Linear Regression

Multivariate linear regression, polynomial regression

Khiem Nguyen

Email	khiem.nguyen@glasgow.ac.uk
MS Teams	khiem.nguyen@glasgow.ac.uk
Whatsapp	+44 7729 532071 (Emergency only)

May 18, 2025

University of Glasgow

Multiple features

	Age	Sex	BMI	BP	Y
(1)	59	2	32.1	101	151
(2)	48	1	21.6	87	75
(3)	72	2	30.5	93	141
(4)	24	1	25.3	84	206
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
(i)	$x_1^{(i)}$	$x_2^{(i)}$	$x_3^{(i)}$	$x_4^{(i)}$	$y^{(i)}$

Multiple features

	Age	Sex	BMI	BP	Y
(1)	59	2	32.1	101	151
(2)	48	1	21.6	87	75
(3)	72	2	30.5	93	141
(4)	24	1	25.3	84	206
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
(i)	$x_1^{(i)}$	$x_2^{(i)}$	$x_3^{(i)}$	$x_4^{(i)}$	$y^{(i)}$

x_j	j^{th} feature
n	number of features
$\vec{x}^{(i)}$	features of the i^{th} training example
$x_j^{(i)}$	value of feature j^{th} in the i^{th} training example

Multiple features

	Age	Sex	BMI	BP	Y
(1)	59	2	32.1	101	151
(2)	48	1	21.6	87	75
(3)	72	2	30.5	93	141
(4)	24	1	25.3	84	206
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
(i)	$x_1^{(i)}$	$x_2^{(i)}$	$x_3^{(i)}$	$x_4^{(i)}$	$y^{(i)}$

x_j	j^{th} feature
n	number of features
$\vec{x}^{(i)}$	features of the i^{th} training example
$x_j^{(i)}$	value of feature j^{th} in the i^{th} training example

$$\vec{x}^{(2)} = \begin{bmatrix} \underbrace{48}_{x_1^{(2)}} & \underbrace{1}_{x_2^{(2)}} & \underbrace{21.6}_{x_3^{(2)}} & \underbrace{87}_{x_4^{(2)}} \end{bmatrix}$$

$$x_3^{(2)} = 21.6$$

$$x_4^{(3)} = ? \qquad x_3^{(1)} = ? \qquad x_2^{(4)} = ?$$

Following this rule (in our lectures):

$\otimes^{(i)}$ refers to row i.

$\otimes_j$ refers to column j.

$\otimes_j^{(i)}$ refers to (row, column) $= (i, j)$.

In our lectures

- m training examples
- n features

Linear regression model

Previously: $f_{w,b}(x) = wx + b$

Linear regression model for multiple features (n features)

$$f_{\vec{w},b}(\vec{x}) = w_1x_1 + \cdots + w_nx_n + b = \sum_{j=1}^{n} w_jx_j + b$$

- $\vec{w} = \begin{bmatrix} w_1 & w_2 & \cdots & w_n \end{bmatrix}$ – vector of coefficients/weights of the model
- b – intercept, just a number (the base for the linear model at $\vec{x} = \vec{0}$)
- $\vec{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$ – vector of features

Vector notation

$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$$

Vectorization: What is it?

➥ Learnable/Trainable parameters

$$\vec{w} = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}$$

b is a real scalar (number)

➥ Features

$$\vec{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

Algebra counts from 1 - Python counts from 0:

❑ Code – Super stupid: Without loop ☹☹☹

```
w = np.array([1.0, 2.5, -2.0])
b = 1
x = np.array([1, 2, 3])
f = w[0] * x[0]
    + w[1] * x[1]
    + w[2] * x[2] + b
```

How about very large n?

$$f_{\vec{w},b}(\vec{x}) = \sum_{j=1}^{n} w_j x_j + b \quad \Big| \quad \sum_{j=1}^{n} \rightarrow j = 1, \dots n$$

❑ Code – Still stupid but less: With loop ☹☹

```
f = 0
for j in range(0, n):
    f = f + w[j] * x[j]
f = f + b
```

Vectorization $f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$

Best: Vectorization ☺ ☺ ☺

```
f = np.dot(w, x) + b
```

Vectorization: Why is it good?

Without vectorization

```python
for j in range(0, 16):
    f = f + w[j] * x[j]
```

t_0

`f + w[0] * x[0]`

t_1

`f + w[0] * x[0]`

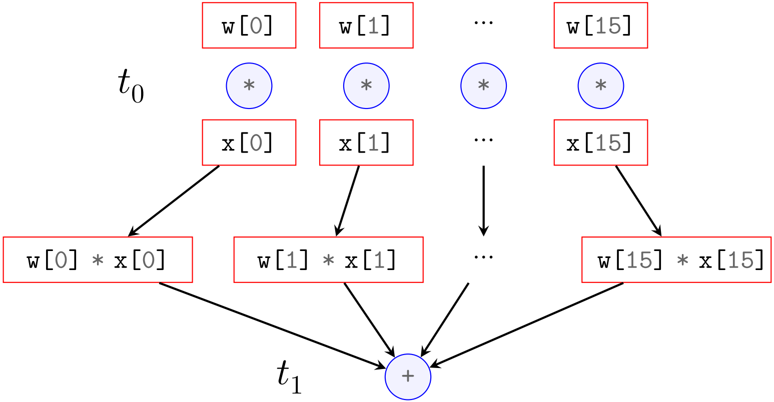
$\vdots$

t_{15}

`f + w[0] * x[0]`

Vectorization

`np.dot(w, x)`



Efficient $\longrightarrow$ Scale to large datasets

Vectorization for gradient descent

➥ *Assume that*

$\vec{w} = (w_1, w_2, \cdots, w_{16})$

$\vec{d} = (d_1, d_2, \cdots, d_{16})$

➥ *Python code*

```
w = np.array([0.5, 1.3, ..., 3.4])
d = np.array([0.3, 0.2, ..., 0.4])
```

Task: Compute $w_j = w_j - \alpha d_j$ for $j = 1, \ldots, 16$

Vectorization for gradient descent

➥ *Assume that*

$\vec{w} = (w_1, w_2, \cdots, w_{16})$

$\vec{d} = (d_1, d_2, \cdots, d_{16})$

➥ *Python code*

```
w = np.array([0.5, 1.3, ..., 3.4])
d = np.array([0.3, 0.2, ..., 0.4])
```

Task: Compute $w_j = w_j - \alpha d_j$ for $j = 1, \ldots, 16$

Without vectorization

```
alpha = 0.1
for j in range(0, 16):
    w[j] = w[j] - alpha * d[j]
```

Vectorization for gradient descent

➥ *Assume that*

$\vec{w} = (w_1, w_2, \cdots, w_{16})$

$\vec{d} = (d_1, d_2, \cdots, d_{16})$

➥ *Python code*

```
w = np.array([0.5, 1.3, ..., 3.4])
d = np.array([0.3, 0.2, ..., 0.4])
```

Task: Compute $w_j = w_j - \alpha d_j$ for $j = 1, \ldots, 16$

Without vectorization

```
alpha = 0.1
for j in range(0, 16):
    w[j] = w[j] - alpha * d[j]
```

With vectorization

```
alpha = 0.1
w = w - alpha * d
# Just one line of code
```

Vectorization: Example and time measurement

```
import numpy as np, time
x, y = np.arange(0, 10.0, 10.0/5e5), np.arange(0,  5.0,  5.0/5e5)
time_list = np.ndarray(50)
for run in range(50): # run 50 times to derive the average running time
    tstart = time.time(), s = 0
    for i in range(len(x)):
        s += x[i] * y[i]
    time_list[run] = time.time() - tstart
print(f"average time = {np.mean(time_list)*1000:.10f} miliseconds")
```

→ *Output* `average time = 134.4122457504 miliseconds`

```
for run in range(100):
    tstart = time.time(), s = np.dot(x, y)
    time_list[run] = time.time() - tstart
print(f"average time (vectorization): {np.mean(time_list) * 1000:.10f} miliseconds")
```

→ *Output* `average time (vectorization): 0.1322603226 miliseconds`

Vectorization leads to almost *1000 times* faster.

Gradient descent for multivariate linear regression

	Previous notation	**Vector notation**
Parameters	$w_1, \ldots, w_n, \quad b$	$\vec{w} = (w_1, \ldots, w_n), \quad b$
Model	$f_{\vec{w},b}(\vec{x}) = w_1 x_1 + \cdots + w_n x_n + b$	$f_{\vec{w},b} = \vec{w} \cdot \vec{x} + b$
Cost function	$J(w_1, \cdots, w_n, b)$	$J(\vec{w}, b)$

Gradient descent for multivariate linear regression

	Previous notation	**Vector notation**
Parameters	$w_1, \dots, w_n, \quad b$	$\vec{w} = (w_1, \dots, w_n), \quad b$
Model	$f_{\vec{w},b}(\vec{x}) = w_1 x_1 + \cdots + w_n x_n + b$	$f_{\vec{w},b} = \vec{w} \cdot \vec{x} + b$
Cost function	$J(w_1, \cdots, w_n, b)$	$J(\vec{w}, b)$

➥ **Gradient descent**

Repeat

$$\begin{cases} w_j = w_j - \alpha \dfrac{\partial J}{\partial w_j}(w_1, \dots, w_n, b) \\ b = b - \alpha \dfrac{\partial J}{\partial b}(w_1, \dots, w_n, b) \end{cases}$$

Repeat

$$\begin{cases} w_j = w_j - \alpha \dfrac{\partial J}{\partial w_j}(\vec{w}, b) \\ b = b - \alpha \dfrac{\partial J}{\partial b}(\vec{w}, b) \end{cases}$$

Repeat

$$\begin{cases} \vec{w} = \vec{w} - \alpha \nabla_{\vec{w}} J(\vec{w}, b) \\ b = b - \alpha \partial_b J(\vec{w}, b) \end{cases}$$

Gradient of cost function

Exactly like univariate linear regression

$$J(\vec{w},b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 = \frac{1}{2m}\sum_{i=1}^{m}(\vec{w}\cdot\vec{x}^{(i)} + b - y^{(i)})^2$$

$$= \frac{1}{2m}\sum_{i=1}^{m}(w_1 x_1^{(i)} + \cdots + w_n x_n^{(i)} + b - y^{(i)})^2 = \frac{1}{2}\sum_{i=1}^{m} G^{(i)}(w_1,\cdots,w_n,b)$$

with

$$G^{(i)}(w_1,\cdots,w_n,b) = \underbrace{(w_1 x_1^{(i)} + \cdots + w_n x_n^{(i)} + b - y^{(i)})^2}_{[(\cdots)^{(i)}]^2} \quad | \quad x_j^{(i)}, y^{(i)} \text{ are just constants here!}$$

Derivative of J with respect to w_1:

$$\frac{\partial J}{\partial w_1} = \frac{1}{2m}\sum_{i=1}^{m}\frac{\partial G^{(i)}}{\partial w_1} = \frac{1}{\cancel{2}m}\sum_{i=1}^{m}\overbrace{\cancel{2}(\cdots)^{(i)}\underbrace{\frac{\partial}{\partial w_1}(\cdots)^{(i)}}_{x_1^{(i)}}}^{\partial G^{(i)}/\partial w_1} = \frac{1}{m}\sum_{i=1}^{m}(\cdots)^{(i)} x_1^{(i)}$$

Gradient of cost function

Derivative of J with respect to w_1

$$\frac{\partial J}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m}(\cdots)^{(i)}x_1^{(i)} = \frac{1}{m}\sum_{i=1}^{m}(w_1x_1^{(i)} + \cdots + w_nx_n^{(i)} + b - y^{(i)})x_1^{(i)}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\Big(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\Big)x_1^{(i)}$$

Generalization:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\Big(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\Big)x_j^{(i)}, \quad j = 1, \dots, n$$

Computation of $\frac{\partial J}{\partial b}$ is the same but easier: ❑ Recall $G^{(i)} = w_1x_1^{(i)} + \cdots + w_nx_n^{(i)} + b$

$$\frac{\partial G^{(i)}}{\partial b} = (\cdots)^{(i)}\cancelto{1}{\frac{\partial}{\partial b}(\cdots)^{(i)}} = (\cdots)^{(i)} = (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})$$

$$\Rightarrow \frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}) \times 1$$

Gradient of cost function

Another way to view the result: If we define

$$w_0 = b, \quad x_0 = 1$$

we can write the model representation

$$f_{\mathbf{w}} = \underbrace{w_0 x_0}_{b \cdot 1} + \sum_{i=1}^{n} w_i x_i = \sum_{i=1}^{n} w_i x_i = \mathbf{w} \cdot \mathbf{x},$$

$$\text{with} \quad \mathbf{w} = (w_0, \vec{w}) = (w_0, \dots, w_n), \quad \mathbf{x} = (x_0, \vec{x}) = (x_0, \dots, x_n).$$

Gradient of the cost function $J(\mathbf{w}) = J(\vec{w}, b)$:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left(f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}, \quad j = 0, \dots, n$$

Why do we even bother to introduce b in the first place?

$\rightarrow$ Mathematical meaning: b is bias/intercept and also `sklearn` uses it.

Gradient descent in detail

- If we denote $x_0^{(i)} = 1$ for all training examples and $w_0 = b$ for the bias,

we can write the gradient descent update for both $\vec{w} = (w_1, \dots, w_n)$ and b according to

$$w_j = w_j - \alpha \underbrace{\frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}}_{\frac{\partial J}{\partial w_j}}, \quad j = 0, 1, \dots, n$$

- The partial derivatives of J w.r.t. w_j is calculated by the speaking rule

The error $(\hat{y} - y)$ times the input x_j.

(for all the examples and then summing)

Gradient descent in detail

1 feature

$$w = w - \alpha \underbrace{\frac{1}{m}\sum_{i}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)x^{(i)}}_{\frac{\partial J}{\partial w}(w,b)}$$

$$b = b - \alpha \underbrace{\frac{1}{m}\sum_{i}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)}_{\frac{\partial J}{\partial b}(w,b)}$$

Simultaneously update w, b

n features

$$w_1 = w_1 - \alpha \underbrace{\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_1^{(i)}}_{\frac{\partial J}{\partial w_1}(\vec{w},b)}$$

$$\vdots = \vdots$$

$$w_n = w_n - \alpha \underbrace{\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_n^{(i)}}_{\frac{\partial J}{\partial w_n}(\vec{w},b)}$$

$$b = b - \alpha \underbrace{\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)}_{\frac{\partial J}{\partial b}(\vec{w},b)}$$

Simultaneously update
$(w_1, \ldots, w_n), b$

An alternative to gradient descent

Normal equation

- Only for linear regression
- Solve for $\vec{w}, b$ without iterations

Disadvantanges

- Does not generalize to other learning algorithms
- Slow if number of features is large (≥ 10.000)

What you need to know

- *Normal equation* method may be used in machine learning libraries that implement linear regression.
- Gradient descent is the recommended method for finding parameters $\vec{w}, b$.

Normal equation for linear regression

Optional: *Just for those who like a bit of mathematics*

- Define:

$$w_0 = b, \qquad \underbrace{\Theta}_{\mathbb{R}^{(n+1)\times 1}} = \begin{bmatrix} w_0 \\ \cdots \\ w_n \end{bmatrix}, \qquad \underbrace{\mathbf{X}}_{\mathbb{R}^{m\times(n+1)}} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}, \qquad \underbrace{\mathbf{y}}_{\mathbb{R}^{m\times 1}} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

- Then:

$$\underbrace{\mathbf{f}}_{\mathbb{R}^{m\times 1}} := f_{\vec{w},b}(\mathbf{X}) = \begin{bmatrix} f_{\vec{w},b}(\vec{x}^{(1)}) \\ \vdots \\ f_{\vec{w},b}(\vec{x}^{(m)}) \end{bmatrix} = \begin{bmatrix} w_0 + w_1 x_1^{(1)} + \cdots + w_n x_n^{(1)} \\ \vdots \\ w_0 + w_1 x_1^{(m)} + \cdots + w_n x_n^{(m)} \end{bmatrix} = \mathbf{X}\Theta$$

- Cost function:

$$\begin{aligned} J(\Theta) := J(\vec{w}, b) &= (\mathbf{X}\Theta - \mathbf{y})^T(\mathbf{X}\Theta - \mathbf{y}) \in \mathbb{R}^{1\times m} \cdot \mathbb{R}^{m\times 1} \\ &= \Theta^T(\underbrace{\mathbf{X}^T\mathbf{X}}_{\mathbf{A}})\Theta - \Theta^T\mathbf{X}^T\mathbf{y} - \underbrace{\mathbf{y}\mathbf{X}\Theta}_{\Theta^T\mathbf{X}^T\mathbf{y}} + \mathbf{y}^T\mathbf{y} \end{aligned}$$

Normal equation: derivation

- Cost function:

$$J(\Theta) = \Theta^T(\underbrace{\mathbf{X}^T\mathbf{X}}_{\mathbf{A}})\Theta - 2\Theta^T\mathbf{X}^T + \mathbf{y}\mathbf{y}^T\mathbf{y}$$
$$= \sum_{i=1}^{n+1}\sum_{j=1}^{n+1}\theta_i A_{ij}\theta_j - 2\sum_{s=1}^{m}\sum_{i=1}^{n+1} X_{si}\theta_i y^{(s)} + \sum_{s=1}^{m} y_s^2$$

Normal equation: derivation

- Cost function:

$$J(\Theta) = \Theta^T(\underbrace{\mathbf{X}^T\mathbf{X}}_{\mathbf{A}})\Theta - 2\Theta^T\mathbf{X}^T + \mathbf{y}\mathbf{y}^T\mathbf{y}$$
$$= \sum_{i=1}^{n+1}\sum_{j=1}^{n+1}\theta_i A_{ij}\theta_j - 2\sum_{s=1}^{m}\sum_{i=1}^{n+1} X_{si}\theta_i y^{(s)} + \sum_{s=1}^{m} y_s^2$$

- Derivatives of cost function $J(\Theta)$:

$$\begin{aligned}\frac{\partial J}{\partial \theta_k} &= \sum_{i=1}^{n+1}\sum_{i=1}^{n+1}\left(\frac{\partial \theta_i}{\partial \theta_k}A_{ij}\theta_j + \theta_i A_{ij}\frac{\partial \theta_j}{\partial \theta_k}\right) - 2\sum_{s=1}^{m}\sum_{i=1}^{n+1} X_{si}\frac{\partial \theta_i}{\partial \theta_k}y^{(s)} \\ &= \sum_{i=1}^{n+1}\sum_{i=1}^{n+1}(\delta_{ik}A_{ij}\theta_j + \theta_i A_{ij}\delta_{jk}) - 2\sum_{s=1}^{m}\sum_{i=1}^{n+1} X_{si}\delta_{ik}y^{(s)}\end{aligned}, \quad \text{with } \delta_{ik} = \begin{cases}1 & i = k\\ 0 & i \neq k\end{cases}$$

$$\sum_{i=1}^{n+1}\delta_{ik}A_{ij} = A_{kj}, \quad \sum_{j=1}^{n+1}A_{ij}\delta_{jk} = A_{ik}, \quad A_{ki} = A_{ik}$$

$$\Longrightarrow \quad \frac{\partial J}{\partial \theta_k} = \sum_{j=1}^{n+1}A_{kj}\theta_j + \sum_{j=1}^{n+1}A_{kj}\theta_j - 2\sum_{s=1}^{m}X_{sk}y^{(s)}$$

Normal equation: derivation

- Repeat the derivative of cost function $J = J(\theta_1, \ldots, \theta_{n+1})$

$$\frac{\partial J}{\partial \theta_k} = 2\sum_{j=1}^{n+1} A_{kj}\theta_j - 2\sum_{s=1}^{m} X_{sk}y^{(s)} \quad \forall k = 1, \ldots n+1$$

- Vector notation

$$\frac{\partial J}{\partial \Theta} = 2(\mathbf{X}^T\mathbf{X})\Theta - 2\mathbf{X}^T\mathbf{y}$$

- Parameters are determined via

$$\frac{\partial J}{\partial \Theta} = \mathbf{0} \quad \Leftrightarrow \quad \mathbf{X}^T\mathbf{X}\Theta - \mathbf{X}^T\mathbf{y} = \mathbf{0} \quad \Leftrightarrow \quad \Theta = (\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y})$$

- Regardless of the number of training examples:

$$\mathbf{X}^T\mathbf{X} \in \mathbb{R}^{(n+1)\times(n+1)}, \quad \mathbf{X}^T\mathbf{y} \in \mathbb{R}^{(n+1)\times 1}$$

Normal equation: example

- Consider one simple example with data set

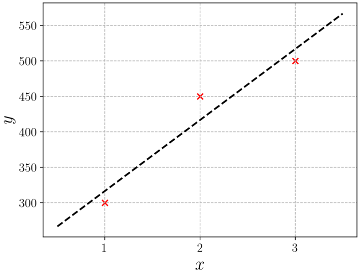
$$\mathrm{D} = \{\mathbf{p}^{(1)} = (1, 300), \mathbf{p}^{(2)} = (2, 450), \mathbf{p}^{(2)} = (3, 500)\}$$

- Then:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 300 \\ 450 \\ 500 \end{bmatrix}, \quad \mathbf{X}^T\mathbf{X} = \begin{bmatrix} 3 & 6 \\ 6 & 14 \end{bmatrix}, \quad (\mathbf{X}^T\mathbf{X})^{-1} = \begin{bmatrix} 7/3 & -1 \\ -1 & 5 \end{bmatrix}$$

$$\Theta = \begin{bmatrix} 7/3 & -1 \\ -1 & 5 \end{bmatrix} \begin{bmatrix} 1250 \\ 2700 \end{bmatrix} = \begin{bmatrix} 216.6666\ldots \\ 100 \end{bmatrix}$$

$$\Rightarrow \begin{cases} b = w_0 = 100 \\ w = w_1 = 216.666\ldots \end{cases}$$



Normal equation: Quick python code

Using *normal equation*

```
x_train = np.array([1, 2, 3])
y_train = np.array([300, 450, 500])

# Vector of ones of same size as x_train
vec_ones = np.ones_like(x_train)
X = np.vstack((vec_ones, x_train)).T

A = X.T @ X
r = X.T @ y_train

Theta = np.linalg.inv(A) @ r
```

Normal equation: Quick python code

Using *sklearn*

```
x_train = np.array([1, 2, 3])
y_train = np.array([300, 450, 500])

from sklearn.linear_model import LinearRegression
linear_regr_model = LinearRegression()
X_train = x_train.reshape((-1, 1))
 # or x_train = np.expand_dims(x_train, axis=1)
linear_regr_model.fit(X_train, y_train)
# weights w in the linear regression model
linear_regr_model.coef_
# bias/intercept b
linear_regr_model.intercept_
```

We will revisit this in the end of the lecture!

Feature scaling

Let us look at the whole data set

Age	Sex	BMI	BP	S1	S2	S3	S4	S5	S6	Y
59	2	32.1	101	157	93.2	38	4	4.8598	87	151
48	1	21.6	87	183	103.2	70	3	3.8918	69	75
24	1	25.3	84	198	131.4	40	5	4.8903	89	206
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

The ranges of data features are very different:

- Age: $20+ \to 80$
- Sex: $1 \to 2$
- BMI: $19 \to 37$
- S1 : $150 \to 200$
- S3: $30 \to 100$
- S4: $2 \to 10$
- S5: $\sim 3 \to \sim 4$

Feature scaling

For simplicity: Oversimplified house price data

Size $[dm^2]$	# bedrooms	Price [1000 £]
6000	2	150
4500	1	100
5500	1	120
7000	3	180
...	...	...

The predicted house price according to linear regression:

$$\hat{y} = w_1 \underbrace{x_1}_{\text{size}} + w_2 \underbrace{x_2}_{\#\text{ bedrooms}} + b$$

How about the size of weight parameters w_1, w_2?

House: $x_1 = 6500, \quad x_2 = 2, \quad y = 150\ [1000£]$

$w_1 = 2, \quad w_2 = 3, \quad b = 40$

$$\Rightarrow \hat{y} = 2 \times 6500 + 3 \times 2 + 40$$
$$= 13000 + 6 + 40 = 13046$$

13.046.000 £ nonsense!

$w_1 = 0.01, \quad w_2 = 25, \quad b = 40$

$$\Rightarrow \hat{y} = 0.01 \times 6500 + 25 \times 2 + 40$$
$$= 65 + 50 + 40 = 145\ [1000£]$$

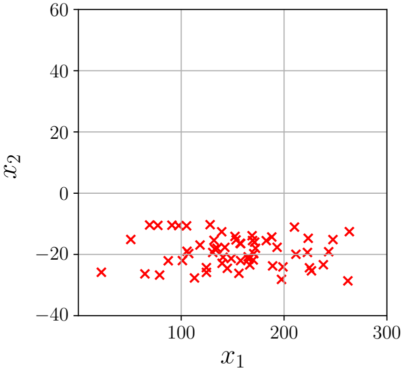
145.000 £ reasonable

Feature scaling

Feature size and parameter size

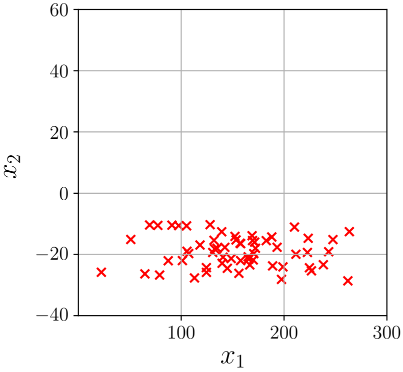
	size of feature x_j	size of parameter w_j
size in dm^2		
#bedrooms		

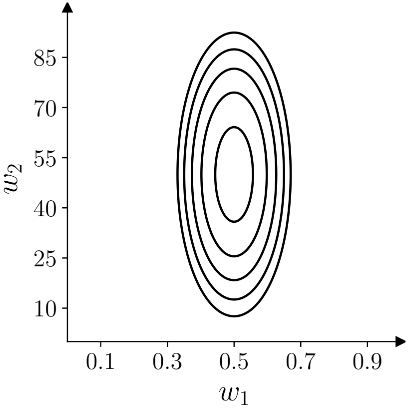
Feature size and gradient descent



Figures are hypothetical, not produced by an actual linear regression problem.

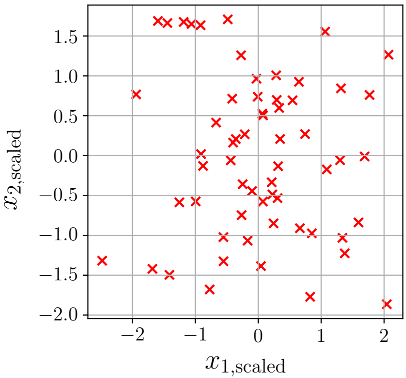
Feature size and gradient descent





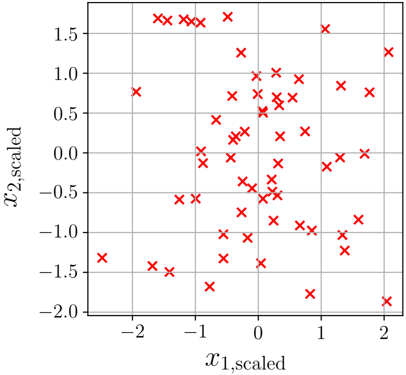
Figures are hypothetical, not produced by an actual linear regression problem.

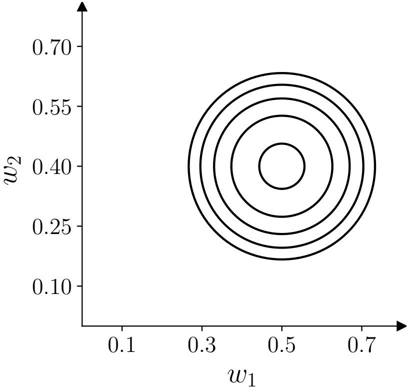
Feature size and gradient descent



Figures are hypothetical, not produced by an actual linear regression problem.

Feature size and gradient descent

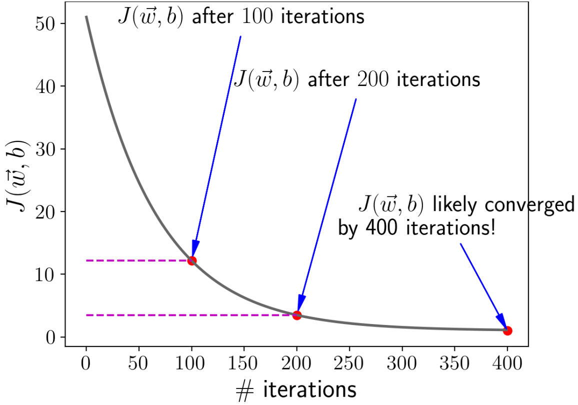




Figures are hypothetical, not produced by an actual linear regression problem.

Make sure gradient descent is working correctly

Objective: $\min_{\vec{w},b} J(\vec{w}, b)$



$J(\vec{w}, b)$ should decrease after every iteration!

Automatic convergence test:

Let ε ('epsilon') be 10^{-3}.

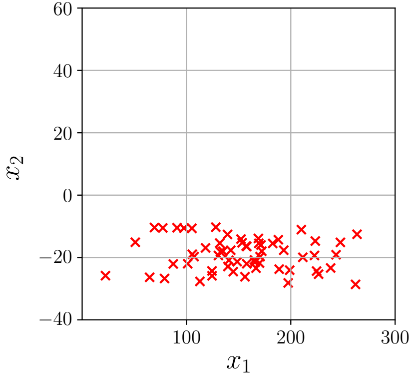
If $J(\vec{w}, b)$ decreases by $\leq \varepsilon$ in one iteration, declare **convergence**

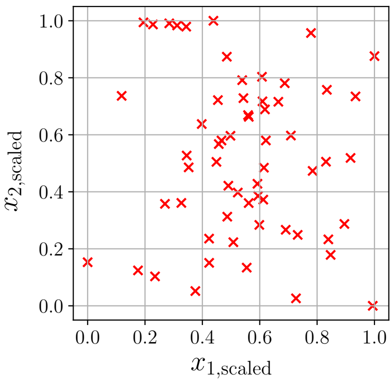
➜ Found parameters $\vec{w}, b$ to get close to global minimum.

Feature scaling: min-max normalization

$$x_{1,\text{scaled}} = \frac{x_1 - \min(x_1)}{\max(x_1) - \min(x_1)}, \quad x_{2,\text{scaled}} = \frac{x_2 - \min(x_2)}{\max(x_2) - \min(x_2)}$$

$$\implies \quad 0 \leq x_{1,\text{scaled}} \leq 1, \quad 0 \leq x_{2,\text{scaled}} \leq 1$$





Feature scaling: Mean normalization

$$\min(x_1) \leq x_1 \leq \max(x_1) \quad \min(x_2) \leq x_2 \leq \max(x_2), \quad \mu_1 = \frac{1}{m}\sum_{i=1}^{m} x_1^{(i)}, \quad \mu_2 = \frac{1}{m}\sum_{i=1}^{m} x_2^{(i)}$$

$$x_{1,\text{scaled}} = \frac{x_1 - \mu_1}{\max(x_1) - \min(x_1)}, \quad x_{2,\text{scaled}} = \frac{x_2 - \mu_2}{\max(x_2) - \min(x_2)}$$

Feature scaling: Mean normalization

$$\min(x_1) \le x_1 \le \max(x_1) \quad \min(x_2) \le x_2 \le \max(x_2), \quad \mu_1 = \frac{1}{m}\sum_{i=1}^{m} x_1^{(i)}, \quad \mu_2 = \frac{1}{m}\sum_{i=1}^{m} x_2^{(i)}$$

$$x_{1,\text{scaled}} = \frac{x_1 - \mu_1}{\max(x_1) - \min(x_1)}, \quad x_{2,\text{scaled}} = \frac{x_2 - \mu_2}{\max(x_2) - \min(x_2)}$$

$$\implies \begin{cases} -1 \le \frac{\min(x_1)-\mu_1}{\max(x_1)-\min(x_1)} \le x_{1,\text{scaled}} \le \frac{\max(x_1)-\mu_1}{\max(x_1)-\min(x_1)} \le 1, \\ -1 \le \frac{\min(x_2)-\mu_2}{\max(x_2)-\min(x_2)} \le x_{2,\text{scaled}} \le \frac{\max(x_2)-\mu_2}{\max(x_2)-\min(x_2)} \le 1 \end{cases}$$

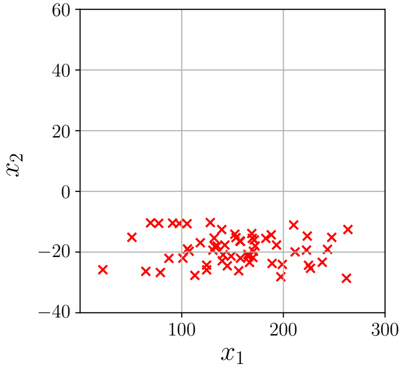
$$-1 \le \frac{\min(x_1) - \mu_1}{\max(x_1) - \min(x_1)} \Leftrightarrow \quad \min(x_1) - \max(x_1) \le \min(x_1) - \mu_1 \Leftrightarrow \quad \mu_1 < \max(x_1)[\text{True}]$$

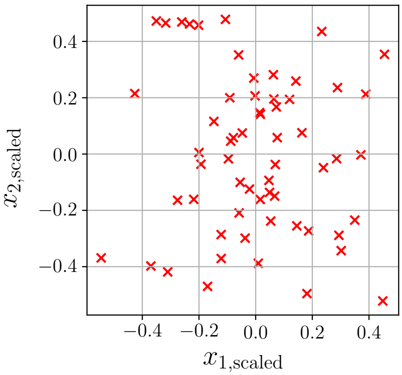
$$\frac{\max(x_1) - \mu_1}{\max(x_1) - \min(x_1)} \le 1 \Leftrightarrow \quad \max(x_1) - \mu_1 \le \max(x_1) - \min(x_1) \Leftrightarrow \quad \min(x_1) \le \mu_1 \quad [\text{True}]$$

Feature scaling: Mean normalization

Notation: $\text{mean}(x_1) = \mu_1, \quad \text{mean}(x_2) = \mu_2$

$$-1 \leq x_{1,\text{scaled}} = \frac{x_1 - \text{mean}(x_1)}{\max(x_1) - \min(x_1)} \leq 1, \quad -1 \leq x_{2,\text{scaled}} = \frac{x_2 - \text{mean}(x_2)}{\max(x_2) - \min(x_2)} \leq 1$$

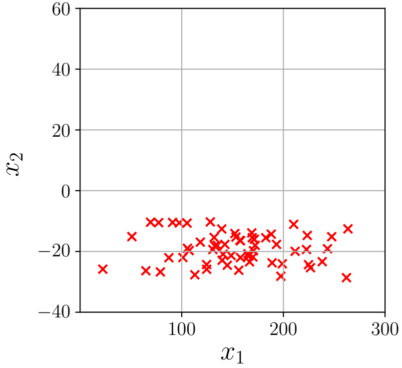


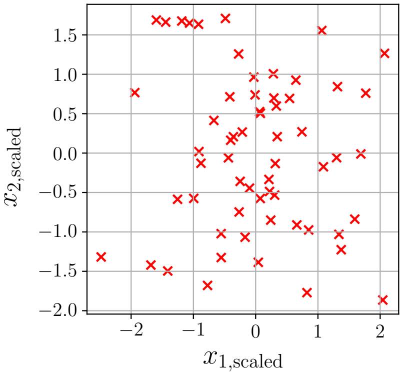


Feature scaling: Z-score normalization

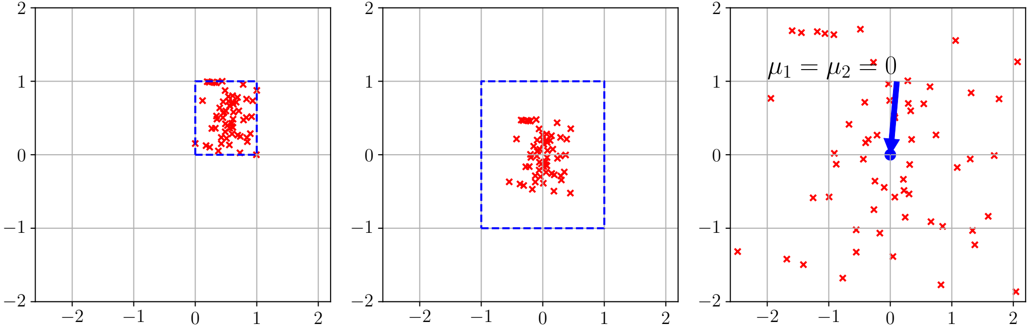
- Mean value: $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$
- Standard deviation: $\sigma_j = \left[\frac{1}{m}\sum_{i=1}^{m}\left(x_j^{(i)} - \mu_j\right)^2\right]^{1/2}$

$$x_{j,\text{scaled}} = \frac{x_j - \mu_j}{\sigma_j} \quad \rightarrow \quad \text{mean}(x_{j,\text{scaled}}) = \frac{\mu_j - \mu_j}{\sigma_j} = 0$$





Feature scaling: Three normalization formulas put together



- min-max normalization $\rightarrow$ scaled data in the bounding box $[0, 1]^n$
- mean normalization $\rightarrow$ scaled data in the bounding box $[-1, 1]^n$, mean of each scaled features is 0, $\mu_j^{\text{scaled}} = 0$, $j = 1, \dots, n$
- Z-score normalization $\rightarrow$ mean of each scaled feature is 0, $\mu_j^{\text{scaled}} = 0$, $j = 1, \dots, n$

Feature scaling: Some considerations

- Aim for about $-1 \leq x_j \leq 1$ for each feature x_j

Feature scaling: Some considerations

- Aim for about $-1 \leq x_j \leq 1$ for each feature x_j

$$\left.\begin{array}{r} -3.0 \leq x_j \leq 3.0 \\ -0.3 \leq x_j \leq 0.3 \end{array}\right\} \text{acceptable ranges}$$

Feature scaling: Some considerations

- Aim for about $-1 \le x_j \le 1$ for each feature x_j

$$\left.\begin{aligned} -3.0 \le x_j \le 3.0 \\ -0.3 \le x_j \le 0.3 \end{aligned}\right\} \quad \text{acceptable ranges}$$

- Further examples:

$$0 \le x_1 \le 3 \qquad\qquad \text{okay, no rescaling}$$

Feature scaling: Some considerations

- Aim for about $-1 \leq x_j \leq 1$ for each feature x_j

$$\left.\begin{array}{r} -3.0 \leq x_j \leq 3.0 \\ -0.3 \leq x_j \leq 0.3 \end{array}\right\} \quad \text{acceptable ranges}$$

- Further examples:

$$0 \leq x_1 \leq 3 \qquad \text{okay, no rescaling}$$

$$-2 \leq x_2 \leq 0.5 \qquad \text{okay, no rescaling}$$

Feature scaling: Some considerations

- Aim for about $-1 \leq x_j \leq 1$ for each feature x_j

$$\left.\begin{array}{r} -3.0 \leq x_j \leq 3.0 \\ -0.3 \leq x_j \leq 0.3 \end{array}\right\} \quad \text{acceptable ranges}$$

- Further examples:

$$\begin{array}{rl} 0 \leq x_1 \leq 3 & \text{okay, no rescaling} \\ -2 \leq x_2 \leq 0.5 & \text{okay, no rescaling} \\ -100 \leq x_3 \leq 100 & \text{too large} \rightarrow \text{rescale} \end{array}$$

Feature scaling: Some considerations

- Aim for about $-1 \leq x_j \leq 1$ for each feature x_j

$$\left.\begin{array}{c} -3.0 \leq x_j \leq 3.0 \\ -0.3 \leq x_j \leq 0.3 \end{array}\right\} \quad \text{acceptable ranges}$$

- Further examples:

$$\begin{array}{cl} 0 \leq x_1 \leq 3 & \text{okay, no rescaling} \\ -2 \leq x_2 \leq 0.5 & \text{okay, no rescaling} \\ -100 \leq x_3 \leq 100 & \text{too large} \rightarrow \text{rescale} \\ -0.001 \leq x_4 \leq 0.001 & \text{too small} \rightarrow \text{rescale} \end{array}$$

Feature scaling: Some considerations

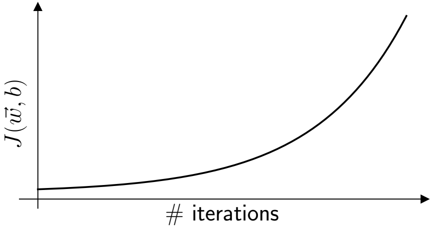
- Aim for about $-1 \leq x_j \leq 1$ for each feature x_j

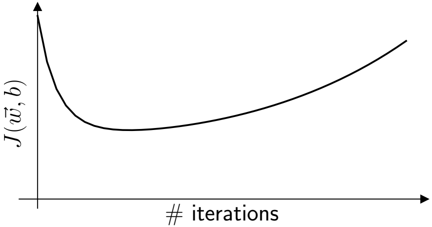
$$\left.\begin{array}{c} -3.0 \leq x_j \leq 3.0 \\ -0.3 \leq x_j \leq 0.3 \end{array}\right\} \quad \text{acceptable ranges}$$

- Further examples:

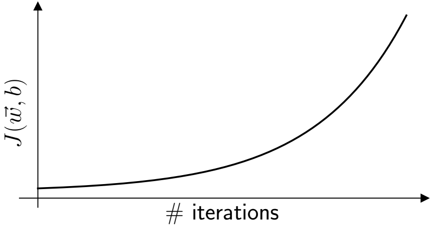
$0 \leq x_1 \leq 3$	okay, no rescaling
$-2 \leq x_2 \leq 0.5$	okay, no rescaling
$-100 \leq x_3 \leq 100$	too large $\rightarrow$ rescale
$-0.001 \leq x_4 \leq 0.001$	too small $\rightarrow$ rescale
$98.6 \leq x_5 \leq 105$	too large $\rightarrow$ rescale

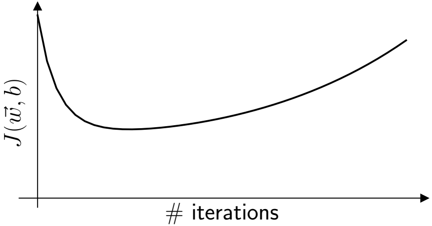
Learning rate: bad learning curve

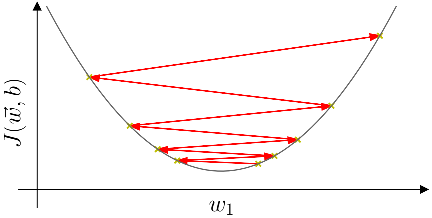




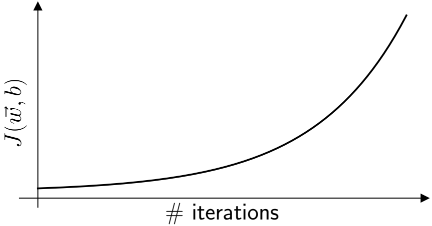
Learning rate: bad learning curve

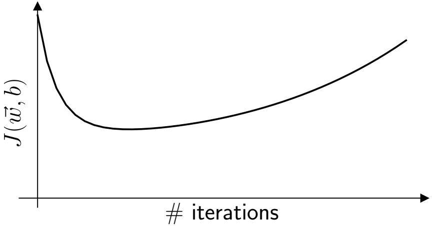






Learning rate: bad learning curve





Potential reasons:

- Bug in the implementation. Example $w_1 = w_1 + \alpha d_1$ ☹ $\Rightarrow w_1 = w_1 - \alpha d_1$ ☺

  Solution: Find the bug, what else can we do?

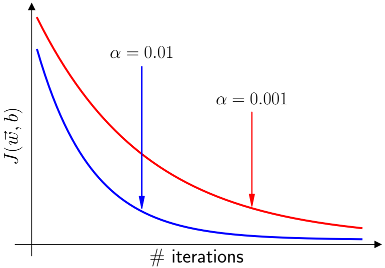
- Learning rate α is too large

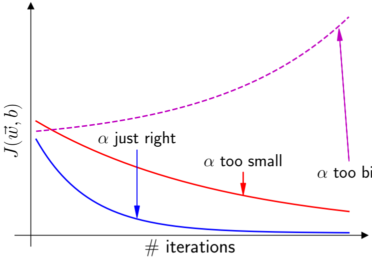
  Solution: Use smaller learning rate α (increase slowly for better convergence rate)

Adjust learning rate

Values of α to try:

$$0.001 \quad 0.003 \quad 0.01 \quad 0.03 \quad 0.1 \quad 0.3 \quad 1$$





- too slow convergence (J decreases slowly) $\rightarrow$ (slightly) increase α
- divergence (J increases) $\rightarrow$ (slightly) decrease α

Feature engineering

$f_{\vec{w},b}(\vec{x}) = w_1 x_1 + w_2 x_2 + b$

x_1 = frontage

x_2 = depth

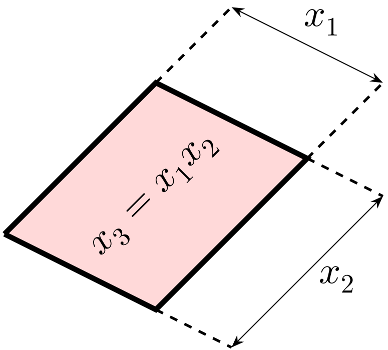
Create new feature

x_3 = area $= x_1 \times x_3$

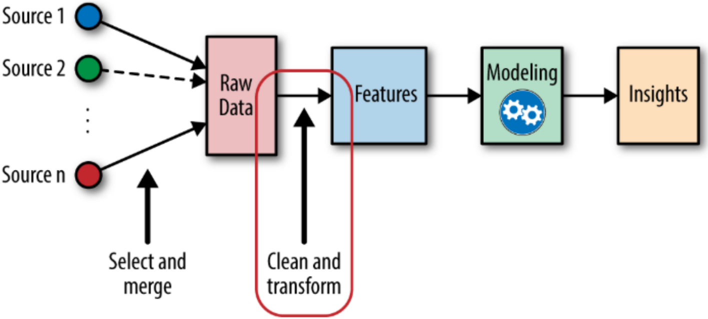
$f_{\vec{w},b}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$

Feature engineering:

Using intuition to design new features, by transforming or combining original features

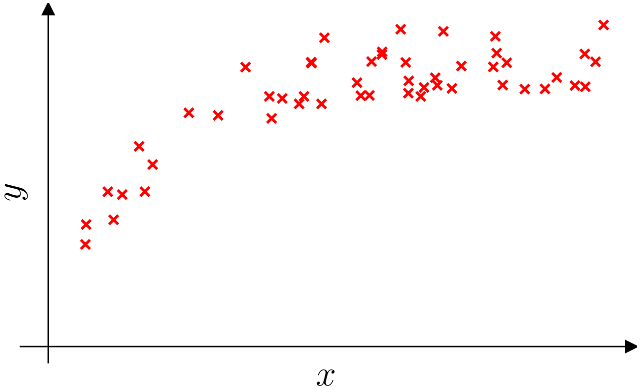


Feature engineering



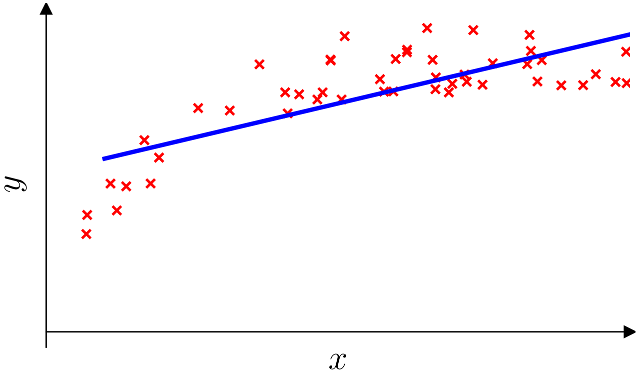
Feature engineering is generally an important step in practical/mega projects.

Polynomial regression



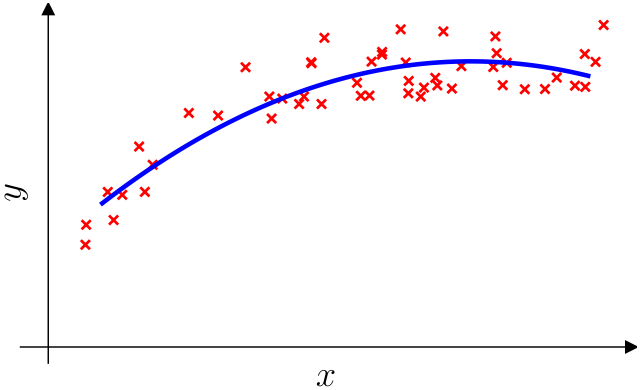
$$f(\vec{w}, x) = \,?$$

Polynomial regression



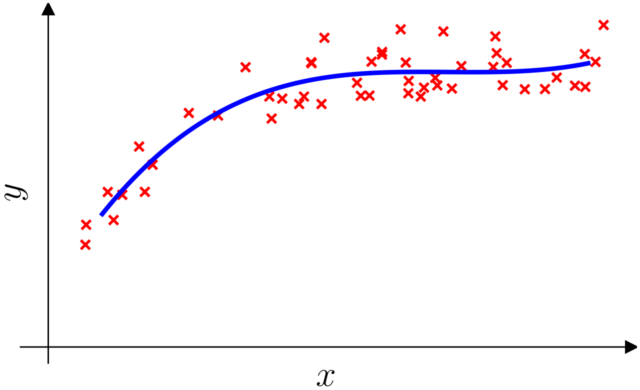
$$f_{w,b}(x) = wx + b$$

Polynomial regression



$$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + b$$

Polynomial regression



$$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + b$$

Mean Squared Error and R squared/Coefficient of Determination

Indicators telling how well fitting model has performed as compared to the test data

Mean Squared Error and R squared/Coefficient of Determination

Indicators telling how well fitting model has performed as compared to the test data

- Mean Squared Error (MSE): *We used it before to construct cost function.*

$$\mathrm{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2$$

Remark: MSE can be defined for just any subset of data examples or a whole data set.

Mean Squared Error and R squared/Coefficient of Determination

Indicators telling how well fitting model has performed as compared to the test data

- Mean Squared Error (MSE): *We used it before to construct cost function.*

$$\mathrm{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2$$

Remark: MSE can be defined for just any subset of data examples or a whole data set.

- Coefficient of Determination / R squared (denoted R^2)

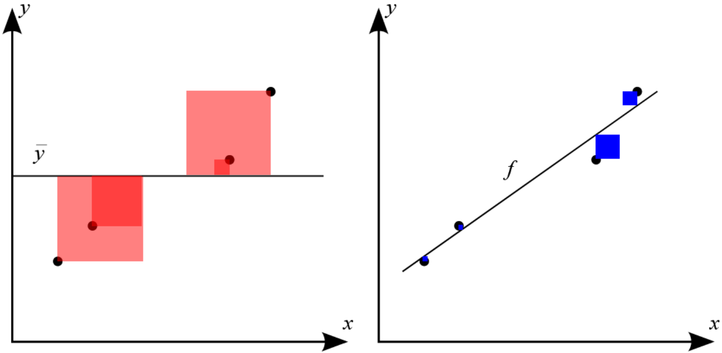
$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}, \qquad \begin{cases} SS_{\text{res}} = \sum_{i=1}(y^{(i)} - \hat{y}^{(i)})^2, \\ SS_{\text{tot}} = \sum_{i=1}(y^{(i)} - \overline{y})^2. \end{cases}$$

with

$$\overline{y} = \mathrm{mean}(y) = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$$

Remark: R squared is not a square of anything. Thus R^2 coefficient is not necessarily non-ngeative; it can be negative

R square/Coefficient of Determination: Intuition



- In the best case, the modeled values exactly match the observed data, which results $SS_{\text{res}} = 0$ and $R^2 = 1$.
- A baseline model, which always predict $\overline{y}$ will have $R^2 = 0$, will have $R^2 = 0$
- Models that have worse prediction than this baseline will have a negative R^2.

Python library `scikitlearn`

- Linear regression can be easily implemented by using the library `scikitlearn`
- `LinearRegression` is the class (a kind of template) doing that job
- `LinearRegression` is imported from `sklearn.linear_model`

Python library scikitlearn

- Linear regression can be easily implemented by using the library scikitlearn
- LinearRegression is the class (a kind of template) doing that job
- LinearRegression is imported from sklearn.linear_model

➥ *A simple procedure*

```
from sklearn.linear_model import LinearRegression
# define an object of class LinearRegression
linear_regr = LinearRegression()     # variable of data type "LinearRegression"
# perform fitting on the training data
linear_regr.fit(X_train, y_train)    # X_train: 2D array, y_train: 1D array
# perform prediction on the test data
linear_regr.predict(X_test)          # X_test: 2D array
# Extract the trainable/learning parameters from the model
w = linear_regr.coef_        # 1D array
b = linear_regr.intercept_   # 1 number/1 scale
```

Quick and dirty 1D example

Assume we already import `numpy` and `matplotlib.pyplot`

Quick and dirty 1D example

Assume we already import `numpy` and `matplotlib.pyplot`

```
w_ref = 3   # reference parameter w
b_ref = 1   # reference parameter b
x_train = np.random.randn(loc=10, scale=1, size=100)    # 1D array
y_train = w_ref * x_train + b_ref                        # 1D array
X_train = x_train.reshape((-1, 1))  # put x_train into 2D array -- column vector here
```

Quick and dirty 1D example

Assume we already import numpy and matplotlib.pyplot

```
w_ref = 3   # reference parameter w
b_ref = 1   # reference parameter b
x_train = np.random.randn(loc=10, scale=1, size=100)    # 1D array
y_train = w_ref * x_train + b_ref                        # 1D array
X_train = x_train.reshape((-1, 1))  # put x_train into 2D array -- column vector here
from sklearn.linear_model import LinearRegression
linear_regr = LinearRegression()
linear_regr.fit(X_train, y_train)
w = linear_regr.coef_[0]    # subscript [0] to get the value in 1D array
b = linear_regr.intercept_
```

Quick and dirty 1D example

Assume we already import numpy and matplotlib.pyplot

```
w_ref = 3    # reference parameter w
b_ref = 1    # reference parameter b
x_train = np.random.randn(loc=10, scale=1, size=100)     # 1D array
y_train = w_ref * x_train + b_ref                        # 1D array
X_train = x_train.reshape((-1, 1))  # put x_train into 2D array -- column vector here
from sklearn.linear_model import LinearRegression
linear_regr = LinearRegression()
linear_regr.fit(X_train, y_train)
w = linear_regr.coef_[0]     # subscript [0] to get the value in 1D array
b = linear_regr.intercept_
xx = np.linspace(5, 15, 101)
yy = w * xx + b
plt.plot(xx, yy, 'k-')       # visualize the linear regressor/linear model
plt.scatter(x_train, y_train, marker='x', s=20, color='r')
# marker with the cross 'x' and of color "red", s: size of the marker
```

MSE and R^2 from sklearn

```python
# We can import MSE, R-squared separately or in the same statement.
from sklearn.metrics import mean_squared_error, r2_score
y_true = [3, -0.5, 2, 7]                    # or using np.array()
y_pred = [2.5, 0.0, 2, 8]                   # or using np.array()

mean_squared_error(y_true, y_pred)  # This gives mean squared error.

# This gives root mean squared error --> root square of the above MSE
mean_squared_error(y_true, y_pred, squared=False)

r2_score(y_true, y_pred)                    # This gives R squared value
```

Z-score normalization/ Standard Scaler from `sklearn`

```
from sklearn.preprocessing import StandardScaler
data = [[0, 0], [0, 0], [1, 1], [1, 1]]
scaler = StandardScaler()   # define an object of class StandardScaler

# compute the mean and std to be used later for scaling
print(scaler.fit(data))
# perform standardization by centering and scaling
scaled_data = scaler.transform(data)

# scaler.fit() and scaler.transform can be combined into one statement.
scaled_data = scaler.fit_transform(data)
# fit_transform() performs fit() first and then transform() -- simple :D
```

Want to learn/use min-max scaling from `sklearn`? $\rightarrow$ Yes, **Google**!

Lazy? Here's the link: Click on me!

PLEASE LEARN HOW TO USE GOOGLE IN THE 21ST CENTURY ☺

Still lazy, Click on me!