Chapter 7. Regression

You probably learned about regression in your high school mathematics class. The specific method you learned was probably what is called **ordinary least squares** (**OLS**) regression. This 200-year-old technique is computationally fast and can be used for many real-world problems. This chapter will start by reviewing it and showing you how it is available in scikit-learn.

For some problems, however, this method is insufficient. This is particularly true when we have many features, and it completely fails when we have more features than datapoints. For those cases, we need more advanced methods. These methods are very modern, with major developments happening in the last decade. They go by names such as Lasso, Ridge, or ElasticNets. We will go into these in detail. They are also available in scikit-learn.

Predicting house prices with regression

Let's start with a simple problem, predicting house prices in Boston; a problem for which we can use a publicly available dataset. We are given several demographic and geographical attributes, such as the crime rate or the pupil-teacher ratio in the neighborhood. The goal is to predict the median value of a house in a particular area. As usual, we have some training data, where the answer is known to us.

This is one of the built-in datasets that scikit-learn comes with, so it is very easy to load the data into memory:

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
```

The `boston` object contains several attributes; in particular, `boston.data` contains the input data and `boston.target` contains the price of houses.

We will start with a simple one-dimensional regression, trying to regress the price on a single attribute, the average number of rooms per dwelling in the neighborhood, which is stored at position `5` (you can consult `boston.DESCR` and `boston.feature_names` for detailed information on the data):

```
>>> from matplotlib import pyplot as plt
>>> plt.scatter(boston.data[:,5], boston.target, color='r')
```

The `boston.target` attribute contains the average house price (our target variable).

We can use the standard least squares regression you probably first saw in high-school. Our first attempt looks like this:

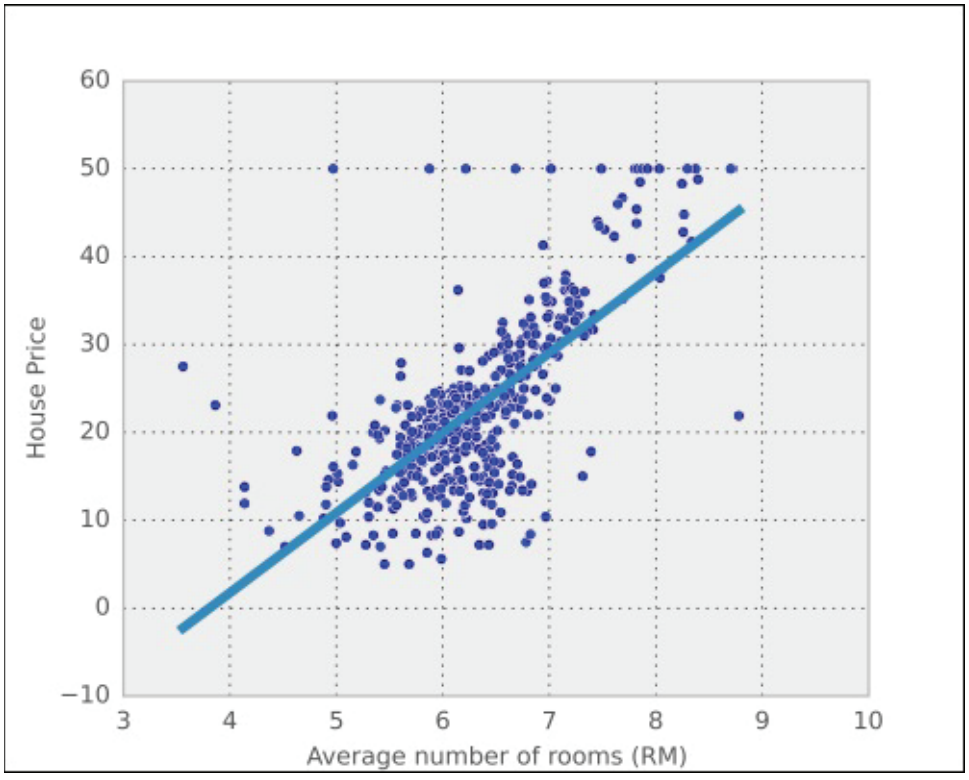
```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression()
```

We import `LinearRegression` from the `sklearn.linear_model` module and construct a `LinearRegression` object. This object will behave analogously to the classifier objects from scikit-learn that we used earlier.

```
>>> x = boston.data[:,5]
>>> y = boston.target
>>> x = np.transpose(np.atleast_2d(x))
>>> lr.fit(x, y)
>>> y_predicted = lr.predict(x)
```

The only nonobvious line in this code block is the call to `np.atleast_2d`, which converts `x` from a one-dimensional to a two-dimensional array. This conversion is necessary as the `fit` method expects a two-dimensional array as its first argument. Finally, for the dimensions to work out correctly, we need to transpose this array.

Note that we are calling methods named `fit` and predict on the `LinearRegression` object, just as we did with classifier objects, even though we are now performing regression. This regularity in the API is one of the nicer features of scikit-learn.



The preceding graph shows all the points (as dots) and our fit (the solid line). We can

see that visually it looks good, except for a few outliers.

Ideally, though, we would like to measure how good of a fit this is quantitatively. This will be critical in order to be able to compare alternative methods. To do so, we can measure how close our prediction is to the true values. For this task, we can use the `mean_squared_error` function from the `sklearn.metrics` module:

```
>>> from sklearn.metrics import mean_squared_error
```

This function takes two arguments, the true value and the predictions, as follows:

```
>>> mse = mean_squared_error(y, lr.predict(x))
>>> print("Mean squared error (of training data):
{:.3}".format(mse))
Mean squared error (of training data): 58.4
```

This value can sometimes be hard to interpret, and it's better to take the square root, to obtain the **root mean square error** (**RMSE**):

```
>>> rmse = np.sqrt(mse)
>>> print("RMSE (of training data): {:.3}".format(rmse))
RMSE (of training data): 6.6
```

One advantage of using RMSE is that we can quickly obtain a very rough estimate of the error by multiplying it by two. In our case, we can expect the estimated price to be different from the real price by, at most, 13 thousand dollars.

Tip

Root mean squared error and prediction

Root mean squared error corresponds approximately to an estimate of the standard deviation. Since most data is at most two standard deviations from the mean, we can double our RMSE to obtain a rough confident interval. This is only completely valid if the errors are normally distributed, but it is often roughly correct even if they are not.

A number such as 6.6 is still hard to immediately intuit. Is this a good prediction? One possible way to answer this question is to compare it with the most simple baseline, the constant model. If we knew nothing of the input, the best we could do is predict that the output will always be the average value of `y`. We can then compare the mean-squared error of this model with the mean-squared error of the null model. This

idea is formalized in the **coefficient of determination**, which is defined as follows:

$$1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \approx 1 - \frac{\text{MSE}}{\text{VAR}(y)}$$

In this formula, y_i represents the value of the element with index *i*, while $\overline{y}_i$ is the estimate for the same element obtained by the regression model. Finally, $\overline{y}$ is the mean value of *y*, which represents the *null model* that always returns the same value. This is roughly the same as first computing the ratio of the mean squared error with the variance of the output and, finally, considering one minus this ratio. This way, a perfect model obtains a score of one, while the null model obtains a score of zero. Note that it is possible to obtain a negative score, which means that the model is so poor that one is better off using the mean as a prediction.

The coefficient of determination can be obtained using `r2_score` of the `sklearn.metrics` module:

```
>>> from sklearn.metrics import r2_score
>>> r2 = r2_score(y, lr.predict(x))
>>> print("R2 (on training data): {:.2}".format(r2))
R2 (on training data): 0.31
```

This measure is also called the R^2 score. If you are using linear regression and evaluating the error on the training data, then it does correspond to the square of the correlation coefficient, R. However, this measure is more general, and as we discussed, may even return a negative value.

An alternative way to compute the coefficient of determination is to use the `score` method of the `LinearRegression` object:

```
>>> r2 = lr.score(x,y)
```

Multidimensional regression

So far, we have only used a single variable for prediction, the number of rooms per dwelling. We will now use all the data we have to fit a model, using multidimensional regression. We now try to predict a single output (the average house price) based on multiple inputs.

The code looks very much like before. In fact, it's even simpler as we can now pass the value of `boston.data` directly to the `fit` method:

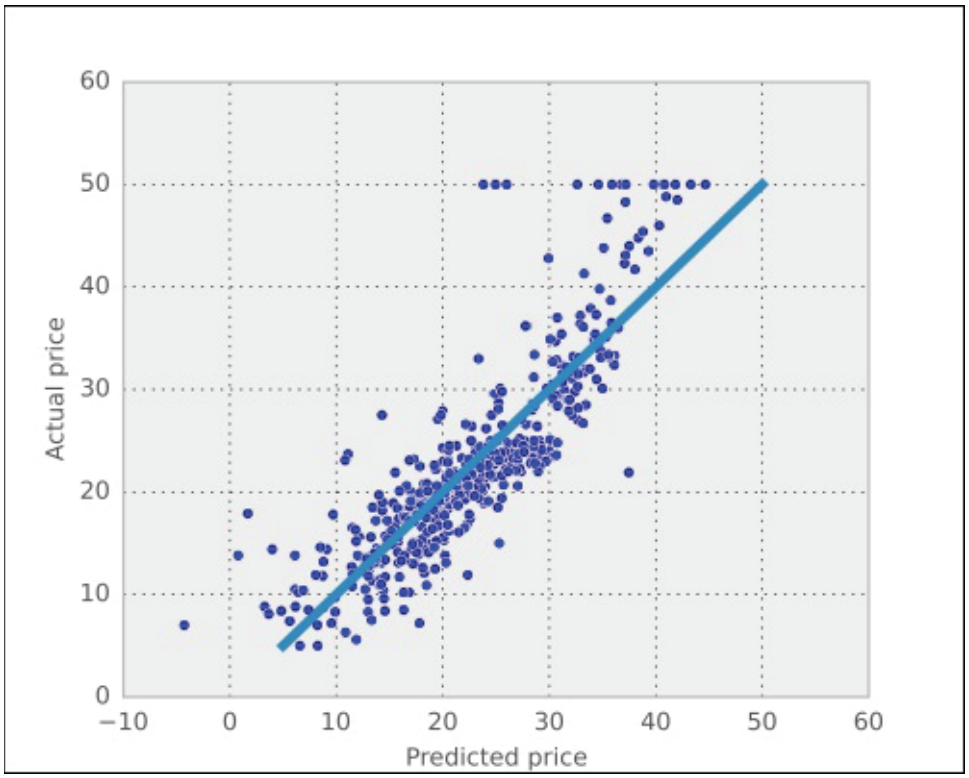
```
>>> x = boston.data
>>> y = boston.target
>>> lr.fit(x, y)
```

Using all the input variables, the root mean squared error is only 4.7, which corresponds to a coefficient of determination of 0.74. This is better than what we had before, which indicates that the extra variables did help. We can no longer easily display the regression line as we did, because we have a 14-dimensional regression hyperplane instead of a single line.

We can, however, plot the prediction versus the actual value. The code is as follows:

```
>>> p = lr.predict(x)
>>> plt.scatter(p, y)
>>> plt.xlabel('Predicted price')
>>> plt.ylabel('Actual price')
>>> plt.plot([y.min(), y.max()], [[y.min()], [y.max()]])
```

The last line plots a diagonal line that corresponds to perfect agreement. This aids with visualization. The results are shown in the following plot, where the solid line shows the diagonal (where all the points would lie if there was perfect agreement between the prediction and the underlying value):



Cross-validation for regression

If you remember when we first introduced classification, we stressed the importance of cross-validation for checking the quality of our predictions. In regression, this is not always done. In fact, we discussed only the training error in this chapter so far. This is a mistake if you want to confidently infer the generalization ability. Since ordinary least squares is a very simple model, this is often not a very serious mistake. In other words, the amount of overfitting is slight. However, we should still test this empirically, which we can easily do with scikit-learn.

We will use the `Kfold` class to build a 5 fold cross-validation loop and test the generalization ability of linear regression:

```
>>> from sklearn.cross_validation import Kfold
>>> kf = KFold(len(x), n_folds=5)
>>> p = np.zeros_like(y)
>>> for train,test in kf:
...    lr.fit(x[train], y[train])
...    p[test] = lr.predict(x[test])
>>> rmse_cv = np.sqrt(mean_squared_error(p, y))
>>> print('RMSE on 5-fold CV: {:.2}'.format(rmse_cv))
RMSE on 5-fold CV: 5.6
```

With cross-validation, we obtain a more conservative estimate (that is, the error is larger): `5.6`. As in the case of classification, the cross-validation estimate is a better estimate of how well we could generalize to predict on unseen data.

Ordinary least squares is fast at learning time and returns a simple model, which is fast at prediction time. For these reasons, it should often be the first model that you try in a regression problem. However, we are now going to see more advanced methods and why they are sometimes preferable.

Penalized or regularized regression

This section introduces penalized regression, also called **regularized regression**, an important class of regression models.

In ordinary regression, the returned fit is the best fit on the training data. This can lead to over-fitting. Penalizing means that we add a penalty for over-confidence in the parameter values. Thus, we accept a slightly worse fit in order to have a simpler model.

Another way to think about it is to consider that the default is that there is no relationship between the input variables and the output prediction. When we have data, we change this opinion, but adding a penalty means that we require more data to convince us that this is a strong relationship.

Tip

Penalized regression is about tradeoffs

Penalized regression is another example of the bias-variance tradeoff. When using a penalty, we get a worse fit in the training data, as we are adding bias. On the other hand, we reduce the variance and tend to avoid over-fitting. Therefore, the overall result might generalize better to unseen (test) data.

L1 and L2 penalties

We now explore these ideas in detail. Readers who do not care about some of the mathematical aspects should feel free to skip directly to the next section on how to use regularized regression in scikit-learn.

The problem, in general, is that we are given a matrix *X* of training data (rows are observations and each column is a different feature), and a vector *y* of output values. The goal is to obtain a vector of weights, which we will call *b**. The ordinary least squares regression is given by the following formula:

$$\vec{b}^* = \arg\min_{\vec{b}} \|\vec{y} - X\vec{b}\|^2$$

That is, we find vector *b* that minimizes the squared distance to the target *y*. In these equations, we ignore the issue of setting an intercept by assuming that the training

data has been preprocessed so that the mean of *y* is zero.

Adding a penalty or a regularization means that we do not simply consider the best fit on the training data, but also how vector $\vec{b}$ is composed. There are two types of penalties that are typically used for regression: L1 and L2 penalties. An L1 penalty means that we penalize the regression by the sum of the absolute values of the coefficients, while an L2 penalty penalizes by the sum of squares.

When we add an L1 penalty, instead of the preceding equation, we instead optimize the following:

$$\vec{b}^* = \arg\min_{\vec{b}} \|\vec{y} - X\vec{b}\|^2 + \alpha \sum_i |b_i|$$

Here, we are trying to simultaneously make the error small, but also make the values of the coefficients small (in absolute terms). Using an L2 penalty, means that we use the following formula:

$$\vec{b}^* = \arg\min_{\vec{b}} \|\vec{y} - X\vec{b}\|^2 + \alpha \sum_i b_i^2$$

The difference is rather subtle: we now penalize by the square of the coefficient rather than their absolute value. However, the difference in the results is dramatic.

Tip

Ridge, Lasso, and ElasticNets

These penalized models often go by rather interesting names. The L1 penalized model is often called the **Lasso**, while an L2 penalized one is known as **Ridge Regression**. When using both, we call this an **ElasticNet** model.

Both the Lasso and the Ridge result in smaller coefficients than unpenalized regression (smaller in absolute value, ignoring the sign). However, the Lasso has the additional property that it results in many coefficients being set to exactly zero! This means that the final model does not even use some of its input features, the model is **sparse**. This is often a very desirable property as the model performs both feature selection and **regression** in a single step.

You will notice that whenever we add a penalty, we also add a weight *α*, which governs how much penalization we want. When *α* is close to zero, we are very close to unpenalized regression (in fact, if you set *α* to zero, you will simply perform OLS), and when *α* is large, we have a model that is very different from the unpenalized one.

The Ridge model is older as the Lasso is hard to compute with pen and paper. However, with modern computers, we can use the Lasso as easily as Ridge, or even combine them to form ElasticNets. An ElasticNet has two penalties, one for the absolute value and the other for the squares and it solves the following equation:

$$\vec{b}^* = \arg\min_{\vec{b}} \|\vec{y} - X\vec{b}\|^2 + \alpha_1 \sum_i |b_i| + \alpha_2 \sum_i b_i^2$$

This formula is a combination of the two previous ones, with two parameters, α_1 and α_2. Later in this chapter, we will discuss how to choose a good value for parameters.

Using Lasso or ElasticNet in scikit-learn

Let's adapt the preceding example to use ElasticNets. Using scikit-learn, it is very easy to swap in the ElasticNet regressor for the least squares one that we had before:

```
>>> from sklearn.linear_model import ElasticNet, Lasso
>>> en = ElasticNet(alpha=0.5)
```

Now, we use `en`, whereas earlier we had used `lr`. This is the only change that is needed. The results are exactly what we would have expected. The training error increases to 5.0 (it was 4.6 before), but the cross-validation error decreases to 5.4 (it was 5.6 before). We trade a larger error on the training data in order to gain better generalization. We could have tried an L1 penalty using the `Lasso` class or L2 using the `Ridge` class with the same code.

Visualizing the Lasso path

Using scikit-learn, we can easily visualize what happens as the value of the regularization parameter (alpha) changes. We will again use the Boston data, but now we will use the `Lasso` regression object:

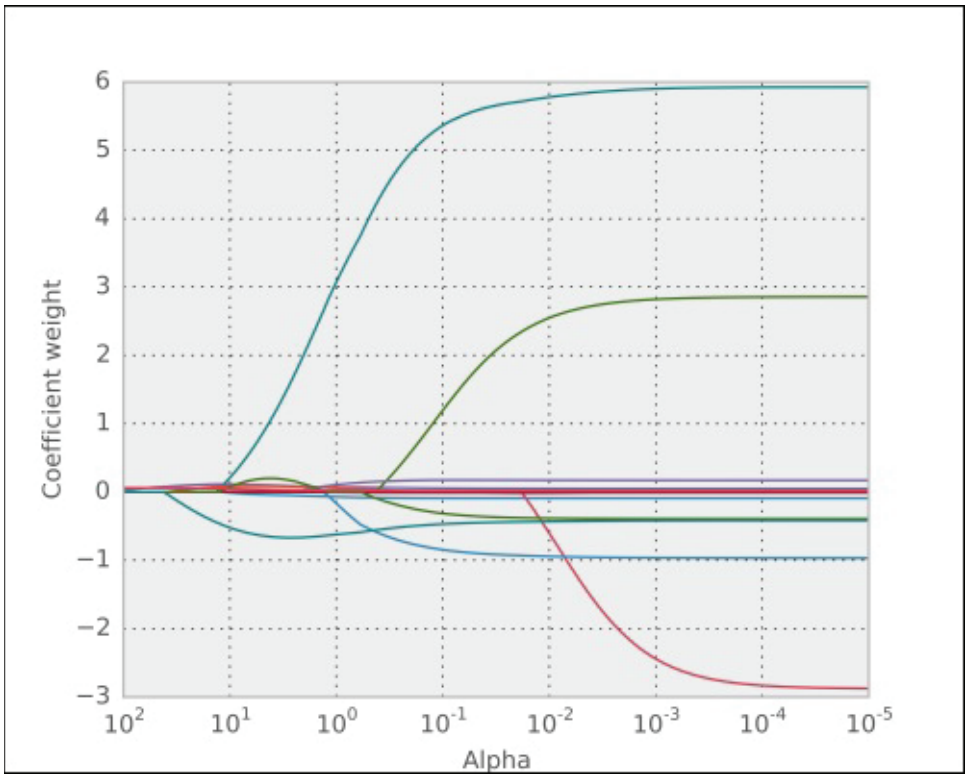
```
>>> las = Lasso(normalize=1)
>>> alphas = np.logspace(-5, 2, 1000)
>>> alphas, coefs, _= las.path(x, y, alphas=alphas)
```

For each value in alphas, the `path` method on the `Lasso` object returns the coefficients that solve the lasso problem with that parameter value. Because the result changes smoothly with alpha, this can be computed very efficiently.

A typical way to visualize this path is to plot the value of the coefficients as alpha decreases. You can do so as follows:

```
>>> fig,ax = plt.subplots()
>>> ax.plot(alphas, coefs.T)
>>> # Set log scale
>>> ax.set_xscale('log')
>>> # Make alpha decrease from left to right
>>> ax.set_xlim(alphas.max(), alphas.min())
```

This results in the following plot (we left out the trivial code that adds axis labels and the title):



In this plot, the *x* axis shows decreasing amounts of regularization from left to right (alpha is decreasing). Each line shows how a different coefficient varies as alpha changes. The plot shows that when using very strong regularization (left side, very high alpha), the best solution is to have all values be exactly zero. As the regularization becomes weaker, one by one, the values of the different coefficients first shoot up, then stabilize. At some point, they all plateau as we are probably already close to the unpenalized solution.

P-greater-than-N scenarios

The title of this section is a bit of inside jargon, which you will learn now. Starting in the 1990s, first in the biomedical domain, and then on the Web, problems started to appear where P was greater than N. What this means is that the number of features, P, was greater than the number of examples, N (these letters were the conventional statistical shorthand for these concepts). These became known as *P greater than N*

problems.

For example, if your input is a set of written documents, a simple way to approach it is to consider each possible word in the dictionary as a feature and regress on those (we will later work on one such problem ourselves). In the English language, you have over 20,000 words (this is if you perform some stemming and only consider common words; it is more than ten times that if you skip this preprocessing step). If you only have a few hundred or a few thousand examples, you will have more features than examples.

In this case, as the number of features is greater than the number of examples, it is possible to have a perfect fit on the training data. This is a mathematical fact, which is independent of your data. You are, in effect, solving a system of linear equations with fewer equations than variables. You can find a set of regression coefficients with zero training error (in fact, you can find more than one perfect solution, infinitely many).

However, and this is a major problem, *zero training error does not mean that your solution will generalize well*. In fact, it may generalize very poorly. Whereas earlier regularization could give you a little extra boost, it is now absolutely required for a meaningful result.

An example based on text documents

We will now turn to an example that comes from a study performed at Carnegie Mellon University by Prof. Noah Smith's research group. The study was based on mining the so-called 10-K reports that companies file with the **Securities and Exchange Commission** (**SEC**) in the United States. This filing is mandated by law for all publicly traded companies. The goal of their study was to predict, based on this piece of public information, what the future volatility of the company's stock will be. In the training data, we are actually using historical data for which we already know what happened.

There are 16,087 examples available. The features, which have already been preprocessed for us, correspond to different words, 150,360 in total. Thus, we have many more features than examples, almost ten times as much. In the introduction, it was stated that ordinary least regression fails in these cases and we will now see why by attempting to blindly apply it.

The dataset is available in SVMLight format from multiple sources, including the book's companion website. This is a format that scikit-learn can read. SVMLight is, as the name says, a support vector machine implementation, which is also available through scikit-learn; right now, we are only interested in the file format:

```
>>> from sklearn.datasets import load_svmlight_file
>>> data,target = load_svmlight_file('E2006.train')
```

In the preceding code, data is a sparse matrix (that is, most of its entries are zeros and, therefore, only the nonzero entries are saved in memory), while the target is a simple one-dimensional vector. We can start by looking at some attributes of the target:

```
>>> print('Min target value: {}'.format(target.min()))
Min target value: -7.89957807347
>>> print('Max target value: {}'.format(target.max()))
Max target value: -0.51940952694
>>> print('Mean target value: {}'.format(target.mean()))
Mean target value: -3.51405313669
>>> print('Std. dev. target: {}'.format(target.std()))
Std. dev. target: 0.632278353911
```

So, we can see that the data lies between -7.9 and -0.5. Now that we have a feel for the data, we can check what happens when we use OLS to predict. Note that we can use exactly the same classes and methods as we did earlier:

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression()
>>> lr.fit(data,target)
>>> pred = lr.predict(data)
>>> rmse_train = np.sqrt(mean_squared_error(target, pred))
>>> print('RMSE on training: {:.2}'.format(rmse_train))
RMSE on training: 0.0025
>>> print('R2 on training: {:.2}'.format(r2_score(target,
pred)))
R2 on training: 1.0
```

The root mean squared error is not exactly zero because of rounding errors, but it is very close. The coefficient of determination is `1.0`. That is, the linear model is reporting a perfect prediction on its training data.

When we use cross-validation (the code is very similar to what we used earlier in the Boston example), we get something very different: RMSE of 0.75, which corresponds to a negative coefficient of determination of -0.42. This means that if we always "predict" the mean value of -3.5, we do better than when using the regression model!

Tip

Training and generalization error

When the number of features is greater than the number of examples, you always get zero training errors with OLS, except perhaps for issues due to rounding off. However, this is rarely a sign that your model will do well in terms of generalization. In fact, you may get zero training error and have a completely useless model.

The natural solution is to use regularization to counteract the overfitting. We can try the same cross-validation loop with an ElasticNet learner, having set the penalty parameter to `0.1`:

```
>>> from sklearn.linear_model import ElasticNet
>>> met = ElasticNet(alpha=0.1)

>>> kf = KFold(len(target), n_folds=5)
>>> pred = np.zeros_like(target)
>>> for train, test in kf:
...    met.fit(data[train], target[train])
...    pred[test] = met.predict(data[test])

>>> # Compute RMSE
>>> rmse = np.sqrt(mean_squared_error(target, pred))
>>> print('[EN 0.1] RMSE on testing (5 fold):
{:.2}'.format(rmse))
[EN 0.1] RMSE on testing (5 fold): 0.4

>>> # Compute Coefficient of determination
>>> r2 = r2_score(target, pred)
>>> print('[EN 0.1] R2 on testing (5 fold): {:.2}'.format(r2))
[EN 0.1] R2 on testing (5 fold): 0.61
```

Now, we get `0.4` RMSE and an R2 of `0.61`, much better than just predicting the mean. There is one problem with this solution, though, which is the choice of alpha. When using the default value (`1.0`), the result is very different (and worse).

In this case, we cheated as the author had previously tried a few values to see which ones would give a good result. This is not effective and can lead to over estimates of confidence (we are looking at the test data to decide which parameter values to use and which we should never use). The next section explains how to do it properly and how this is supported by scikit-learn.

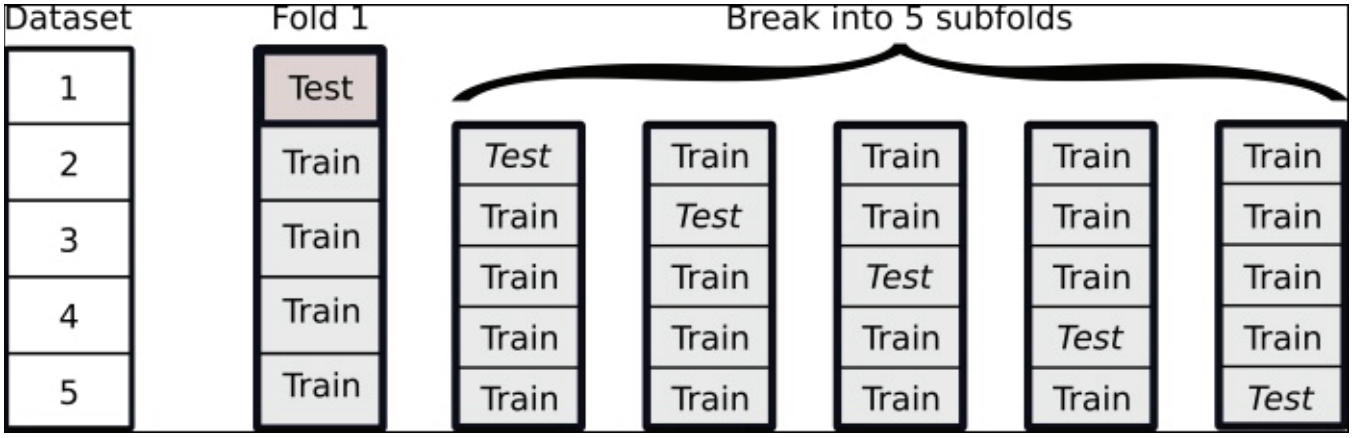
Setting hyperparameters in a principled way

In the preceding example, we set the penalty parameter to `0.1`. We could just as well have set it to 0.7 or 23.9. Naturally, the results vary each time. If we pick an overly large value, we get underfitting. In the extreme case, the learning system will just return every coefficient equal to zero. If we pick a value that is too small, we are very

close to OLS, which overfits and generalizes poorly (as we saw earlier).

How do we choose a good value? This is a general problem in machine learning: setting parameters for our learning methods. A generic solution is to use cross-validation. We pick a set of possible values, and then use cross-validation to choose which one is best. This performs more computation (five times more if we use five folds), but is always applicable and unbiased.

We must be careful, though. In order to obtain an estimate of generalization, we have to use **two-levels of cross-validation**: one level is to estimate the generalization, while the second level is to get good parameters. That is, we split the data in, for example, five folds. We start by holding out the first fold and will learn on the other four. Now, we split these again into 5 folds in order to choose the parameters. Once we have set our parameters, we test on the first fold. Now, we repeat this four other times:



The preceding figure shows how you break up a single training fold into subfolds. We would need to repeat it for all the other folds. In this case, we are looking at five outer folds and five inner folds, but there is no reason to use the same number of outer and inner folds, you can use any number you want as long as you keep the folds separate.

This leads to a lot of computation, but it is necessary in order to do things correctly. The problem is that if you use a piece of data to make any decisions about your model (including which parameters to set), you have contaminated it and you can no longer use it to test the generalization ability of your model. This is a subtle point and it may not be immediately obvious. In fact, it is still the case that many users of machine learning get this wrong and overestimate how well their systems are doing, because they do not perform cross-validation correctly!

Fortunately, scikit-learn makes it very easy to do the right thing; it provides classes named `LassoCV`, `RidgeCV`, and `ElasticNetCV`, all of which encapsulate an inner cross-validation loop to optimize for the necessary parameter. The code is almost exactly

like the previous one, except that we do not need to specify any value for alpha:

```
>>> from sklearn.linear_model import ElasticNetCV
>>> met = ElasticNetCV()
>>> kf = KFold(len(target), n_folds=5)
>>> p = np.zeros_like(target)
>>> for train,test in kf:
...    met.fit(data[train],target[train])
...    p[test] = met.predict(data[test])
>>> r2_cv = r2_score(target, p)
>>> print("R2 ElasticNetCV: {:.2}".format(r2_cv))
R2 ElasticNetCV: 0.65
```

This results in a lot of computation, so you may want to get some coffee while you are waiting (depending on how fast your computer is). You might get better performance by taking advantage of multiple processors. This is a built-in feature of scikit-learn, which can be accessed quite trivially by using the `n_jobs` parameter to the `ElasticNetCV` constructor. To use four CPUs, make use of the following code:

```
>>> met = ElasticNetCV(n_jobs=4)
```

Set the `n_jobs` parameter to `-1` to use all the available CPUs:

```
>>> met = ElasticNetCV(n_jobs=-1)
```

You may have wondered why, if ElasticNets have two penalties, the L1 and the L2 penalty, we only need to set a single value for alpha. In fact, the two values are specified by separately specifying alpha and the `l1_ratio` variable (that is spelled *ell-1-underscore-ratio*). Then, α1 and α2 are set as follows (where *ρ* stands for `l1_ratio`):

$$\alpha_1 = \rho\alpha$$

$$\alpha_2 = (1 - \rho)\alpha$$

In an intuitive sense, alpha sets the overall amount of regularization while `l1_ratio` sets the tradeoff between the different types of regularization, L1 and L2.

We can request that the `ElasticNetCV` object tests different values of `l1_ratio`, as is shown in the following code:

```
>>> l1_ratio=[.01, .05, .25, .5, .75, .95, .99]
>>> met = ElasticNetCV(
                l1_ratio=l1_ratio,
```

```
                    n_jobs=-1)
```

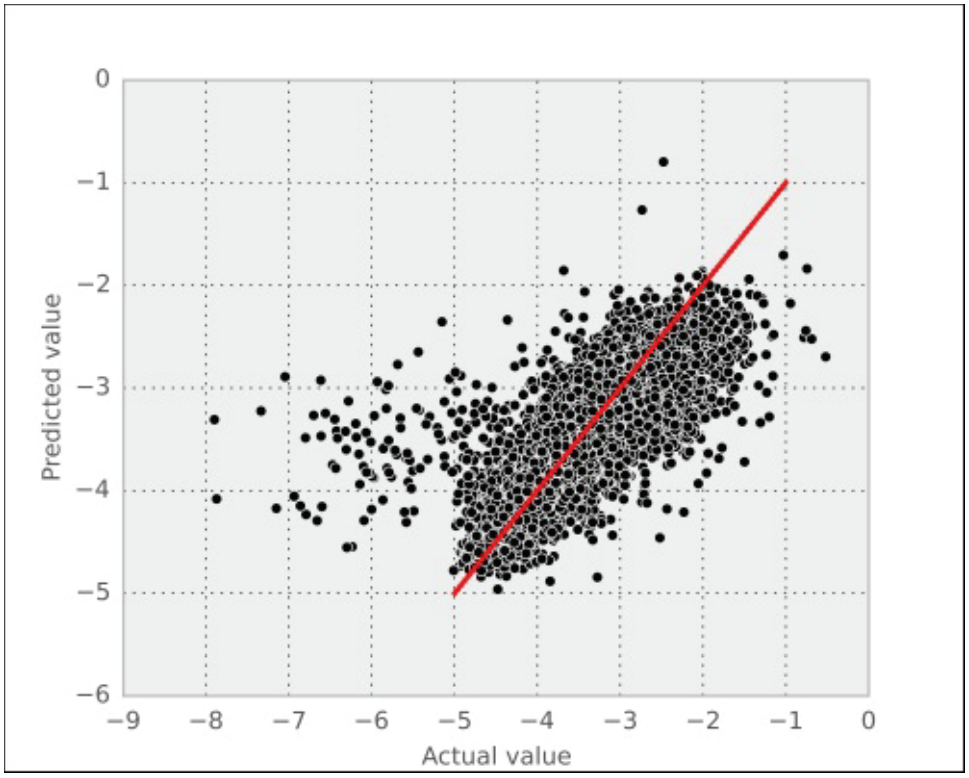
This set of `l1_ratio` values is recommended in the documentation. It will test models that are almost like Ridge (when `l1_ratio` is 0.01 or 0.05) as well as models that are almost like Lasso (when `l1_ratio` is 0.95 or 0.99). Thus, we explore a full range of different options.

Because of its flexibility and the ability to use multiple CPUs, `ElasticNetCV` is an excellent default solution for regression problems when you don't have any particular reason to prefer one type of model over the rest.

Putting all this together, we can now visualize the prediction versus real fit on this large dataset:

```
>>> l1_ratio = [.01, .05, .25, .5, .75, .95, .99]
>>> met = ElasticNetCV(
                l1_ratio=l1_ratio,
                n_jobs=-1)
>>> p = np.zeros_like(target)
>>> for train,test in kf:
...     met.fit(data[train],target[train])
...    p[test] = met.predict(data[test])
>>> plt.scatter(p, y)
>>> # Add diagonal line for reference
>>> # (represents perfect agreement)
>>> plt.plot([p.min(), p.max()], [p.min(), p.max()])
```

This results in the following plot:



We can see that the predictions do not match very well on the bottom end of the value range. This is perhaps because there are so many fewer elements on this end of the target range (which also implies that this affects only a small minority of datapoints).

One last note: the approach of using an inner cross-validation loop to set a parameter is also available in scikit-learn using a grid search. In fact, we already used it in the previous chapter.

Summary

In this chapter, we started with the oldest trick in the book, ordinary least squares regression. Although centuries old, it is still often the best solution for regression. However, we also saw more modern approaches that avoid overfitting and can give us better results especially when we have a large number of features. We used Ridge, Lasso, and ElasticNets; these are the state-of-the-art methods for regression.

We saw, once again, the danger of relying on training error to estimate generalization: it can be an overly optimistic estimate to the point where our model has zero training error, but we know that it is completely useless. When thinking through these issues, we were led into two-level cross-validation, an important point that many in the field still have not completely internalized.

Throughout this chapter, we were able to rely on scikit-learn to support all the operations we wanted to perform, including an easy way to achieve correct cross-validation. ElasticNets with an inner cross-validation loop for parameter optimization (as implemented in scikit-learn by `ElasticNetCV`) should probably become your default method for regression.

One reason to use an alternative is when you are interested in a sparse solution. In this case, a pure Lasso solution is more appropriate as it will set many coefficients to zero. It will also allow you to discover from the data a small number of variables, which are important to the output. Knowing the identity of these may be interesting in and of itself, in addition to having a good regression model.

In the next chapter, we will look at recommendations, another machine learning problem. Our first approach will be to use regression to predict consumer product ratings. We will then see alternative models to generate recommendations.