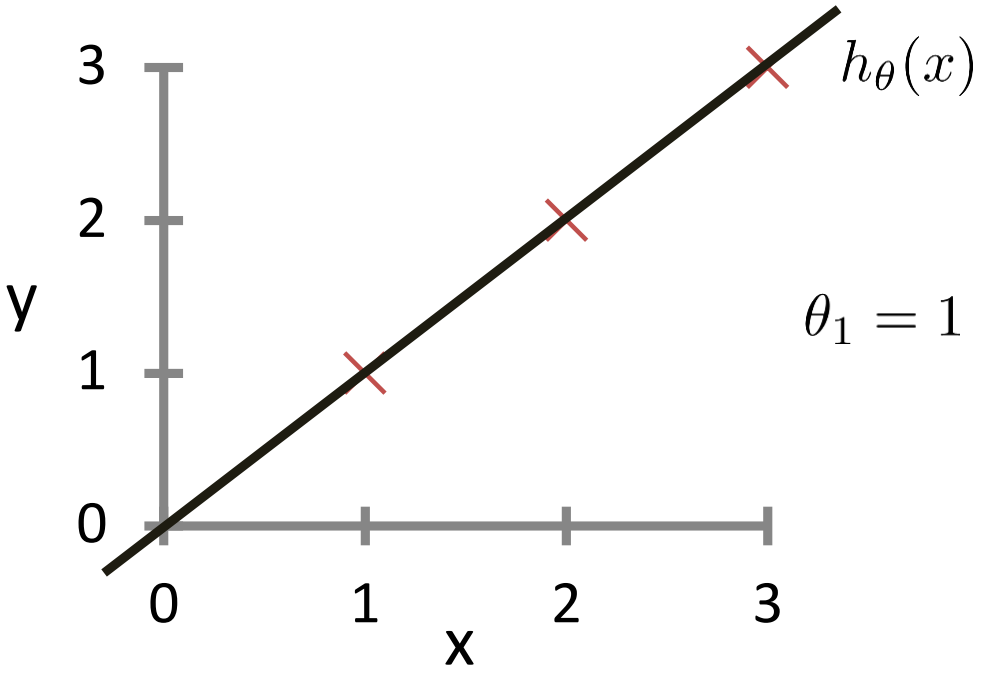
$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

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

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

$h_\theta(x)$

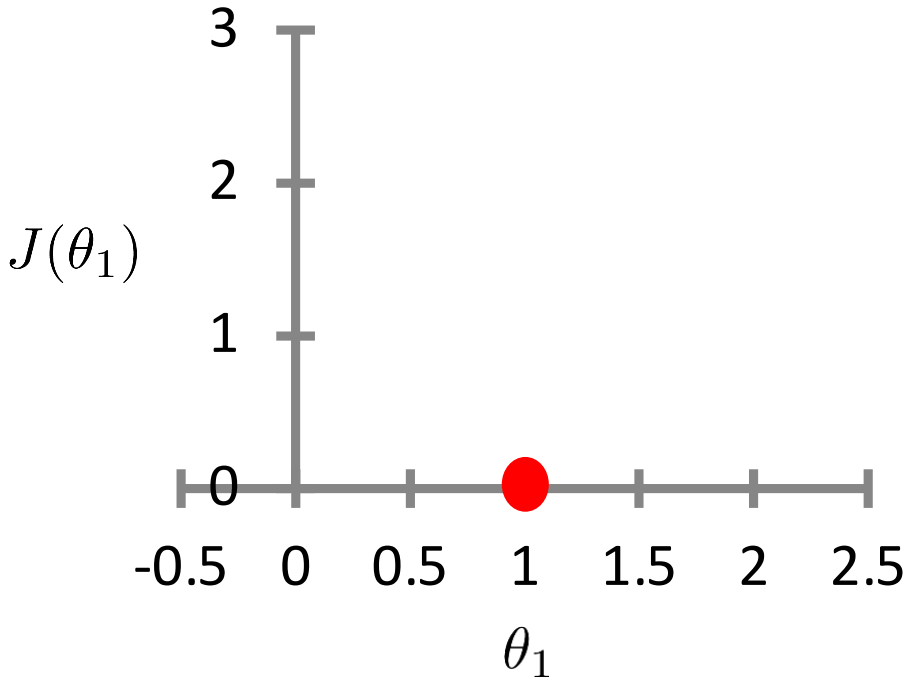
(for fixed θ_1, this is a function of x)



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

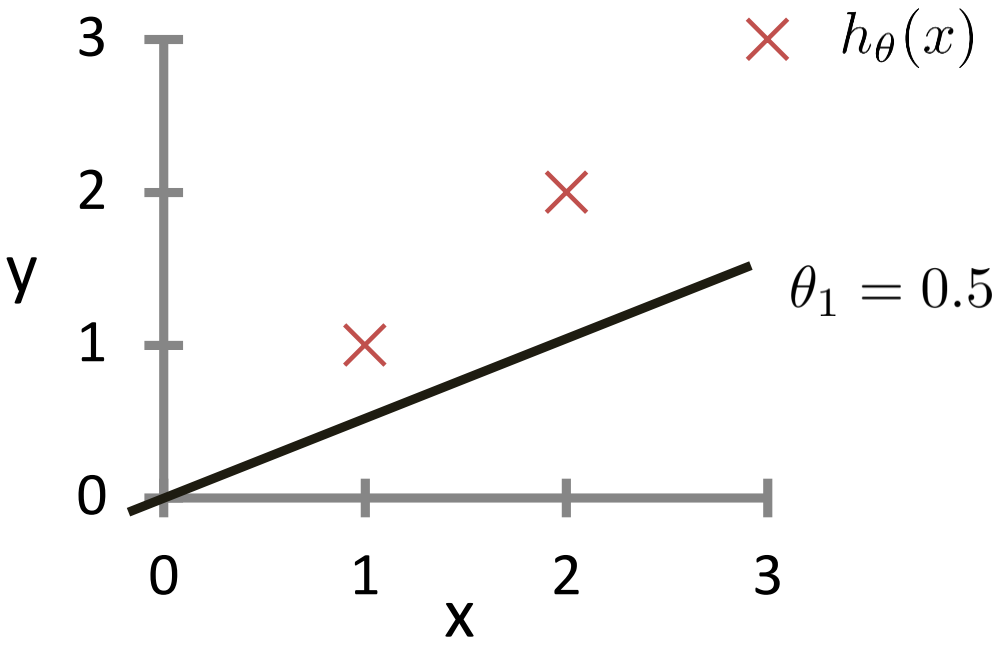
$J(\theta_1)$

(function of the parameter θ_1)



$h_\theta(x)$

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

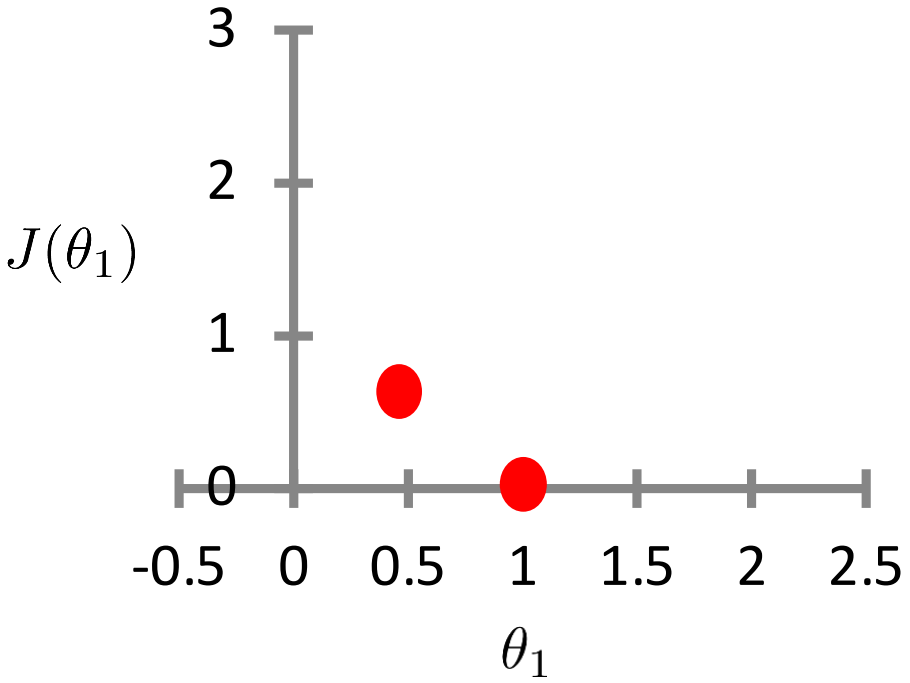


$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

J(0.5) = 1/2m $[(0.5-1)^2+(1-2)^2+(1.5-3)^2] = 0.58$

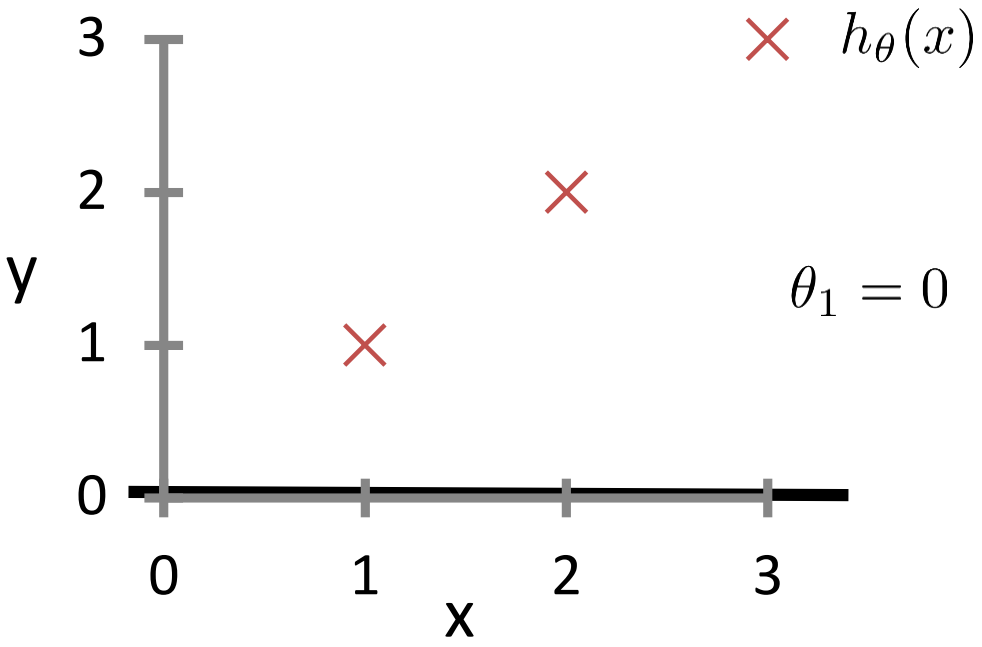
$J(\theta_1)$

(function of the parameter θ_1)



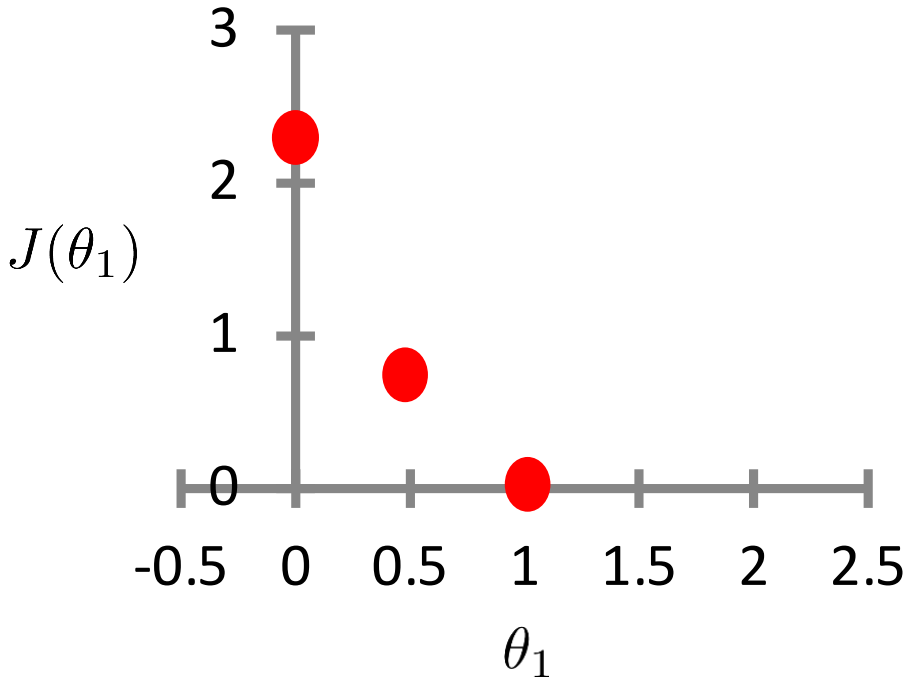
$h_\theta(x)$

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



$J(\theta_1)$

(function of the parameter θ_1)



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

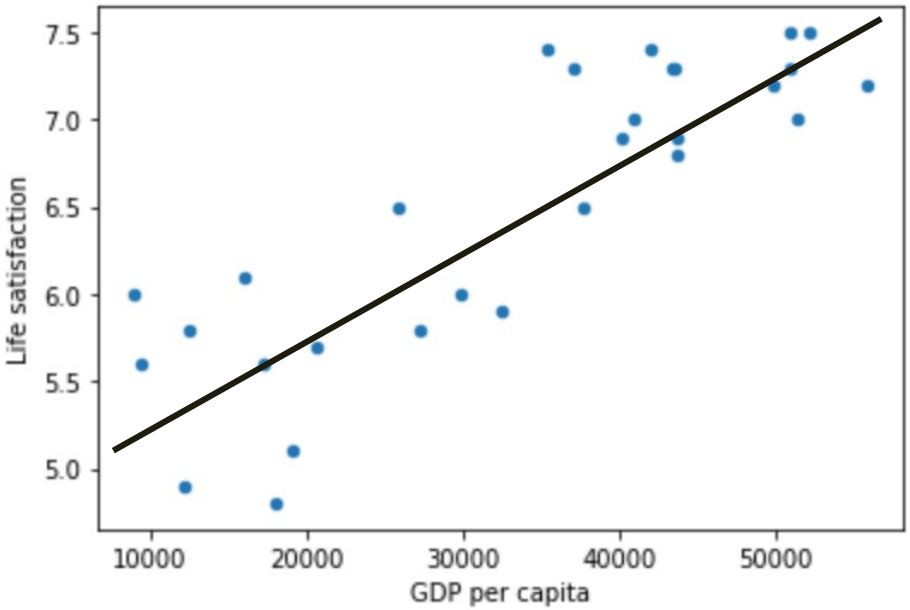
Parameters: θ_0, θ_1

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

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

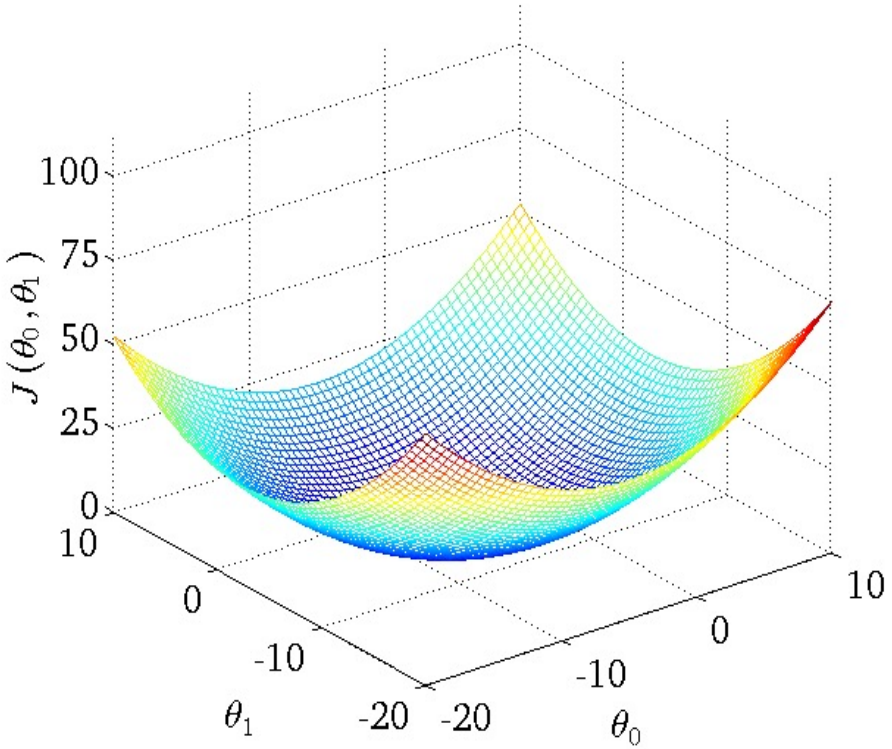
$h_\theta(x)$

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



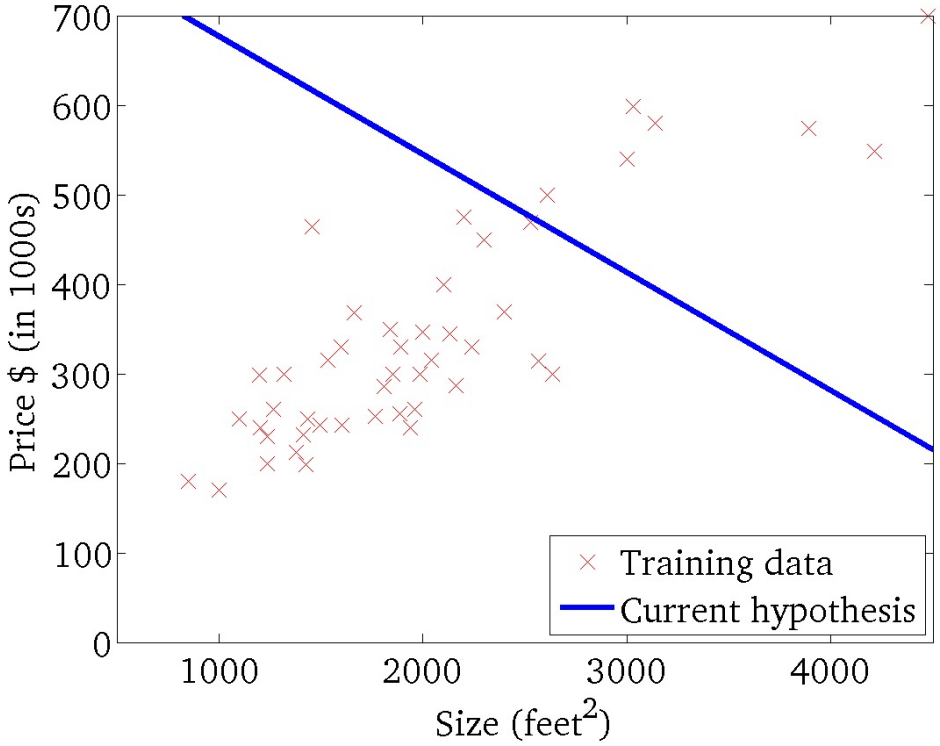
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



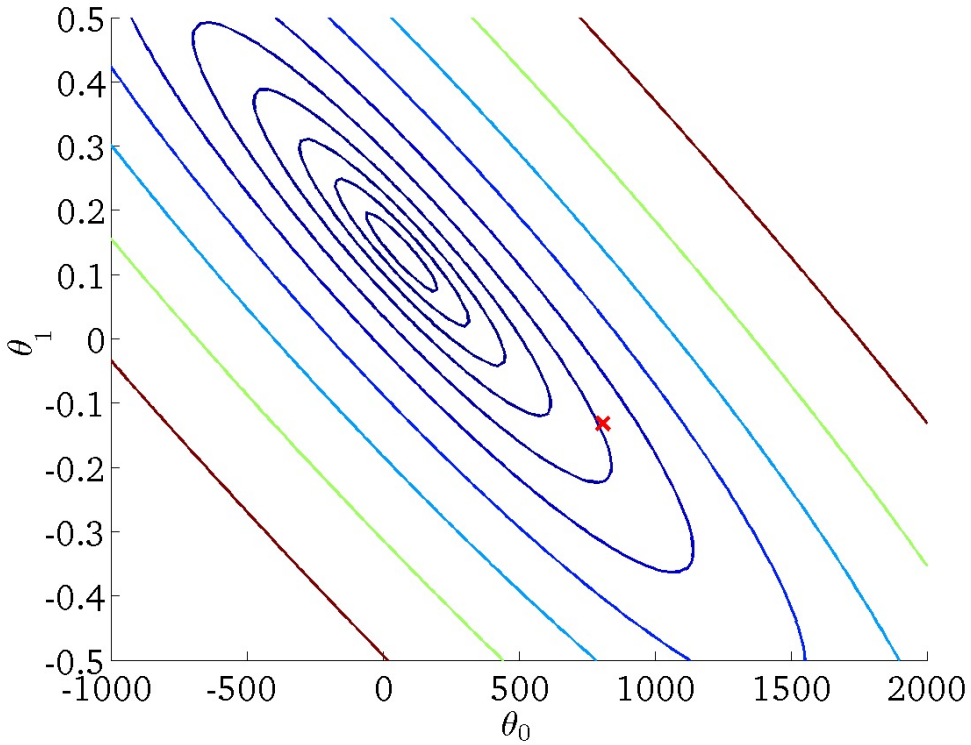
$h_\theta(x)$

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



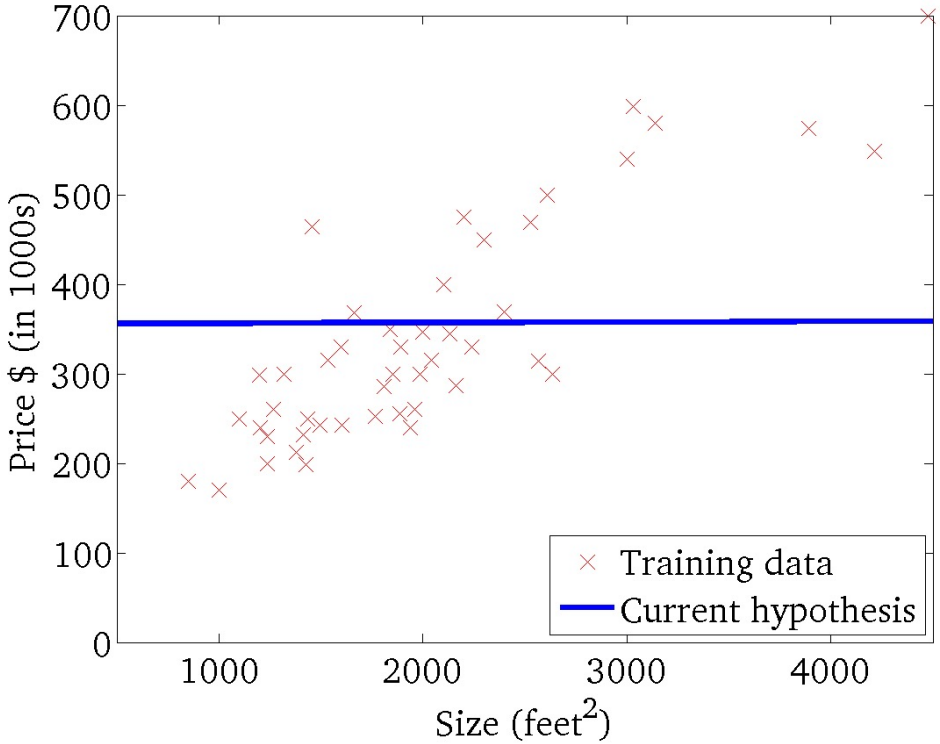
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



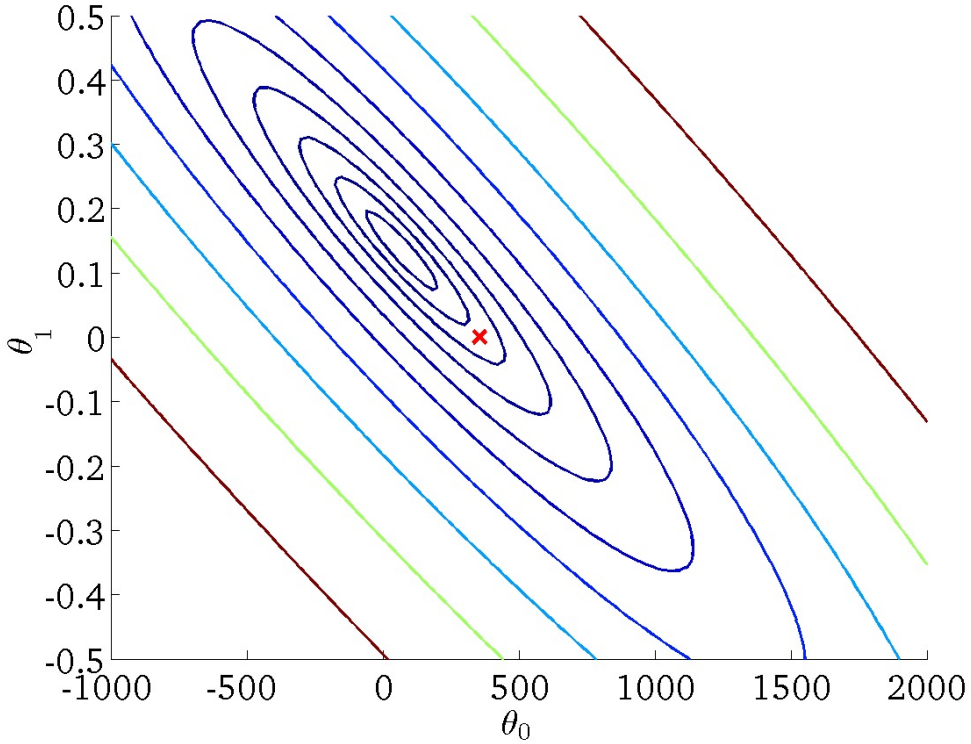
$h_\theta(x)$

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



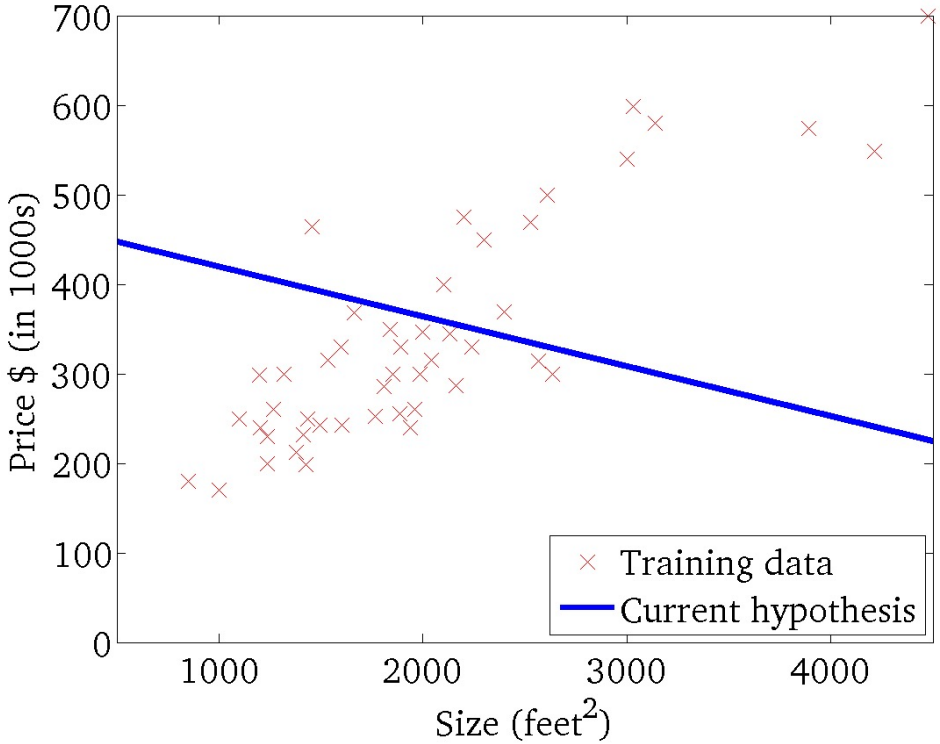
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



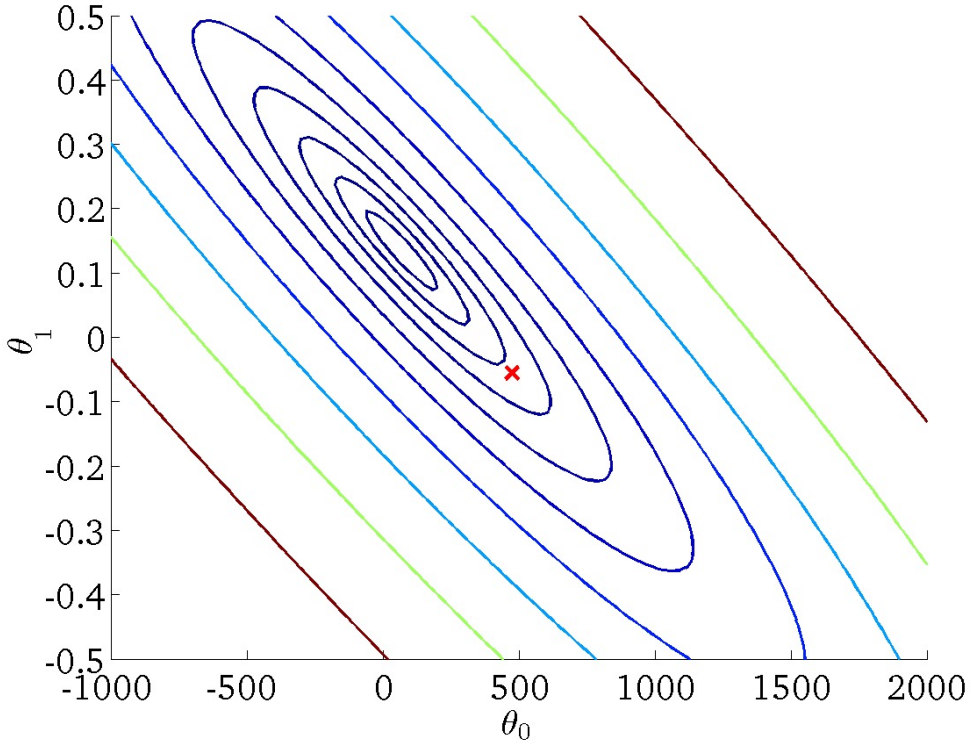
$h_\theta(x)$

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



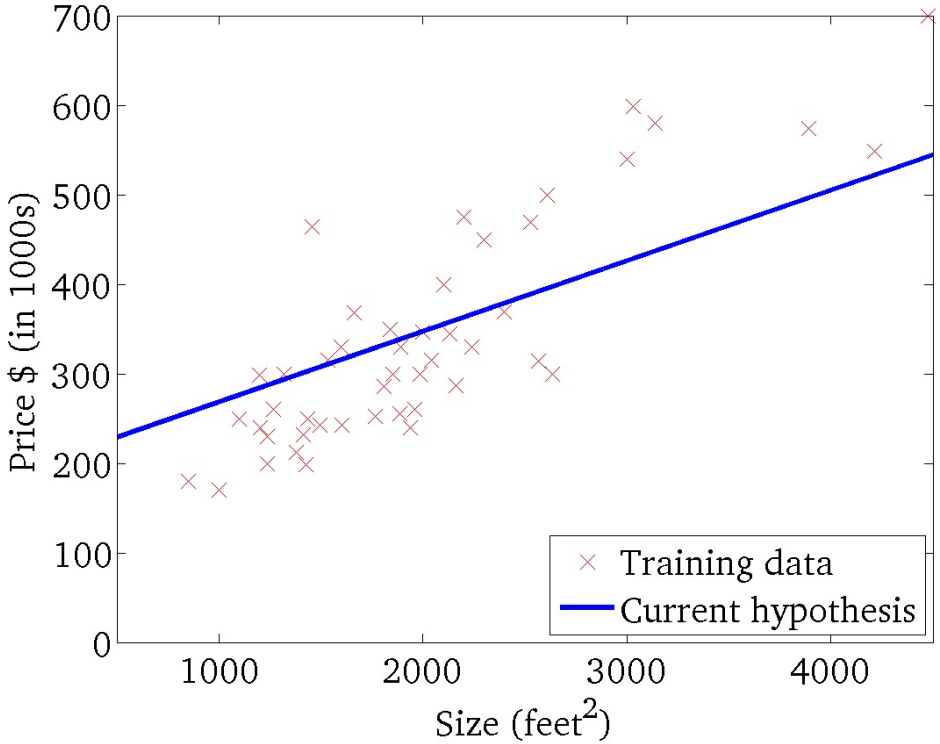
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



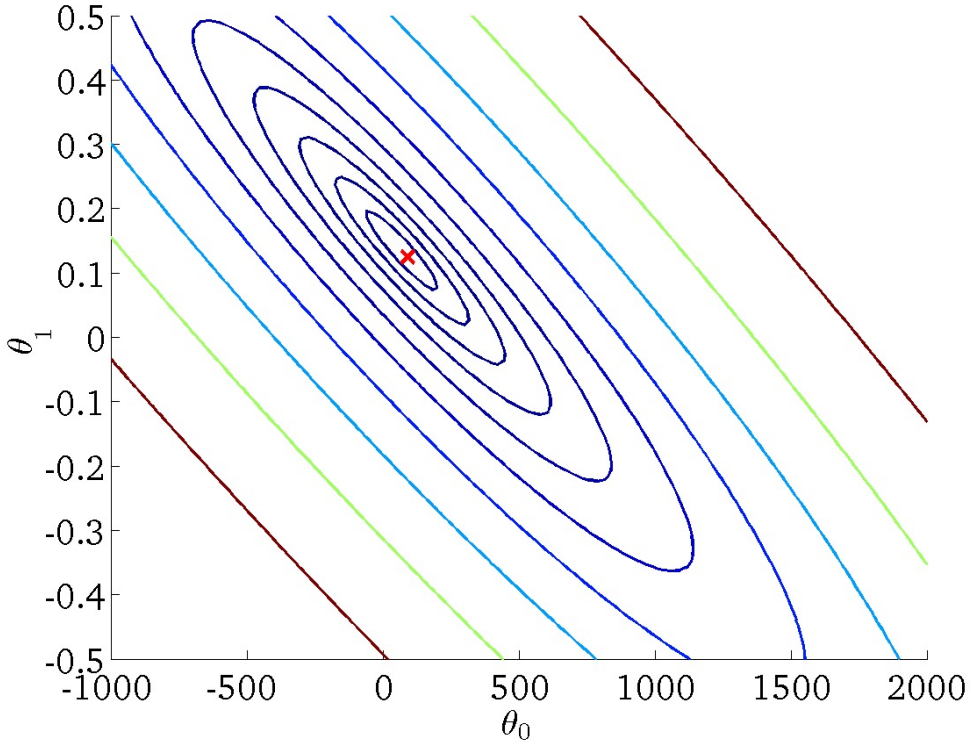
$h_\theta(x)$

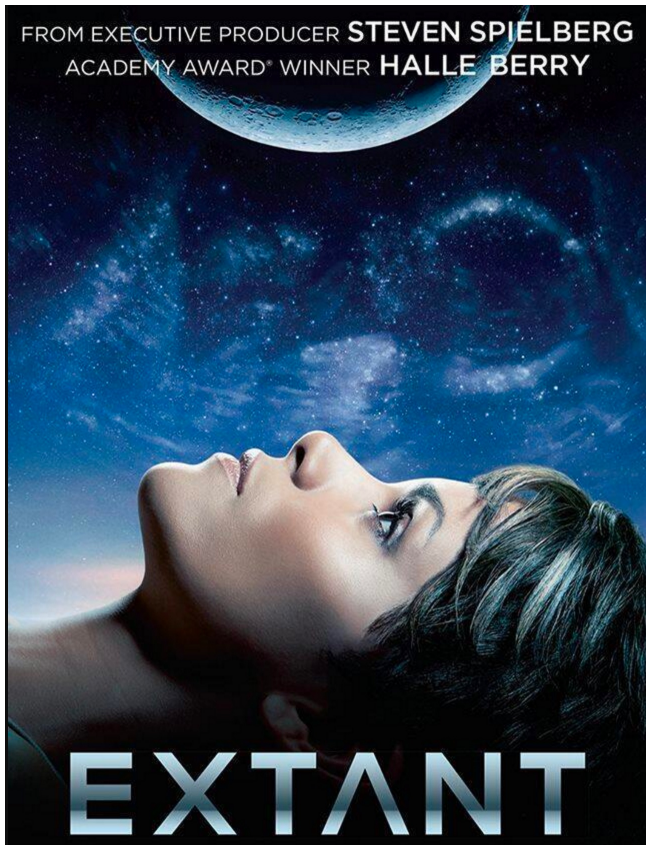
(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)





Machine Learning

Solution for Cost Function

Closed-Form Equation

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

m = 3 即， (x^1, y^1)， (x^2, y^2)， (x^3, y^3)

$$\boldsymbol{X\theta} = \begin{pmatrix} 1 & x^1 \\ 1 & x^2 \\ 1 & x^3 \end{pmatrix} \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix} = \begin{pmatrix} \theta_0 + \theta_1 x^1 \\ \theta_0 + \theta_1 x^2 \\ \theta_0 + \theta_1 x^3 \end{pmatrix}$$

$$J(Q) = (X\theta - y)^T (X\theta - y)$$
$$= \theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y$$

symmetric matrix $A^T = A$

$$J(Q) = \theta^T X^T X\theta - 2\theta^T X^T y + y^T y$$

$$\frac{dJ(Q)}{d\theta} = -2y^T X + \theta^T X^T X + \theta^T (X^T X)^T$$
$$= -2y^T X + 2\theta^T X^T X = 0$$

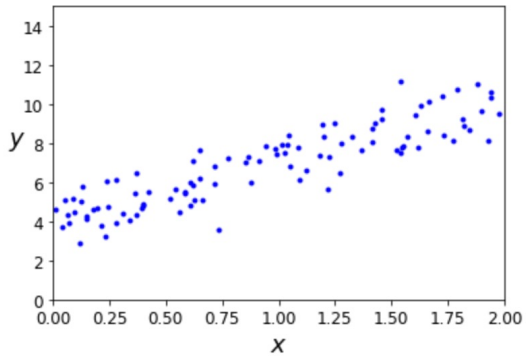
$$\Rightarrow 2y^T X = 2\theta^T X^T X \Rightarrow \theta^T = y^T X\ (X^T X)^{-1}$$
$$\Rightarrow \theta = (X^T X)^{-1} X^T y$$

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

```
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)
```

```
import numpy as np

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

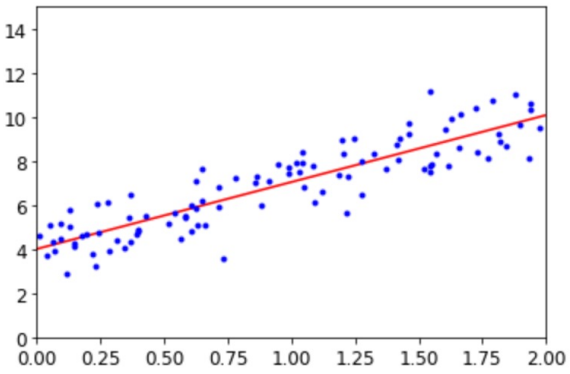
```
plt.plot(X, y, "b.")
plt.xlabel("$x$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```

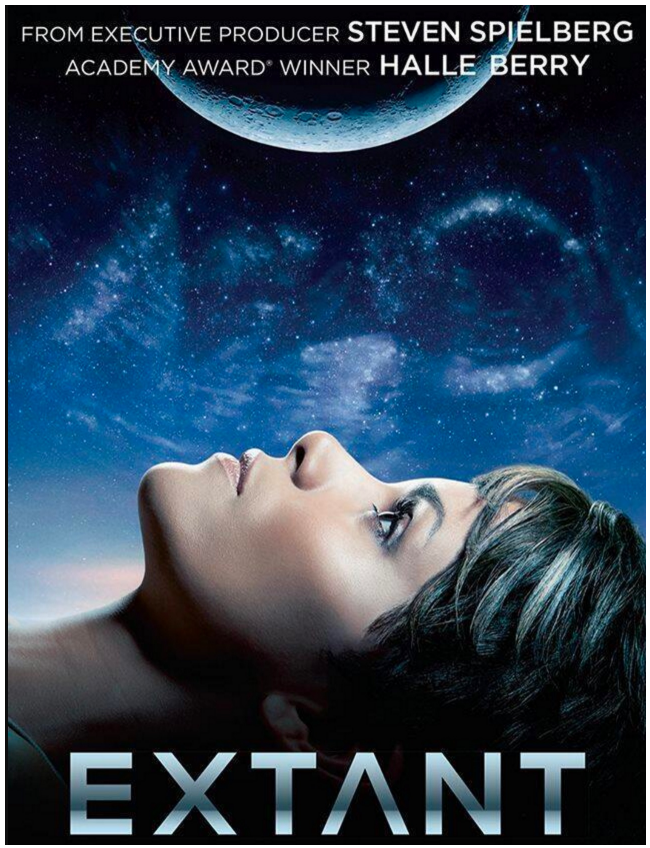



```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
# theta_best
```

```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
y_predict = X_new_b.dot(theta_best)
# X_new,y_predict
```

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```





Machine Learning

Solution for Cost Function

Gradient descent

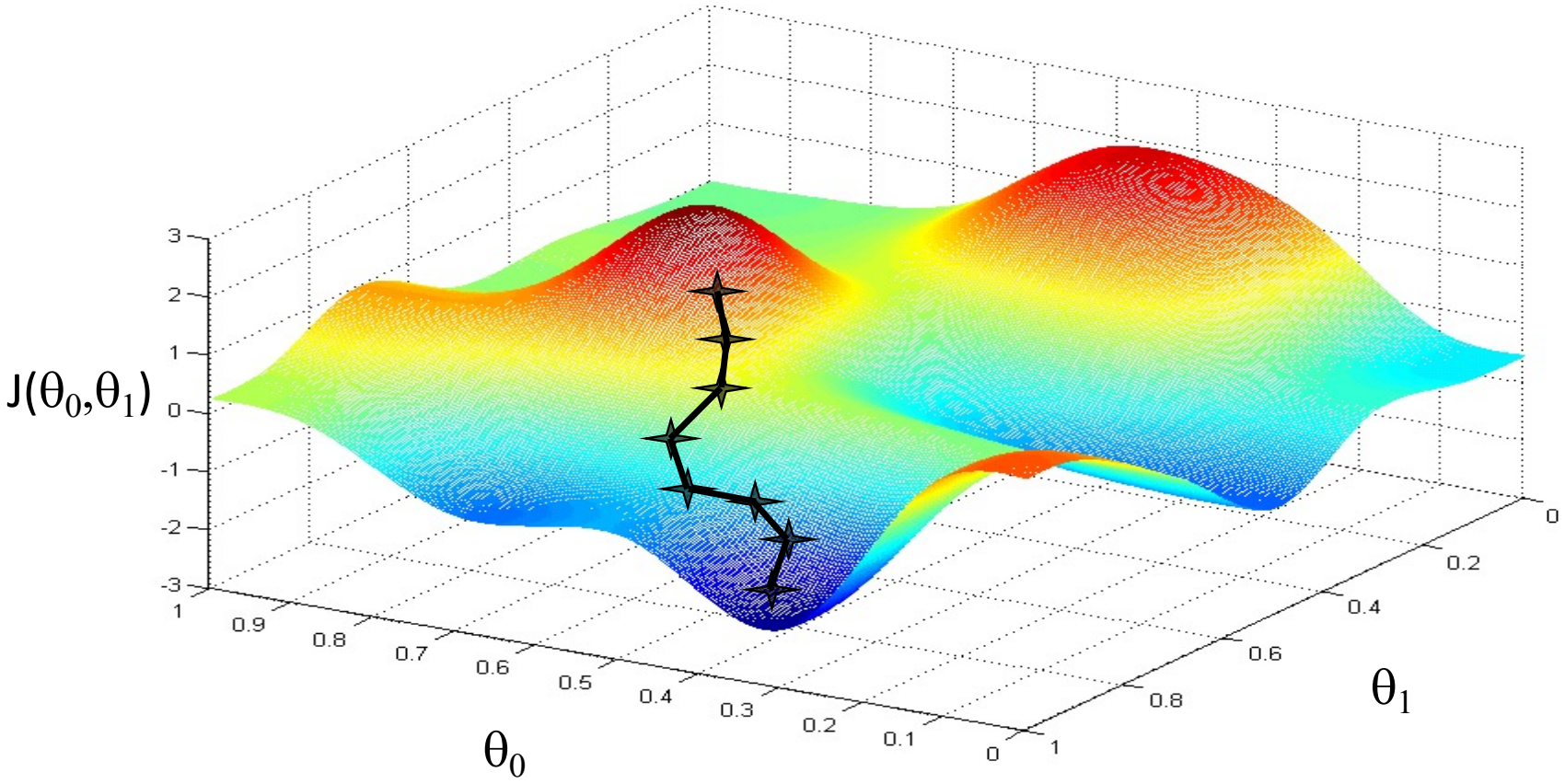
Have some function $J(\theta_0, \theta_1)$

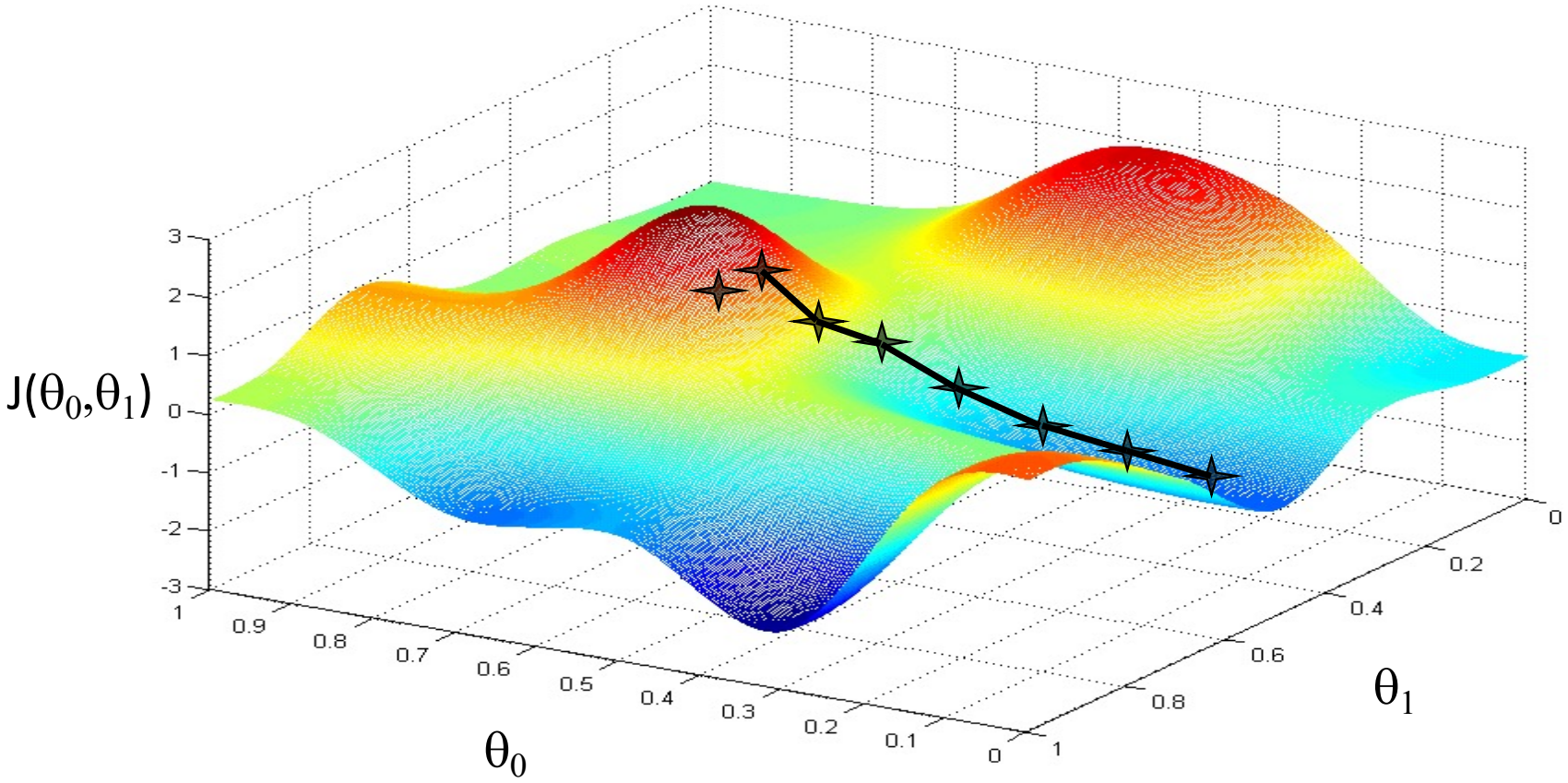
Want $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

Outline:

- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$

  until we hopefully end up at a minimum


J(θ0,θ1)
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ0
0
0.2
0.4
0.6
0.8
1
θ1


J(θ0,θ1)
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ0
0
0.2
0.4
0.6
0.8
1
θ1

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \qquad (\text{for } j = 0 \text{ and } j = 1)$$

}

Correct: Simultaneous update

$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\theta_1 := \text{temp1}$$

Incorrect:

$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_1 := \text{temp1}$$

```
In [9]: from sklearn.linear_model import LinearRegression

        lin_reg = LinearRegression()
        lin_reg.fit(X, y)
        lin_reg.intercept_, lin_reg.coef_

Out[9]: (array([4.21509616]), array([[2.77011339]]))

In [10]: lin_reg.predict(X_new)

Out[10]: array([[4.21509616],
                [9.75532293]])
```

The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name stands for "least squares"), which you could call directly:

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-
6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

This function computes $\hat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$, where $\mathbf{X}^{+}$ is the ***pseudoinverse*** of X (specifically, the Moore-Penrose inverse). You can use `np.linalg.pinv()` to compute the pseudoinverse directly:

```
>>> np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])
```

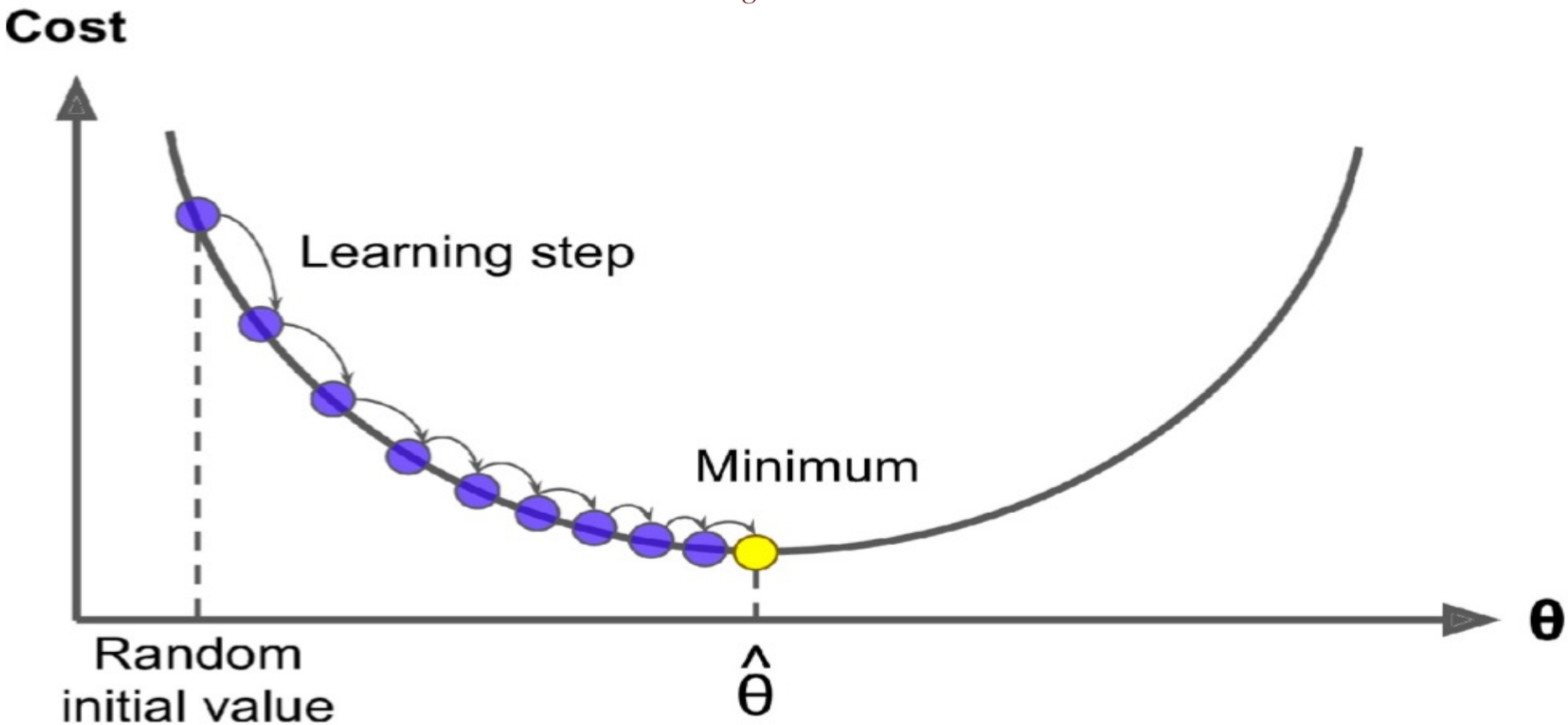
The pseudoinverse itself is computed using a standard matrix factorization technique called Singular Value Decomposition (SVD) that can decompose the training set matrix X into the matrix multiplication of three matrices U Σ Vᵀ .

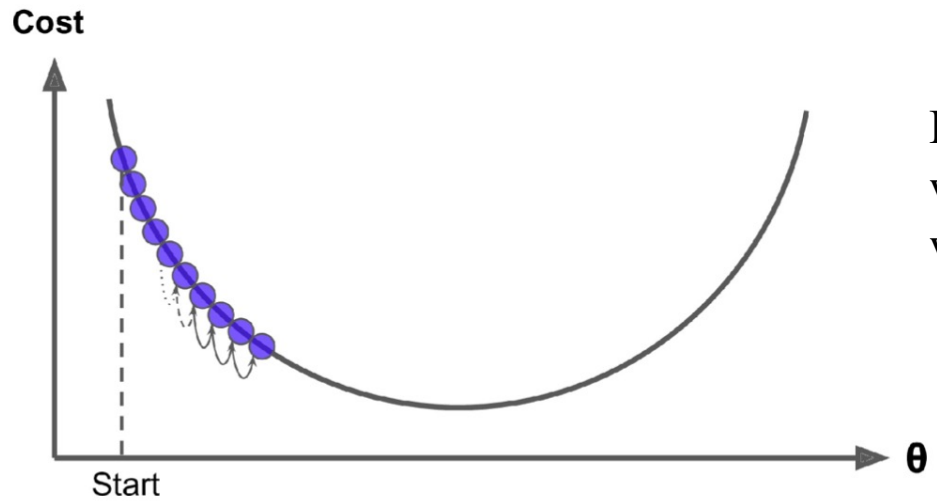
repeat until convergence {

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

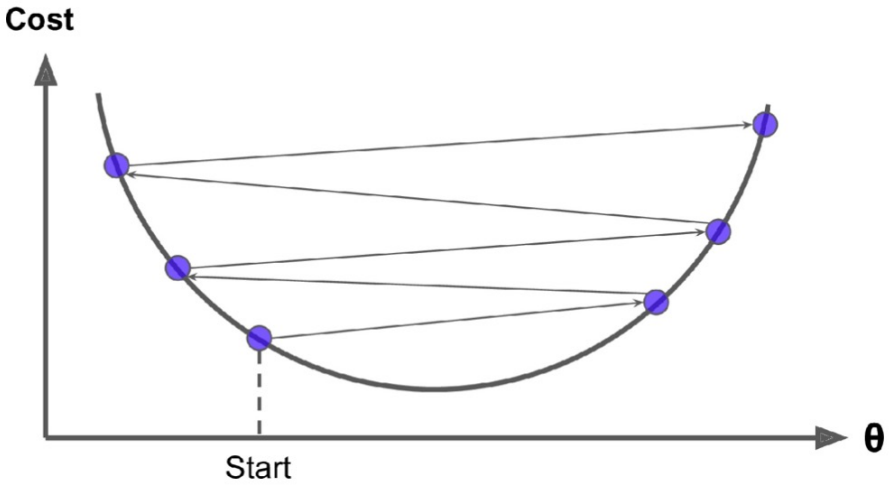
}

Learning rate





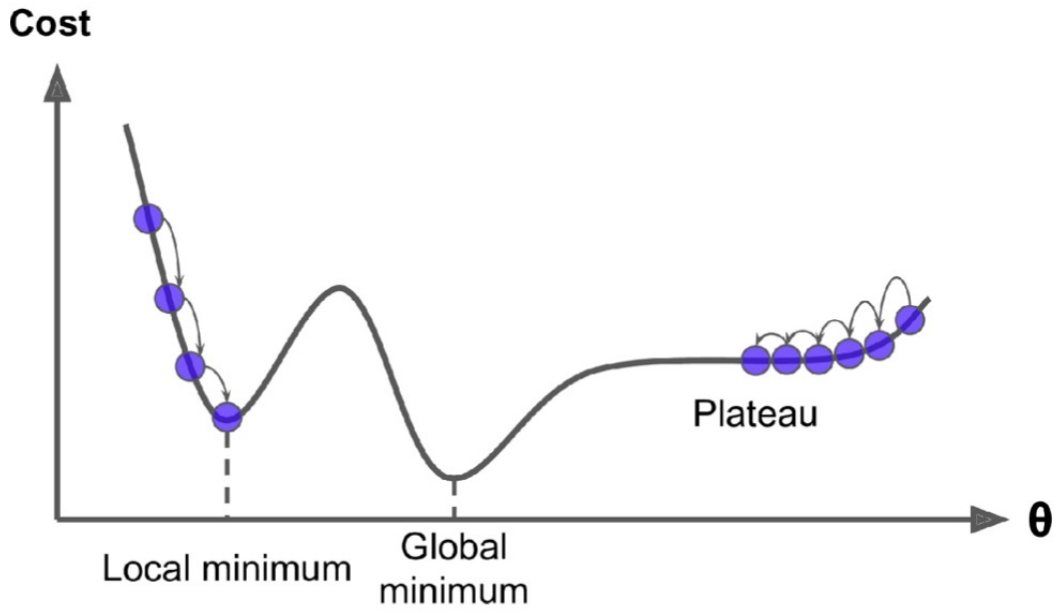
If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time



If the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values

- If the random initialization starts the algorithm on the left, then it will converge to a ***local minimum***.

- If it starts on the right, then it will take a very long time to cross the plateau. And if you stop too early, you will never reach the global minimum.



Convex Function

The MSE cost function for a Linear Regression model happens to be a ***convex function.*** If you pick any two points on the curve, the line segment joining them never crosses the curve

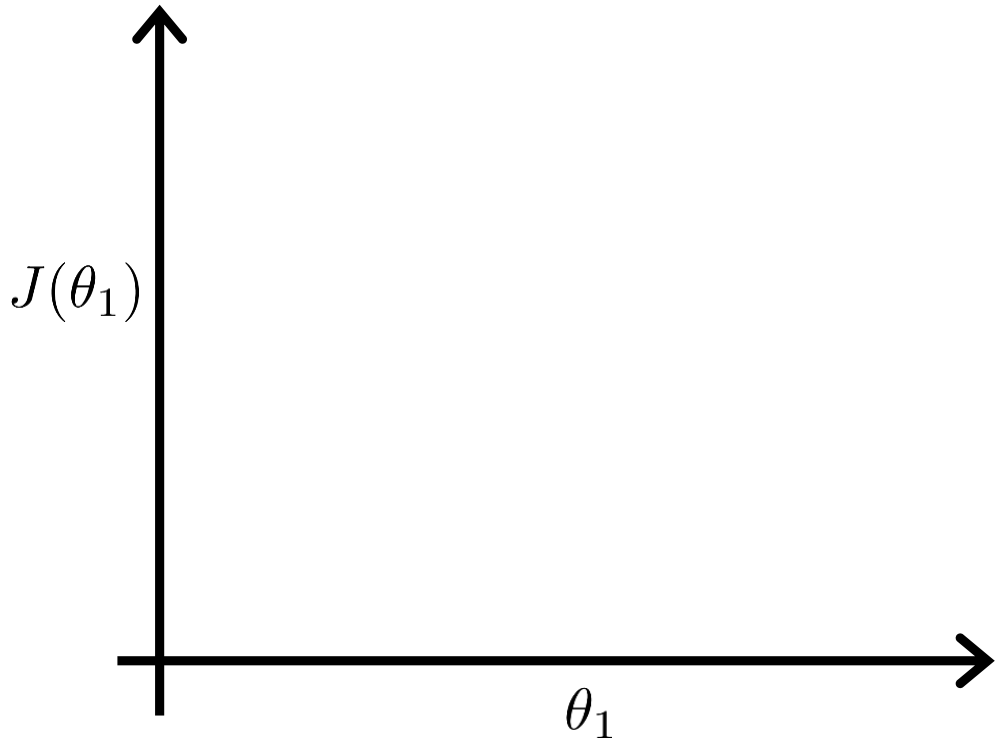
This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly.

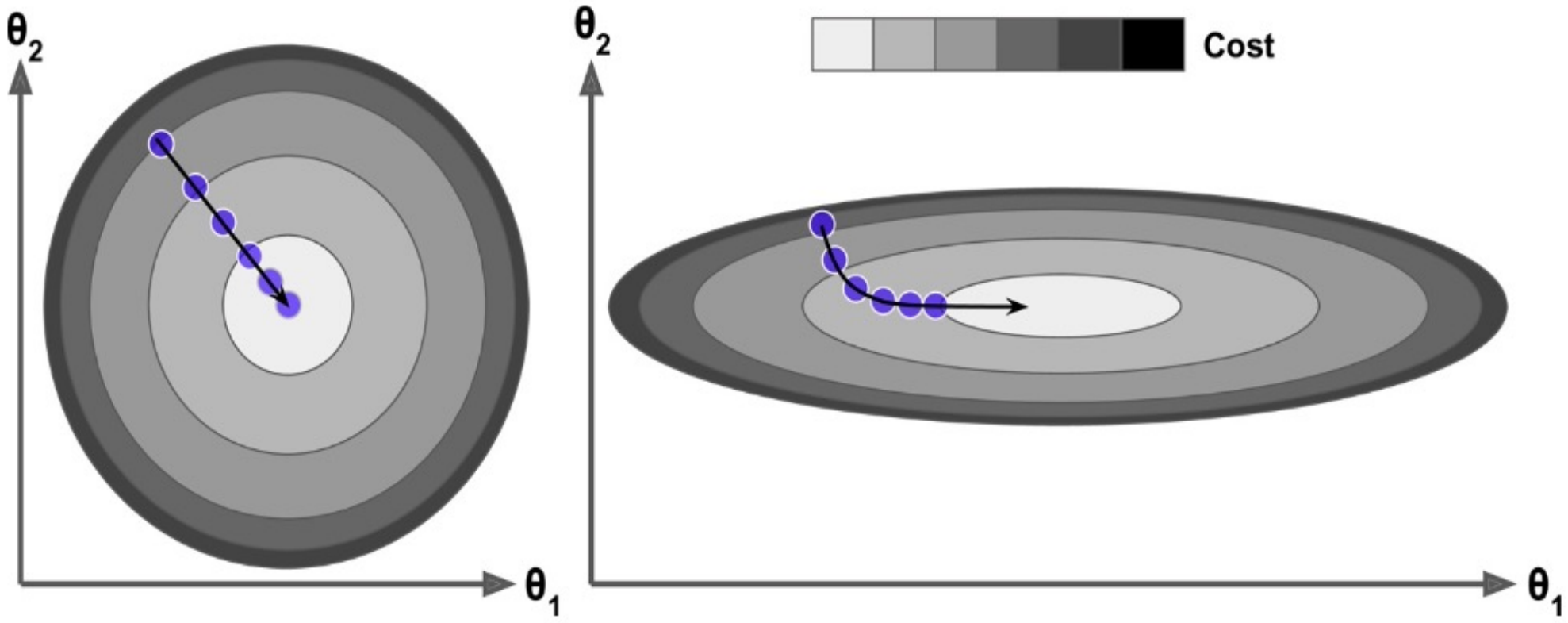
Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

Gradient descent can converge to a local minimum, even with the learning rate α fixed.

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

As we approach a local minimum, gradient descent will automatically take smaller steps. So, no need to decrease α over time.





WARNING

When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

Gradient descent algorithm

repeat until convergence {

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

(for $j = 1$ and $j = 0$)

}

Linear Regression Model

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

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

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

$$j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) =$$

$$j = 1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) =$$

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

$$= \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)}\right)^2$$

$$j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)$$

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

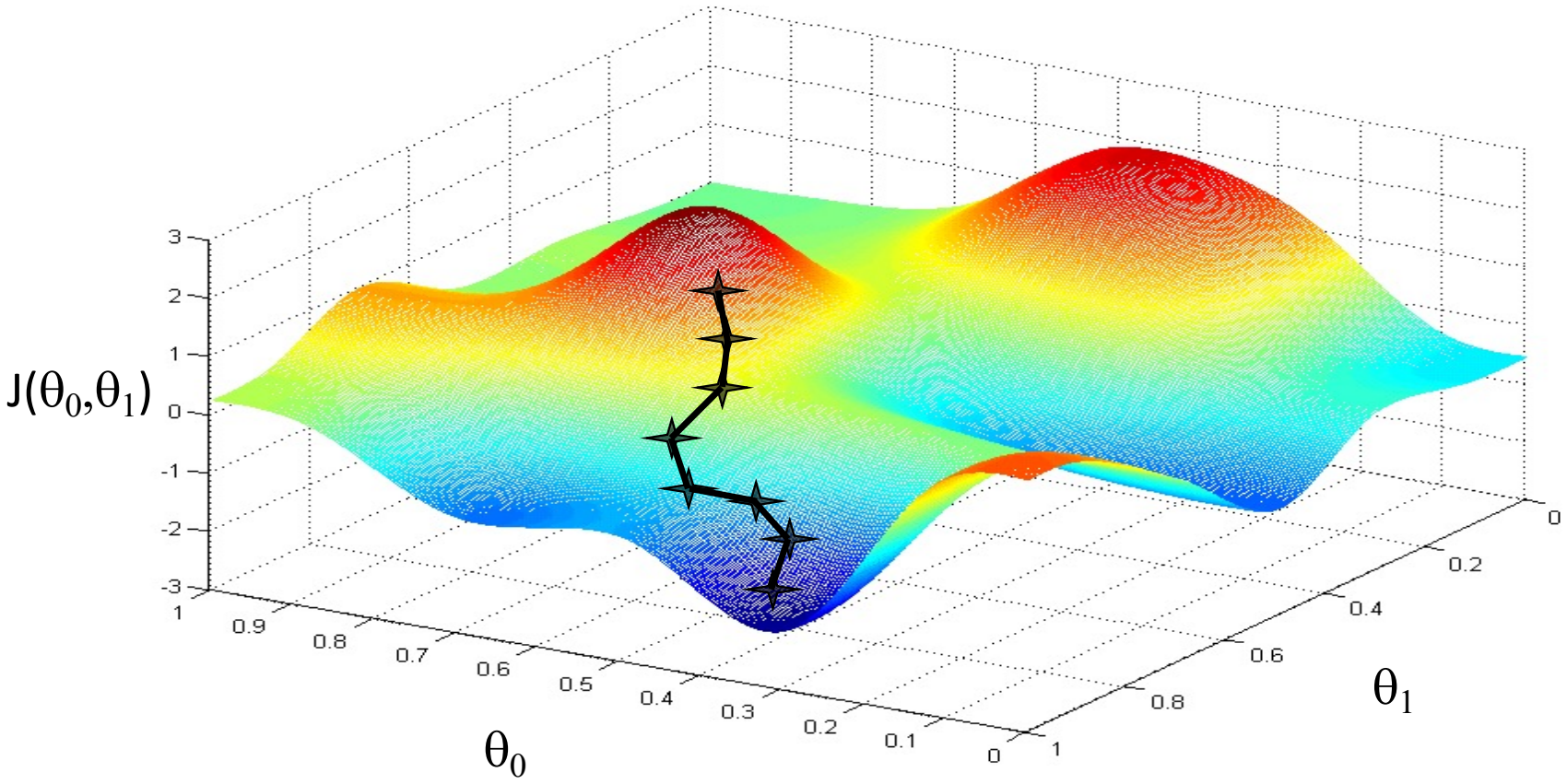
Gradient descent algorithm

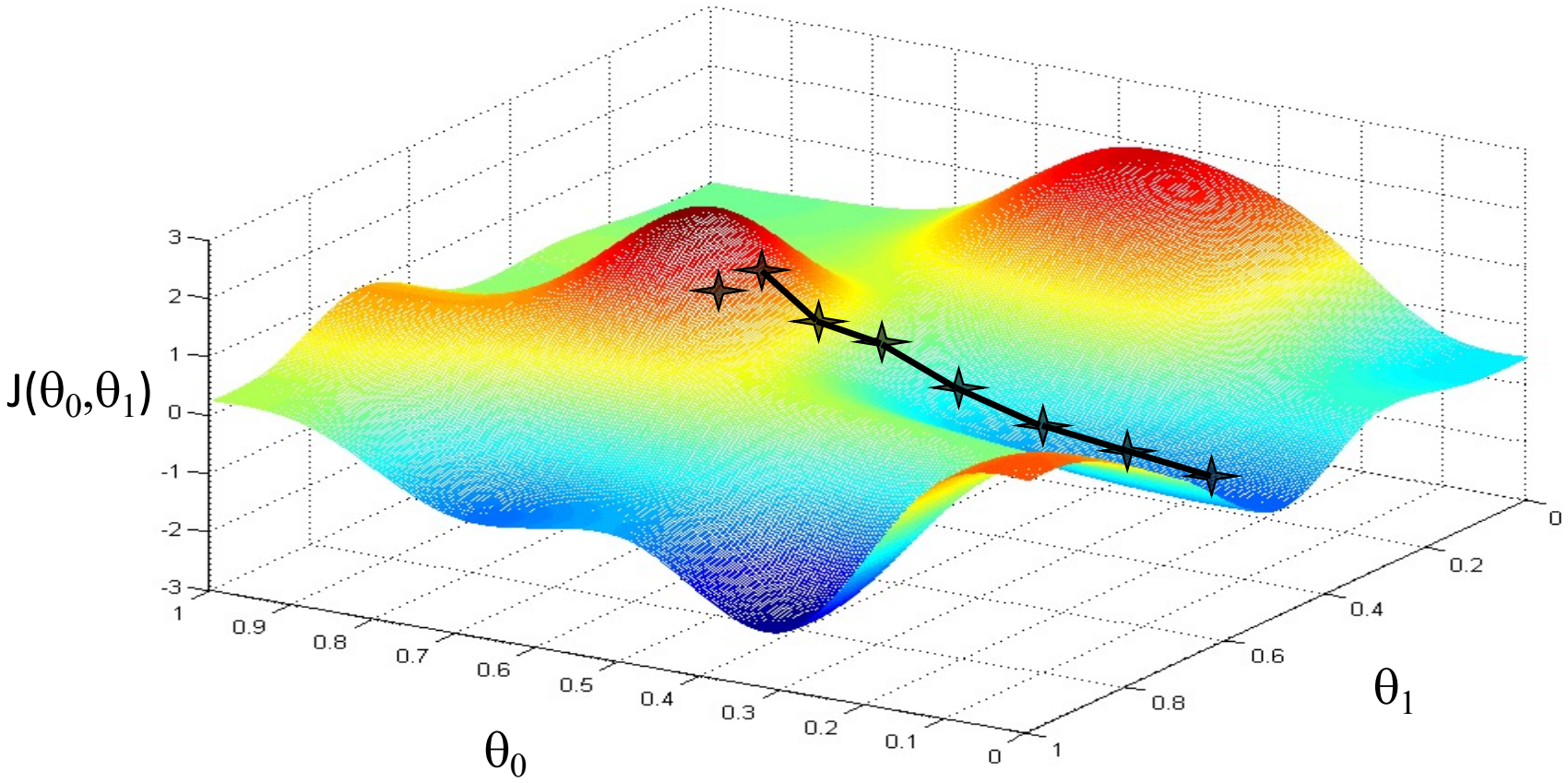
repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

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

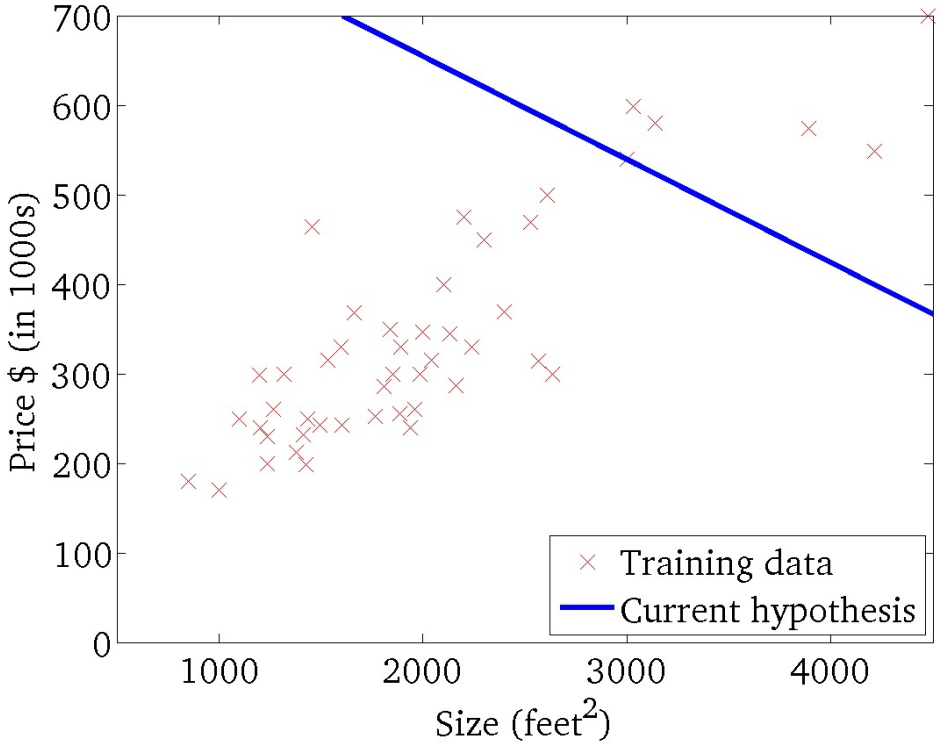
} update θ_0 and θ_1 simultaneously


J(θ0,θ1)
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ0
1
0.8
0.6
0.4
0.2
0
θ1


$J(\theta_0,\theta_1)$
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ_0
0
0.2
0.4
0.6
0.8
1
θ_1

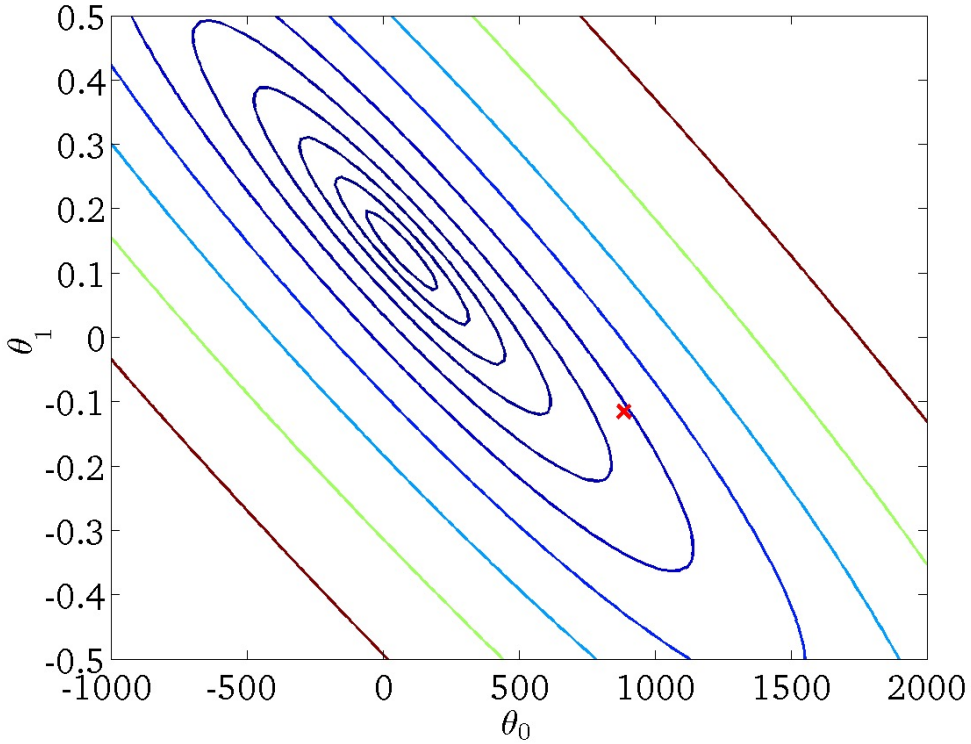
$h_\theta(x)$

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



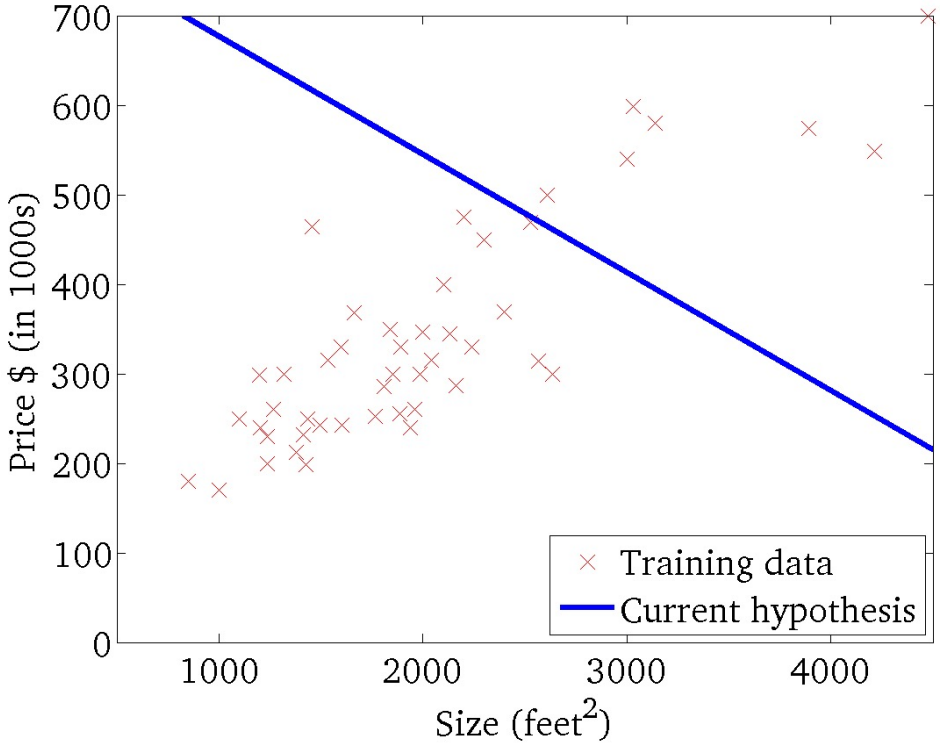
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



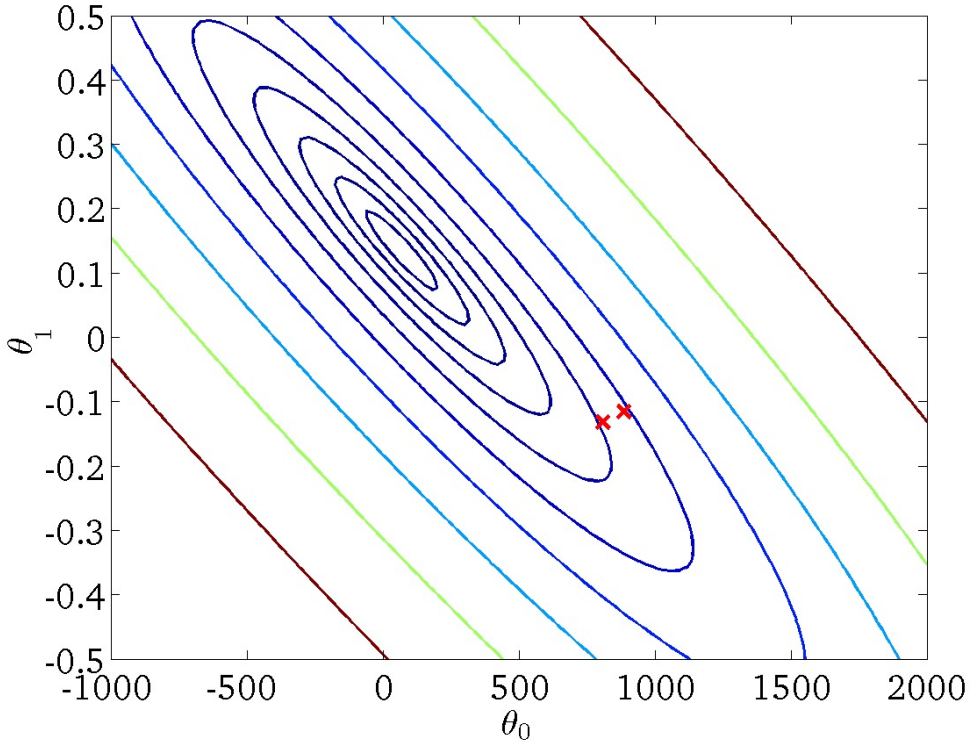
$h_\theta(x)$

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



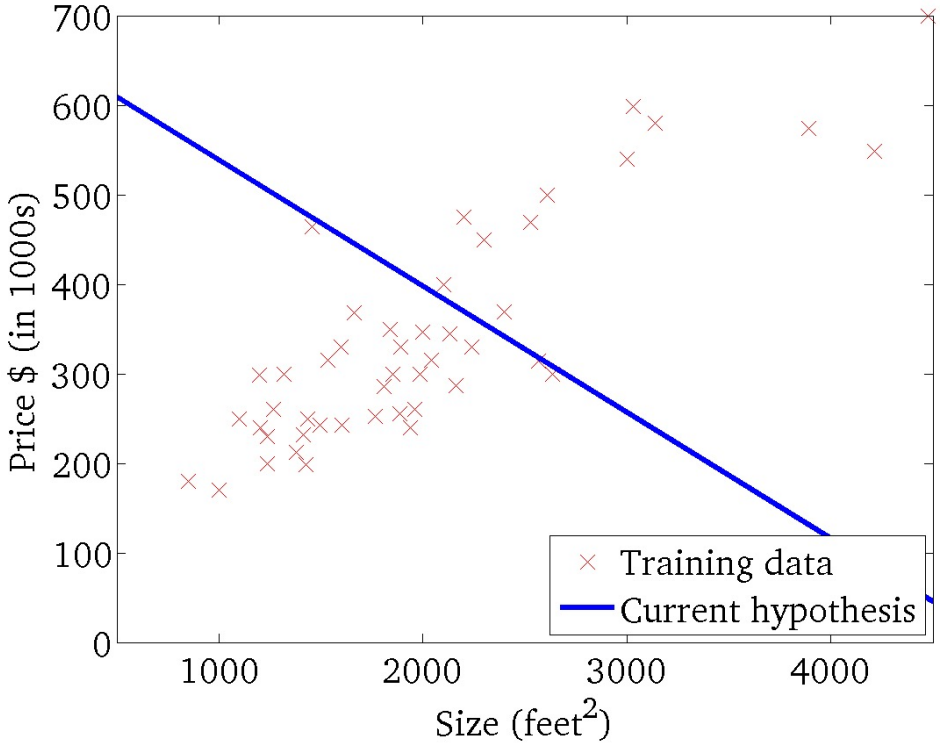
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



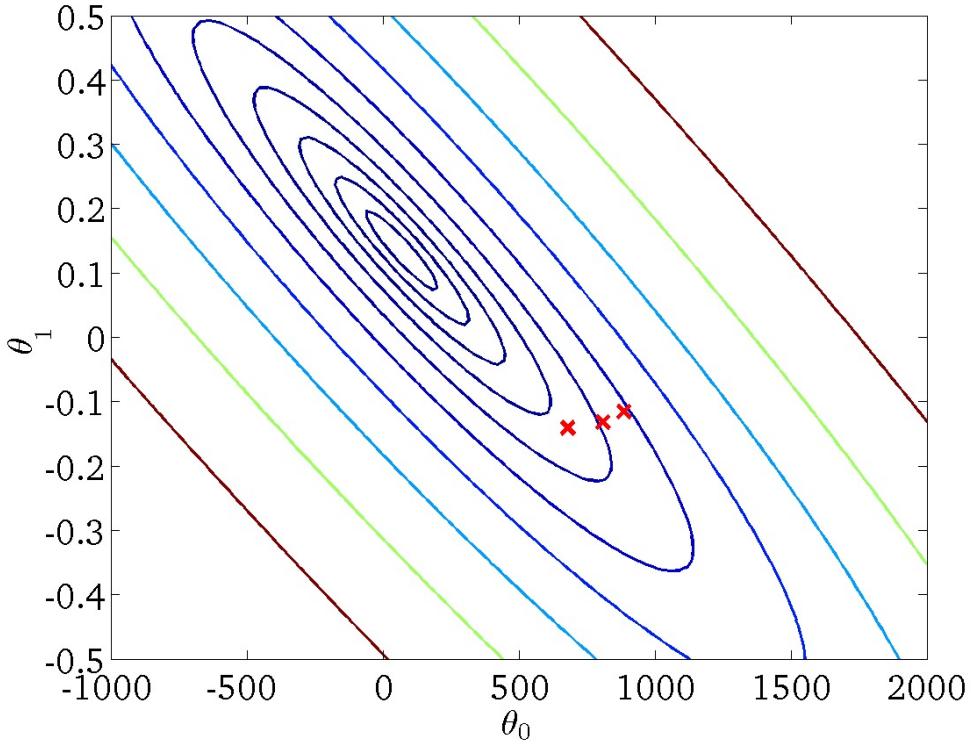
$h_\theta(x)$

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



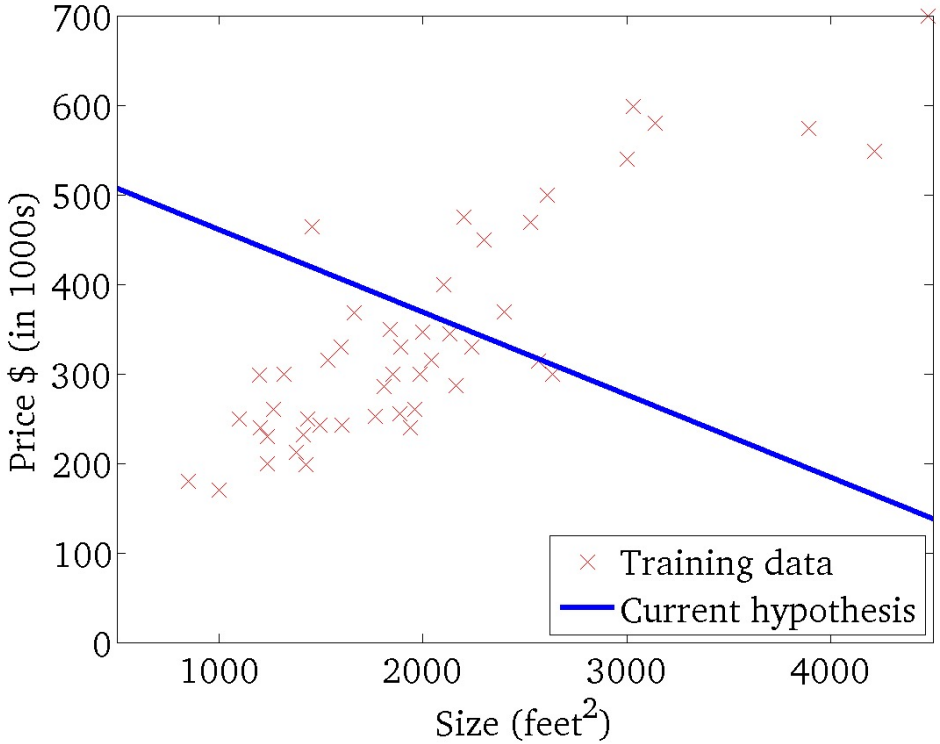
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



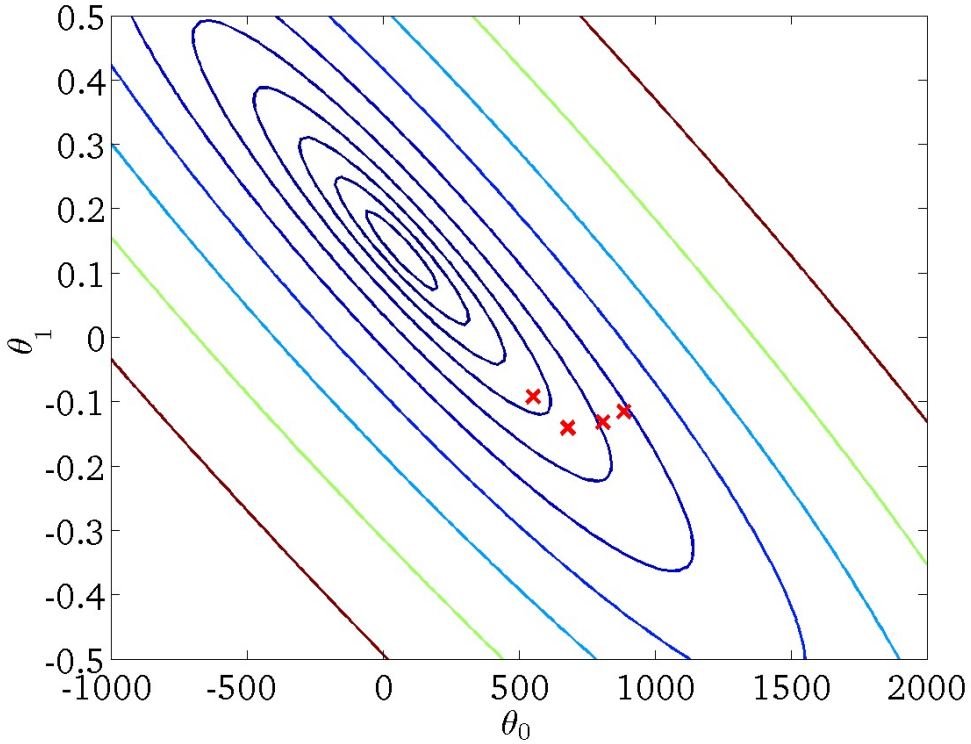
$h_\theta(x)$

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



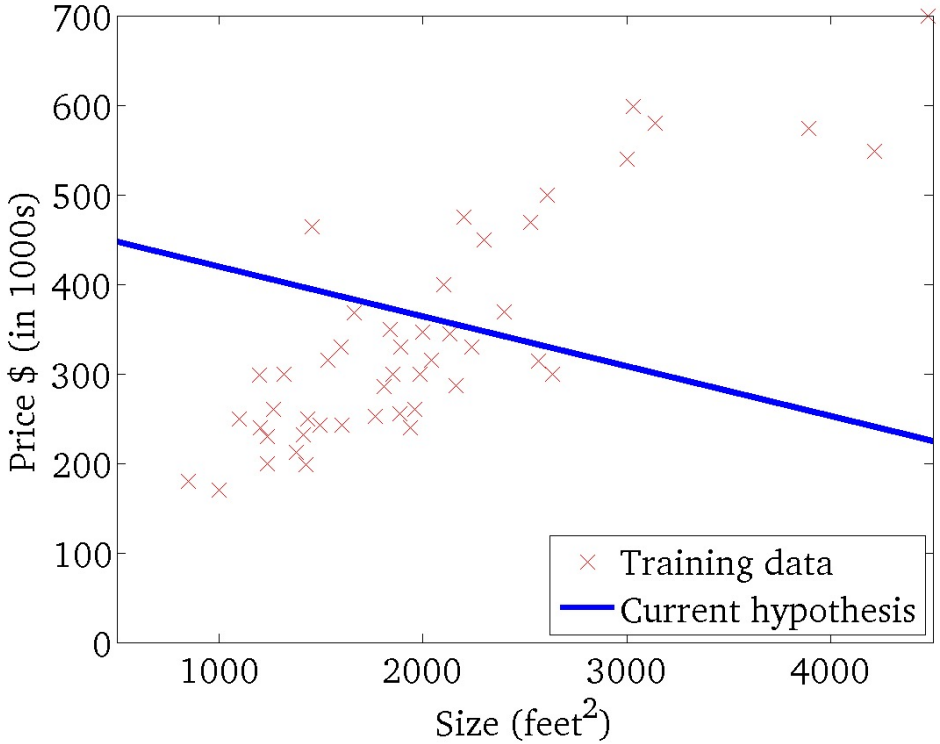
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



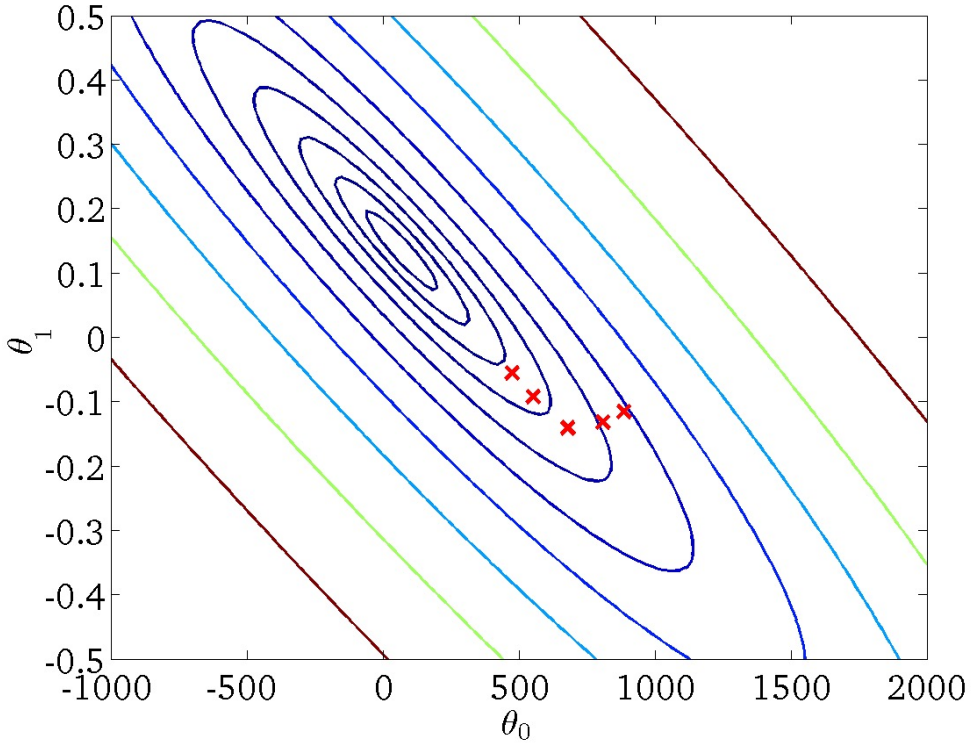
$h_\theta(x)$

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



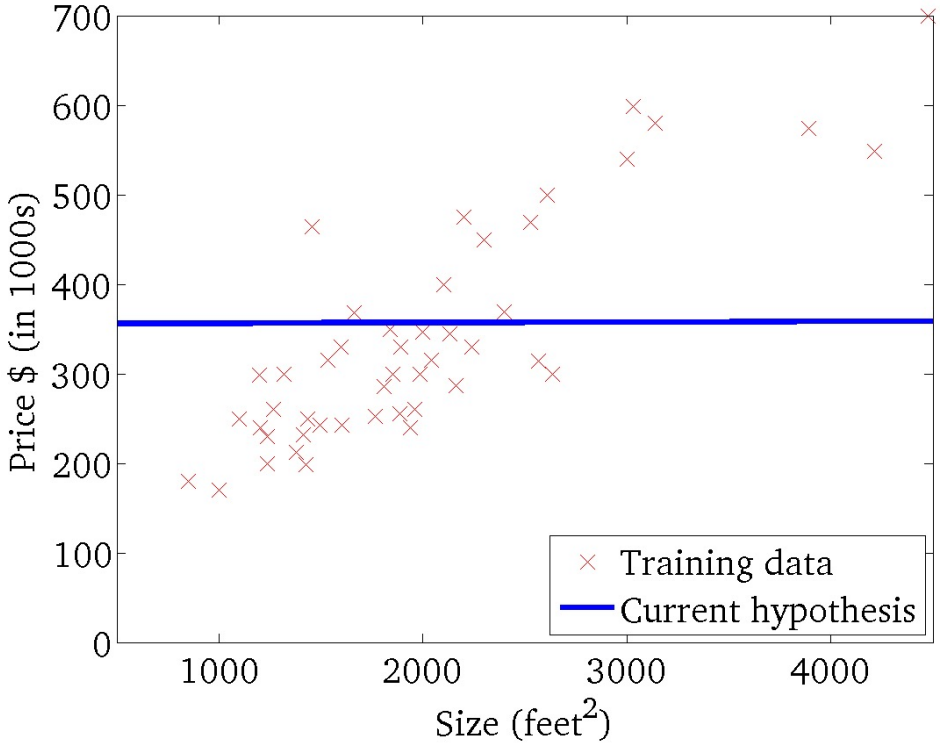
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



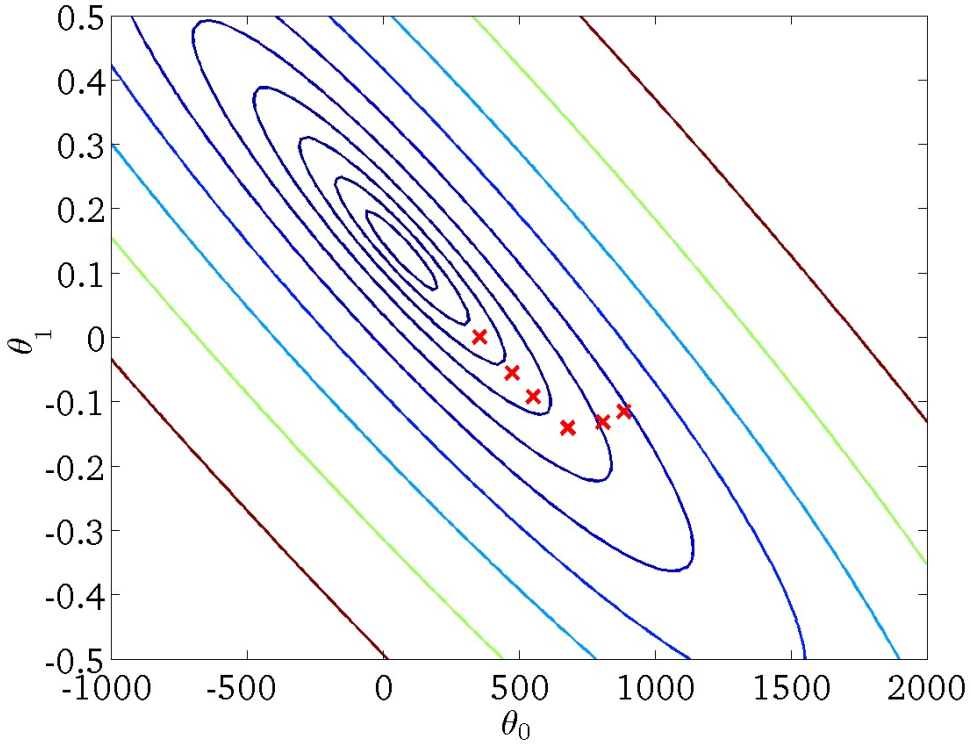
$h_\theta(x)$

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



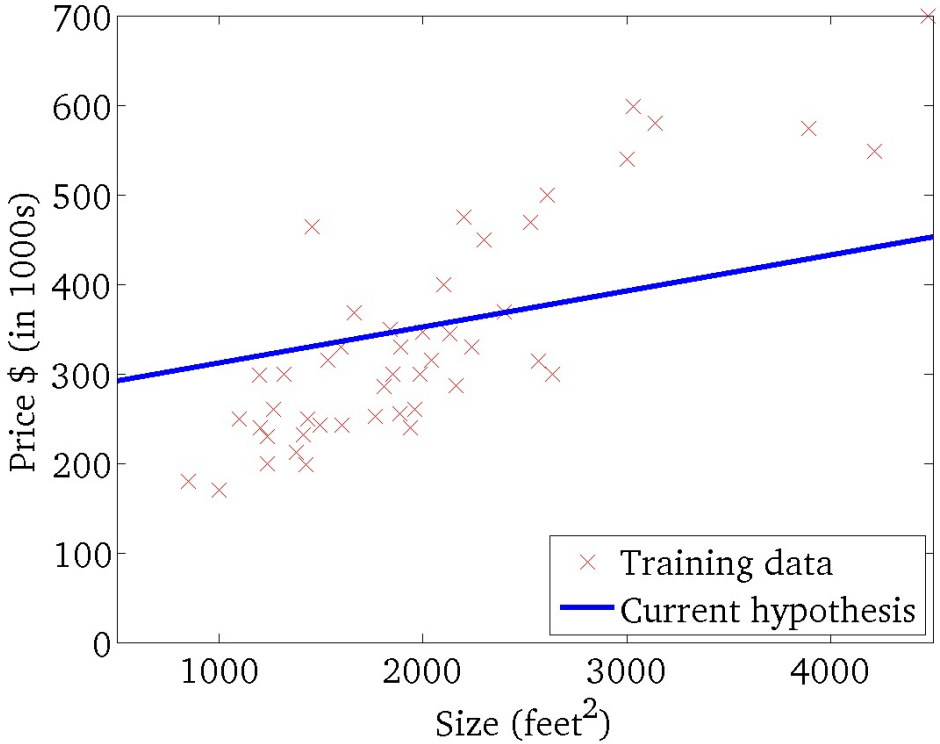
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



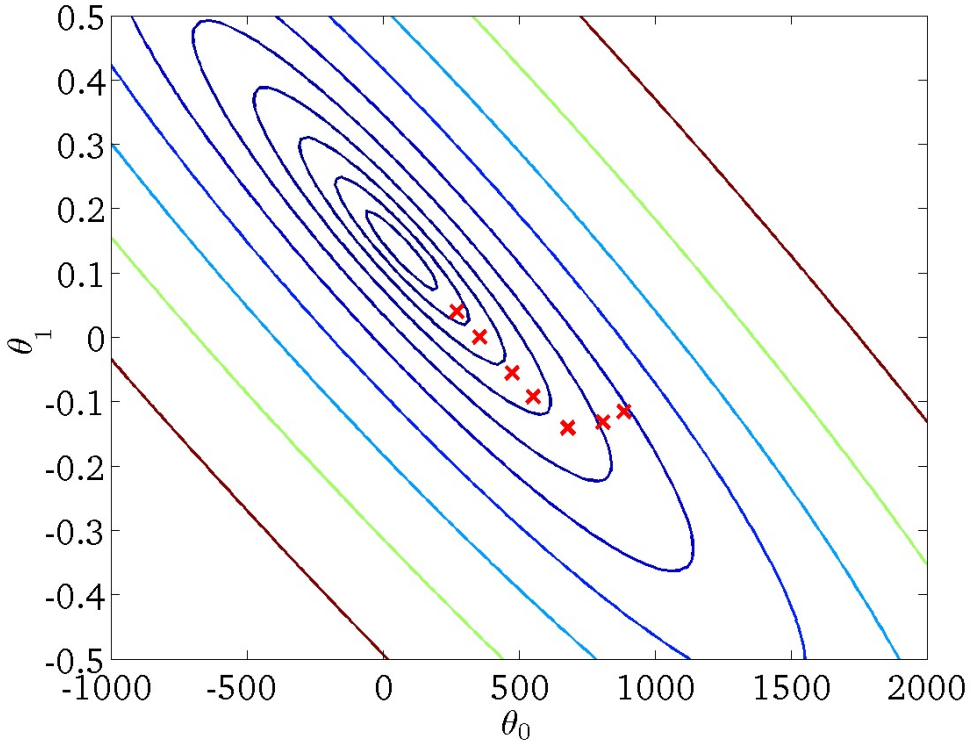
$h_\theta(x)$

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



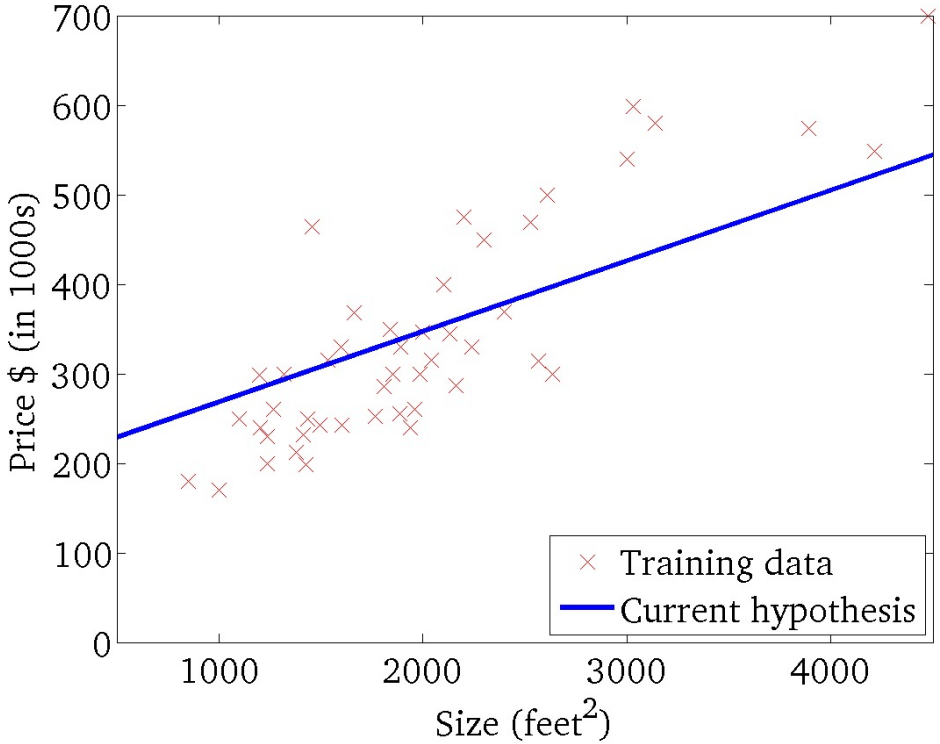
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



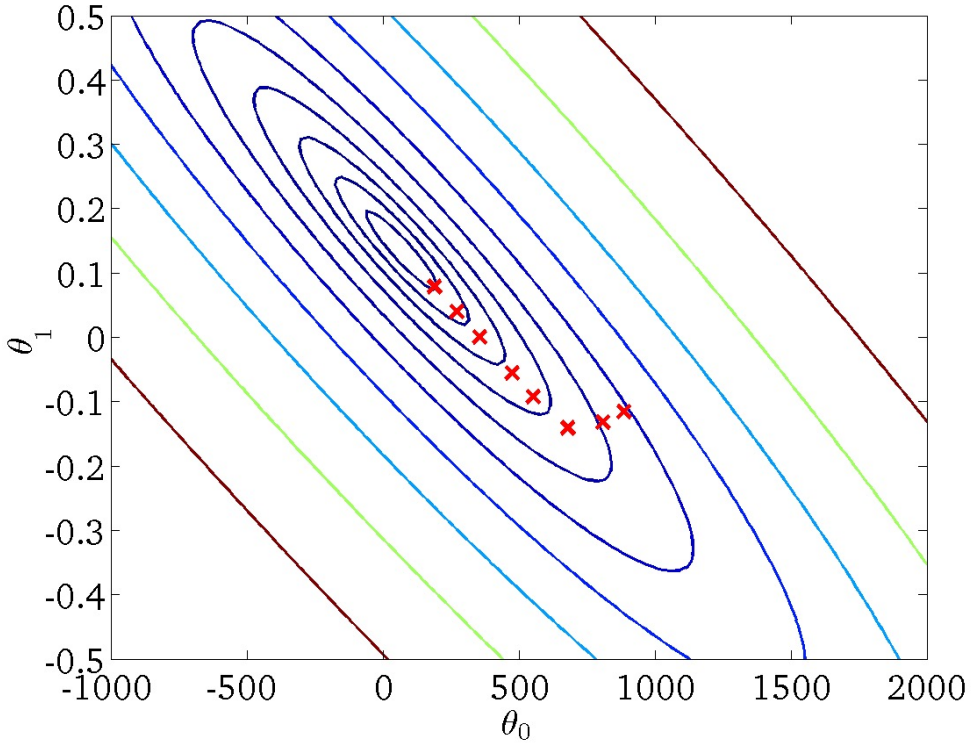
$h_\theta(x)$

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



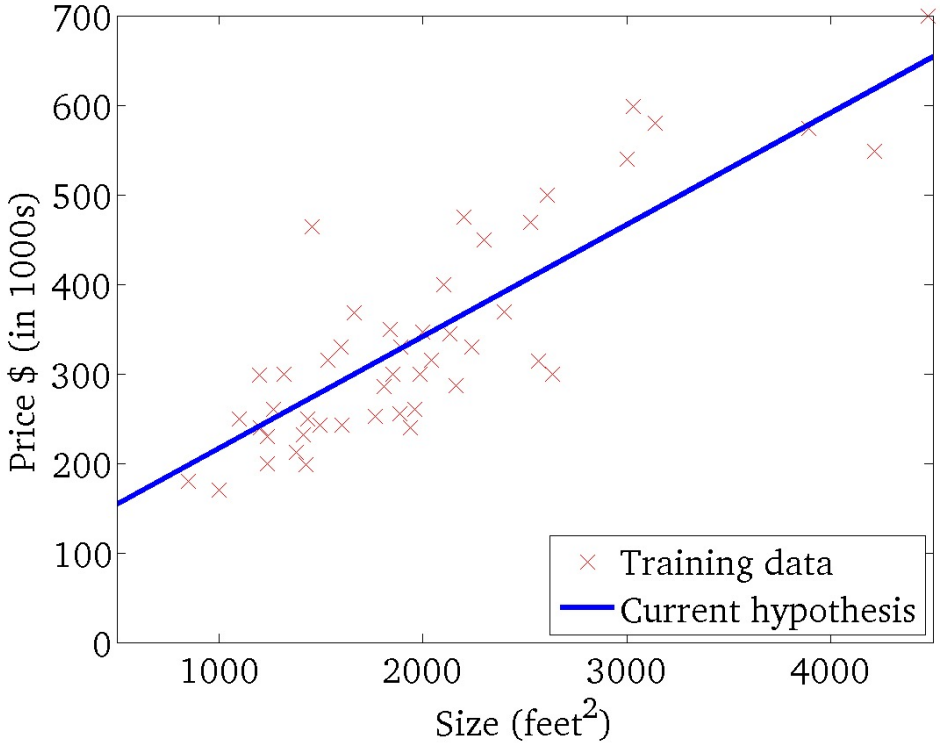
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



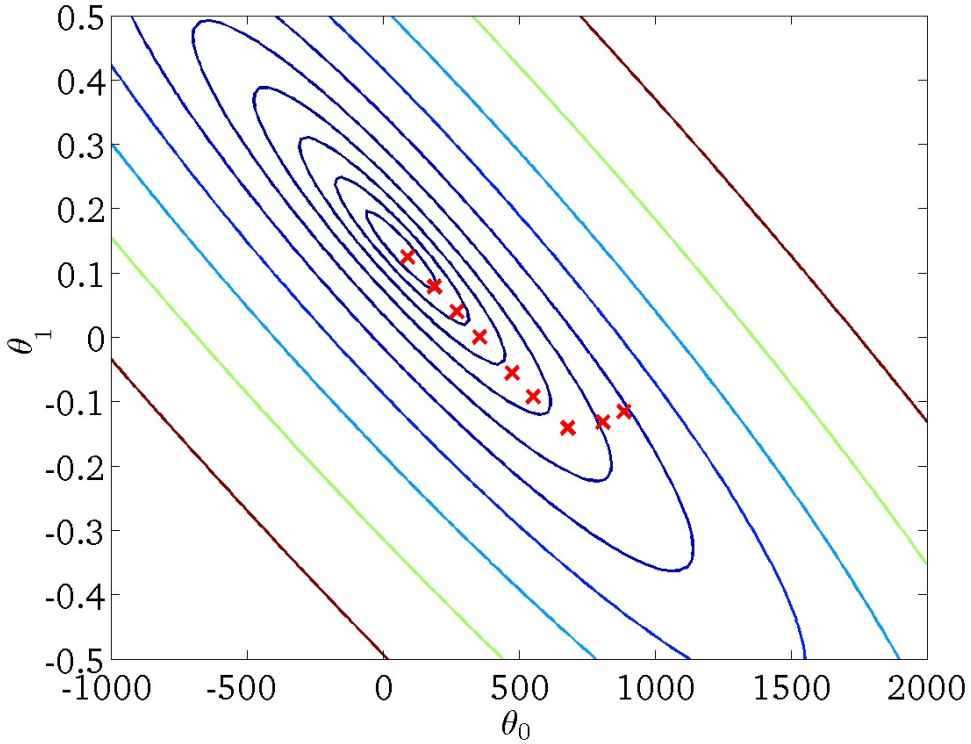
$h_\theta(x)$

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



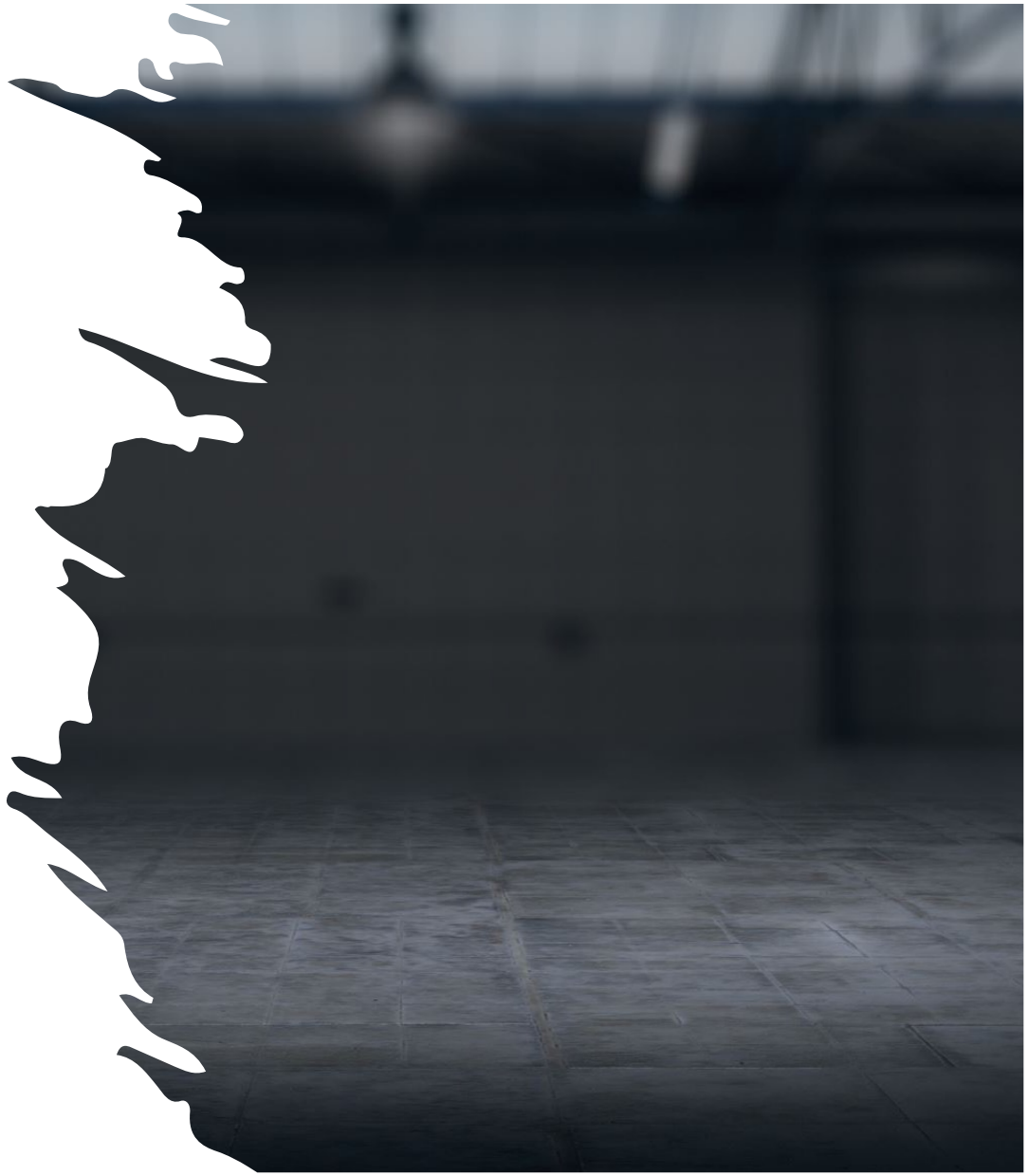
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



Batch Gradient Descent

- Each step of gradient descent uses all the training examples.



Partial derivatives of the cost function

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m_i} \sum_{i=1}^{m} (\theta^T \cdot x^{(i)} - y^{(i)}) x_j{}^{(i)}$$

Gradient vector of the cost function

$$\nabla_{\boldsymbol{\theta}} \operatorname{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \operatorname{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \operatorname{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \operatorname{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^{\intercal} (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Gradient Descent step

$$\theta^{(next\_step)} = \theta - \eta \nabla_{\theta} MSE(\theta)$$

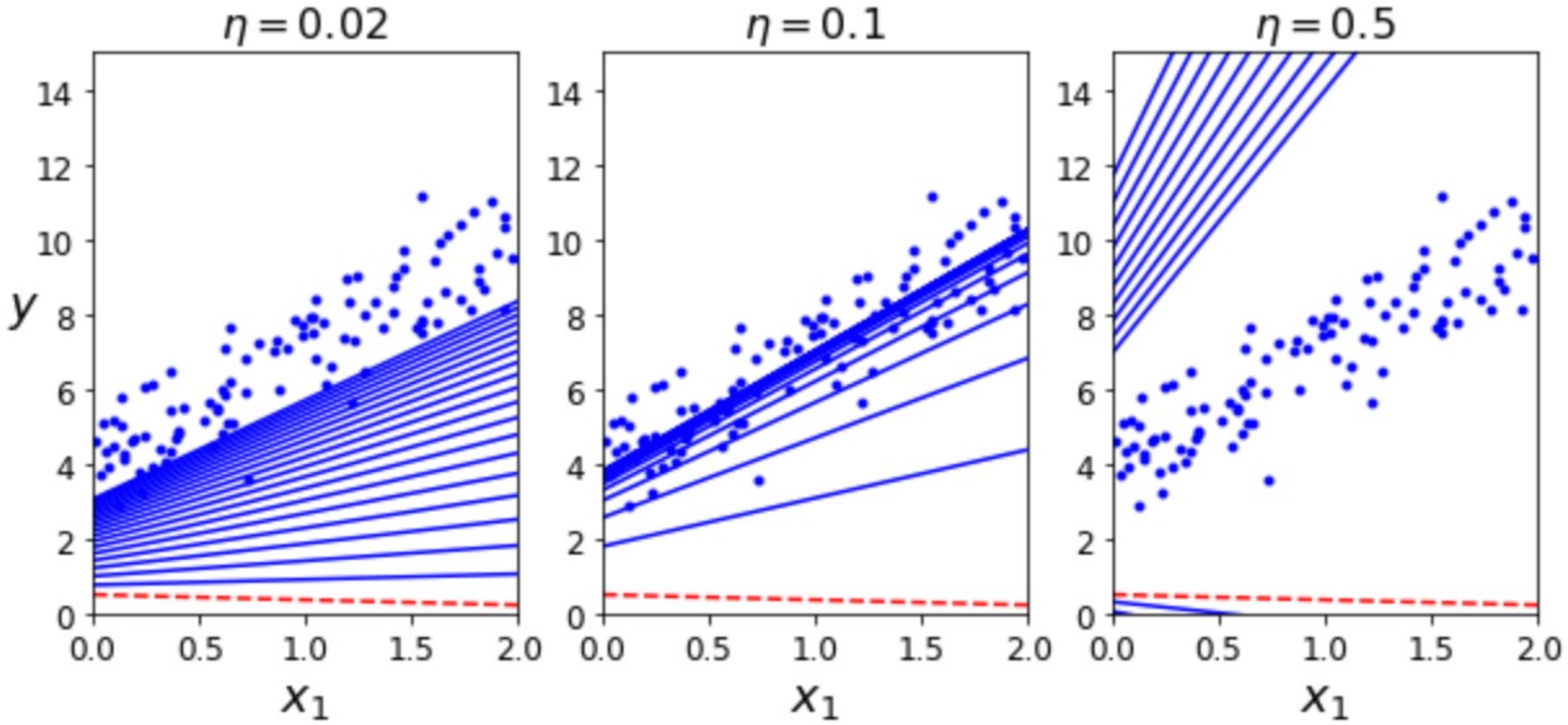
```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

```
theta
```

```
array([[4.21509616],
       [2.77011339]])
```

```
X_new_b.dot(theta)
```

```
array([[4.21509616],
       [9.75532293]])
```



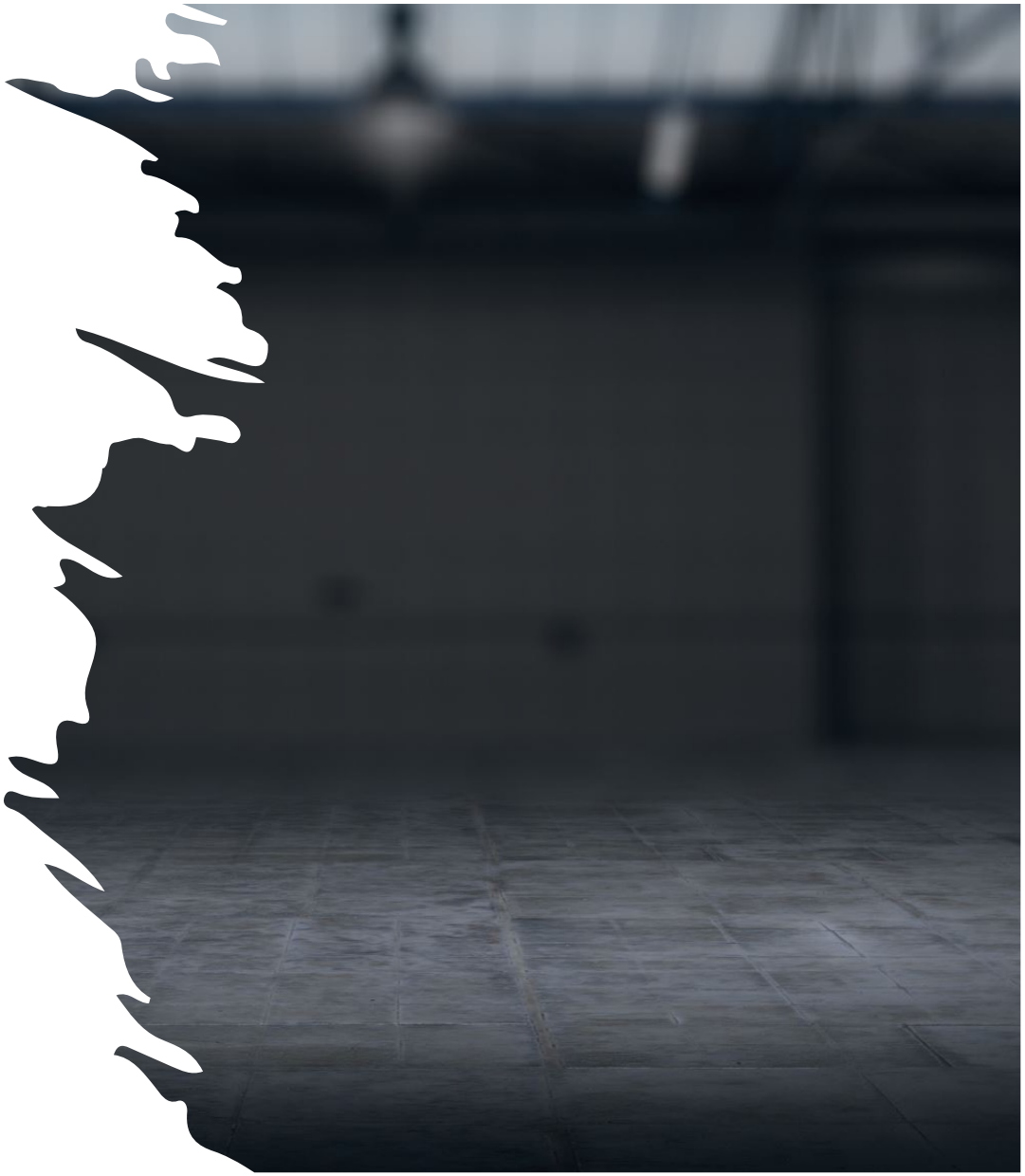
A simple solution is to set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny—that is, when its norm becomes smaller than a tiny number $\boldsymbol{\varepsilon}$ (called the ***tolerance***) —because this happens when Gradient Descent has (almost) reached the minimum.

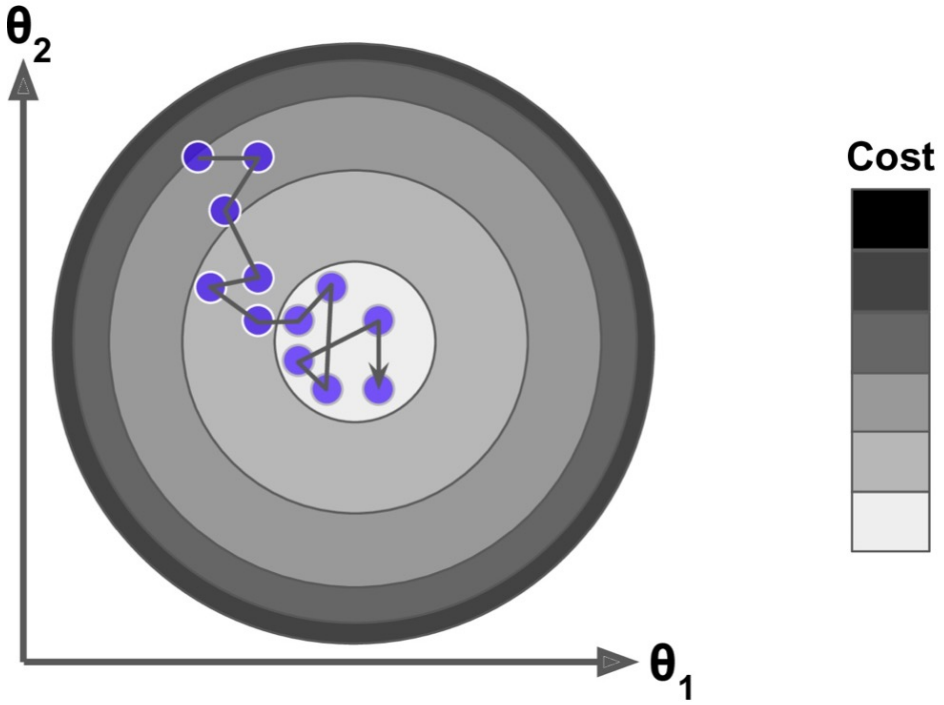
Batch Gradient Descent

- Each step of gradient descent uses all the training examples.

Stochastic Gradient Descent

- picks one random instance in the training set, much faster
- less regular, good but not optimal





When the cost function is very irregular , this can actually help the algorithm jump out of local minima.

Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

Randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum.

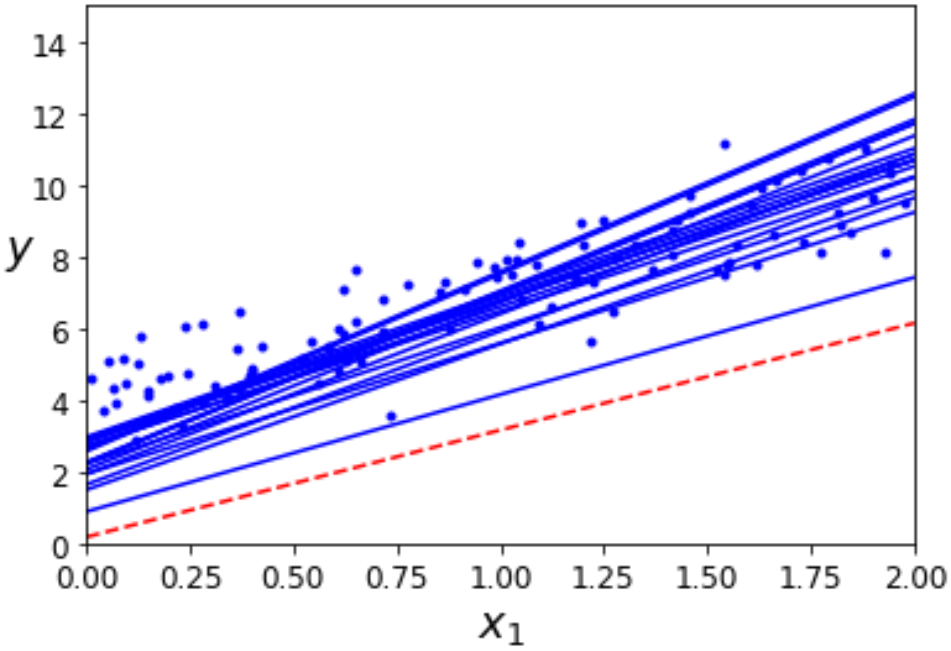
- One solution to this dilemma is to gradually reduce the learning rate.
  - The steps start out large (which helps make quick progress and escape local minima),
  - then get smaller and smaller, allowing the algorithm to settle at the global minimum.

This process is akin to ***simulated annealing***.

learning schedule:

- If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum.
- If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.

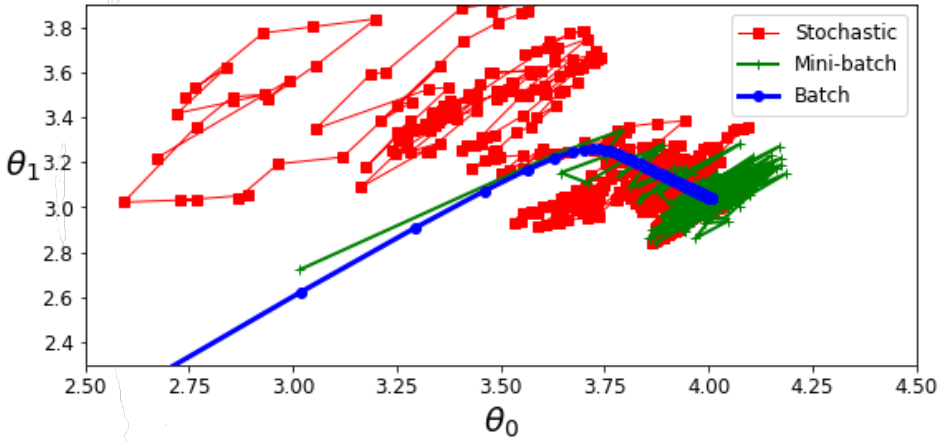
```
n_epochs = 100
t0, t1 = 5, 50  # learning schedule hyperparameters

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

theta = np.random.randn(2,1)  # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20:                    # not shown in the book
            y_predict = X_new_b.dot(theta)           # not shown
            style = "b-" if i > 0 else "r--"         # not shown
            plt.plot(X_new, y_predict, style)        # not shown
        random_index = np.random.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)                 # not shown
```



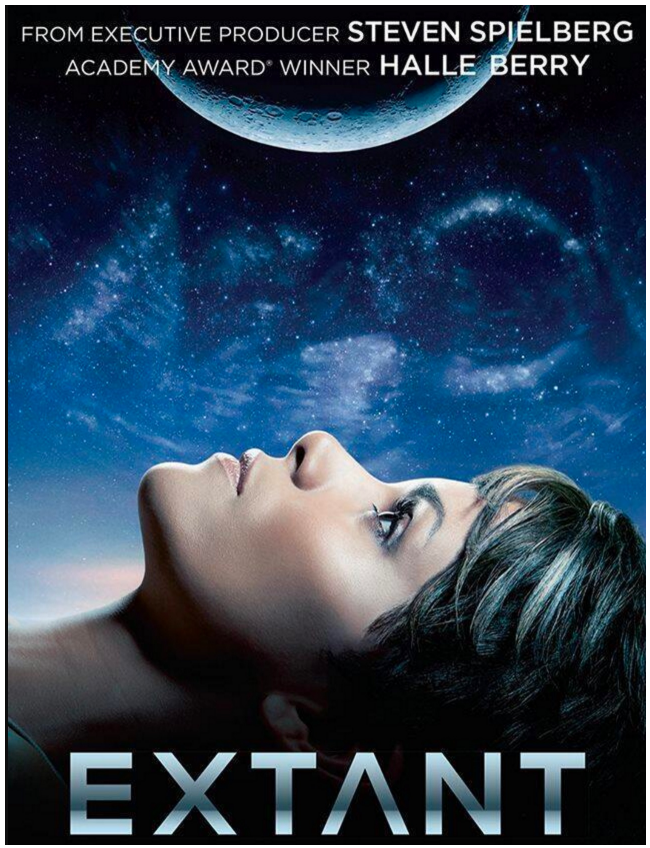

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3,
                       penalty=None, eta0=0.1, random_state=42)
sgd_reg.fit(X, y.ravel())
```

Mini - Batch Gradient Descent

- Small random sets
- Performance boost from hardware optimization of matrix operations, GPU
- Less regular, good but not optimal
- Hard to escape from local minima



Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`



Machine Learning

Linear Regression with multiple variables

Multiple features

Multiple features (variables).

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

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

Multiple features (variables).

Size (feet^2)	Number of bedrooms	Number of floors	Age of home (years)	Price ($1000)
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
…	…	…	…	…

Notation:

n = number of features

$x^{(i)}$ = input (features) of i^{th} training example.

$x_j^{(i)}$ = value of feature j in i^{th} training example.

Hypothesis:

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

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$$

e.g. $h_\theta(x) = 80 + 0.1 x_1 + 0.01 x_2 + 3 x_3 - 2 x_4$

x_2: ↑

x_3: ↑

x_4: ↑ age

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

For convenience of notation, define $x_0 = 1$.

Multivariate linear regression.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

For convenience of notation, define $x_0 = 1$. $(x_0^{(i)} = 1)$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n \quad (x_0 = 1)$$

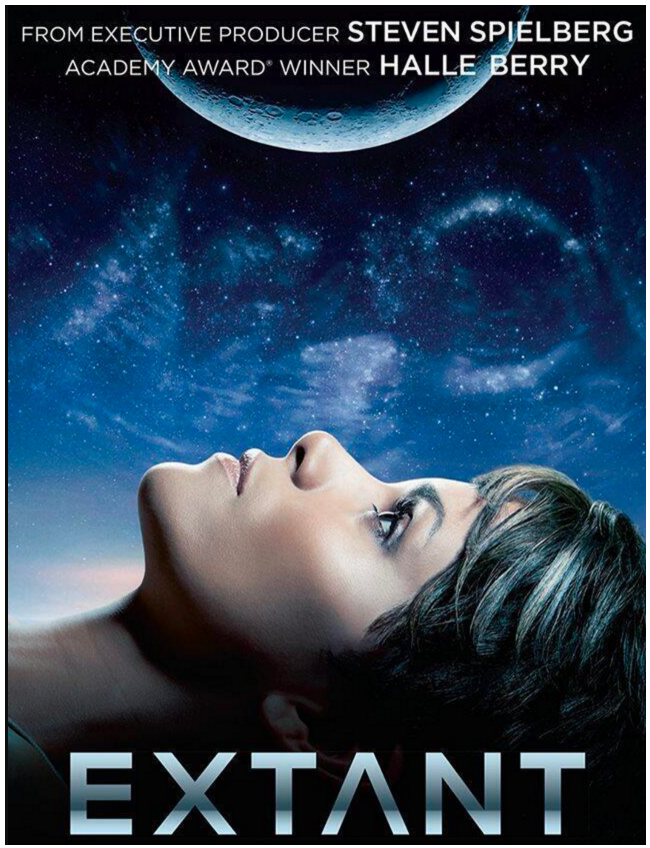
$$= \theta^T x$$

θ^T: $(n+1) \times 1$ matrix

$$[\theta_0 \; \theta_1 \cdots \theta_n] \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$\theta^T x$

Multivariate linear regression.



Machine Learning

Linear Regression with multiple variables

Gradient descent for multiple variables

Hypothesis: $h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$

Parameters: $\theta_0, \theta_1, \ldots, \theta_n$

Cost function:

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

Gradient descent:

Repeat $\{$

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

$\}$ (simultaneously update for every $j = 0, \ldots, n$)

Gradient Descent

Previously (n=1):

Repeat $\{$

$$\theta_0 := \theta_0 - \alpha \underbrace{\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})}_{\frac{\partial}{\partial\theta_0}J(\theta)}$$

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

(simultaneously update θ_0, θ_1)

$\}$

New algorithm $(n \geq 1)$:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

(simultaneously update θ_j for $j = 0, \ldots, n$)

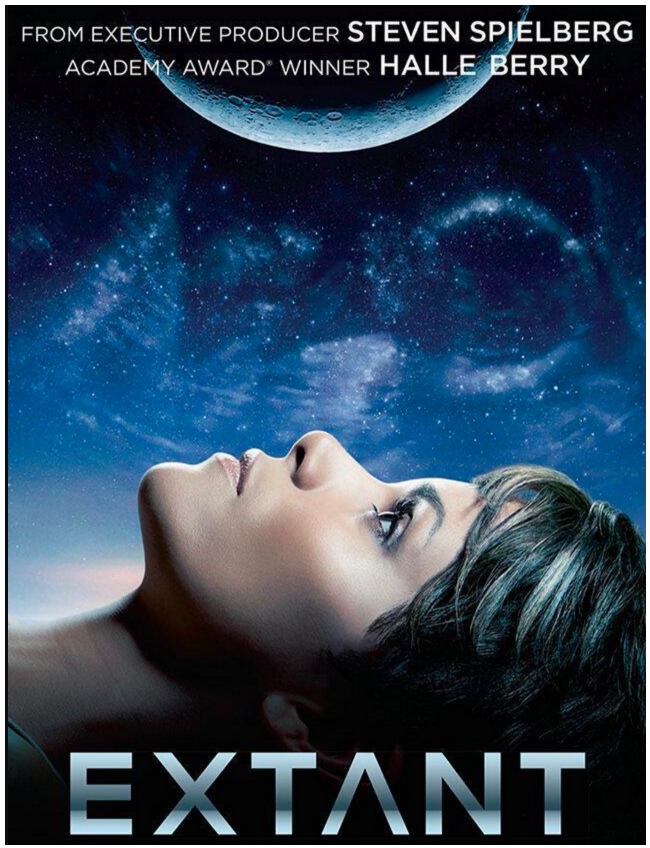
$\}$

$$\theta_0 := \theta_0 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_2^{(i)}$$

$\ldots$



Machine Learning

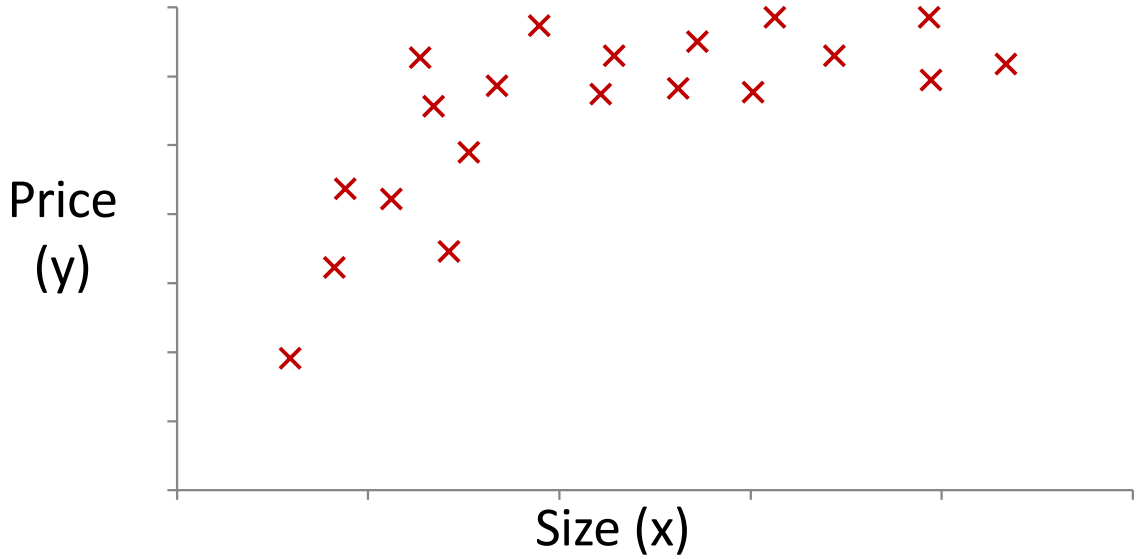
Linear Regression with multiple variables

Features and polynomial regression

Housing prices prediction

$$h_\theta(x) = \theta_0 + \theta_1 \times frontage + \theta_2 \times depth$$

Polynomial regression



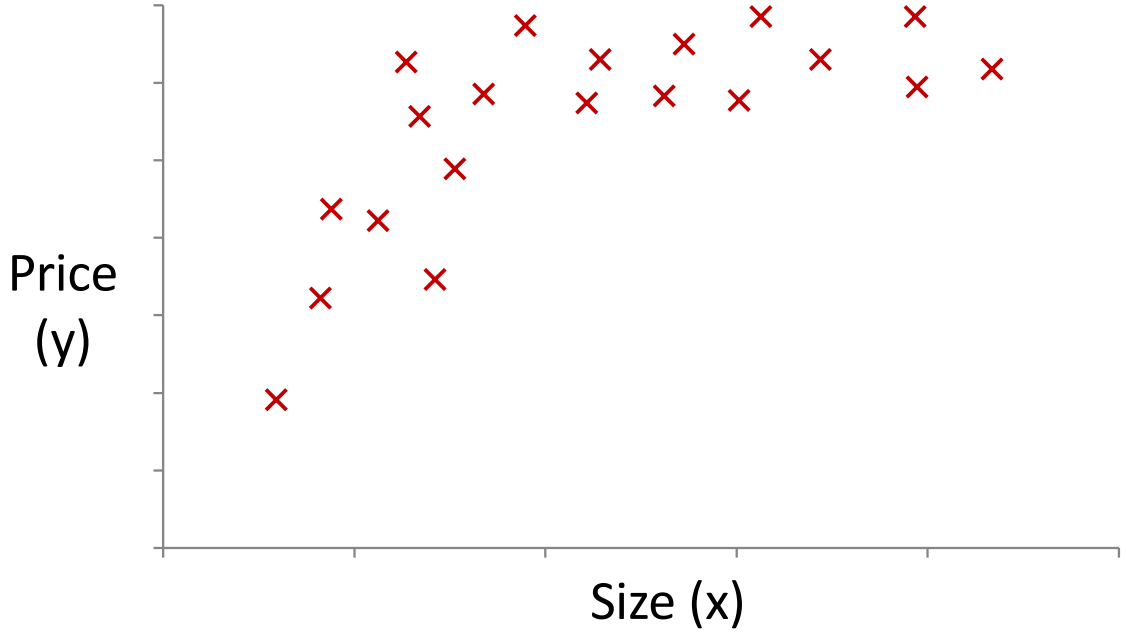
$$\theta_0 + \theta_1 x + \theta_2 x^2$$

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

$$\begin{aligned} h_\theta(x) &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 \\ &= \theta_0 + \theta_1 (size) + \theta_2 (size)^2 + \theta_3 (size)^3 \end{aligned}$$

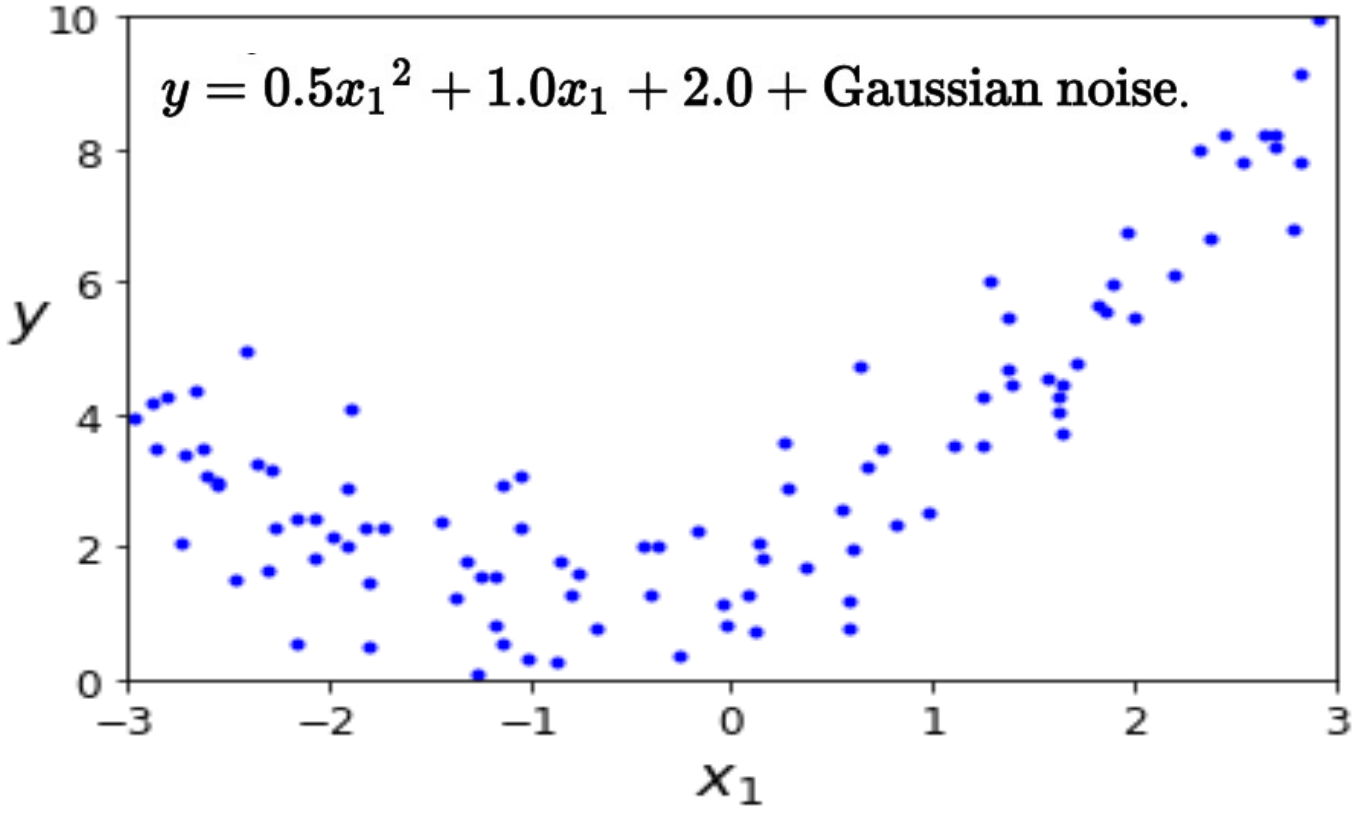
$$\begin{aligned} x_1 &= (size) \\ x_2 &= (size)^2 \\ x_3 &= (size)^3 \end{aligned}$$

Choice of features



$$h_\theta(x) = \theta_0 + \theta_1(size) + \theta_2(size)^2$$

$$h_\theta(x) = \theta_0 + \theta_1(size) + \theta_2\sqrt{(size)}$$

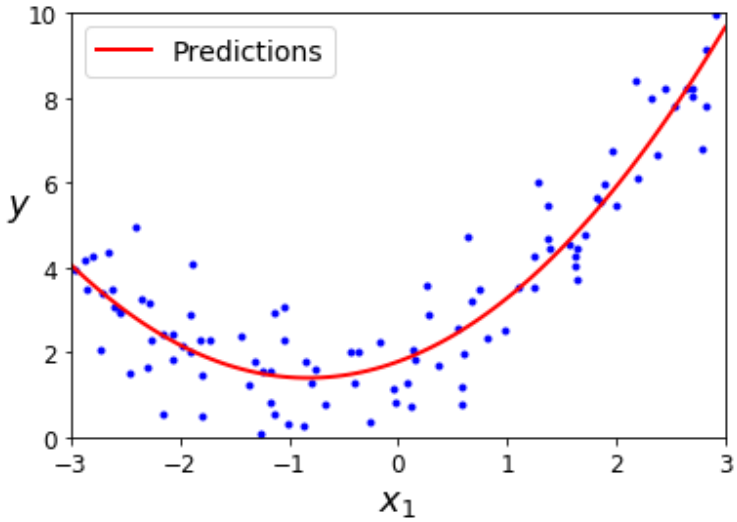


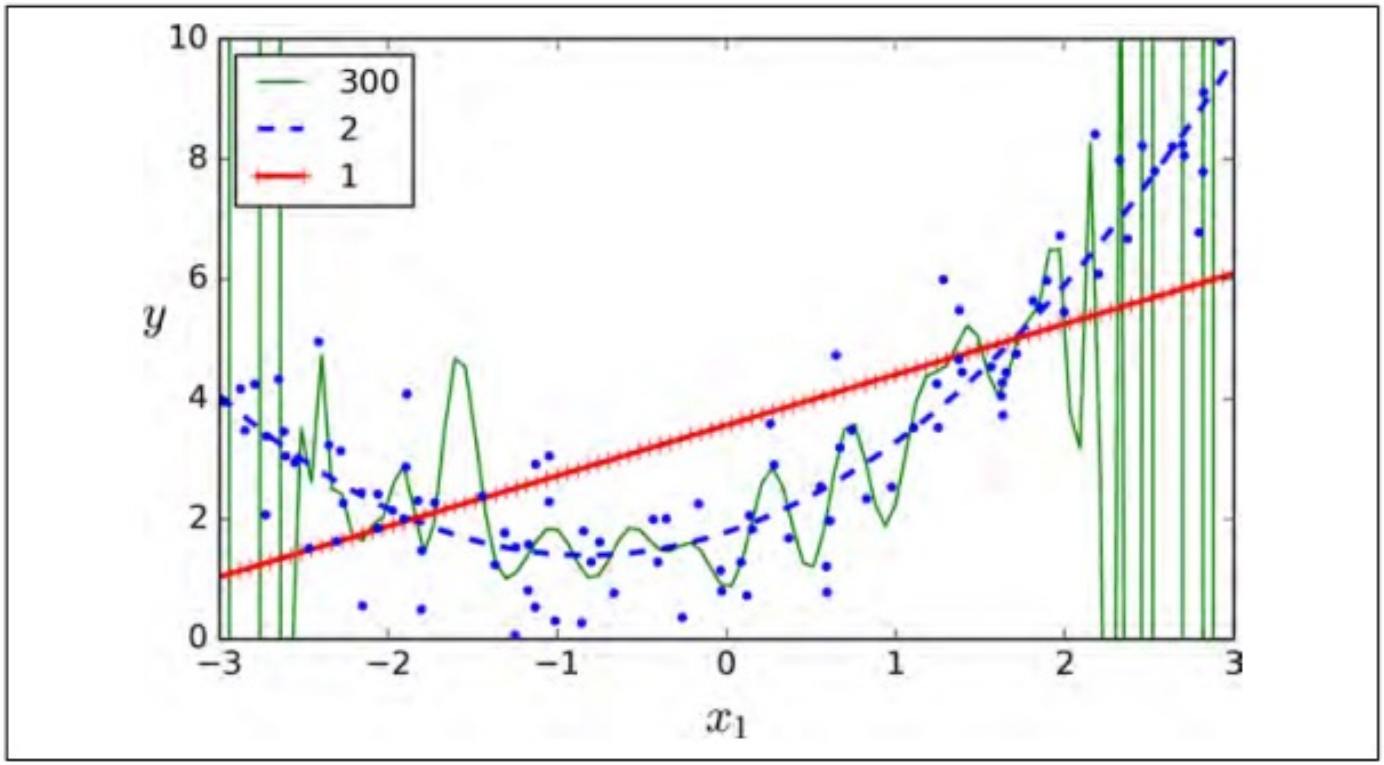

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
```

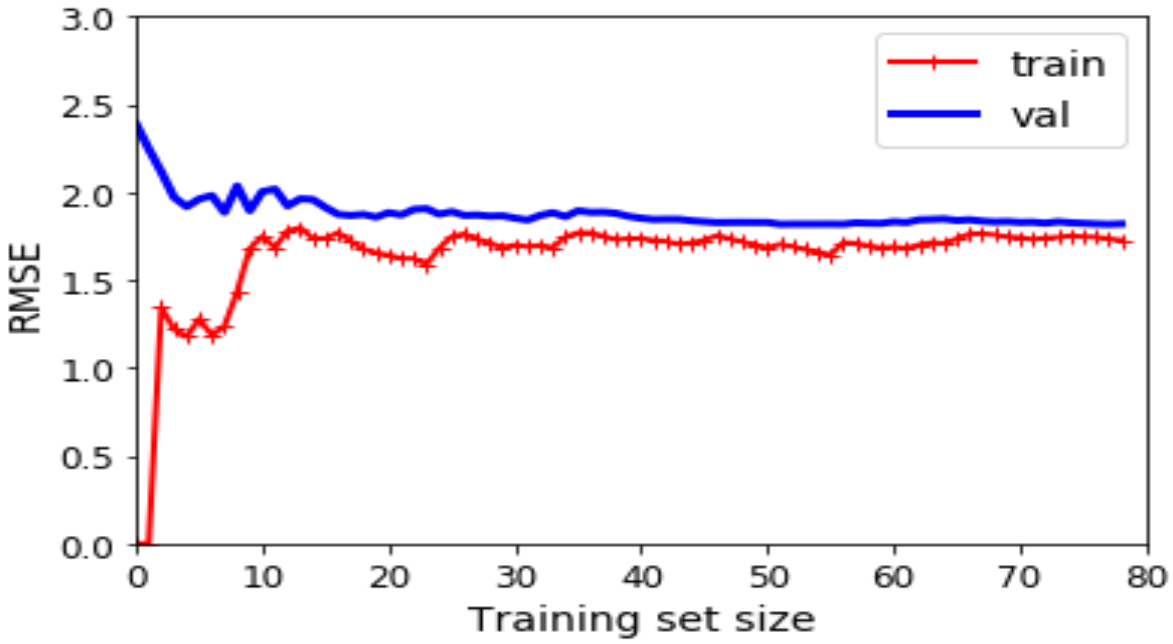
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
X_new=np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)

plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")

plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])
plt.show()
```





linear model : underfitting 300-degree model : overfitting

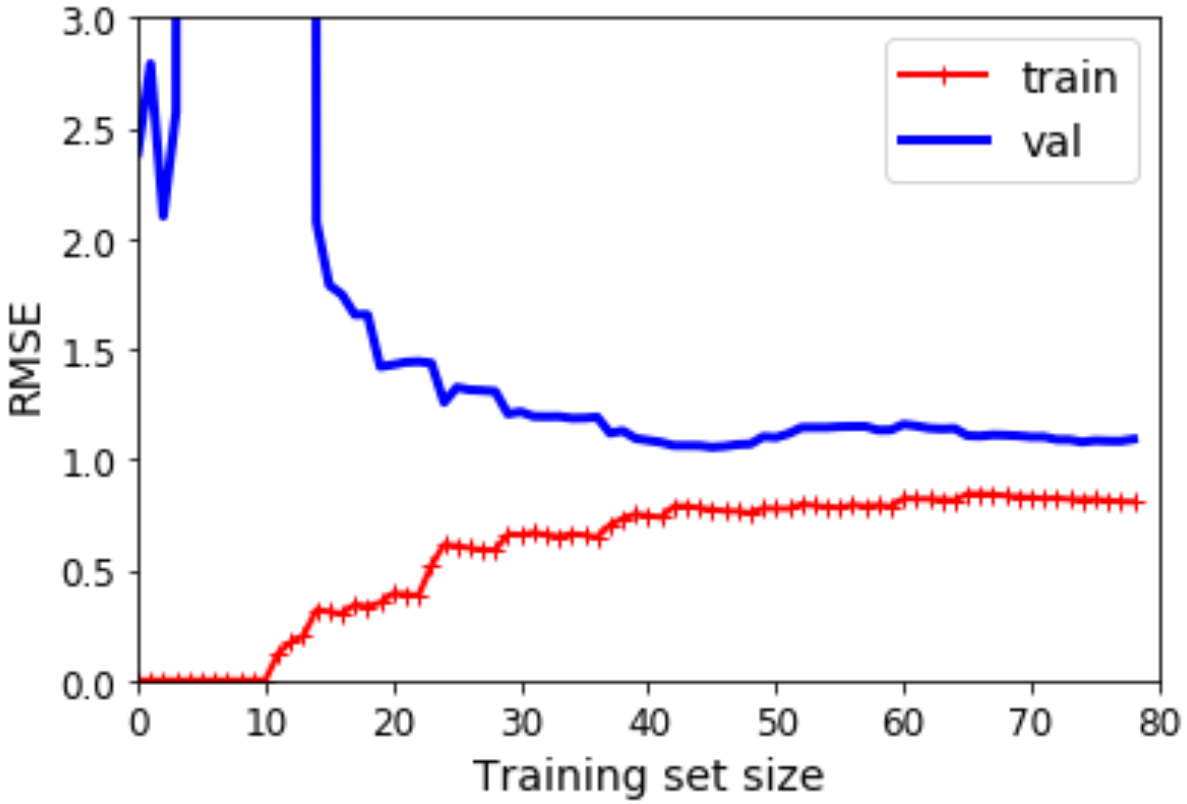



```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

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[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
    plt.legend(loc="upper right", fontsize=14)   # not shown in the book
    plt.xlabel("Training set size", fontsize=14) # not shown
    plt.ylabel("RMSE", fontsize=14)              # not shown
```

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3])                          # not shown in the book
plt.show()                                       # not shown
```

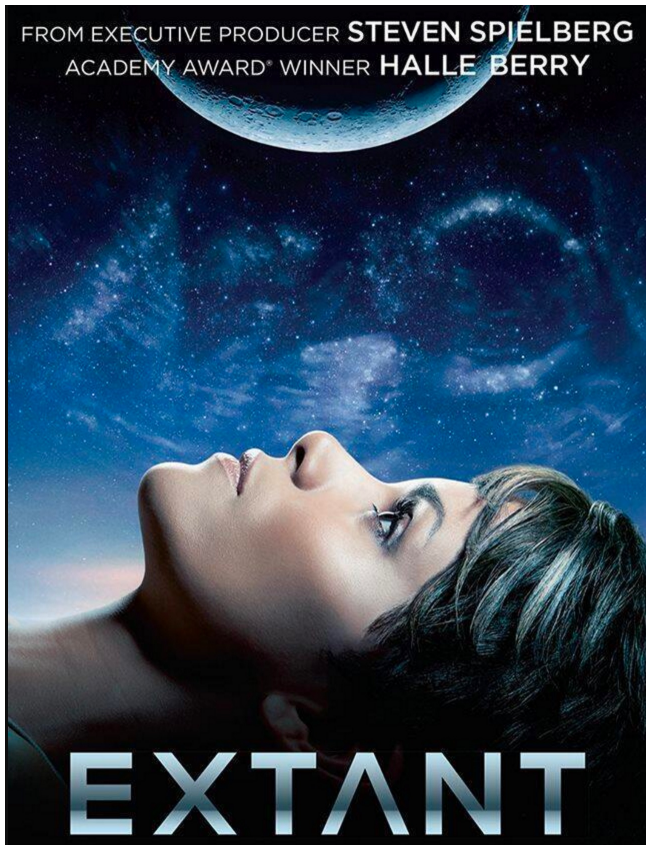
These learning curves are typical of a model that's underfitting. Both curves have reached a plateau; they are close and fairly high.

If your model is underfitting the training data, adding more training examples will not help. You need to use a more complex model or come up with better features.



There is a gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an overfitting model.

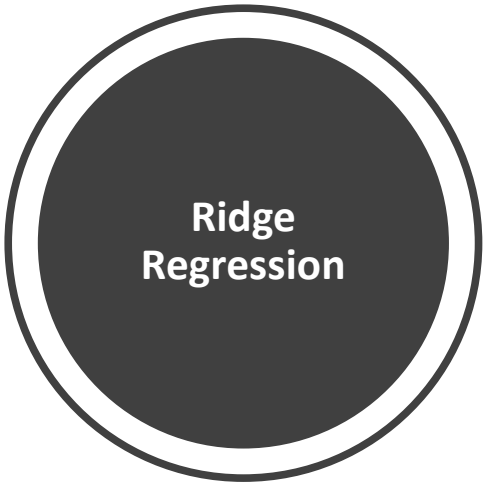
One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.



Machine Learning

Linear Regression with multiple variables

Regularized Linear models



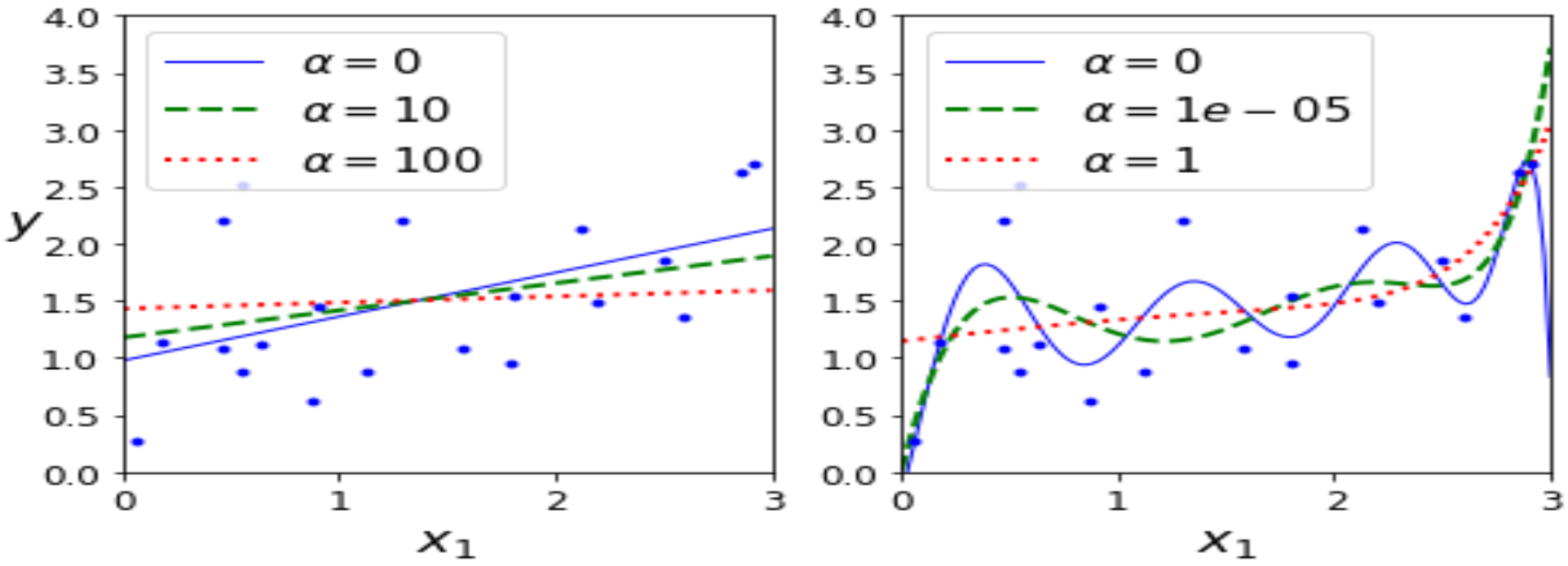
- ***Ridge Regression*** (also called ***Tikhonov regularization***) is a regularized version of Linear Regression

Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to use the unregularized performance measure to evaluate the model's performance.

Ridge Regression

cost function $J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$

regularization term $\frac{1}{2}\alpha \left(\|\vec{w}\|_2 \right)^2$ L2 norm of the weight vector



Ridge Regression

closed-form solution $\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

```
array([[1.55071465]])
```

```
ridge_reg = Ridge(alpha=1, solver="sag", random_state=42)
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

Ridge Regression

Stochastic Gradient Descent

```
from sklearn.linear_model import Ridge

sgd_reg = SGDRegressor(penalty="l2", max_iter=1000, tol=1e-3, random_state=42)
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

- The penalty hyperparameter sets the type of regularization term to use.
- Specifying "l2" indicates that you want SGD to add a regularization term to the cost function equal to half the square of the l_2 norm of the weight vector

Lasso Regression

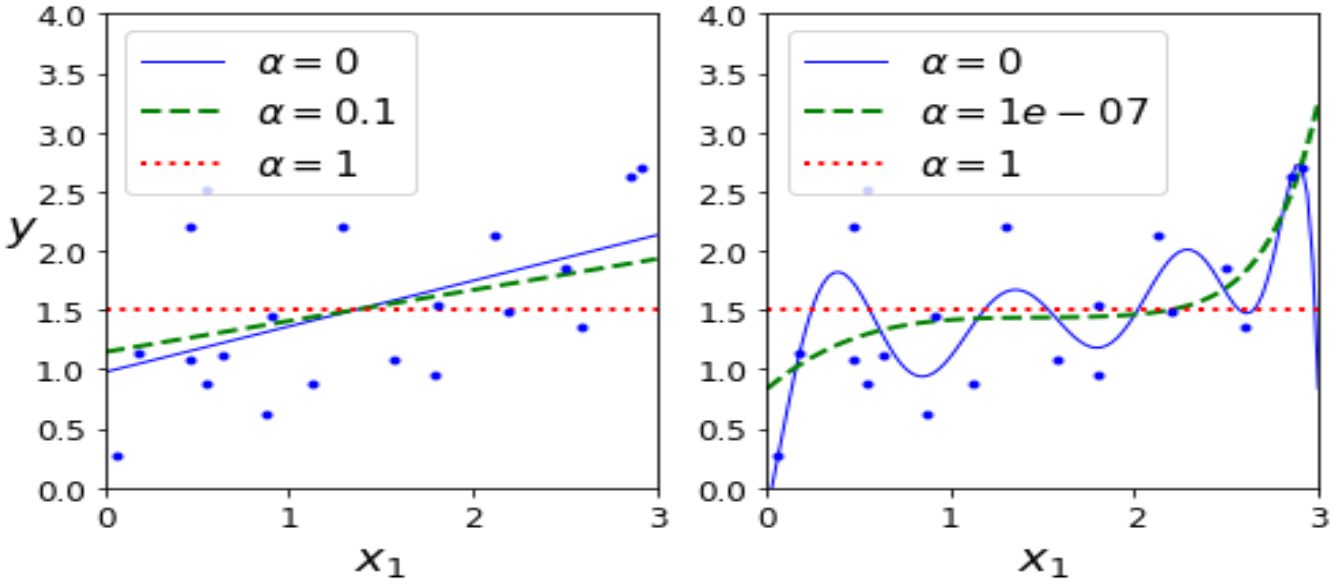
Lasso : Least Absolute Shrinkage and Selection Operator Regression

An important characteristic of Lasso Regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero).

Lasso Regression

cost function $J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$

regularization term $\alpha\left(\|\vec{w}\|_1\right)$ L2 norm of the weight vector



The dashed line (with $\alpha = 10^{-7}$) looks quadratic, almost linear: all the weights for the high-degree polynomial features are equal to zero.

Elastic Net

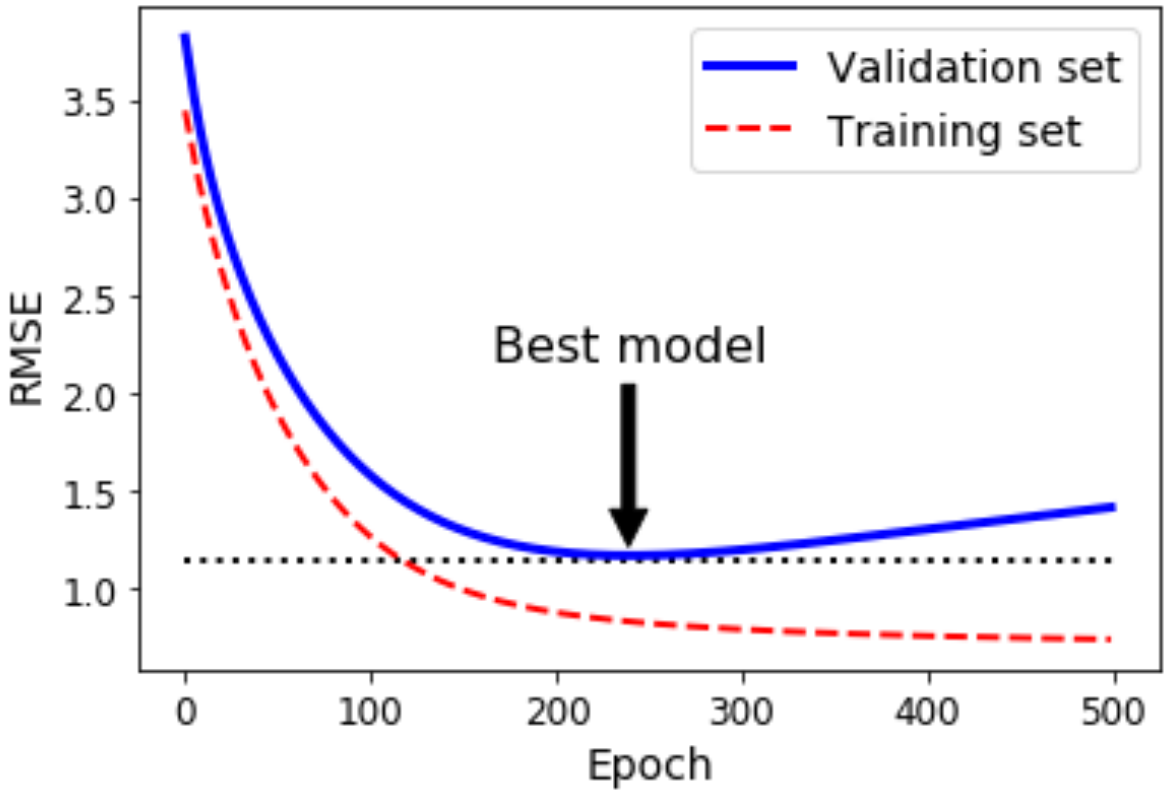
cost function $J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$

regularization term $r\alpha\|\vec{w}\|_1 + \frac{1-r}{2}\alpha\|\vec{w}\|_2^2$

```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
elastic_net.fit(X, y)
elastic_net.predict([[1.5]])
```

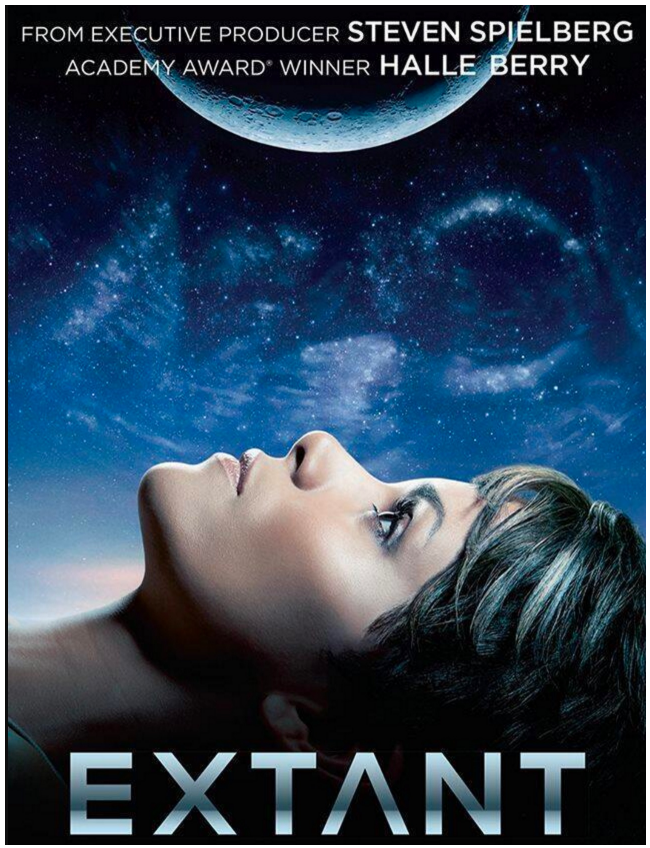
α r

E



With early stopping you just stop training as soon as the validation error reaches the minimum.

It is such a simple and efficient regularization technique that Geoffrey Hinton called it a "beautiful free lunch."



Machine Learning

Logistic Regression

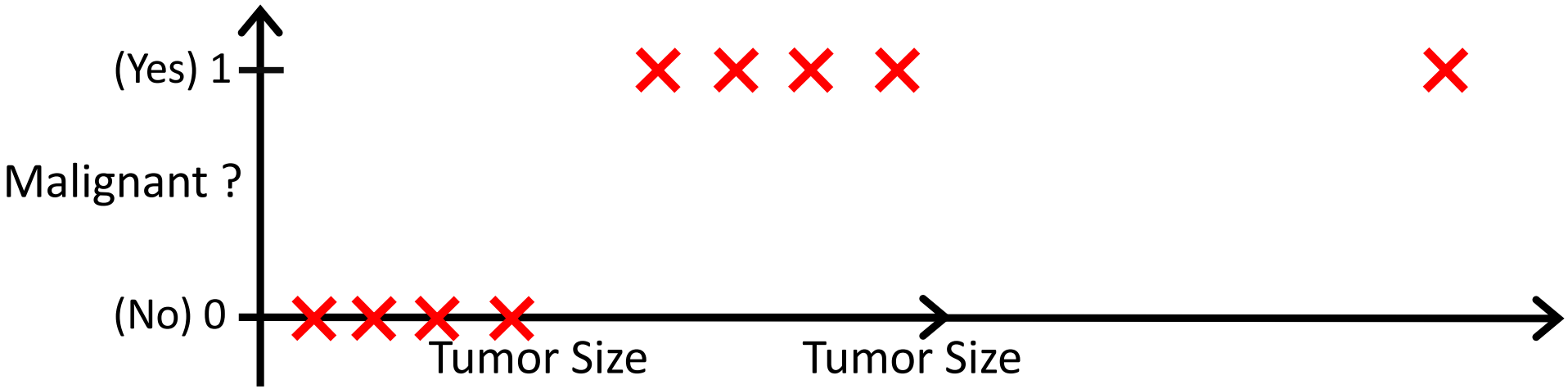
Classification

Classification

Email: Spam / Not Spam?
Online Transactions: Fraudulent (Yes / No)?
Tumor: Malignant / Benign ?

$y \in \{0, 1\}$

0: “Negative Class” (e.g., benign tumor)
1: “Positive Class” (e.g., malignant tumor)



Threshold classifier output $h_\theta(x)$ at 0.5:

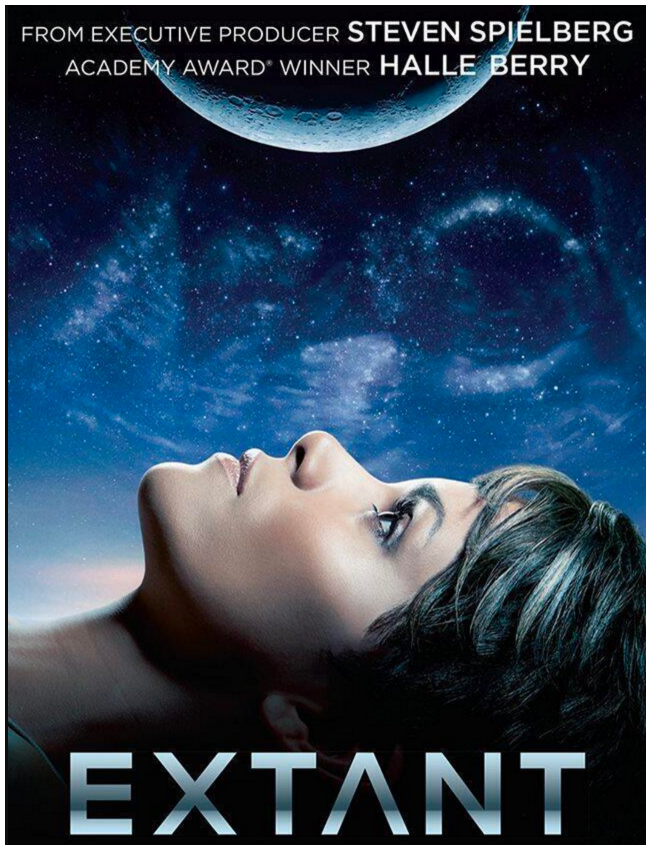
If $h_\theta(x) \geq 0.5$, predict “y = 1”

If $h_\theta(x) < 0.5$, predict “y = 0”

Classification: y = 0 or 1

$h_\theta(x)$ can be > 1 or < 0

Logistic Regression: $0 \leq h_\theta(x) \leq 1$



Machine Learning

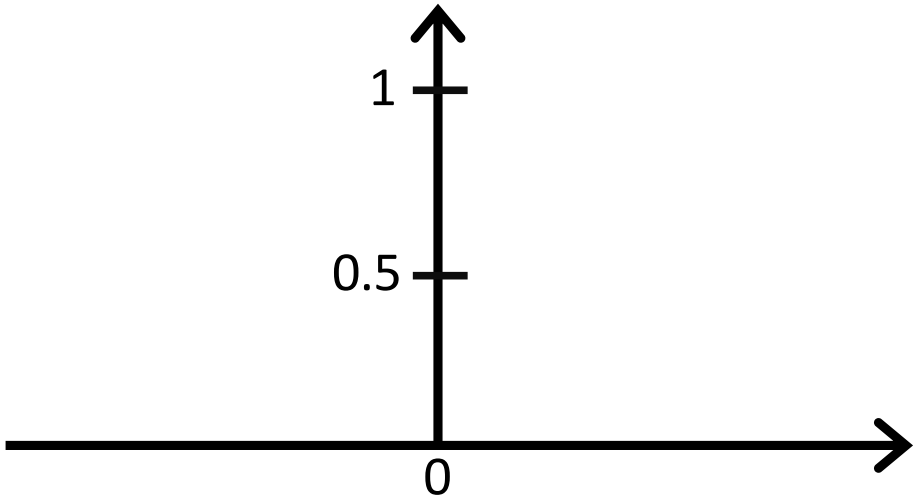
Logistic Regression

Hypothesis Representation

Logistic Regression Model

Want $0 \leq h_\theta(x) \leq 1$

$$h_\theta(x) = \quad \theta^T x$$



Sigmoid function

Logistic function

Interpretation of Hypothesis Output

$h_\theta(x)$ = estimated probability that y = 1 on input x

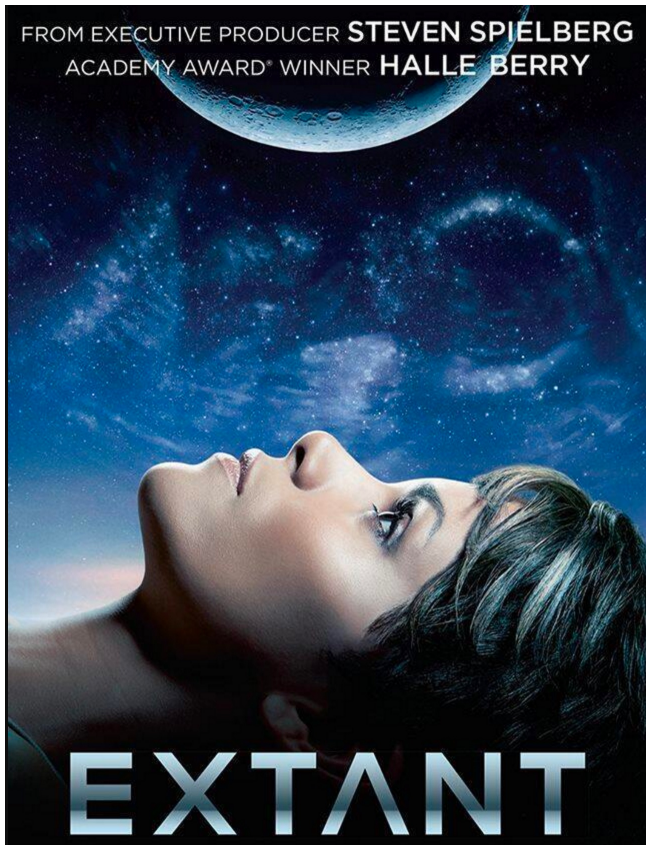
Example: If $x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{tumorSize} \end{bmatrix}$

$$h_\theta(x) = 0.7$$

Tell patient that 70% chance of tumor being malignant

"probability that y = 1, given x, parameterized by θ"

$$P(y = 0|x;\theta) + P(y = 1|x;\theta) = 1$$
$$P(y = 0|x;\theta) = 1 - P(y = 1|x;\theta)$$



Machine Learning

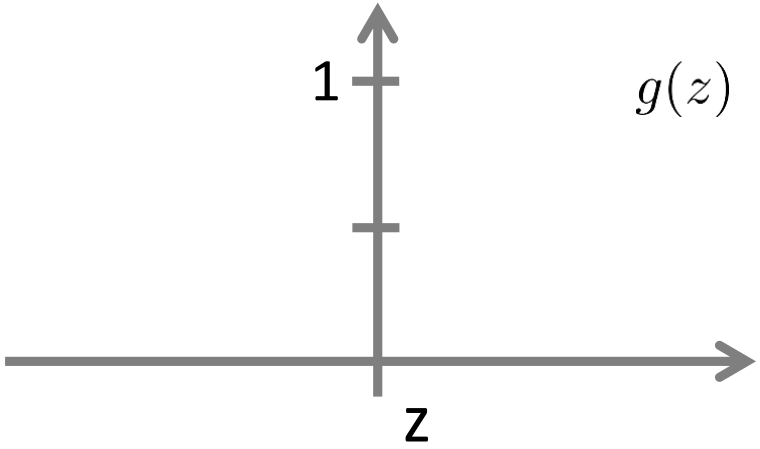
Logistic Regression

Decision boundary

Logistic regression

$$h_\theta(x) = g(\theta^T x)$$

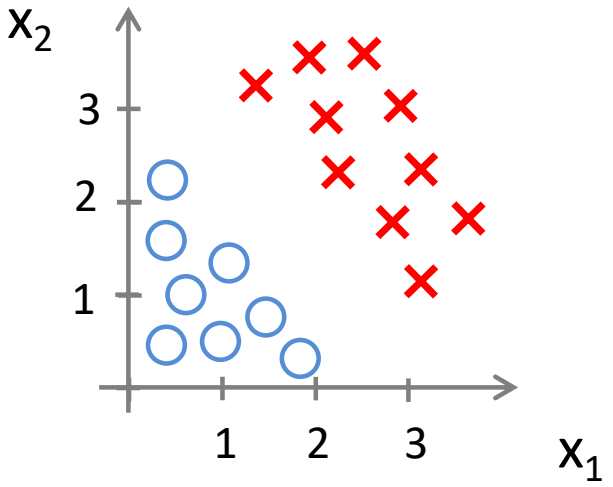
$$g(z) = \frac{1}{1+e^{-z}}$$



Suppose predict "$y = 1$" if $h_\theta(x) \geq 0.5$

predict "$y = 0$" if $h_\theta(x) < 0.5$

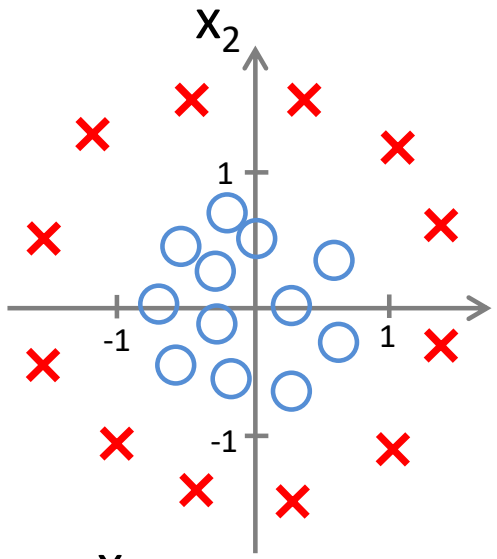
Decision Boundary



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

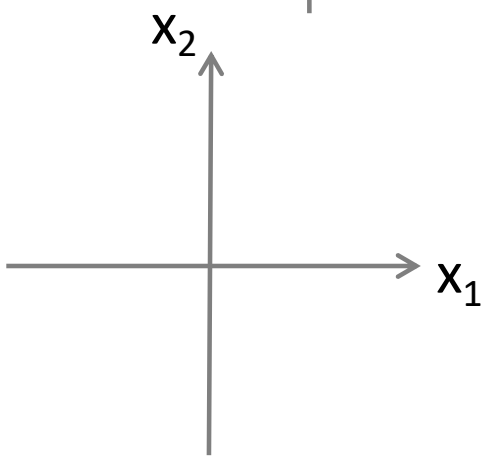
Predict "$y = 1$" if $-3 + x_1 + x_2 \geq 0$

Non-linear decision boundaries

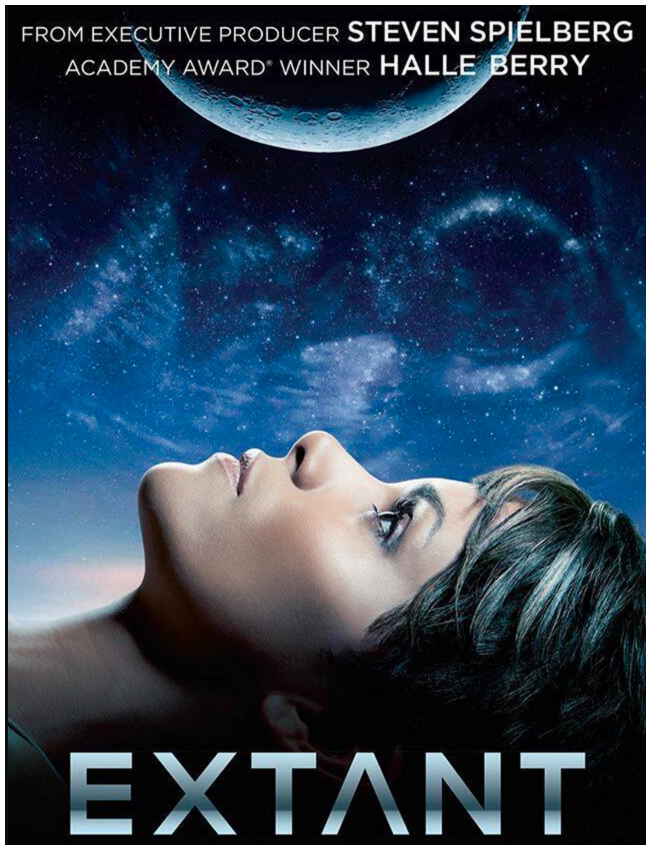


$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$$

Predict "$y = 1$" if $-1 + x_1^2 + x_2^2 \geq 0$



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^2 x_2^2 + \theta_6 x_1^3 x_2 + \dots)$$



Machine Learning

Logistic Regression

Cost function

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

m examples $\quad x \in \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} \quad x_0 = 1, y \in \{0, 1\}$

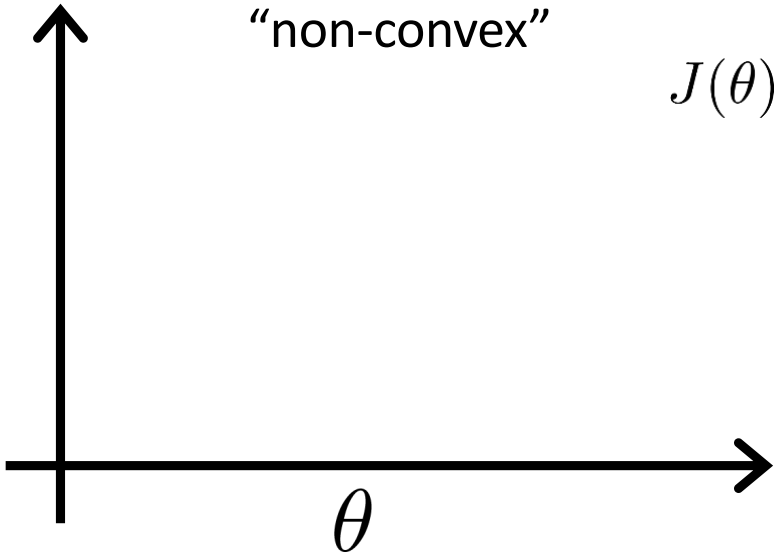
$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

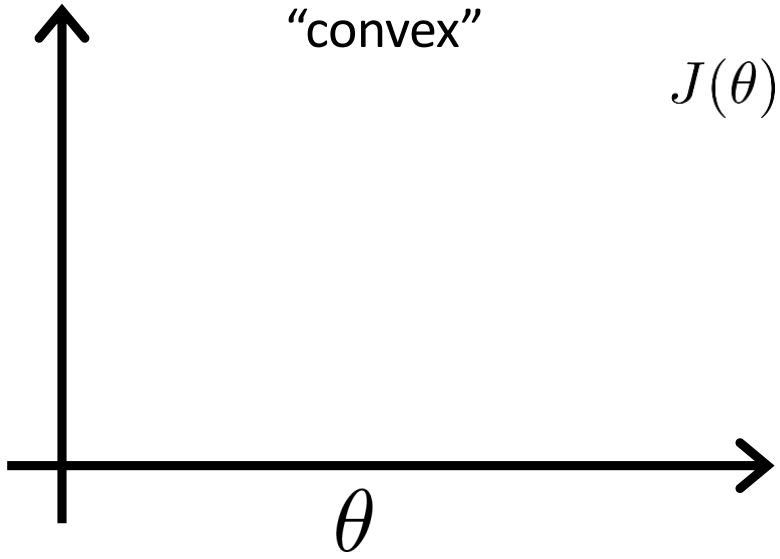
How to choose parameters θ ?

Cost function

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

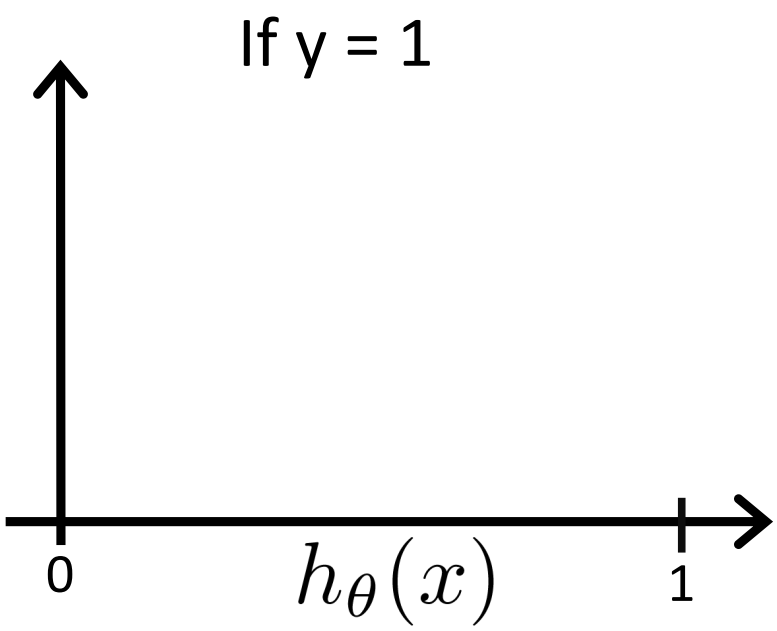
$$\text{Cost}(h_\theta(x^{(i)}), y^{(i)}) = \frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$





Logistic regression cost function

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$



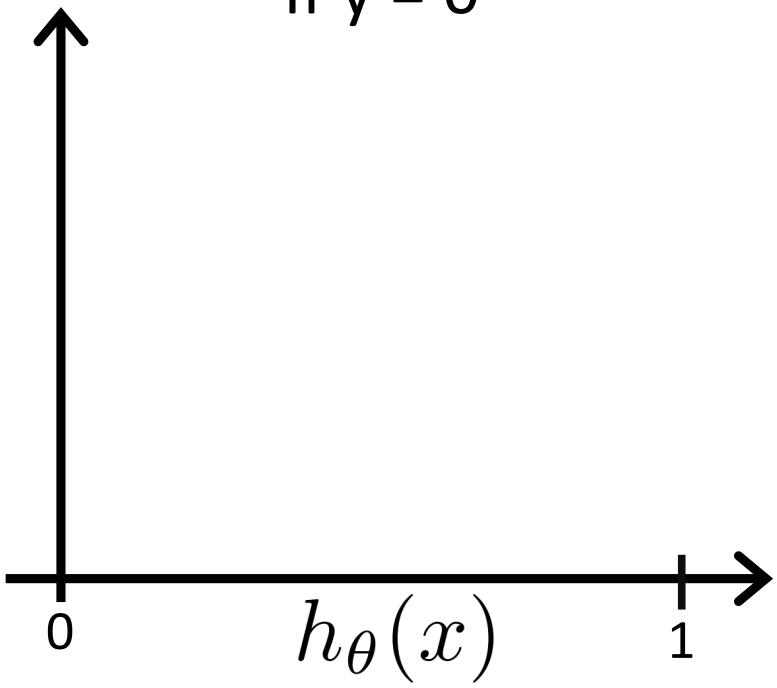
$\text{Cost} = 0$ if $y = 1, h_\theta(x) = 1$
But as $h_\theta(x) \to 0$
$Cost \to \infty$

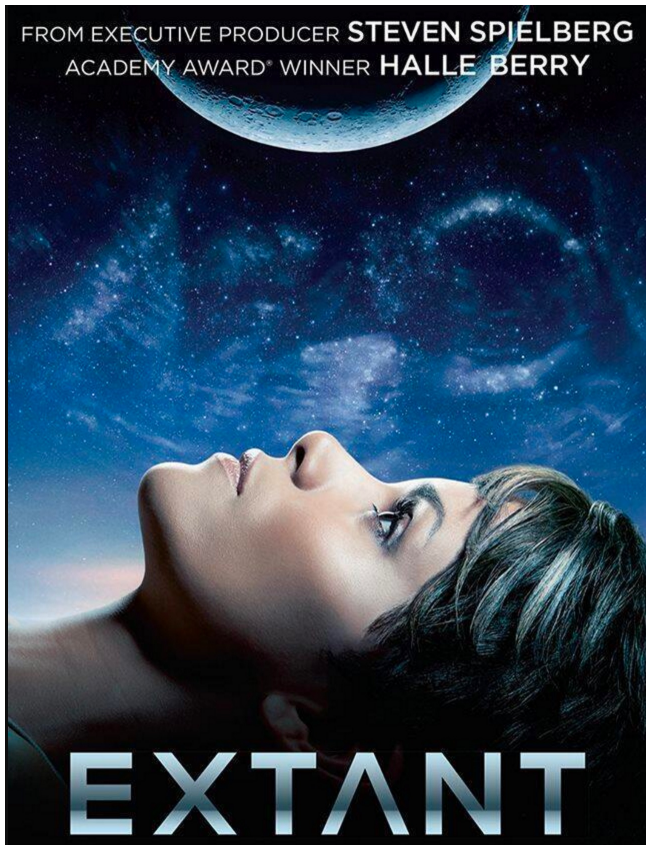
Captures intuition that if $h_\theta(x) = 0$, (predict $P(y = 1|x; \theta) = 0$), but $y = 1$, we'll penalize learning algorithm by a very large cost.

Logistic regression cost function

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

If y = 0





Machine Learning

Logistic Regression

Simplified cost function and gradient descent

Logistic regression cost function

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Note: $y = 0$ or 1 always

Logistic regression cost function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$= -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

To fit parameters θ:

$$\min_\theta J(\theta)$$

To make a prediction given new x:

Output $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

Gradient Descent

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

Want $\min_\theta J(\theta)$:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

(simultaneously update all θ_j)

$\}$

Gradient Descent

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

Want $\min_\theta J(\theta)$:

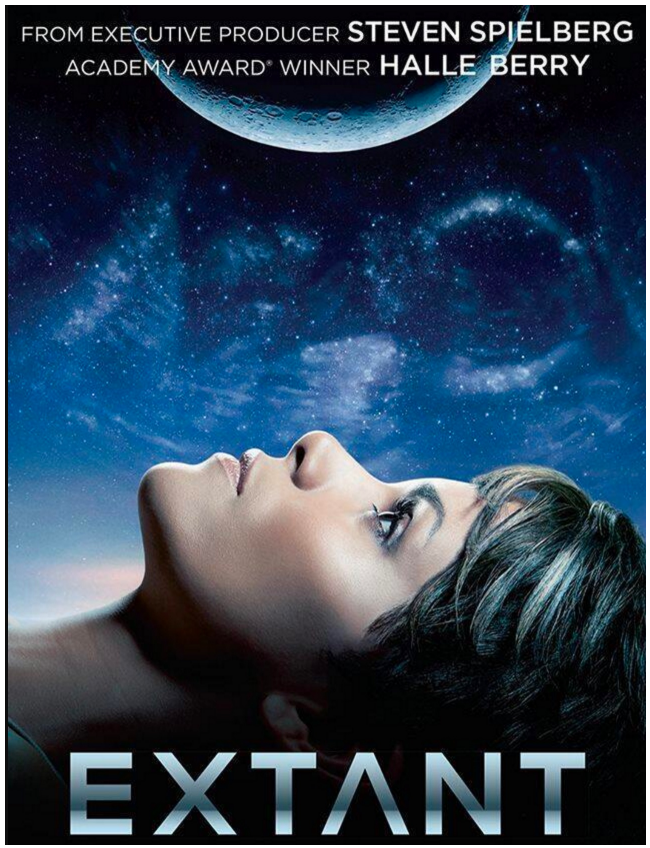
Repeat $\{$

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

$\}$

Algorithm looks identical to linear regression!



Machine Learning

Logistic Regression

Advanced optimization

Optimization algorithm

Cost function $J(\theta)$. Want $\min_\theta J(\theta)$.

Given θ, we have code that can compute

- $J(\theta)$
- $\frac{\partial}{\partial \theta_j} J(\theta)$ (for $j = 0, 1, \ldots, n$)

Gradient descent:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$\}$

Optimization algorithm

Given θ, we have code that can compute

- $J(\theta)$
- $\frac{\partial}{\partial \theta_j} J(\theta)$ (for $j = 0, 1, \ldots, n$)

Optimization algorithms:

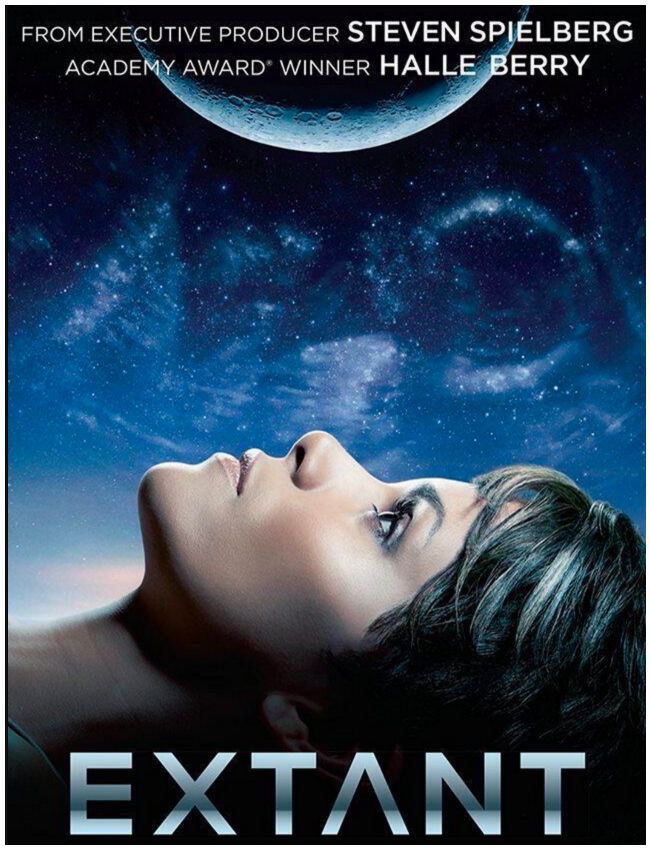
- Gradient descent
- Conjugate gradient
- BFGS
- L-BFGS

Advantages:

- No need to manually pick α
- Often faster than gradient descent.

Disadvantages:

- More complex



Machine Learning

Logistic Regression

Multi-class classification: One-vs-all

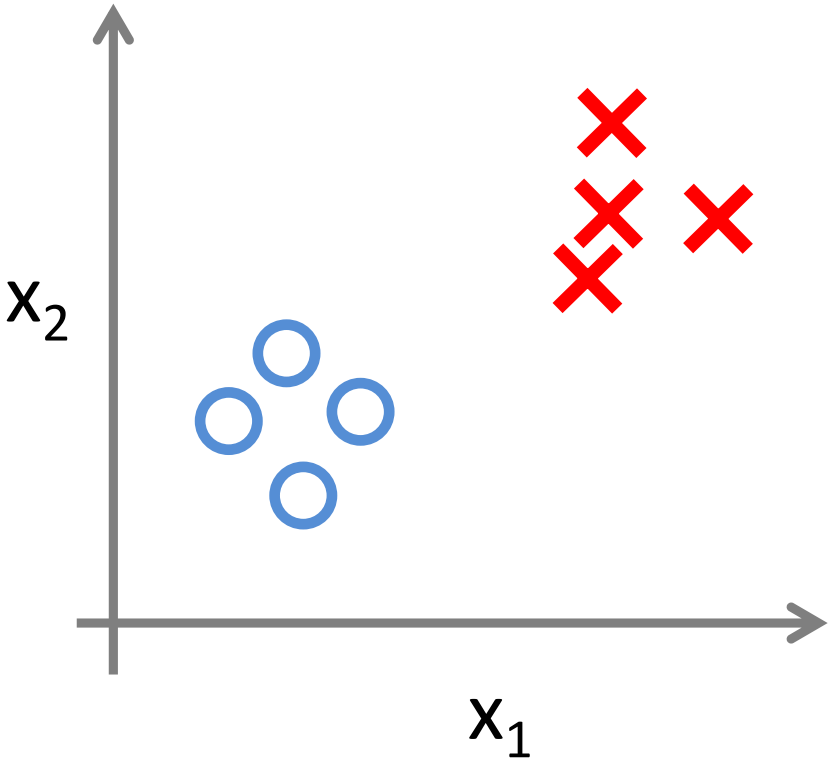
Multiclass classification

Email foldering/tagging: Work, Friends, Family, Hobby

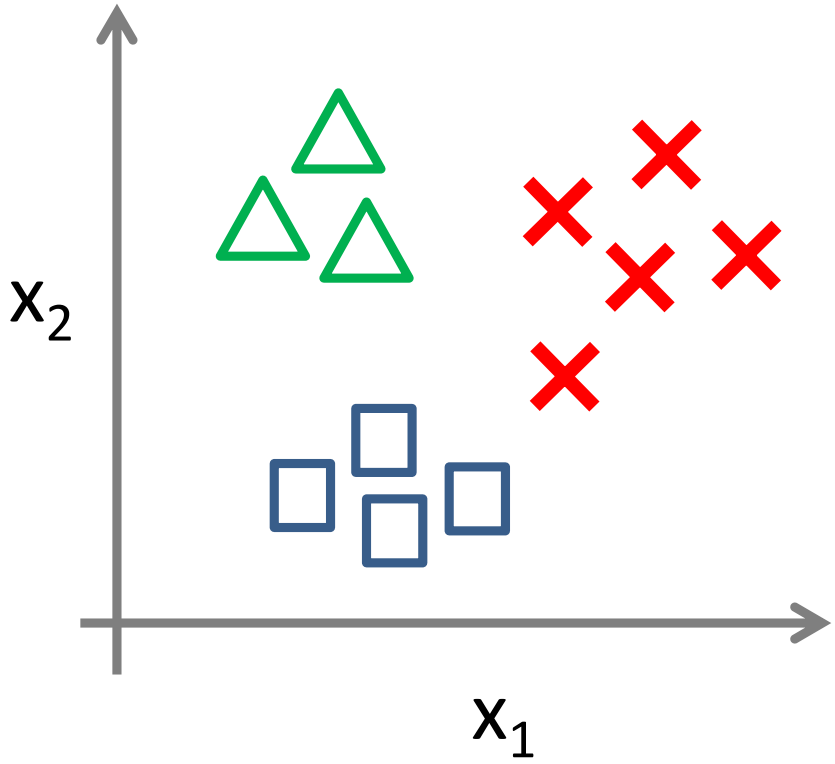
Medical diagrams: Not ill, Cold, Flu

Weather: Sunny, Cloudy, Rain, Snow

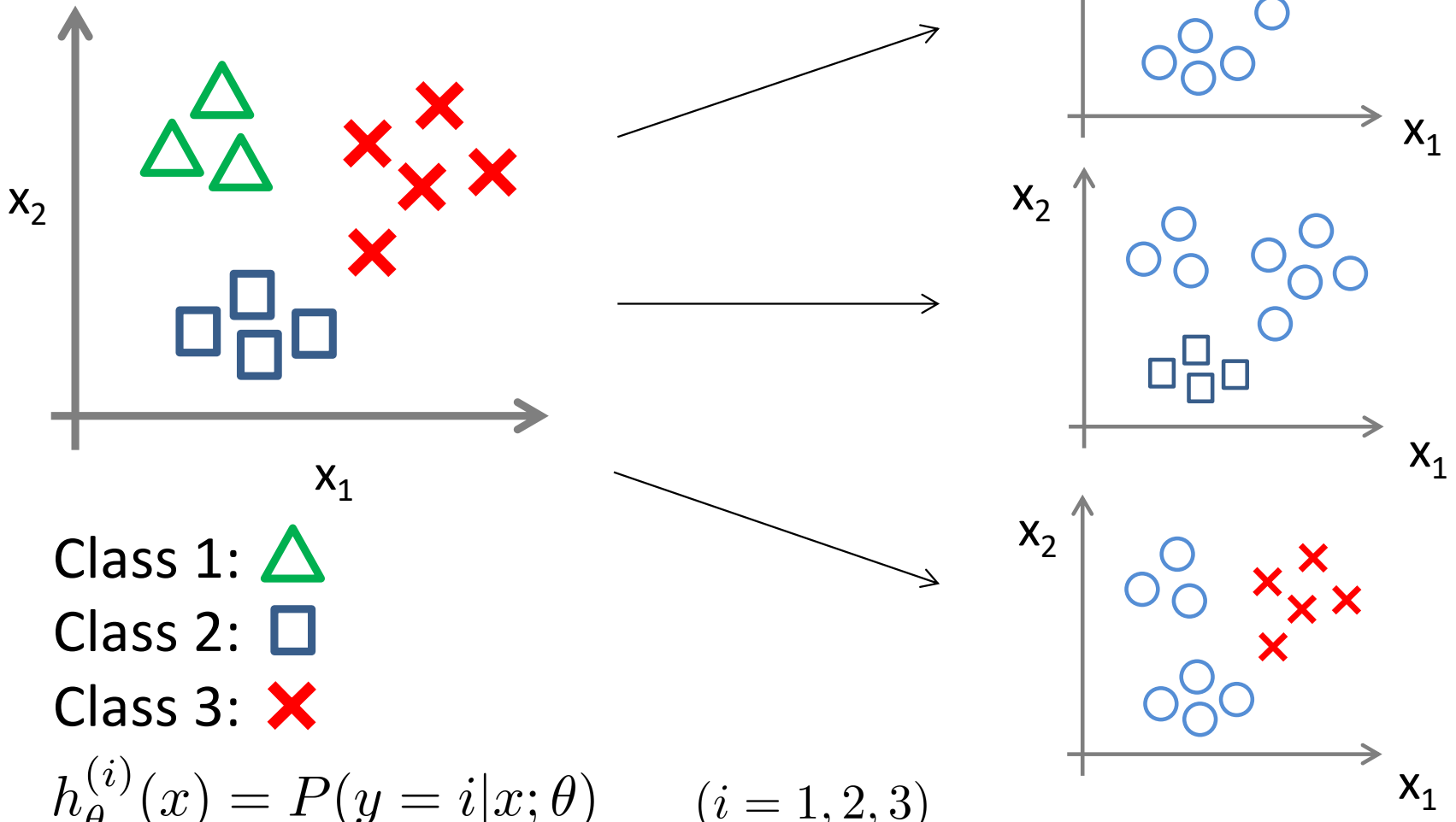
Binary classification:



Multi-class classification:



One-vs-all (one-vs-rest):



Class 1: △

Class 2: □

Class 3: ×

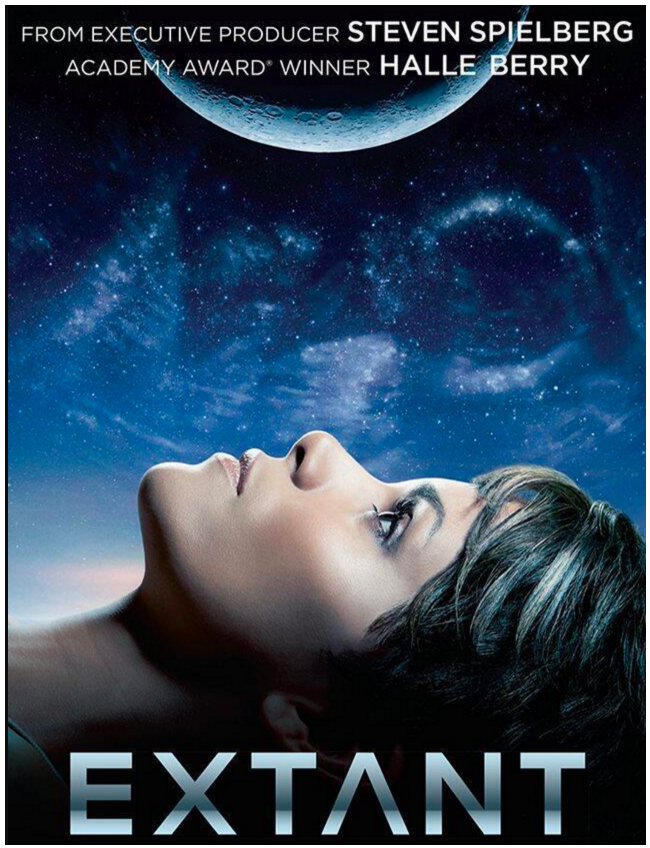
$h_\theta^{(i)}(x) = P(y = i|x;\theta) \qquad (i = 1, 2, 3)$

One-vs-all

Train a logistic regression classifier $h_\theta^{(i)}(x)$ for each class i to predict the probability that $y = i$.

On a new input x, to make a prediction, pick the class i that maximizes

$$\max_i h_\theta^{(i)}(x)$$



Machine Learning

Logistic Regression

Multi-class classification: softmax regression

softmax regression

Softmax function $$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k given the scores of each class for that instance.

Softmax Regression classifer prediction

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left(\theta_k{}^T \cdot \mathbf{x}\right)$$

softmax regression

Cross entropy cost function $J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$

Cross entropy gradient vector for class *k*

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\mathbf{x}^{(i)}$$

softmax regression

```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10, random_state=42)
softmax_reg.fit(X, y)
```

```
LogisticRegression(C=10, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class=u'multinomial',
          n_jobs=1, penalty='l2', random_state=42, solver=u'lbfgs',
          tol=0.0001, verbose=0, warm_start=False)
```

multi_class: “multinomial” Multi-class logistic regression
“ovr” one-vs-rest
solver: “lbfgs” “newton-cg” “liblinear”
C: The reciprocal of penalty coefficients