Shanghai Advanced Institute of Finance
上海高级金融学院

量化俱乐部-机器学习-Regression

2019-08-04

Menu

1. Linear Regression
2. Gradient Descent
3. Polynomial Regression
4. Regularized Linear Models
5. Logistic Regression

自我介绍

张骁喆，2010年大连理工大学软件学院本科毕业，高金FMBA 2017PTE。9年计算机工作经验，熟悉开发，测试，运维，项目管理，产品设计各环节。
曾就职eBay，唯品会，2016加入SAP，现担任SAP Cloud部门开发团队主管。
2017开始接触量化投资和python，目前毕业论文研究方向使用机器学习在A股进行量化投资。
量化投资/机器学习咨询培训/项目管理产品管理。

Intro

In this chapter, we will start by looking at the Linear Regression model, discuss two very different ways to train it(close-form and Gradient Descent)

Next we will look at Polynomial Regression and several regularization techniques that can reduce the risk of overfitting the training set.

Finally, we will look at two more models that are commonly used for classification tasks: Logistic Regression and Softmax Regression

Linear Regression

More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the bias term (also called the intercept term)

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

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the ith feature value.
- θ_j is the jth model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

Linear Regression

This can be written much more concisely using a vectorized form

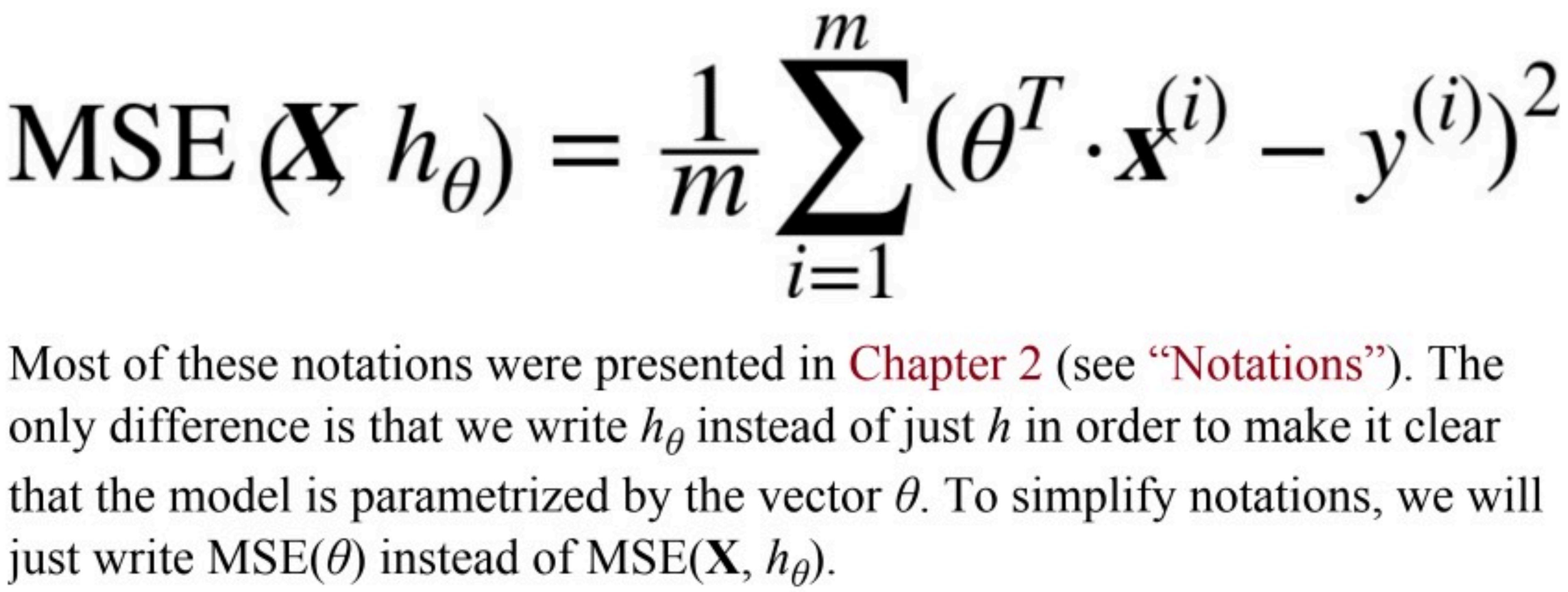
$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

- θ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.

- θ^T is the transpose of θ (a row vector instead of a column vector).
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- $\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and $\mathbf{x}$.
- h_θ is the hypothesis function, using the model parameters θ.

Linear Regression

MSE: performance measure of a regression model



$$\mathrm{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Most of these notations were presented in Chapter 2 (see "Notations"). The only difference is that we write h_θ instead of just h in order to make it clear that the model is parametrized by the vector θ. To simplify notations, we will just write $\mathrm{MSE}(\theta)$ instead of $\mathrm{MSE}(\mathbf{X}, h_\theta)$.

Linear Regression-TheNormalEuqation

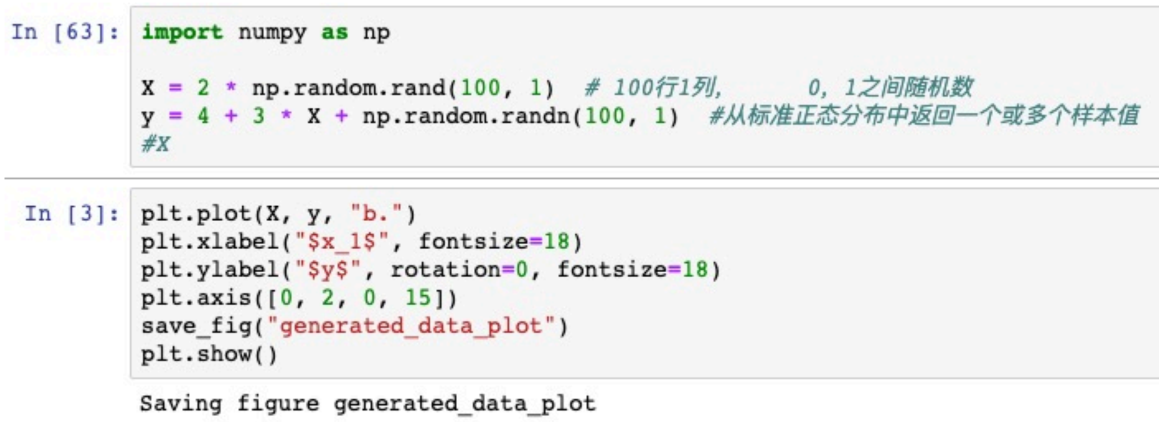
To find the value of θ that minimizes the cost function, there is a closed-form solution — in other words, a mathematical equation that gives the result directly. This is called the Normal Equation:

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

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

Linear Regression-TheNormalEuqation

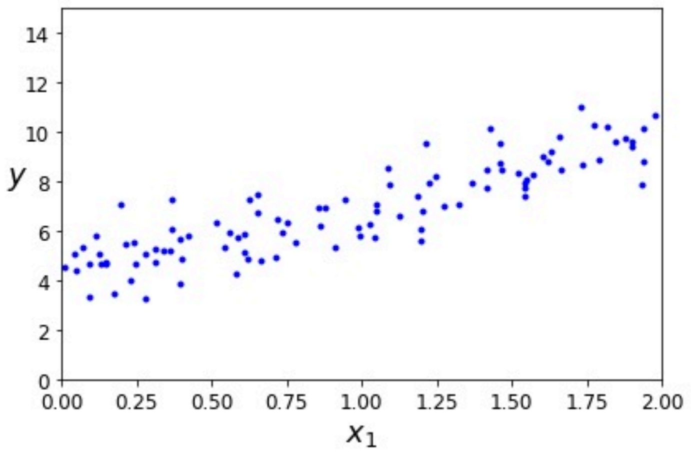
Let’ s generate some linear-looking data to test this equation
y = 4 + 3x + Gaussian noise



```
In [63]: import numpy as np

         X = 2 * np.random.rand(100, 1)  # 100行1列,        0, 1之间随机数
         y = 4 + 3 * X + np.random.randn(100, 1)  #从标准正态分布中返回一个或多个样本值
         #X
```

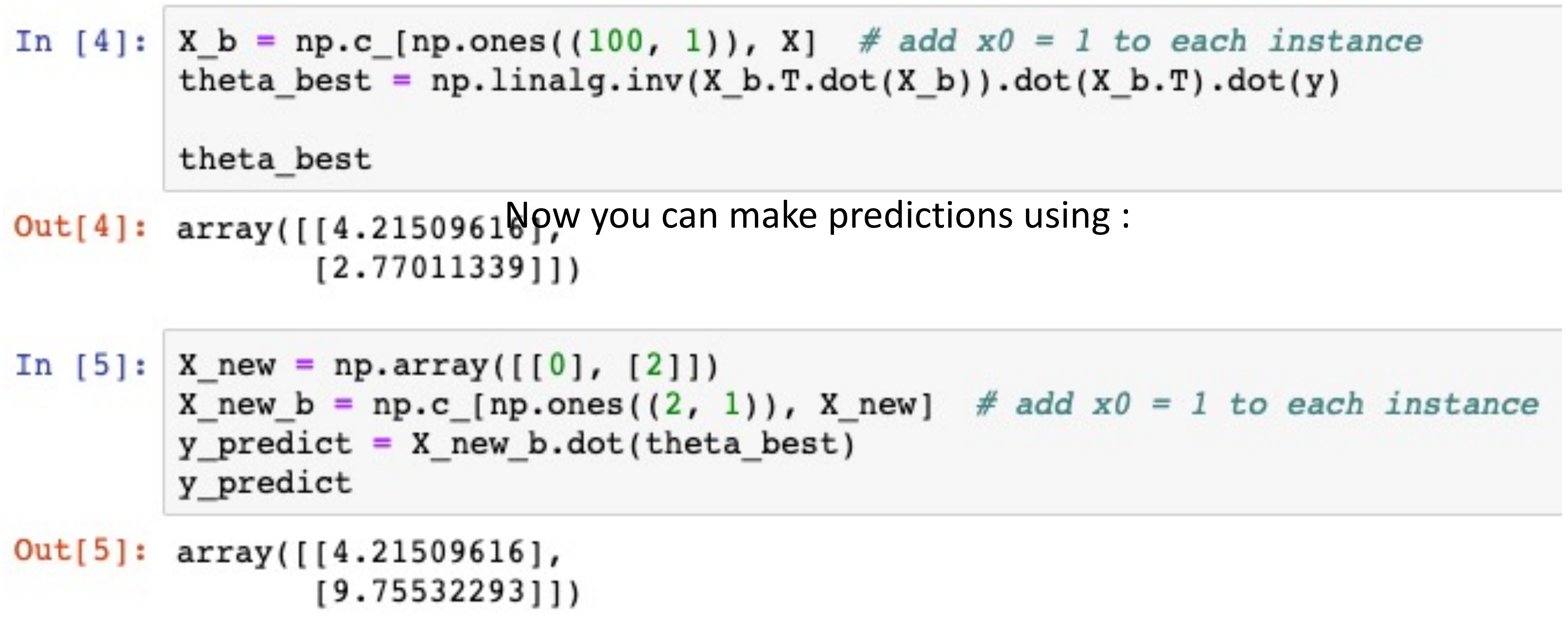
```
In [3]: plt.plot(X, y, "b.")
        plt.xlabel("$x_1$", fontsize=18)
        plt.ylabel("$y$", rotation=0, fontsize=18)
        plt.axis([0, 2, 0, 15])
        save_fig("generated_data_plot")
        plt.show()
```

```
Saving figure generated_data_plot
```



Linear Regression-TheNormalEuqation

Let' s see what the equation found
Now you can make predictions using θ:



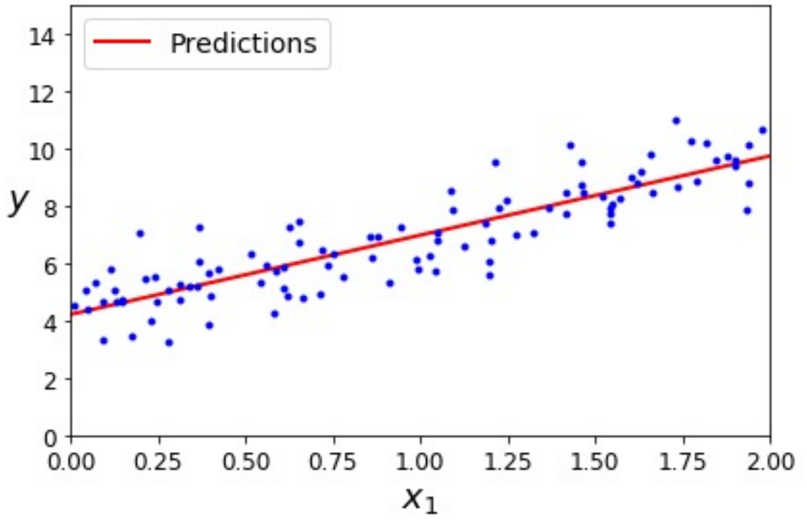
```
In [4]: X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
        theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

        theta_best

Out[4]: array([[4.21509616],
               [2.77011339]])
```

Now you can make predictions using :

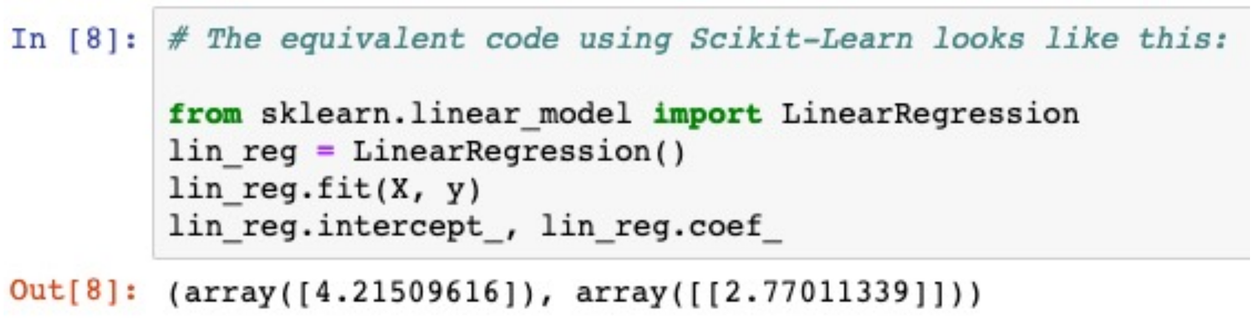
```
In [5]: X_new = np.array([[0], [2]])
        X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
        y_predict = X_new_b.dot(theta_best)
        y_predict

Out[5]: array([[4.21509616],
               [9.75532293]])
```

Linear Regression-TheNormalEuqation

plot this model’ s predictions



The equivalent code using Scikit-Learn



```
In [8]: # The equivalent code using Scikit-Learn looks like this:

        from sklearn.linear_model import LinearRegression
        lin_reg = LinearRegression()
        lin_reg.fit(X, y)
        lin_reg.intercept_, lin_reg.coef_

Out[8]: (array([4.21509616]), array([[2.77011339]]))
```

Linear Regression-Complexity

Computational Complexity:

1. for Normal Equation, on).If you double the number of features, you multiply the computation time by roughly 2`2.4 = 5.3 to 2`3 = 8
2. On the positive side, this equation is linear with regards to the number of instances in the training set (it is O(m)).
3. In other words, making predictions on twice as many instances (or twice as many features) will just take roughly twice as much time.

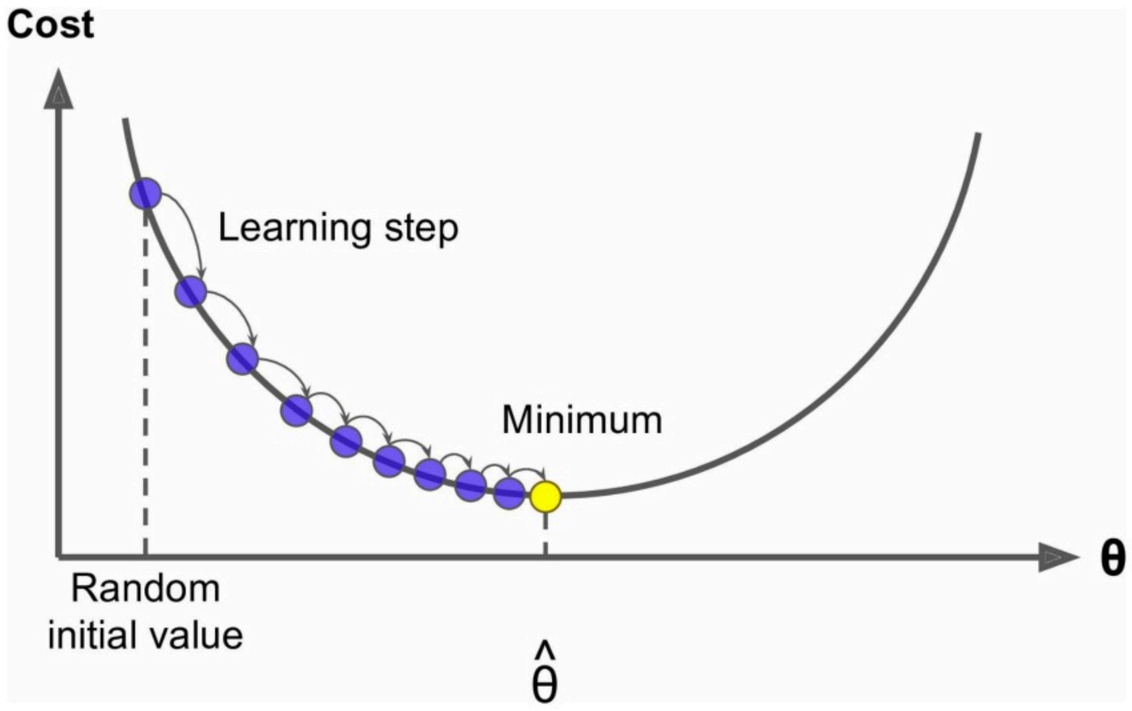
Now we will look at very different ways to train a Linear Regression model, better suited for cases where there are a large number of features, or too many training instances to fit in memory.

Gradient Descent

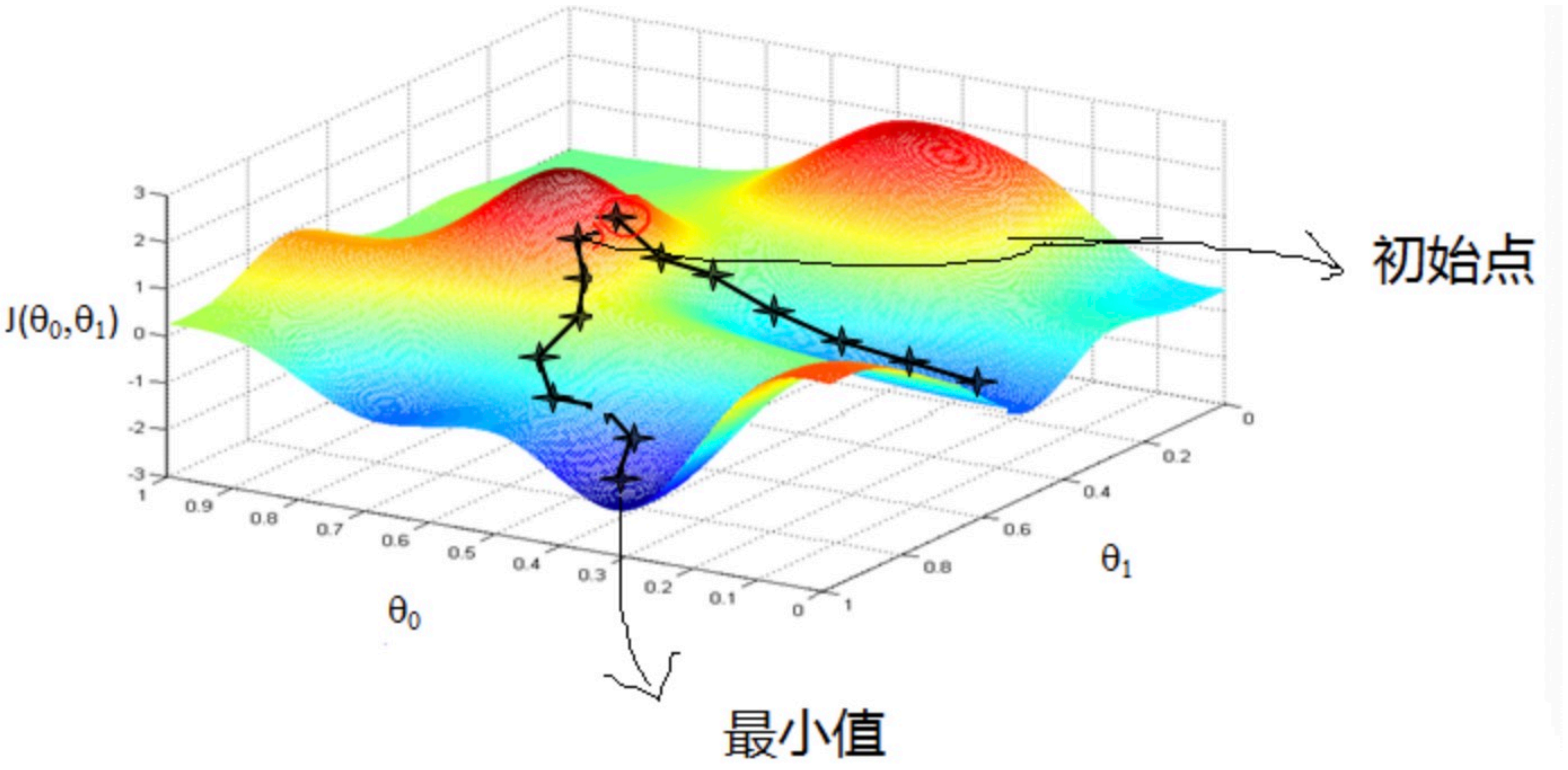
Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The **general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function**.

Gradient Descent

Concretely, you start by filling θ with random values (this is called random initialization), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum

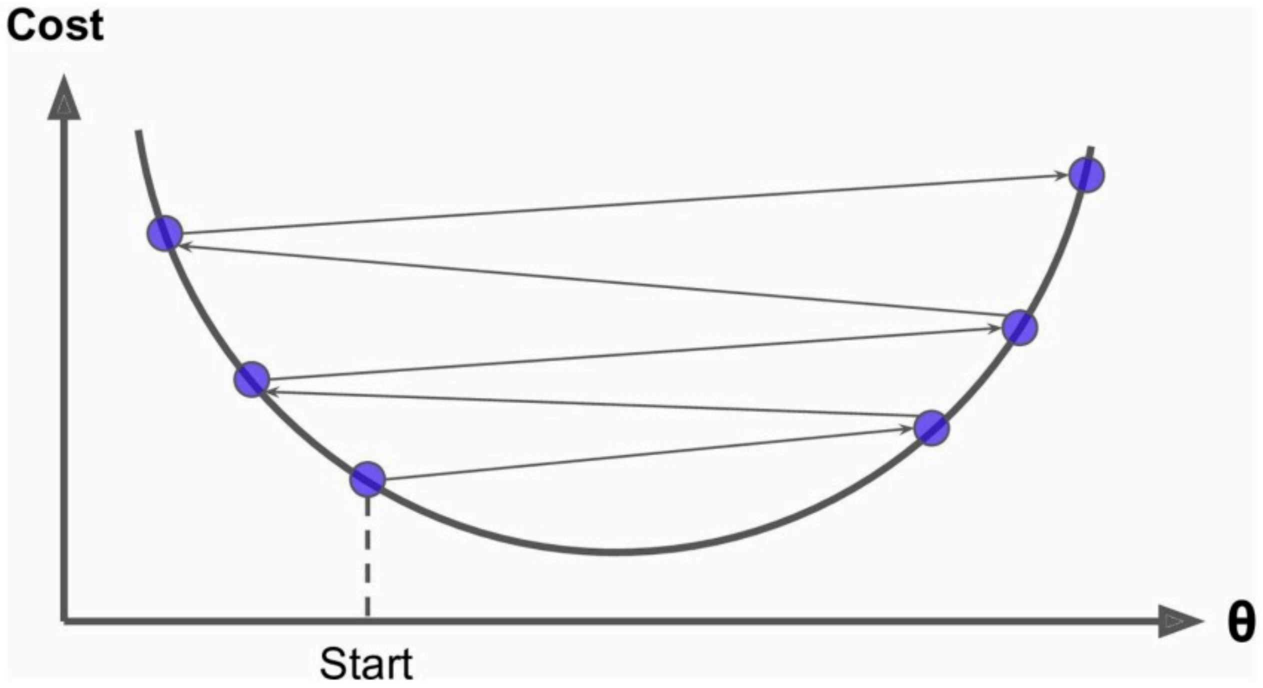


Gradient Descent



Gradient Descent

On the other hand, 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, failing to find a good solution



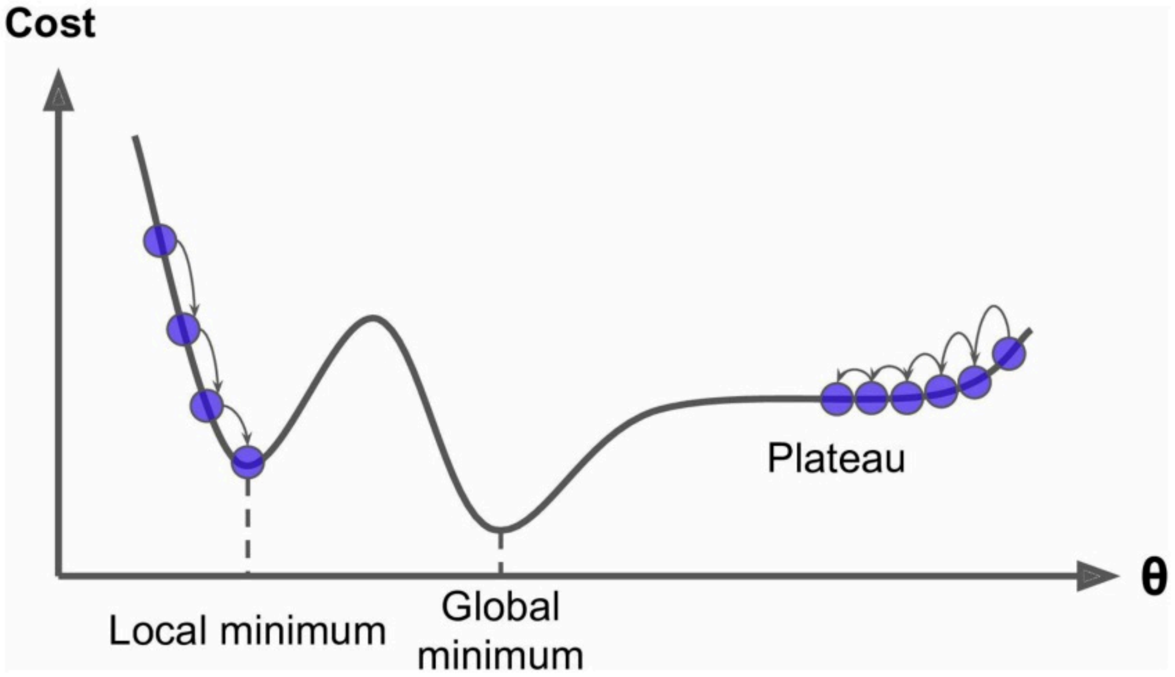
Gradient Descent

What is parameters and hyperparameters?

1. parameters: learned automatically with training data
2. hyperparameters: tweak manually to optimize models(learning-rate etc.)

Gradient Descent

Two main challenges:
if the random initialization starts the algorithm on the left, then it will converge to a local minimum, which is not as good as the global 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 globalminimum.



Gradient Descent

About Features’　scale:
the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales

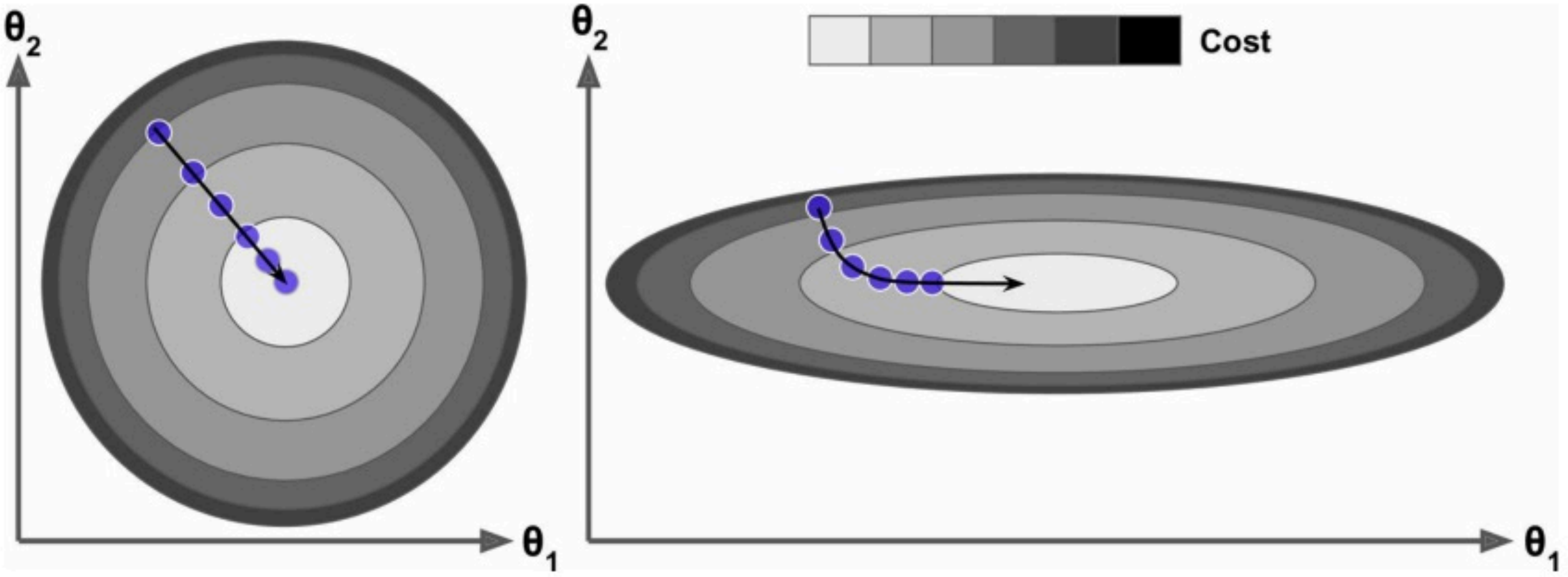


Figure 4-7. Gradient Descent with and without feature scaling

Gradient Descent-Batch GD

computes the partial derivative of the cost function with regards to parameter θj

Equation 4-5. Partial derivatives of the cost function

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

Gradient Descent-Batch GD

Instead of computing these gradients individually, you can use Equation 4-6 to compute them all in one go. The gradient vector, noted ∇θMSE(θ), contains all the partial derivatives of the cost function (one for each model parameter)

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

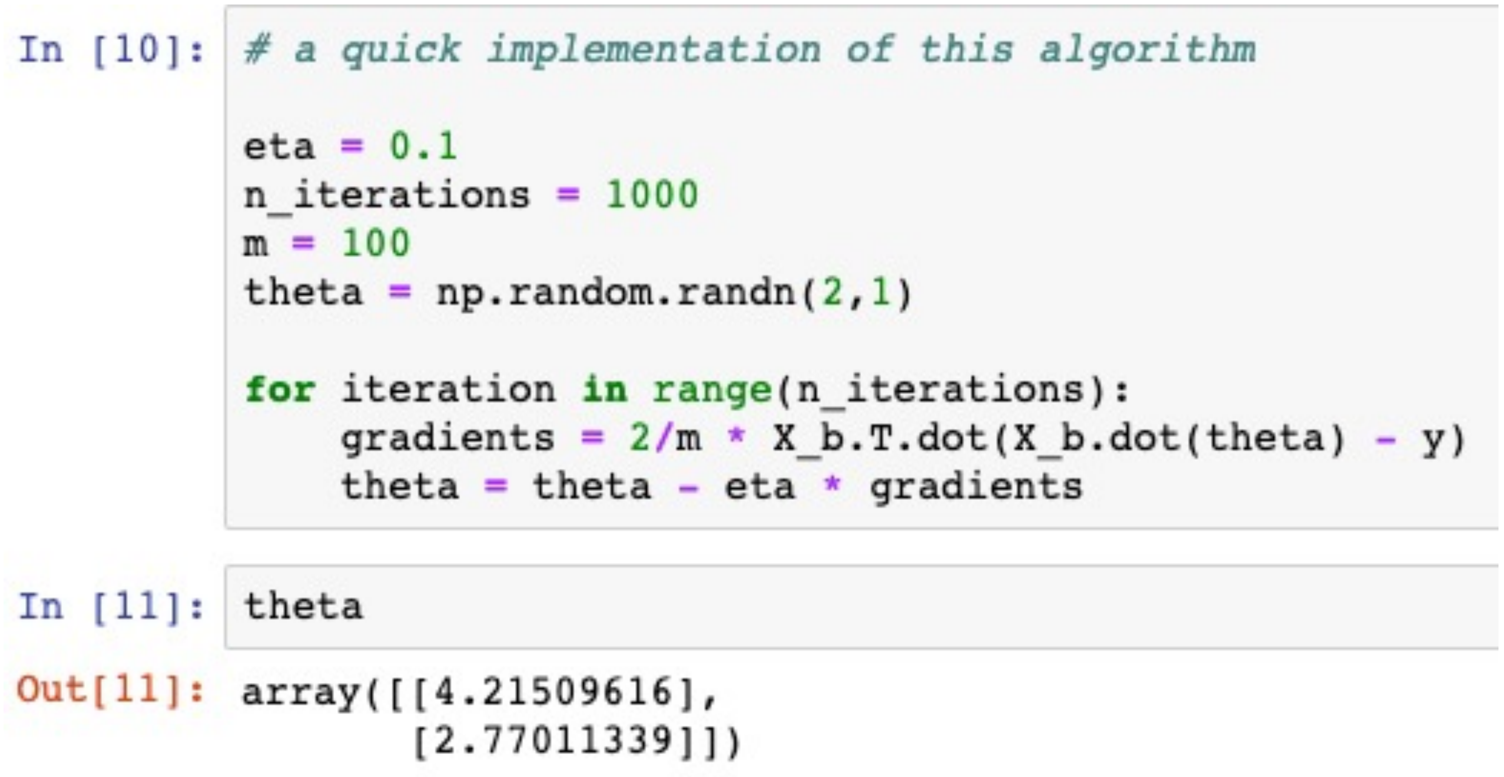
Gradient Descent-Batch GD

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting ∇θMSE(θ) from θ. This is where the learning rate η comes into play:
multiply the gradient vector by η to determine the size of the downhill step

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

Gradient Descent-Batch GD

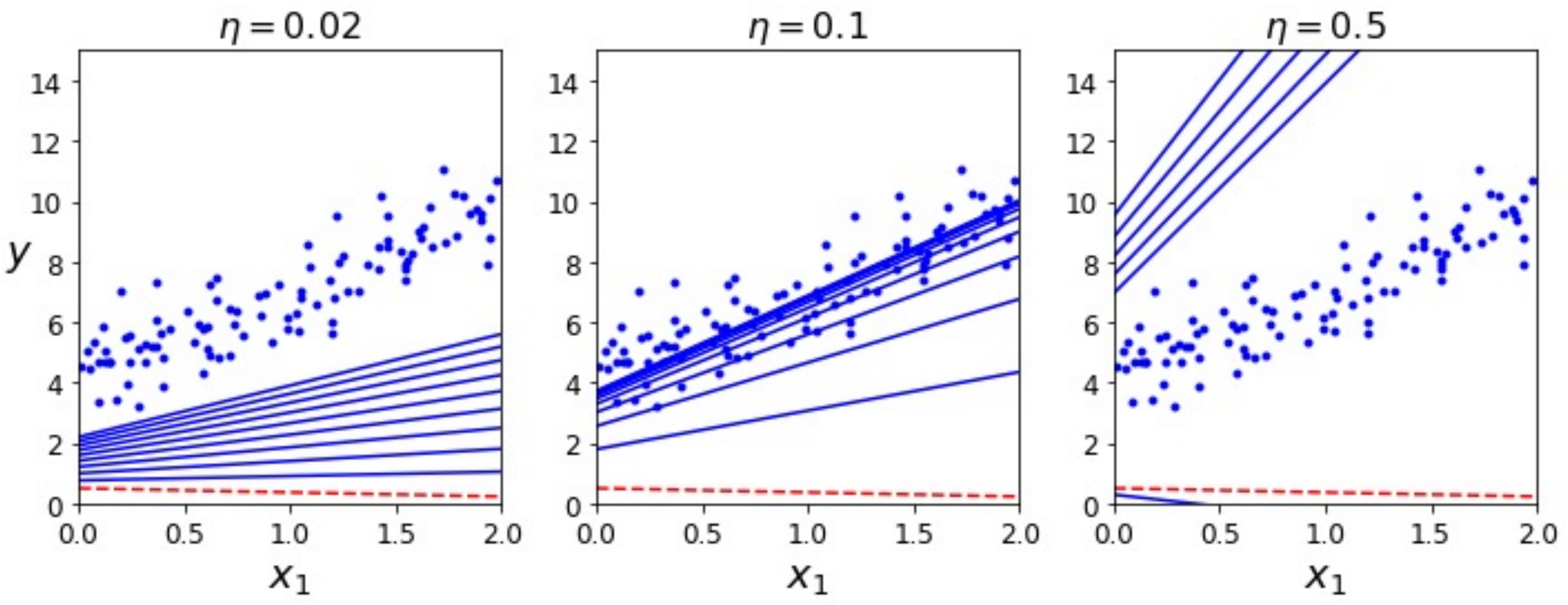
a quick implementation of this algorithm



```
In [10]: # a quick implementation of this algorithm

         eta = 0.1
         n_iterations = 1000
         m = 100
         theta = np.random.randn(2,1)

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

In [11]: theta

Out[11]: array([[4.21509616],
                [2.77011339]])
```

Gradient Descent-Batch GD

what if you had used a different learning rate eta



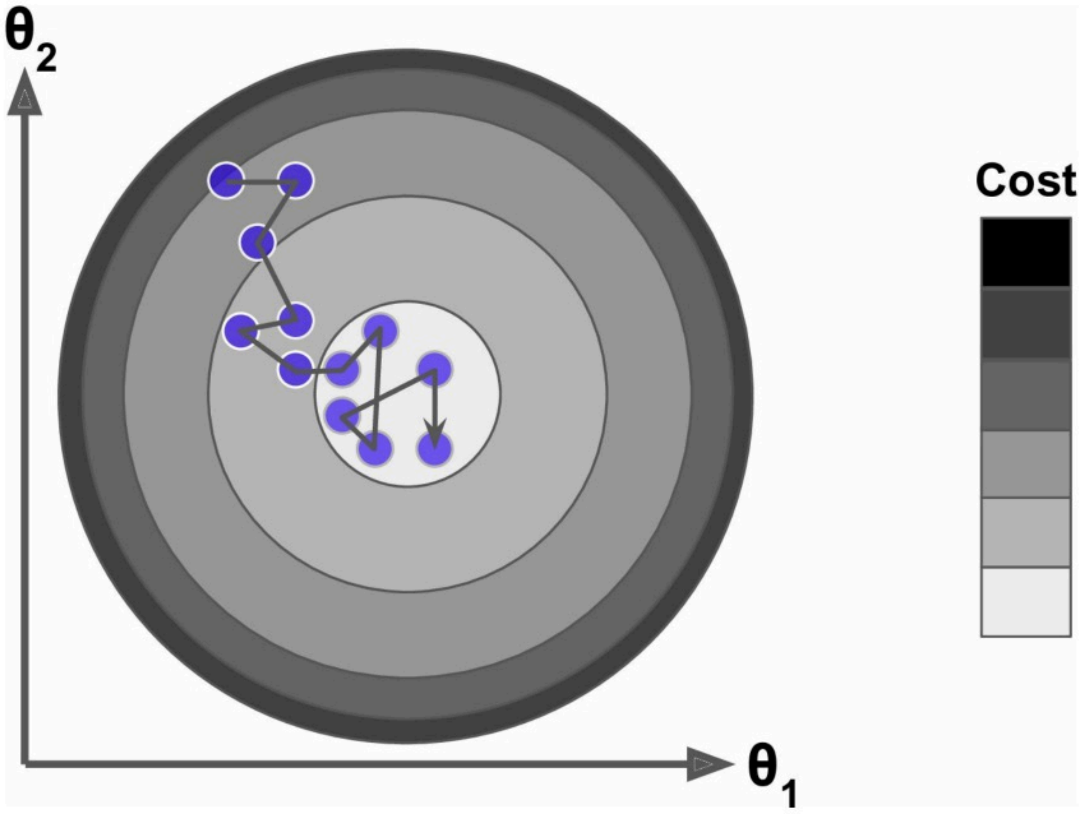
Gradient Descent-Batch GD

You may wonder how to set the number of iterations. If it is too low, you will still be far away from the optimal solution when the algorithm stops, but if it is too high, you will waste time while the model parameters do not change anymore.

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 ϵ (called the tolerance) — because this happens when Gradient Descent has (almost) reached the minimum.

Gradient Descent-Stochastic GD

BGD uses the whole training set to compute the gradients at every step. At the opposite extreme, Stochastic Gradient Descent just picks a random instance in the training set at every step and computes the gradients based only on that single instance



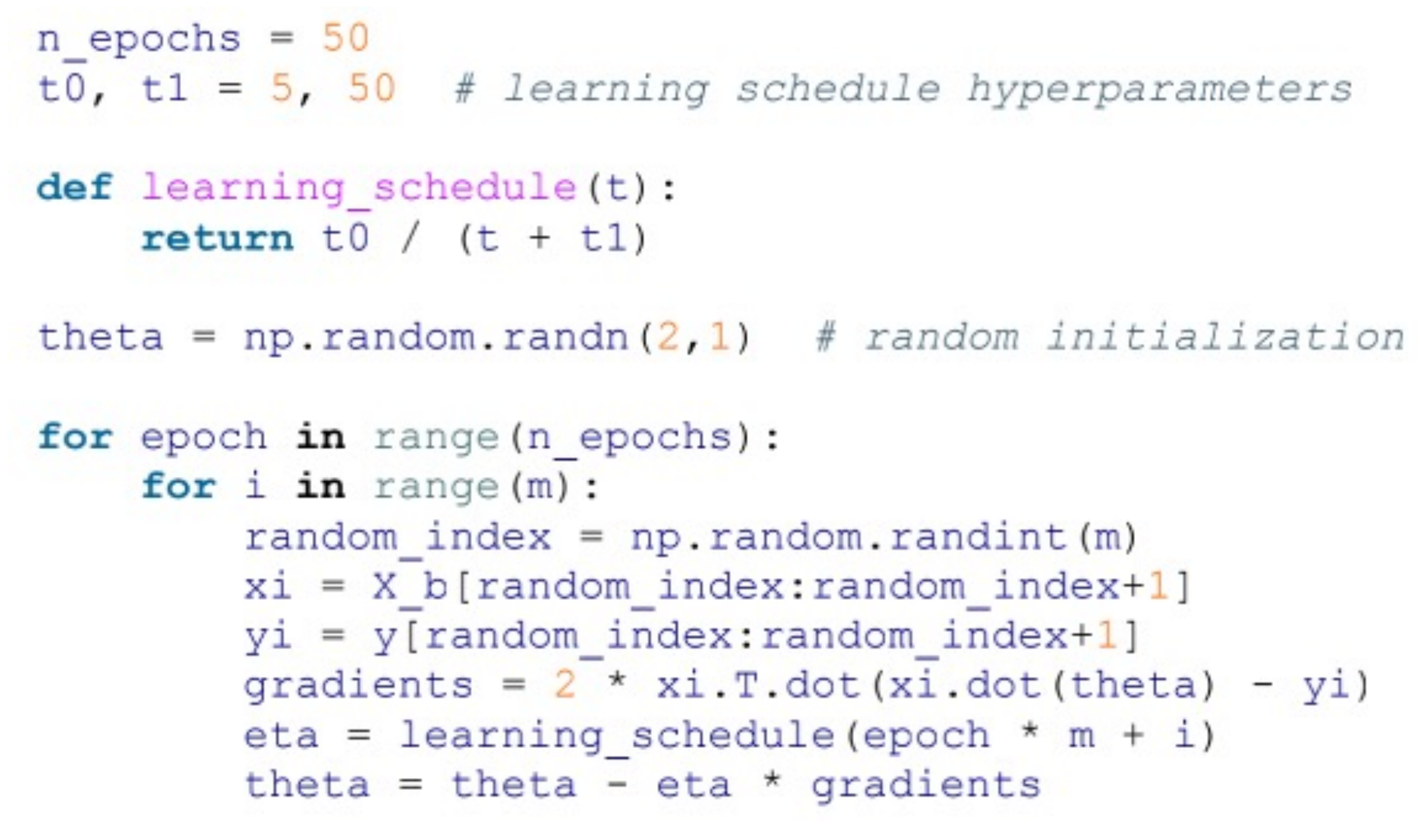
Gradient Descent-Stochastic GD

Therefore 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 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.

Gradient Descent-Stochastic GD

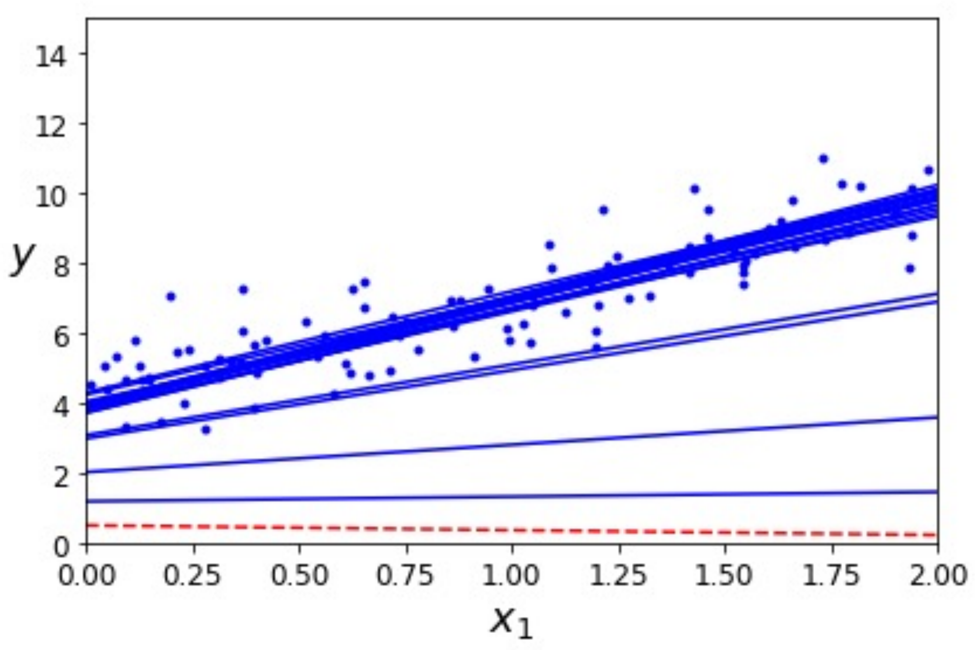
This code implements Stochastic Gradient Descent using a simple learning schedule:



```
n_epochs = 50
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):
        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
```

Gradient Descent-Stochastic GD

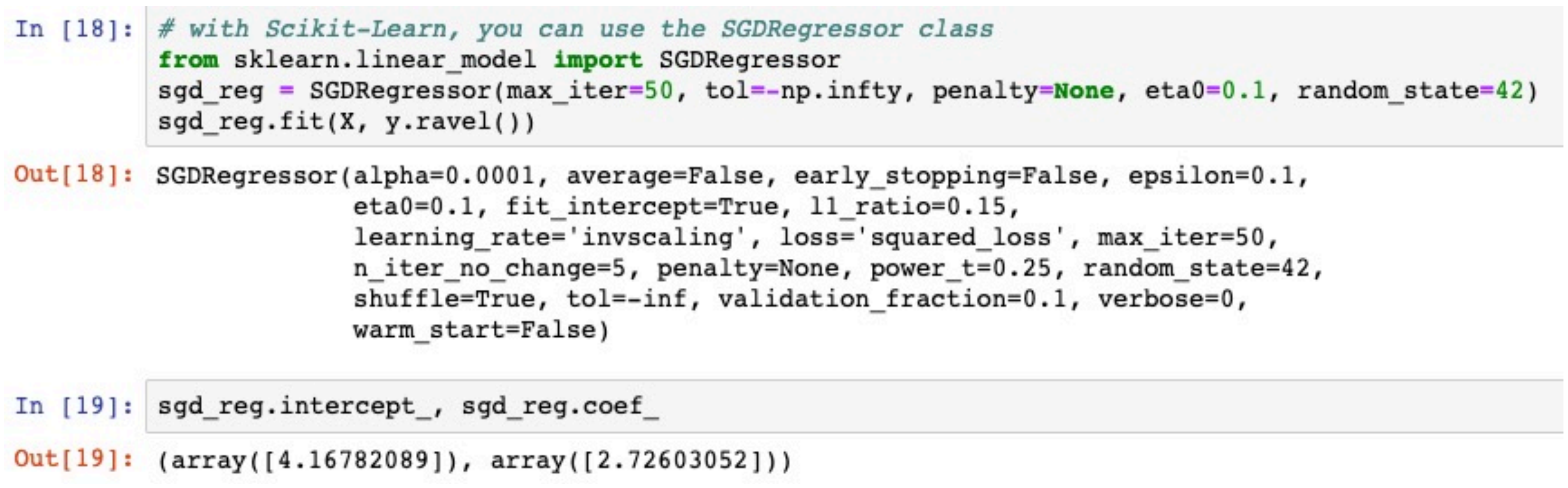
Below shows the first 10 steps of training (notice how irregular the steps are)



```
In [17]: theta

Out[17]: array([[4.21076011],
                [2.74856079]])
```

Gradient Descent-Stochastic GD

perform Linear Regression using SGD with Scikit-Learn,



```
In [18]: # with Scikit-Learn, you can use the SGDRegressor class
         from sklearn.linear_model import SGDRegressor
         sgd_reg = SGDRegressor(max_iter=50, tol=-np.infty, penalty=None, eta0=0.1, random_state=42)
         sgd_reg.fit(X, y.ravel())

Out[18]: SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
                eta0=0.1, fit_intercept=True, l1_ratio=0.15,
                learning_rate='invscaling', loss='squared_loss', max_iter=50,
                n_iter_no_change=5, penalty=None, power_t=0.25, random_state=42,
                shuffle=True, tol=-inf, validation_fraction=0.1, verbose=0,
                warm_start=False)

In [19]: sgd_reg.intercept_, sgd_reg.coef_

Out[19]: (array([4.16782089]), array([2.72603052]))
```

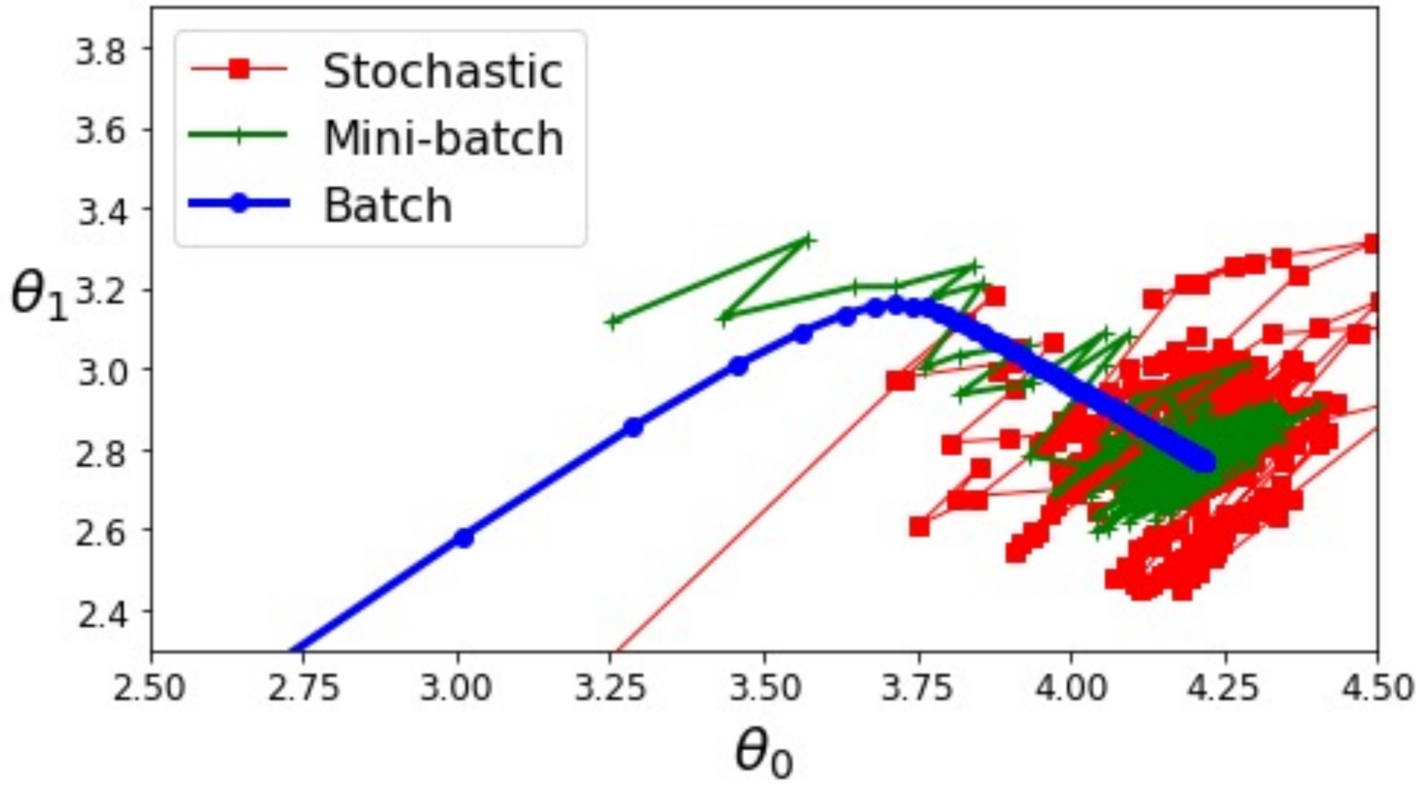
Gradient Descent-Mini-batch GD

The last Gradient Descent algorithm we will look at is called Mini-batch:

It is quite simple to understand once you know Batch and
Stochastic Gradient Descent: at each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called mini-batches

Gradient Descent-Mini-batch GD

Shows the paths taken by the three Gradient Descent algorithms in parameter space during training.



Gradient Descent-Mini-batch GD

Let’ s compare the algorithms we’ ve discussed so far for Linear Regression(recall that m is the number of training instances and n is the number of features)

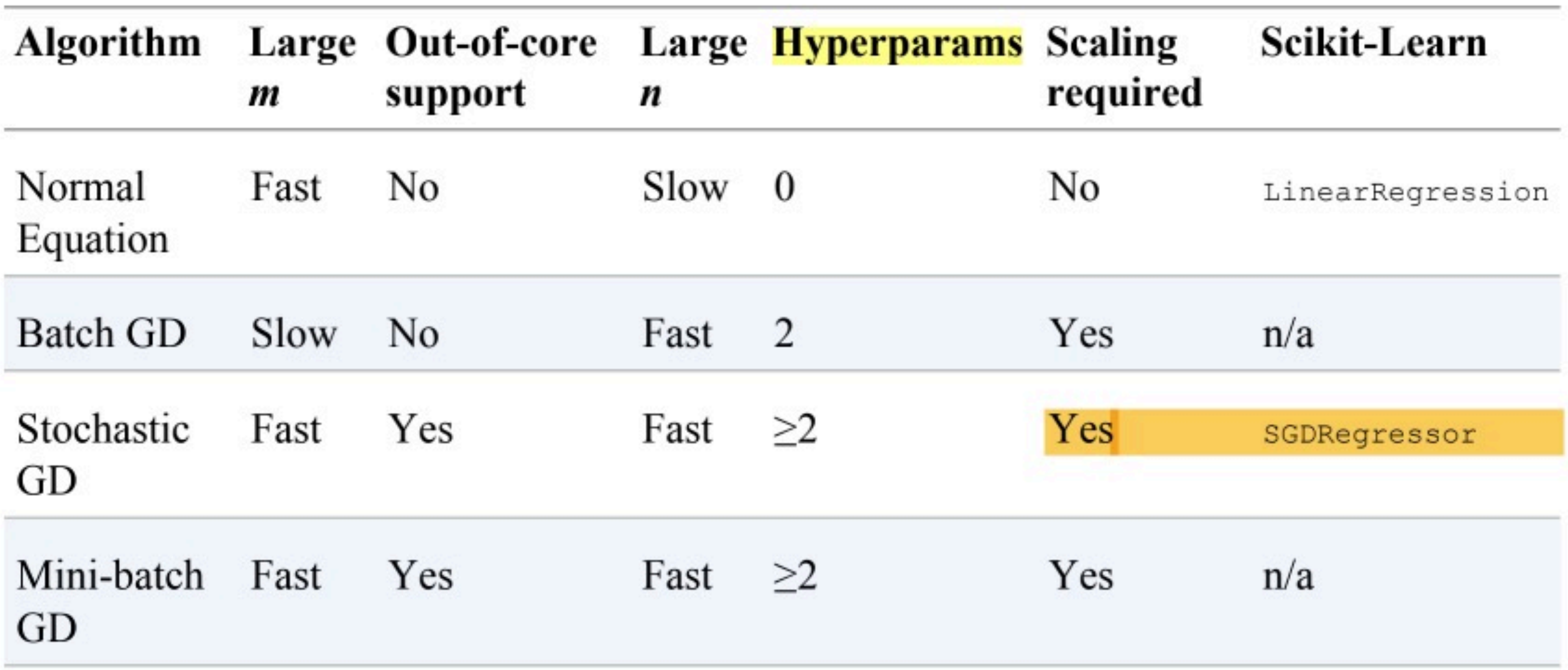


Table 4-1. Comparison of algorithms for Linear Regression

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

Polynomial Regression

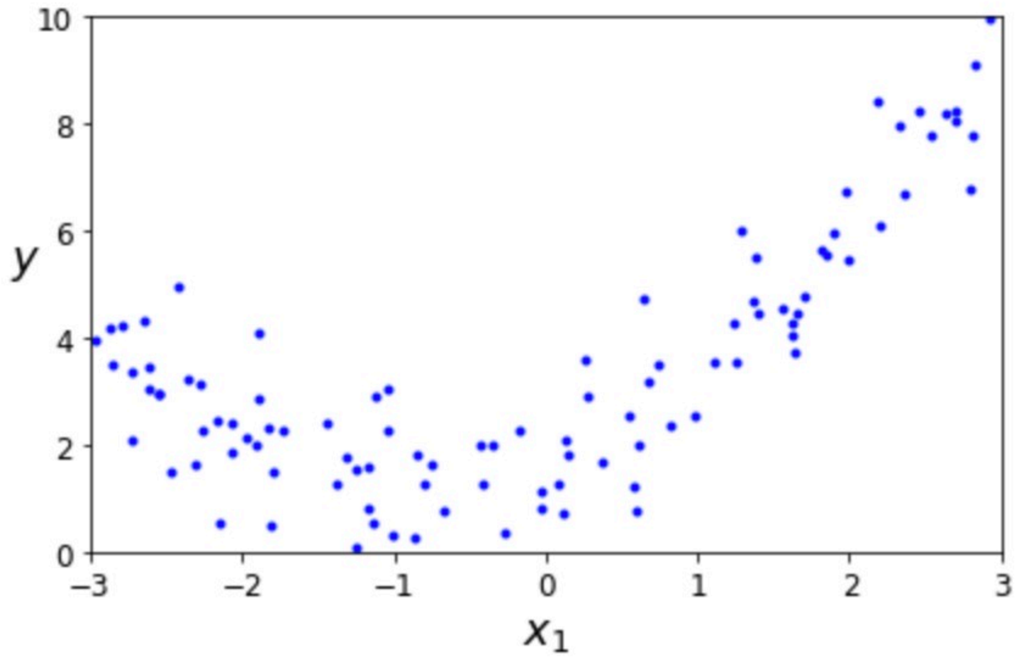
1. this is to add powers of each feature as new features,
2. then train a linear model on this extended set of features

This technique is called Polynomial Regression

Polynomial Regression

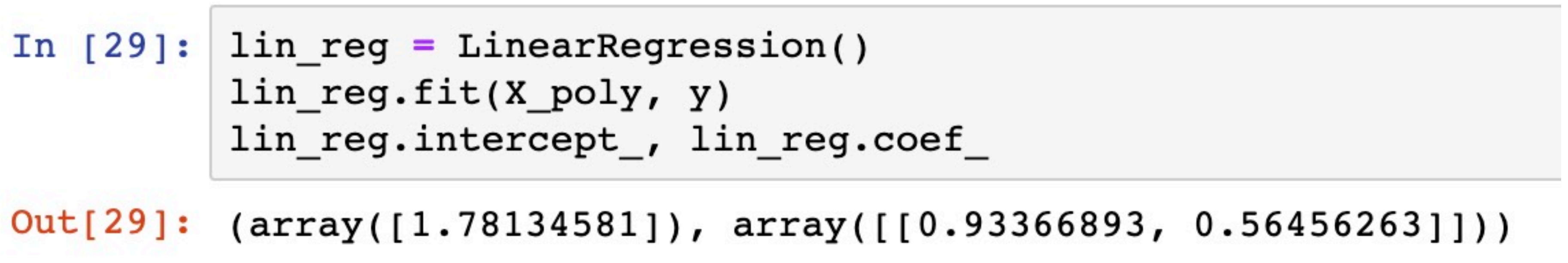
Example with some nonlinear data

```
In [25]: m = 100
         X = 6 * np.random.rand(m, 1) - 3
         y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```



Polynomial Regression

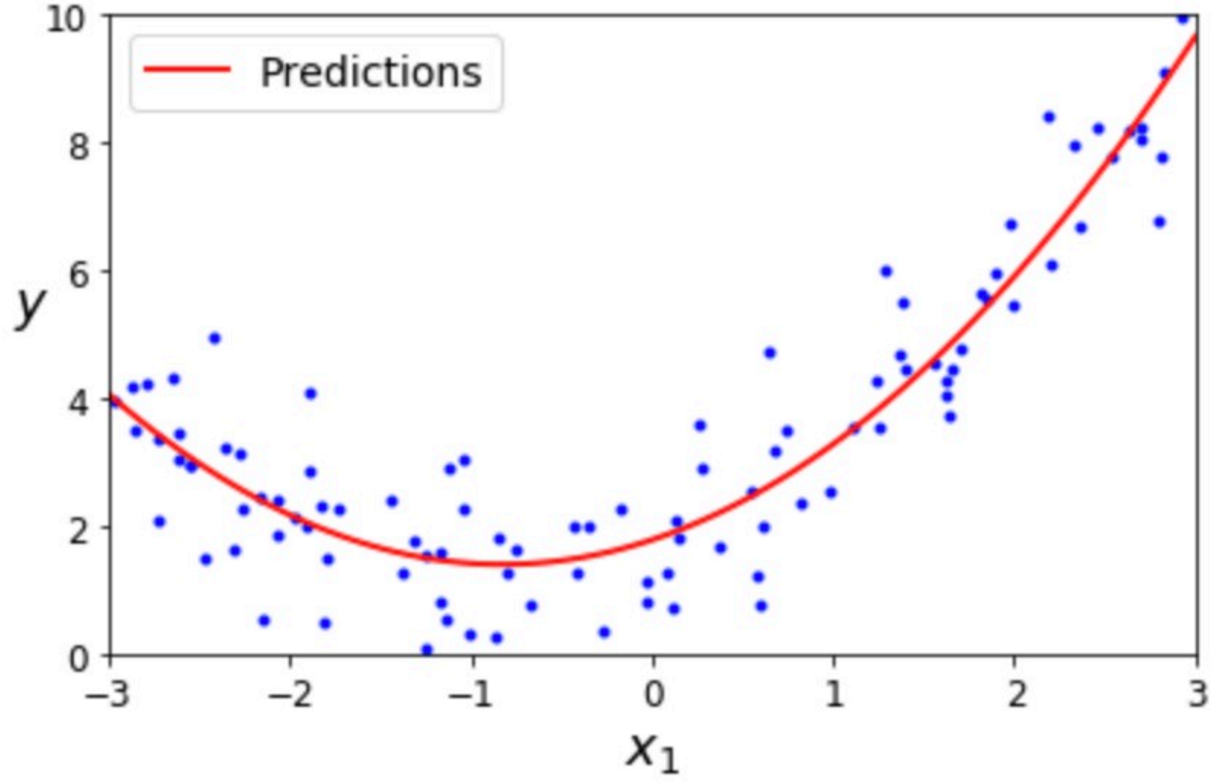
use Scikit-Learn’ s PolynomialFeatures class to transform our training data, adding the square (2nd-degree polynomial) of each feature in the training set as new features



```
In [29]: lin_reg = LinearRegression()
         lin_reg.fit(X_poly, y)
         lin_reg.intercept_, lin_reg.coef_

Out[29]: (array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

Polynomial Regression

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



Polynomial Regression

Note that when there are multiple features, Polynomial Regression is capable of finding relationships between features (which is something a plain Linear Regression model cannot do).

This is made possible by the fact that PolynomialFeatures also adds all combinations of features up to the given degree.

For example, if there were two features a and b, PolynomialFeatures with degree=3 would not only add the features a^2 , a^3 , b^2 , b^3 , but also ab, a^2b, ab^2

Regularized Linear Models

For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at Ridge Regression, Lasso Regression, and Elastic Net, which implement three different ways to constrain the weights

Regularized Linear Models-Ridge

Ridge Regression (also called Tikhonov regularization) is a regularized version of Linear Regression: a regularization term equal to a*θ^2 is added to the cost function.

This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible.

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

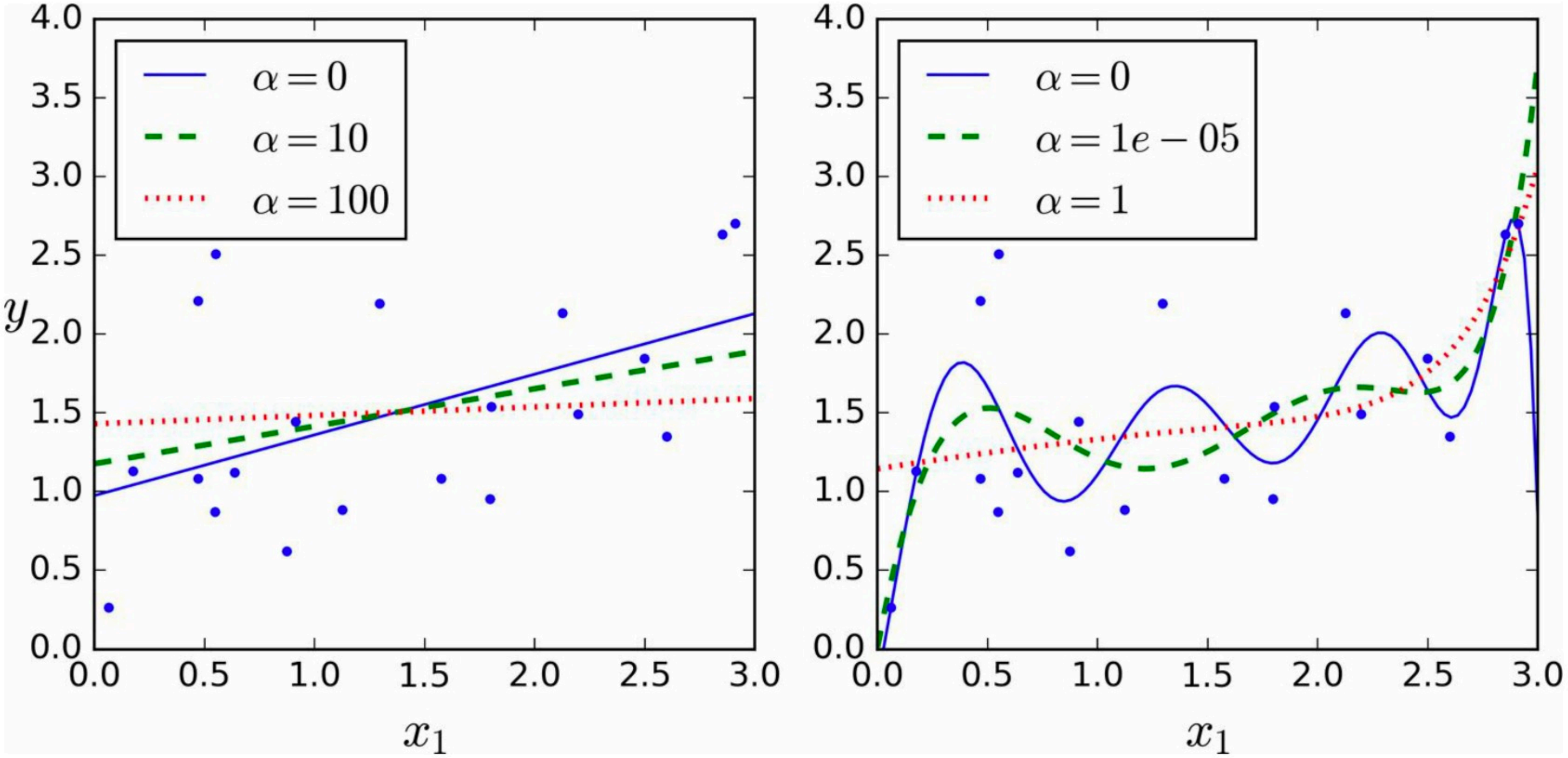
Regularized Linear Models-Ridge

The hyperparameter α controls how much you want to regularize the mode

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

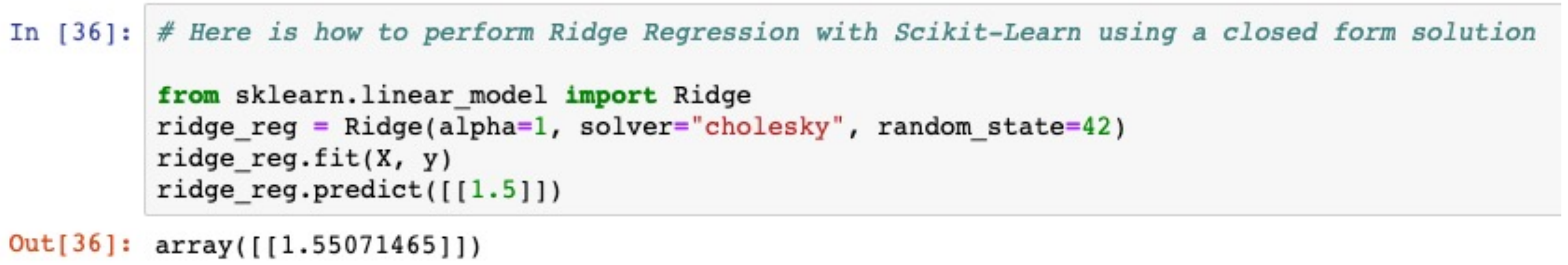
Regularized Linear Models-Ridge

shows several Ridge models trained on some linear data using different α value

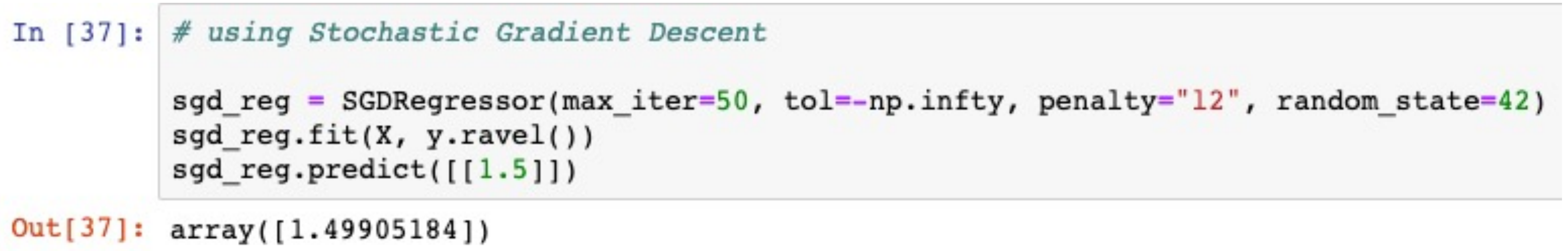


Regularized Linear Models-Ridge

Here is how to perform Ridge Regression with Scikit-Learn using a closed-form solution and SGD.



```
In [36]: # Here is how to perform Ridge Regression with Scikit-Learn using a closed form solution

         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]])

Out[36]: array([[1.55071465]])
```



```
In [37]: # using Stochastic Gradient Descent

         sgd_reg = SGDRegressor(max_iter=50, tol=-np.infty, penalty="l2", random_state=42)
         sgd_reg.fit(X, y.ravel())
         sgd_reg.predict([[1.5]])

Out[37]: array([1.49905184])
```

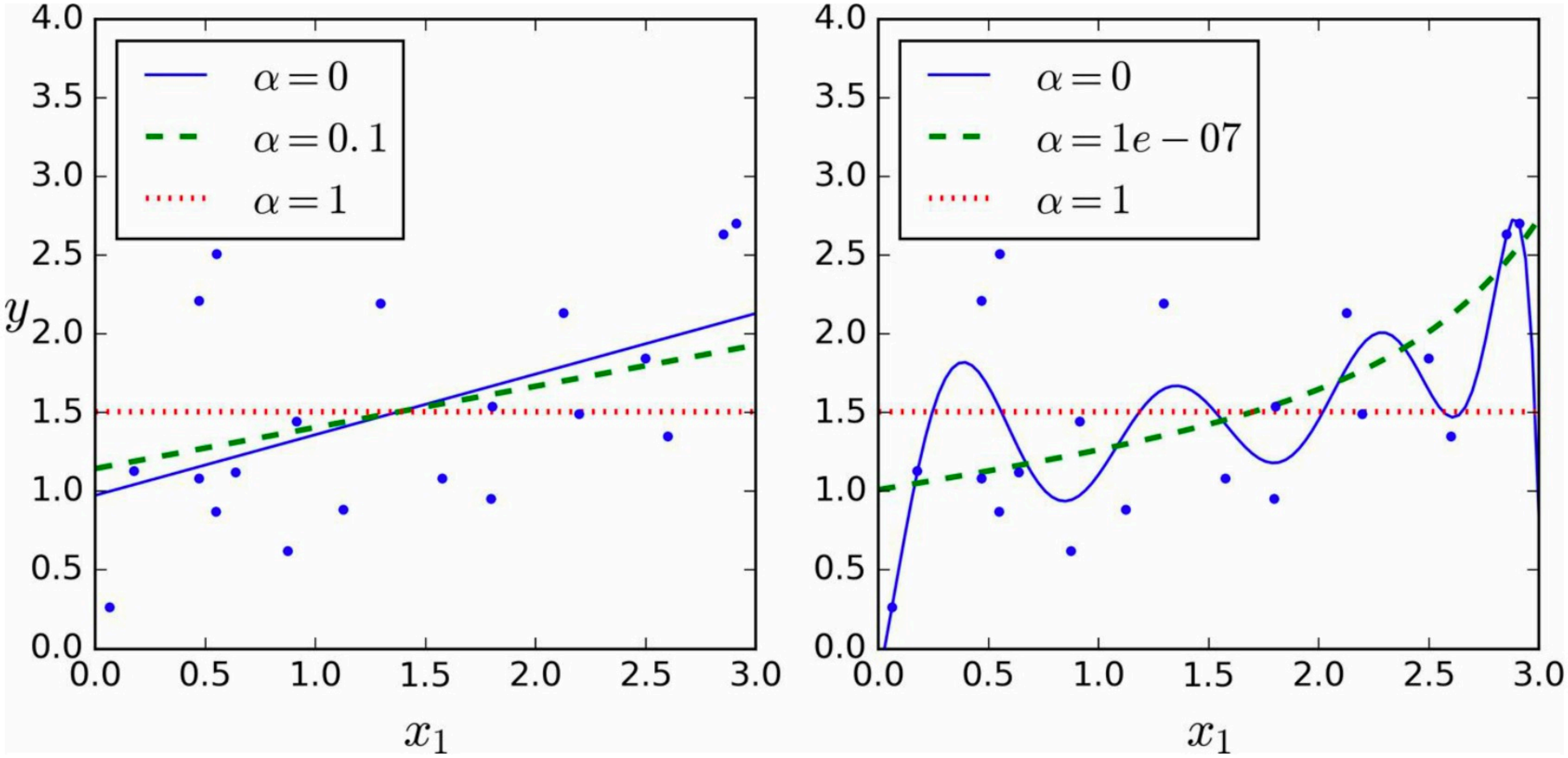
Regularized Linear Models-Lasso

Lasso Regression is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ1 norm of the weight vector instead of half the square of the ℓ2 norm

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

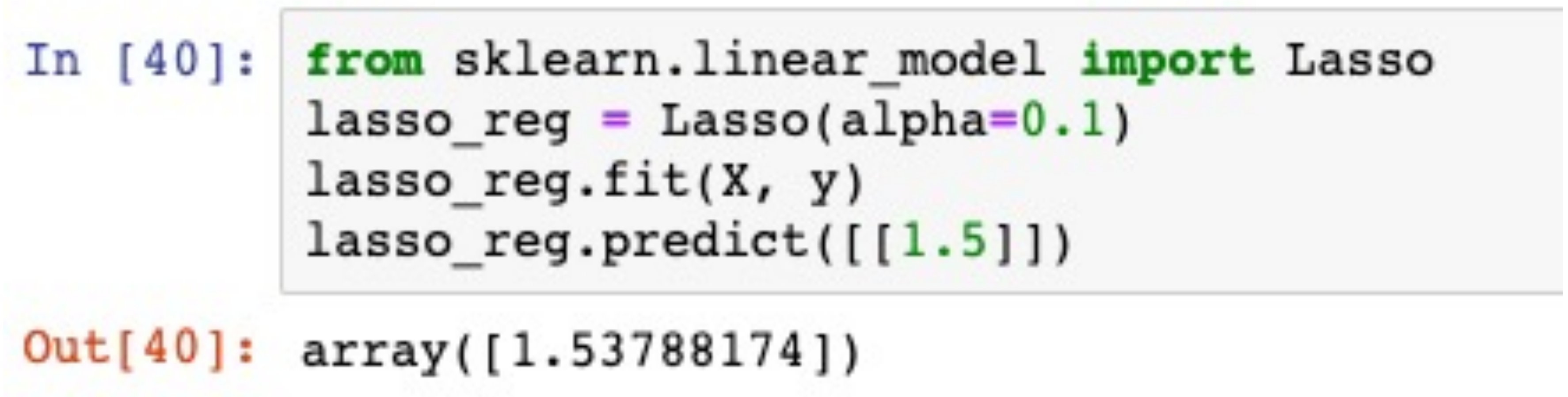
Regularized Linear Models-Lasso

shows the same thing but replaces Ridge models with Lasso models and uses smaller α values



Regularized Linear Models-Lasso

Here is a small Scikit-Learn example using the Lasso class. Note that you could instead use an SGDRegressor(penalty="l1")



```
In [40]: from sklearn.linear_model import Lasso
         lasso_reg = Lasso(alpha=0.1)
         lasso_reg.fit(X, y)
         lasso_reg.predict([[1.5]])

Out[40]: array([1.53788174])
```

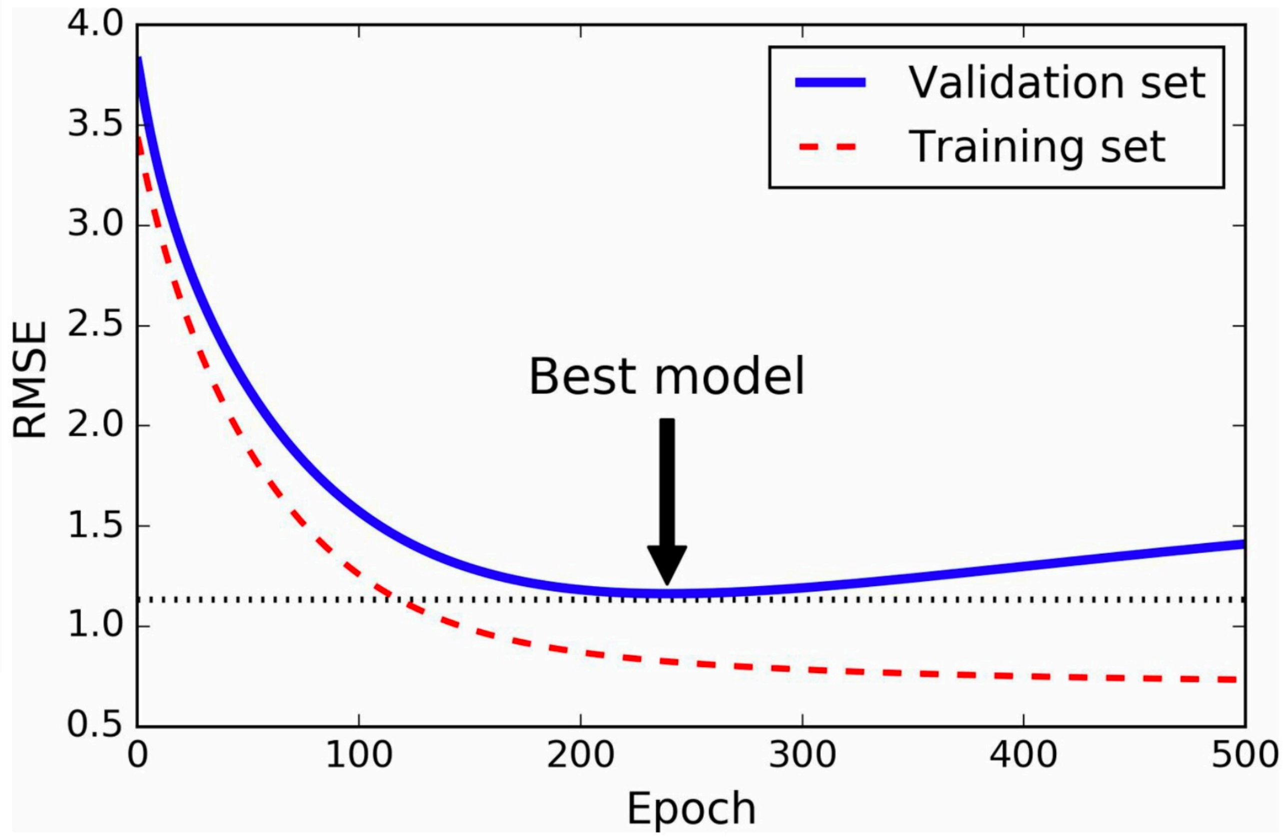
Regularized Linear Models-Elastic

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso’ s regularization terms, and you can control the mix ratio r. When r = 0, Elastic Net is equivalent to Ridge Regression, and when r = 1, it is equivalent to Lasso Regression

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

Regularized Linear Models-Early Stopping

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called early stopping.



Logistic Regression

As we discussed early, some regression algorithms can be used for classification as well (and vice versa). Logistic Regression (also called Logit Regression) is commonly used to estimate the probability that an instance belongs to a particular class.

Logistic Regression

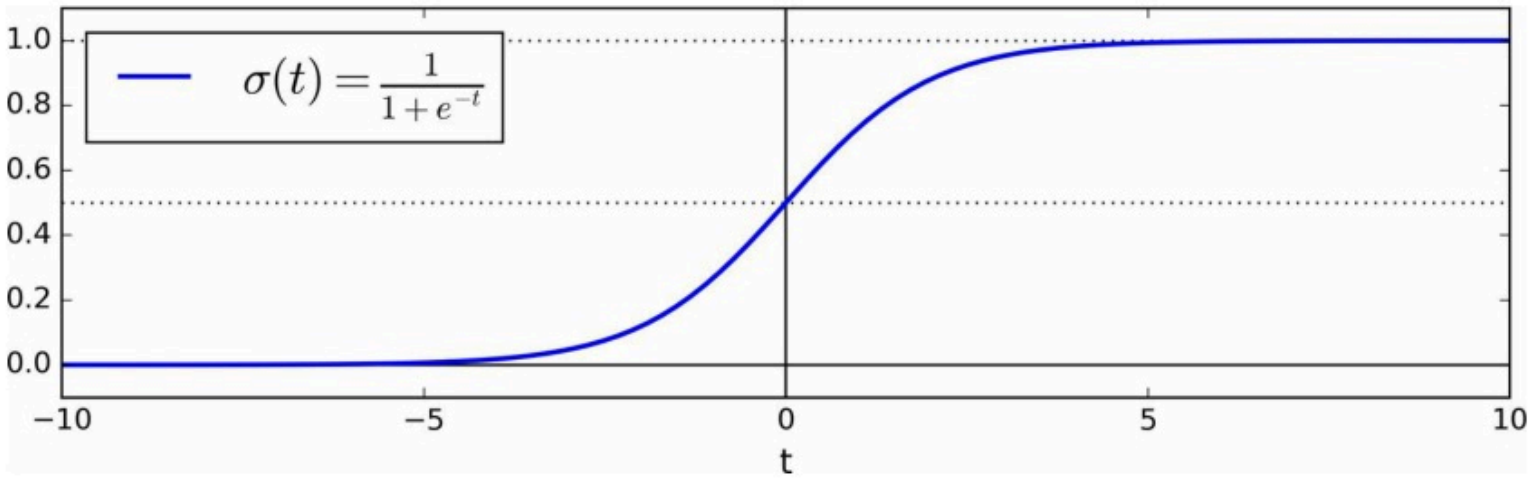
Logistic Regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the logistic of this result

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

Logistic Regression

The logistic — also called the logit, noted σ(·) — is a sigmoid function (i.e., S-shaped) that outputs a number between 0 and 1. It is defined as shown

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



Logistic Regression

Once the Logistic Regression model has estimated the probability = hθ(x) that an instance x belongs to the positive class, it can make its prediction ŷ easily

Notice that σ(t) < 0.5 when t < 0, and σ(t) ≥ 0.5 when t ≥ 0, so a LogisticRegression model predicts 1 if θT· x is positive, and 0 if it is negative.

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases}$$

Logistic Regression-Cost Function

The objective of training is to set the parameter vector θ so that the model estimates high probabilities for positive instances (y = 1) and low probabilities for negative instances (y = 0)

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

Logistic Regression-Cost Function

The cost function over the whole training set is simply the average cost over all training instances. It can be written in a single expression (as you can verify easily), called the log loss:

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

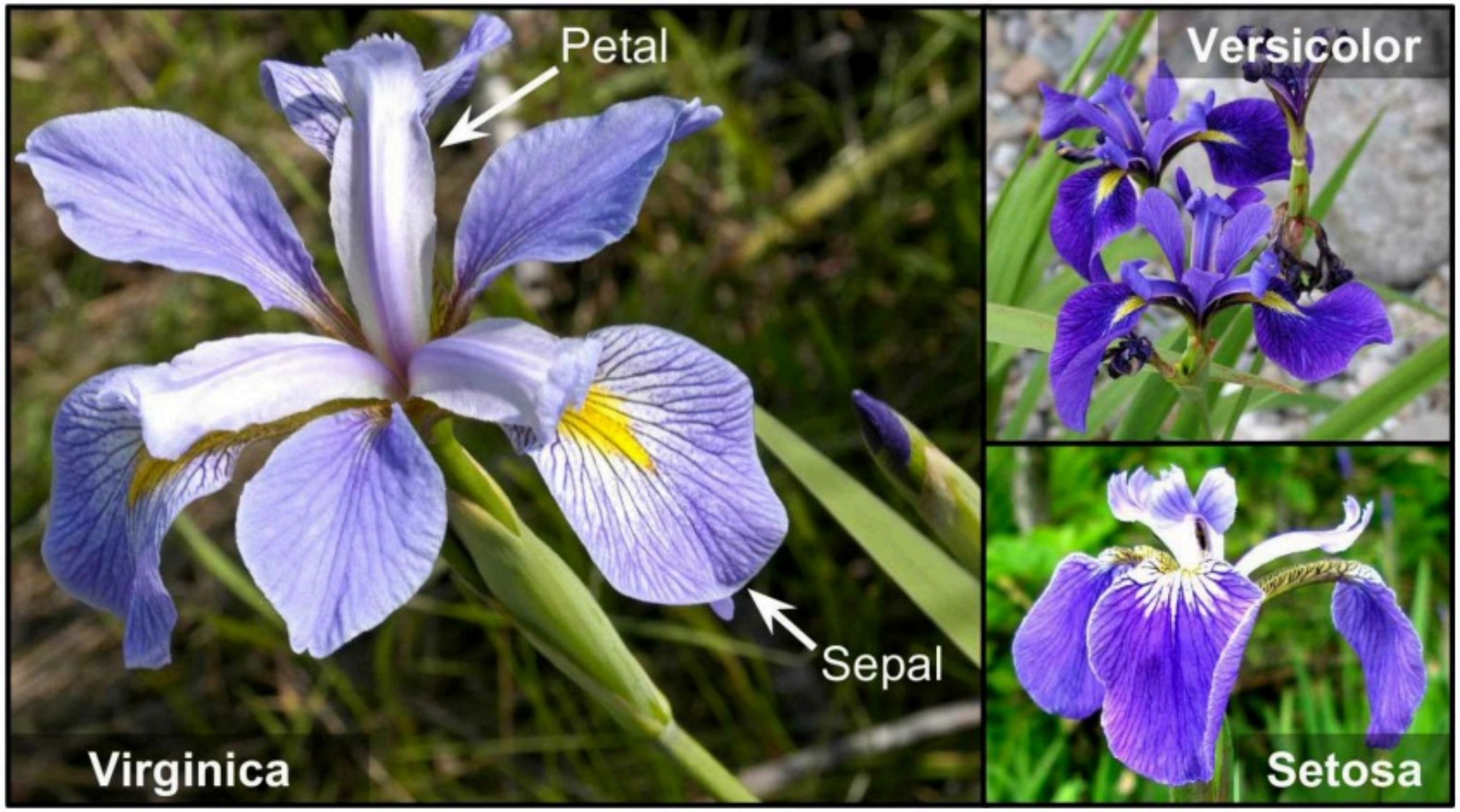
Logistic Regression-Cost Function

The partial derivatives of the cost function with regards to the j_th model parameter θj is given below

$$\frac{\partial}{\partial \theta_j} \mathrm{J}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

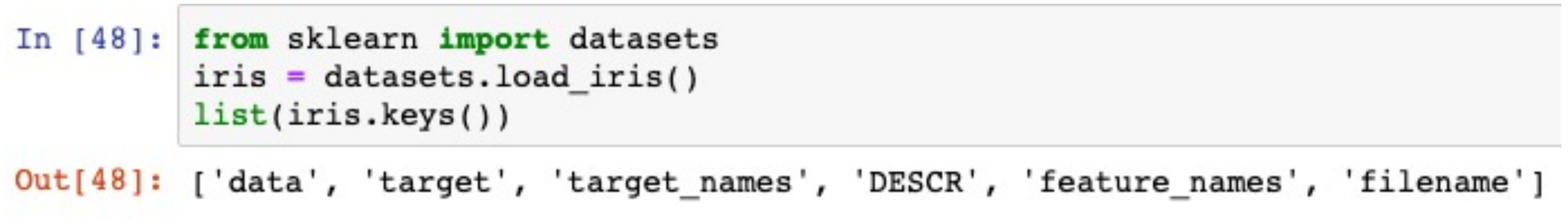
Logistic Regression-Decision Boundary

Let’ s use the iris dataset to illustrate Logistic Regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: Iris-Setosa, Iris-Versicolor, and Iris-Virginica

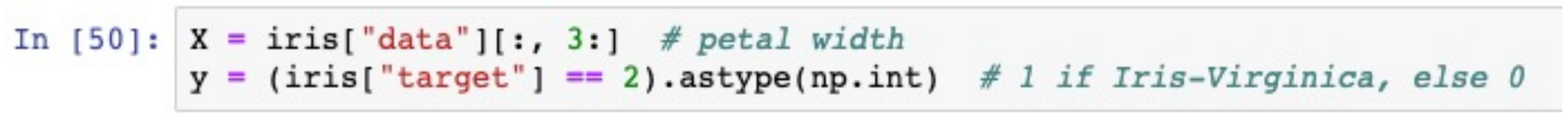


Logistic Regression-Decision Boundary

Let’ s try to build a classifier to detect the Iris-Virginica type based only on the petal width feature. First let’ s load the data



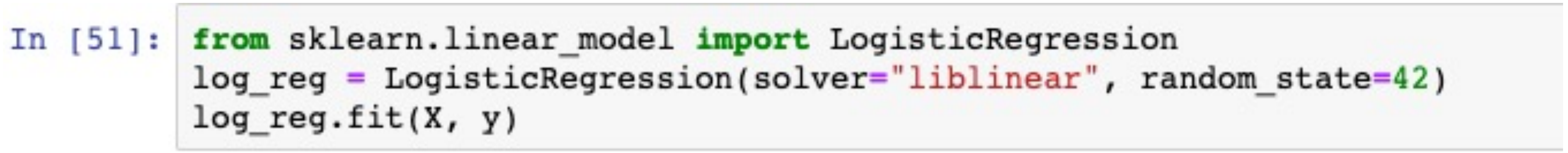
```
In [48]: from sklearn import datasets
         iris = datasets.load_iris()
         list(iris.keys())

Out[48]: ['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
```



```
In [50]: X = iris["data"][:, 3:]  # petal width
         y = (iris["target"] == 2).astype(np.int)  # 1 if Iris-Virginica, else 0
```

Logistic Regression-Decision Boundary

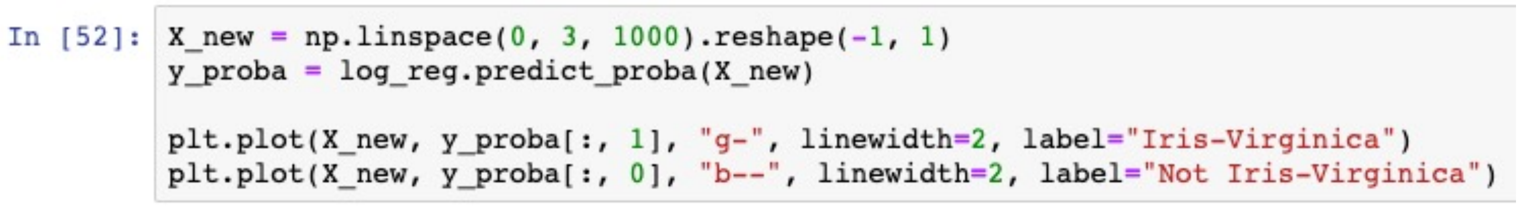
Now let’ s train a Logistic Regression model



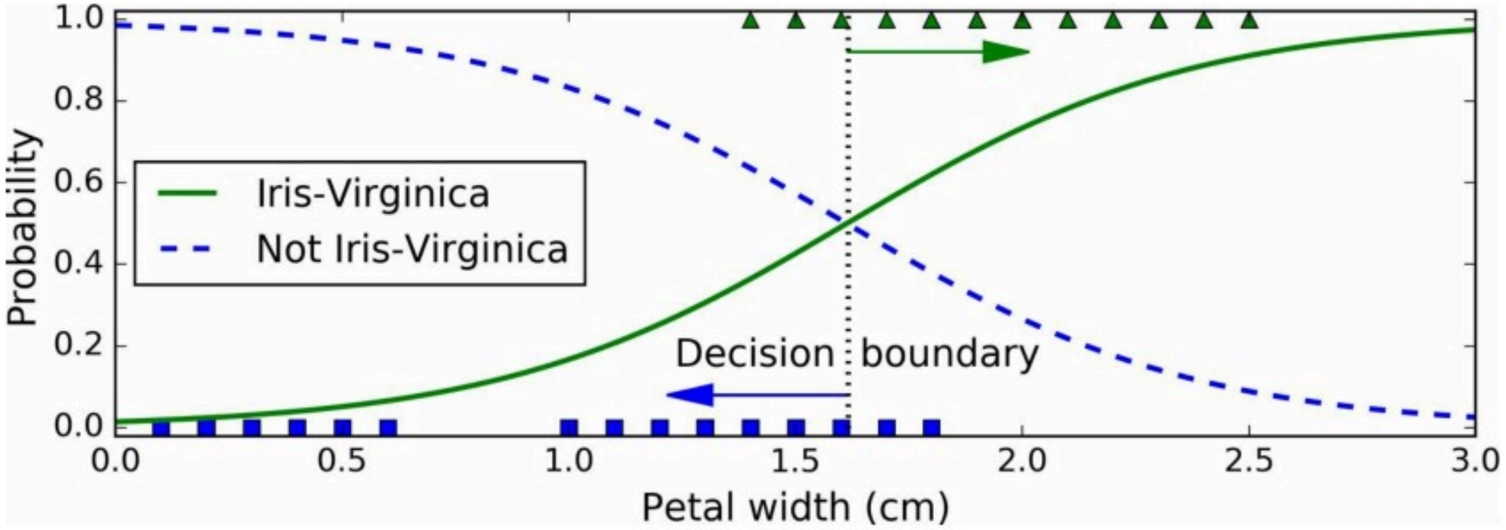
```
In [51]: from sklearn.linear_model import LogisticRegression
         log_reg = LogisticRegression(solver="liblinear", random_state=42)
         log_reg.fit(X, y)
```

Logistic Regression-Decision Boundary

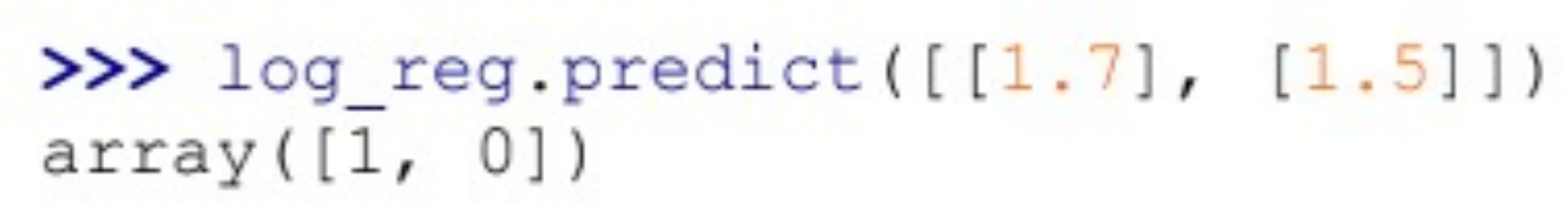
Let’ s look at the model’ s estimated probabilities for flowers with petal widths varying from 0 to 3 cm



```
In [52]: X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
         y_proba = log_reg.predict_proba(X_new)

         plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris-Virginica")
         plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not Iris-Virginica")
```



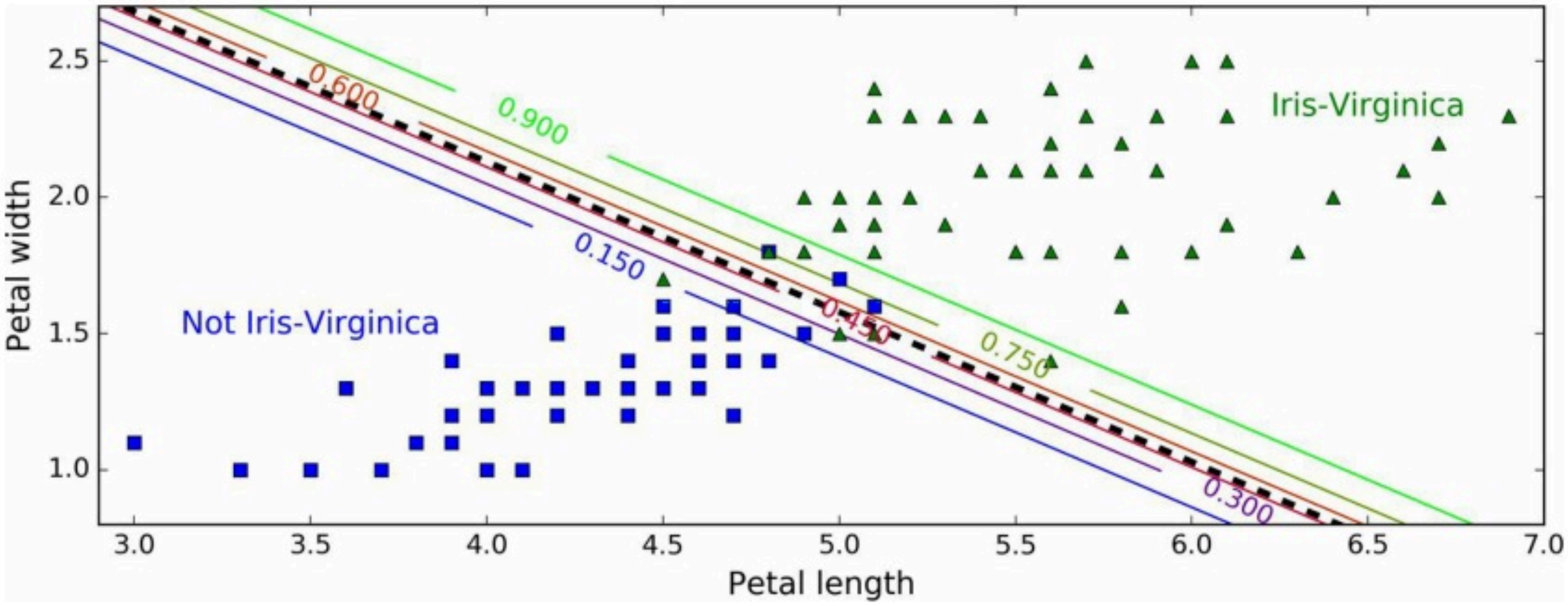
Logistic Regression-Decision Boundary



```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

Logistic Regression-Decision Boundary

shows the same dataset but this time displaying two features: petal width and length.



SAIF

Shanghai Advanced Institute of Finance

上海高级金融学院