Training Models

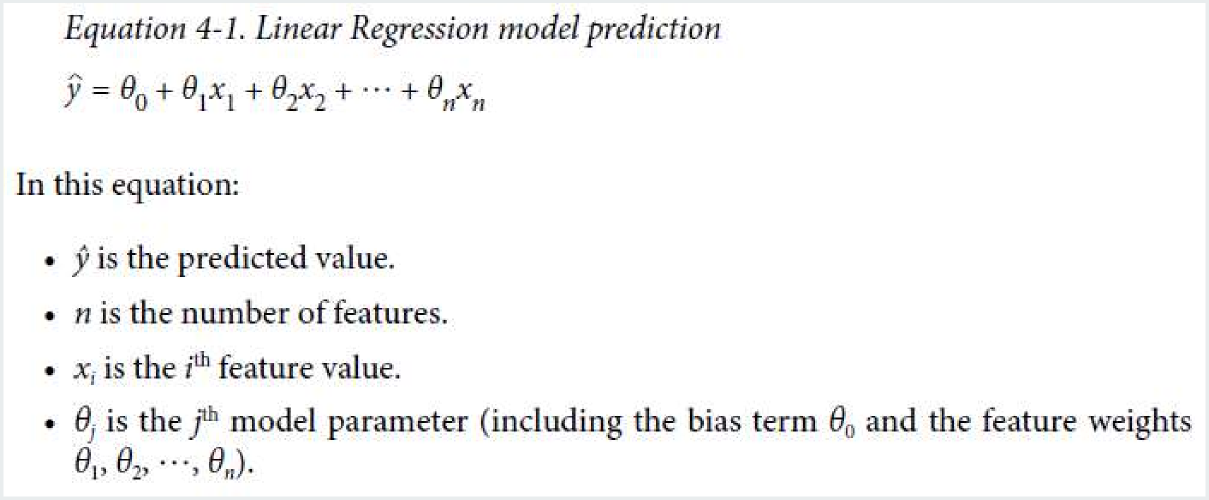
Linear Regression

Outline

- Linear Regression
- Gradient Descent
- Polynomial Regression
- Regularized Linear Regression
- **Logistic Regression**

Linear Regression

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

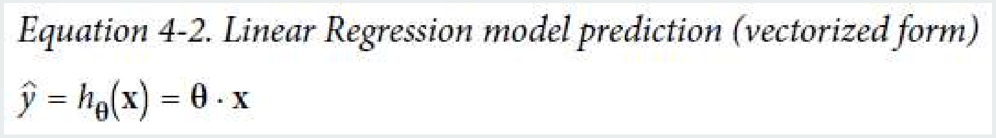


Equation 4-1. Linear Regression model prediction

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

In this equation:

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

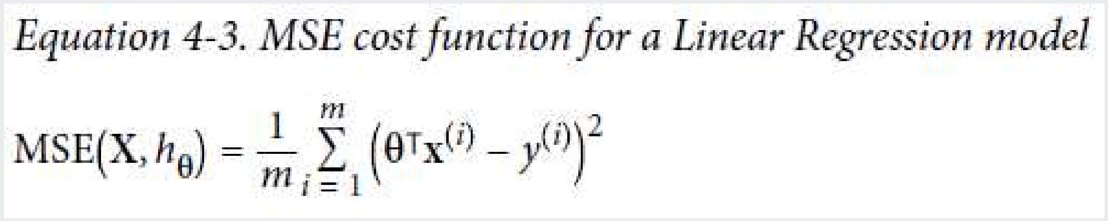


Equation 4-2. Linear Regression model prediction (vectorized form)

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

Linear Regression: how to solve for theta

- Using your training set X to find the set of theta that can minimize the MSE.

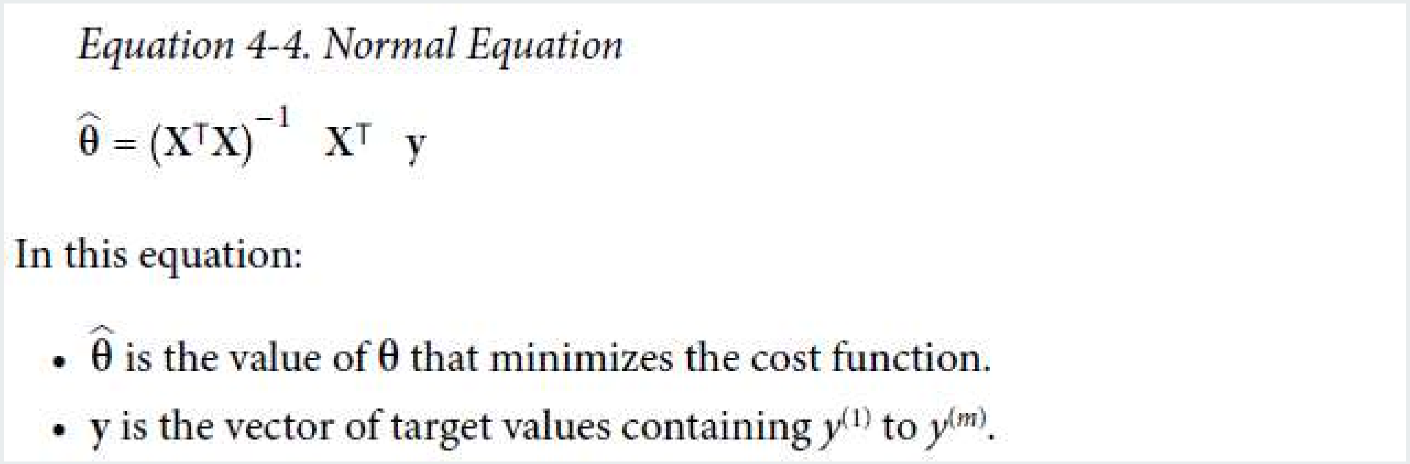


Equation 4-3. MSE cost function for a Linear Regression model

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

Linear Regression: the normal equation

- Leveraging optimization theory, you will be able to find closed-form solutions for the theta.



Equation 4-4. Normal Equation

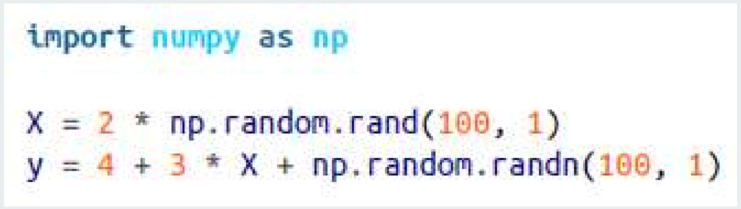
$$\hat{\theta} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \ \mathbf{X}^\intercal \ \mathbf{y}$$

In this equation:

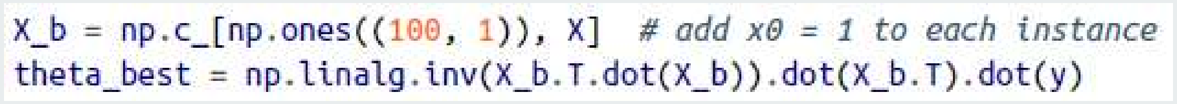
- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Linear Regression: the normal equation

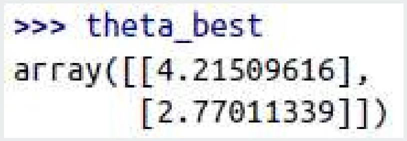
- Leveraging optimization theory, you will be able to find closed-form solutions for the theta.



```
import numpy as np

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



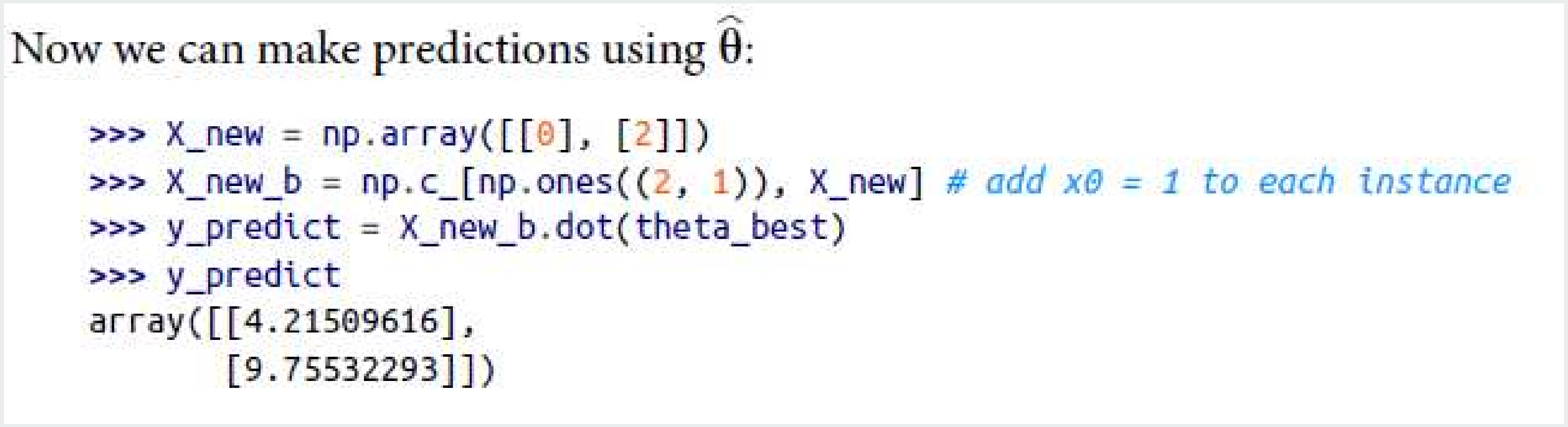
```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```



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

Linear Regression: prediction

- Prediction


Now we can make predictions using $\hat{\theta}$:

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

Linear Regression: plot the model's predictions

- Prediction

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

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

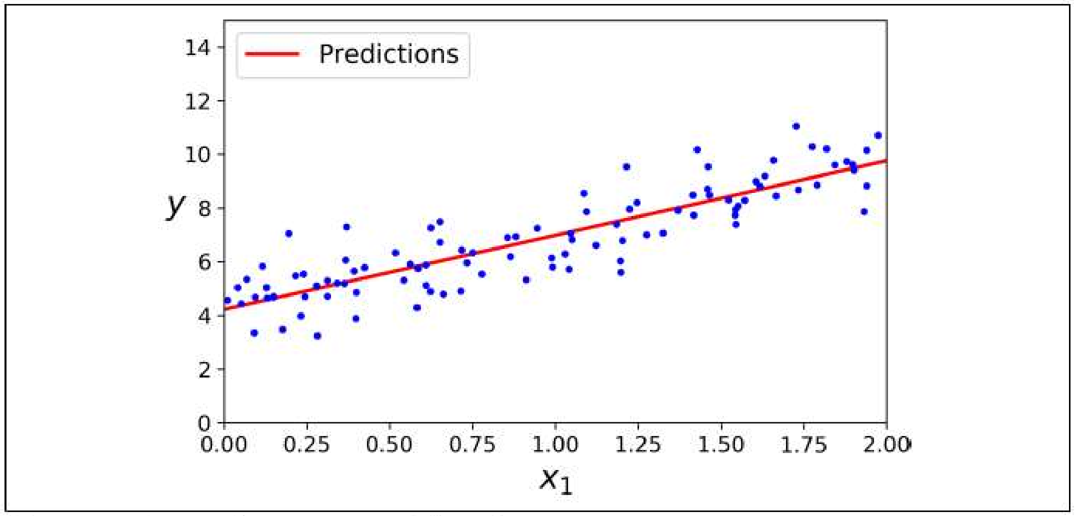
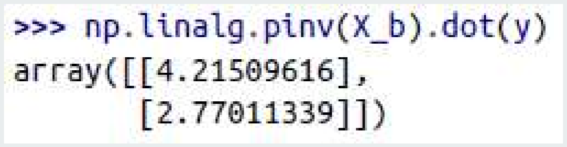


Figure 4-2. Linear Regression model predictions

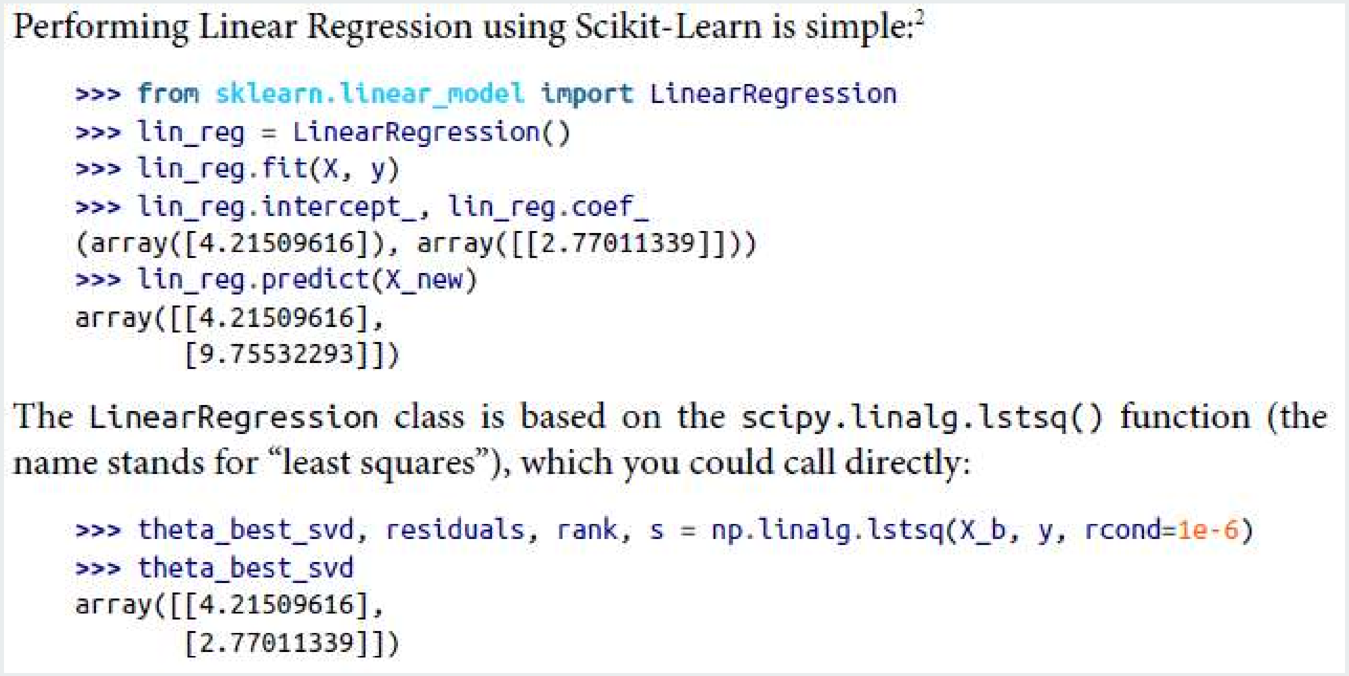
Scikit-Learn

$$\widehat{\theta} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y}$$

$$\widehat{\theta} = \mathbf{X}^+ \mathbf{y}$$



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



Performing Linear Regression using Scikit-Learn is simple:[2]

```
>>> 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]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

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

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

Linear Regression: plot the model’s predictions

- Prediction

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

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

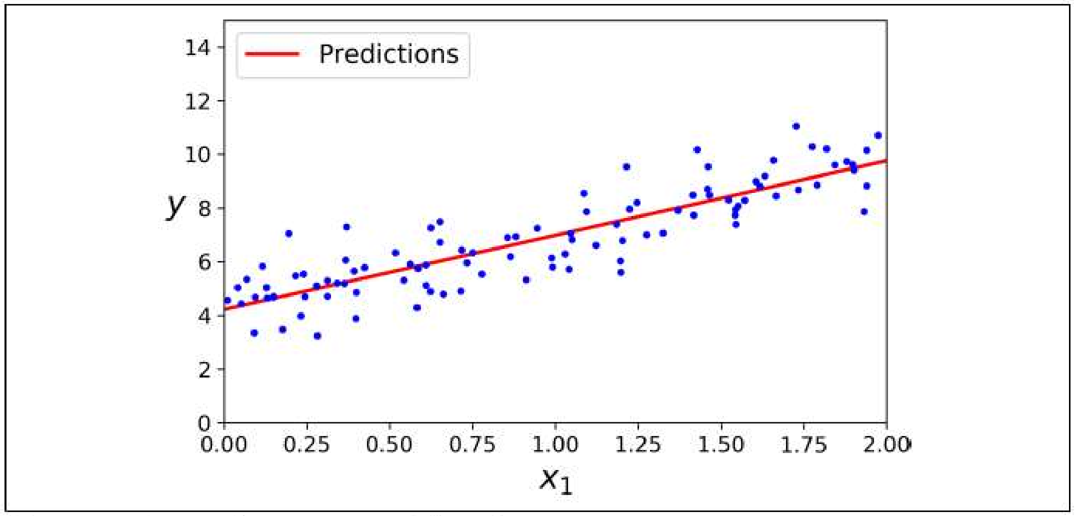


Figure 4-2. Linear Regression model predictions

Gradient Descent

- *Gradient Descent* is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems.

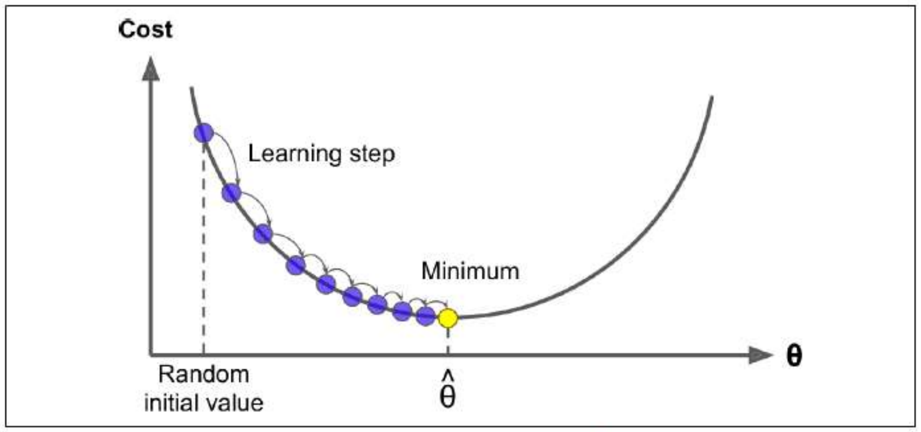


Figure 4-3. In this depiction of Gradient Descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the parameters approach the minimum

Gradient Descent: The learning rate

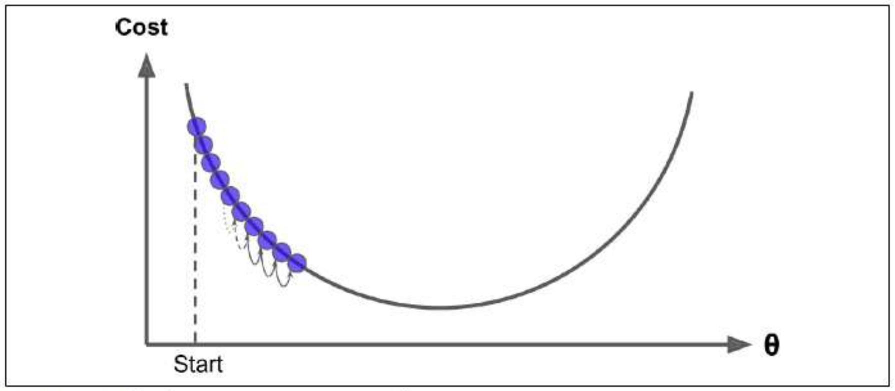


Figure 4-4. The learning rate is too small

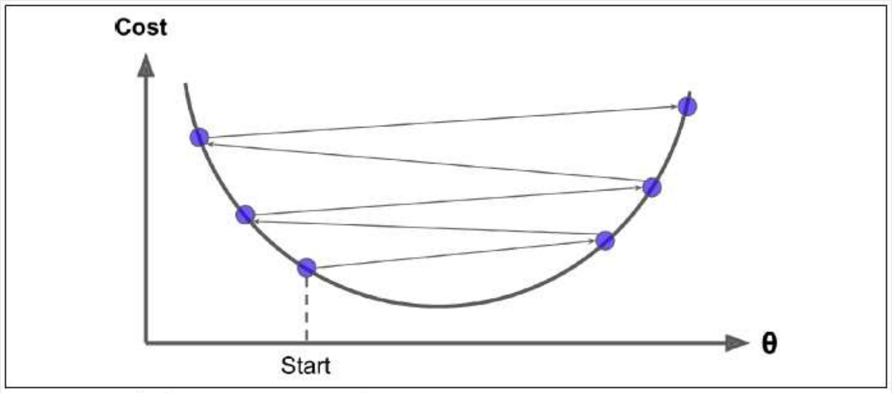


Figure 4-5. The learning rate is too large

Gradient Descent: Pitfalls

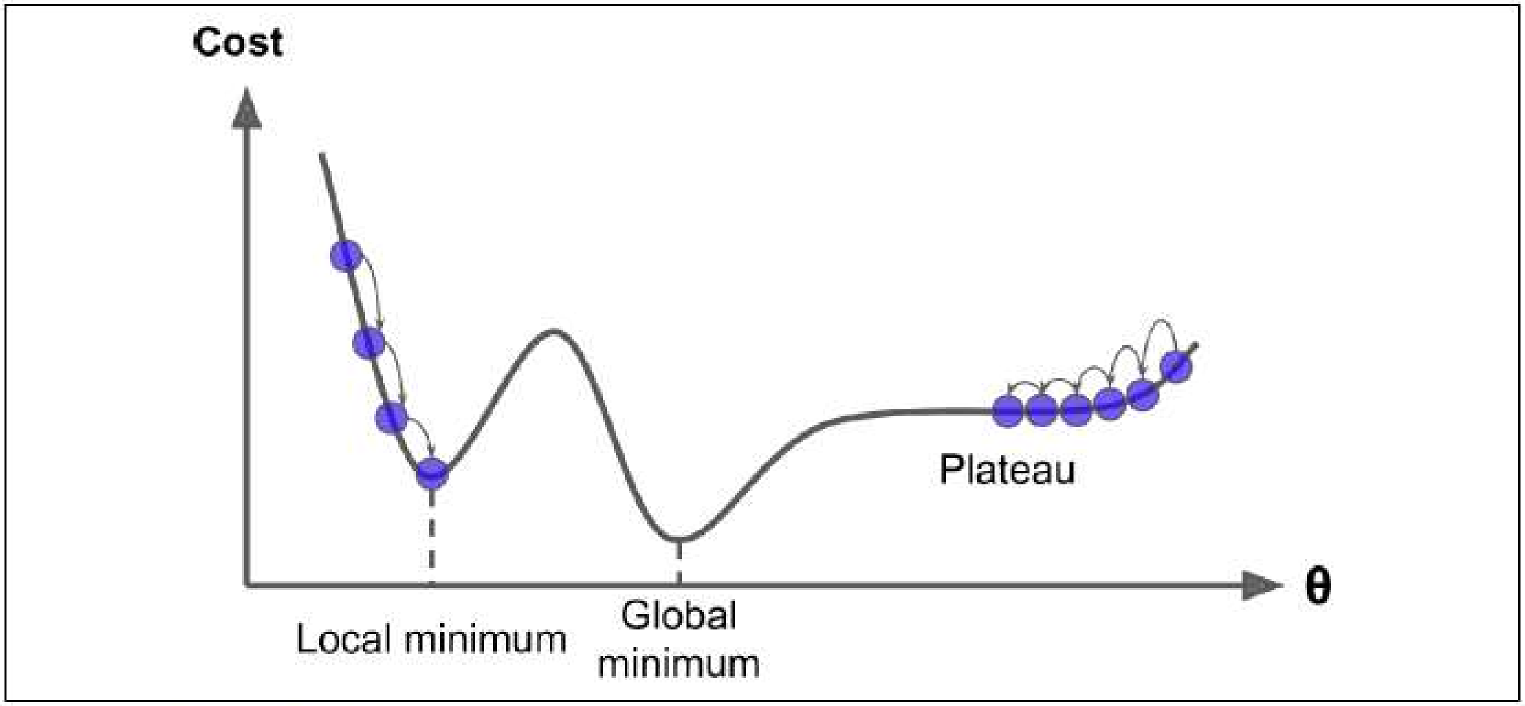


Figure 4-6. Gradient Descent pitfalls

Gradient Descent: Pitfalls

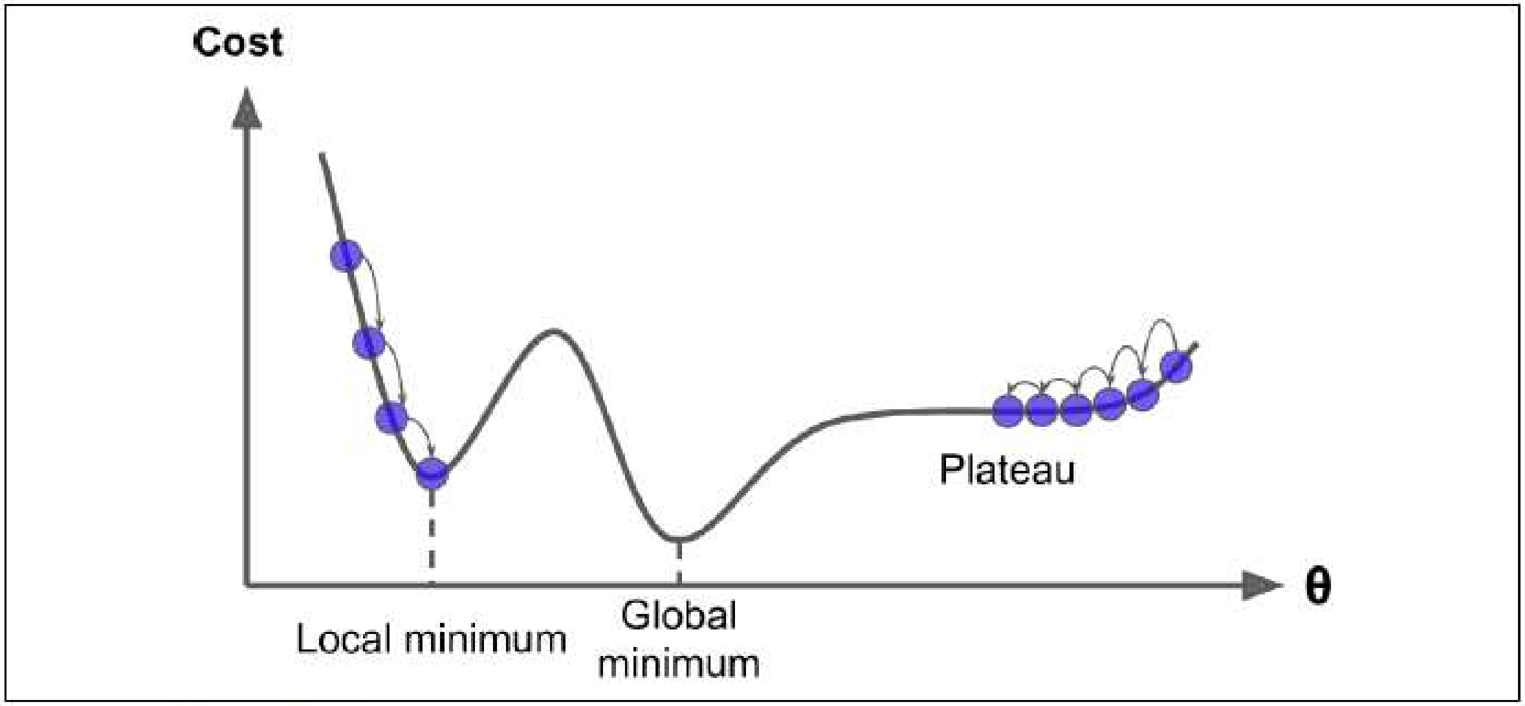


Figure 4-6. Gradient Descent pitfalls

Gradient Descent: Scaling made easy

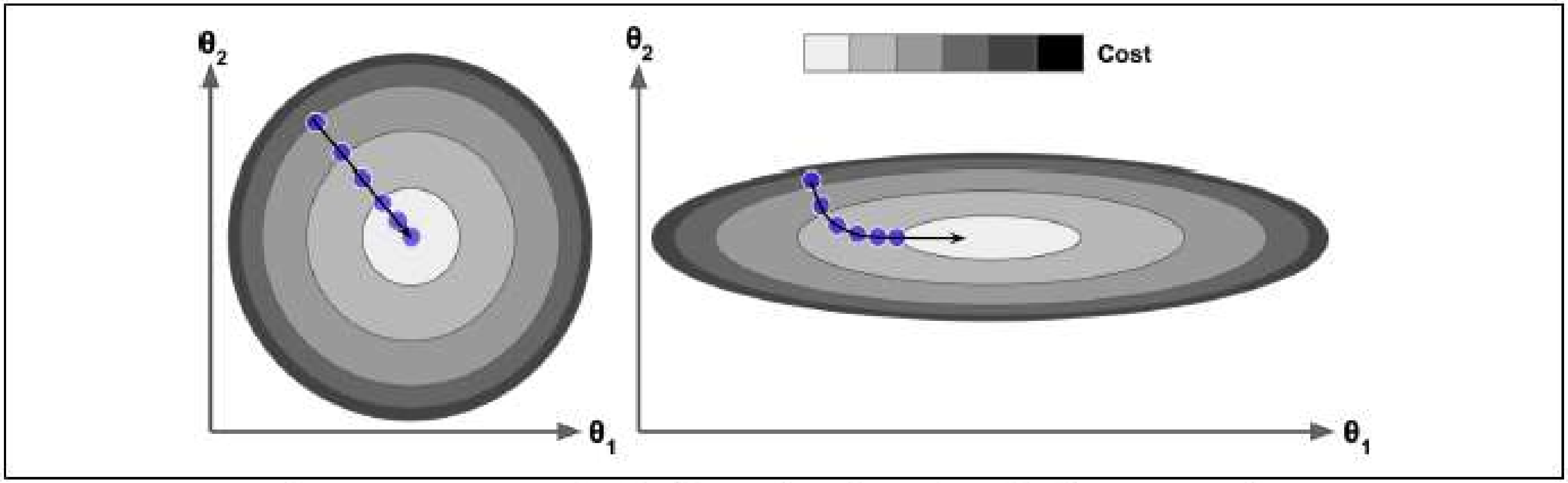


Figure 4-7. Gradient Descent with (left) and without (right) feature scaling

Batch Gradient Descent

- Partial derivative

- It is like asking “What is the slope of the mountain under my feet if I face east?” and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions).

Equation 4-5. Partial derivatives of the cost function

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

Equation 4-6. Gradient vector of the cost function

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

Batch Gradient Descent: Gradient vector

- Gradient vector

Equation 4-6. Gradient vector of the cost function

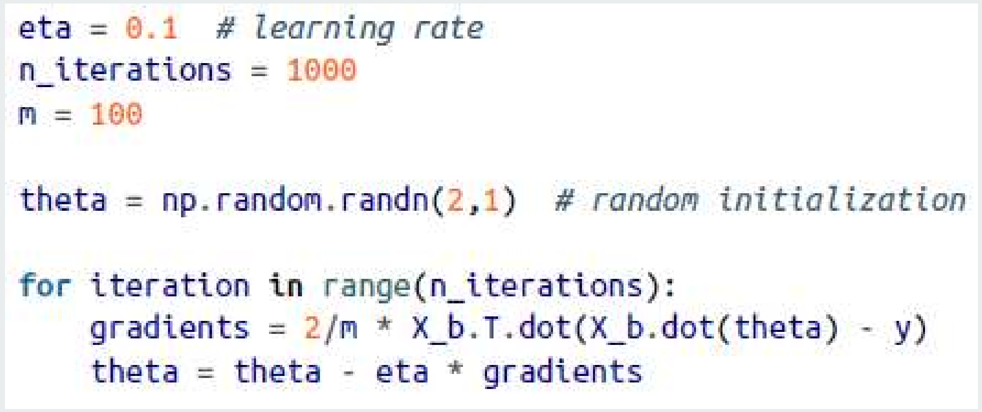
$$\nabla_{\boldsymbol{\theta}} \operatorname{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \operatorname{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \operatorname{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \operatorname{MSE}(\boldsymbol{\theta})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

- Learning rate

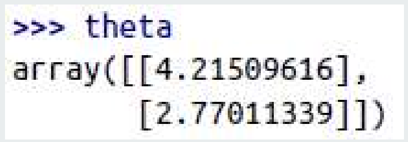
Equation 4-7. Gradient Descent step

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

Batch Gradient Descent: Python



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

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

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



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

- The SAME solutions as using normal equations (closed form solution)

Batch Gradient Descent: learning rate

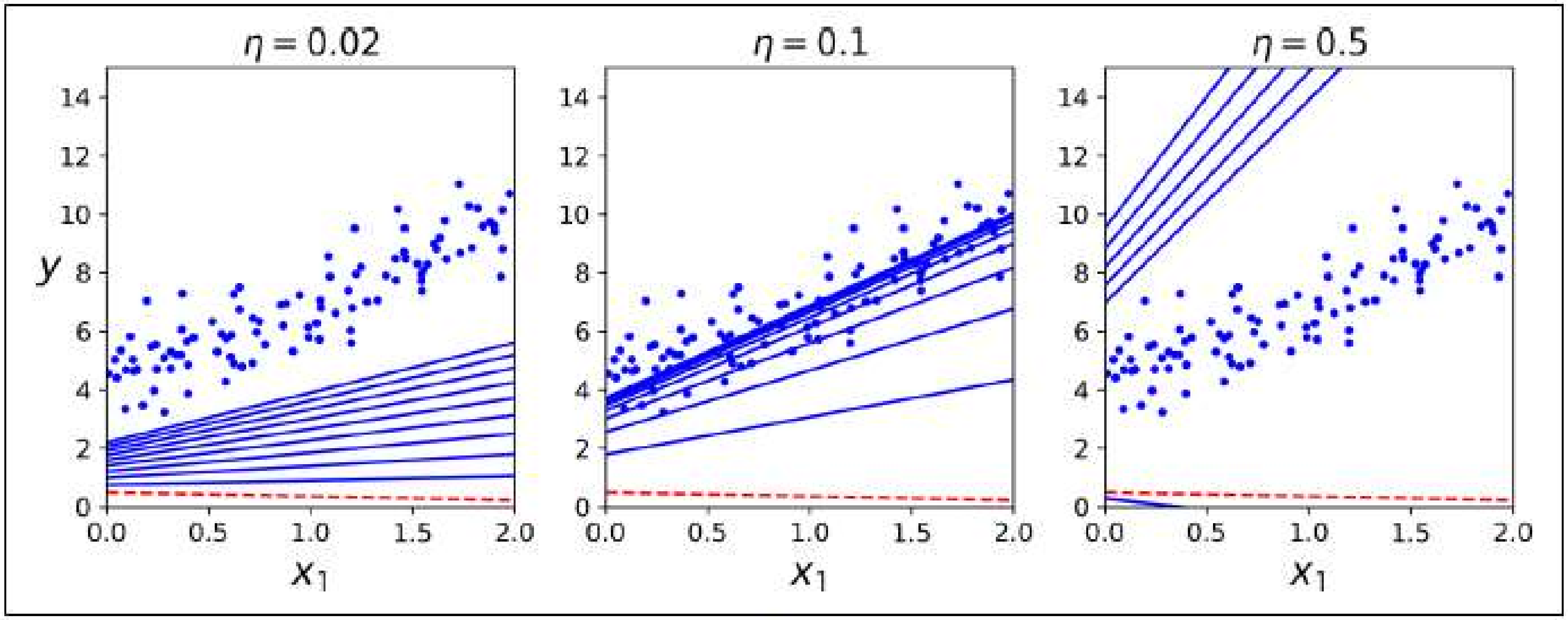


Figure 4-8. Gradient Descent with various learning rates

Stochastic Gradient Descent

- Stochastic gradient descent picks a random instance in the training set at every step and computes the gradients based only on a single instance.

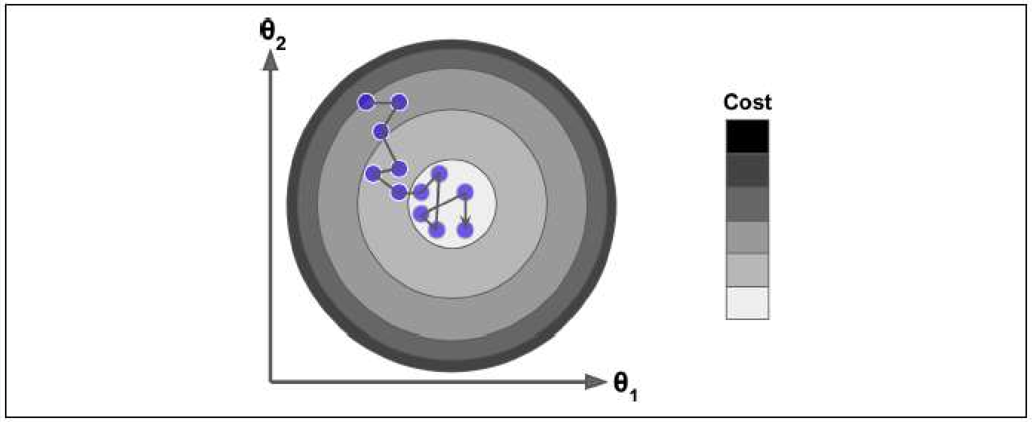
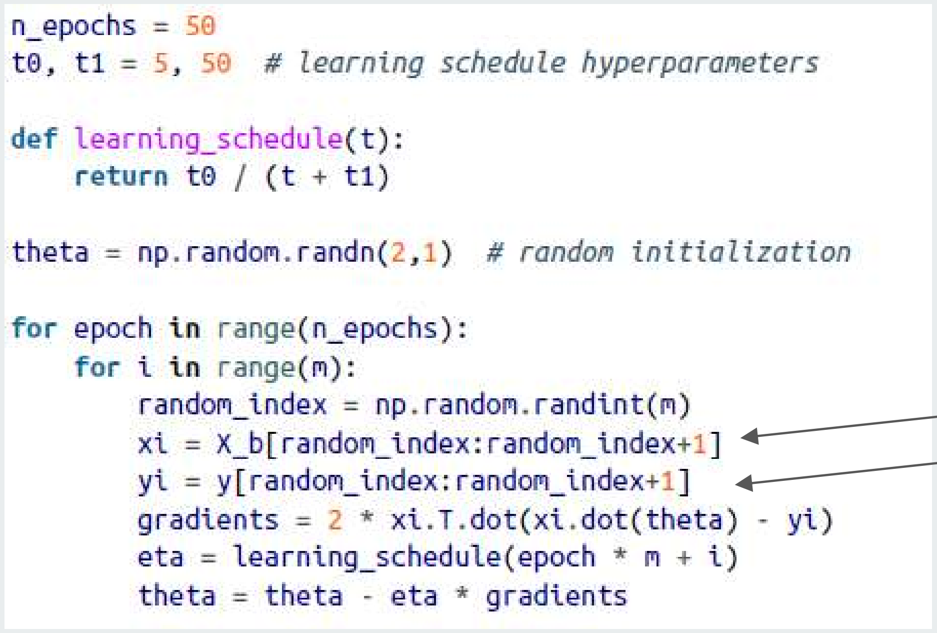


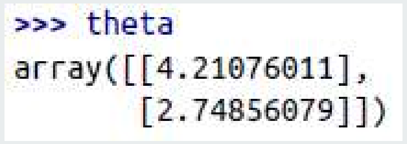
Figure 4-9. With Stochastic Gradient Descent, each training step is much faster but also much more stochastic than when using Batch Gradient Descent

Stochastic Gradient Descent: Python

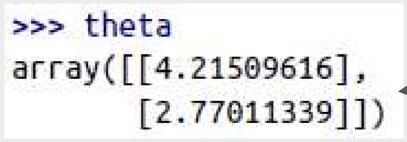


```python
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

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
```

- One random instance



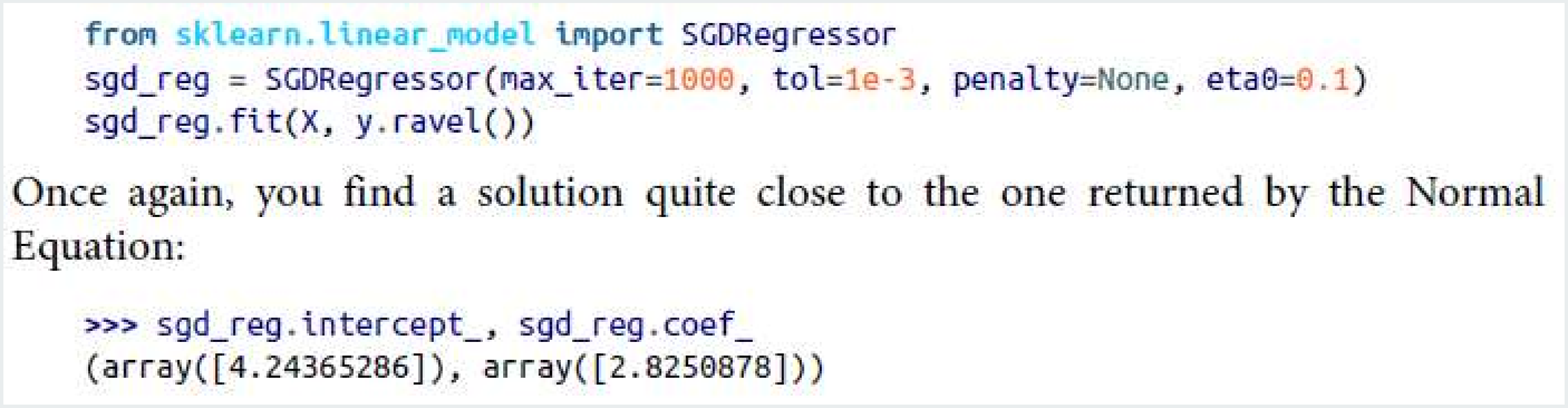
```
>>> theta
array([[4.21076011],
       [2.74856079]])
```



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

- Gradient descent

Stochastic Gradient Descent: Scikit-Learn



```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Once again, you find a solution quite close to the one returned by the Normal Equation:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

Mini-batch Gradient Descent

- It is 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*.

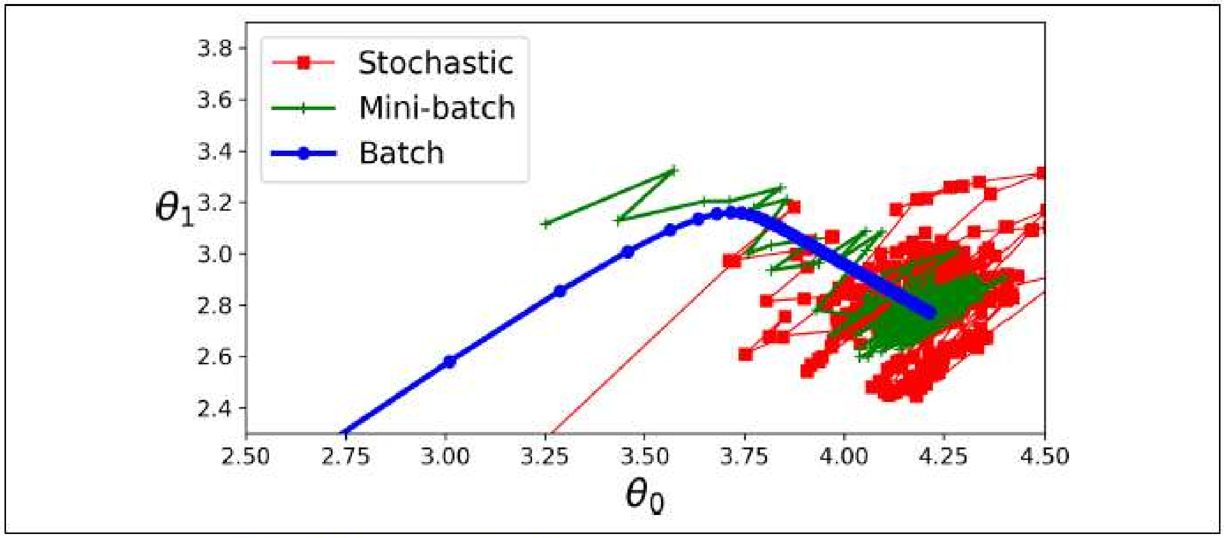


Figure 4-11. Gradient Descent paths in parameter space

Linear regression

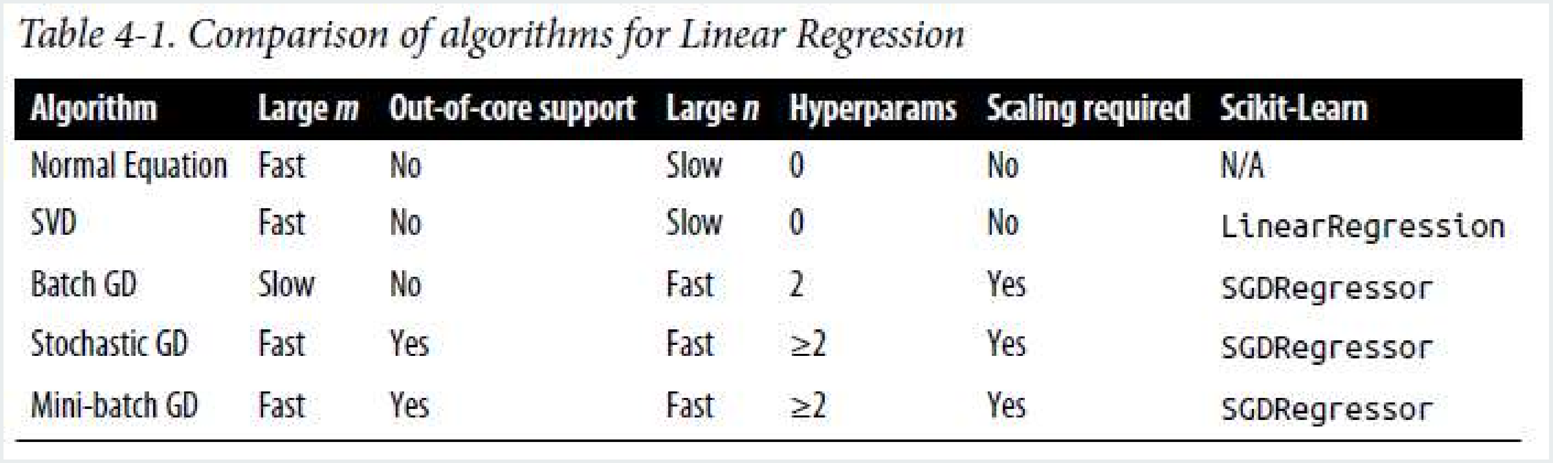
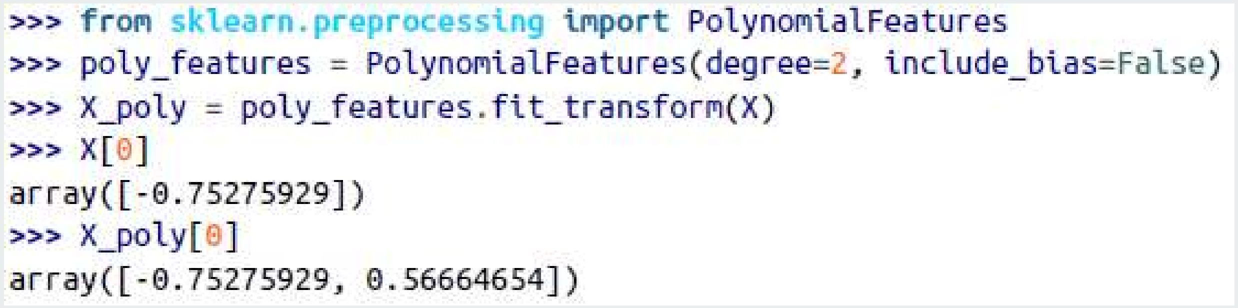


Table 4-1. Comparison of algorithms for Linear Regression

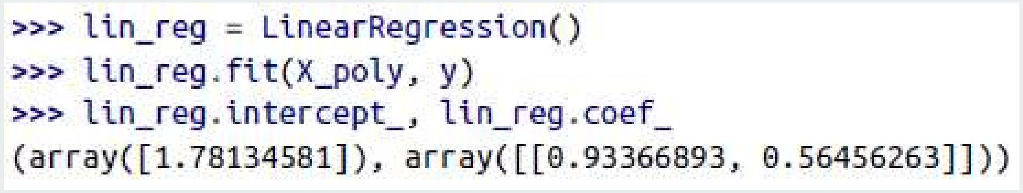
Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

Polynomial Regression

- Polynomial regression: you can use a linear model to fit nonlinear data



```
>>> 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])
```



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

Polynomial Regression

- Polynomial regression: you can use a linear model to fit nonlinear data

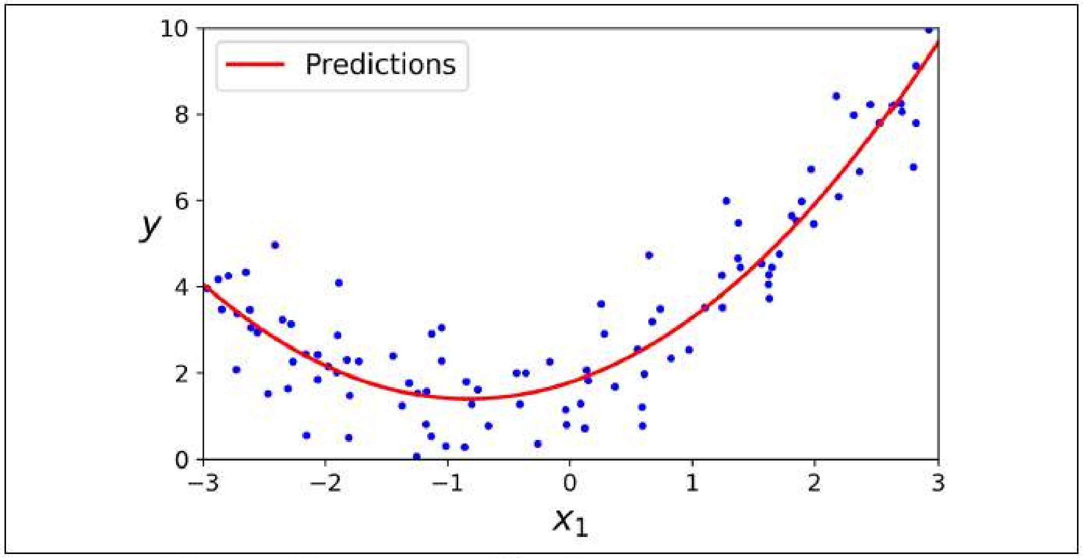


Figure 4-13. Polynomial Regression model predictions

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

Learning Curves

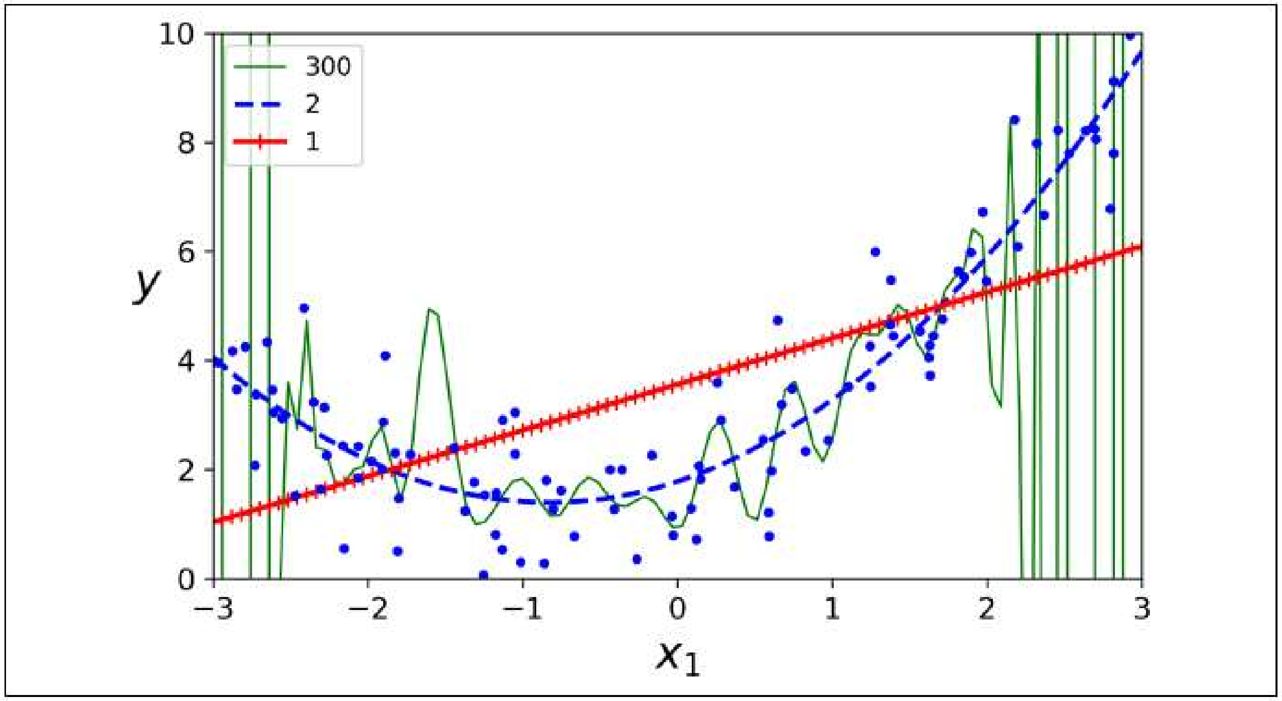


Figure 4-14. High-degree Polynomial Regression

Learning Curves

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

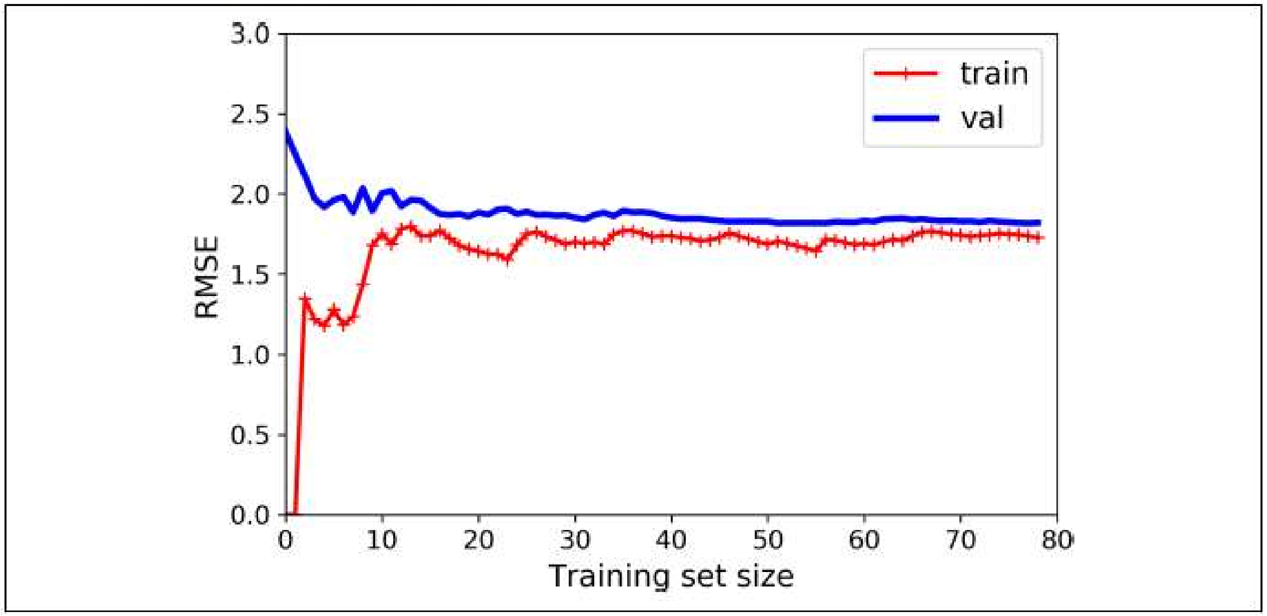


Figure 4-15. Learning curves

Learning Curves

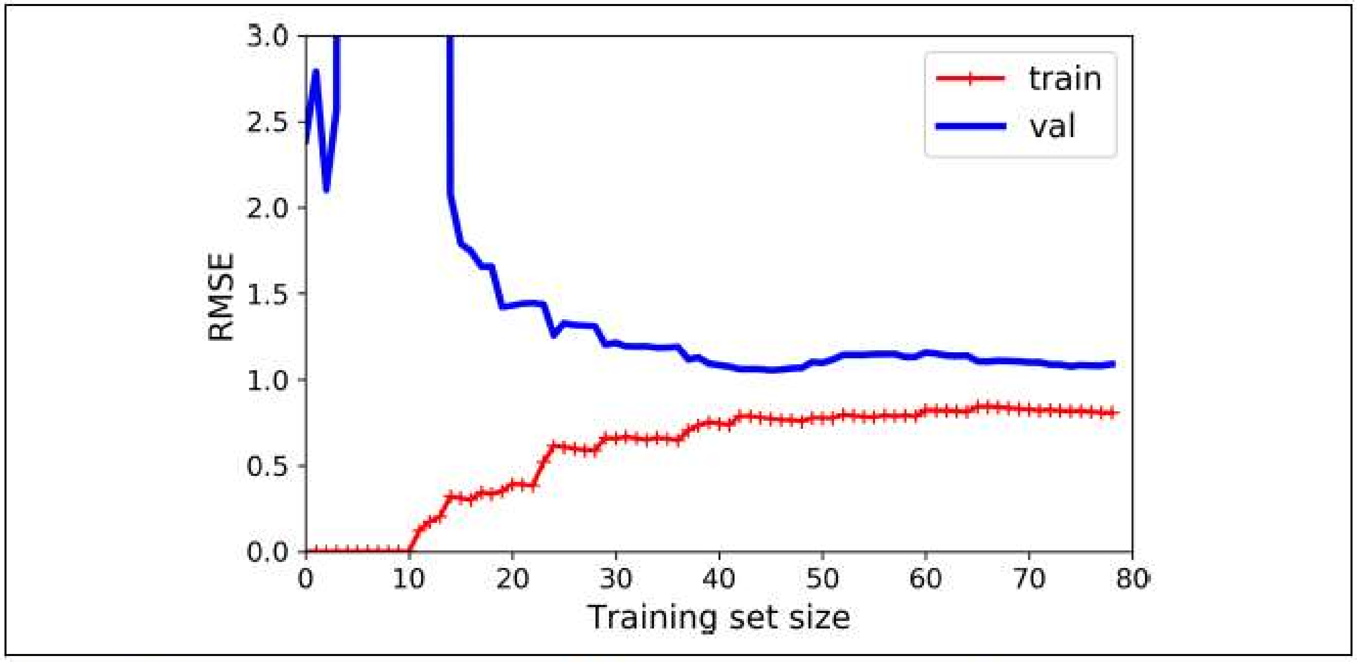
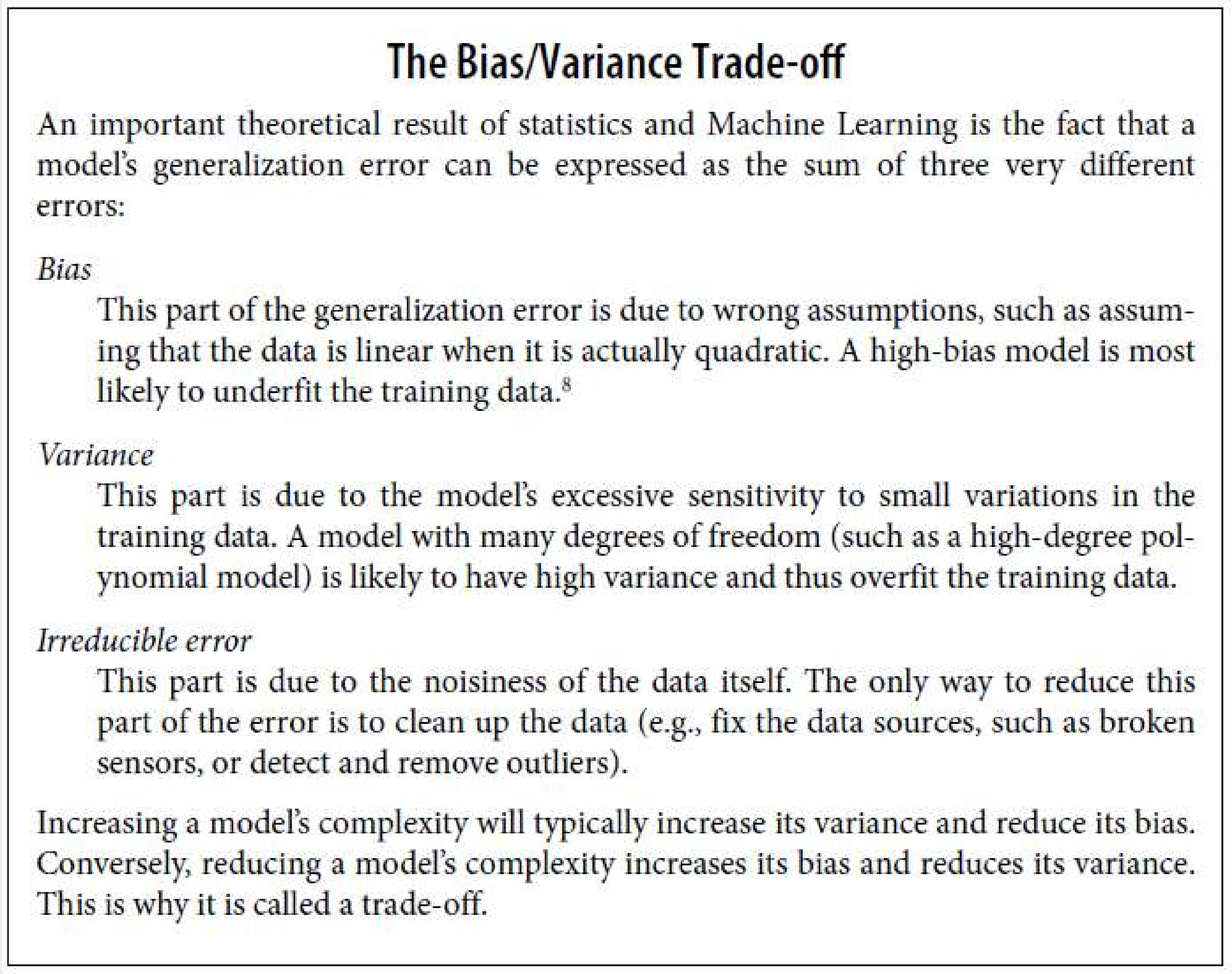


Figure 4-16. Learning curves for the 10th-degree polynomial model



The Bias/Variance Trade-off

An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

Bias

This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.[8]

Variance

This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.

Irreducible error

This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a trade-off.

Bias-Variance Tradeoff

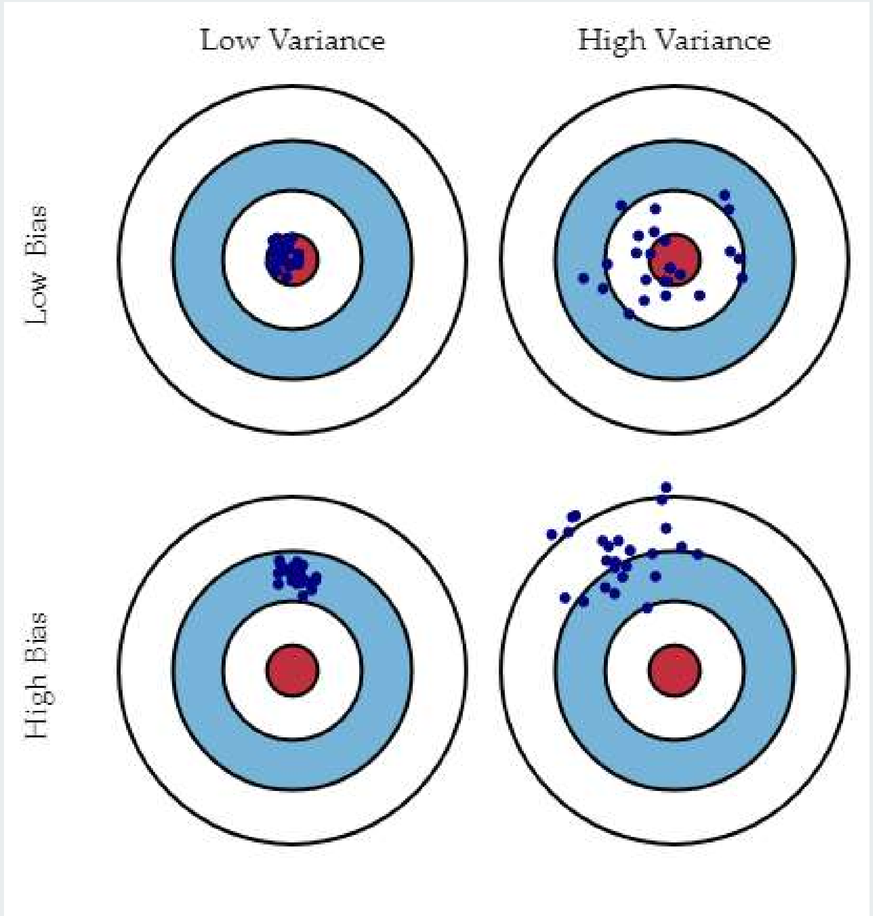


Fig. 1 Graphical illustration of bias and variance.

- http://scott.fortmann-roe.com/docs/BiasVariance.html

Bias-Variance Tradeoff

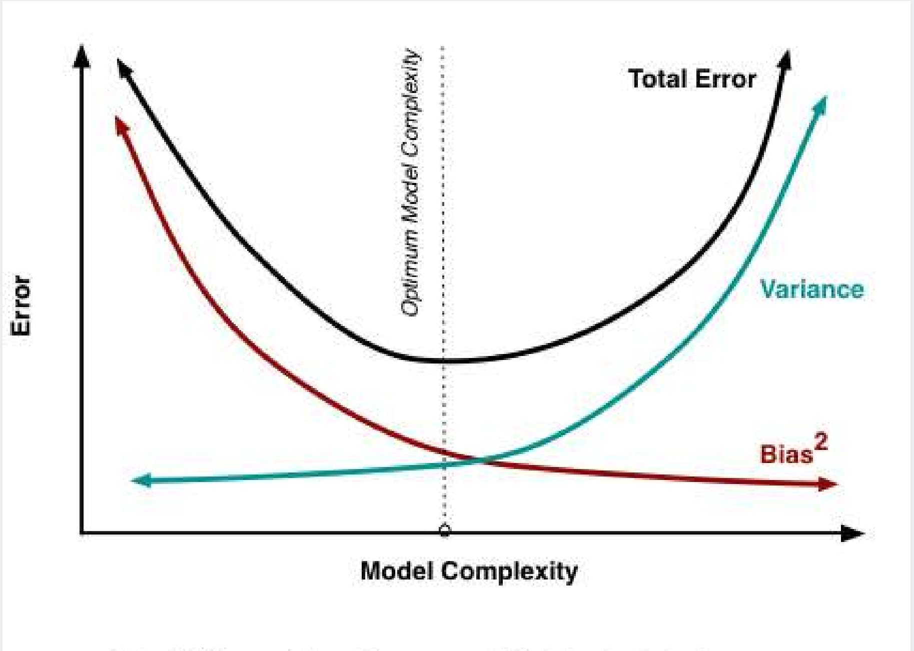


Fig. 6 Bias and variance contributing to total error.

- http://scott.fortmann-roe.com/docs/BiasVariance.html

Regularized Linear Models: Ridge Regression

- *Ridge Regression* is a regularized version of Linear Regression: a *regularization term* equal to is added $\alpha \Sigma_{i=1}^{n} \theta_i^2$ ost function (L2 norm).

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

Equation 4-8. Ridge Regression cost function

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

Regularized Linear Models: Ridge Regression

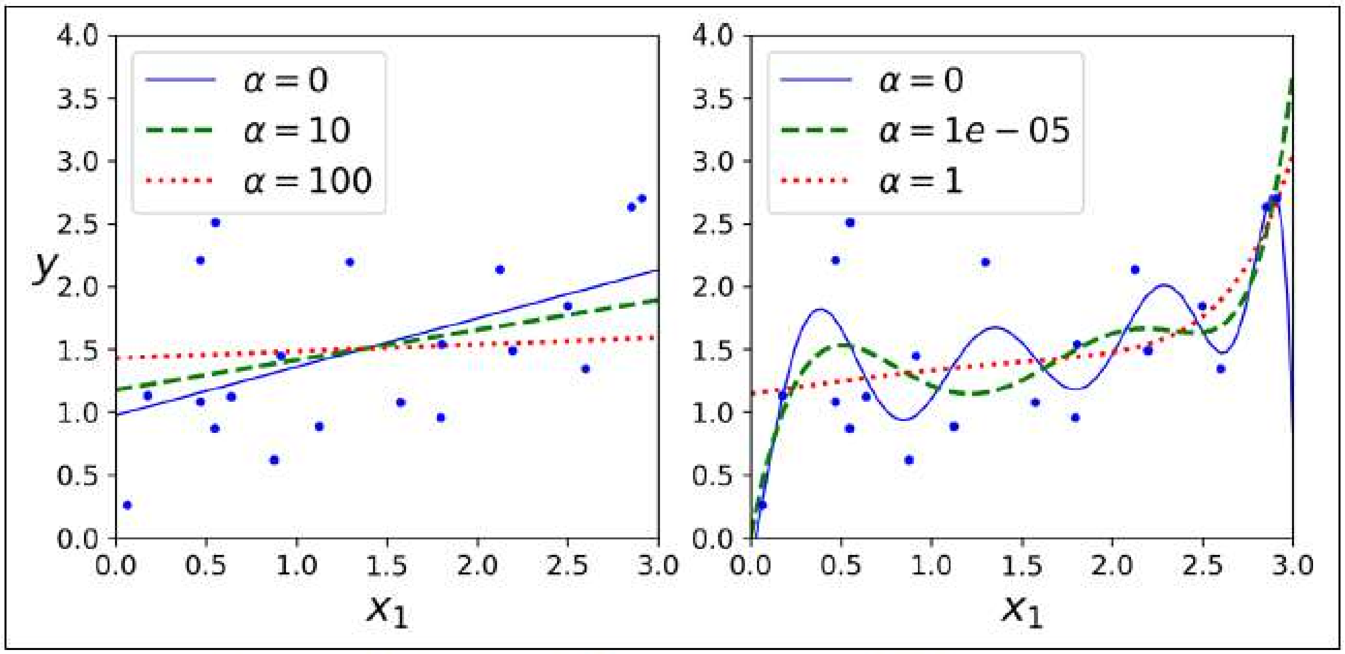
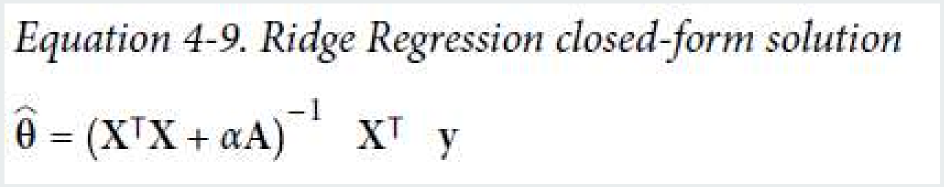
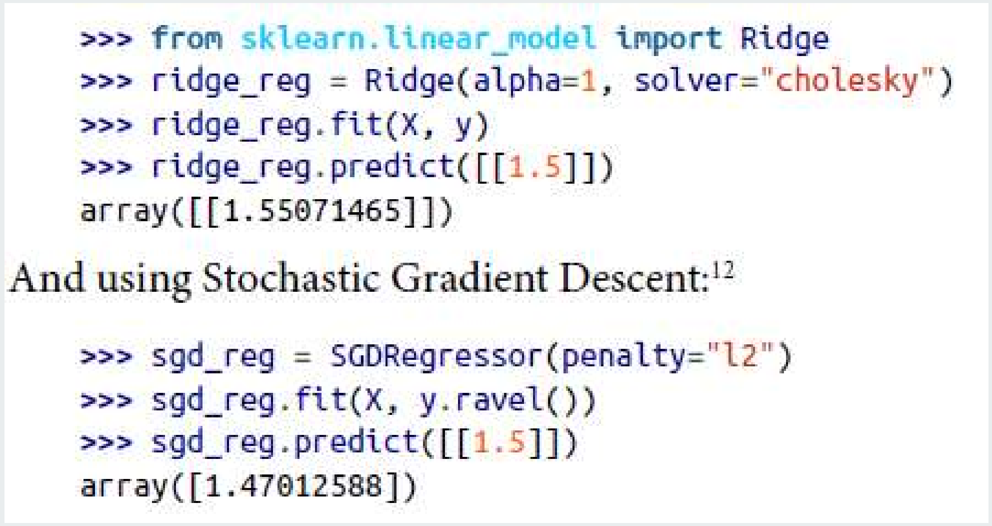


Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

Regularized Linear Models: Ridge Regression


Equation 4-9. Ridge Regression closed-form solution

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


```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

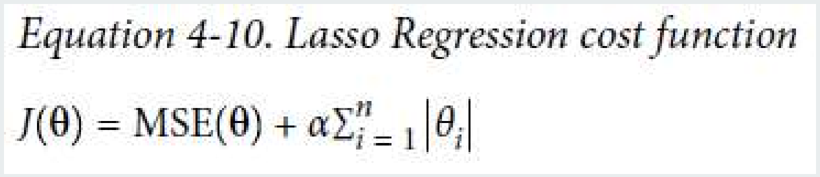
And using Stochastic Gradient Descent:[12]

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

A is the (n + 1) × (n + 1) *identity matrix*

Regularized Linear Models: Lasso Regression

○ *Least Absolute Shrinkage and Selection Operator Regression* (usually simply called *Lasso Regression*) is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, it is L1 norm of the weight vector.



Equation 4-10. Lasso Regression cost function

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \Sigma_{i=1}^{n} \left|\theta_i\right|$$

Regularized Linear Models: Lasso Regression

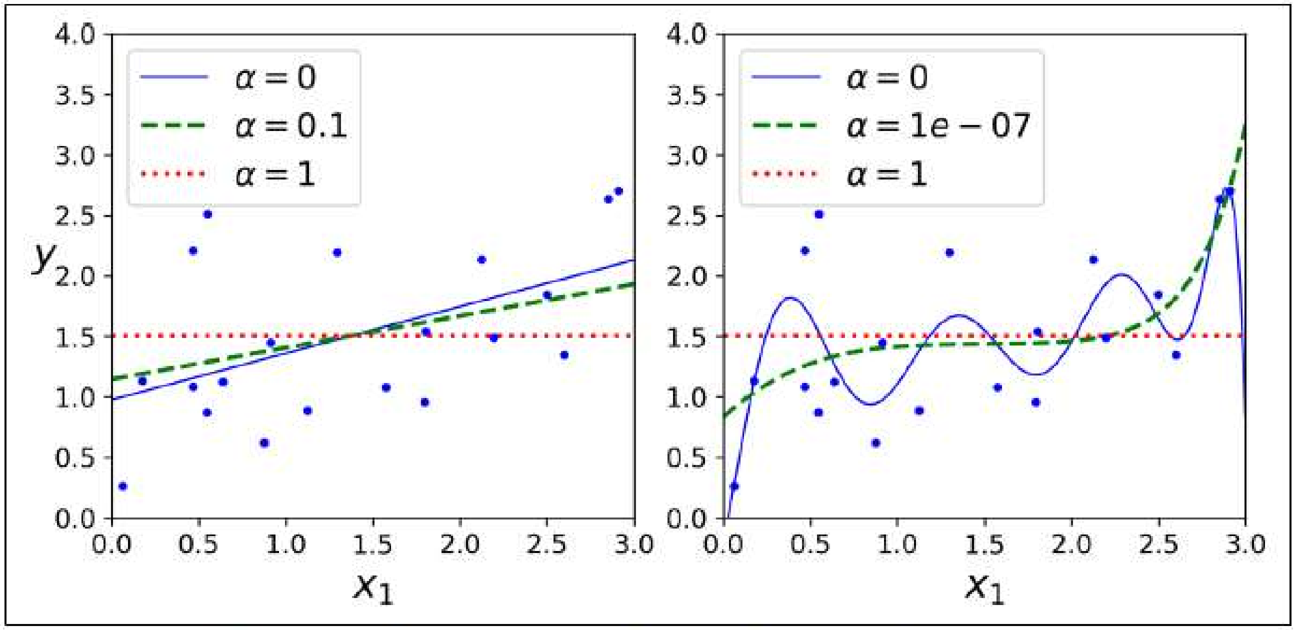
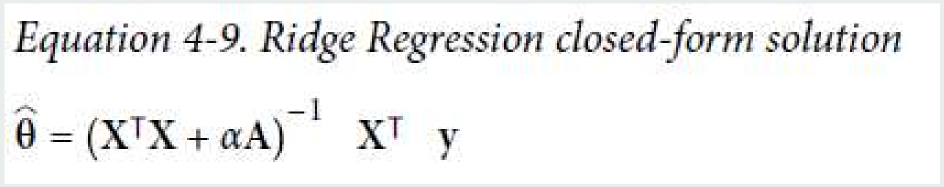


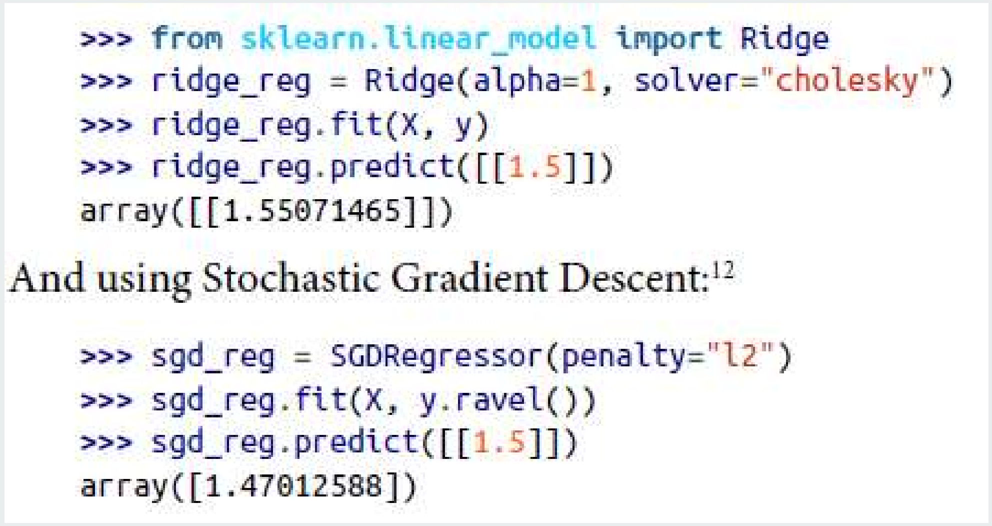
Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

Regularized Linear Models: Ridge Regression


Equation 4-9. Ridge Regression closed-form solution

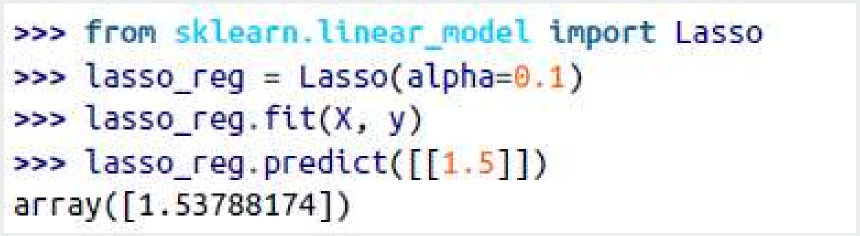
$$\hat{\theta} = \left(\mathbf{X}^{\intercal}\mathbf{X} + \alpha\mathbf{A}\right)^{-1} \quad \mathbf{X}^{\intercal} \quad \mathbf{y}$$

A is the $(n + 1) \times (n + 1)$ *identity matrix*


```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

And using Stochastic Gradient Descent:[12]

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```


```
>>> 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])
```

Regularized Linear Models

$\alpha\sum_{i=1}^{n}|\theta_i|$

$\alpha\sum_{i=1}^{n}\theta_i^2$

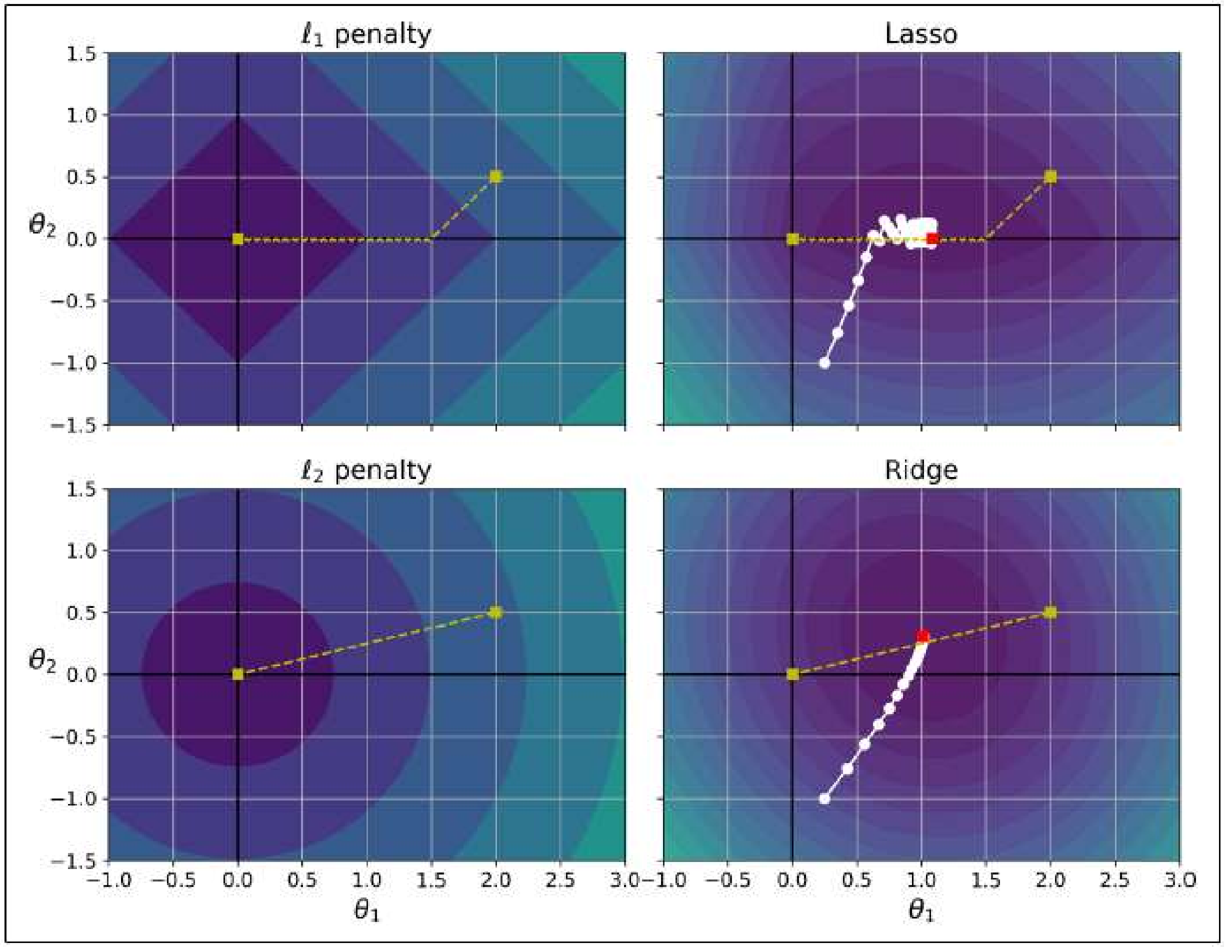
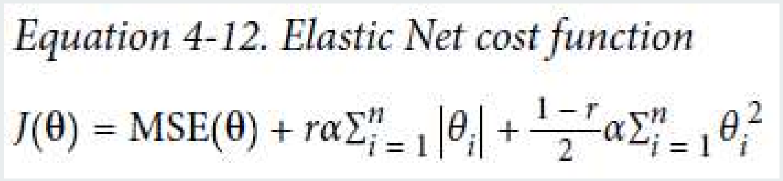


Figure 4-19. Lasso versus Ridge regularization

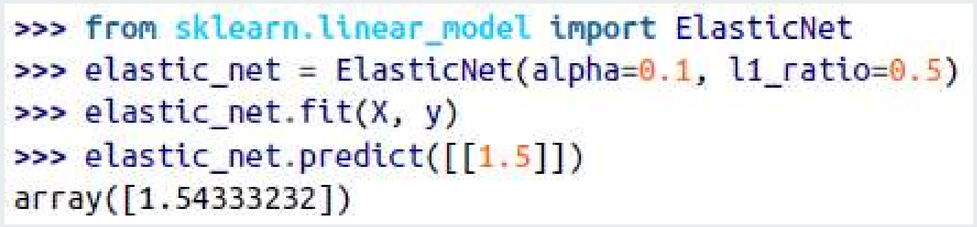
Regularized Linear Models: Elastic Net

- Elastic Net is a middle ground between Ridge Regression and Lasso Regression.



Equation 4-12. Elastic Net cost function

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



```
>>> 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])
```

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*.

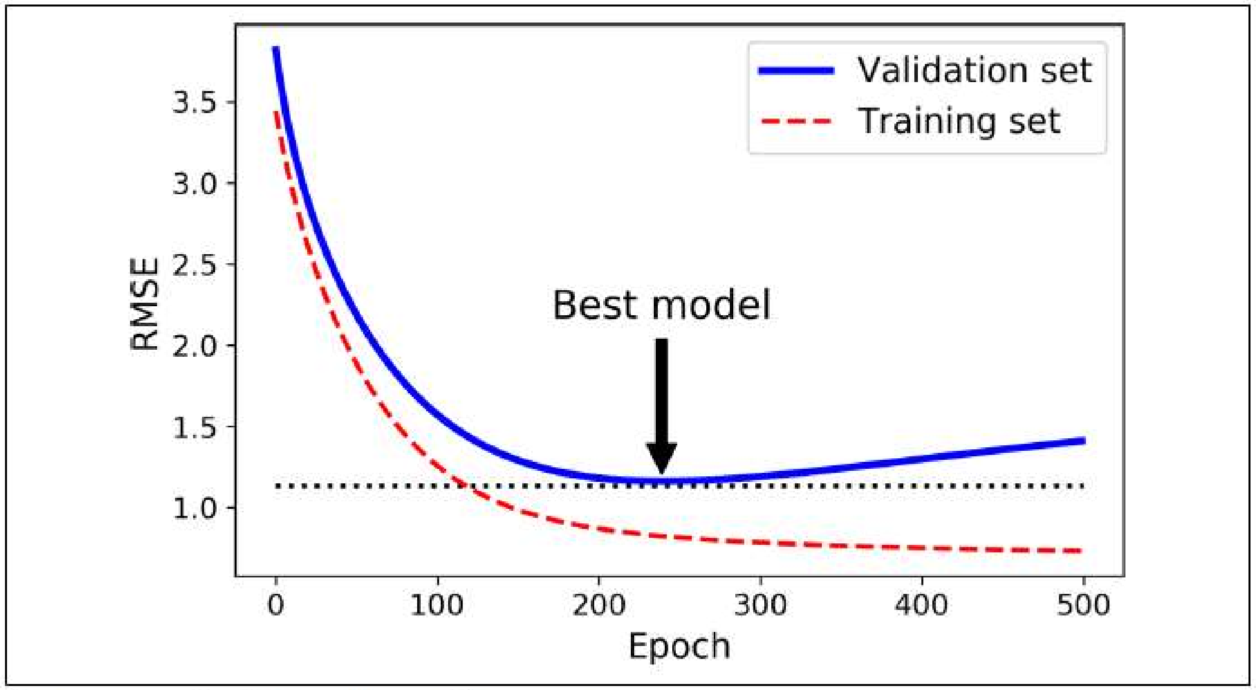
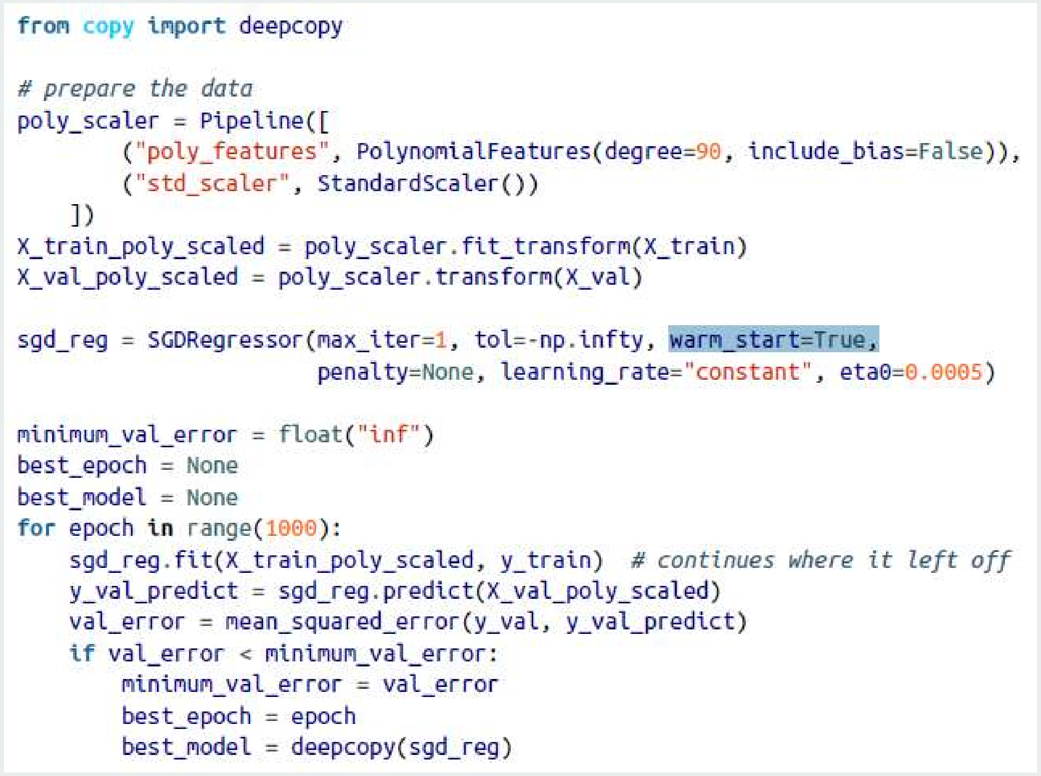


Figure 4-20. Early stopping regularization

Regularized Linear Models: Early Stopping



```python
from copy import deepcopy

# prepare the data
poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
    ])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = deepcopy(sgd_reg)
```

Logistic Regression

- *Logistic Regression* (also called *Logit Regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g.,what is the probability that this email is spam?).

- If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the *positive class*, labeled “1”), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled “0”). This makes it a binary classifier.

Logistic Regression: Estimating Probabilities

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

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

Equation 4-14. Logistic function

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

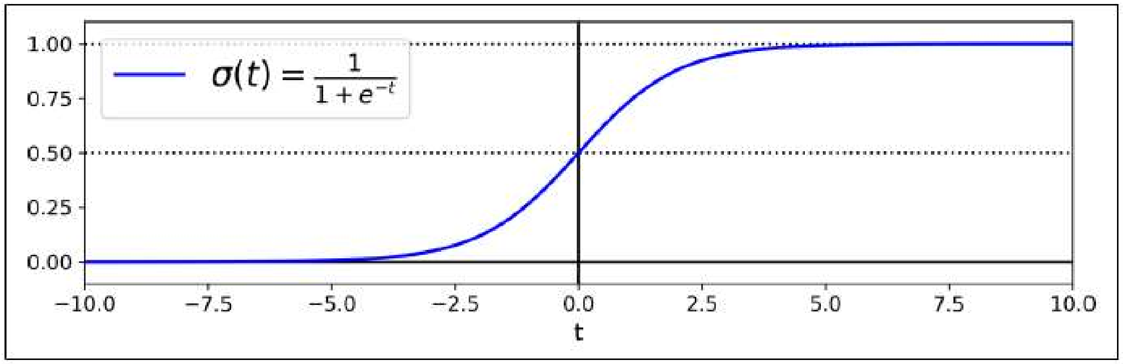


Figure 4-21. Logistic function

Logistic Regression: Cost Function

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

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

- The bad news is that there is no known closed-form equation to compute the value of θ that minimizes this cost function (there is no equivalent of the Normal Equation).

- The good news is that this cost function is convex, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough).

Logistic Regression: Decision Boundaries

- Iris flower dataset: 150 iris flowers with three difference species: Iris setosa, Iris versicolor, and Iris virginica
- Petal length, petal width, sepal length, and sepal width

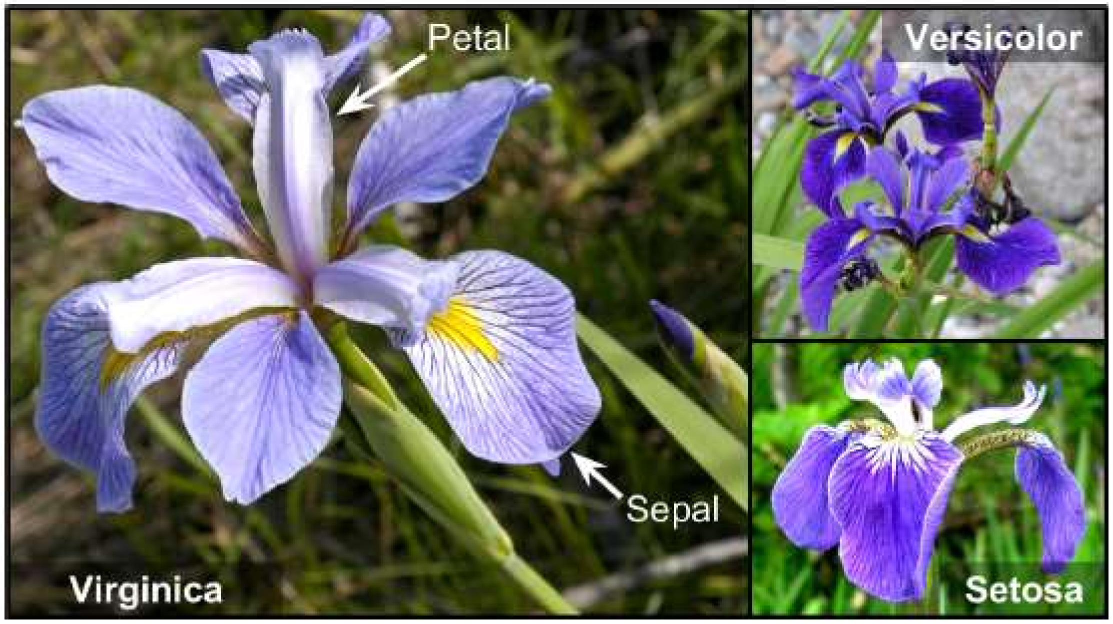


Figure 4-22. Flowers of three iris plant species[14]

Logistic Regression: Decision Boundaries

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

```
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-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris virginica")
# + more Matplotlib code to make the image look pretty
```

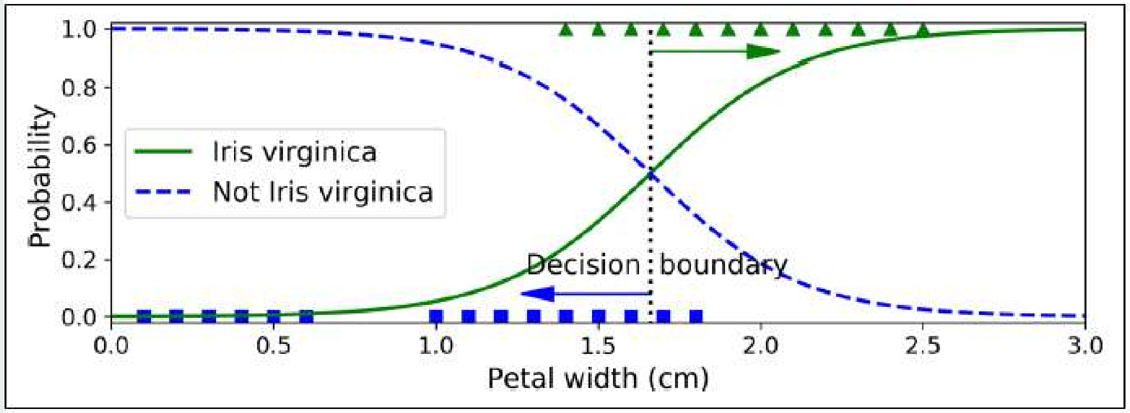


Figure 4-23. Estimated probabilities and decision boundary

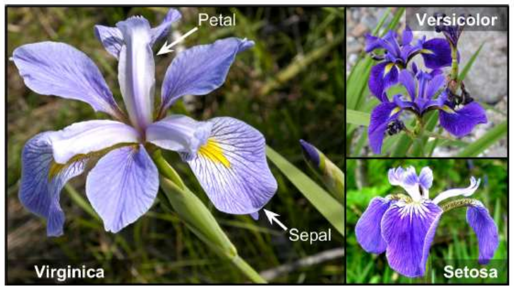


Figure 4-22. Flowers of three iris plant species[14]

Logistic Regression: Decision Boundaries

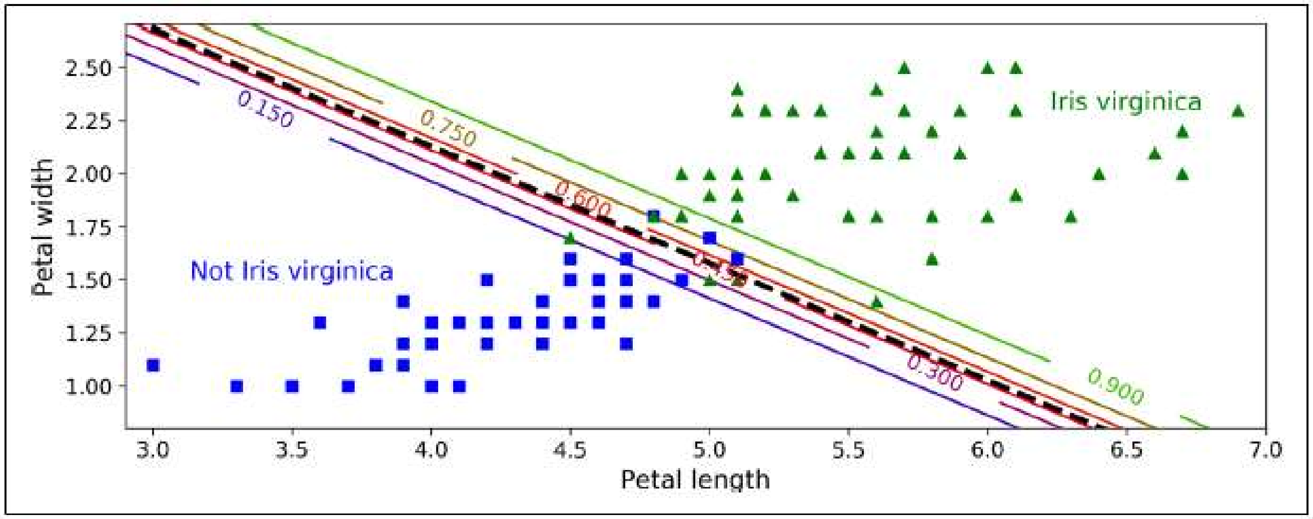


Figure 4-24. Linear decision boundary

Just like the other linear models, Logistic Regression models can be regularized using ℓ_1 or ℓ_2 penalties. Scikit-Learn actually adds an ℓ_2 penalty by default.

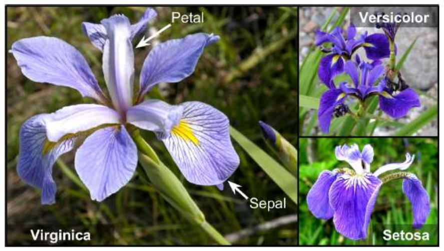


Figure 4-22. Flowers of three iris plant species[14]

Softmax Regression

- The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers. This is called *Softmax Regression*, or *Multinomial Logistic Regression*.

- The idea is simple: when given an instance **x**, the Softmax Regression model first computes a score *sk*(**x**) for each class *k*, then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores.

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = \left(\theta^{(k)}\right)^\intercal \mathbf{x}$$

Equation 4-20. Softmax function

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

Softmax Regression

- The Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score)

Equation 4-21. Softmax Regression classifier prediction

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

- The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes).

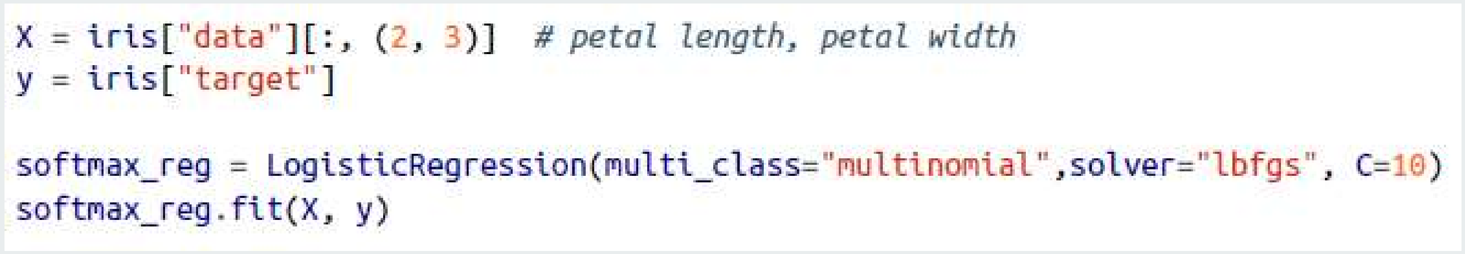
Equation 4-22. Cross entropy cost function

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

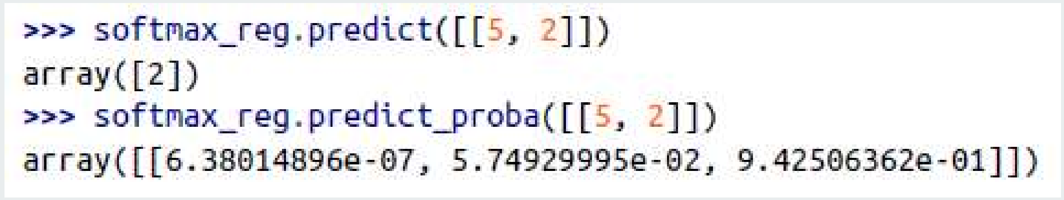
Equation 4-23. Cross entropy gradient vector for class k

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

Softmax Regression: Scikit-Learn



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

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



```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

Softmax Regression: Decision Boundaries

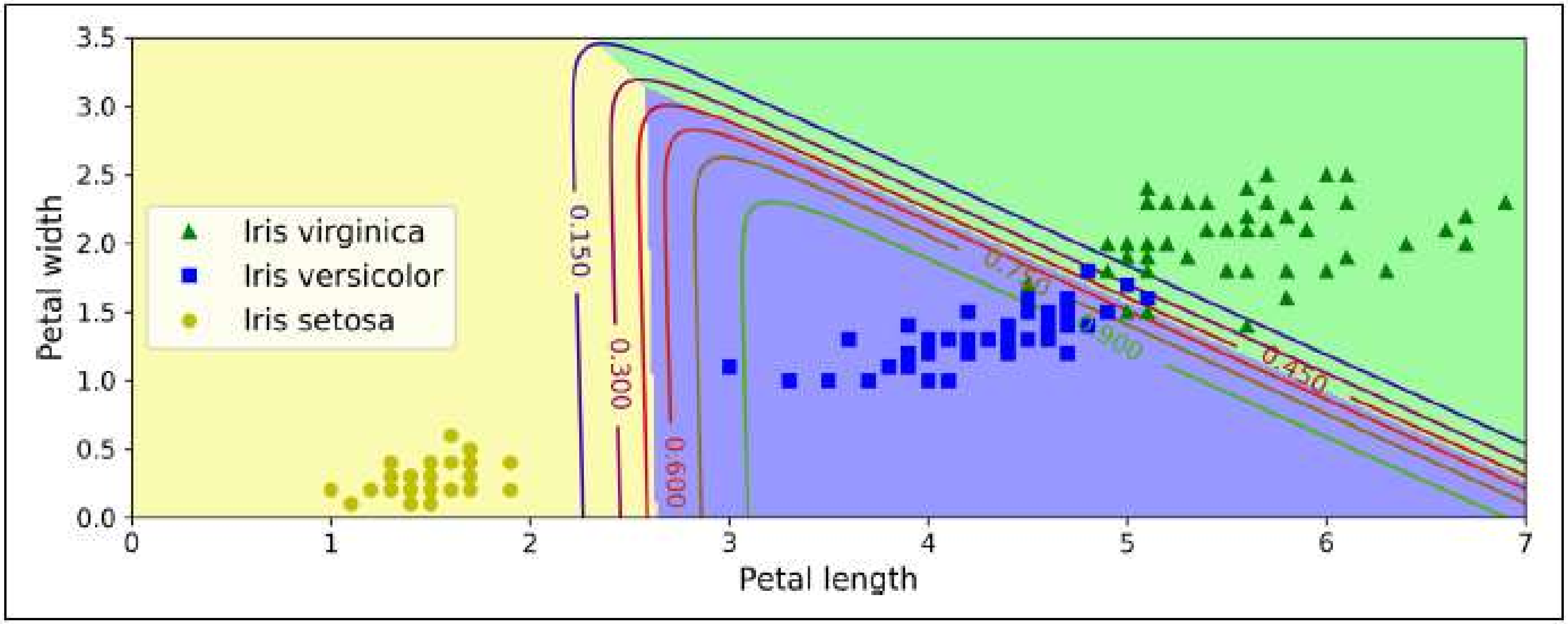


Figure 4-25. Softmax Regression decision boundaries