Chapter_4_Training_Model

August 29, 2020

Chapter 4 – Training Linear Models

Before we used ML models without understanding how they work. Now we will try to look under the hood. We will look at two versions of Linear regression: regular closed-form equation $\beta = \frac{X'Y}{X'X}$ and iterative optimization Gradient Descent (GD) approach which tweaks model parameters to minimize cost function. Even though eventually GD will converge to the parameters β estimated by the first method, GD is the important concepts that will help us to study neural networks.

Next we will look at the polynomial regression, which is more flexible than linear regression. It has more parameters and more prone to overfitting. Finally, we will look at regression models used in classification: Logistic Regression and Softmax Regression.

In Chapter 1, we looked at simple regression: $life\_satisfaction = \theta_0 + \theta_1 \times GDP\_percapita$ This is a linear function of the input feature of one input feature. The model parameters are θ_0 and θ_1. In general regression has the following form:

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

* $\hat{y}$ is the predicted value * n is the number of features * x_i is the i^{th} feature * θ_j is the j^{th} model parameters. Number of parameters is $n+1$ because of the constant

We can write linear regression in the vector form:

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

- θ^T is the transpose of vector of parameters or feature weights θ.
- $\theta^T \cdot x$ is the dot product of vector of parameters θ^T and vector of features x.
- $h_\theta(x)$ is the hypothesis function or prediction using parameters θ

First we train linear regression on training data minimizing Mean Square Error (MSE). Usually we use Root Mean Square Error (RMSE) as a metric to compare OLS models, the same set of parameters minimize both MSE and RMSE.

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

Later we will refer to h_θ as just h, and $MSE(\theta)$ instead of $MSE(X, h_\theta)$

Normal regression find θ that minimizes MSE:

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

* $\hat{\theta}$ is the θ that minimizes MSE * $\boldsymbol{\theta}$ is the vector of target values $y^{(1)}$ to $y^{(m)}$.

Note: the first releases of the book implied that the `LinearRegression` class was based on the Normal Equation. This was an error, my apologies: as explained above, it is based on the pseudoinverse, which ultimately relies on the SVD matrix decomposition of $\mathbf{X}$ (see chapter 8 for details about the SVD decomposition). Its time complexity is $O(n^2)$ and it works even when $m < n$ or when some features are linear combinations of other features (in these cases, $\mathbf{X}^T\mathbf{X}$ is not invertible so the Normal Equation fails), see issue #184 for more details. However, this does not change the rest of the description of the `LinearRegression` class, in particular, it is based on an analytical solution, it does not scale well with the number of features, it scales linearly with the number of instances, all the data must fit in memory, it does not require feature scaling and the order of the instances in the training set does not matter.

There is a mistake in the book: the LinearRegression class does not actually use the Normal Equation, it computes the pseudoinverse of X (specifically the Moore-Penrose pseudoinverse), and it multiplies it by y. It gives the same result as the normal equation, but it has two important advantages:

It is $O(n^2)$ instead of $O(n^2.4)$ to $O(n^3)$. So it's much faster than the Normal Equation when there are many features. It behaves better when some of the eigenvalues of X are small (or equal to zero). In plain English, this is when some features are highly correlated (or perfectly correlated, that is when one feature is a linear combination of the others). It also behaves well when m < n. More details below.

When some features are linear combinations of the others, or when m < n, the matrix X^T, X is not invertible (it is said to be "singular" or "degenerate"). So the Normal Equation cannot be used. However, the pseudoinverse is always defined: it is based on the SVD decomposition of the matrix X, which finds the eigenvalues and eigenvectors of the matrix (as explained in chapter 8 on dimensionality reduction, when talking about PCA), so it can simply ignore the eigenvalues that are smaller than a tiny threshold. This means that the LinearRegression class will find the optimal solution even when some features are redundant or when m < n. This also explains two parameters that the LinearRegression predictor learns during training:

rank_ is the rank of the matrix X, i.e., the number of eigenvalues that are not tiny or zero. singular_ is the list of eigenvalues (just like PCA().fit(X).singular_values_). If a feature is a linear combination of other features, then one of these features will have a tiny or zero eigenvalue. Fortunately, this error does not change much of what I wrote in the book about the LinearRegression class: it is based on an analytical solution, it performs poorly when there is a large number of features, it is linear with regards to the number of instances, it does not support out-of-core (i.e., all data must fit in memory to use it), it does not require feature scaling, the order of the instances in the training set does not matter, and so on. You could say it's an implementation detail, but it's quite an important one, and I apologize for the error.

1 Setup

First, let's make sure this notebook works well in both python 2 and 3, import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures:

[ ]:
```
# Python 3.5 is required
import sys
assert sys.version_info >= (3, 5)
```

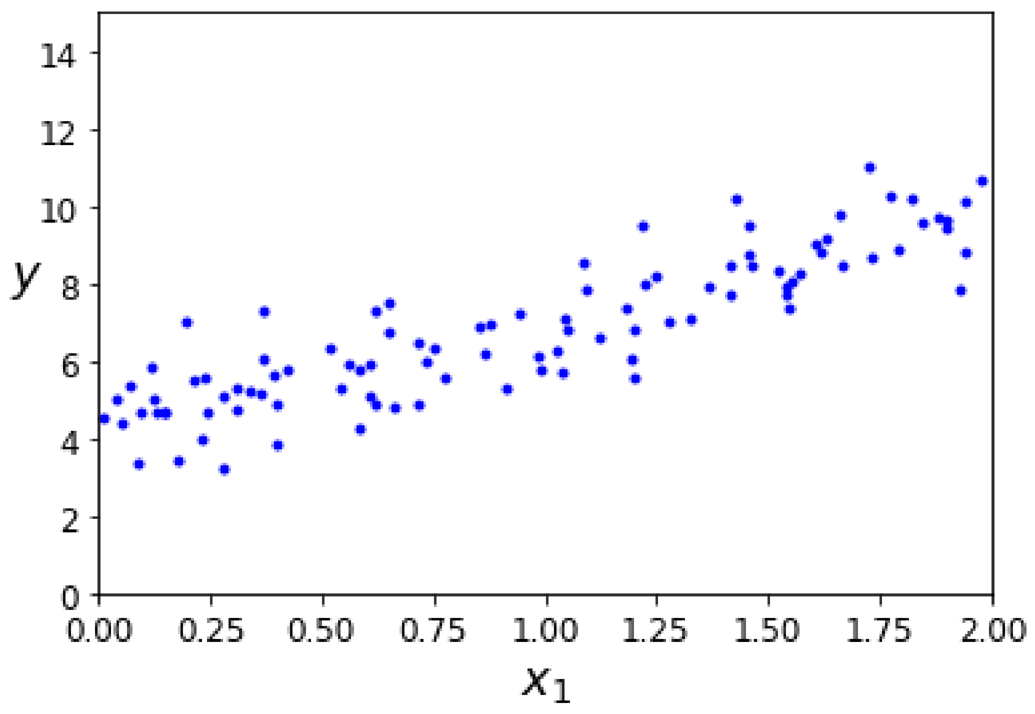
```
# Scikit-Learn 0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
#PROJECT_ROOT_DIR = "."
#CHAPTER_ID = "training_linear_models"
#IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
#os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

2 Linear regression using the Normal Equation

[ ]:
```
np.random.seed(42)
import numpy as np
#Generate random numbers between 0 and 1.
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

We generate data with a noisy linear trend, let's plot it.

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



Let's compute $\hat{\theta}$ using $(X'X)^{-1} \cdot (X'y)$. Command inv() inverts a matrix. First we add constant term to the X, which will have two features (X_1 and constant). $X = X[1, X_1]$. In Python matrix transposition is X_b.T $= X'$, matrix multuplication $X'X =$ X_b.T.dot.X_b

```
[ ]: 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]])
```

Actual function to generate noise was $y = 4 + 3x + u$, where u is the Gaussian noise. If we were to have a lot of data we would have estimated $\theta_0 = 4$ and $\theta_1 = 3$, instead we estimate $\theta_0 = 4.21$ and $\theta_1 = 2.77$. We can see how adding more observations moves us closer to the desired parameters. We are getting closer to $[4, 3]$ as $n \to \infty$

```
[ ]: np.random.seed(42)
     # Define function to estimate OLS as with different number of observations
```

```
def OLS(n):
    np.random.seed(42)
    X = 2 * np.random.rand(n, 1)
    y = 4 + 3 * X + np.random.randn(n, 1)
    X_b = np.c_[np.ones((n, 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)
    return theta_best
theta_best = OLS(100)
print("n= 100", "theta0 =" ,theta_best[0] , "theta1 =", theta_best[1] )
theta_best = OLS(1000)
print("n= 1000", "theta0 =" ,theta_best[0] , "theta1 =", theta_best[1] )
theta_best = OLS(10000)
print("n= 10000", "theta0 =" ,theta_best[0] , "theta1 =", theta_best[1] )
```

```
n= 100 theta0 = [4.21509616] theta1 = [2.77011339]
n= 1000 theta0 = [4.17478026] theta1 = [2.92260742]
n= 10000 theta0 = [4.03177675] theta1 = [2.98034911]
```

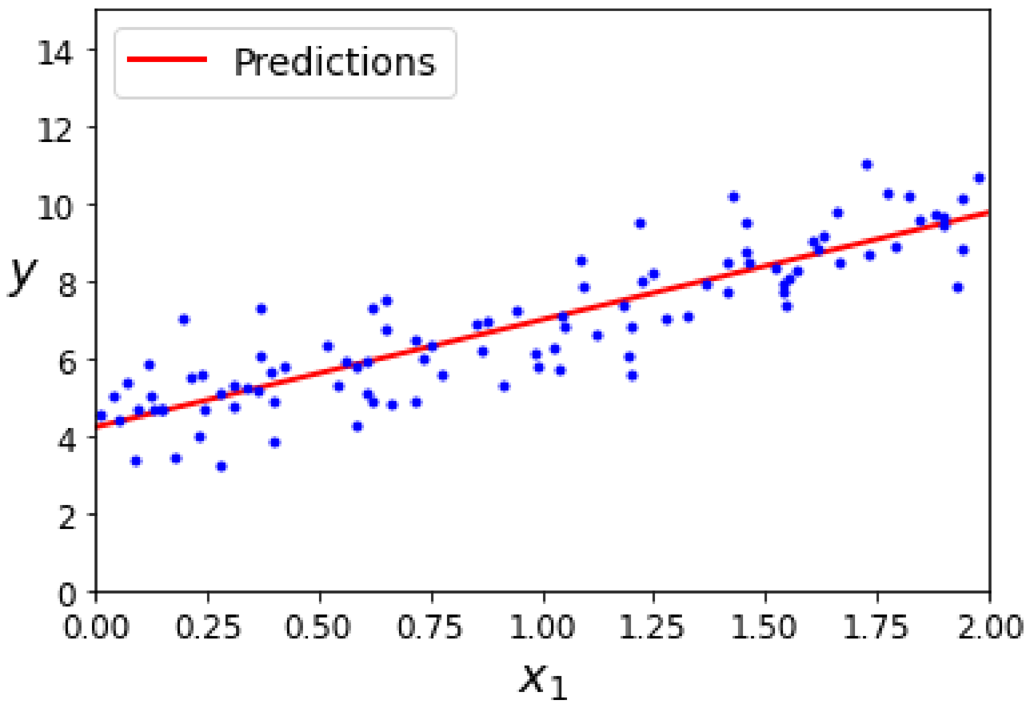
We can predict y for the values of x = (0,2), extreme value of x to plot the regression function.

[ ]:
```
np.random.seed(42)
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
# Extereme value of X
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
# Extreme value for y, knowing both bounds of X and Y will allow us to draw a␣
 ↪line between them.
y_predict = X_new_b.dot(theta_best)
y_predict
```

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

Plot the regression line using extreme values.

[ ]:
```
plt.plot(X_new, y_predict, "r-", linewidth=2, label="Predictions")
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([0, 2, 0, 15])
plt.show()
```



We can simply use prepackaged Linear Regression model from sklearn.

```
[ ]: 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]])
```

This function computes 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).dot(y)
```

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

If we have complete linear dependence, the classication regression will fail, but SVD will not.

```
[ ]: np.random.seed(42)
     Q = 2 * np.random.rand(100, 1)
     W = 4 + 3 * Q + np.random.randn(100, 1)
     Q_b = np.c_[np.ones((100, 1)), Q, Q]
     theta_best = np.linalg.inv(Q_b.T.dot(Q_b)).dot(Q_b.T).dot(W)
     print(theta_best)
```

```
          ␣
    ↪---------------------------------------------------------------------------

          LinAlgError                               Traceback (most recent call␣
    ↪last)

          <ipython-input-20-f7034e5526ca> in <module>()
            3 W = 4 + 3 * Q + np.random.randn(100, 1)
            4 Q_b = np.c_[np.ones((100, 1)), Q, Q]
      ----> 5 theta_best = np.linalg.inv(Q_b.T.dot(Q_b)).dot(Q_b.T).dot(W)
            6 print(theta_best)


          <__array_function__ internals> in inv(*args, **kwargs)


          /usr/local/lib/python3.6/dist-packages/numpy/linalg/linalg.py in inv(a)
          545     signature = 'D->D' if isComplexType(t) else 'd->d'
          546     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
      --> 547     ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
          548     return wrap(ainv.astype(result_t, copy=False))
          549


          /usr/local/lib/python3.6/dist-packages/numpy/linalg/linalg.py in␣
    ↪_raise_linalgerror_singular(err, flag)
           95
           96 def _raise_linalgerror_singular(err, flag):
      ---> 97     raise LinAlgError("Singular matrix")
           98
           99 def _raise_linalgerror_nonposdef(err, flag):
```

```
LinAlgError: Singular matrix
```

[ ]:
```
np.linalg.pinv(Q_b).dot(W)
```

[ ]:
```
array([[4.21509616],
       [1.38505669],
       [1.38505669]])
```

[ ]:
```
lin_reg.fit(Q_b, W)
lin_reg.intercept_, lin_reg.coef_
```

[ ]:
```
(array([4.21509616]), array([[0.        , 1.38505669, 1.38505669]]))
```

3 Linear regression using batch gradient descent

The Normal Equation computes the inverse of $X^T \cdot X$, which is an $n \times n$ matrix (where n is the number of features). The computational complexity of inverting such a matrix is typically about O($n^{2.4}$) to $O(n^3)$ (depending on the implementation). In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$. * However, the question is linear to the number of instances (observations) $O(m)$, so we can handle large datasets if they fit in the memory. * Once the coefficients are estimated, predictions are fast to make, the complexity is linear with respect to both features and instances, twice as many features will take twice the time.

Next we will look how to estimate linear with very large datasets

Gradient descent minimizes a cost function. Starting with several random numbers, you try steps in different directions testing if a step minimizes a cost function. We stop the algorithm converges – reduction in cost function stopped. (picture). * An important parameter in Gradient Descent is the size of the steps, determined by the learning rate hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time. * If the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. (picture) * Real-life cost function is complex and multi-dimensional. Starting point, direction of change, and the step-size can all lead to success or failure of optimization (picture).

MSE function of linear regression is convininent for optimization. It is: * Convex – if you pick any two points on the curve, the line segment joining them never crosses the curve). Hence, there are not local minima, only global minimum. * Lipschitz Continuous – the slope does not change very rapidly, i.e. the derivative is bounded a by a real number: $\mid f(x_1) - f(x_2) \mid \leq K \mid x_1 - x_2 \mid$, where $K \in \mathbf{R}$

These two facts have a great consequence: Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

Regression with two parameters looks like a bowl, where need to find a minimum. If the features are scaled the bowl is round, and we will reach the bottom relatively quickly. If the features are not scaled different scales confuse the algorithm into thinking that features are important than others.

Training a model means searching for a combination of model parameters that minimizes a cost function. It is a search in the model's parameter space: the more parameters a model has, the more dimensions this space has, and the harder the search is. Fortunately, since the cost function is convex in the case of Linear Regression, the needle is simply at the bottom of the bowl, so there is only one minimum point.

4 Batch Gradient Descent

To implement Gradient Descent, we compute the gradient of the cost function with regards to each model parameter $_j$. In other words, you need to calculate how much the cost function will change if you change j just a little bit.This is called a partial derivative. (Ex. mountain slopes facing different directions). Next we compute the partial derivative of the cost function with regards to parameter θ_j, noted $\frac{\partial MSE(\theta)}{\partial \theta_j}$.

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

We estimate a vector of gradients in one go. The gradient vector, noted $\nabla \theta MSE(\theta)$, contains all the partial derivatives of the cost function (one for each model parameter):

$$\nabla \theta MSE(\theta) \begin{cases} \frac{\partial MSE(\theta)}{\partial \theta_0} \\ \frac{\partial MSE(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial MSE(\theta)}{\partial \theta_n} \end{cases}$$

Notice that this formula involves calculations over the full training set X, at each Gradient Descent step! This is why the algorithm is called Batch Gradient Descent: it uses the whole batch of training data at every step. As a result it is terribly slow on very large training sets. However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation.

Once you have the gradient vector, which points uphill, just go in the opposite direction. This means subtracting $\nabla \theta MSE(\theta)$ 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:

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

[ ]:
```
# set step at 0.1
eta = 0.1
# number of steps
n_iterations = 1000
# number of observations
m = 100
# randomly set the starting point
theta = np.random.randn(2,1)
# Walk 1000 steps.
```

```
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]])
```

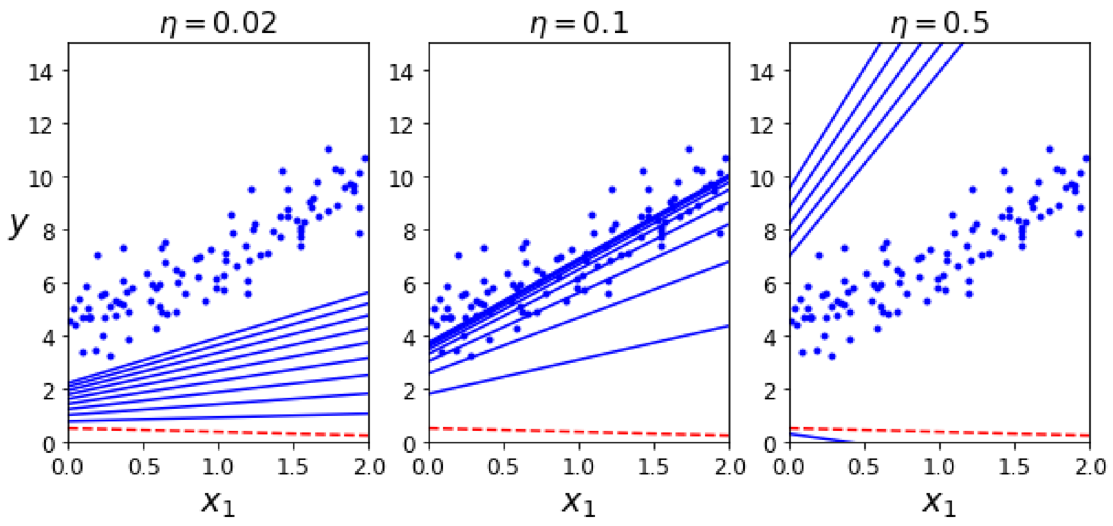
[ ]:
```
# This is what we got from Matrix OLS.
theta_best = OLS(100)
print("n= 100", "theta0 =" ,theta_best[0] , "theta1 =", theta_best[1] )
```

```
n= 100 theta0 = [4.21509616] theta1 = [2.77011339]
```

Let's look at the first 10 steps of gradient descent algorithms with different step sizes.

[ ]:
```
# set a path vector
theta_path_bgd = []
# program that plots gradient descent graph
def plot_gradient_descent(theta, eta, theta_path=None):
    # Number of observations m
    m = len(X_b)
    # Plot target and features in scatter plot
    plt.plot(X, y, "b.")
    # set number of iterations
    n_iterations = 1000
    # loop over the iterations
    for iteration in range(n_iterations):
        theta_path_bgd.append(theta)
    # plot first 10 iterations
        if iteration < 10:
            # preduct y using existing theta (starting values)
            y_predict = X_new_b.dot(theta)
            # Plot first iteration with red, others in blue color
            style = "b-" if iteration > 0 else "r--"
            # Plot regression line predicting y
            plt.plot(X_new, y_predict, style)
        # Calculate gradient using formula above.
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
        # Update theta using the gradient
        theta = theta - eta * gradients
        #Save estimate thetas in the vector theta_path
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, 0, 15])
    plt.title(r"$\eta = {}$".format(eta), fontsize=16)
```

```
[ ]: np.random.seed(42)
     theta = np.random.randn(2,1)  # random initialization

     plt.figure(figsize=(10,4))
     plt.subplot(131); plot_gradient_descent(theta, eta=0.02)
     plt.ylabel("$y$", rotation=0, fontsize=18)
     plt.subplot(132); plot_gradient_descent(theta, eta=0.1,␣
      ↪theta_path=theta_path_bgd)
     plt.subplot(133); plot_gradient_descent(theta, eta=0.5)

     plt.show()
```



- On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time.
- In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution
- On the right, the learning rate is too high: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step

If you have enough time it is always better to use small steps and a lot of iterations. If the data is too big for that, then you need to experiment using grid search.

How to set the number of iterations? If you set it too small, your may not find the minimum, if it is too large you may waste time without improving the fit. A good ways it set minimum gradient that would continue the iterations. If the rate of improvement is less than the minimum tolerance ϵ, it is a good idea to stop the iterations.

When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), the Batch Gradient Descent with a fixed learning rate has a convergence rate of $O = \frac{1}{Iterations}$. In other words, if you divide the tolerance ϵ by 10 (to have a more precise solution), then the algorithm will have to run about 10 times more iterations.

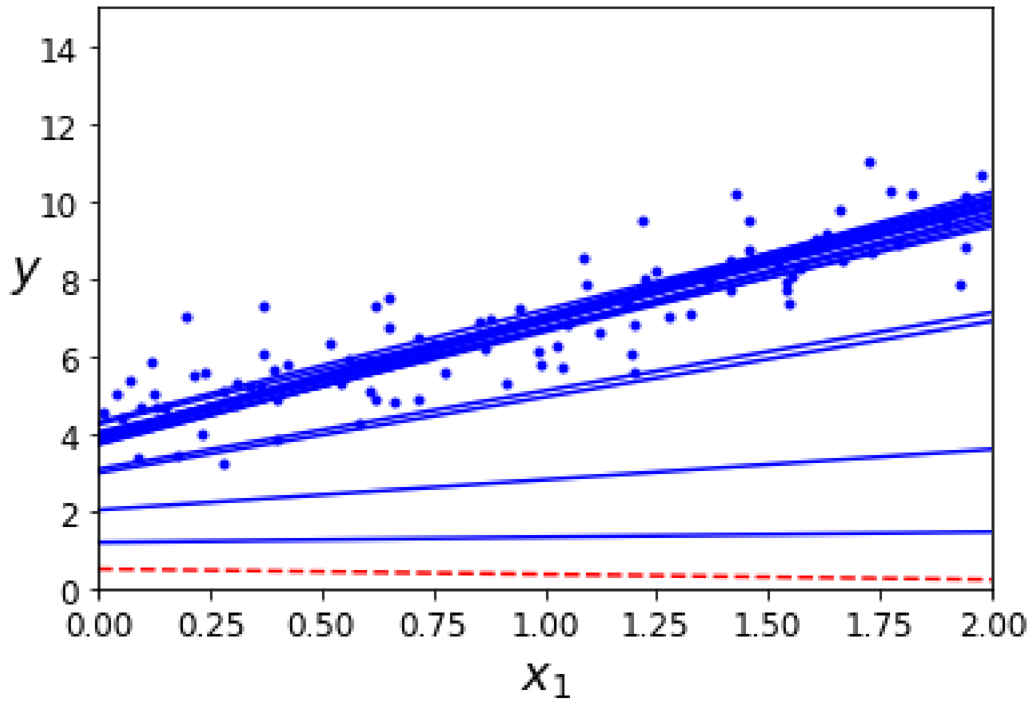
5 Stochastic Gradient Descent

- Batch Gradient Descent uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.
- Stochastic Gradient Descent (SGD) just picks a random instance in the training set at every step and computes the gradients based only on that single instance. This makes the algorithm much faster since it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration.
- SGD, due to its stochastic (i.e., random) nature, is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down . So once the algorithm stops, the final parameter values are good, but not optimal. (picture)

Note that since instances are picked randomly, some instances may be picked several times per epoch while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, another approach is to shuffle the training set, then go through it instance by instance, then shuffle it again, and so on. However, this generally converges more slowly.

[ ]:
```
# set path array
theta_path_sgd = []
m = len(X_b)
np.random.seed(42)
```

[ ]:
```
# number of iterations
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)
# start with randomly generated starting parameters theta
theta = np.random.randn(2,1)  # random initialization
# loop over iterations
for epoch in range(n_epochs):
    # add theta to the vector
    theta_path_sgd.append(theta)                 # not shown
    # loop over observations. Here we loop over all observations, we can just␣
↪had a few draw, like range(20)
    for i in range(m):
        # show regression lines for the first 20 iterations
        if epoch == 0 and i < 20:                    # not shown in the book
            # Predict y using existing theta
            y_predict = X_new_b.dot(theta)           # not shown
            # first line is red
            style = "b-" if i > 0 else "r--"         # not shown
            #other lines are blue. Plot predictions
            plt.plot(X_new, y_predict, style)        # not shown
        # Randomly select number between 1 and m.
        random_index = np.random.randint(m)
        # extract randomly selected observation for x
        xi = X_b[random_index:random_index+1]
        # extract randomly selected observation for y
        yi = y[random_index:random_index+1]
        # Calculate gradient for randomly selected observations
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        # Tolerance level, is 50 / (epoch * m + i + 5)
        eta = learning_schedule(epoch * m + i)
        # update theta
        theta = theta - eta * gradients


plt.plot(X, y, "b.")                                 # not shown
plt.xlabel("$x_1$", fontsize=18)                     # not shown
plt.ylabel("$y$", rotation=0, fontsize=18)           # not shown
plt.axis([0, 2, 0, 15])                              # not shown
plt.show()                                           # not shown
```



[ ]:

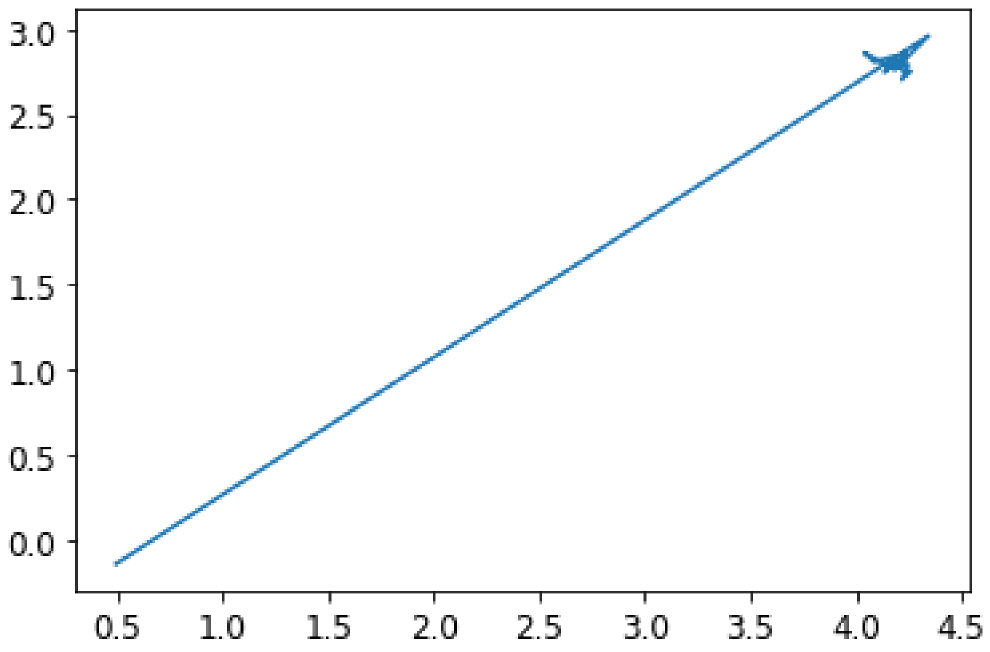
```
[ ]: thetapath = np.array(theta_path_sgd)
[ ]: thetapath[49,:]
```

```
[ ]: array([[4.21200431],
            [2.74968529]])
```

Remember that correct values are [4.21509616, 2.77011339], we are very close

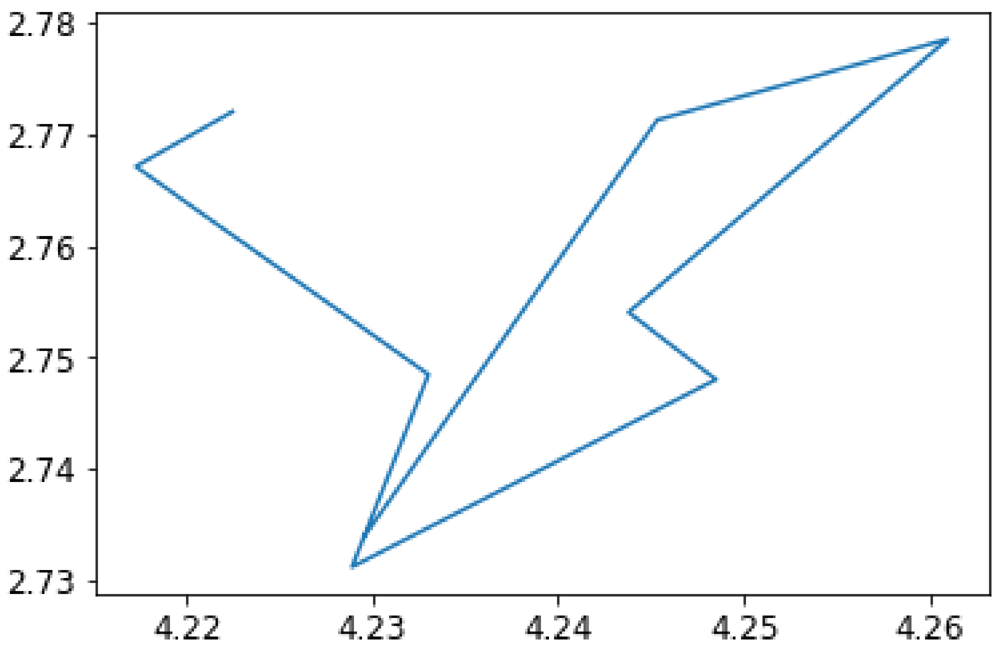
```
[ ]: # First 20 steps of iteration 1
     plt.plot(thetapath[0:20,0],thetapath[0:20,1])
```

```
[ ]: [<matplotlib.lines.Line2D at 0x7f4309b88780>]
```



```
[ ]: #40-50 steps
     plt.plot(thetapath[40:49,0],thetapath[40:49,1])
```

```
[ ]: [<matplotlib.lines.Line2D at 0x7f4309bdc0b8>]
```



We are close, but we are not really converging to the correct value. Next we estimate a SGD regression included in sklearn. Number of iterations is 50. The defaults to optimizing the squared error cost function. We run 50 epochs, starting with a learning rate of 0.1 (eta0=0.1), using the default learning schedule (different from the preceding one), and it does not use any regularization (penalty=None; more details on this shortly):

```
[ ]: from sklearn.linear_model import SGDRegressor
     sgd_reg = SGDRegressor(max_iter=50, penalty=None,  eta0 = 0.1,  random_state=42)
     sgd_reg.fit(X, y.ravel())
     sgd_reg.intercept_, sgd_reg.coef_
```

```
[ ]: (array([4.24365286]), array([2.8250878]))
```

Not great, let's increase the number of iterations

```
[ ]: from sklearn.linear_model import SGDRegressor
     sgd_reg = SGDRegressor(max_iter=500, penalty=None, eta0=0.1,random_state=42)
     sgd_reg.fit(X, y.ravel())
     sgd_reg.intercept_, sgd_reg.coef_
```

```
[ ]: (array([4.24365286]), array([2.8250878]))
```

Much better. Alternatively we can reduce tolerance 10 times.

```
[ ]: from sklearn.linear_model import SGDRegressor
     sgd_reg = SGDRegressor(max_iter=5000, penalty=None, eta0=0.1,random_state=42)
```

```
sgd_reg.fit(X, y.ravel())
sgd_reg.intercept_, sgd_reg.coef_
```

```
[ ]: (array([4.24365286]), array([2.8250878]))
```

The results got worse we are dancing aroung correct value.

6 Mini-batch gradient descent

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. The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

The algorithm's progress in parameter space is less erratic than with SGD, especially with fairly large mini-batches. As a result, Mini-batch GD will end up walking around a bit closer to the minimum than SGD. But, on the other hand, it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike Linear Regression as we saw earlier). Next we will show the paths taken by the three Gradient Descent algorithms in parameter space during training. They all end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around. However, the Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.

```
[ ]: theta_path_mgd = []

n_iterations = 50
# We will just use 20 observations out of 100
minibatch_size = 20

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

t0, t1 = 200, 1000
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    # Shuffle the data x and y. Because we will draw a series of 20 observation␣
 ↪order now can matter a great deal. Before we
    # were just
    shuffled_indices = np.random.permutation(m)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    # Loop each batch of observations from 0 to 20
    for i in range(0, m, minibatch_size):
        # set counter t
        t += 1
        # Get first of observation of the batch
        xi = X_b_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]
        # calculate the gradient
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        # calculate new step
        eta = learning_schedule(t)
        # update theta
        theta = theta - eta * gradients
        # record the path
        theta_path_mgd.append(theta)
```

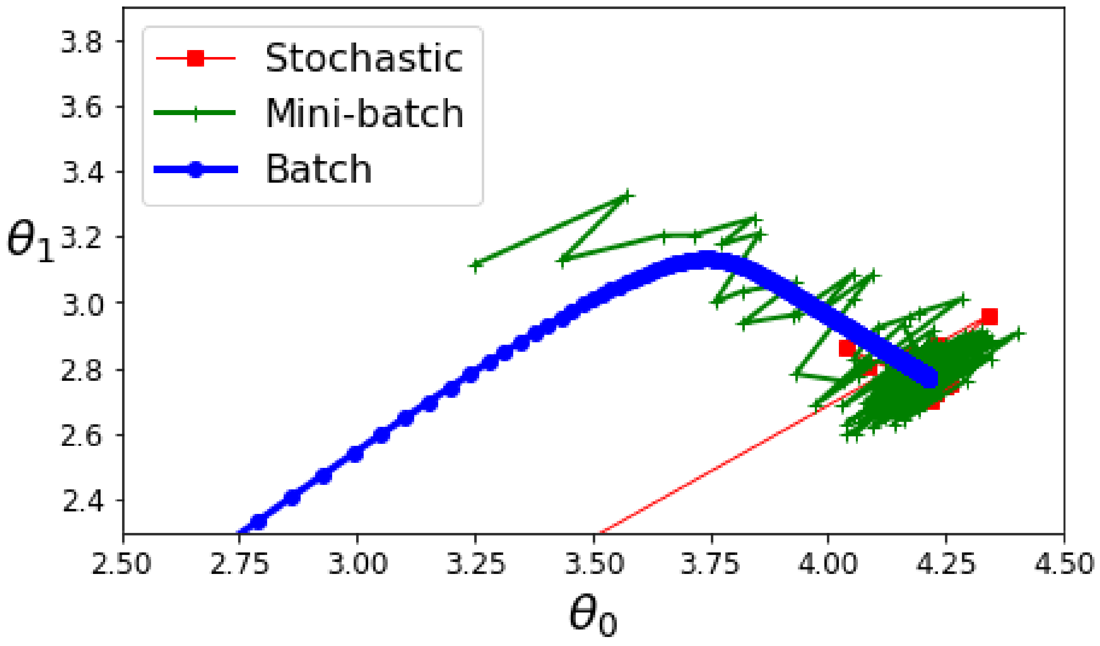
[ ]:
```
print("final theta =",theta)
```

```
final theta = [[4.25214635]
 [2.7896408 ]]
```

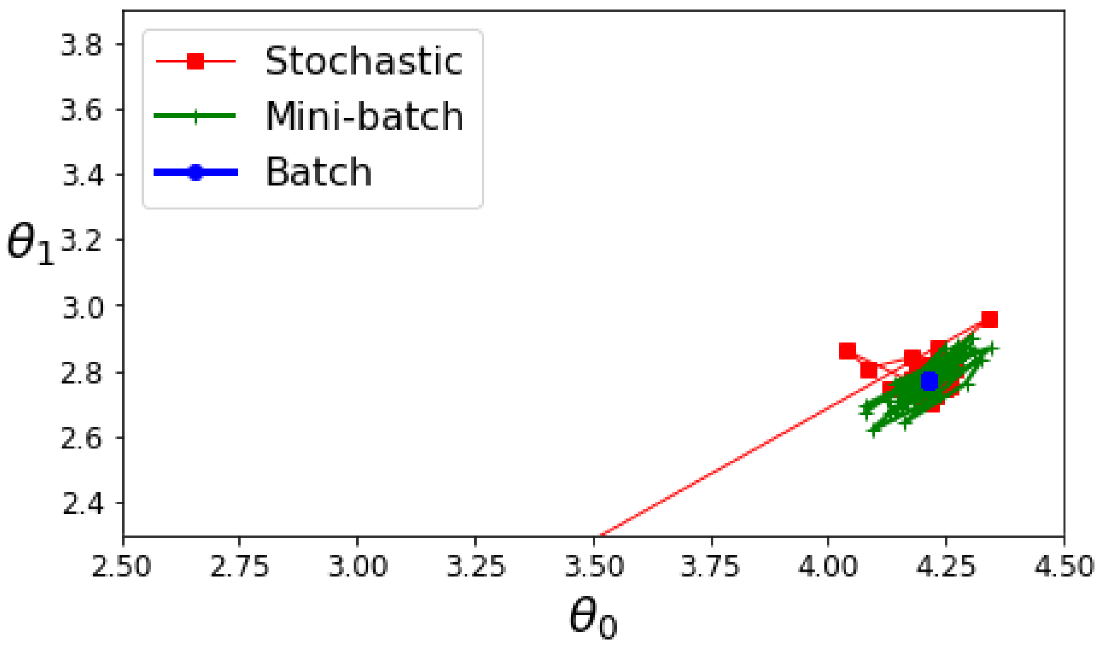
[ ]:
```
# Lets create vetors for our paths
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
```

[ ]:
```

```

[ ]:
```
plt.figure(figsize=(7,4))
plt.plot(theta_path_sgd[:, 0], theta_path_sgd[:, 1], "r-s", linewidth=1,␣
 ↪label="Stochastic")
plt.plot(theta_path_mgd[:, 0], theta_path_mgd[:, 1], "g-+", linewidth=2,␣
 ↪label="Mini-batch")
# We called this function 3 times, so the vector has 3000 obs instead of 1000.␣
 ↪We just use the first 1000
plt.plot(theta_path_bgd[:1000, 0], theta_path_bgd[:1000, 1], "b-o",␣
 ↪linewidth=3, label="Batch")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$\theta_0$", fontsize=20)
plt.ylabel(r"$\theta_1$   ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])
plt.show()
```



[ ]:
```
# Plot last 50 steps
plt.figure(figsize=(7,4))
plt.plot(theta_path_sgd[-50:, 0], theta_path_sgd[-50:, 1], "r-s", linewidth=1,␣
 ↪label="Stochastic")
plt.plot(theta_path_mgd[-50:, 0], theta_path_mgd[-50:, 1], "g-+", linewidth=2,␣
 ↪label="Mini-batch")
# We called this function 3 times, so the vector has 3000 obs instead of 1000.␣
 ↪We just use the first 1000
plt.plot(theta_path_bgd[950:1000, 0], theta_path_bgd[950:1000, 1], "b-o",␣
 ↪linewidth=3, label="Batch")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$\theta_0$", fontsize=20)
plt.ylabel(r"$\theta_1$   ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])
plt.show()
```



7 Polynomial regression

What if your data is actually more complex than a simple straight line? A simple way to use Linear model for non-linear data is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called Polynomial Regression.

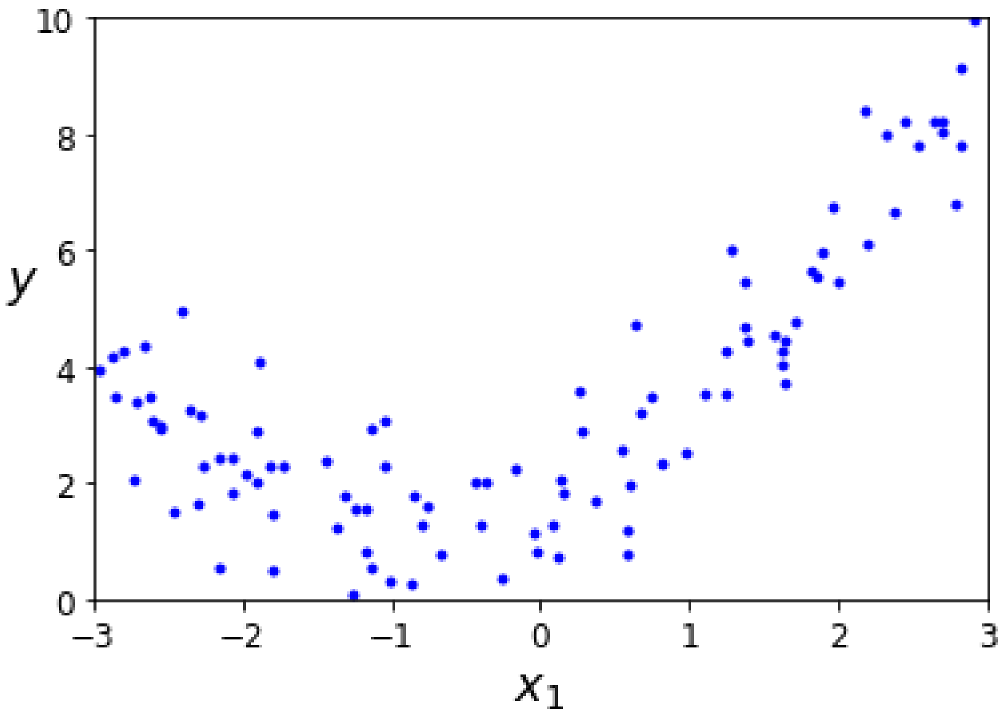
```
[ ]: import numpy as np
     import numpy.random as rnd
     np.random.seed(42)
```

X is 100 observations draws from $X = 6 * R - 3$, where $R \in (0, 1)$. Then we estimate $Y = 0.5X^2 + X + 2 + R$.

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

Plot X and Y

```
[ ]: plt.plot(X, y, "b.")
     plt.xlabel("$x_1$", fontsize=18)
     plt.ylabel("$y$", rotation=0, fontsize=18)
     plt.axis([-3, 3, 0, 10])
     plt.show()
```



```
[ ]: # Import package to craeate polinomial features
     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])
```

```
[ ]: print(X_poly[0])
     print((-0.75275929)**2)
```

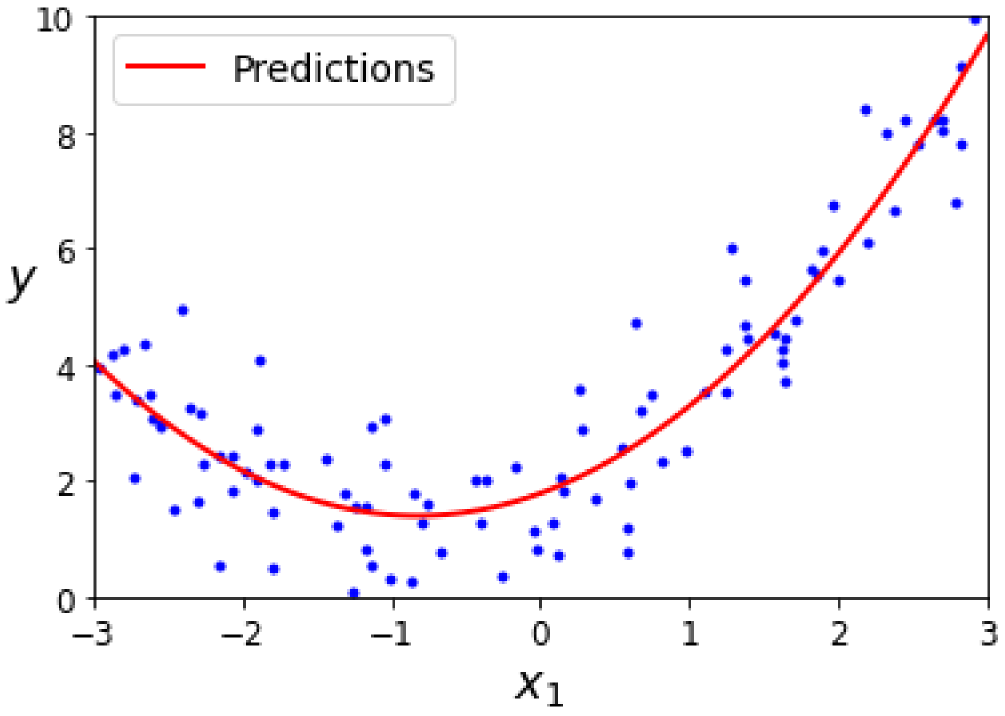
```
[-0.75275929  0.56664654]
0.566646548681304
```

```
[ ]: # Fit polinomial regression
     lin_reg = LinearRegression()
     lin_reg.fit(X_poly, y)
     lin_reg.intercept_, lin_reg.coef_
```

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

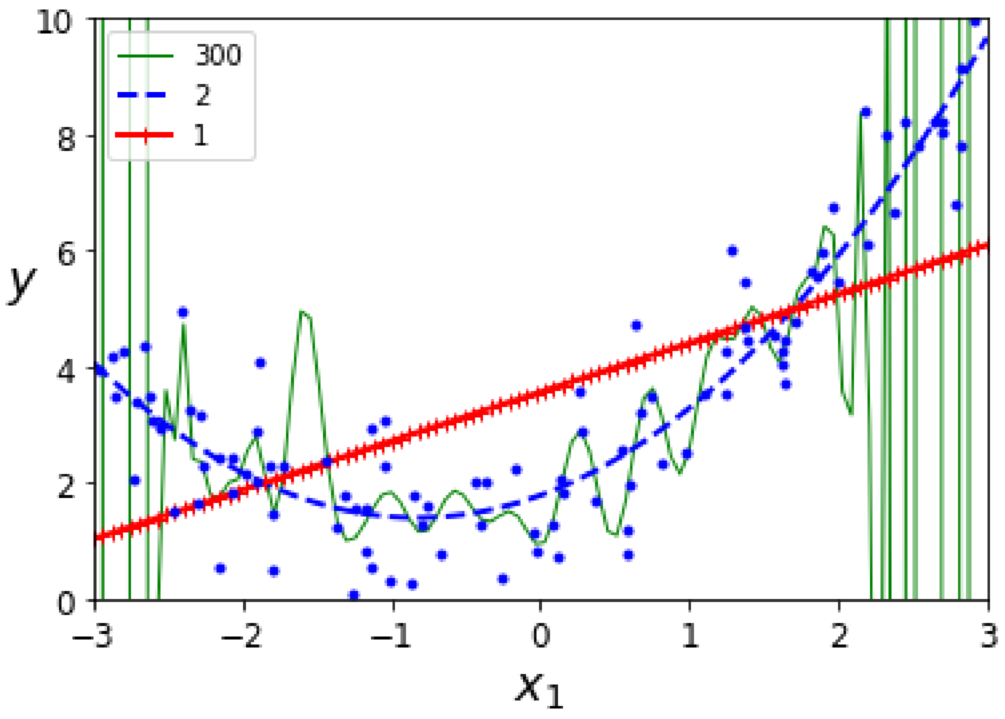
The model estimates $\hat{Y} = 0.56X^2 + 0.93X + 1.78 + R$, whereas a real model is $Y = 0.5X^2 + X + 2 + R$. Polinomial regresion is capable of finding relationships between features by addint all combinations of features up to the given degree. For example, if there were two features a and b, PolynomialFeatures with degree=3 would not only add the features a^2, a^3, b^2, and b^3, but also the combinations ab, a^2b, and ab^2. Polynomial features models with a degree d and n features, would produce $\frac{(n+d)!}{d!n!}$ combinations of features.

```
[ ]: # Generate space of drawing a solid prediction line.
     X_new=np.linspace(-3, 3, 100).reshape(100, 1)
     # square the X
     X_new_poly = poly_features.transform(X_new)
     y_new = lin_reg.predict(X_new_poly)
     # Draw scattered data
     plt.plot(X, y, "b.")
     # Draw prediction line
     plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
     plt.xlabel("$x_1$", fontsize=18)
     plt.ylabel("$y$", rotation=0, fontsize=18)
     plt.legend(loc="upper left", fontsize=14)
     plt.axis([-3, 3, 0, 10])
     plt.show()
```



High-degree Polynomial Regression fits the training data much better than the plain Linear Regression. Next we estimate a 300-degree polynomial model to the preceding training data, and compare the result with a pure linear model and a quadratic model (2nd degree polynomial).

[ ]:
```python
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ])
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)

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



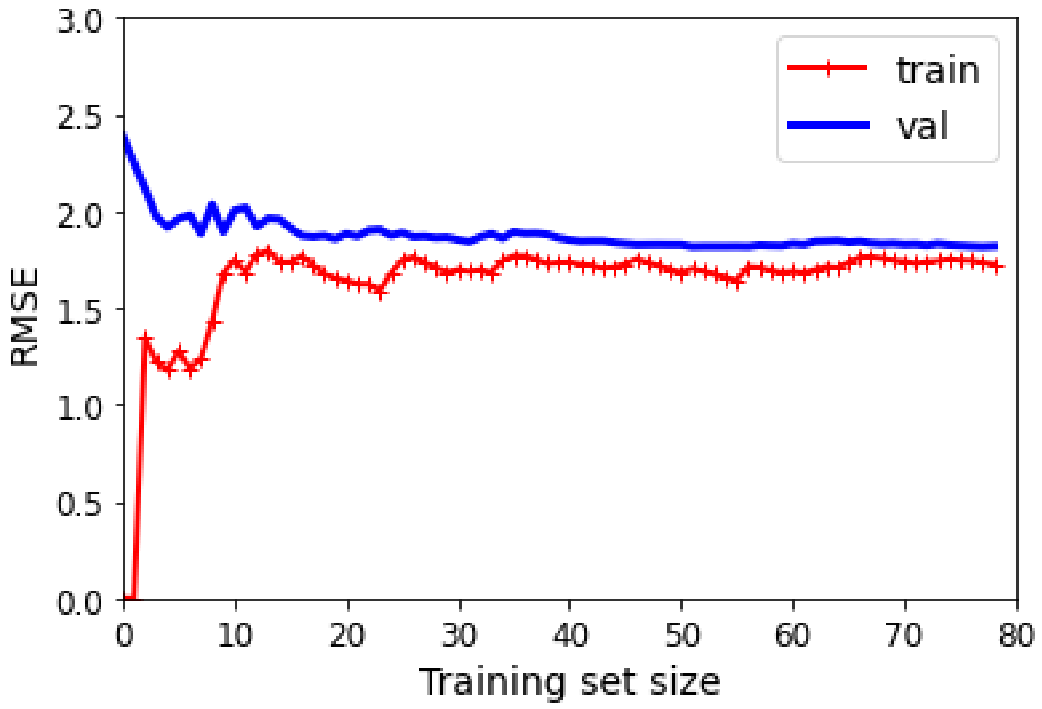
- This high-degree Polynomial Regression model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model.

- The data was generated using a quadratic model, but in general you won't know what function generated the data. In Chapter 2 you used cross-validation to get an estimate of a model's generalization performance.

Another way is to look at the learning curves: these are plots of the model's performance on the training set and the validation set as a function of the training set size. To generate the plots, simply train the model several times on different sized subsets of the training set. The following code defines a function that plots the learning curves of a model given some training data:

[ ]:
```
# Import MS and train-test split
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    # Split data into test and train
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=10)
    # create vectors to save errors
    train_errors, val_errors = [], []
    # loop through all observations adding one at a time
    for m in range(1, len(X_train)):
        # fit the model
        model.fit(X_train[:m], y_train[:m])
        # predict y on training
        y_train_predict = model.predict(X_train[:m])
        # predict on validation
        y_val_predict = model.predict(X_val)
        # calculate training and validation errors
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
# plot errors and sample size
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
    plt.legend(loc="upper right", fontsize=14)   # not shown in the book
    plt.xlabel("Training set size", fontsize=14) # not shown
    plt.ylabel("RMSE", fontsize=14)              # not shown
```

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

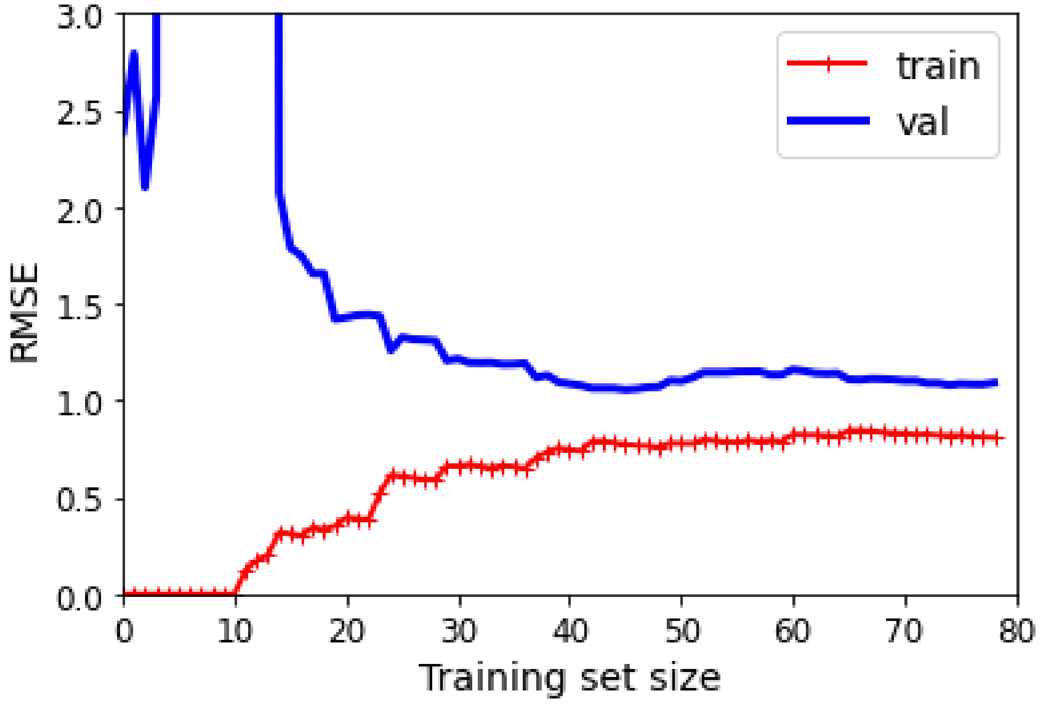


When there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero. But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau, at which point adding new instances to the training set doesn't make the average error much better or worse.

When the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite big. Then as the model is shown more training examples, it learns and thus the validation error slowly goes down. However, once again a straight line cannot do a good job modeling the data, so the error ends up at a plateau, very close to the other curve. These learning curves are typical of an underfitting model. Both curves have reached a plateau; they are close and fairly high.

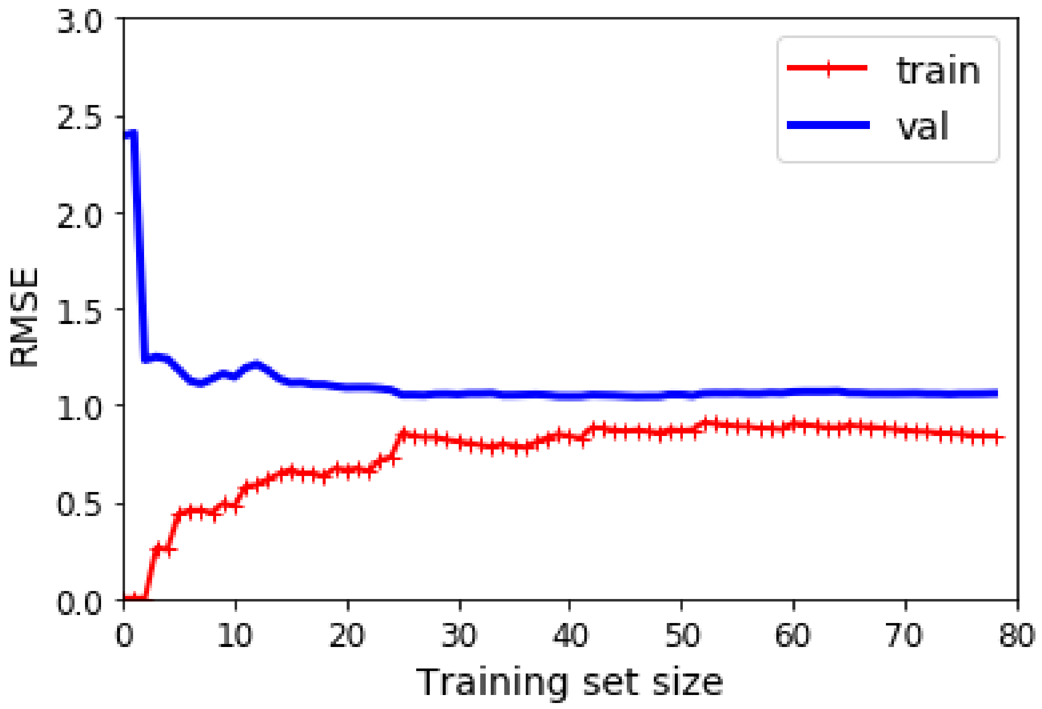
Now let's look at the learning curves of a 10th-degree polynomial model on the same data.

[ ]:
```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])           # not shown
plt.show()                                  # not shown
```



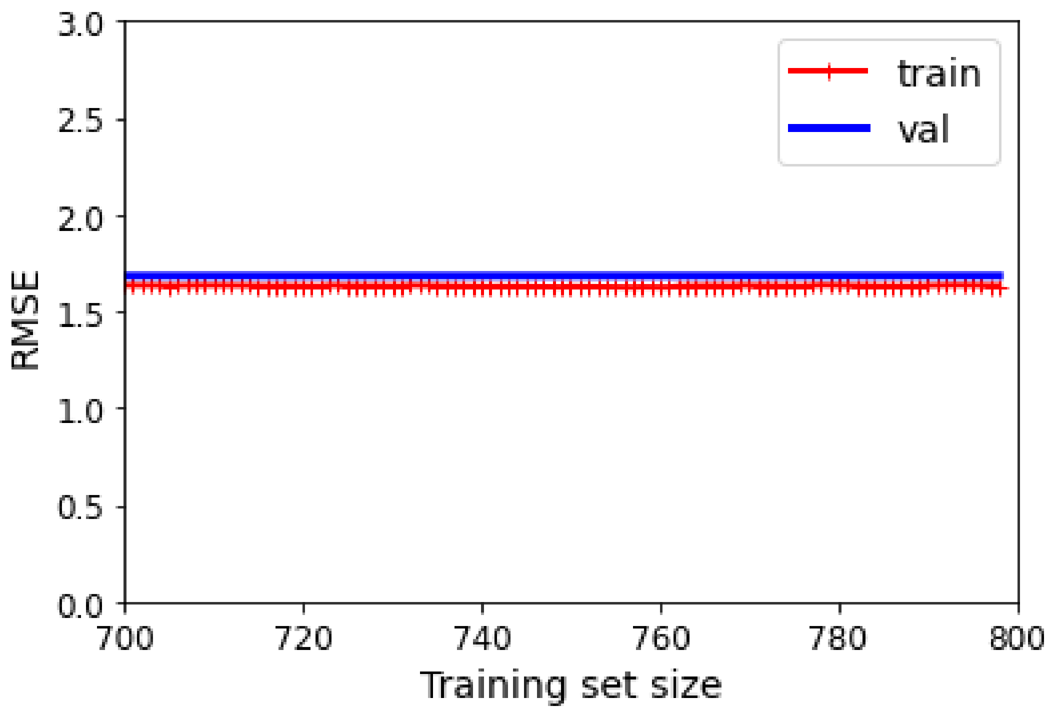
These learning curves look a bit like the previous ones, but there are two very important differences: 1. The error on the training data is much lower than with the Linear Regression model. 2. There is a persistent gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an overfitting model. However, if you used a much larger training set, the two curves would continue to get closer.

[ ]:
```
polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])           # not shown
plt.show()
```

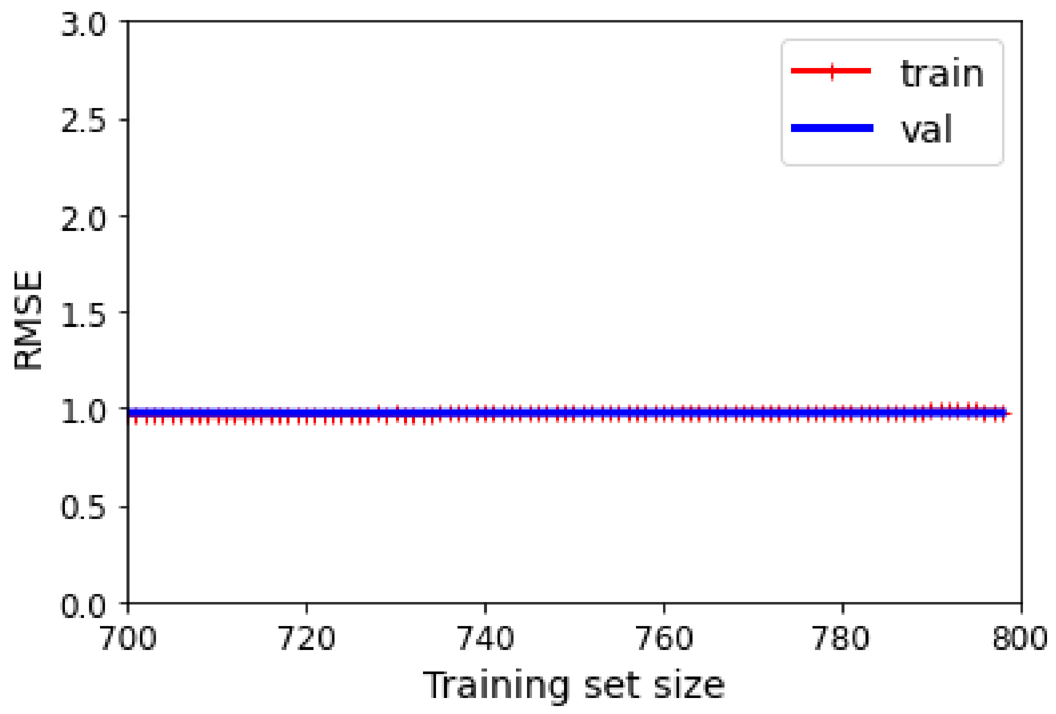


Maybe we have too few obervations?

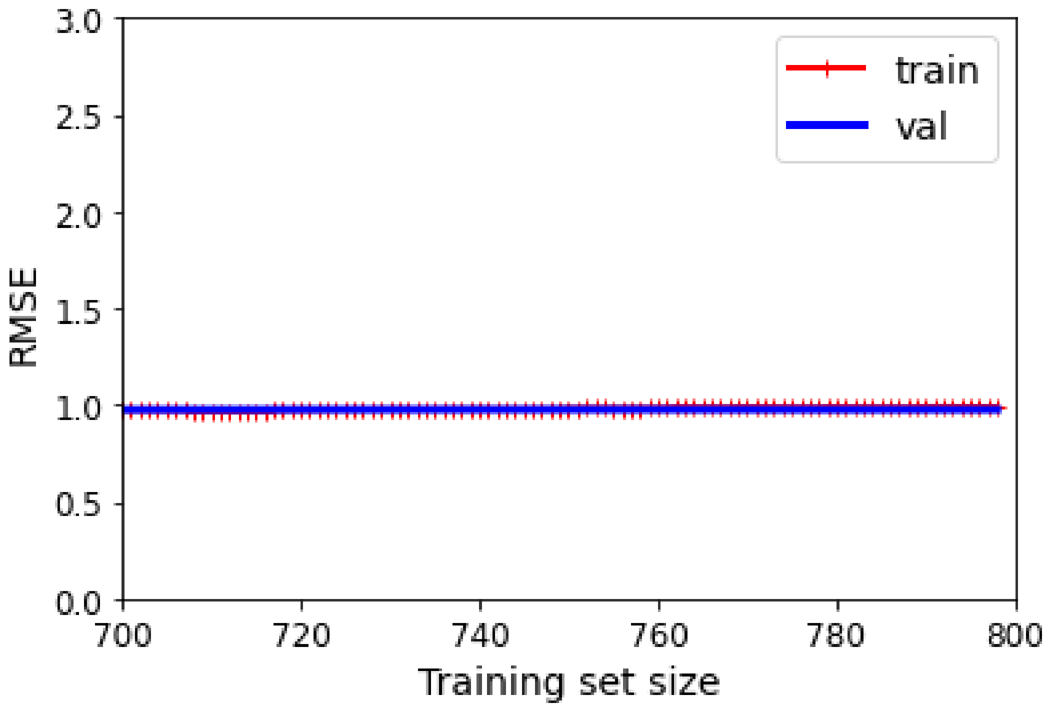
```
[ ]: # Set large data
np.random.seed(42)
m = 1000
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)

lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([700, 800, 0, 3])                         # not shown in the book
plt.show()                                       # not shown
```



```
[ ]: polynomial_regression = Pipeline([
             ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
             ("lin_reg", LinearRegression()),
         ])

     plot_learning_curves(polynomial_regression, X, y)
     plt.axis([700, 800, 0, 3])           # not shown
     plt.show()                           # not shown
```



```
[ ]: polynomial_regression = Pipeline([
             ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
             ("lin_reg", LinearRegression()),
         ])

     plot_learning_curves(polynomial_regression, X, y)
     plt.axis([700, 800, 0, 3])           # not shown
     plt.show()                           # not shown
```



We fixed overfitting issues by using more data.

8 Regularized models

A good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. For example, a simple way to regularize a polynomial model is to reduce the number of polynomial degrees. 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.

9 Ridge Regression

Ridge Regression (also called Tikhonov regularization) is a regularized version of Linear Regression: a regularization term equal to 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.

The hyperparameter α controls how much you want to regularize the model. If $\alpha = 0$ then Ridge Regression is just Linear Regression. If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean. Ridge regression cost function, where we add a sum of squared parameters.

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

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define w as the vector of feature weights (θ_1 to θ_n), then the regularization term is simply equal to $\frac{1}{2}(\|w\|_2)^2$, where $\|.\|_2$ is teh l_2 norm of the weight vector. It is important to scale the data (e.g., using a StandardScaler) before performing Ridge Regression, as it is sensitive to the scale of the input features. This is true of most regularized models. For Gradient Descent, just add αw to the MSE gradient vector.

Next we will show several Ridge models trained on some linear data using different α value. On the left, plain Ridge models are used, leading to linear predictions. On the right, the data is first expanded using PolynomialFeatures(degree=10), then it is scaled using a StandardScaler, and finally the Ridge models are applied to the resulting features: this is Polynomial Regression with Ridge regularization.

Increasing α leads to flatter (i.e., less extreme, more reasonable) predictions; this reduces the model's variance but increases its bias.

As with Linear Regression, we can perform Ridge Regression either by computing a closed-form equation or by performing Gradient Descent. The closed-form solution (where A is the $n \times n$ identity matrix except with a 0 in the top-left cell, corresponding to the bias term).

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

Remember the regression without ridge it:

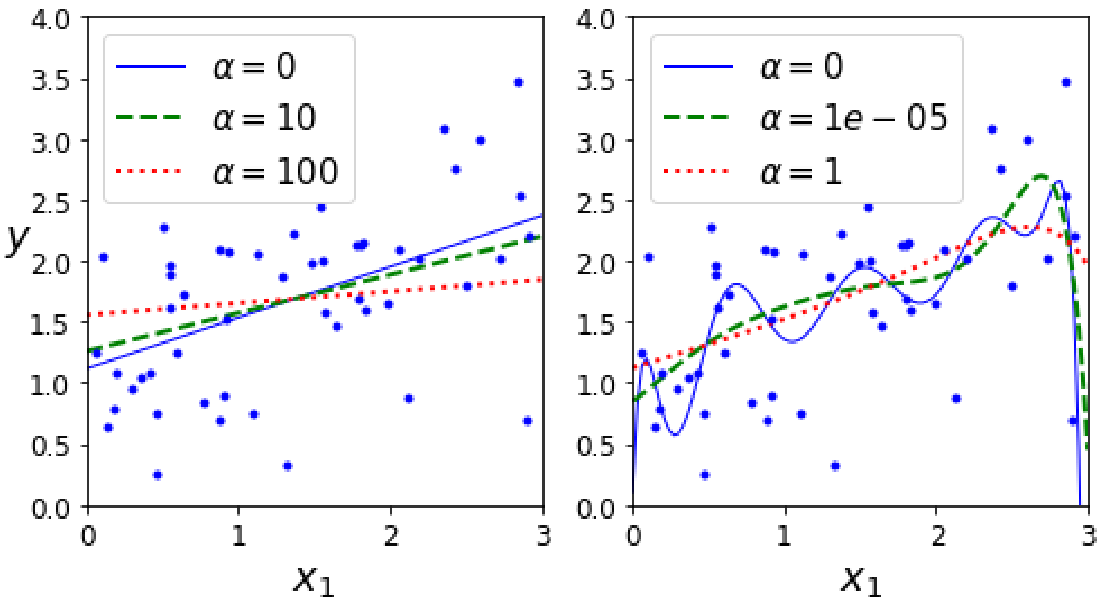
$$\hat{\theta} = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

```
[ ]: from sklearn.linear_model import Ridge
     train_errors = []
     val_errors = []
     np.random.seed(42)
     # Take 30 obs
     m = 50
     X = 3 * np.random.rand(m, 1)
     y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5

     X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
      ↪random_state=10)


     X_new = np.linspace(0, 3, 100).reshape(100, 1)

     def plot_model(model_class, polynomial, alphas, **model_kargs):
         for alpha, style in zip(alphas, ("b-", "g--", "r:")):
```

```
        model = model_class(alpha, **model_kargs) if alpha > 0 else␣
↪LinearRegression()
        if polynomial:
            model = Pipeline([
                    ("poly_features", PolynomialFeatures(degree=10,␣
↪include_bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ])
        model.fit(X_train, y_train)
        print("Coefficients from",model_class,"with alpha =",alpha )
        if polynomial:
            classifier = model.named_steps['regul_reg']
            print(classifier.coef_)
        else:
            print(model.coef_)
        y_new_regul = model.predict(X_new)
        # look at cross validation error
        y_train_predict = model.predict(X_train[:m])
        # predict on validation
        y_val_predict = model.predict(X_val)
        # calculate training and validation errors
        train_errors.append(mean_squared_error(y_train, y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
        lw = 2 if alpha > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\alpha =␣
↪{}$".format(alpha))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 4])

plt.figure(figsize=(8,4))
plt.subplot(121)
# Plot ridge regression
plot_model(Ridge, polynomial=False, alphas=(0, 10, 100), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Ridge, polynomial=True, alphas=(0, 10**-5, 1), random_state=42)
plt.show()
print("Training Errors Linear regression with Ridge",train_errors[0:3])
print("Validation Errors Linear regression with Ridge",val_errors[0:3])
print("Training Errors Polynomial regression with Ridge",train_errors[3:])
print("Validation Errors Polinomial regression with Ridge",val_errors[3:])
```

```
Coefficients from <class 'sklearn.linear_model._ridge.Ridge'> with alpha = 0
[[0.4176916]]
```

```
Coefficients from <class 'sklearn.linear_model._ridge.Ridge'> with alpha = 10
[[0.31349263]]
Coefficients from <class 'sklearn.linear_model._ridge.Ridge'> with alpha = 100
[[0.09660269]]
Coefficients from <class 'sklearn.linear_model._ridge.Ridge'> with alpha = 0
[[ 2.79676283e+01 -8.70301876e+02  1.02393993e+04 -6.14861326e+04
   2.14563424e+05 -4.60902057e+05  6.18449228e+05 -5.05246948e+05
   2.29908470e+05 -4.46826954e+04]]
Coefficients from <class 'sklearn.linear_model._ridge.Ridge'> with alpha = 1e-05
[[  0.78077474   2.17580376 -15.03623374  26.89195817  -5.61192609
  -23.29096036  -4.96673856  21.51691619  22.82230017 -24.96631119]]
Coefficients from <class 'sklearn.linear_model._ridge.Ridge'> with alpha = 1
[[ 0.32396213  0.04946447  0.02169417  0.04508521  0.05805455  0.04884026
   0.01942467 -0.02537665 -0.08092134 -0.14346268]]
```



```
Training Errors Linear regression with Ridge [0.3850221633585592,
0.3931885655081954, 0.462567291650715]
Validation Errors Linear regression with Ridge [0.2931679786457559,
0.3258256737772605, 0.44095925341635367]
Training Errors Polynomial regression with Ridge [0.32527325068563034,
0.3597310299494832, 0.3788227343500468]
Validation Errors Polinomial regression with Ridge [0.2874513215172753,
0.29210668090163644, 0.2793681709990138]
```

Ridge regression improves fit of complex polynomial model and worsens the fit of simple linear model.

10 Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression (simply called 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 l_1 norm of the weight vector instead of half the square of the l_2 norm.

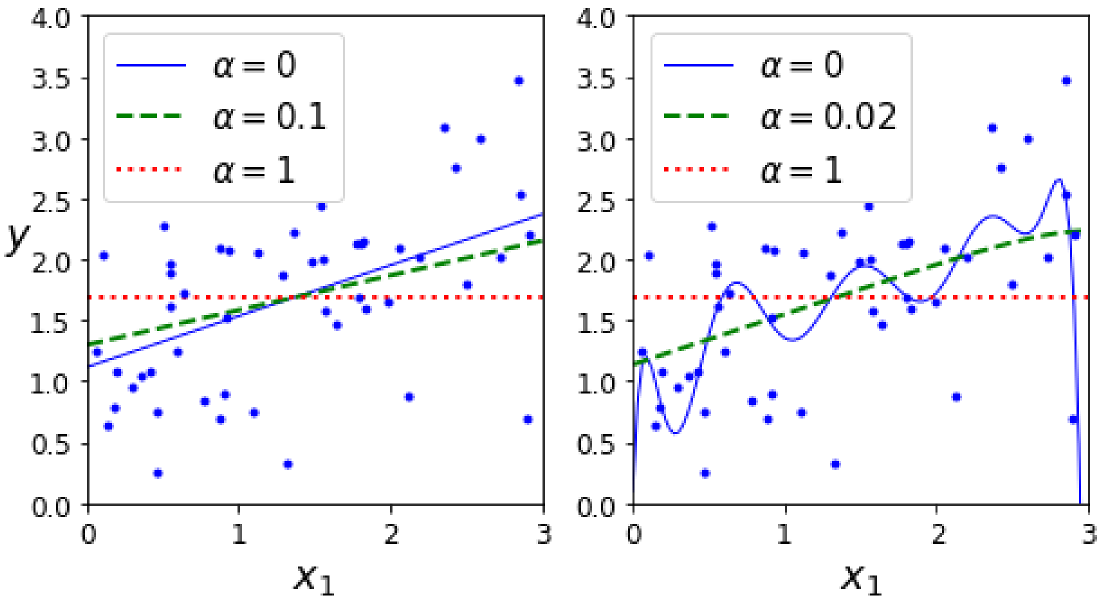
$$J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{m} |\theta_i|$$

An important characteristic of Lasso Regression is that it tends to completely eliminate the weights of the least important features (i.e., set them to zero). For example, the dashed line in the right plot on a figure below (with $\alpha = 10^{-7}$) looks quadratic, almost linear: all the weights for the high-degree polynomial features are equal to zero. In other words, Lasso Regression automatically performs feature selection and outputs a sparse model (i.e., with few nonzero feature weights).

```
[ ]: from sklearn.linear_model import Lasso
     train_errors = []
     val_errors = []
     plt.figure(figsize=(8,4))
     plt.subplot(121)
     plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1), random_state=42)
     plt.ylabel("$y$", rotation=0, fontsize=18)
     plt.subplot(122)
     plot_model(Lasso, polynomial=True, alphas=(0, 0.02, 1), tol=1, random_state=42)
     plt.show()
     print("Training Errors Linear regression with Lasso",train_errors[0:3])
     print("Validation Errors Linear regression with Lasso",val_errors[0:3])
     print("Training Errors Polynomial regression with Lasso",train_errors[3:])
     print("Validation Errors Polinomial regression with Lasso",val_errors[3:])
```

```
Coefficients from <class 'sklearn.linear_model._coordinate_descent.Lasso'> with
alpha = 0
[[0.4176916]]
Coefficients from <class 'sklearn.linear_model._coordinate_descent.Lasso'> with
alpha = 0.1
[0.28473922]
Coefficients from <class 'sklearn.linear_model._coordinate_descent.Lasso'> with
alpha = 1
[0.]
Coefficients from <class 'sklearn.linear_model._coordinate_descent.Lasso'> with
alpha = 0
[[ 2.79676283e+01 -8.70301876e+02  1.02393993e+04 -6.14861326e+04
   2.14563424e+05 -4.60902057e+05  6.18449228e+05 -5.05246948e+05
   2.29908470e+05 -4.46826954e+04]]
Coefficients from <class 'sklearn.linear_model._coordinate_descent.Lasso'> with
alpha = 0.02
[ 0.3569844   0.          -0.          -0.          -0.          -0.
```

```
 -0.00764842 -0.00799212 -0.00746378 -0.00698341]
Coefficients from <class 'sklearn.linear_model._coordinate_descent.Lasso'> with
alpha = 1
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```



```
Training Errors Linear regression with Lasso [0.3850221633585592,
0.3983174004851076, 0.5162468019225317]
Validation Errors Linear regression with Lasso [0.2931679786457559,
0.3374260613813013, 0.5127463561383779]
Training Errors Polynomial regression with Lasso [0.32527325068563034,
0.383267741215846, 0.5162468019225317]
Validation Errors Polinomial regression with Lasso [0.2874513215172753,
0.30262353461319974, 0.5127463561383779]
```

In our example Lasso regression worsens the fit.

For example, the dashed line in the right plot on (with $\alpha = 0.02$) looks almost linear: it is not completely zero, there are positive weights on X^8, X^9, and X^10. In other words, Lasso Regression automatically performs feature selection and outputs asparse model (i.e., with few nonzero feature weights).

11 Elastic net

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.

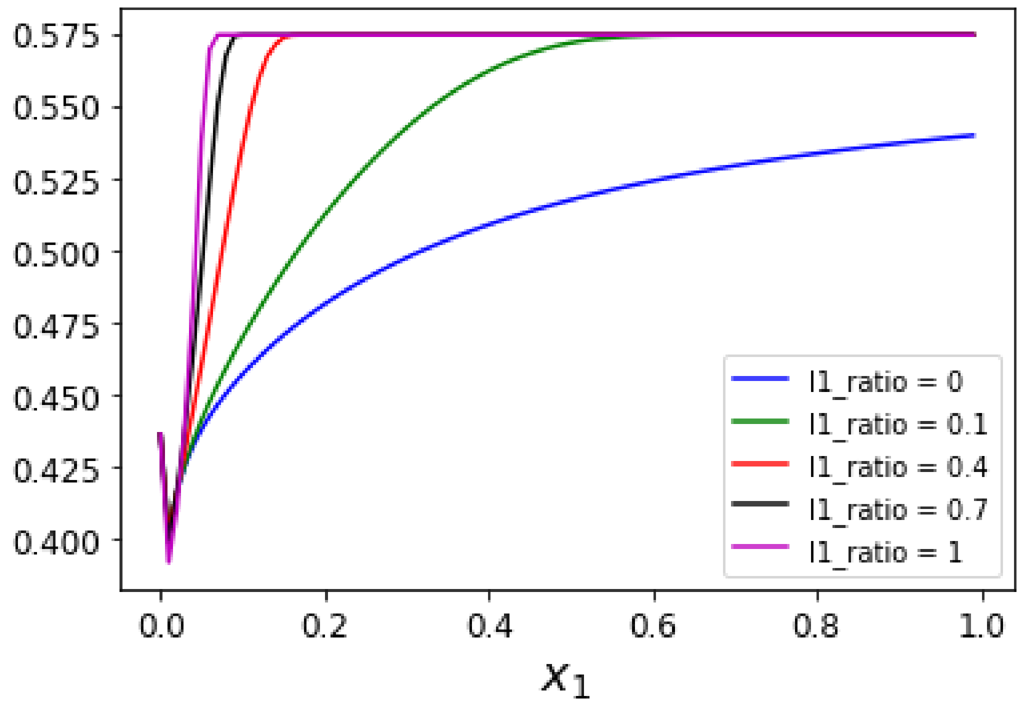
$$J(\theta) = MSE(\theta) + r\alpha\frac{1}{2}\sum_{i=1}^{m}|\theta_i| + \frac{1-r}{2}\alpha\sum_{i=1}^{m}\theta_i^2$$

So when should you use Linear Regression, Ridge, Lasso, or Elastic Net? It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain Linear Regression. * Ridge is a good default, * if you suspect that only a few features are actually useful, you should prefer Lasso or Elastic Net since they tend to reduce the useless features' weights down to zero. * In general, Elastic Net is preferred over Lasso since Lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

```
[ ]: np.random.seed(42)
     def warn(*args, **kwargs):
         pass
     import warnings
     warnings.warn = warn




     from sklearn.linear_model import ElasticNet
     from sklearn.model_selection import cross_val_score
     m = 70
     X = 3 * np.random.rand(m, 1)
     y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
     poly_features = PolynomialFeatures(degree=10, include_bias=False)
     X_poly = poly_features.fit_transform(X)
     scaler = StandardScaler()
     X_poly_std = scaler.fit_transform(X_poly)
     valvec = []
     elnet = ElasticNet(alpha = 0.1, l1_ratio = 0.5, max_iter=1000000 )
     cores = -cross_val_score(elnet, X_poly_std, y, cv=5,␣
      ↪scoring='neg_mean_squared_error')
     alphavec = np.arange(0, 1, 0.01)
     l1ratios = [0, 0.1, 0.4, 0.7, 1]
     for l1rat, style in zip(l1ratios, ("b", "g", "r","k", "m")):
         mean_score = []
         min_score = 100
         for alpha in alphavec:
             elnet = ElasticNet(alpha = alpha, l1_ratio = l1rat,  normalize=True) if␣
      ↪alpha > 0 else LinearRegression()
             score = -cross_val_score(elnet, X_poly, y, cv=5,␣
      ↪scoring='neg_mean_squared_error')
             mean_score.append(np.mean(score))
             if np.mean(score) < min_score:
                 min_score = np.mean(score)
                 best_alpha = alpha
```

```
            best_l1 = l1rat
    plt.plot(alphavec, mean_score, style, label=r"l1_ratio = {}".format(l1rat))
plt.legend(loc="bottom right", fontsize=10)
plt.xlabel("$x_1$", fontsize=18)
print("best model has alpha =", best_alpha, "l1_ratio =" ,l1rat)
```

```
best model has alpha = 0.01 l1_ratio = 1
```

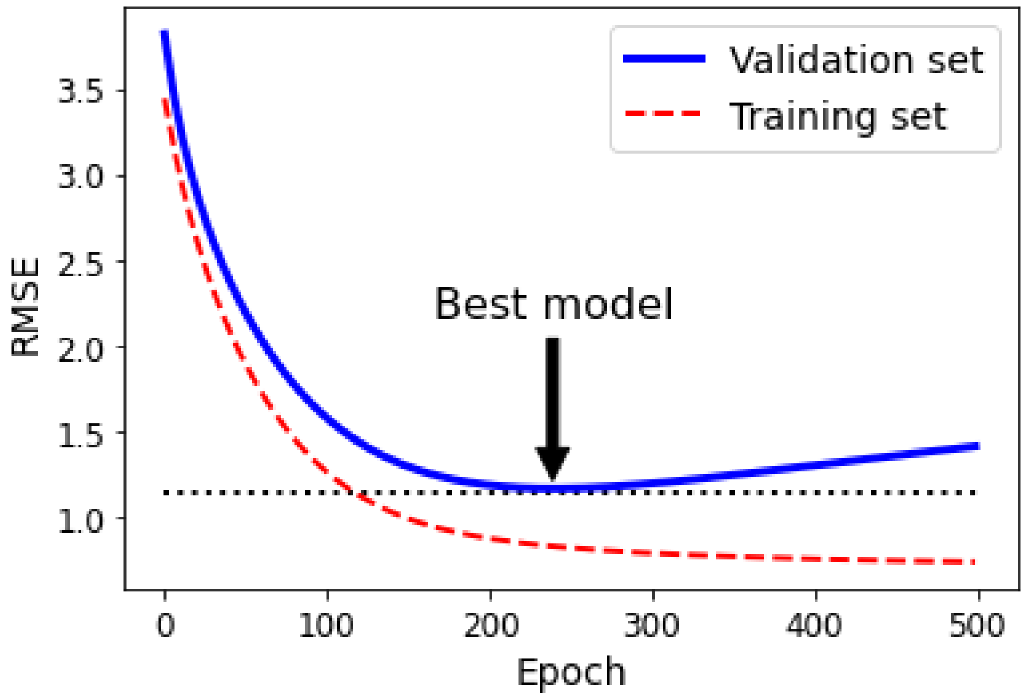


```
[ ]:
```

12 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 next shows a complex model (in this case a high-degree Polynomial Regression model) being trained using Batch Gradient Descent. As the epochs go by, the algorithm learns and its prediction error (RMSE) on the training set naturally goes down, and so does its prediction error on the validation set. However, after a while the validation error stops decreasing and actually starts to go back up. This indicates that the model has started to overfit the training data. With early stopping you just stop training as soon as the validation error reaches the minimum.

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

     X_train, X_val, y_train, y_val = train_test_split(X[:50], y[:50].ravel(),␣
      ↪test_size=0.5, random_state=10)

     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,
                            penalty=None,
                            eta0=0.0005,
                            warm_start=True,
                            learning_rate="constant",
                            random_state=42)

     n_epochs = 500
     train_errors, val_errors = [], []
     for epoch in range(n_epochs):
         sgd_reg.fit(X_train_poly_scaled, y_train)
         y_train_predict = sgd_reg.predict(X_train_poly_scaled)
         y_val_predict = sgd_reg.predict(X_val_poly_scaled)
         train_errors.append(mean_squared_error(y_train, y_train_predict))
         val_errors.append(mean_squared_error(y_val, y_val_predict))

     best_epoch = np.argmin(val_errors)
     best_val_rmse = np.sqrt(val_errors[best_epoch])

     plt.annotate('Best model',
                  xy=(best_epoch, best_val_rmse),
                  xytext=(best_epoch, best_val_rmse + 1),
                  ha="center",
                  arrowprops=dict(facecolor='black', shrink=0.05),
                  fontsize=16,
                 )

     best_val_rmse -= 0.03  # just to make the graph look better
     plt.plot([0, n_epochs], [best_val_rmse, best_val_rmse], "k:", linewidth=2)
     plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Validation set")
     plt.plot(np.sqrt(train_errors), "r--", linewidth=2, label="Training set")
     plt.legend(loc="upper right", fontsize=14)
     plt.xlabel("Epoch", fontsize=14)
     plt.ylabel("RMSE", fontsize=14)
     plt.show()
```



13 Logistic regression

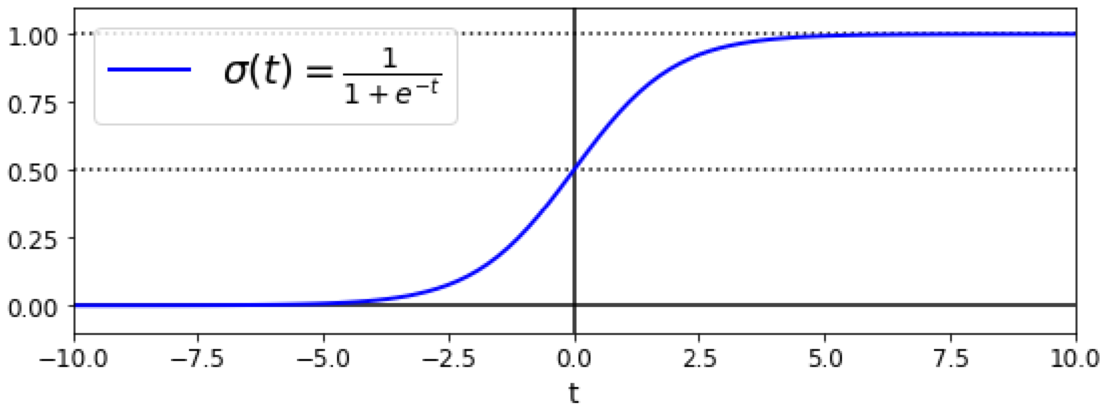
Some regression algorithms can be used for classification as well (and vice versa). Logistic Regression (also called Logit Regression) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled "1"), or else it predicts that it does not (i.e., it belongs to the negative class, labeled "0"). This makes it a binary classifier

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

Logit function is:

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

[ ]:
```
t = np.linspace(-10, 10, 100)
sig = 1 / (1 + np.exp(-t))
plt.figure(figsize=(9, 3))
plt.plot([-10, 10], [0, 0], "k-")
plt.plot([-10, 10], [0.5, 0.5], "k:")
plt.plot([-10, 10], [1, 1], "k:")
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(t, sig, "b-", linewidth=2, label=r"$\sigma(t) = \frac{1}{1 + e^{-t}}$")
plt.xlabel("t")
plt.legend(loc="upper left", fontsize=20)
plt.axis([-10, 10, -0.1, 1.1])
plt.show()
```



if $\hat{p} < 0.5$ then $\hat{y} = 0$ if $\hat{p} > 0.5$ then $\hat{y} = 1$

14 Training and 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 ($y = 0$). This idea is captured by the cost function shown:

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

$-log(\hat{p})$ grows very large as $\hat{p}$ approaches 0, and $-log(1-\hat{p})$ grows very large as $\hat{p}$ approaches 1. The cost of the training set is the sum of the costs function for all instances:

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

There is no closed form solution. However, the gradient is convex, so we can find global minimum. The partial deriviatives of the cost function with regards to the jth model parameters θ_j is:

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

After we have vector of partial deriviatives we can use it in the Batch Gradient Descent algorithm, or mini-Batch or Stochastic.

Decisions Boundaries

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. We will build a classifier to detect the Iris-Virginica type based only on the petal width feature. First let's load the data:

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

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

[ ]:
```

```

[ ]:
```
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more …
```

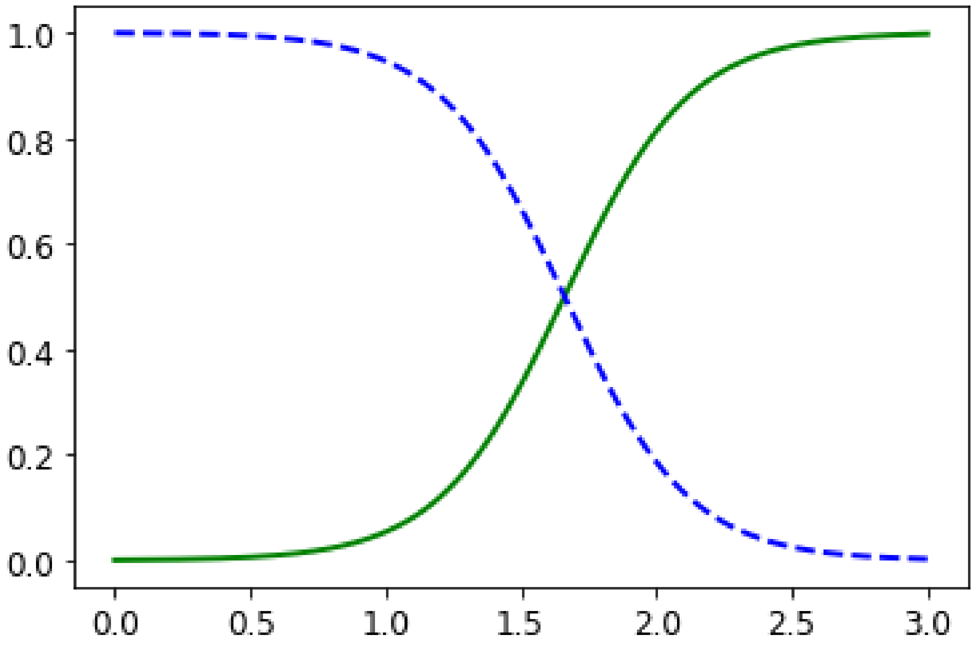
[ ]:
```python
import numpy as np
# Set one X - petal width
X = iris["data"][:, 3:]  # petal width
# Y is the classification of Iris-Virginica.
y = (iris["target"] == 2).astype(np.int)  # 1 if Iris-Virginica, else 0
```

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

[ ]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                    warm_start=False)
```

```
[ ]: # Green is the probability of Iris-Virginica, blue is the probability of Not␣
      ↪Iris-Virginica.
     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")
```

```
[ ]: [<matplotlib.lines.Line2D at 0x7f4309e00908>]
```

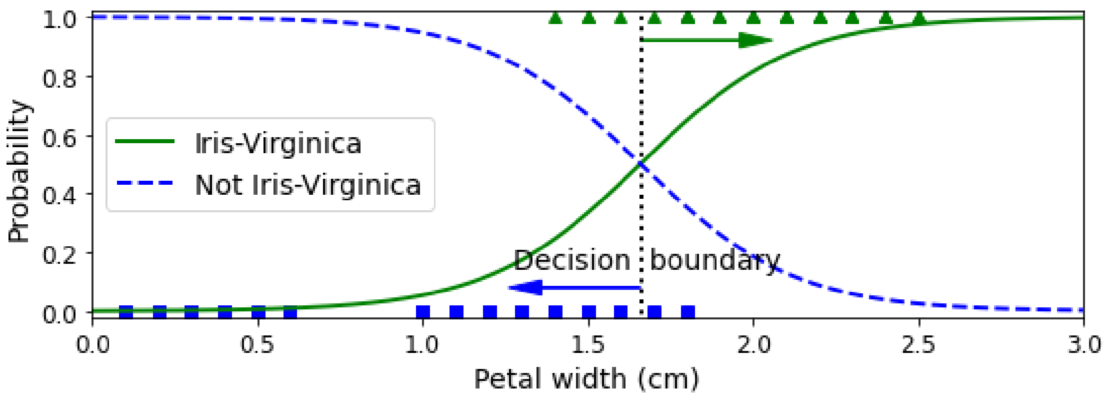


The figure in the book actually is actually a bit fancier:

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

     plt.figure(figsize=(8, 3))
     plt.plot(X[y==0], y[y==0], "bs")
     plt.plot(X[y==1], y[y==1], "g^")
     plt.plot([decision_boundary, decision_boundary], [-1, 2], "k:", linewidth=2)
     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")
     plt.text(decision_boundary+0.02, 0.15, "Decision  boundary", fontsize=14,␣
      ↪color="k", ha="center")
```

```
plt.arrow(decision_boundary, 0.08, -0.3, 0, head_width=0.05, head_length=0.1,␣
 ↪fc='b', ec='b')
plt.arrow(decision_boundary, 0.92, 0.3, 0, head_width=0.05, head_length=0.1,␣
 ↪fc='g', ec='g')
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 3, -0.02, 1.02])
save_fig("logistic_regression_plot")
plt.show()
```

```
Saving figure logistic_regression_plot
```



The petal width of Iris-Virginica flowers (represented by triangles) ranges from 1.4 cm to 2.5 cm, while the other iris flowers (represented by squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm. Notice that there is a bit of overlap. Above about 2 cm the classifier is highly confident that the flower is an Iris-Virginica (it outputs a high probability to that class), while below 1 cm it is highly confident that it is not an Iris-Virginica (high probability for the “Not Iris-Virginica” class). In between these extremes, the classifier is unsure.

```
[ ]: # 50% probability is at
     decision_boundary
```

```
[ ]: array([1.66066066])
```

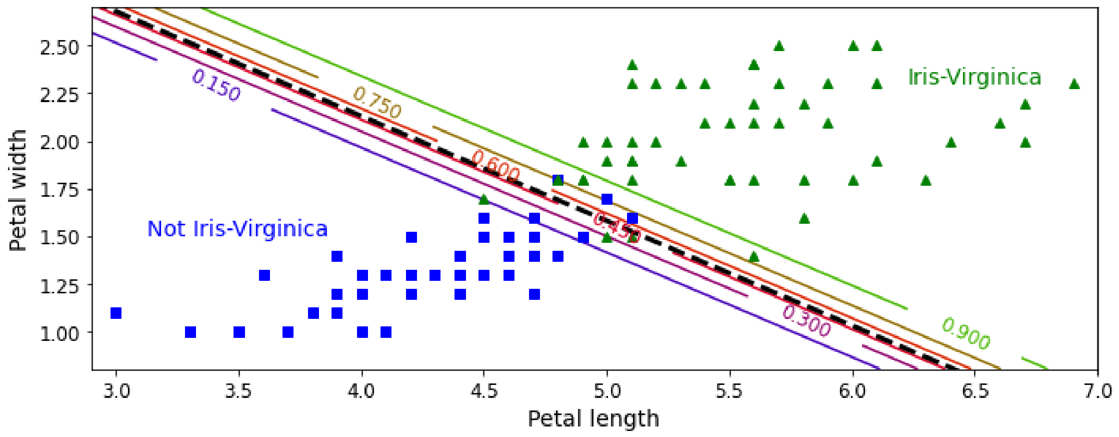
```
[ ]: # Predict with sepal lenths of 1.7 and 1.5.
     log_reg.predict([[1.7], [1.5]])
```

```
[ ]: array([1, 0])
```

Next we will show the same dataset but this time displaying two features: petal width and length. Once trained, the Logistic Regression classifier can estimate the probability that a new flower is an Iris- Virginica based on these two features. The dashed line represents the points where the model estimates a 50% probability: this is the model's decision boundary. Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right). All the flowers beyond the top-right line have an over 90% chance of being Iris-Virginica according to the model.

```
[ ]: from sklearn.linear_model import LogisticRegression

     X = iris["data"][:, (2, 3)]  # petal length, petal width
     y = (iris["target"] == 2).astype(np.int)
     # C is the regularization parameter, we will cover it more in the next lecture
     log_reg = LogisticRegression(C=10**10, random_state=42)
     log_reg.fit(X, y)

     x0, x1 = np.meshgrid(
             np.linspace(2.9, 7, 500).reshape(-1, 1),
             np.linspace(0.8, 2.7, 200).reshape(-1, 1),
         )
     X_new = np.c_[x0.ravel(), x1.ravel()]

     y_proba = log_reg.predict_proba(X_new)

     plt.figure(figsize=(10, 4))
     plt.plot(X[y==0, 0], X[y==0, 1], "bs")
     plt.plot(X[y==1, 0], X[y==1, 1], "g^")

     zz = y_proba[:, 1].reshape(x0.shape)
     contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)


     left_right = np.array([2.9, 7])
     boundary = -(log_reg.coef_[0][0] * left_right + log_reg.intercept_[0]) /␣
      ↪log_reg.coef_[0][1]

     plt.clabel(contour, inline=1, fontsize=12)
     plt.plot(left_right, boundary, "k--", linewidth=3)
     plt.text(3.5, 1.5, "Not Iris-Virginica", fontsize=14, color="b", ha="center")
     plt.text(6.5, 2.3, "Iris-Virginica", fontsize=14, color="g", ha="center")
     plt.xlabel("Petal length", fontsize=14)
     plt.ylabel("Petal width", fontsize=14)
     plt.axis([2.9, 7, 0.8, 2.7])
     save_fig("logistic_regression_contour_plot")
     plt.show()
```

```
Saving figure logistic_regression_contour_plot
```



15 Multinomial Logit 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. For each instance x the 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:

$$s_k(x) = \theta_k^T \cdot x$$

Each class k has it's own vector of parameters θ_k. For each instance x we compute a probability $\hat{p}_k$ of belonging to class k. The probability is the ratio of the score k and the sum of score of all other classes:

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

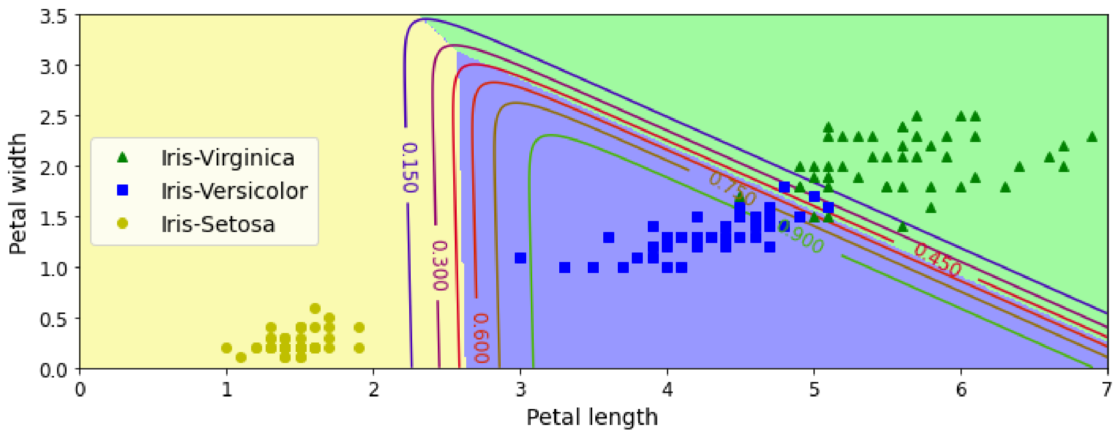
Multinomial logit predict class k according to the highest probability. In the estimation of multinomial logit we minimize a cost function similar to the binomial logit.

```
[ ]: # now we estimate multinomial logit using three classes:
     X = iris["data"][:, (2, 3)]  # petal length, petal width
     y = iris["target"]

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

```
[ ]: LogisticRegression(C=10, class_weight=None, dual=False, fit_intercept=True,
                        intercept_scaling=1, l1_ratio=None, max_iter=100,
                        multi_class='multinomial', n_jobs=None, penalty='l2',
                        random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                        warm_start=False)
```

```
[ ]: x0, x1 = np.meshgrid(
             np.linspace(0, 8, 500).reshape(-1, 1),
             np.linspace(0, 3.5, 200).reshape(-1, 1),
         )
     X_new = np.c_[x0.ravel(), x1.ravel()]


     y_proba = softmax_reg.predict_proba(X_new)
     y_predict = softmax_reg.predict(X_new)

     zz1 = y_proba[:, 1].reshape(x0.shape)
     zz = y_predict.reshape(x0.shape)

     plt.figure(figsize=(10, 4))
     plt.plot(X[y==2, 0], X[y==2, 1], "g^", label="Iris-Virginica")
     plt.plot(X[y==1, 0], X[y==1, 1], "bs", label="Iris-Versicolor")
     plt.plot(X[y==0, 0], X[y==0, 1], "yo", label="Iris-Setosa")

     from matplotlib.colors import ListedColormap
     custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])

     plt.contourf(x0, x1, zz, cmap=custom_cmap)
     contour = plt.contour(x0, x1, zz1, cmap=plt.cm.brg)
     plt.clabel(contour, inline=1, fontsize=12)
     plt.xlabel("Petal length", fontsize=14)
     plt.ylabel("Petal width", fontsize=14)
     plt.legend(loc="center left", fontsize=14)
     plt.axis([0, 7, 0, 3.5])
     save_fig("softmax_regression_contour_plot")
     plt.show()
```

```
Saving figure softmax_regression_contour_plot
```



[ ]:
```
softmax_reg.predict([[5, 2]])
```

[ ]:
```
array([2])
```

[ ]:
```
softmax_reg.predict_proba([[5, 2]])
```

[ ]:
```
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```