

بـه نـام خـدا  
خـلاصـه فـصل  
۴ كـتاب

**HANDS-ON  
MACHINE  
LEARNING**

# TRAINING MODELS

- داخل فصل های قبل صرفاً اومدیم از یه سری مدل استفاده کردیم بدون اینکه بدونیم چطوری پیاده سازی شدن. تو این فصل بیشتر میخوایم بریم سراغ اینکه داخل یه سری مدل دیپ بشیم و ببینیم چطوری پیاده سازی شدن.
- اول بریم سراغ Linear Regression. داخل فصل قبل یه مدل خطی از gdp و میزان خوشحالی مردم دیدیم. به طور کلی این مدل میاد یه جمعی از ضریب وزن دار فیچر ها به علاوه یه ترم bias (عدد ثابت) هستش.

*Equation 4-1. Linear regression model prediction*

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

In this equation:

- $\hat{y}$  is the predicted value.
- $n$  is the number of features.
- $x_i$  is the  $i^{\text{th}}$  feature value.
- $\theta_j$  is the  $j^{\text{th}}$  model parameter, including the bias term  $\theta_0$  and the feature weights  $\theta_1, \theta_2, \dots, \theta_n$ .

# CONTINUE

- میشه فرمول صفحه قبل رو به صورت یک حاصل ضرب دو بردار دید. به علاوه اینکه یه سری تعریف

هایی داریم:

*Equation 4-2. Linear regression model prediction (vectorized form)*

$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

In this equation:

- $h_{\theta}$  is the hypothesis function, using the model parameters  $\boldsymbol{\theta}$ .
- $\boldsymbol{\theta}$  is the model's *parameter vector*, containing the bias term  $\theta_0$  and the feature weights  $\theta_1$  to  $\theta_n$ .
- $\mathbf{x}$  is the instance's *feature vector*, containing  $x_0$  to  $x_n$ , with  $x_0$  always equal to 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$  is the dot product of the vectors  $\boldsymbol{\theta}$  and  $\mathbf{x}$ , which is equal to  $\theta_0x_0 + \theta_1x_1 + \theta_2x_2 + \dots + \theta_nx_n$ .

- حالا که فهمیدیم مدلمون شامل چه فرمول و چه پارامتر هایی هستش دقیقا باید بفهمیم چطوری این پارامتر هارو یاد میگیره؟

# CONTINUE

- اولین کار اینه که یک معیار برای این که بفهمیم چقدر مدلمون خوبه رو تعریف کنیم. همونطور که از فصلای قبل دیدم یه معیار برای اینکه نشون بدھ مدلمون چقدر ارور داره MSE بود. در واقع همین که بیايم ارور مدل رو کم کنیم تعریف دیگه اینه عملکرد مدل رو بهتر کنیم. MSE برا Linearon Regressi این چنین تعریف میشه:

*Equation 4-3. MSE cost function for a linear regression model*

$$\text{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^m (\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)})^2$$

- برای پیدا کردن پارامتر هایی که بیاد تابع MSE رو مینیمم کنه دو تا روش وجود داره:
- ۱. با استفاده از جبرخطی میایم تابع MSE رو مشتق میگیریم بر حسب پارامتر هامون و مساوی صفر میداریم که به چنین فرمولی برای پارامتر ها به طور کلی میرسیم:

*Equation 4-4. Normal equation*

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

In this equation:

- $\hat{\boldsymbol{\theta}}$  is the value of  $\boldsymbol{\theta}$  that minimizes the cost function.
- $\mathbf{y}$  is the vector of target values containing  $y^{(1)}$  to  $y^{(m)}$ .

# TEST NORMAL EQUATION

- بیایم از این فرمول استفاده کنیم. اول یه سری دیتا خطی با یه مقدار نویز میسازیم تا ببینیم چطوری میشه پارامتر های مناسبش رو پیدا کنیم.

```
import numpy as np

np.random.seed(42) # to make this code example reproducible
m = 100 # number of instances
X = 2 * np.random.rand(m, 1) # column vector
y = 4 + 3 * X + np.random.randn(m, 1) # column vector
```

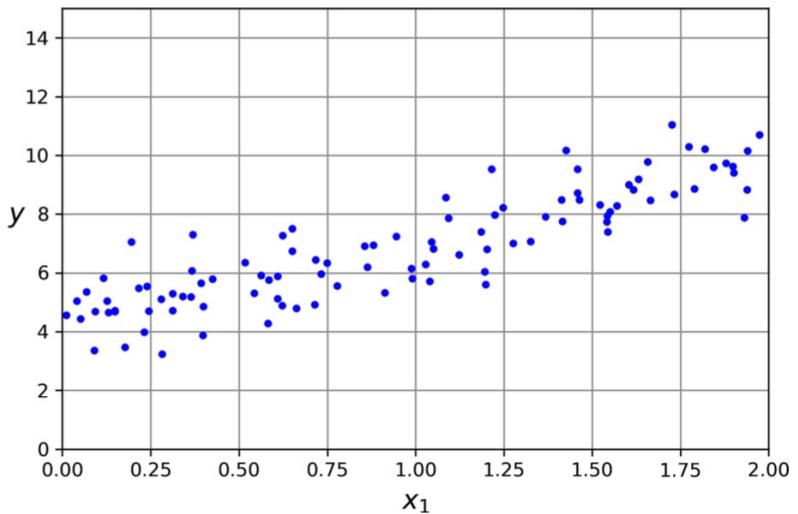


Figure 4-1. A randomly generated linear dataset

حالا با استفاده از `np.linalg` که برای من متد های وارون گرفتن و دات کردن رو پیاده کرده بیایم پارامتر های مورد نظر رو پیدا میکنیم

```
from sklearn.preprocessing import add_dummy_feature

X_b = add_dummy_feature(X) # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```



The `@` operator performs matrix multiplication. If `A` and `B` are NumPy arrays, then `A @ B` is equivalent to `np.matmul(A, B)`. Many other libraries, like TensorFlow, PyTorch, and JAX, support the `@` operator as well. However, you cannot use `@` on pure Python arrays (i.e., lists of lists).

# CONTINUE

- تابعی که از اول ساخته بودیم  $3x+4$  بود بیایم ببینیم که با این کار به چه پارامتر هایی رسیدیم:

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

We would have hoped for  $\theta_0 = 4$  and  $\theta_1 = 3$  instead of  $\theta_0 = 4.215$  and  $\theta_1 = 2.770$ . Close enough, but the noise made it impossible to recover the exact parameters of the original function. The smaller and noisier the dataset, the harder it gets.

Let's plot this model's predictions (Figure 4-2):

```
import matplotlib.pyplot as plt  
  
plt.plot(X_new, y_predict, "r-", label="Predictions")  
plt.plot(X, y, "b.")  
[...] # beautify the figure: add labels, axis, grid, and legend  
plt.show()
```

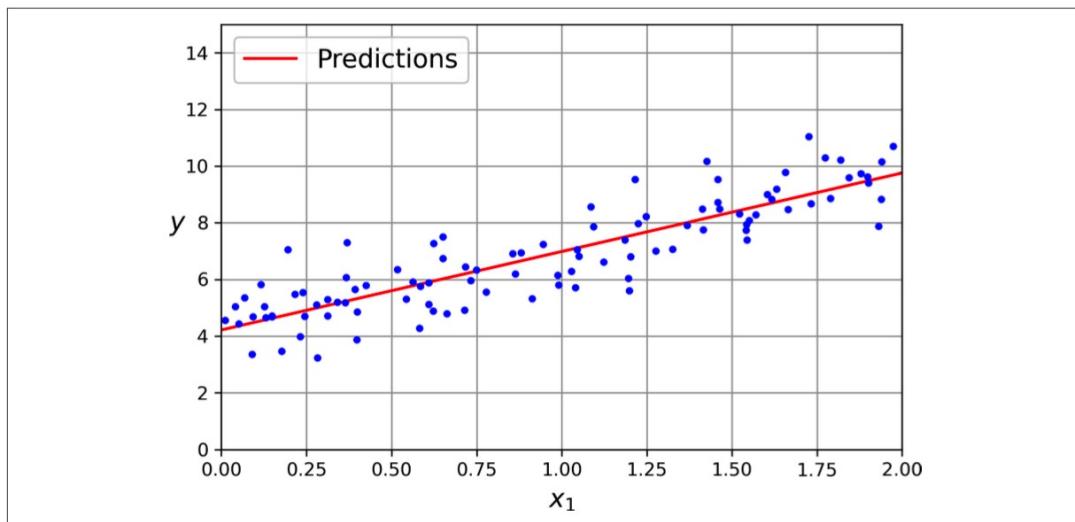


Figure 4-2. Linear regression model predictions

- بیایم حالا خطی که مدل پیدا کرده رو رسم کنیم:

# CONTINUE

- پیاده سازی Linear Regression رو خود scikit-learn داره برای همین صرفا یه مدل ازش میسازیم و دیتا رو بهش پاس میدیم تا پارامتر هارو یاد بگیره.

```
>>> from sklearn.linear_model import LinearRegression  
>>> lin_reg = LinearRegression()  
>>> lin_reg.fit(X, y)  
>>> lin_reg.intercept_, lin_reg.coef_  
(array([4.21509616]), array([[2.77011339]]))
```

- همینطور که میدونیم همه ماتریس ها وارون پذیر نیستن. تو این مورد کاری که انجام میدیم اینه با استفاده از شبهوارون pseudoinverse میایم یه تقریب میزنیم از وارون ماتریسمون و به این شکلی مشکل وارون ناپذیری ماتریس X رو برطرف میکنیم. پایه شبهوارون تجزیه svd هستش که به طور کلی همیشه جواب میده.

This function computes  $\hat{\theta} = X^+y$ , where  $X^+$  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) @ y  
array([[4.21509616],  
       [2.77011339]])
```

# COMPUTATIONAL COMPLEXITY

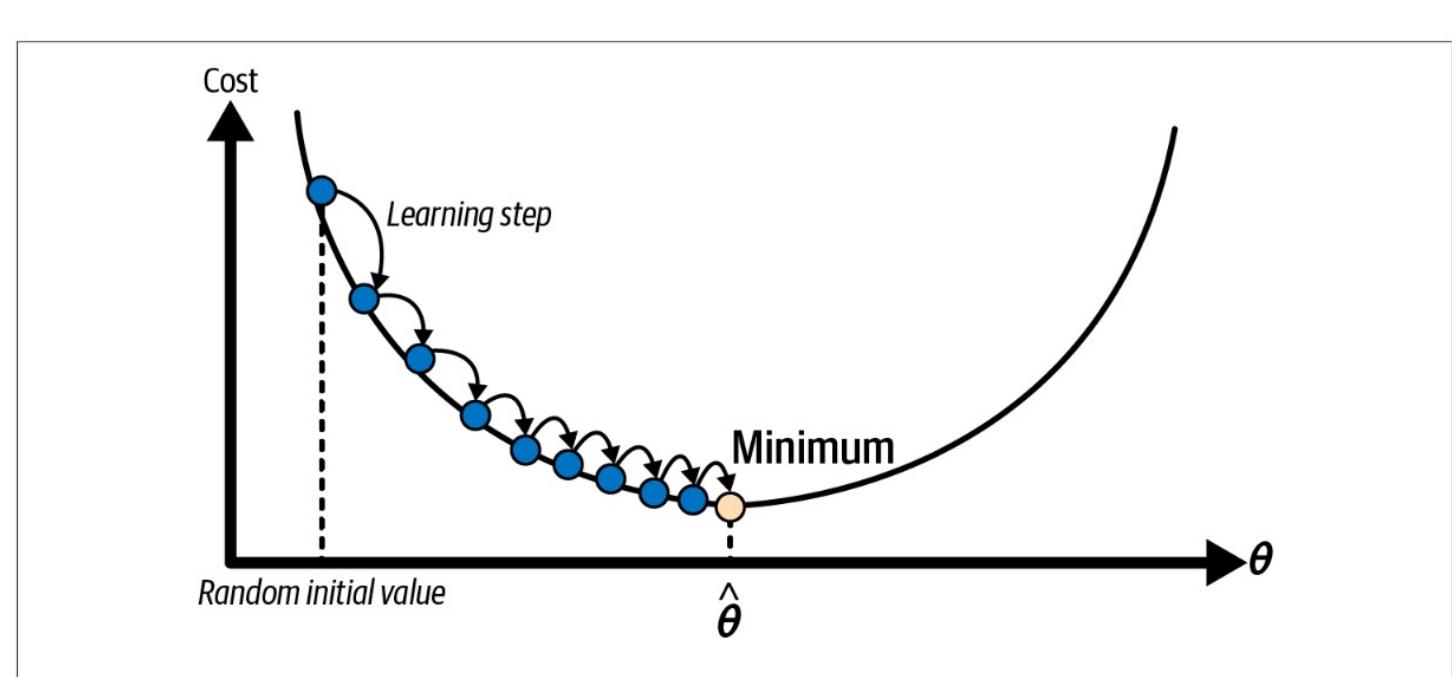
- نکته ای که وجود داره اوردر واورن گرفتن از یه ماتریس از  $O(n^3)$  هستش. که نسبتاً زیاد هستش مخصوصاً وقتی  $n$  زیادی بزرگی داشته باشیم. حتی اگه از svd استفاده کنیم بازم از  $O(n^2)$  هستش.
- این پیچیدگی محاسباتی به تعداد فیچر ها ربط داره. پس اگه تعداد نمونه ها زیاد باشه خیلی هم مشکل ساز نیست برامون. اما یه روشنی نیاز داریم تا اگه فیچر ها زیاد بودن بتونیم از این اوردر بالا رهایی پیدا کنیم.

- این روش به طور کلی یه الگوریتم بهینه سازی هستش که میتونه جواب های خوبی برای مسائل بهینه سازی برامون پیدا کنه. ایده کلی پشت این الگوریتم این هستش که پارامتر هارو پیمایش کنیم تا به یه جواب خوبی برسیم. یه مثال اورده کتاب برای درک بهتر الگوریتم:

Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what gradient descent does: it measures the local gradient of the error function with regard to the parameter vector  $\theta$ , and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

# CONTINUE

- اول میایم پارامتر هامون رو به صورت رندوم مقدار دهی میکنیم. در هر پیمایش داخل الگوریتم سعی میکنیم که به پارامتر هایی دست پیدا کنیم که جواب بهتری هستن برآمون تا زمانی که به یک جواب خوب کلی برسیم.



*Figure 4-3. In this depiction of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the cost approaches the minimum*

# LEARNING RATE

- نکته مهمی که داخل Gradient Descent وجود داره اینه که اندازه قدم هایی که هر سری برمیداریم چقدر باشه؟ باید یه اندازه خوبی باشه منطقا اما چرا مهمه؟ اگه خیلی کوچیک باشه کلی قدم باید برداریم تا برسیم به نقطه مینیمم و اگه خیلی بزرگ باشه ممکنه از نقطه مینیمم عبور کنیم

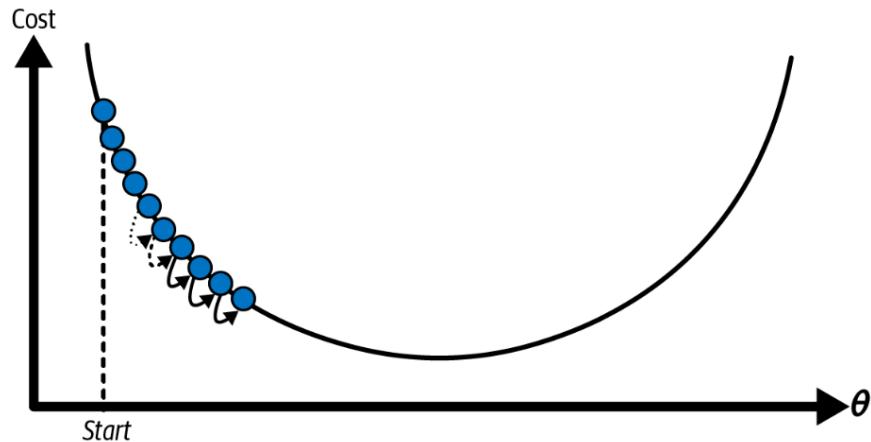


Figure 4-4. Learning rate too small

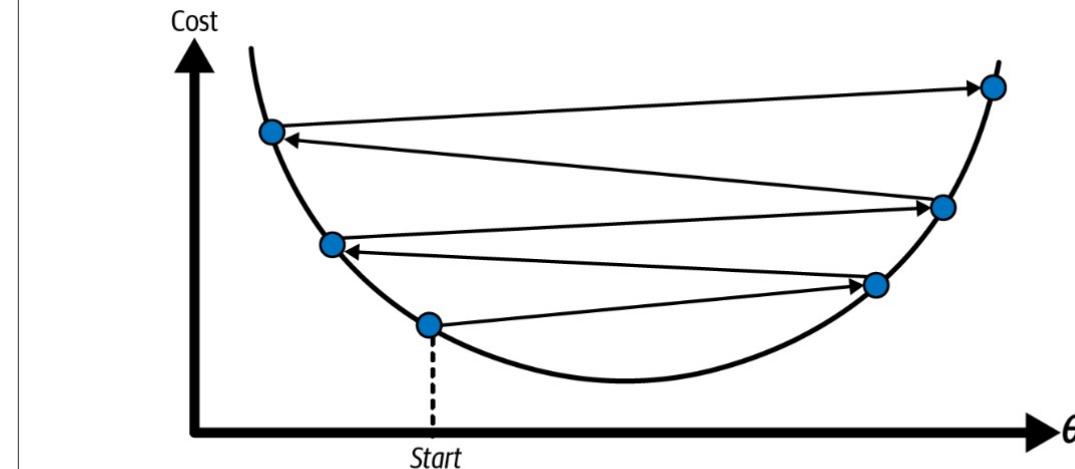


Figure 4-5. Learning rate too high

# CONTINUE

- اما نکته دیگه ای که وجود داره همه loss function ها یه شکل سهمی طور قشنگ ندارن که صرفا یه نقطه مینیمم باشه و همونم جواب ما باشه ممکنه شکل های عجیبی داشته باشه که باعث بشه وقتی به یه نقطه مینیمم برسیم اون نقطه مینیمم کلی ما نباشه و یک سری پارامتر دیگه باشن که اونا جواب بهتری باشن برامون. این حالت خیلی به این که چطوری مقدار دهی اولیه میکنیم پارامتر هامون رو ربط داره.

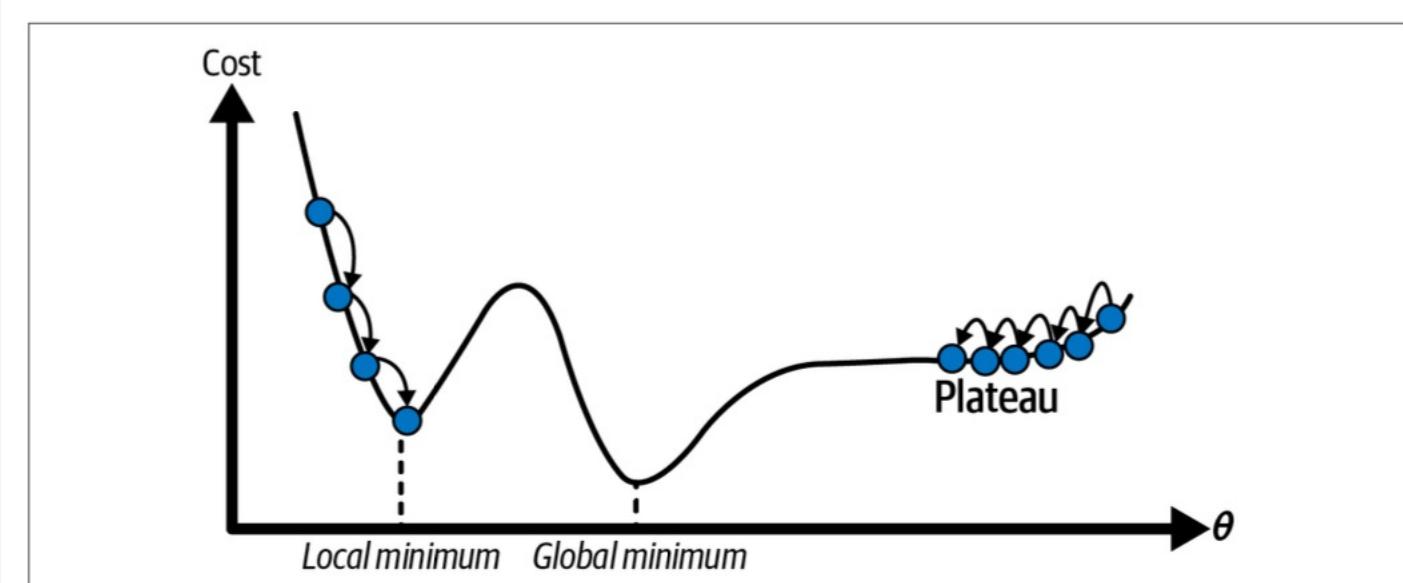


Figure 4-6. Gradient descent pitfalls

# CONTINUE

- صرفا خبر خوب اینه داخل Linear Regression تابع سهمی شکل هستش که مینیمم محلی و کلی یکین. پس خیلی نگران مشکل صفحه قبلی داخل مدل Linear Regression نیستیم.
- با توجه به scale متفاوت عدد های فیچرهامون شکل متفاوت داره تابع MSE روی فضای پارامتر ها، مثل شکل زیر:

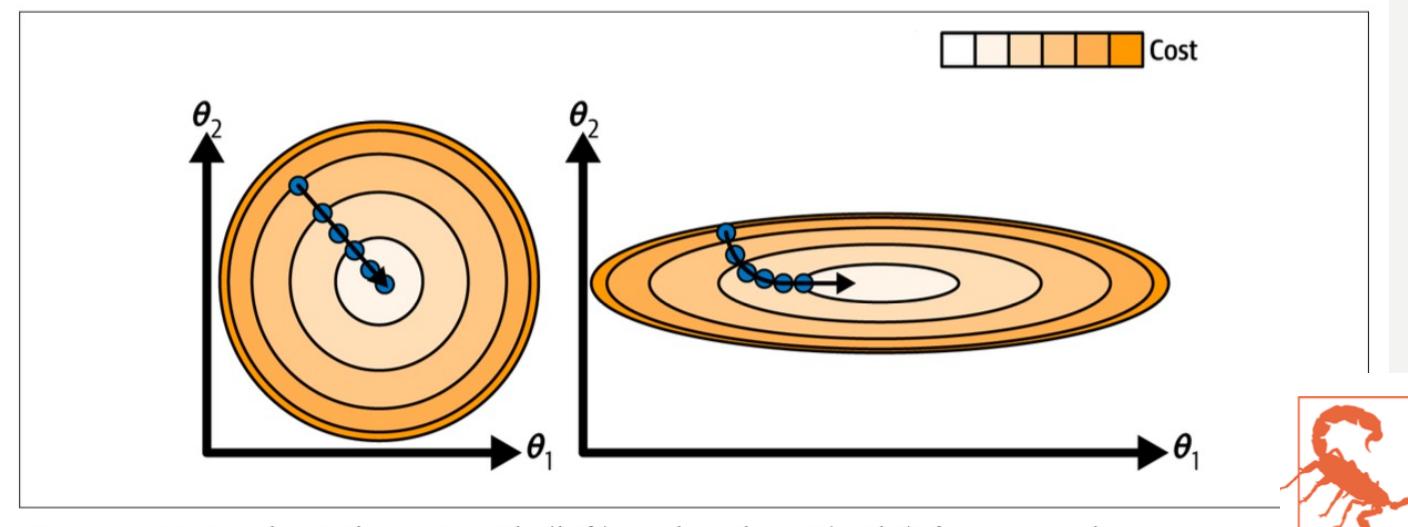


Figure 4-7. Gradient descent with (left) and without (right) feature scaling

- داخل فضای سمت چپ سریعتر به مینیمم کلی میرسیم ولی سمت راستی با قدمای بیشتر. پس خوبه که فیچر هارو قبل شروع فرآیند یادگیری scale به مقادیر یکسان کنیم تا به شکل سمت چپ برسیم.

# CONTINUE

- در واقع کاری که داریم میکنیم داخل فضای فیچر ها دنبال یه ترکیبی از فیچر ها میگردیم تا بتوانیم یه ترکیب خوبی ازشون پیدا کنیم و تابع MSE رو مینیمم کنیم. یه نکته ای که هست صرفا اینجا هم با یه مشکل بالا بودن تعداد فیچر ها به یه شکل دیگه هستیم. وقتی تعداد فیچر ها بره بالا، فضای پارامتریمون از  $2^3$  بعد به  $n$ (تعداد فیچر ها) میرسه و وقتی داریم تو این فضا  $n$  بعدی دنبال یه نقطه بگردیم کار نسبتا سخت تریه تا داخل  $2^3$  بعد دنبال اون جواب خوب بگردیم.
- حالا سراغ روش های مختلف استفاده از Gradient Descent میریم.

# BATCH GRADIENT DESCENT

- برای پیاده سازی Gradient Descent را cost function کاری که باید بکنیم اینه که مشتق تابع بر حسب فیچر های مختلف حساب کنیم.

Equation 4-5 computes the partial derivative of the MSE with regard to parameter  $\theta_j$ , noted  $\frac{\partial \text{MSE}(\boldsymbol{\theta})}{\partial \theta_j}$ .

*Equation 4-5. Partial derivatives of the cost function*

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^m (\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

- به جای محاسبه تک به تک این موارد میتوانیم همه رو یه جا داخل یه بردار داشته باشیم که به شکل زیر حساب میشه.

*Equation 4-6. Gradient vector of the cost function*

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

# CONTINUE

- داخل فرمول صفحه قبل باید دقت کنیم که  $X$  در واقع همه داده های داخل training set هست و بخارط همین هم این روش batch gradient descent هستش. پس اگه یه training set با داده های خیلی زیاد داشته باشیم این روش خیلی خوب نیست و سرعت کمی داره. اما خوبیش اینه برخلاف equation خیلی تعداد فیچر روش تاثیر نمیداره و سرعت خوبی برای تعداد فیچر های بالا داره.
- حالا که مشتق(شیب) اون نقطه رو بدست اوردیم برای اینکه به منطقه پایین تری بررسیم باید صرفاً خلاف جهت شیب حرکت کنیم (علامت منفی) و با استفاده از یک پارامتر میزان اندازه قدم رو مشخص کنیم.

## Equation 4-7. Gradient descent step

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

Let's look at a quick implementation of this algorithm:

```
eta = 0.1 # learning rate
n_epochs = 1000
m = len(X_b) # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1) # randomly initialized model parameters

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

That wasn't too hard! Each iteration over the training set is called an *epoch*. Let's look at the resulting *theta*:

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

- پیاده سازی:

# CONTINUE

دیدیم که همون جوابی که بهمون داد رو gradient descent normal equation هم بهش رسید. اما اگه دیگه ای انتخاب میکردیم نتیجه چی میشد؟ عکس های زیر نتایج این آزمایش رو بهمون نشون میده.

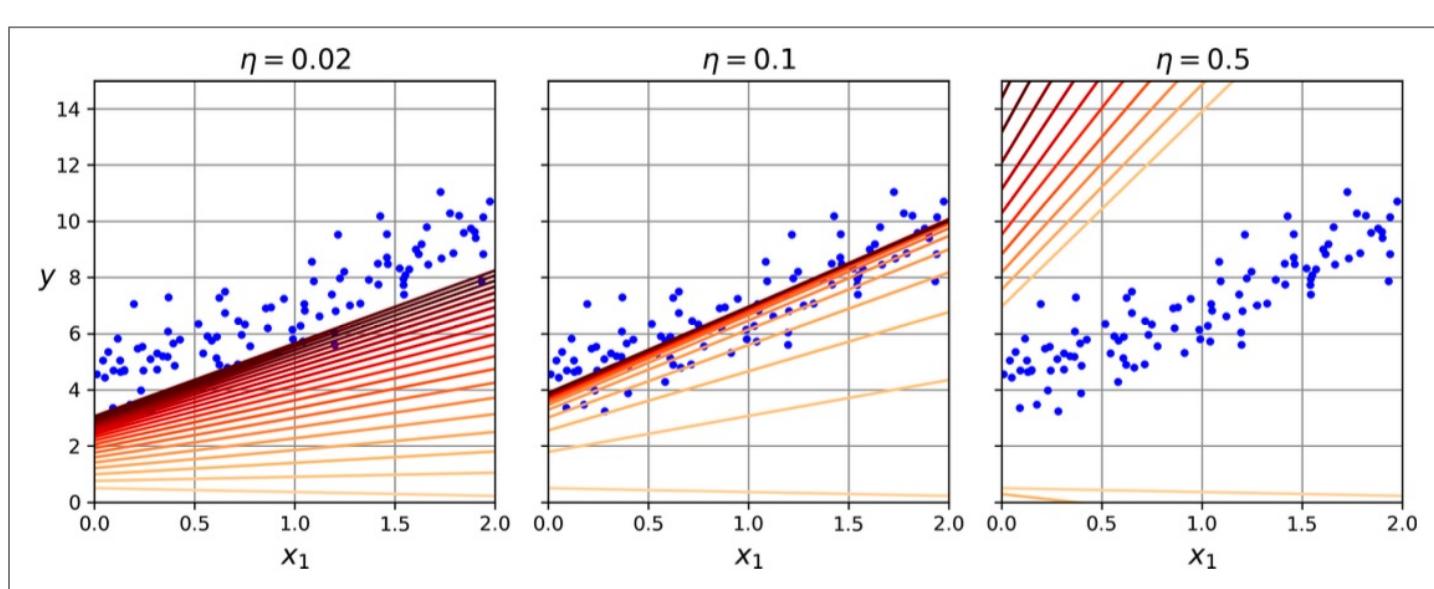


Figure 4-8. Gradient descent with various learning rates

داخل سمت چپ هم به نتیجه رسیدیم ولی چون قدم ها خیلی کوچیک بوده بعد از کلی قدم بهش رسیدیم. وسط حالت مطلوبه و سمت راستی هم بخارط اینکه قدمامون بزرگ بوده از نقطه مینیمم رد شدیم و بهش نرسیدیم. با استفاده از grid search که داخل فصل ای قبل خوندیم میتوانیم مناسب پیدا کنیم.

# CONTINUE

- همین قضیه برای epoch (تعداد iteration با درنظر گرفتن تمامی training set) هم صادقه. اگه خیلی کم باشه ممکنه به جواب مطلوب نرسیم و اگه خیلی زیاد هم باشه ممکنه صرفا هدر دادن وقتمن باشه چون تغییر خاصی بعد از یک تعداد iteration نخواهیم داشت. برای حل این مشکل epoch رو خیلی زیاد میداریم صرفا میگیم هر وقت MSE از یه حدی که کمتر شد break کنه.
- ۲. stocahstice Gradient Descent: مهمترین مشکل روش قبلی این بود که باید مشتق رو داخل هر epoch دوباره حساب میکردیم و این خیلی زمان بر هست وقتی training set خیلی بزرگی داشته باشیم. برخلاف روش قبل صرفا یک دیتا رو به طور رندوم از training set برمیداریم و همون روش قبل رو این سری صرفا روی یه دیتا پیاده میکنیم و انقدر به این کار ادامه میدیم تا به یه جواب خوبی پرسیم. قطعا چون صرفا به یه دیتا تو هر مرحله کار داریم سرعتمن خیلی بیشتره و این جایی به درد میخوره که training set خیلی بزرگی داریم. البته به خاطر ماهیت رندوم بودن و تک دیتا بودن دقت کمتری نسبت به روش قبلی داره.

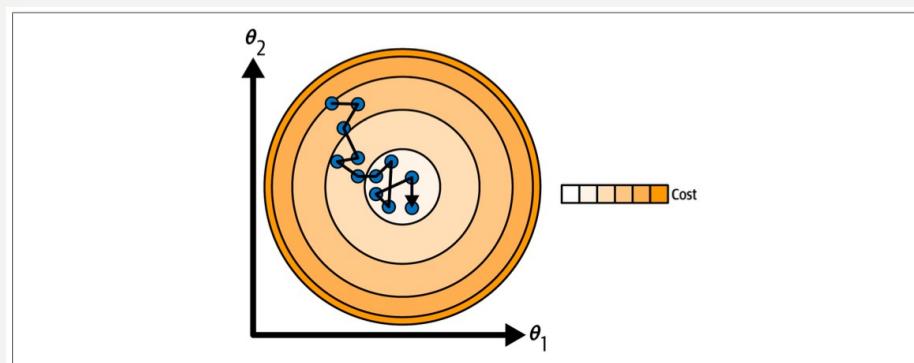


Figure 4-9. With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent

# CONTINUE

- یه خوبی دیگه این روش هم اینه اگه با تابع های پیچیده سر و کار داشته باشیم توانایی اینو داره از اون نقطه مینیمم فرار کنه. به طور کلی بهترین روش اینه با یه learning rate بزرگ شروع کنیم و به تدریج کم و کم ترش کنیم تا قدم هامون تو انتهای فرآیند یادگیری کوچیک تر بشه. پیاده سازی:

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

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

np.random.seed(42)
theta = np.random.randn(2, 1) # random initialization

for epoch in range(n_epochs):
    for iteration in range(m):
        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 @ (xi @ theta - yi) # for SGD, do not divide by m
        eta = learning_schedule(epoch * m + iteration)
        theta = theta - eta * gradients
```

نتیجه ای که بهش رسیده رو  
میتونیم ببینیم چه پارامترهایی رو  
پیدا کردः:

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

# CONTINUE & MINI-BATCH GRADIENT DESCENT

- داخل stochastic خیلی مهمه که دیتاهای را رندوم برداریم چون اگه این کارو نکنیم و یه سری دیتای خاص رو برداریم صرفا یه مدل خوب روی اون دیتاهای داریم.
- تمامی این کارهایی که تا الان کردیم و خودمون پیاده کردیم scikit-learn خودش پیاده کرده و میتونیم به راحتی به شکل زیر استفاده کنیم.

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None, eta0=0.01,
                      n_iter_no_change=100, random_state=42)
sgd_reg.fit(X, y.ravel()) # y.ravel() because fit() expects 1D targets
```

- ۳. اخرين نوع از الگوريتم gradient descent. به طور کلي ايده پشت اين الگوريتم اينطوری هستش که از دو الگوريتم قبلی استفاده ميکنیم به اين شکلی که به طور رندوم يه تعداد نمونه برمیداریم و با توجه به اين حجم نمونه ميایم روش های قبلی رو ادامه ميديم. هم به طور رندوم يه تعداد برمیداریم هم ديگه متکی فقط به یه ديتا نيسیم.

# CONTINUE

- یه میکسی از مزیت و معایب دو نوع دیگه هم هستش. به طور کلی نزدیک تر به مینیمم کلی میشه در نهایت ولی براش سخت تره تا از local minimum فرار کنه. به طور خلاصه همچین عملکردی دارن:

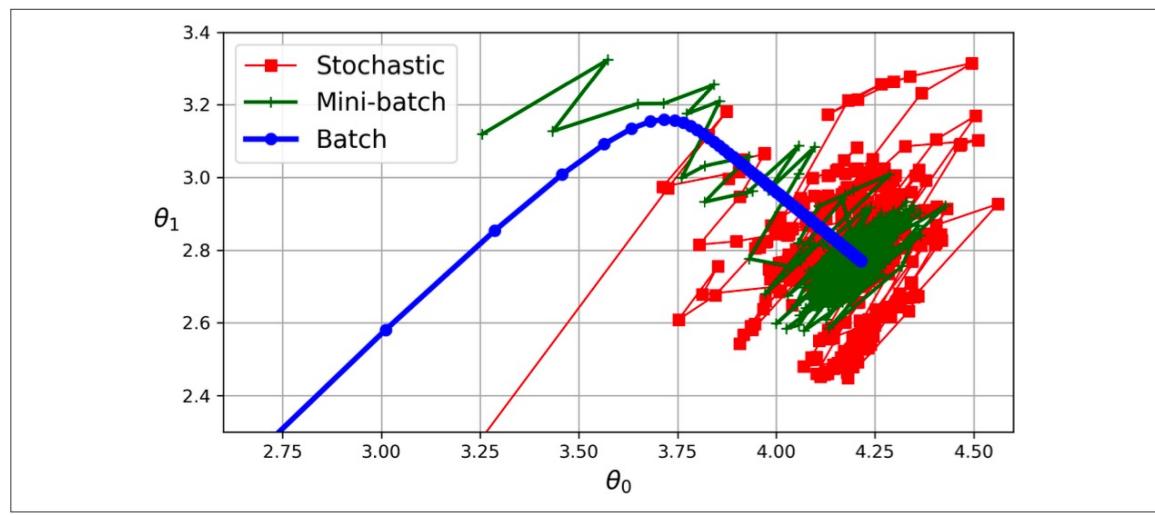


Figure 4-11. Gradient descent paths in parameter space

Table 4-1. Comparison of algorithms for linear regression

Algorithm	Large $m$	Out-of-core support	Large $n$	Hyperparams	Scaling required	Scikit-Learn
Normal equation	Fast	No		Slow	0	No
SVD	Fast	No		Slow	0	No
Batch GD	Slow	No		Fast	2	Yes
Stochastic GD	Fast	Yes		Fast	$\geq 2$	Yes
Mini-batch GD	Fast	Yes		Fast	$\geq 2$	N/A

# POLYNOMIAL REGRESSION

- اما اگه دیتامون چند جمله‌ای باشه و صرفا خطی نباشه چی کار کنیم؟ به طرز جالبی با مدل‌های خطی میشه مدل‌های غیرخطی رو تقریب زد. چطور؟ توان‌های مختلفی از فیچر‌ها رو به لیست فیچر‌ها اضافه میکنیم و linear regression میزنیم. به این مدل polynomial regression میگن. اول ببایم

based on a simple *quadratic equation*—that's an equation of the form  $y = ax^2 + bx + c$ —plus some noise:

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```

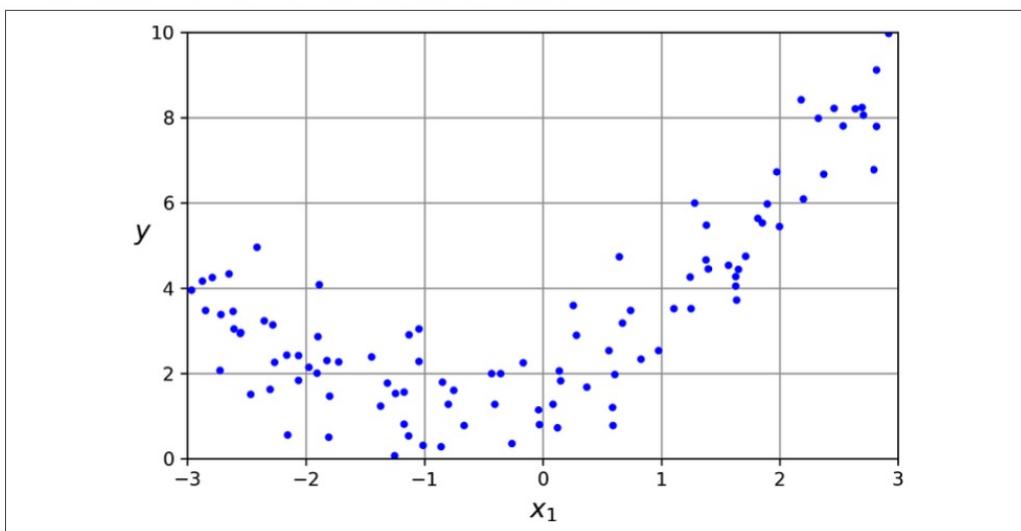


Figure 4-12. Generated nonlinear and noisy dataset

یه سری دیتایی از درجه ۲ بسازیم:

همچین دیتایی به طور واضح با یک مدل خطی به طور خوبی تقریب نمیشه زد. و یک تابع درجه ۲ هست پس تمام توان دو فیچر‌های مختلف رو به فیچر هامون اضافه میکنیم:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

$X_{\text{poly}}$  now contains the original feature of  $X$  plus the square of this feature. Now we can fit a LinearRegression model to this extended training data (Figure 4-13):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

# CONTINUE

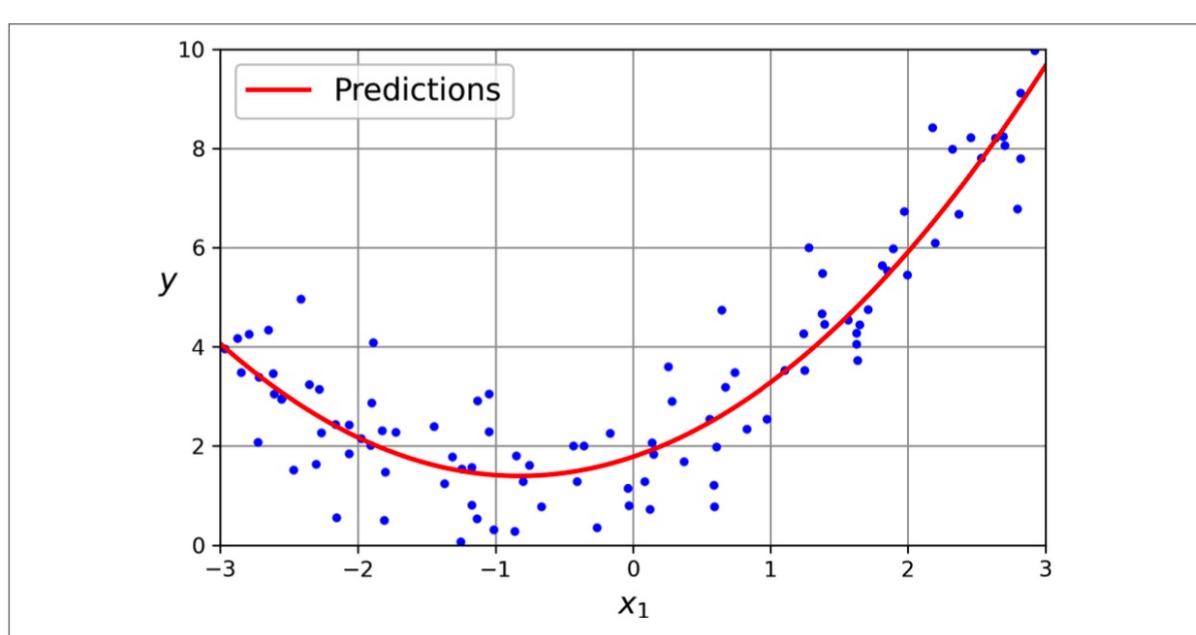


Figure 4-13. Polynomial regression model predictions

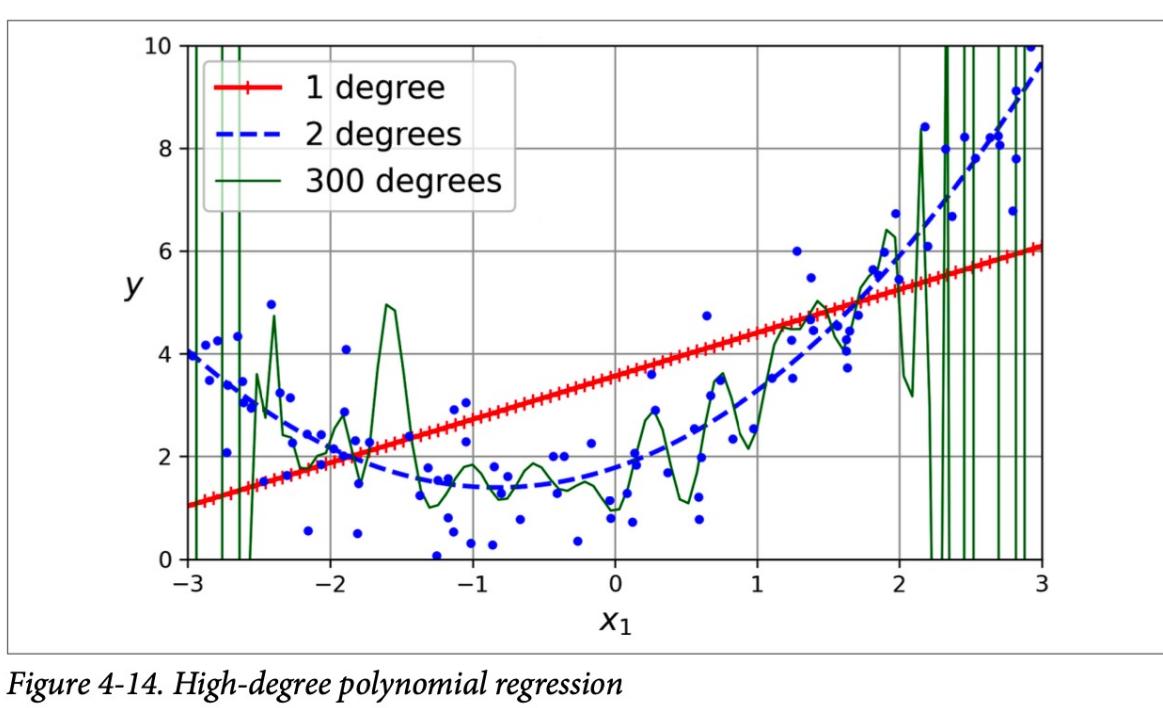
Not bad: the model estimates  $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$  when in fact the original function was  $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$ .

- ببینیم خروجی مدل‌مون چه شکلی شده:

شاید جالب باشه بدونید که با این کار حتی میتونیم ارتباط بین فیچر هارو پیدا کنیم. چرا؟  
چون وقتی  $\text{degree}=3$  میذاریم فقط توان ۳ فیچر های مختلف ساخته نمیشن بلکه مثلًا  $a^{**2}*b$  هم ساخته میشه.

# LEARNING CURVES

- بیايم همون تابع قبلی رو با درجه ۳۰۰ تقریب بزنیم ببینیم چی میشه:



میبینیم که مدل با درجه ۳۰۰ Overfit کرده مدل خطی هم underfit رخ داده براش. منطقی هم هست از اول این دیتا ها رو با یه تابع درجه ۲ ساختیم اما چطوری متوجه بشیم یه تابع ناشناخته براش چه درجه ای به کار ببریم؟ یه روش که همون رو شیوه فصل قبل گفتیم پرفورمنس مدل رو بیايم روی training set و validation set حساب کنیم

- اما یه روش دیگه اینه که از learning curves استفاده کنیم. هدف اصلی اینه که ارور های گفته شده رو بر حسب plot, training iteration کنیم.

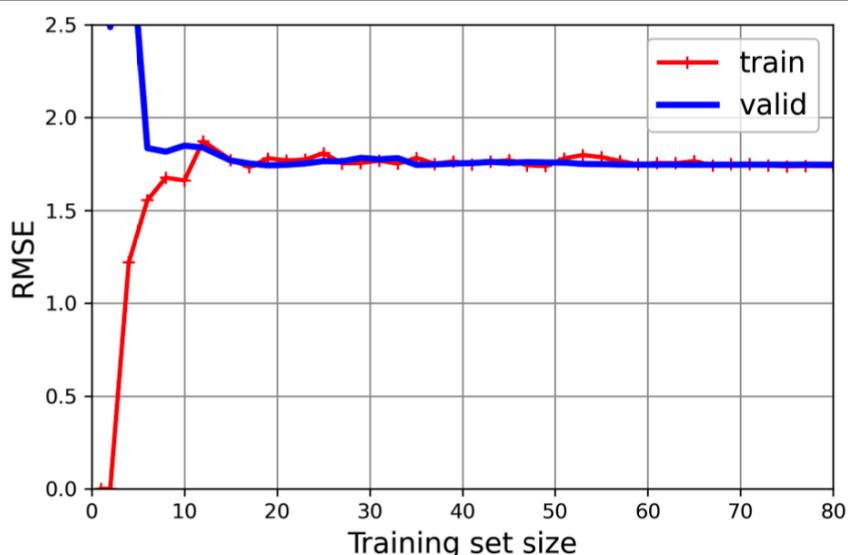
# CONTINUE

- به لطف learning\_curve یه تابع scikit-learn داره که میاد همین کارو برامون با استفاده از cross-validation انجام میده. پیاده سازی:

```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, valid_scores = learning_curve(
    LinearRegression(), X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
train_errors = -train_scores.mean(axis=1)
valid_errors = -valid_scores.mean(axis=1)

plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="train")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="valid")
[...] # beautify the figure: add labels, axis, grid, and legend
plt.show()
```



بعد از پلات کردن، مهم تحلیل پلاتیه که بدست اوردیم. مدل به وضوح underfit شده، چون نرخ ارور به مرور برای training set وقتی سایزش زیاد میشه، بیشتر میشه. که نشون میده نمیتونه روی training set پرفورمنس خوبی validation set باشه. همین قضیه برای خطا زیادی داره.

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

# CONTINUE

- همین حرکت رو برای یه مدل درجه ۱۰ بزنیم ببینیم چی میشه:

Now let's look at the learning curves of a 10th-degree polynomial model on the same data (Figure 4-16):

```
from sklearn.pipeline import make_pipeline

polynomial_regression = make_pipeline(
    PolynomialFeatures(degree=10, include_bias=False),
    LinearRegression())

train_sizes, train_scores, valid_scores = learning_curve(
    polynomial_regression, X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
[...] # same as earlier
```

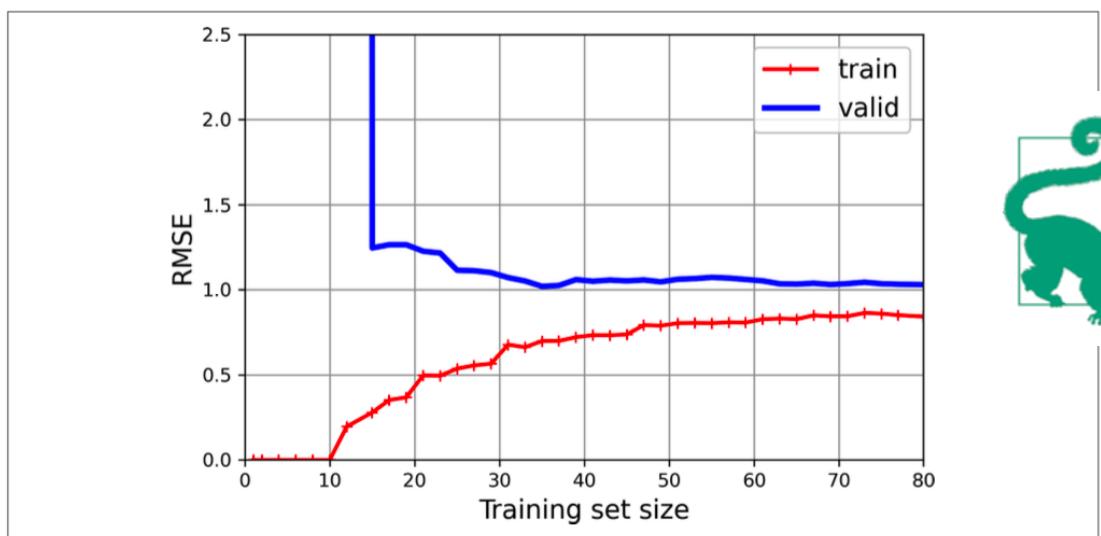


Figure 4-16. Learning curves for the 10th-degree polynomial model

شبيه قبليه شد اما دو تا تفاوت مهم داره يكى اينكه ارور روی training set خيلي کمتر شد و همین که يه فاصله بين ارور هاي دو تا set مون پيدا شده. اما خب اين يعني چي؟ در واقع معنی overfit هميشه. وقتی ارور روی training set کم باشه و ارور روی validation set به همون نسبت کم نسيست.



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

# REGULARIZED LINEAR MODELS

- بعد از اینکه به این برخورد کردیم مدل overfit کرده باید دنبال یه سری روش برای اینکه مدل از بیشتری برخورد دار باشه بگردیم. به این کار regularization میگن. یه راه ساده برای اینکه مدل regularization، polynomial بشه اینه که درجه اون رو کمتر کنیم. برای یه مدل خطی این کار به این شکله که بگیم وزن فیچر ها نمیتونن از یه حدی بزرگتر بشه. حالا به روش های مختلف regularization میپردازیم:

- Ridge Regression. این regularization مدل های linear regression هستش. و به این شکل عمل میکنه یه بخشی رو به MSE اضافه میکنه که باعث میشه مجبور کنه الگوریتم یادگیری رو به اینکه تا جای ممکن ضریب فیچر ها کوچیک باشه.  
a regularization term equal to  $\frac{\alpha}{m} \sum_{i=1}^n \theta_i^2$  is added to the MSE.

صرفا این ترم رو به cost function داخل یادگیری اضافه میکنیم و وقتی میخوایم پرفورمنس مدل رو روی test set حساب کنیم اینو اضافه نمیکنیم. یه hyperparameter الفا داره که باعث میشه کنترل کنیم تا چه حد regularization میخوایم. اگه الفا صفر باشه همون MSE خالی رو داریم و اگه خیلی بزرگ باشه regularization خیلی سنگینی میندازه که باعث میشه صرفا یه خط صاف از وسط میانگین دیتاهای داشته باشیم.

Equation 4-8. Ridge regression cost function

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

# CONTINUE

- حالا اینجا مدل های مختلف که با الگوهای مختلف train شدن رو میبینیم:

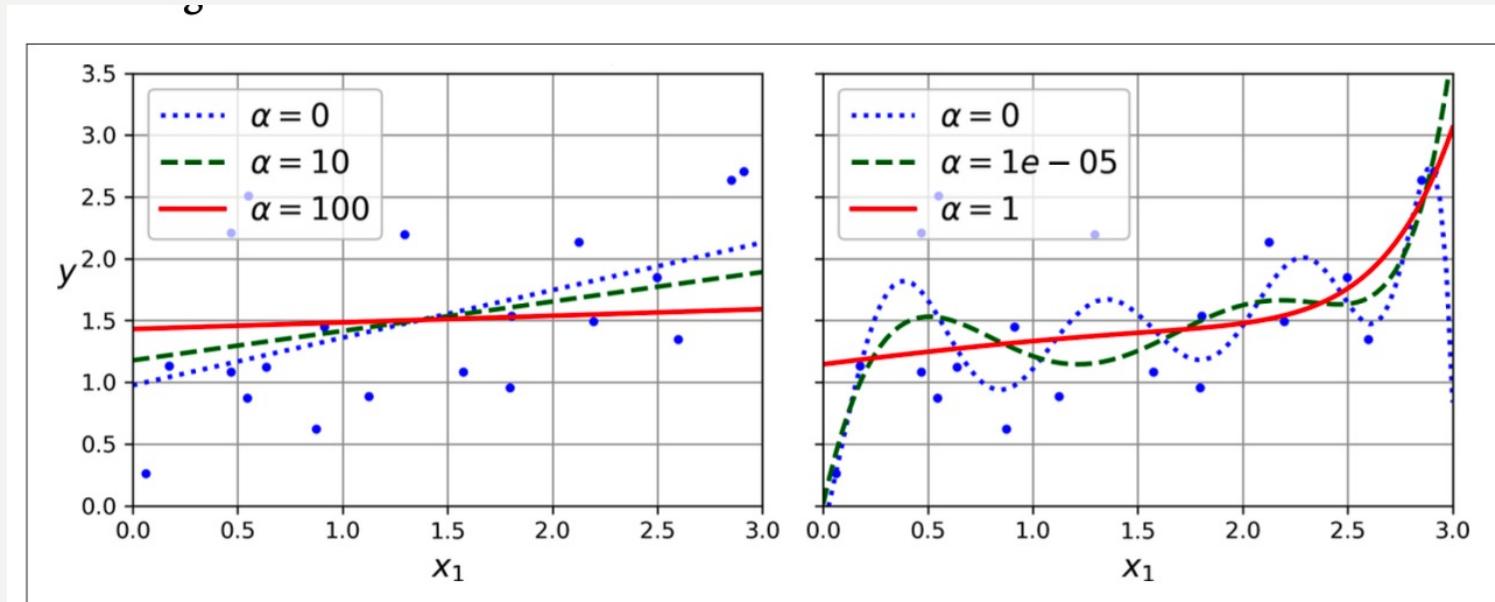


Figure 4-17. Linear (left) and a polynomial (right) models, both with various levels of ridge regularization

- علاوه بر اینکه داخل normal-equation میتوانیم ازش استفاده کنیم داخل Gradient Descent قابلیت رو داریم که فرمولش این شکلیه (A:identity matrix)

Equation 4-9. Ridge regression closed-form solution

$$\hat{\theta} = (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^T \mathbf{y}$$

# IMPLEMENTATION

Here is how to perform ridge regression with Scikit-Learn using a closed-form solution (a variant of [Equation 4-9](#) that uses a matrix factorization technique by André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge  
>>> ridge_reg = Ridge(alpha=0.1, solver="cholesky")  
>>> ridge_reg.fit(X, y)  
>>> ridge_reg.predict([[1.5]])  
array([1.55325833])
```

And using stochastic gradient descent:<sup>10</sup>

```
>>> sgd_reg = SGDRegressor(penalty="l2", alpha=0.1 / m, tol=None,  
...                           max_iter=1000, eta0=0.01, random_state=42)  
...  
>>> sgd_reg.fit(X, y.ravel()) # y.ravel() because fit() expects 1D targets  
>>> sgd_reg.predict([[1.5]])  
array([1.55302613])
```

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 MSE cost function equal to `alpha` times the square of the  $\ell_2$  norm of the weight vector. This is just like ridge regression, except there's no division by  $m$  in this case; that's why we passed `alpha=0.1 / m`, to get the same result as `Ridge(alpha=0.1)`.

# LASSO REGRESSION

- اینم مثل قبلی یه متدهای برای Linear regression هستش و میاد یه ترم رو به cost function اضافه میکنه. اینجا به جای توان ۲ پارامترها، اندازه پارامترهارو با هم جمع میکنیم و دیگه وابستگی به  $m$  نداریم بخلاف قبلی، همه چی بر اساس الfa تعیین میشه.

Equation 4-10. Lasso regression cost function

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

- نتایج ازمایش های مختلف:

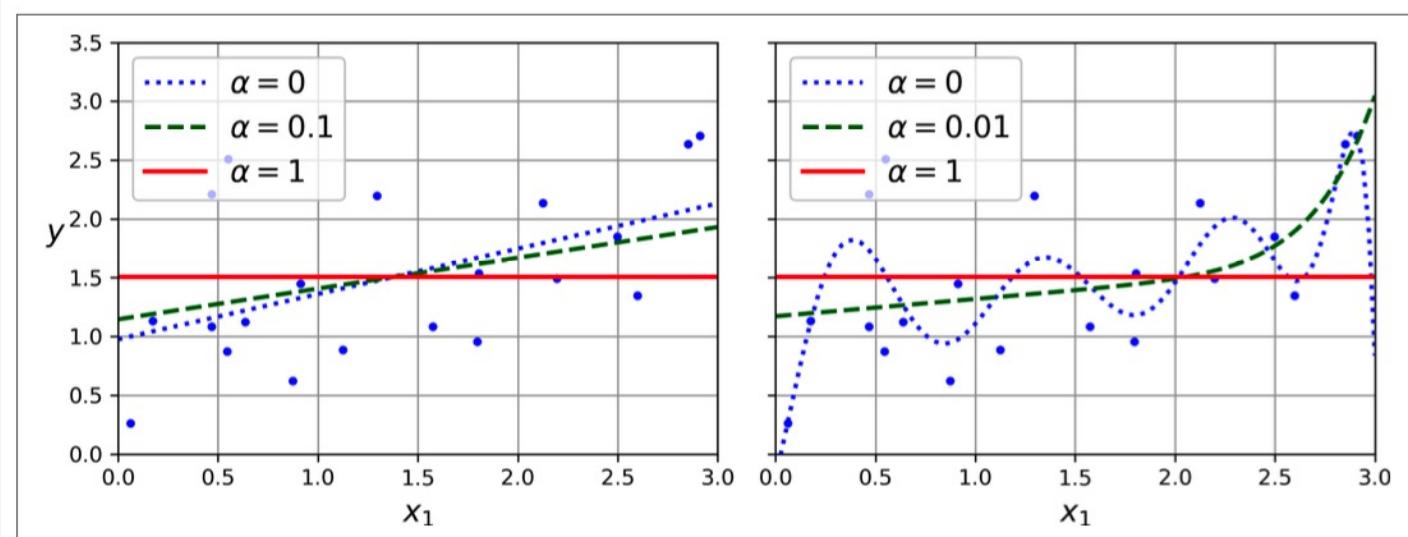


Figure 4-18. Linear (left) and polynomial (right) models, both using various levels of lasso regularization

# LASSO VS RIDGE REGULARIZATION

بهترین نقطه داخل این فضا (جواب مدنظر) پرزنگ ترین نقطس. حالا میتوانیم بررسی کنیم این دو تا روش رو. L1 و L2 صرفا اندازه وزن ها و توان ۲ وزن ها هستن که به ترتیب تا برسن به  ${}^{\circ}$  و  ${}^{\circ}$  مینیمم میشن وقتی ridge و lasso رو مقایسه میکنیم به این فرم میرسیم:

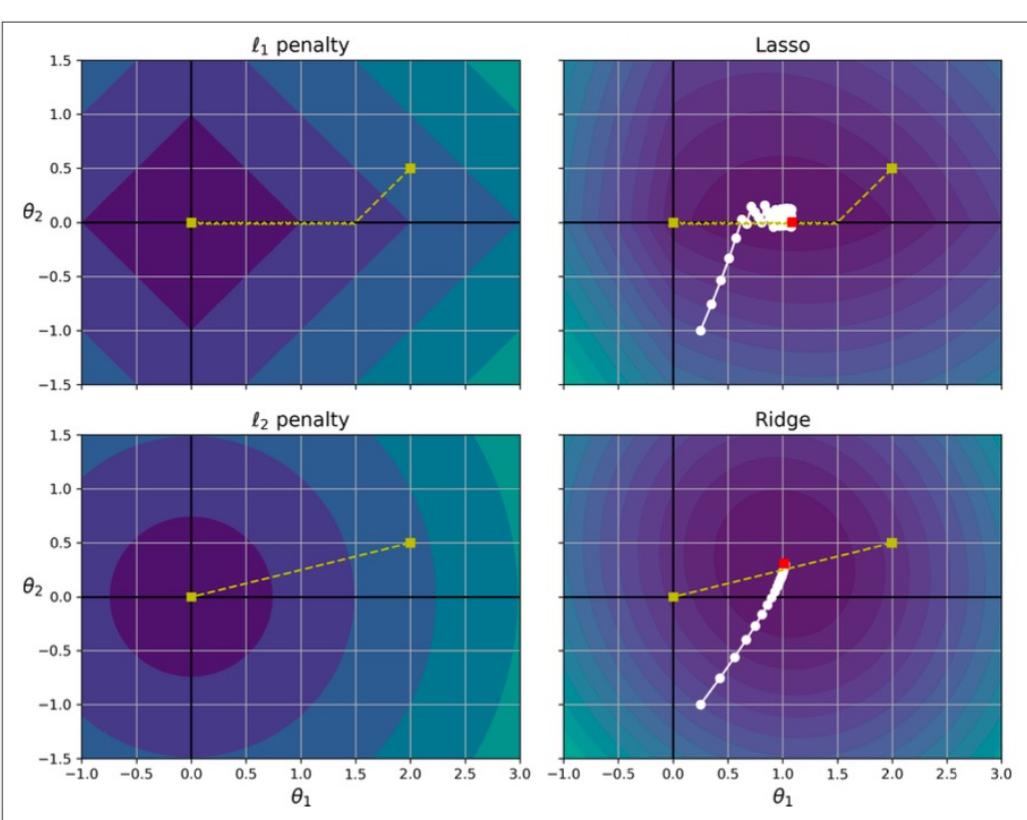


Figure 4-19. Lasso versus ridge regularization

The two bottom plots show the same thing but with an  $\ell_2$  penalty instead. In the bottom-left plot, you can see that the  $\ell_2$  loss decreases as we get closer to the origin, so gradient descent just takes a straight path toward that point. In the bottom-right plot, the contours represent ridge regression's cost function (i.e., an MSE cost function plus an  $\ell_2$  loss). As you can see, the gradients get smaller as the parameters approach the global optimum, so gradient descent naturally slows down. This limits the bouncing around, which helps ridge converge faster than lasso regression. Also note that the optimal parameters (represented by the red square) get closer and closer to the origin when you increase  $\alpha$ , but they never get eliminated entirely.

The lasso cost function is not differentiable at  $\theta_i = 0$  (for  $i = 1, 2, \dots, n$ ), but gradient descent still works if you use a *subgradient vector*  $\mathbf{g}^{11}$  instead when any  $\theta_i = 0$ . [Equation 4-11](#) shows a subgradient vector equation you can use for gradient descent with the lasso cost function.

[Equation 4-11. Lasso regression subgradient vector](#)

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) + 2\alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{where } \text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Here is a small Scikit-Learn example using the Lasso class:

```
>>> from sklearn.linear_model import Lasso  
>>> lasso_reg = Lasso(alpha=0.1)  
>>> lasso_reg.fit(X, y)  
>>> lasso_reg.predict([[1.5]])  
array([1.53788174])
```

Note that you could instead use `SGDRegressor(penalty="l1", alpha=0.1)`.

# ELASTIC NET REGRESSION

- این هم دقیقا یه چیزی بین دو تا روش قبلی هستش، با یه  $r$  hyperparameter میاد کنترل میکنه چقدر از هرکدام تاثیر گذار باشه.

*Equation 4-12. Elastic net cost function*

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

- در نهایت سوالی که ایجاد میشه کی از کدوم استفاده کنیم؟ یه مقدار regularization همه مدل ها نیاز دارن. به طور کلی Ridge اوکیه اما اگه احساس بکنیم یه سری فیچر اضافن میتونیم از lasso استفاده کنیم تا ضریب اون فیچر هارو مساوی صفر بذاره. اگه هم جفتشو بخوایم elastic میزنیم.

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

# EARLY STOPPING

- یه روش دیگه که میتونیم regularization رو انجام بدیم، early stopping. و ایده کلیش اینه که هر وقت دیدیم ارور زیاد میشه همونجا واپسیم. فلسفه پشتیش اینه با پیماش هایی که داخل هر epoch داریم به مرور مدل داره دیتاهای training set رو باز میبینه و از یه جا به بعد داره سعی میکنه خودش رو هی بهتر و بهتر کنه فقط برای training set پس داره overfitting رخ نمیده.

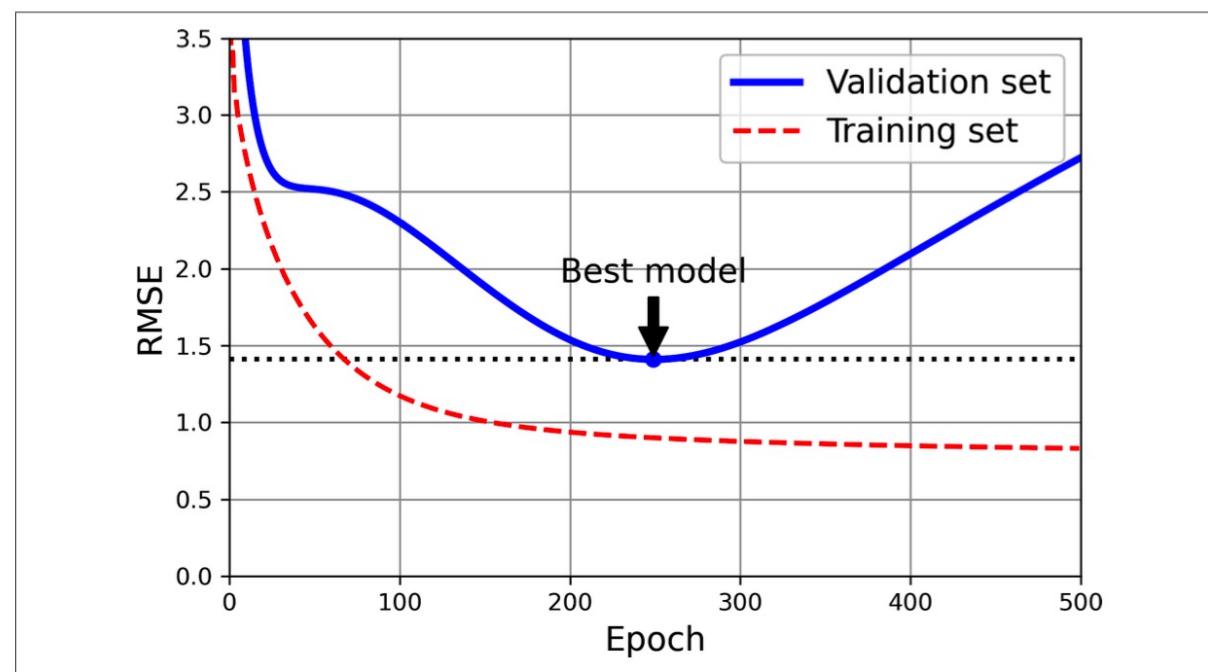


Figure 4-20. Early stopping regularization

Here is a basic implementation of early stopping:

```
from copy import deepcopy
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

X_train, y_train, X_valid, y_valid = [...] # split the quadratic dataset

preprocessing = make_pipeline(PolynomialFeatures(degree=90, include_bias=False),
                             StandardScaler())
X_train_prep = preprocessing.fit_transform(X_train)
X_valid_prep = preprocessing.transform(X_valid)
sgd_reg = SGDRegressor(penalty=None, eta0=0.002, random_state=42)
n_epochs = 500
best_valid_rmse = float('inf')

for epoch in range(n_epochs):
    sgd_reg.partial_fit(X_train_prep, y_train)
    y_valid_predict = sgd_reg.predict(X_valid_prep)
    val_error = mean_squared_error(y_valid, y_valid_predict, squared=False)
    if val_error < best_valid_rmse:
        best_valid_rmse = val_error
        best_model = deepcopy(sgd_reg)
```



With stochastic and mini-batch gradient descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not. One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum.

# LOGISTIC REGRESSION

- با استفاده از این مدل دیدیم میشه مسائل classification رو انجام داد. ایدش اینه بیاد احتمال حساب کنه و اگه بیشتر از ۵۰ درصد (معمولا) بود بگه متعلق به کلاسمون هست اگه پایینتر باشه بگه نیست.
- اما دقیقا چه شکلی کار میکنه؟ اینم مثل Linear Regression جمع ضریب وزن دار فیچر هارو حساب میکنه اما اون عدد رو به جای خروجی دادن به تابع sigmoid میده و خروجی اون (که بین ۰ و ۱ هست) رو خروجی میده. اگه خروجی که تابع sigmoid میده بهمون مثلا بیشتر از ۰.۵ باشه میگیم متعلق هست و بر عکس.

Equation 4-13. Logistic regression model estimated probability (vectorized form)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \mathbf{x})$$

Equation 4-14. Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

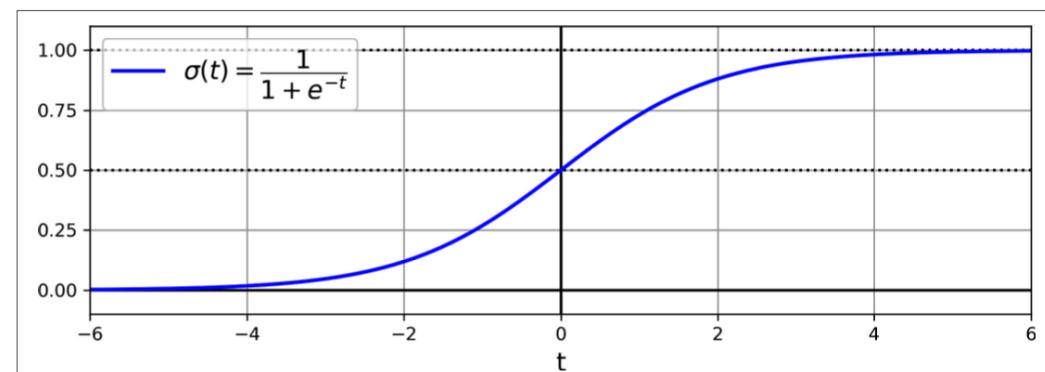


Figure 4-21. Logistic function

# TRAINING AND COST FUNCTION

- نحوه خروجی دادنش رو فهمیدیم اما دقیقاً چطوری پارامتر هارو یاد میگیره؟ هدف اینه پارامتر هارو یه طوری تعیین کنیم که به ازای شی های متعلق به کلاس یک عدد بزرگ خروجی بده و برعکس. همین ایده cost function میشه.

*Equation 4-16. Cost function of a single training instance*

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

This cost function makes sense because  $-\log(t)$  grows very large when  $t$  approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be large if the model estimates a probability close to 1 for a negative instance. On the other hand,  $-\log(t)$  is close to 0 when  $t$  is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want.

*Equation 4-17. Logistic regression cost function (log loss)*

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

# CONTINUE

- تابع صفحه قبلی که به عنوان cost function معرفی شده عشقی ننوشتن، اثبات میشه اگه بیايم این تابع رو مینیمم کنیم به maximum likelihood میرسیم. وقتی از این cost function استفاده میکنیم فرض میکینیم دیتامون از توزیع گوسی پیروی میکنه. در واقع وقتی از MSE هم استفاده میکنیم دیتامون خطیه و از توزیع گوسی پیروی میکنه. اگه دیتامون همچین فرض هایی نداشته باشن و از این cost function استفاده کنیم به یه مدل (underfit) high bias برخورد میکنیم.
- داخل اینجا برخلاف جاهای قبل یه تساوی مستقیم نداریم و مجبوریم از gradient descent استفاده کنیم برای اینکه به global minimum برسیم. پس نیاز به مشتق جزئی داریم.

*Equation 4-18. Logistic cost function partial derivatives*

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m (\sigma(\boldsymbol{\theta}^\top \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

- وقتی که این مشتق رو به ازای تموی فیچر ها محاسبه کنیم میتونیم یه iteration انجام بدیم داخل .gradient descent

# DECISION BOUNDARIES

یه دیتاست معروف به اسم iris داریم. که یه سری گل با عرض و طول گل برگشون رو ذخیره کرد. میتوانیم این شکلی اطلاعات دیتاست رو ببینیم:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> list(iris)
['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
 'filename', 'data_module']
>>> iris.data.head(3)
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0              5.1             3.5            1.4             0.2
1              4.9             3.0            1.4             0.2
2              4.7             3.2            1.3             0.2
>>> iris.target.head(3) # note that the instances are not shuffled
0    0
1    0
2    0
Name: target, dtype: int64
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

حالا میتوانیم یه logistic regression بزنیم برای اینکه یه مدل داشته باشیم برای اینکه ببینیم یه گل متعلق به گل از نوع virginica هست یا نه

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = iris.data[["petal width (cm)"]].values
y = iris.target_names[iris.target] == 'virginica'
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

# CONTINUE

- حالا میتوانیم نحوه عملکرد مدل و اینکه از کجا به بعد رو به عنوان این گل درنظر میگیره برای یک بازه از یکی از فیچر هاش ببینیم:

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1) # reshape to get a column vector
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0, 0]

plt.plot(X_new, y_proba[:, 0], "b--",
          label="Not Iris virginica proba")
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica proba")
plt.plot([decision_boundary, decision_boundary], [0, 1], "k:", linewidth=2,
          label="Decision boundary")
[...] # beautify the figure: add grid, labels, axis, legend, arrows, and samples
plt.show()
```

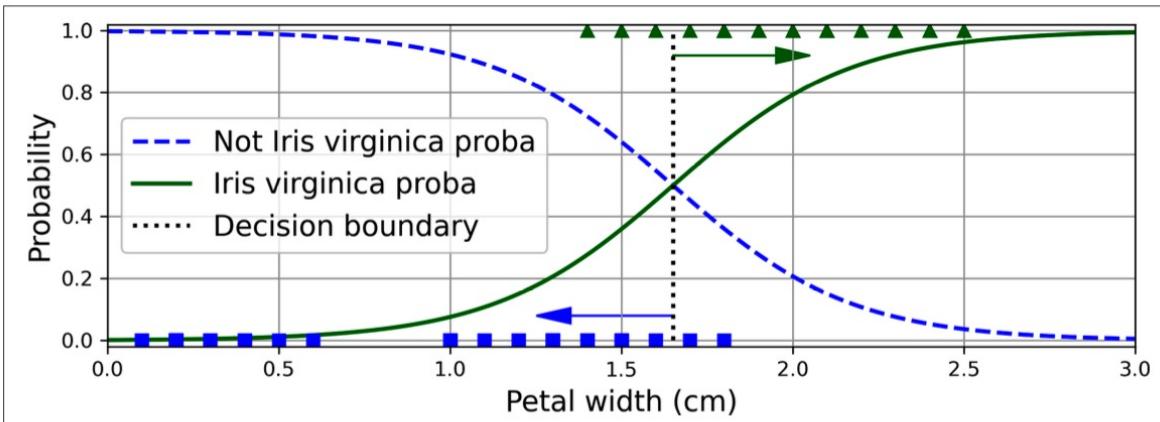


Figure 4-23. Estimated probabilities and decision boundary

این نوع از گل عرض گلبرگ ۱.۴ به بالا داره و مدل decision boundary رو برای خودش ۱.۸ گذاشته که یکم اورلپ داره بین گل هایی که از این دسته هستند و از این دسته نیستند. اما بالای ۲ سانت چون نمونه دیگه ای از گل های غیر این کلاس نیست با اطمینان بالایی میتوانیم بگیم متعلق به این کلاسه.

# CONTINUE

- در واقع مدل میاد یه خط رسم میکنه و میگه هرچی بالای این باشه متعلق به کلاس هستش و اگه پایینش باشه متعلق بهش نیست و حالا هرچقدر دورتر یا نزدیک تر به این خط بشی درصد احتمالی رو بالا پایین میکنه. داخل اینجا هم خط های مختلف میزان احتمال تعلق رو میگن اما خط دش دقیقا احتمال ۵۰ درصد رو داره.

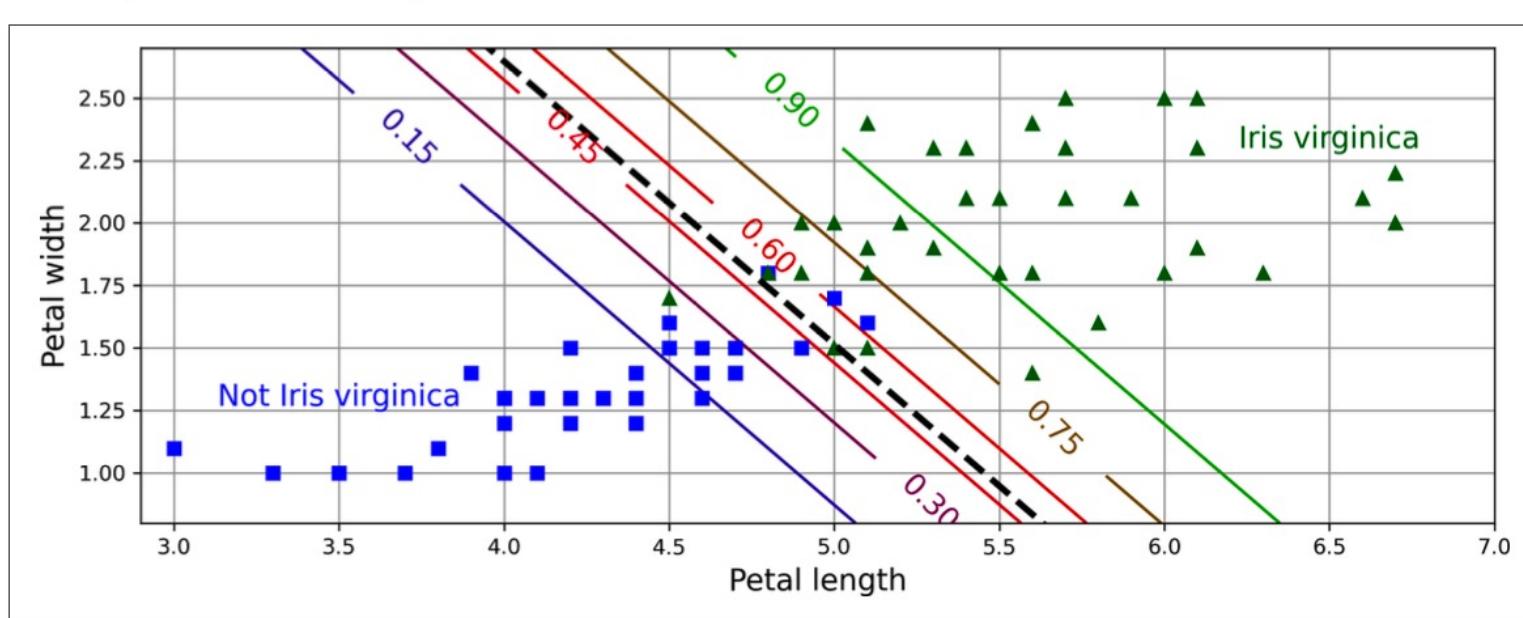


Figure 4-24. Linear decision boundary

# SOFTMAX REGRESSION

- میتوانیم مدل logistic regression را تعمیم بدهیم تا بتوانه چند کلاس مختلف رو به طور مستقیم پوشش بده. در واقع نیاز نیست دیگه برای هر کلاس یه مدل train کنیم با train کردن یه مدل میتوانیم بفهمیم یه نمونه متعلق به کدام کلاسه. ایده پشتیش هم این شکلیه که نمونه که بھش میدن و احتمال اینکه متعلق به اون کلاس باشه رو حساب میکنه و در نهایت هر کدام احتمال بیشتری داشت به عنوان خروجی بهمون میده. میزان عضویت به کلاس  $k$  با این تابع حساب میشه، بعدش نرمال میشه. در نهایت بیشترین احتمال متعلق به یه کلاس به عنوان کلاس مورد نظر خروجی داده میشه

Equation 4-19. Softmax score for class  $k$

$$s_k(\mathbf{x}) = (\boldsymbol{\theta}^{(k)})^\top \mathbf{x}$$

Note that each class has its own dedicated parameter vector  $\boldsymbol{\theta}^{(k)}$ . All these vectors are typically stored as rows in a *parameter matrix*  $\boldsymbol{\Theta}$ .

the softmax function (Equation 4-20). The function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials). The scores are generally called logits or log-odds (although they are actually unnormalized log-odds).

Equation 4-20. Softmax function

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

Equation 4-21. Softmax regression classifier prediction

$$\hat{y} = \operatorname{argmax}_k \sigma(\mathbf{s}(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k ((\boldsymbol{\theta}^{(k)})^\top \mathbf{x})$$

The *argmax* operator returns the value of a variable that maximizes a function. In this equation, it returns the value of  $k$  that maximizes the estimated probability  $\sigma(\mathbf{s}(\mathbf{x}))_k$ .

# COST FUNCTION

- برای این بخش هم داشته باشیم. به cost function برای این بخش میگن.

*Equation 4-22. Cross entropy cost function*

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

In this equation,  $y_k^{(i)}$  is the target probability that the  $i^{\text{th}}$  instance belongs to class  $k$ . In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.

Notice that when there are just two classes ( $K = 2$ ), this cost function is equivalent to the logistic regression cost function (log loss; see [Equation 4-17](#)).

- در واقع به ازای احتمال های بالایی که متعلق به اون کلاس نیستن مقدار بالایی تولید میکنه که باعث زیاد شدن ارور میشه و برعکس که همون چیزیه که از تابع cost function انتظار داریم. بعدش نیاز به مشتق داریم که این شکلی حساب میشه:

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

- حالا میشه متدهای یادگیری مثل gradient descent را به کار برد. میتونیم همین کار رو برای کل دیتابست iris بزنیم تا گل های مختلف رو شناسایی کنه.

# FINISH

```
X = iris.data[["petal length (cm)", "petal width (cm)"]].values  
y = iris["target"]  
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)  
  
softmax_reg = LogisticRegression(C=30, random_state=42)  
softmax_reg.fit(X_train, y_train)
```

So the next time you find an iris with petals that are 5 cm long and 2 cm wide, you can ask your model to tell you what type of iris it is, and it will answer *Iris virginica* (class 2) with 96% probability (or *Iris versicolor* with 4% probability):

```
>>> softmax_reg.predict([[5, 2]])  
array([2])  
>>> softmax_reg.predict_proba([[5, 2]]).round(2)  
array([[0. , 0.04, 0.96]])
```

curved lines (e.g., the line labeled with 0.30 represents the 30% probability boundary). Notice that the model can predict a class that has an estimated probability below 50%. For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.

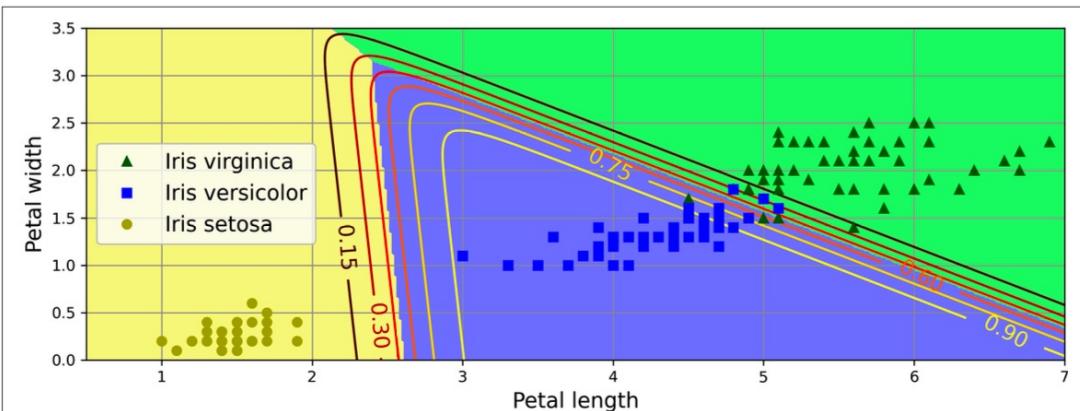


Figure 4-25. Softmax regression decision boundaries