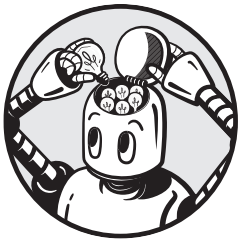
# B1

## SCIKIT-LEARN



This chapter is a bonus chapter for my book *Deep Learning: A Visual Approach*. You can order the book from No Starch Press at *https://nostarch.com/deep-learning-visual-approach/*. The official version of this chapter can be found for free on my GitHub at *https://github.com/blueberrymusic/* (look for the repository "Deep-Learning-A-Visual-Approach"). All of the figures in this chapter, and all of the notebooks with complete, running implementations of the code discussed here, can also be found for free on the book's GitHub repository.

Machine learning is a practical field. Although this book is focused on concepts, there's nothing quite like putting something into practice to transform ideas into understanding.

Implementing an idea makes us face decisions that we might have otherwise brushed off as easy or unimportant, and can reveal misunderstandings and holes in our knowledge. In a field like machine learning, where so much is still an art, and we make decisions based on intuition and

what we've learned from previous mistakes and successes, experience is invaluable.

In short, writing our own code is a challenging, instructive, and vastly illuminating activity. Unfortunately, it's also time-consuming, and there's a lot of work on infrastructure that isn't relevant to machine learning.

If you're of a mind to write your own code, I strongly encourage you to do it! It's well worth the effort. There are many guides online and offline to help you along that path [Müller-Guido16][Raschka15][VanderPlas16].

On the other hand, if you'd like to use existing library code, that's fine, too. Life is short, and we need to put our time and energies where we each feel they're best directed.

In this chapter, we'll take the approach of learning use an existing library. Once you're familiar with what different techniques do, you can always write your own implementations if you feel the itch.

Our library of choice is a popular Python-based toolkit for machine learning called *scikit-learn* (pronounced sy′-kit-lern). We're using scikit-learn because it's widely used, well supported and documented, stable, open-source, and fast.

We'll be using scikit-learn version 0.24, released in December 2020 [scikit-learn20]. Versions released after that are likely to be compatible with the code we'll be using. Downloading and installing the library is usually easy on most systems. See the download instructions on the library's home page for detailed instructions.

A pleasant way to use Python is to work interactively with a Python interpreter. The free Jupyter system provides a simple and flexible interface to interpreted Python that runs inside of any major browser [Jupyter17]. Jupyter notebooks containing running versions of all the programs in this chapter are available for free download on GitHub at *https://github.com/blueberrymusic*. Open the repo named *Deep-Learning-A-Visual-Approach*, then *Notebooks*, then *Bonus01-Scikit-Learn*.

## Introduction

The Python library *scikit-learn* is a free library of common machine-learning algorithms.

To use it well requires some familiarity with the Python language and the NumPy library. NumPy (pronounced num′-pie) is a Python library focused on manipulating and calculating with numerical data. This nesting of dependencies is shown in Figure B1-1.

There are several other scientific "kits" in Python, each called "scikit-something." For example, *scikit-image* is a popular system for image processing. These kits are free and open-source projects produced and maintained by serious developers who largely pursue these efforts on their free time as a public service. Another widely-used library is named *SciPy* (pronounced sie′-pie), and it provides a wealth of routines for scientific and engineering work.

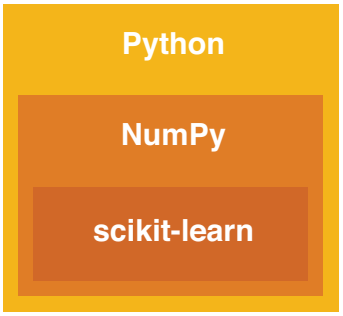



*Figure B1-1: The scikit-learn library uses routines from the numerical calculation library NumPy, which in turn depends upon Python.*

Getting deep into Python, NumPy, or scikit-learn, is not an afternoon project. Each of these entities is big, and has a lot to offer. On the other hand, a weekend of work on any of them can provide a solid start.

If you're new to any of these topics, the References section offers some resources for getting started. If you ever see a term or routine that seems unfamiliar, fountains of documentation on all of these topics are just an internet search away. In this chapter, our focus is on scikit-learn, so we'll assume that you are familiar enough with Python and NumPy to at least follow along.

Even though this chapter is focused on scikit-learn, we won't go deep into the library. Like any substantial piece of software, scikit-learn has its own conventions, defaults, assumptions, limitations, shortcuts, best practices, common pitfalls, easily-committed mistakes, and so on. And each routine within the library has its own specific details that are important to know in order to get the best results. To give these topics proper attention would require a book of its own, and indeed such books exist [Müller-Guido16].

Our goal in this chapter is to give an overview of the library and how it's used. For details on any routine or technique, consult the online documentation, reference books, or even some of the helpful web pages and blog posts online.

## Python Conventions and Libraries

The code examples in this chapter (and the accompanying notebooks) are written for Python 3.7.6, but any version of Python 3 should work.

Scikit-learn is *big*. The library offers around 400 routines, which range from small utilities to large machine-learning algorithms bundled up into a single library call. These routines are extensively documented on the library's website. As of this writing, the current release (version 0.24) groups the library's routines into a whopping 35 categories.

Since we're taking a very general overview here, we'll invent our own smaller set of categories to organize our discussion: *Estimators*, *Clustering*, *Transformations*, *Data Refinement*, *Ensembles*, *Automation*, *Datasets*, and *Utilities*.

Many elements of scikit-learn are arranged like interlocking pieces, and some are designed to be used in particular combinations. We'll see some examples of those combinations later when we demonstrate specific algorithms.

Scikit-learn is not directed to neural networks, which are the hallmark of deep learning. Rather, these tools are intended for both standalone data science, and as utilities for neural networks, helping us explore, understand, and modify our data. Many deep learning projects begin with a thorough exploration of the data carried out with scikit-learn.

This chapter inevitably has code listings, but we have tried to keep these small and focused. Full code listings are great resources for learning the details of a system, and how to manage issues like structuring data and calling the right operations in the proper sequence. On the other hand, long blocks of code are boring to read. And real projects usually spend considerable time and effort on little details that are particular to that specific program, and don't aid our understanding of the general principles.

So while we will build real systems to do real work, we'll keep these project-specific steps to a minimum. We'll typically build up our projects one step at a time. So we'll show and discuss each step, but then consider it part of the program and not repeat that code every time as the program grows. In a few instances, we will gather up a bunch of code into one big listing, but usually we leave the complete summaries to the Jupyter notebooks that accompany the chapter, along with details like reading and writing data, and creating plots and other graphics.

To use scikit-learn, we must `import` it first, since Python doesn't load it automatically. When we import scikit-learn, it goes by the shorthand name `sklearn`. This name is just the top-level name for the library. The routines themselves are organized into a set of *modules*, which we also need to import.

There are two popular ways to import a module. The first is to import the whole module. Then we can name anything in that module in our code by prefixing it with the module's name, followed by a period. Listing B1-1 shows this approach, where we create an object named `Ridge`, which comes from the scikit-learn module named `linear_model`.

```
from sklearn import linear_model

ridge_estimator = linear_model.Ridge()
```

*Listing B1-1: We can import the whole module `linear_model` and then use it as a prefix to name the `Ridge` object. Here we make a `Ridge` object with no arguments and save it in the variable `ridge_estimator`.*

The other approach is to import only what we need from the module, and then we can use that object without referring to its module in the code, as in Listing B1-2.

```
from sklearn.linear_model import Ridge

ridge_estimator = Ridge()
```

*Listing B1-2: We can import just the `Ridge` object from `linear_model`, and then we don't need to name the module when we use the object. The result of this code is identical to the result of Listing B1-1.*

Generally speaking, it's usually easier during development to import the whole module, since then we can try out different objects from that module without constantly changing the `import` statement. When we're all done, we often go back and clean things up to import only what we're actually using, since that's considered a bit cleaner. In practice, both importing the whole module and importing just specific pieces are common, and often even mixed in the same file.

We will be using NumPy a lot, so we'll frequently import it as well. Conventionally, when we import NumPy we use the abbreviation `np`. This is just a widely used convention, and not a rule.

We also frequently include the `math` module. The convention is not to rename this library, so we refer to it in our code as `math`.

Another common library is the `matplotlib` graphics library which is the standard way to produce graphs and other plots. This too has a conventional name. The module we usually use from this library is called `pyplot`, and the tradition is to call this `plt`, or "plot" without the o. It's weird and arbitrary, but `plt` is used almost universally.

Finally, the Seaborn library offers some additional graphics routines, and also modifies the visuals produced by `matplotlib` to look more attractive [Waskom17]. When we import Seaborn, it is conventional to call it `sns` (this is a rather elaborate inside joke. According to the author of Seaborn, sns refers to the initials of the fictional Communications Director Samuel Norman Seaborn on the TV show *The West Wing* [Github14]). In this chapter our only use of Seaborn will be to import it, which will cause it to replace `matplotlib`'s default aesthetics with its own, causing our graphics to look much nicer. We do this by calling `sns.set()`, and we usually place that call on the same line as the `import` statement, separated by a semicolon.

Listing B1-3 shows typical `import` statements for these other libraries.

```
import sklearn
import numpy as np
import math
import matplotlib.pyplot as plt
import seaborn as sns ; sns.set()
```

*Listing B1-3: This set of `import` statements will be our starting point in almost every project.*

Seaborn lets us scale up the text in all of our graphs and plots at once. For instance, to scale up the font to about 14 point we could replace the `set()`call with `sns.set(font_scale=1.3)`. We'll leave the text size as-is in this chapter.

To find the right module to import for each scikit-learn routine we want to use, we can refer to the online API Reference. That reference also contains a complete breakdown of everything in scikit-learn, though often in very terse language. The API Reference is great when we treat it as just that, a reference when we want to remember what something's called or how to use it. But because of its brevity, it's less useful for explanations. That information is best found from one of the books or sites listed in the References section.

The people who manage scikit-learn have very high standards for including new routines, so additions are rare. The library is updated when there are important bug fixes and other improvements, and it's very occasionally re-organized. As a result, some routines become marked as "deprecated," which means that they are planned for removal, but are left in for a while so people can update their code. If you call a deprecated routine, you get a warning that often suggests how to update your code to be compliant with the new way of doing things. Deprecated routines are often later subsumed into more general library features, or simply re-organized into a different module.

Many of the routines in scikit-learn are made available to us through *objects*, in the object-oriented sense. For example, if we want to analyze some data in a particular way, we usually create an instance of an object designed for that kind of analysis. Then we get things done by calling that object's methods, instructing the object to do things like analyze a set of data, calculate statistics on the data, process it in some way, or predict values for new data.

Thanks to Python's built-in garbage collection, we don't have to worry about recycling or disposing of our objects when we're through with them. Python will automatically reclaim memory for us when it's safe to do so.

Using objects in this way works very well in practice, since it lets us keep our focus on what we want to do, and less on the mechanics.

A good way to learn a huge library like scikit-learn is to casually look up each routine or object the first time we see it used, often while typing in code from a reference, or studying someone else's program. A quick look at the options and defaults can help flesh out our sense of what the routine is able to do. Later, while working on another project, something may ring a bell, or otherwise draw us back to this routine or object.

Then we can look at the documentation again and read it more closely. In this way we can gradually learn a bit more about the breadth and depth of each feature as we find ourselves using it, which is much easier than trying to memorize everything about every object and routine the first time we encounter it.

Almost all objects and routines in `sklearn` accept multiple arguments. Many of these are optional and take on carefully-chosen default values if we don't refer to them. For simplicity, in this chapter we'll usually pass only the mandatory arguments, and leave all the optional arguments at their defaults.

## Estimators

*See the Jupyter notebook* Bonus01-Scikit-Learn-1-Estimators.ipynb.

We use scikit-learn's *estimators* to carry out supervised learning. We create an estimator object, and then give it our labeled training data to learn from. We can later give our object a new sample it hasn't seen before, and it will do its best to predict the correct value.

All estimators have a core set of common routines and usage patterns.

As we mentioned earlier, each estimator is an individual *object* in the object-oriented programming sense. The object knows how to accept data, learn from it, and then describe new data it hasn't seen before, maintaining everything it needs to remember in its own instance variables. So the general process is to first *create* the object, and then send it *messages* (that is, call its built-in routines, or methods) causing it to take actions. Some of those actions only modify the internal state of the estimator object, while others compute results that are returned to us.

Figure B1-2 shows what we'll be aiming for in this section. We'll start with a bunch of points, and use a scikit-learn routine to find the best straight line through them.

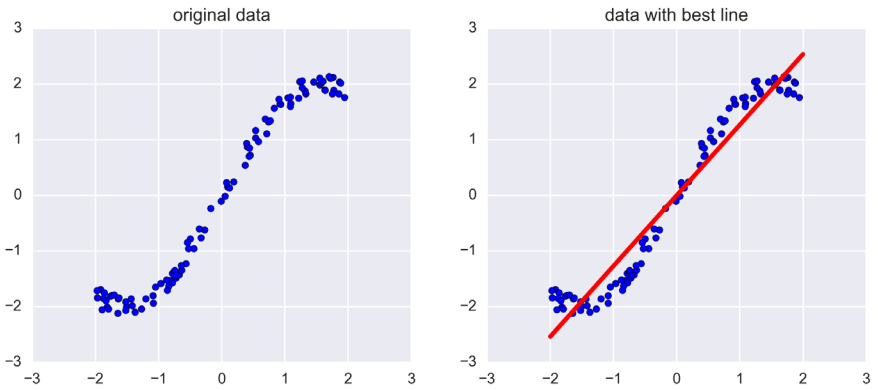



*Figure B1-2: straight line fit by a scikit-learn estimator*

### Creation

The first step is to *create*, or *instantiate*, the *estimator object*, often just called the *estimator*. As an example, let's instantiate an estimator called `Ridge` from the `linear_model` module. The `Ridge` object is a versatile estimator for doing this sort of regression task, and it includes built-in regularization. We just name the object (prefixing it with its module name), along with any arguments we want or need to provide. Since we're sticking with all the defaults, we won't pass any arguments. Listing B1-4 shows the code.

```
from sklearn import linear_model

ridge_estimator = linear_model.Ridge()
```

*Listing B1-4: Creating a `Ridge` estimator object*

After this assignment, the variable `ridge_estimator` refers to an object that implements the Ridge regression algorithm. When we use the simplest version of the estimator with 2D data, as we're doing here, this finds the best straight line that goes through the data.

All estimators have two routines named `fit()` and `predict()`. We use these respectively to teach our estimator, and to have it evaluate new data for us.

## Learning with fit()

The first routine we'll look at is called `fit()`, and it is provided by every estimator. It takes two mandatory arguments, containing the samples that the estimator will learn from and the values (or targets) associated with them. The samples are usually arranged as a big numerical NumPy grid (called an *array*, even when there are many dimensions), where each row contains one sample.

Let's look at one example of generating data in the right format, so we can get a feeling for the process. We'll create the data shown on the left of Figure B1-2.

In Listing B1-5, we start by seeding NumPy's pseudo-random number generator, so we'll get back the same values every time. This lets us re-run the code and still generate the same data. Then we set the number of points we want to make in `num_pts`. We'll make our data based on a piece of a sine wave (a nice curve that's built into the `math` library), but we'll add a little noise to each value. Then we run a loop, appending point X and Y values to the arrays `x_vals` and `y_vals`. The array `x_vals` contains our samples, and `y_vals` contains the target value for each corresponding sample.

```
np.random.seed(42)
num_pts = 100
noise_range = 0.2
x_vals = []
y_vals = []
(x_left, x_right) = (-2, 2)
for i in range(num_pts):
    x = np.random.uniform(x_left, x_right)
    y = np.random.uniform(-noise_range, noise_range) + (2*math.sin(x))
    x_vals.append(x)
    y_vals.append(y)
```

*Listing B1-5: Making data for training*

The `x_vals` variable holds a list. But the `Ridge` estimator (like many others) wants to see its input data in the form of a 2D grid, where each entry is a list of features on its own row. Since we have only one feature, we can use NumPy's `reshape()` routine to turn `x_vals` into a column. Listing B1-6 shows this step.

```
x_column = np.reshape(x_vals, [len(x_vals), 1])
```

*Listing B1-6: Reshaping our `x_vals` data into a column*

Now that we have our data in the form expected by our `Ridge` object, we hand the samples over and ask it to find a straight line through the points. The first argument gives the samples and the second argument gives the labels associated with those samples. In this case, the first argument contains the X coordinate of each point, and the second contains the Y coordinate. We could turn the second argument into a column as well if we wanted, but `fit()` is happy to accept this argument as a 1D list, as shown in Listing B1-7.

```
ridge_estimator.fit(x_column, y_vals)
```

*Listing B1-7: We train an estimator by calling its `fit()` method with the training data as an argument.*

The `fit()` routine analyzes the input data and saves its results internally in the `Ridge` object. We can think of `fit()` as meaning, "analyze this data, and save the results so that you can answer future questions on this and related data."

Conceptually, we can think of `fit()` like a tailor who starts with a bolt of fabric, and then upon meeting and measuring a customer, proceeds to cut, sew, and *fit* a custom set of clothes for that person. In the same way, `fit()` calculates data that is customized to this particular input data, and saves that inside the estimator. If we call this object's `fit()` again with different set of training samples, it will start over and build a new set of internal data to fit that new input. This all happens inside of the object, so the `fit()` routine doesn't return anything new (for convenience, it does return a reference to the object it was called on. So in our case, it just returns the same `ridge_estimator` we just used to call `fit()` itself).

This means that we can make multiple estimator objects and keep them all around at the same time. We might train them all on the same data and then compare their results, or we might make many instances of the same estimator and train it with different data sets. The key thing is that each object is independent of the others, containing the parameters needed to answer questions about the data it was given.

Our program so far puts together everything from Listings B1-4 to B1-7.

## Predicting with predict()

Once we've trained our estimator, we can ask it to evaluate new samples with `predict()`. This takes at least one argument, containing the new samples that we want the estimator to assign values to. The routine returns the information that describes the new data. Usually, this is a NumPy array that contains either one number or one class per sample.

Since our estimator has created a straight line to the data, we can get back that line by asking for the Y value for any two different values of X. Let's ask it for the Y values at the X values corresponding to the leftmost and rightmost values in our input data, which we can find by asking NumPy for the minimum and maximum values in the column array of X values. We'll give the estimator these minimum and maximum X values, and ask it

for the corresponding Y values. We have to turn the X values into 2D arrays because that's what `predict()` expects, so we just nest them both in two pairs of brackets, making a list that holds a list that holds our one item. The values that come back are the endpoints of our line. Listing B1-8 shows the calls.

```
x_left = np.min(x_column)
x_right = np.max(x_column)
y_left = ridge_estimator.predict([[x_left]])
y_right = ridge_estimator.predict([[x_right]]])
```

*Listing B1-8: Getting the Y values of the straight line at two values of X*

Let's put this all together. We'll import the stuff we need, make the data, make the estimator, fit the data, and get the left and right Y values of the line through the data. We'll just combine Listings B1-4 through B1-8. To keep things short, we'll replace the data-generating step in Listing B1-5 with a comment.

Listing B1-9 shows the code.

```
import numpy as np
import math
from sklearn.linear_model import Ridge

# Make the data. Save the samples in column form
# as x_column, and the target values as y_vals

# Make our estimator
ridge_estimator = Ridge()

# Call fit() to find the best straight-line fit to our data.
ridge_estimator.fit(x_column, y_vals)

# Find the Y values at the left and right ends of the data
x_left = np.min(x_column)
x_right = np.max(x_column)
y_left = ridge_estimator.predict([[x_left]])
y_right = ridge_estimator.predict([[x_right]])
```

*Listing B1-9: Using the `Ridge` object to fit some 2D data*

Figure B1-3 repeats Figure B1-2, showing the result of Listing B1-9, after we add a few lines of code to create the plots. We just draw all the data points, and then draw a red line from the point `(x_left, y_left)` to the point `(x_right, y_right)`.

We leave the plotting details to the code in the notebooks. Information about using `matplotlib` and `seaborn` can be found in each library's own documentation online, or in online or print references [VanderPlas16].

Now that we have the endpoints for this line, we know the equation of the line. We can find the estimated Y value for any X by just plugging that X value into the line's equation (or we could be lazy and ask our estimator to do that for us).

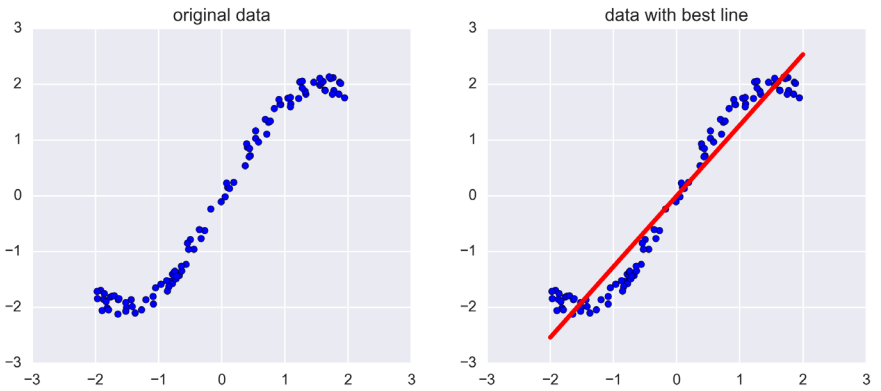



*Figure B1-3: A straight line fit by `Ridge` to 2D data, using the code of Listing B1-9*

## Using decision_function() and predict_proba()

We just saw how to use an estimator to solve a regression problem.

The procedure is very similar for classification problems, which use classification estimators.

For classifiers, the `predict()` function gives us a single category as a result. But sometimes we want to know how "close" a sample came to being categorized into each of the other classes. For example, a classifier might decide that a given photo has a 49% chance of being a tiger, but a 48% chance of being a leopard (with the other 3% being other types of cats). The `predict()` function would tell us the photo is a tiger, but we might want to know that the leopard was a very close second.

For these situations where we care about the probabilities of our input with respect to all of the possible classes, many categorizers offer some additional options.

The routine `decision_function()` takes a set of samples as input, and returns a "confidence" score for every class and every input, where larger values represent more confidence. Note that it's possible for many categories to have large scores.

By contrast, the routine `predict_proba()` returns a *probability* for each class. Unlike the output of `decision_function()`, all of the probabilities for each sample always add up to 1. This often makes the output of `predict_proba()` easier to understand at a glance than that from `decision_function()`. It also means we can use the output of `predict_proba()` as a probability distribution function, which is sometimes convenient if we're doing additional analysis of the results.

Though all classifiers provide `fit()`, not all of them also offer either or both of these other routines. As always, the API documentation for each classifier tells us what it supports.

## Clustering

*See the Jupyter notebook* Bonus01-Scikit-Learn-2-Clusters.ipynb.

Clustering is an unsupervised learning technique, where we provide the algorithm with a bunch of samples, and it does its best to group similar samples together.

Our input data will be a big collection of 2D points with no other information.

There are lots of clustering options in scikit-learn. Let's use the *k-means* algorithm. The value of *k* tells the algorithm how many clusters to build (though the routine calls that argument `n_clusters`, for "number of clusters," rather than the shorter and more cryptic, though traditional, single letter *k*).

We start by creating a `KMeans` object. To get access to that, we need to import it from its module, `sklearn.cluster`. When we make the object, we can tell it how many clusters we're going to want it to build once we give it data. This argument, named `n_clusters`, defaults to 8 if we don't give it a value of our own. Listing B1-10 shows these steps, including passing a value for the number of clusters.

```
from sklearn.cluster import KMeans

num_clusters = 5
kMeans = KMeans(n_clusters=num_clusters)
```

*Listing B1-10: Importing `KMeans` and then creating an instance*

Clustering algorithms are often used with data that has many dimensions (or features), perhaps dozens or hundreds. But when we create our clustering object we don't have to tell it how many features we'll be using, because it will extract that information from the data itself when we provide that later via a call to `fit()`.

As usual, let's use 2 dimensions (that is, 2 features per sample) so we can draw pictures of our data and the results. To create our dataset, we'll make 7 random Gaussian blobs, and pick points at random from each one. Figure B1-4 shows the result. We also show the data color-coded by the blob that generated each sample, but that's strictly for our human eyes. The computer only sees a list of X and Y values.

Remember, we're just giving it the black dots from Figure B1-4, so the algorithm knows nothing about these points except for their location. As before, we'll shape our data as a NumPy array that is a big column, where each row has two dimensions, or features: the X and Y values of that row's point. We'll call that data `XY_points`. To build the clusters from this data, we only have to hand it to our object's `fit()` routine. Listing B1-11 shows the code.

```
kMeans.fit(XY_points)
```

*Listing B1-11: We give our data to `kMeans` and it will work out the number of clusters we requested when we made the object.*


*Figure B1-4: The data we'll use for clustering. Left: The data the algorithm will get. Right: A cheat sheet that shows us which Gaussian blob was used to generate each sample.*

As soon as `fit()` returns, the clustering is done. To visualize how it broke up our starting data, we'll run through the same data again, and ask for the predicted cluster for each sample. Conveniently, we get this information using the object's `predict()` method, as shown in Listing B1-12.

```
predictions = kMeans.predict(XY_points)
```

*Listing B1-12: We can find the predicted cluster for our original data just by handing it to `predict()`.*

The variable `predictions` is a NumPy array that's shaped as a column. That is, it has one row for every point in the input, and each row has one value: an integer telling us which cluster the routine has assigned to the corresponding point in our `XY_points` input. For example, the fifth row of `XY_points` holds the X and Y values of a point, and the fifth row of `predictions` holds an integer telling us which cluster that point was assigned to.

Let's run this process for a bunch of different numbers of clusters. Leaving out the plotting code, it looks like Listing B1-13.

```
for num_clusters in range(2, 10):
    kMeans = KMeans(n_clusters=num_clusters)
    kMeans.fit(XY_points)
    predictions = kMeans.predict(XY_points)
    # plot the XY_points and predictions
```

*Listing B1-13: Trying out a range of different numbers of clusters*

The results are shown in Figure B1-5.

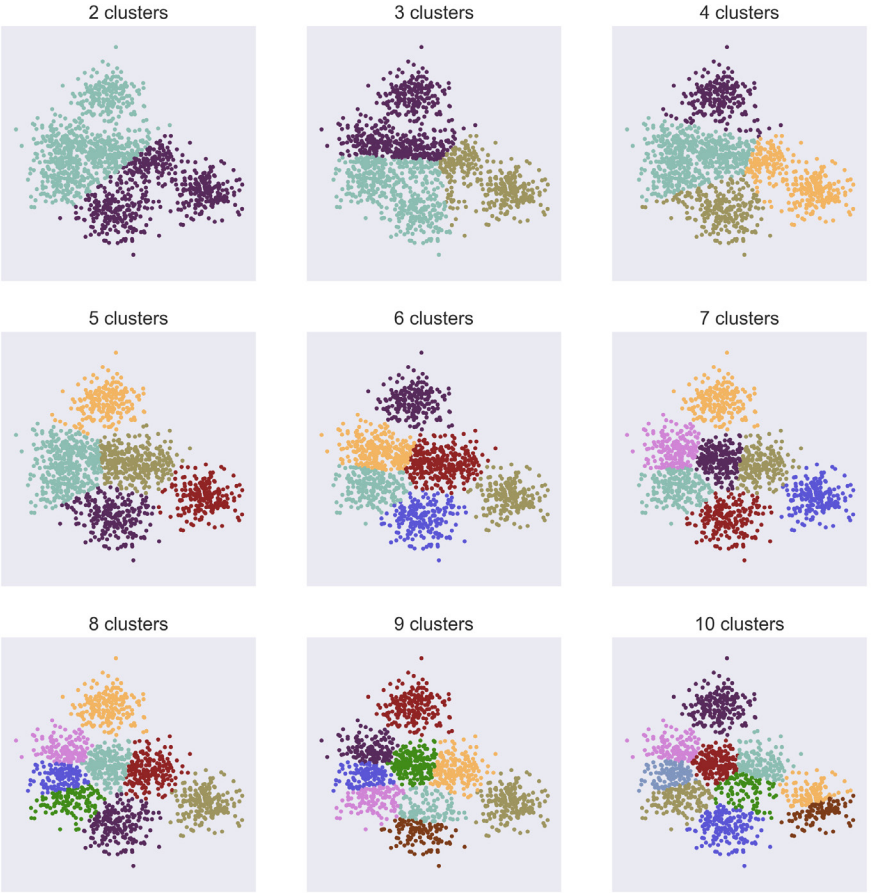



*Figure B1-5: The KMeans algorithm clustering our starting data from Figure B1-4 into different numbers of clusters. Each cluster is shown in a different color.*

Visually, things start to look reasonable starting at about 5 clusters. Because we used overlapping Gaussians to create our dataset, the groups are not all cleanly distinguished. But starting around 5 clusters we can see that the topmost and rightmost chunks get identified well, with the main mass getting subdivided into smaller regions. Starting at around 9 clusters those outer chunks seem be getting split up themselves. So somewhere between 5 and 8 seems like a good choice, which is a satisfying result knowing that we started with 7 overlapping Gaussians.

Scikit-learn offers multiple clustering routines because they approach the problem in different ways, and produce different kinds of clusters.

Clustering algorithms are a great way to get a feeling for the spatial distribution of our data. A downside of clustering is that we have to pick the number of clusters. There are algorithms that try to pick the best number of clusters for us [Kassambara17], but often we still have to look at the results and decide if they make sense.

## Transformations

*See the Jupyter notebook* Bonus01-Scikit-Learn-3-Transformations.ipynb.

Much of the time we'll want to *transform*, or modify, our data before we use it. Often this is because we want to make sure that the data fits the expectations of the algorithms we'll give it to. For instance, many techniques work best if they receive data in which every feature is centered at 0, and scaled to the range [–1, 1].

Scikit-learn offers a wide range of objects, called *transformers*, that carry out many different kinds of data transformations. Each routine accepts a NumPy array holding sample data, and returns a transformed array. Note that this name unfortunately conflicts with the name of the popular *transformer* architecture, which we discussed in Chapter 20. The name as used here is actually a bit more appropriate, as these routines carry out operations that are traditionally referred to as transformations. The transformer models in Chapter 20 are an entirely different kind of thing, and we won't be referring to them again in this chapter.

We usually choose which transformer to apply based on the estimator we will ultimately give our data to. Each estimator that wants its data in a specific form gives that information in its documentation, but it's up to us to explicitly carry out any transformations they suggest or require.

We learn our transformation from the training data, but it's essential that we then apply the same transformation to all further data. To help us carry this out, each transformer offers distinct routines for creating the transformation object, analyzing data to find the parameters for its transformation, and applying that transformation.

We create the object in the same way that we created other objects above, by calling its creation routine with any arguments we want to pass.

To find the parameters for a specific transformation, we call the `fit()` method, just as we do for an estimator. We give `fit()` our data, and the object analyzes it to determine the parameters for the transformation performed by that object. Then we actually transform data with the `transform()` method, which takes a set of data, and returns the transformed version. Then any time we have data for the trained model, whether it's the original training data, or validation data, or even new data arriving after deployment, we'll first run it through this object's `transform()` method.

Notice how nicely this encapsulates the necessity of retaining the transformation. Our object learns what it needs to do just one time, up front, when we call `fit()`, and then it will apply that same transformation to any data we hand it from then on.

Figure B1-6 illustrates the idea.

After creating a scaling transformer, we give it the training data by calling `fit()`. The scaler analyzes the data and determines the scaling transformation. Then we transform the training data with `transform()` and train our estimator with it. From then on, all new data we want to use also goes through the scaling object's `transform()` routine.

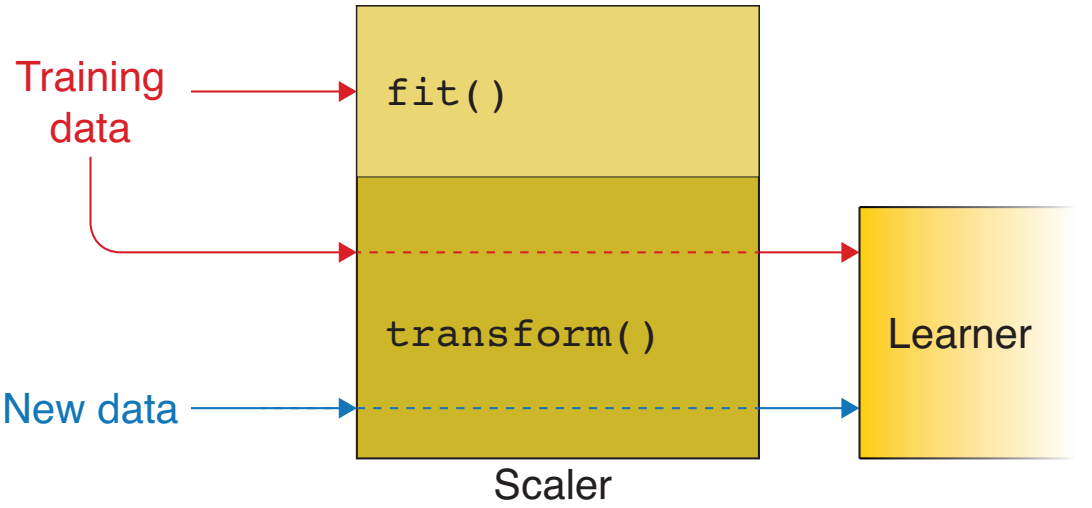



*Figure B1-6: Using a scaling transformer to modify our data before training*

Let's see this in action. We'll use a transformer with a very obvious visual effect: each feature is scaled to the range [0, 1]. As usual, we'll use 2D data, so there are two features to be scaled: X and y.

We'll use a scikit-learn transformer called the `MinMaxScaler`, which was designed for just this kind of task. We can give it data with any number of features, and by default each feature will be independently transformed to the range [0,1].

Let's start with the data in Figure B1-7. The X values are in the range of about [–1.5, 2.5] and the Y values are in the range of about [–1, 1.5].

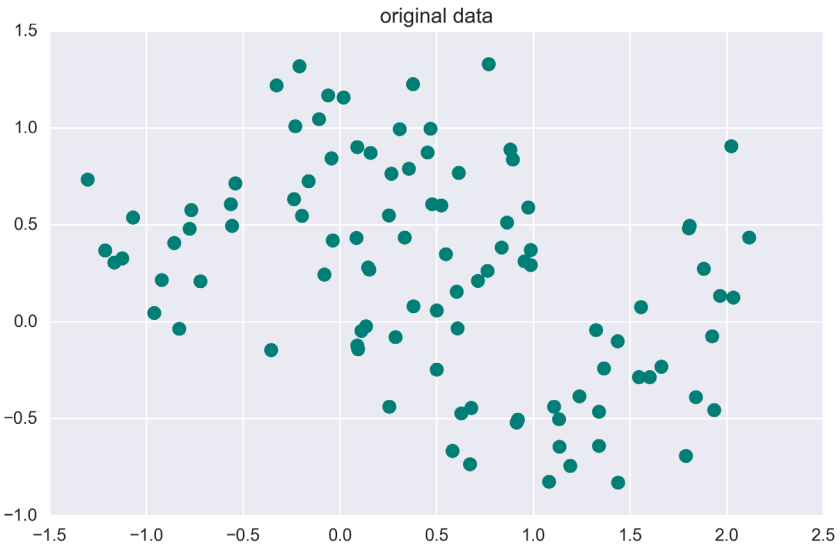



*Figure B1-7: The starting two-dimensional (or two-feature) data for scaling.*

To use the scaler, as usual we have to first import the proper module from `sklearn`. In this case, the documentation tells us that it's `sklearn.preprocessing`. Our first step is to create the scaler, and save it in a variable, in Listing B1-14.

```
from sklearn.preprocessing import MinMaxScaler

mm_scaler = MinMaxScaler()
```

*Listing B1-14: Creating a `MinMaxScaler` object to scale all of our features at once.*

Now that we have our object, let's have it analyze our training data and work out the transformation by calling `fit()` with our training data. As before, we'll arrange our data in tabular form, where each row contains all the features for one sample. So our data will have two entries per row (one each for the point's X and Y values). Listing B1-15 shows the call.

```
mm_scaler.fit(training_samples)
```

*Listing B1-15: When we call `fit()`, our `MinMaxScaler` will work out the transformation to scale each feature to [0,1].*

Now we're ready to transform our data. We just call `transform()` on our set of samples, and save the transformed values. We can then hand these to our estimator for training in Listing B1-16.

```
transformed_training_samples = mm_scaler.transform(training_samples)
```

*Listing B1-16: We call `transform()` on any set of data to scale it using the transformation determined when we called `fit()`.*

The result of transforming our training data is shown in Figure B1-8.

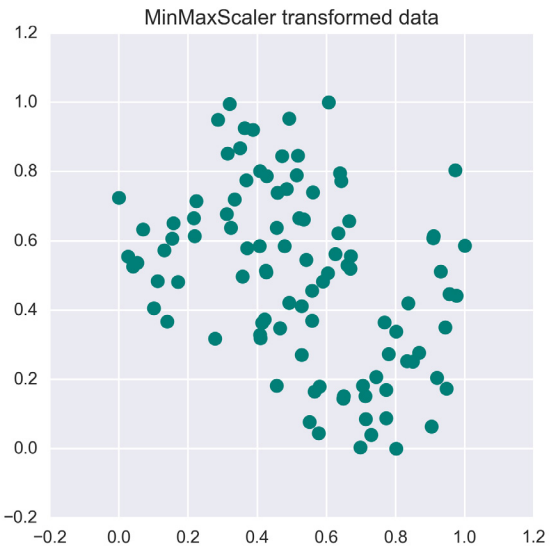



*Figure B1-8: Our data from Figure B1-7 transformed by our `MinMaxScaler` so that each feature is independently scaled to [0,1].*

We can see that our data now spans the range [0,1] in both X and Y.

Let's suppose that we now we want to evaluate the quality of our estimator using some test, or validation, data. We know that before we give this to our estimator, we have to transform it first in the same way that we transformed the training data. We'll do that in Listing B1-17.

```
transformed_test_samples = mm_scaler.transform(test_samples)
```

*Listing B1-17: We call `transform()` on our new data to transform it before using it.*

Figure B1-9 shows some test data, and its transformed result.

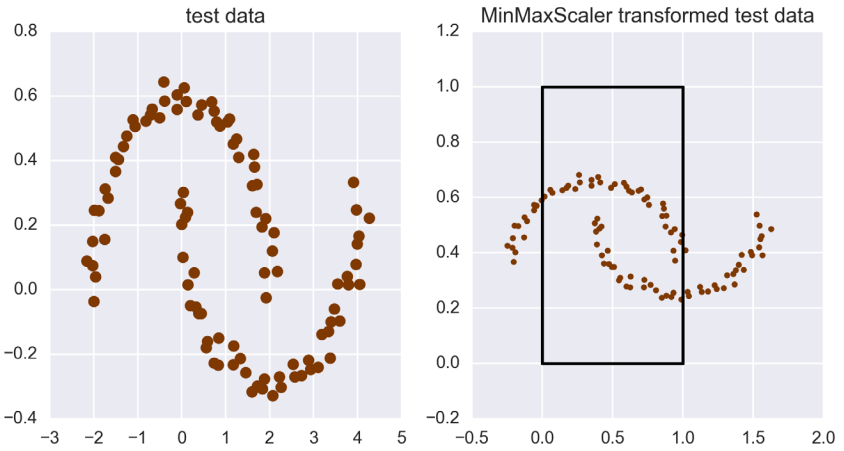



*Figure B1-9: Transforming new data. Left: The test data. Right: The test data after transformation. The black rectangle shows the box from (0,0) to (1,1).*

Notice that the transformed data is *not* compressed and scaled to the range [0,1] in X or in Y, as we can see from the box from (0,0) to (1,1) in the figure. This is exactly what we'd expect from new data that has larger values than were found in the training data. In this case, the new test data x values are in the range of about [–3, 4], which is much larger than the training data's range of [–1.5, 2.5]. So the transformed X values will fall outside the [0,1] range. The Y range of the test data is about [–0.4, 0.6], which is much less than the training data's range of about [–1, 1.5]. Thus the y values only use up a small piece of the range [0,1] on the Y axis.

So although the transformation shifted and compressed the X values, there's still data falling outside of the X range [0,1]. In the same way, the system shifted and compressed the Y range, but it left a lot of empty space in the Y range [0,1]. Although our estimator works best for data in the range [–1, 1], it will still work well for data that's "close" to that range. The exact meaning of "close" will vary from one estimator to another, so it pays to make sure that the data we give the system is at least vaguely the same as the data it trained on, or the inputs can be much larger or smaller than expected after they're transformed.

### Inverse Transformations

We can use `predict()` to get an estimator's output for some data. Remember that these results are based on the data which we fed into the estimator, which we'd transformed first. Whether we're looking at training data, test data, or deployment data, it all went through the transformation.

In the book we discussed a regression problem that asked us to predict the number of cars on a street given the temperature the previous midnight. We transformed the input data, so the value predicted by the estimator was also transformed. We saw that we had to *undo*, or *invert*, the transform so that it represented a number of cars, rather than a value from 0 to 1.

That process is typical. If we want to compare the results that come out of an estimator with our original, un-transformed data, we have to somehow *undo* the transformation. In our scenario above with the `MinMaxScaler`, we want to un-stretch the samples in the exact opposite ways that we originally stretched them.

Every scikit-learn transformer provides a method called `inverse_transform()` for precisely this purpose, where "inverse" means "opposite." So this routine applies the opposite of whatever `transform()` did. Figure B1-10 shows the idea, where we're showing a generic "learner" rather than a specific estimator.

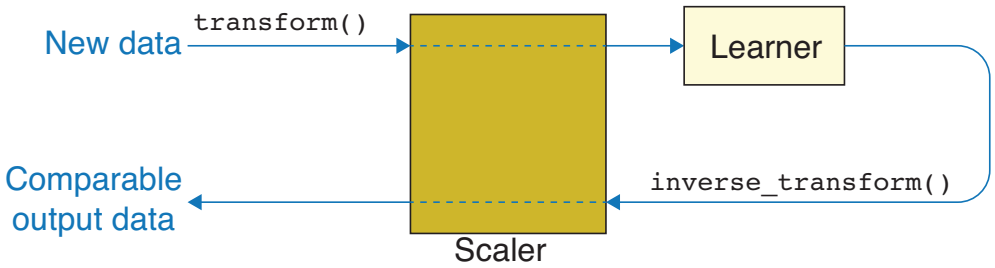



*Figure B1-10: To convert data back to its original form, we call the transformer's `inverse_transform()` method on that output data.*

If we took the learner out of this loop, the values in the lower-left would be identical to those in the upper-left.

For example, we can feed the samples on the right side of Figure B1-9 to `inverse_transform()`, and we'd get back the samples on the left. Listing B1-18 shows the code.

```
recovered_test_samples = \
    mm_scaler.inverse_transform(transformed_test_samples)
```

*Listing B1-18: We call `inverse_transform()` to undo the transformation applied by this object.*

Let's see this in action. On the left of Figure B1-11 we see our starting data, which is similar to data we used earlier. The data has an X range of about [0,6] and a Y range of about [–2, 2]. After setting up a new

`MinMaxScaler` and calling `fit()` with this data, and then running it through `transform()`, we get the version on the right, where both ranges are now [0,1].

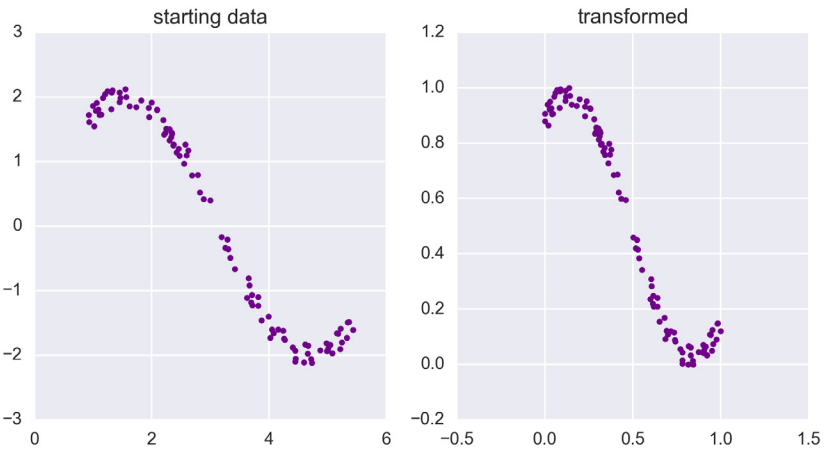



*Figure B1-11: Left: Some test data. Right: After setting up a new `MinMaxScaler` object with this data, and then transforming it, both X and Y are in the range [0,1].*

Let's now fit a line to our transformed data. We'll use the `Ridge` estimator we saw earlier.

The leftmost image of Figure B1-12 shows the line that our `Ridge` estimator fit to the transformed data, along with the transformed data itself. In the middle image we show that line, along with the *original*, non-transformed input data. It's a terrible match! That's because the line was fit to the transformed data, not the original.

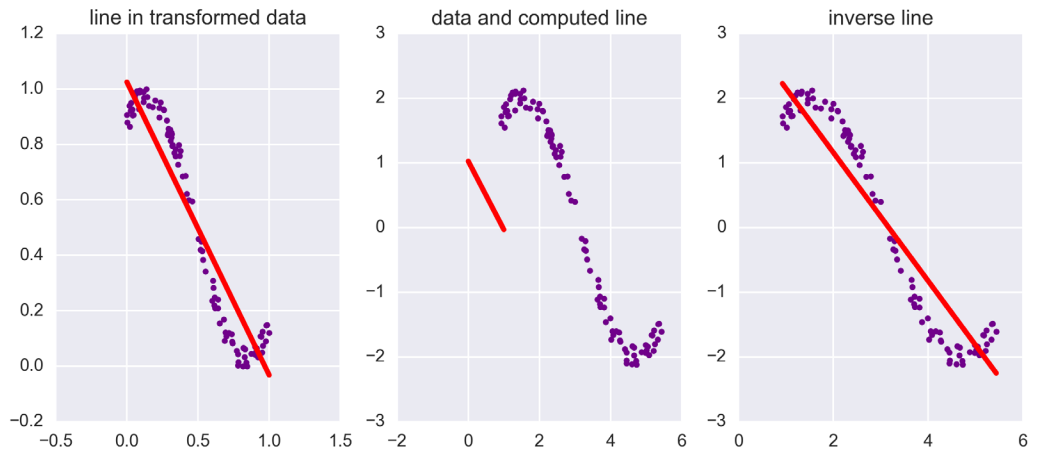



*Figure B1-12: Fitting a line. Left: Our transformed data. Middle: Our original data and the line we found from the transformed data (note the different values on the axes). Right: Our inverse-transformed line correctly matches our starting data.*

The line in the center figure doesn't match the data at all, because it was fit to the transformed data. To get the line to match up with our input data, we need to run it through the `inverse_transform()` method of the `MinMaxScaler` that we used. To do this, we'll treat the endpoints of the line as two samples, and inverse-transform those two points.

The original data, along with the line after this inverse transformation, are shown on the rightmost image of Figure B1-12.

## Data Refinement

*See the Jupyter notebook* Bonus01-Scikit-Learn-PCA.ipynb.

Sometimes we have too much data.

Maybe some of the features in our data are redundant. For example, if our data records deliveries from a pizzeria, we might have fields for the size of each pizza and the size of the box it should be placed into. It's probably the case that the box size can be predicted from the pizza's size, and vice-versa.

The routines that perform *data refinement* are designed to locate and remove such redundancies from our data either by removing some features altogether, or by making new features out of combinations of others. Another class of routines, such as those related to the Principal Components Analysis (PCA) method, discussed in Chapter 10, are also able to compress data that's related but not entirely redundant. These trade off some loss of information for a simpler database by combining features.

Let's see an example of data compression, from 3D to 2D. Figure B1-13 shows a dataset of points drawn from a 3D blob shaped like a bulging ellipse. The X range is around [-0.8, 0.8], the Y range is around [-0.3, 0.3] and the Z range is about [–0.7, 0.7].

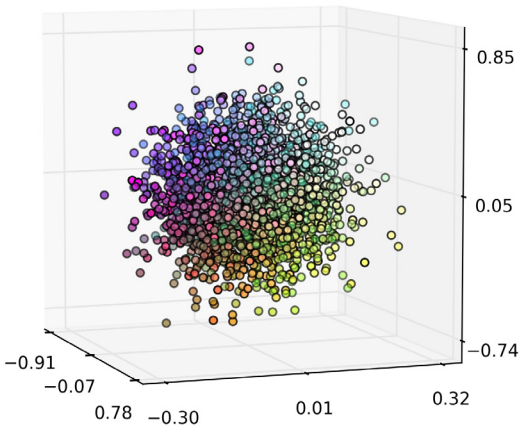



*Figure B1-13: Our starting 3D blob. The smallest range is in the Y values. The colors are just to help us keep track of where points are located.*

Let's reduce this 3D blob down to just 2 features (that is, we'll compress it into 2D data). We'll use PCA for this. Leaving out the code to build the blob and draw the plots, the core steps are only three: create the `PCA` object (and tell it how many dimensions we want), call `fit()` so it can determine which features to remove, and call `transform()` to apply the transformation. Listing B1-19 shows the code. Here we're also asking PCA to "whiten" the data, which can help PCA produce its best results.

```
from sklearn.decomposition import PCA

# make blob_data, the 3D data forming the "blob"

pca = PCA(n_components=2, whiten=True)
pca.fit(blob_data)
reduced_blob = pca.transform(blob_data)
```

*Listing B1-19: To apply PCA, we first make the `PCA` object. Here we tell it to reduce our data down to 2 features, and to whiten each feature along the way. Then we call `fit()` with our 3D data so the PCA algorithm can determine how to reduce it. Finally, we call `transform()` to apply the transformation and get back our reduced, 2D data.*

The result is shown in Figure B1-14. Each data point is now described by only two values, rather than three. In other words, the result of PCA is that our data has gone from being three-dimensional to two-dimensional. We requested PCA to save reduce our original 3 features into only 2 while retaining as much information as possible.

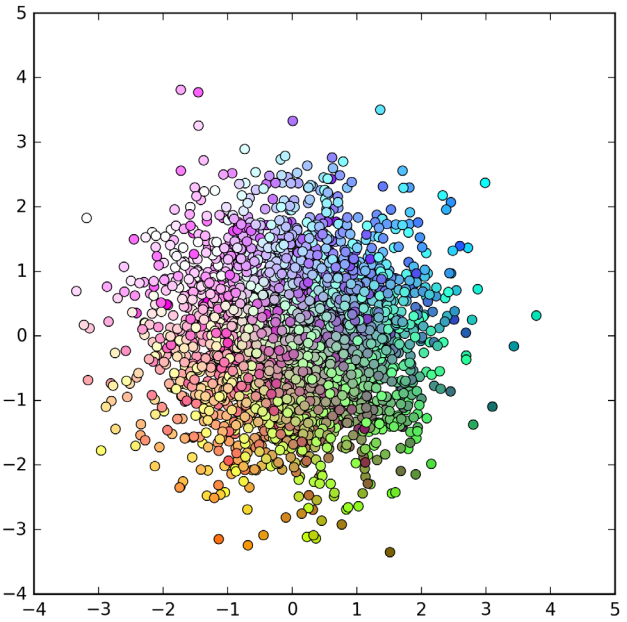



*Figure B1-14: Our starting blob of Figure B1-13 after running it through PCA.*

Note that it wasn't simply squashed along one dimension. As we discussed in Chapter 10, the algorithm placed a plane through the 3D blob so that it captured as much variance in the data as possible, and then projected each point onto that plane.

## Ensembles

*See the Jupyter notebook* Bonus01-Scikit-Learn-5-Ensembles.ipynb.

Sometimes it's useful to create a bunch of similar but different estimators, and let them all come up with their own predictions. Then we can use some kind of policy (usually voting) to choose the "best" prediction, and return that as the group's best result.

As we saw in Chapter 12, such collections of estimators are called *ensembles*. Let's see how to build and use ensembles in scikit-learn.

The system is set up so that we can treat an ensemble just like any other estimator. That is, we wrap up all the individual estimators into one big estimator, and use that directly. Scikit-learn takes care of all the internal details for us.

So just like any other estimator, our first step in using an ensemble is to create an ensemble object. Then we call its `fit()` method to give it data to analyze. Finally, we can call its `predict()` method to evaluate new data. We don't have to concern ourselves with the fact that there are multiple estimators hiding inside the estimator object we're using.

Some of the ensembles in scikit-learn will make these collections out of any kind of estimator, while others are limited to just classifiers, just regression algorithms, or even just specific instances of algorithms.

Let's build an ensemble to do classification. We'll use a data set of 1000 points, formed into 5 spiral arms, one for each of 5 classes. This starting data is shown in Figure B1-15.

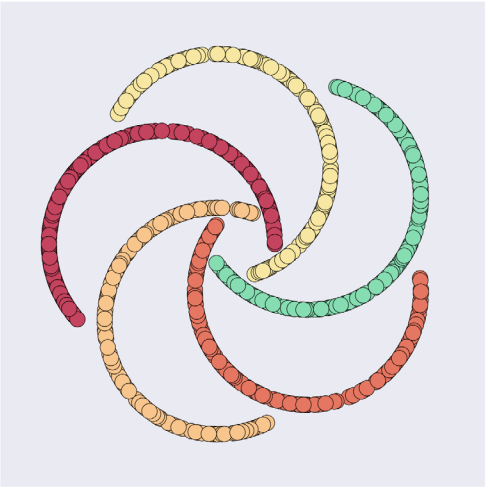


*Figure B1-15: Our starting data for ensemble classification. Each of the five arms contains 200 points of a different class.*

We'll use 2/3 of these points, randomly selected, as our training data, and the remaining 1/3 as test data.

We'll use a general-purpose ensemble maker which can build a group of classifiers form almost any specific classifier algorithm. The ensemble maker is called `AdaBoostClassifier`. The word `Boost` comes from the fact that internally it uses the technique of *boosting*, which we discussed in Chapter 12.

When we create the ensemble with this object, we tell it which algorithm it should build multiple copies of. We'll use the `RidgeClassifier` classifier, which is the classifier version of the Ridge regression algorithm we used above. Listing B1-20 shows the steps.

```
from sklearn.linear_model import RidgeClassifier
from sklearn.ensemble import AdaBoostClassifier

ridge_ensemble = AdaBoostClassifier(RidgeClassifier(), algorithm='SAMME')
```

*Listing B1-20: Creating an ensemble of `RidgeClassifier` objects using the `AdaBoostClassifier` ensemble object. The `algorithm` argument needs to be set to `SAMME` for this classifier (this is not a typo of the word "same"), as explained in the documentation.*

The documentation for `AdaBoostClassifier` explains that it has two different modes, depending on what methods are supported by the classifier it's building a collection of. In the case of the `RidgeClassifer`, we need to specify the `SAMME` algorithm (note that this is not the word *same*, but instead an acronym with two `M`s).

We can control how many classifiers go into our ensemble using the optional `n_estimators` argument. For this demonstration we'll leave it at the default value of 50 estimators. We'll also leave all the other arguments (which include the learning rate) at their defaults.

Now that our collection is made, using this ensemble of estimators is just like using one of them. We train the ensemble (and all the estimators inside of it) with samples we pass into it using `fit()`. Since we've build this ensemble for supervised learning of categories, the `fit()` routine takes in both the samples and their labels. Listing B1-21 shows the call to our ensemble's `fit()` routine.

```
ridge_ensemble.fit(training_samples, training_labels)
```

*Listing B1-21: Fitting our ensemble object.*

Finally, we can get new values out by asking the ensemble to predict them using `predict()`, as in Listing B1-22.

```
predicted_classes = ridge_ensemble.predict(new_samples)
```

*Listing B1-22: Using our ensemble object to make new predictions.*

Internally, ensembles usually pick the final output by using some kind of voting algorithm, where the most frequently predicted category becomes the final result.

Figure B1-16 shows the classification of our test data from our ensemble of 50 Ridge classifiers. These results aren't terribly encouraging.

*Figure B1-16: Using our ensemble of 50 Ridge classifiers. Left: The test data, color-coded by the category each point was assigned to. Middle: The regions created by our ensemble. Right: Overlaying the spiral points on the regions.*

The trouble is that we're trying to fit 50 straight lines to this data in a way that will let them correctly categorize our swirling data.

We could try to improve these results by experimenting with the number of estimators, the learning rate, or the arguments to the estimators.

Another approach would be to try another estimator. Let's use decision trees. The only changes we need to make are to import the necessary module, and then tell `AdaBoostClassifier()` to build the ensemble out of these classifiers, as in Listing B1-23.

```
from sklearn.tree import DecisionTreeClassifier

tree_ensemble = AdaBoostClassifier(DecisionTreeClassifier())
```

*Listing B1-23: Building an ensemble out of decision trees. For decision trees, we can leave the `algorithm` argument at its default value.*

Now we use `fit()` and `predict()` just as before Figure B1-17 shows the results of classifying with 50 decision trees. For this data, and using all the defaults, decision trees turned in a nearly perfect performance!

*Figure B1-17: The results of an ensemble made of 50 decision tree classifiers. Left: The test data, color-coded by the category each point was assigned to. Middle: The regions created by our ensemble. Right: Overlaying the spiral points on the regions.*

## Automation

*See the Jupyter notebook* Bonus01-Scikit-Learn-6-Automation.ipynb.

*See the Jupyter notebook* Bonus01-Scikit-Learn-7-Polynomials.ipynb.

Machine learning systems have lots of parameters, and often lots of hyperparameters, too. The distinction is that the system learns its parameters from the data, while we set the hyperparameters. Typical hyperparameters include the learning rate, the number of clusters in a clustering algorithm, the number of estimators in an ensemble, and the amount of regularization to apply.

We often want to try out many values of these hyperparameters to find the combination that gives us the best performance for a given system and data. If we can't find the very best results, we'd at least like to try a few combinations and use the ones that do the best.

If we want to automate this search process, we need two basic pieces. The first automates the selection of hyperparameters, choosing the values that get used to build and train a learner. The second piece evaluates that learner and assigns a score to its performance. We can then choose the combination that resulted in the best score.

Scikit-learn offers tools for both of these steps. We think of them as *automation* tools, since they handle this repetitive process for us.

### Cross-Validation

Most learning algorithms have multiple parameters and hyperparameters that control how quickly and how well they learn. Finding the best combination of these values can be difficult, because they depend on the nature of the data we're training with.

As we saw in Chapter 8, we can use *cross-validation* to determine the quality of a model on a given set of data. This technique is particularly attractive when we don't have a big training set, because it doesn't require us to remove a permanent validation set from the training data. Instead, we break up the training into equal-sized pieces called *folds*, and then train and evaluate our model independently many times, using the data in one of the folds as our validation set. The performance of the model is typically reported as the average performance of these multiple versions.

Scikit-learn lets us automate this process. We hand a routine our estimator, our data, and the number of folds we want it to use, and it carries out the whole process for us, reporting the average score when it's done. Many scikit-learn estimators have specialized versions that carry out cross-validation. These are consistently named by appending the routine's usual name with the letters `CV` at the end. They're all listed in the API documentation.

For example, let's suppose we want to use the Ridge classifier we used earlier to learn categories from a set of labeled training data. To evaluate its performance, we'll use cross-validation.

The Ridge classifier is implemented as an object called `RidgeClassifier`. Following the convention we just described, the version of `RidgeClassifier` object with cross-variance built in is called `RidgeClassifierCV`. So our first step is just to make one of these objects.

To carry out cross-validation on this estimator we need only call its `score()` method with our training data and labels. This one call does the whole cross-validation process for us from start to finish. The `score()` routine takes care of chopping up the data into folds, building and evaluating a `RidgeClassifier` for each version of the training data, and then returning the average of the scores. Listing B1-24 shows the steps.

```
from sklearn.linear_model import RidgeClassifierCV

ridge_classifier__cv = RidgeClassifierCV()
mean_accuracy = ridge_classifier_cv.score(training_samples, training_labels)
```

*Listing B1-24: To run the `RidgeClassifier` object through cross-validation, we first create an instance of the `RidgeClassifierCV` object. Then we call its `score()` routine with our training data and labels. The result is the average accuracy value from running through the training and validation process for each fold.*

We can pass a variety of optional parameters into our `RidgeClassifierCV` object when we create it. Perhaps the most important parameter is the number of folds we want to use (here we left it at the default value of 8).

`RidgeClassifierCV` uses a reasonable fold-making algorithm that just breaks up the data into equal pieces. This often works pretty well, but scikit-learn offers several alternatives, called *cross-validation generators*, that can sometimes do better. For instance, the `StratifiedKFold` cross-validation generator pays attention to the data when it creates the folds, and tries to

distribute the number of instances in each class equally. In other words, it tries to make sure each fold has roughly the same makeup as the entire dataset, which is always a good idea.

To use this approach, we first create a `StratifiedKFold` object, and tell it how many folds to use (that's the value of `K` in the object's name). The name of this argument for this object is unfortunately not named something like "number of folds," but is instead `n_splits`.

The `StratifiedKFold` object doesn't need our data when we create it, because the cross-validator will send the data to it for us when it uses this object to build the folds. We just provide our cross-validation stratifier object as the value of an argument named `cv` to the cross-validation object `RidgeClassiferCV`, and the rest happens automatically. Listing B1-25 shows the code.

```
from sklearn.model_selection import StratifiedKFold

strat_fold = StratifiedKFold(n_splits=10)
ridge_classifier_cv = RidgeClassifierCV(cv=strat_fold)
```

*Listing B1-25: To use a stratifier of our own choice, we have to first create it, which means we also have to import it. Here we import the `StratifiedKFold` object and tell it to use 10 folds. We pass that to our `RidgeClassifierCV` object through the argument `cv`.*

Let's look at this process visually. The scikit-learn model of cross-validation with a classifier is summarized in Figure B1-18. The input to the system consists of the labeled training data and the number of folds, along with the model constructor (that is, the routine that creates the classifier) and its parameters (which we've been leaving at their defaults in our examples). The routine then loops through each fold, removing it from the data, training on what remains, and evaluating the result on the extracted fold. The resulting scores are produced as output, or are averaged together to present just a single value. In this figure, and those to follow, a wavy box with arrows represents a loop. In this case, it's the loop that runs through the process above once for each fold.

Each time through the loop, one fold is extracted from the training data. The remaining samples are used to train a model, and then we evaluate the result on the validation fold, producing a score. The result is one score for each fold. We either present those scores, or their average, as output.

Figure B1-18 is missing any transformations. That is, we're not scaling or standardizing our data, or running feature compression on it, or any of the other transformations that we've seen, which can help our models learn.

To include these transformations in cross-validation we have to be careful. If we just drop a transformation into the inner loop of Figure B1-18 without thinking things through, we can risk information leakage. We saw in Chapter 10 that this can give us a distorted view of the performance of our model.

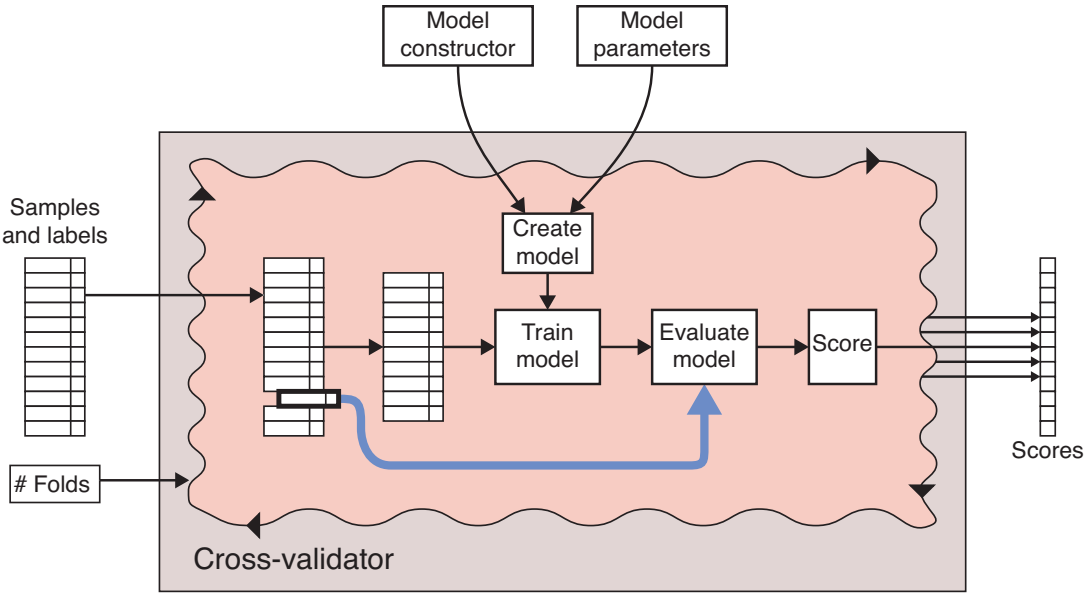



*Figure B1-18: A visualization of cross-validation with a classifier. The wavy box with arrows represents the loop of the process.*

To include transformations properly we can use another scikit-learn tool called a *pipeline*. A good way to understand pipelines is to see how they're used when we search for hyperparameters, which is an important topic on its own.

So we'll look at searching and pipelines next, and then we'll come back to cross-validation and see how to use a pipeline to include transformations.

## Hyperparameter Searching

We often distinguish between *parameters*, which are usually adjusted by the algorithms themselves in response to data, and *hyperparameters*, which we generally set by hand.

For example, suppose we want to run a clustering algorithm on a dataset. The cluster sizes and centers are parameters that are learned from the data, and maintained "inside" the algorithm. In contrast, the number of clusters to use is set by us "outside" of the algorithm, and we call such values hyperparameters.

Finding the best values for all the hyperparameters in a learning system is often a challenging task. There may be many values to tinker with, and they might influence one another. So using an automated approach to searching for them can save us a lot of time and hassle.

We can even generalize the idea of searching for hyperparameter values to searching for a choice of algorithm. For example, we might want to try out several different clustering algorithms, looking for the one that does the best job with our data.

To make this search easier, scikit-learn provides a bunch of routines that automate these types of searches for parameters, hyperparameters, and algorithms.

We identify the types of things we want to search over and the values we want them to try out, and the criteria on which to judge each result. Our search process then runs through every combinations of values, ultimately reporting the set that produced the best results.

Note that although scikit-learn makes it easy to set up and run this search, it's not any faster than if we were to do it by hand (except for the typing time). It just methodically grinds though the values to be searched, then builds, trains, and measures the resulting algorithm, over and over again. The upside is that we can be asleep, or reading a book, or doing anything else we please while the computer works its way through the combinations.

There are two popular ways to approach this kind of search: the *regular grid* and the *random search*.

The regular grid tests every combination of the parameters. The word "grid" is in its name because we can visualize all the combinations it tries as points on a grid. A 2D example is shown in Figure B1-19.

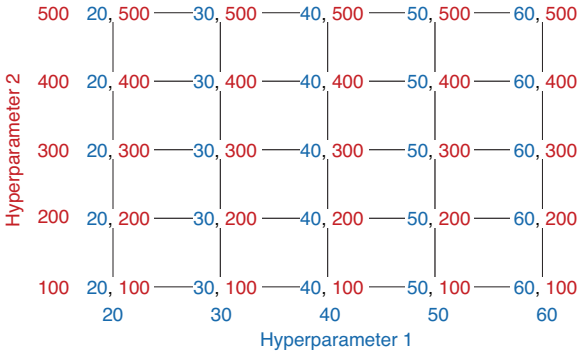



*Figure B1-19: A regular grid search using two hyperparameters, each with 5 values. For every combination of the two values, we build, train, and test a new system. There are 25 combinations to be tried.*

In this example, we have 5 values for each of two hyperparameters, giving us $5 \times 5 = 25$ combinations to examine.

As we include more parameters to be searched, or more values for each one, the size of this grid, and thus the number of training/measuring steps we have to carry out, will grow correspondingly. This means the whole thing will require more and more time to run. Eventually it could require more time than we have available. For example, if we have 4 hyperparameters to search, and 6 values for each one, that's $6 \times 6 \times 6 \times 6 = 1{,}296$ different combinations. If it takes an hour to train and test each model, that search would take 54 days of non-stop computing!

It would be great to cut away parts of the grid and save time, but that's a risky step because we can't know beforehand what combination of variables might turn up the best results.

A grid search usually proceeds methodically, working its way through all the possible combinations in a predictable, fixed order, such as left to right, and top to bottom.

A faster but less informative alternative is to search these combinations *randomly*, rather than in some fixed order. The algorithm picks a random combination of hyperparameter values that hasn't been tried yet, trains and measures the resulting model, and then picks another untried random combination, and so on. We can think of many different conditions to control when this process ought to stop. For example, the algorithm could stop when it's searched every combination, or it's tried a certain number of combinations that we've specified, or it's run longer than an amount of time that we've specified, or we simply get tired of waiting and stop it manually. Then it returns the best combination it found. Figure B1-20 illustrates the idea. Random sampling can give us an overall feeling for where the big scores are located without exhaustively trying every combination first. Here, we ran through 9 steps of picking a random combination, then built and measured the resulting model.

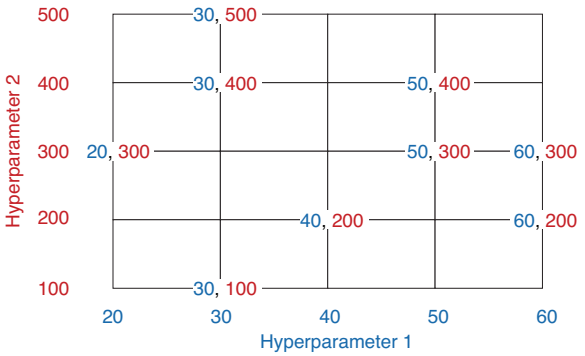



*Figure B1-20: A random search doesn't test every combination in sequence, but tests them in random order.*

The advantage of this approach over the regular grid is that we can watch the results as they come in, and perhaps get an idea for where we can focus our search. Then we can stop and start again in the neighborhood that looks promising. For example, suppose that in Figure B1-20 the combination (40, 200) performed much better than any of the other combinations. As a result, we might decide to run a new close-up search in that area, perhaps using a regular grid, as in Figure B1-21.

This is called *multiresolution searching*, since we're changing the resolution, or step size, between grid elements as we look more and more closely for the best combination.

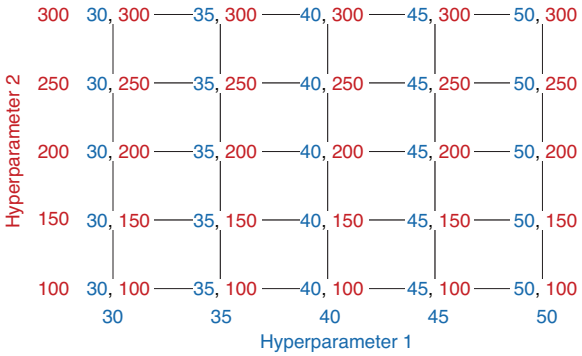



*Figure B1-21: When we think we've found the neighborhood of the best values, we can run a new search that zooms in on that region. Here we look around the region where hyperparameter 1 has a value of 40, and hyperparameter 2 is 200.*

Let's look at how to use scikit-learn to run a regular, methodical grid search first. We'll see that we set up and call the randomized version in almost an identical way.

## Exhaustive Grid Search

The exhaustive, methodical grid-based search is provided by an object called `GridSearchCV` (the `CV` at the end reminds us that the algorithm will use cross-validation to evaluate the performance of each model it tries out).

`GridSearchCV` produces each combination of parameters one by one, builds and trains the model, and then measures how well the corresponding model performs on our data, returning a final score for that model.

Note that the estimator we hand to this routine shouldn't perform cross-validation itself (that is, its name shouldn't end in `CV`), because the grid searcher is handling that step.

Figure B1-22 shows the pieces that go into making this grid-search object.

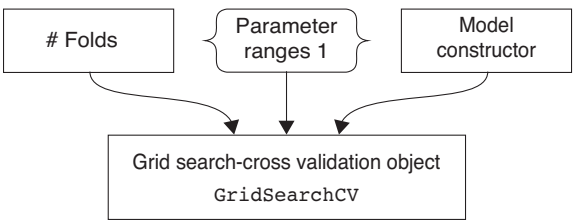



*Figure B1-22: The pieces that go into making a basic grid-search object. We supply the number of folds to use during each cross-validation, the parameters we want searched, and the values we want them to take on, and a constructor for the model that we're investigating.*

The object takes a number of folds for the cross-validation step (the default is 3), a set of parameters and their values that should be searched, and the routine that builds our learner. There are many more optional parameters that we won't get into.

Conceptually, we can think of the grid search process in two steps. The first step builds a list of every combination of values of the parameters. So if there are just two parameters, we could save this list as a 2D grid. If there are three parameters, we could think of it as a 3D volume, and so on.

The second step runs through those combinations one at a time, building the model, evaluating it with cross-validation, and saving the score. We could save that score in another list (or grid, volume, or bigger structure) of the same shape and size as the list of parameter combinations. Then we can quickly see what score got assigned to each combination.

Figure B1-23 shows a visual summary of the whole process. At the left we see the routine that creates our learning model (perhaps a decision tree algorithm, or a regression algorithm like Ridge). There's also a collection of the parameters we want to vary, and the values we want to try for each one. Let's assume we're interested in exploring two hyperparameters, and we want to try 7 possible values for the first, and 5 possible values for the other.

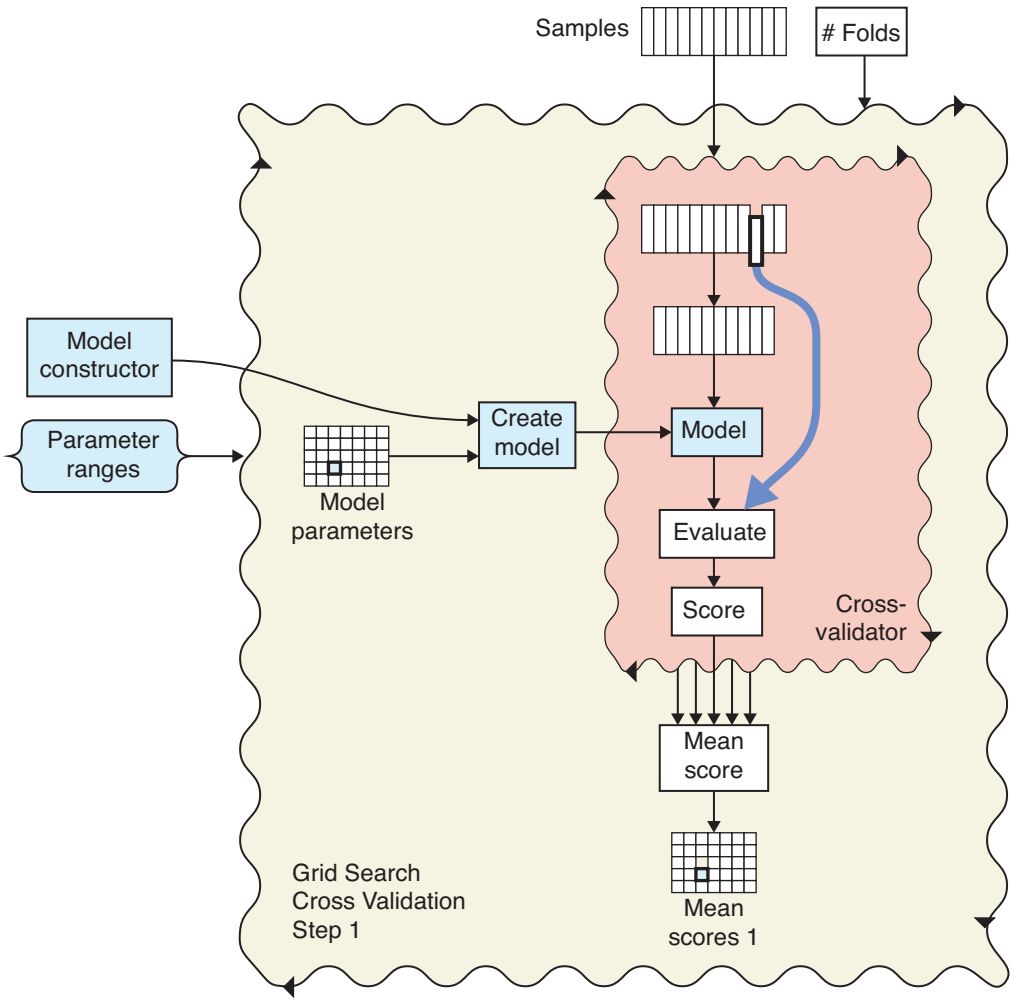



*Figure B1-23: Grid searching with cross-validation.*

In the first step of processing, we find all combinations of those parameters and values and save them in the grid marked "model parameters." This grid is 7 by 5, with one entry for each combination of values for the two hyperparameters.

Now we start running the loop. The outer wavy box represents the main loop of the grid searcher. Each time through the loop it uses the model constructor, and one set of model parameters from the grid, to create our model. At the top we can see the samples that make up our training set. If we're using a supervised learner, these samples will have labels. We also provide the number of folds to use when cross-validating.

All of this goes into the inner wavy box, which contains the cross-validation step we saw in Figure B1-18 (though turned on its side). The scores from the cross-validation steps are saved and then averaged, with the result saved in a grid the same shape and size as the model parameters, so it's easy to find the score for each combination of parameters.

Once the searching loop is completed, we look through all the scores saved during the search process and find the parameters that produced the best results. We make a new model using those parameters, train it, and return it, as in Figure B1-24.

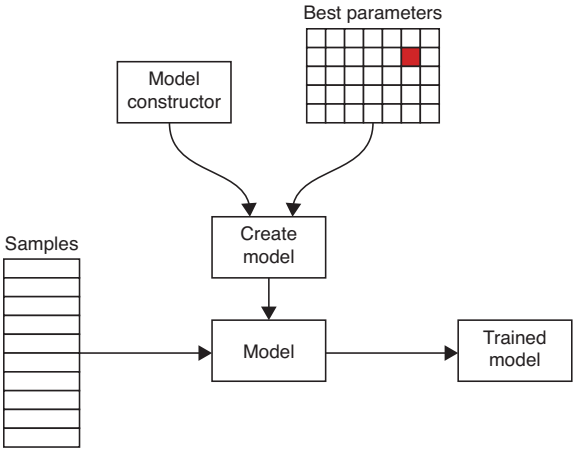



*Figure B1-24: We finish the grid-search process by identifying the parameters that gave us the best score. We build a new model from those parameters, train it on all the data, and return that model.*

Let's see some code.

We'll use a Ridge classifier from before to classify a bunch of data. The `RidgeClassifier` object takes a lot of arguments, which we've so far been ignoring. Let's pick two of those arguments and search for their best values for a particular set of data.

We'll start with the parameter named `alpha`, which is the regularization strength (this is often called lambda (λ), but it's not unusual to see different Greek letters used for this). Recall from Chapter 9 that regularization helps

us prevent over-fitting. In the RidgeClassifier, alpha is a floating-point value that's 0 or greater. Larger values mean more, or stronger, regularization. As a first guess, we'll try these six values of alpha: 1, 2, 3, 5, 10, 20.

The second parameter we'll search for is called solver. This is the algorithm used internally by RidgeClassifier to do its job. Without going into the details of each one, let's say we'd like to try out a few and see if any one performs particularly well. The documentation for RidgeClassifier tells us that we refer to these algorithms with a string. Let's arbitrarily pick three of the available options: 'svd', 'lsqr', and 'sag'.

Now we need to tell the grid searcher that we want to train and score multiple versions of the RidgeClassifier, using these values for the parameters named alpha and solver.

To communicate which values we want to assign to each parameter, we use a Python *dictionary*. In a nutshell, a dictionary is a list of key/value pairs enclosed in curly braces. Our keys will be the strings that name our parameters, and our values will be lists that the parameters can take on.

Thanks to how Python is structured, we can just name the parameters we want to search on as strings, and use them as the keys in our dictionary. The values we want each argument to take on are named in a list, and assigned to the value for that key. Our dictionary is given in Listing B1-26.

```
parameter_dictionary = {
    'alpha': (1, 2, 3, 5, 10, 20),
    'solver': ('svd', 'lsqr', 'sag')
}
```

*Listing B1-26: Building a dictionary to hold the values we want to search over.*

When the grid searcher builds a new RidgeClassifier, it will assign one value in the alpha entry to the alpha argument, and one value in the solver entry to the solver argument. It does this by matching up the names, so the names in our dictionary must exactly match the parameter names used by RidgeClassifier().

The grid searcher will build and score 6 × 3 = 18 different models from this dictionary. The models will be made with calls like those in Listing B1-27, though this all happens invisibly to us. Because of how Python's dictionaries are defined, the algorithm might not go through the choices in this order, but that detail will usually also be invisible to us.

```
ridge_model = RidgeClassifier(alpha=1, solver='svd')
ridge_model = RidgeClassifier(alpha=1, solver='lsqr')
ridge_model = RidgeClassifier(alpha=1, solver='sag')

ridge_model = RidgeClassifier(alpha=2, solver='svd')
ridge_model = RidgeClassifier(alpha=2, solver='lsqr')
ridge_model = RidgeClassifier(alpha=2, solver='sag')

ridge_model = RidgeClassifier(alpha=3, solver='svd')
....
```

```
ridge_model = RidgeClassifier(alpha=20, solver='lsqr')
ridge_model = RidgeClassifier(alpha=20, solver='sag')
```

*Listing B1-27: The start and end of the 18 models that the grid searcher will automatically build, based on the dictionary of Listing B1-26. Each model is automatically trained and measured using cross-validation. They may not be generated in this order.*

To run the search, we make a `GridSearchCV` object with the three items shown in Figure B1-22. These are the number of folds to use, the dictionary of parameters and ranges, and the model constructor. Listing B1-28 shows the code.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import RidgeClassifier

ridge_model = RidgeClassifier()
parameter_dictionary = {
    'alpha': (1, 2, 3, 5, 10, 20),
    'solver': ('svd', 'lsqr', 'sag')
}
num_folds = 3

grid_searcher = GridSearchCV(estimator=ridge_model,
                             param_grid=parameter_dictionary,
                             cv=num_folds)
```

*Listing B1-28: Building a grid searcher. We provide it with the estimator we want it to use (here a `RidgeClassifier`), the parameter dictionary with parameter names and values, and the number of folds to use. We could put the number of folds in the dictionary if we wanted to search different values of that as well. To show that we don't need to, here we're passing it as a constant value.*

Hold on a second. Something's fishy here. We saw in Listing B1-27 that the grid searcher is going to make 18 different `RidgeClassifier` objects, each with different parameters. But in Listing B1-28 we made one `RidgeClassifier` and stored it in the variable `ridge_model`, and we gave it no parameters at all. How does the grid searcher take this one object and make 18 new versions of it, each with different parameters?

The answer to this question is also based on how the Python language is set up. Let's look at what's happening at a very high level.

We're providing to the searcher a routine that makes a `RidgeClassifier` object as an argument to its `estimator` parameter. As a result, the grid searcher is able to call that routine with new arguments to make new models. In other words, it can do exactly what Listing B1-27 shows, but using the argument to `estimator`. Since the value we're assigning to this argument is itself a procedure, invoking that procedure is the same as explicitly invoking `RidgeClassifier()`. This mechanism makes it easy for us to later call `GridSearchCV()` with a completely different estimator if we want, just by changing the routine we provide to `estimator`. If the new model takes the same `alpha` and `solver` parameters, then we can leave our dictionary just as it is. If it takes other parameters, we'd want use a different dictionary.

In short, the grid searcher generates all combinations of the parameters, passing each combination to whatever routine we provided as a value of `estimator` in order to create a new instance of the estimator object. In this case, it will make a new `RidgeClassifier` object. It then runs that object through cross-validation using our data and evaluates its performance.

Creating the `GridSearchCV` object prepares it to run the search, but doesn't actually start the process. To run the search, we call the `GridSearchCV` object's `fit()` method. Since in this example we're training a classifier, we'll give it our samples and our labels. Listing B1-29 shows the code.

```
grid_searcher.fit(training_samples, training_labels)
```

*Listing B1-29: To run the search, we call the searcher's `fit()` method with our training data and samples.*

That's all it takes. We make that call and then everything happens automatically. When this line returns, the system has exhaustively searched all combinations of our parameters.

We should always pause for a moment before starting a grid search, because we can be in for a long wait. A *long* wait. As we saw before, if we're searching a lot of parameters, and each run takes a while, we might wait for hours or weeks (or more!) for `fit()` to finish its work.

When `fit()` is done, we can query our `grid_searcher` object to discover what it found. The object saves its results in internal variables. Each of its variables ends with an underscore to help us avoid confusing those names with our own variables.

The documentation provides a complete list of all the internal variables that contain our results. Let's look at three of the most useful variables. `best_estimator_` (note the trailing underscore) tells us the explicit construction call that the searcher made to make the estimator that resulted in the best score, with all the arguments (including all the defaulted arguments we didn't specify). That best score itself is given by `best_score_` (again with a trailing underscore). If we just want the best set of parameters, then `best_parameters_` (with a trailing underscore again) contains a dictionary that has just the best value for each parameter from our original dictionary.

Let's put this into action. In Figure B1-25 we show some data in a pair of half-moons (generated with the scikit-learn data-making utility `make_moons()`, which we'll see below). On the right we show the results of creating the grid search in Listing B1-28 followed by the fitting step in Listing B1-29. Since we're using a `RidgeClassifier`, we'd expect the straight line that does the best job of separating these two classes, though no straight line is doing to do a perfect job. The figure shows that the classifier found a good straight-line approximation to split the data. A more complicated classifier could have found a more complex and accurate boundary curve.

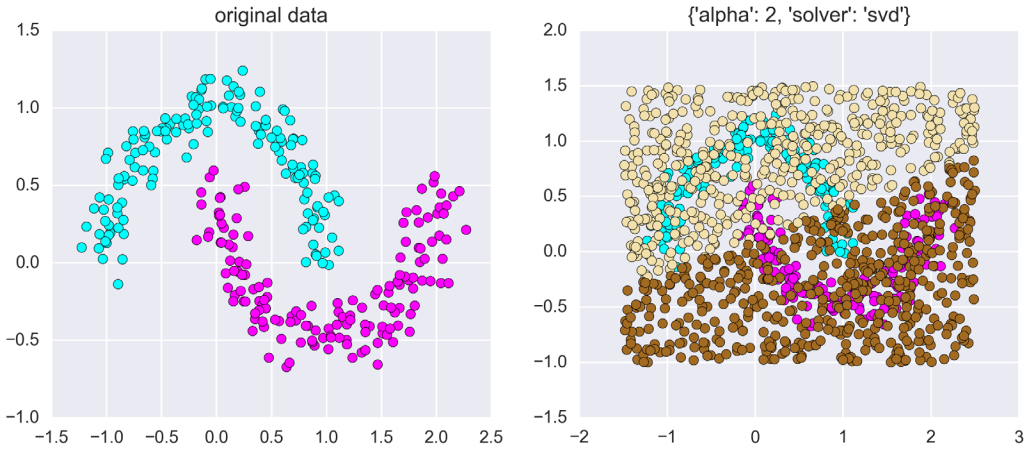



*Figure B1-25: On the left, our original training data with 2 categories. On the right, the result of our grid search, with 500 predicted points drawn on top of the original data.*

For clarity, Figure B1-26 shows just the test data by itself. Each point is colored by the class it was assigned by the classifier.

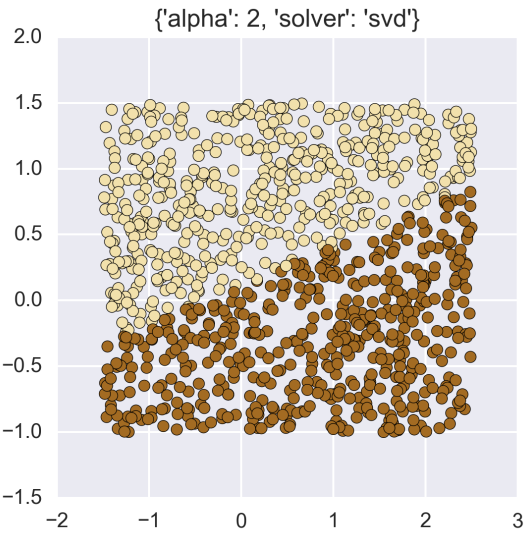



*Figure B1-26: Just the test data from Figure B1-25*

In Listing B1-30 we show the resulting values of three variables we just discussed, and their values. Outputs from the computer are shown in red.

```
grid_searcher.best_estimator_
    RidgeClassifier(alpha=2, class_weight=None, copy_X=True,
    fit_intercept=True, max_iter=None, normalize=False,
    random_state=None, solver='svd', tol=0.001)
```

```
grid_searcher.best_score_
    0.87
grid_searcher.best_parameters_
    {'solver': 'svd', 'alpha': 2}
```

*Listing B1-30: The variables describing the best combination found for our data in Figure B1-25*

Note that the best estimator provides us with the full call to `RidgeClassifier` with all of its parameters, including the optional ones. The parameter `best_parameters_` is a dictionary, here with 2 key/value pairs.

The output shows that `best_estimator_` contains everything that's in `best_parameters_`, but the latter restricts itself to just the parameters we were searching on.

If we dive deeper into the variables in our `grid_searcher` object we can look at `cv_results_`, which is a huge dictionary with detailed results from the search. In Figure B1-27, we've plotted data from two entries from that variable, `mean_train_score` and `mean_test_score`, which give us the scores for the training and testing data for each of the 18 parameter combinations (these names don't end with an underscore, because they are members of the `cv_results_` dictionary which does have an underscore. It's a quirk of scikit-learn's naming policy). We can see that in this run that three combinations of the parameters gave us the same, best testing results. That is, the choice of `solver` didn't make any difference in this situation.

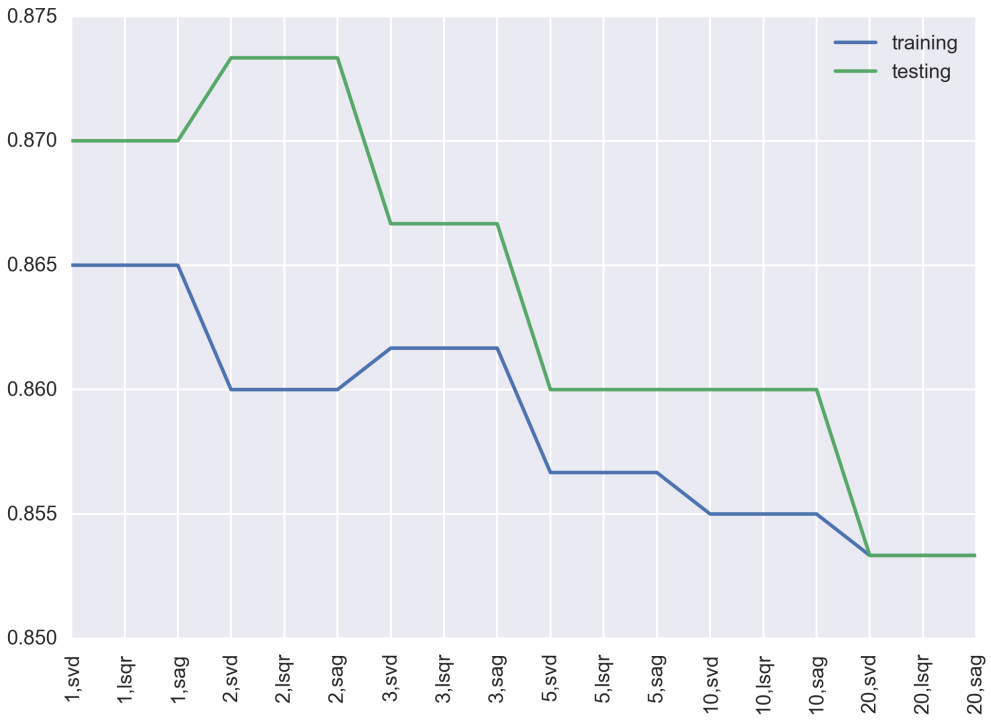



*Figure B1-27: Training and testing scores for each of our 18 searches. The best training results came from using `alpha=2` and any of the three `solver` choices.*

The results show that using an alpha value of 2 produced the best results on the testing sets, regardless of the algorithm used internally by `RidgeClassifier`. Since the testing data is a proxy for real data, we'd want to use `alpha=2` if we were going to deploy this algorithm. The system picked the `svd` algorithm in the "best" variables from the 3 equally performing combinations because it was the first one encountered.

## Random Grid Search

Now that we know how to do the exhaustive version of grid searching, making the switch to the random version is almost no work at all.

Instead of using a `GridSearchCV` object, we use a `RandomizedSearchCV` object (it also comes from the `model_selection` module).

This object takes the same arguments as `GridSearchCV`, but the randomized version also takes an argument called `n_iter` (for "number of iterations"), telling it how many unique, randomly-chosen parameter combinations it should try (it defaults to 10).

## Pipelines

Suppose that as part of our grid search, we'd like to include some form of data pre-processing. We might want to normalize or standardize the training set, or do something more complex like running PCA on it.

It would be natural to search for the best parameters for that transformation while we're searching for other parameters, like those we saw above. For example, we might want to try using PCA to reduce an 8-dimensional data set down to 5, 4 and 3 dimensions, and see which (if any) of those choices give us the best performance.

But now we have two objects inside the loop: the pre-processing object and the classifying object. How do we tell the searcher, whether systematic or randomized, to use both of these, and how to we tell it which parameters should be delivered to which object?

The answer is to package up the whole sequence of actions we want performed into a *pipeline*. Then the searcher will proceed as usual, only instead of just calling an estimator, it will call the pipeline, and execute all the steps inside of it.

Let's demonstrate this using the same half-moons data we used in Figure B2-25.

To illustrate the pre-processing step, let's transform our data every time through the loop in such a way that the `RidgeClassifier` object will be able to fit a curve to the data, rather than just a straight line.

To do this, we'll use a pre-processing step to produce *more data* for the classifier. This is a technique we haven't seen before, so let's see what it does before we go into the code to create and use it.

When a `RidgeClassifier` object finds a boundary curve, it builds it from combinations of the features in the samples. If all we give our classifier are the $x$ and $y$ values of the two features in our samples, as we did earlier, then looking at the math we'd see that the most complex shape the classifier can

build from combining them is a straight line. In other words, the reason we got a straight line from `RidgeClassifier` in Figure B1-25 isn't because of the classifier, but because the data we gave it had only the two features of $x$ and $y$.

If we create some additional features out of the original data, then the classifier will have more to work with. We'll make those features by multiplying together the $x$ and $y$ values in different ways.

For instance, we can multiply the $y$ value with itself a bunch of times. This would make $y \times y$, or $y \times y \times y$, or $y \times y \times y \times y$, and so on. Mathematicians call these *polynomials*. The number of features that are used in these little expressions is called the *degree* of the polynomial. Figure B1-28 shows the shapes of these polynomials made up of repeatedly multiplying $x$ by itself.

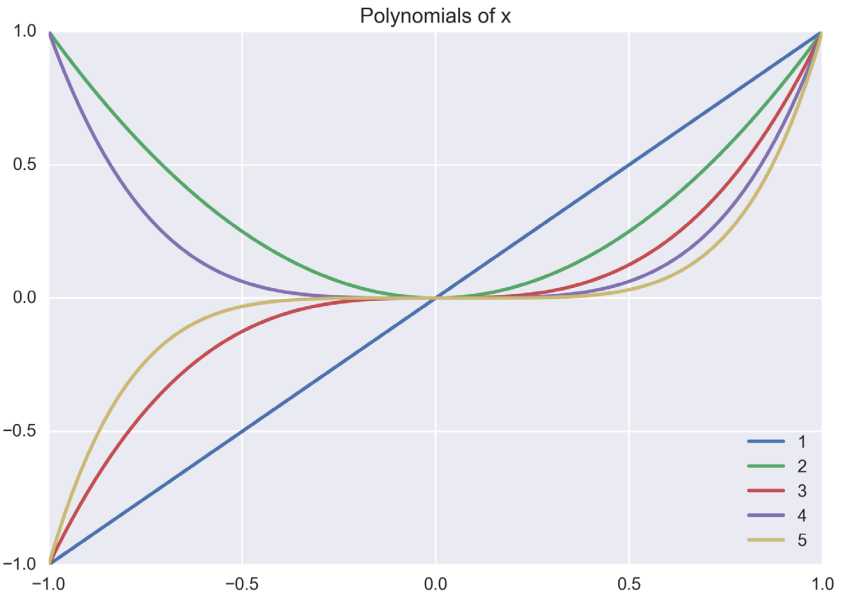



*Figure B1-28: The polynomials of x plotted from –1 to 1. The number in the legend is the degree, and tells us how many variables are used to make that curve.*

So $x$ by itself is a polynomial of degree 1, $x \times x$ is of degree 2, $x \times x \times x$ is of degree 3, and so on. Note that the odd-numbered polynomials are both negative and positive, while the even-numbered polynomials are strictly 0 or larger.

We've shown just polynomials built from $x$, but we can also build versions of $y$ multiplied by itself any number of times. And we can also mix $x$ and $y$ together.

For example, the three possible second-degree polynomials are $x \times x$, and $y \times y$, and $x \times y$. When we get up to the third degree we have four types of expressions: $x \times x \times x$, and $x \times x \times y$, and $x \times y \times y$, and $y \times y \times y$. There are only four combinations because the order in which we multiply the values doesn't matter to the end result.

As we mentioned, when the Ridge classifier only gets the polynomial of degree 1 for both $x$ and $y$ (that is, we give it just the values of $x$ and $y$

themselves), the most complex shape it can make is a straight line. If we also give it the polynomials of degree 2 (as we saw, these are $x \times x$, and $y \times y$, and $x \times y$), then it can combine these to make a more interesting curve. The higher the degree of the polynomials we give to the Ridge algorithm, the more complex a curve it can create.

Figure B1-29 shows just a few combinations of the curves in Figure B1-28 (these curves contain only polynomials of $x$). This shows that by adding up several of these curves, each with its own strength, we can make some pretty complicated shapes. At each point we found the values of the 5 curves, multiplied each value by a scaling factor for that curve, and added up the results.

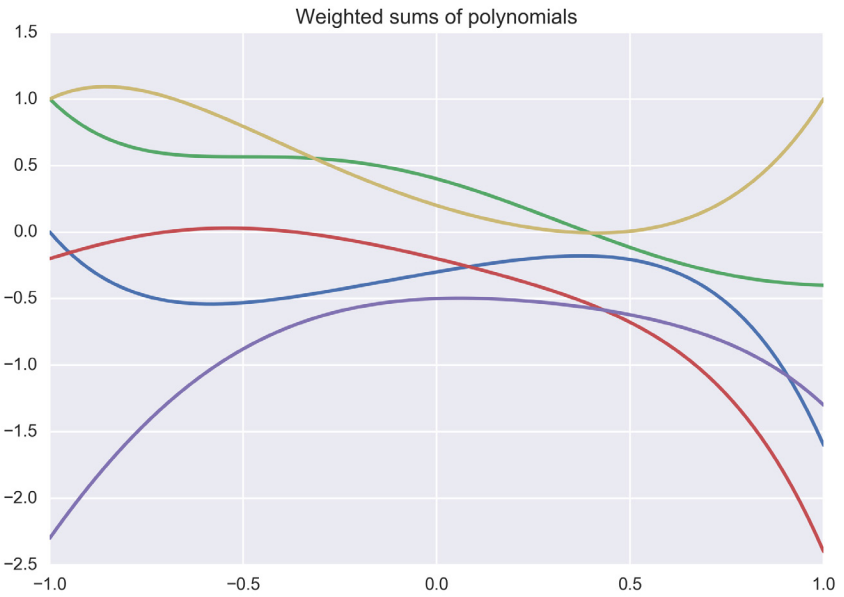



*Figure B1-29: Some weighted combinations of the polynomials of x in Figure B1-28*

Figure B1-29 shows curves using just the $x$ value. Things get really interesting when we use polynomials in both $x$ and $y$ simultaneously. Without going into the details, we can follow a recipe like that in Figure B1-29 to create a wide variety of 2D curves. Figure B1-30 shows a few examples.

If we give `RidgeClassifier` not just the $x$ and $y$ values, but these polynomials that come from multiplying those values together in different ways, it can create these kinds of curves. This is much better than a straight line!

We say that by creating these additional features, we're *extending* or *augmenting* our original list of features with new *polynomial features* made from the original values of $x$ and $y$. In our NumPy array of data, it just means that each row gets longer, from having only 2 features to having more of them, computed by multiplying various combinations of $x$ and $y$.

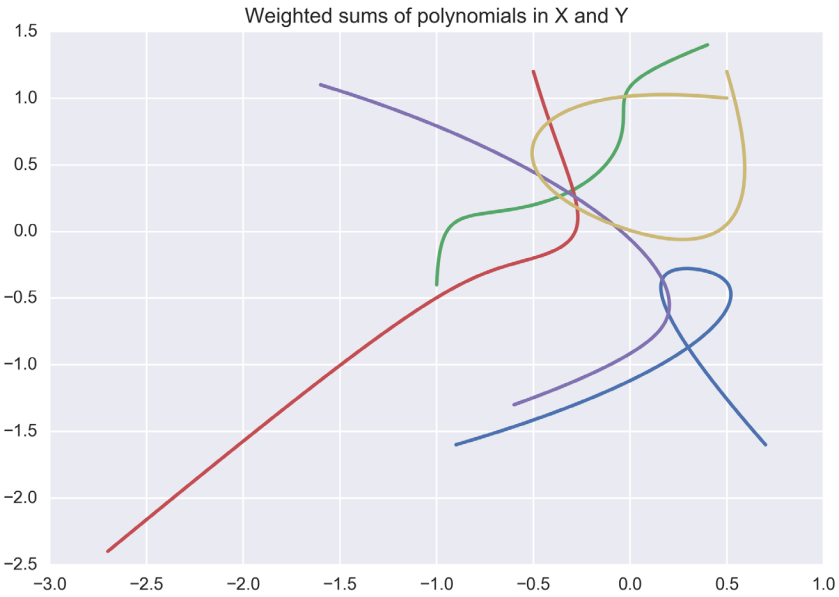



*Figure B1-30: Some weighted combinations of polynomials in both X and Y*

The first step of our pipeline will be to build these new polynomial features for every sample, so the Ridge classifier will have them to work with. Now we're ready to identify the new object that will do the job, named `PolynomialFeatures`. We tell it the degree of the features we want it to make, and it cranks them out and appends them to each sample. The more polynomials we include with each sample (that is, the higher their degree), the more complicated Ridge's boundary curve can become.

Wait a second. When we create these polynomial features, we're not including any new information to our data set. We're just using what we have and multiplying the values together. How does that give us curves rather than straight lines?

The answer comes down to the details of how these algorithms are implemented. It's true that we're not providing any new information. But `RidgeClassifier` is designed to work with the data it gets, not all possible variations on that data that might help it produce a better answer. So if we give `RidgeClassifier` our simple 2-feature data, it finds a line. By providing it with the polynomials, it uses those as part of its calculations to find a curve.

Conceptually, we could have an argument to `RidgeClassifier` that we could use to tell it to create this extra data all by itself, internally. But scikit-learn is instead set up to make it our responsibility to create that extra data. That way we have complete control over just what we give to `RidgeClassifier`, and thereby control the complexity and shape of the boundaries it can find.

But how much more data is best? As we saw in Chapter 9, at a certain point these increasingly complicated curves can start to overfit the data. So we'll want to stop including new features before that happens. On the other hand, we have the regularization parameter `alpha` in the Ridge classifier

which helps us control overfitting. Maybe by increasing the value of `alpha` we can use more of these combined features, and thus use a more complicated (and perhaps better-fitting) curve.

This is getting very complicated! What's the best balance between curve complexity (given by the parameter `degree` to `PolynomialFeatures`), and regularization strength (given by the parameter `alpha` to `RidgeClassifier`)?

There's no need to try to work this out ourselves. Let's just build a two-step pipeline from these two objects and give the searcher a bunch of values to search. Then it can do the work to find which combination works best. If we use the grid searcher, it will try every combination of every parameter.

Figure B1-31 shows a grid searcher with a two-step pipeline replacing the single model we had in Figure B1-23. The first step creates a `PolynomialFeatures` object, and the second creates a `RidgeClassifier` object that uses the augmented data that comes out of the first step. This is a simplified diagram, because the fold we're using for validation is going through a transformation (marked T) that doesn't seem to be coming from anywhere. We'll return to this diagram in the next section and fill in that gap.

Each step in the pipeline can have its own set of parameters to be searched. As before, each wavy box represents a loop.

Our first step in building our pipeline will be to create our `PolynomialFeatures` object, which produces those extra combinations of *x* and *y*. The only argument we care about now for this object is `degree`, which tells it how many polynomials to build from the original features. Larger values of `degree` will generate and save more of these multiplied-together features, for every sample, which will let the Ridge classifier find more complicated boundary curves.

For the Ridge classifier, let's also search for just one parameter, the regularization strength `alpha`. We'll leave the internal algorithm at its default value (that's the string `auto`, which automatically picks the best algorithm).

Now that we know the two steps that make up our pipeline, let's write the code to create it.

Scikit-learn offers more than one way to build a pipeline. We'll take the approach that's easiest to program and use.

We begin by making the objects that will go into our pipeline. In this case, it's the `PolynomialFeatures` object and the `RidgeClassifier` object. We can make them as in Listing B1-31.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import RidgeClassifier

pipe_polynomial_features = PolynomialFeatures()
pipe_ridge_classifier = RidgeClassifier()
```

*Listing B1-31: We start making our pipeline by creating the objects that will go into it.*

As before, neither of these objects has any arguments, because the searcher will fill those in for us as it searches.

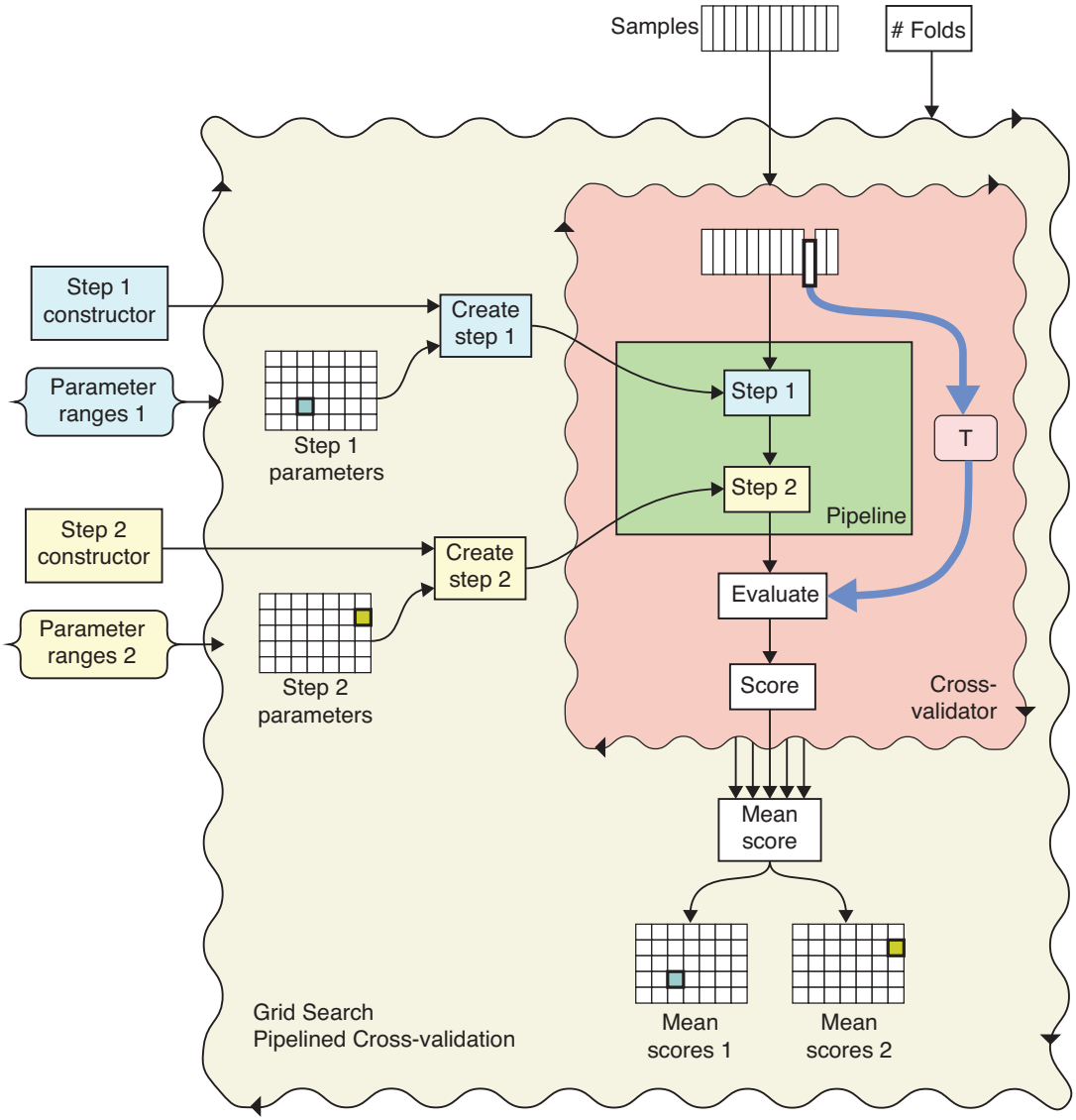



*Figure B1-31: A simplified version of our grid search object with a pipeline (the innermost, green box) replacing the single model of Figure B1-23*

Now that we have our objects, we can build the pipeline. To do this, we create an object named `Pipeline`, and we hand it a list that contains one entry for each step. Each entry is itself a list with two elements: a name we pick for that step, and the object that implements it.

The pipeline is built from this list of objects, *in the order in which we name them*. The list can be as long as we like, as long as we have a name and object for each step.

Listing B1-32 shows this construction step for our two-step pipeline. Notice the order of the objects. We name the `PolynomialFeatures` object first because it comes first in our intended sequence of steps.

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([('poly', pipe_polynomial_features), \
                     ('ridge', pipe_ridge_classifier)])
```

*Listing B1-32: To create our pipeline object, we give `Pipeline()` a list of steps. Each step is itself a list, containing a name we give to that step, and the object that performs it. Our objects are those we just created in Listing B1-31.*

The names we pick when we make the pipeline are like the names we pick for variables: they can be anything we like, but it's best to pick something descriptive. We're using very short names here to conserve space.

We've completed our pipeline, and we will soon hand this to a `GridSearchCV` object as the value of its `estimator` argument.

The last thing to do is make the dictionary of parameters for the searcher to build its combinations from.

If we make our dictionary of parameters for the pipeline in just the same way as we made our dictionary before, sooner or later we'll hit a problem. For example, we know that `RidgeClassifier` takes an argument called `alpha`. What if the `PolynomialFeatures` object also took an argument called `alpha` (it doesn't, but it could)? And what if we wanted to use values (1,2,3) for the first `alpha`, and (`'dog'`, `'cat'`, `'frog'`) for the second `alpha`? We need some way to tell the grid searcher which set of values should go to which parameter in which object.

The way scikit-learn answers this question is both sensible and weird. The sensible part is that we name each parameter using the name of the pipeline object for that parameter (that's why we gave our objects names when we made the pipeline), followed by the name of the parameter. Note that we don't use the name of the object (in our example, `pipe_polynomial_features` or `pipe_ridge_classifier`), but the name of the *pipeline step* (in our example, `poly` or `ridge`).

The weird bit is that we join the pipeline object's name and the parameter's name with *two underscore characters*. That is, two _ in a row, or __. This is easy to confuse with just one underscore, which is unfortunate. But two underscores must be used.

So to refer to the `alpha` parameter for our pipeline step named `ridge`, we'd name it in the dictionary as `ridge__alpha`, with two underscores. Similarly, the `degree` parameter for our `poly` pipeline step is named `poly__degree`, again using two underscores.

We always create our dictionary names by assembling the pipeline step name, two underscores, and the parameter name, even when there's no chance of confusion.

Listing B1-33 shows a dictionary for those two parameters, along with some values we'll try for them.

```
pipe_parameter_dictionary = {
    'poly__degree': (0, 1, 2, 3, 4, 5, 6),
    'ridge__alpha': (0.25, 0.5, 1, 2 )
}
```

*Listing B1-33: A dictionary for our pipeline. Note that each parameter is given by the name of the pipeline step and the name of the parameter, joined with two underscore characters.*

This will cause the loop to run 7 × 4 = 28 times, but for this small dataset that takes only a few seconds on 2014-era iMac, without even touching the GPU.

Now we're set. We just build our search object as before, and then call its `fit()` routine. Listing B1-34 shows the code.

```
pipe_searcher = GridSearchCV(estimator=pipeline,
                             param_grid=pipe_parameter_dictionary,
                             cv=num_folds)

pipe_searcher.fit(training_samples, training_labels)
```

*Listing B1-34: Building our grid search object for our pipeline, and then executing the search.*

The results for our half-moon data are shown in Figure B1-32. The best combination the system found was `alpha=0.25` and `degree=5`. Thanks to the additional features we provided to `RidgeClassifier`, it was able to find a curve to split the data.

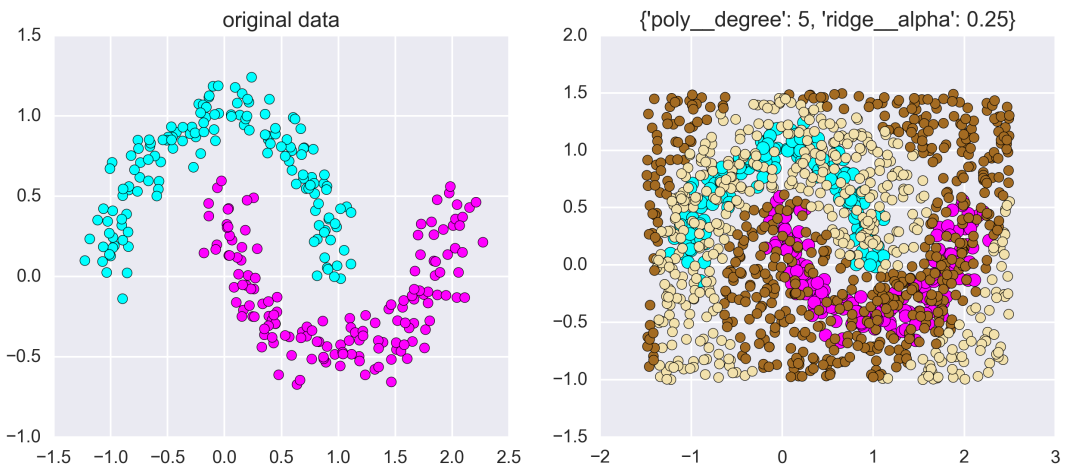



*Figure B1-32: Results for searching over our more flexible pipeline object*

For clarity, Figure B1-33 shows just the test data by itself.

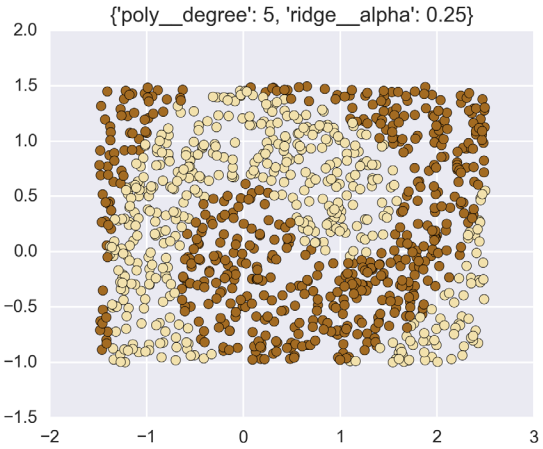



*Figure B1-33: Just the test data from Figure B1-25*

It's interesting to note that the boundaries are rather symmetrical, which makes sense given the symmetry in the input. We also see them seeming to curve around and re-appear in the corners. This is kind of wild, but entirely permitted by our data. After all, as long as we're classifying all the data correctly (and we are), it doesn't really matter what happens elsewhere (though we probably prefer simplicity when we can get it, just on general principles [Domingos12]).

Since the dots only give an impression of the boundary, let's look at it in high resolution in Figure B1-34. Just to see what things look like away from the data, we'll also zoom out and look at the boundary curve with a larger range of values.

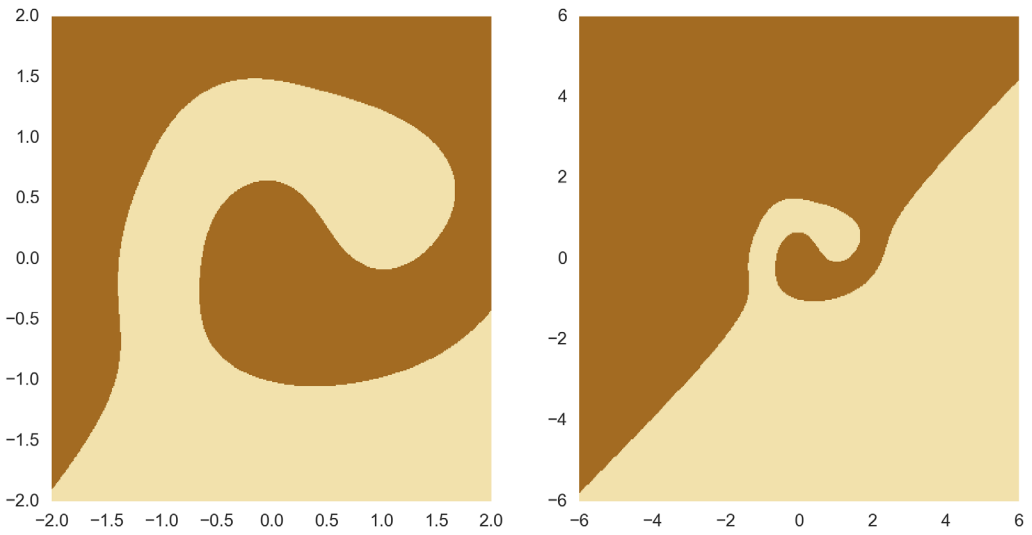



*Figure B1-34: Looking at the boundary curves from Figures B1-32 and B1-33. Left: Close up to the data of the two moons, but with a slightly larger range than Figure B1-33. Right: A larger plotting area.*

A graph of the scores for the different combinations is shown in Figure B1-35. The best results came from setting `degree` to 5, and `alpha` to 0.25.

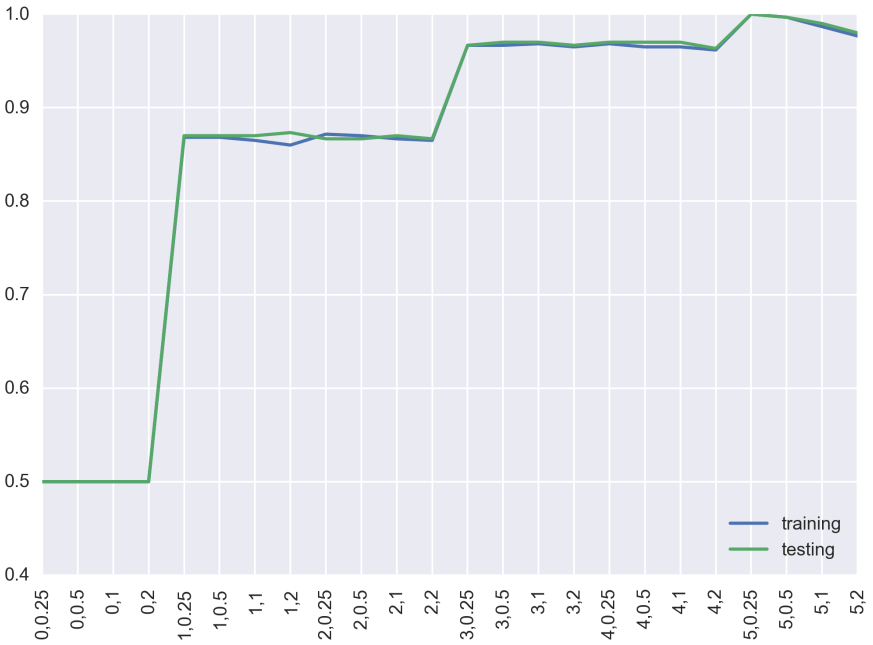



*Figure B1-35: Scores from our pipeline search. The straight-line version is shown at the far left, for degree values of 0.*

The best combination of `alpha` at 0.25 and `degree` at 5, located near the right end of the graph, produced a perfect training and testing score of 1.0. As we might expect, the regularization parameter didn't have much effect until the curve got complicated enough to start overfitting, and even then on this simple example it didn't make much of a difference.

## Looking at the Decision Boundary

Let's take another look at the decision boundaries we've found for our half-moon data.

In Figures B1-25 and B1-26 we found a linear boundary. In those figures, we drew each dot with the color of the class returned by `predict()`. But as we mentioned before, we can get the relative confidences of the classifier from `decision_function()` and `predict_proba()`. So instead of getting just a single integer telling us which class is assigned to each input, we get back a confidence value for each category. Since we have only 2 classes, this means we'll get two floating-point values, one for each category.

Let's plot the values of `decision_function()` for our straight-line fit in 3D. The left diagram in Figure B1-36 shows the result.

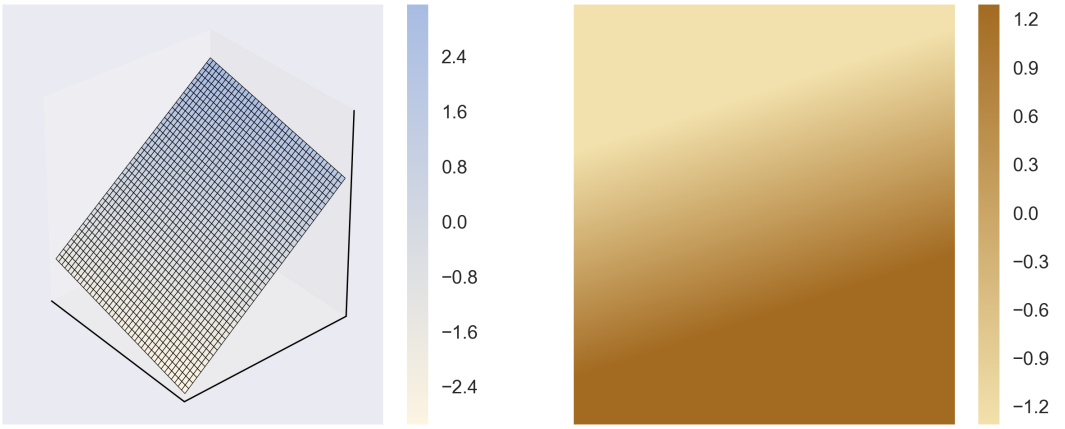



*Figure B1-36: Looking at the confidence values for the linear boundary between our two half-moons, as shown in Figures B1-25 and B1-26. Left: The output of `decision_function()`. Right: A top-down view of the 3D plot.*

When we look at the confidence of each category, rather than asking for just the most likely one, we can see that there's a transition from one to the other. As we might expect, the boundary from one class to the other isn't instantaneous. This surface is really a single flat plane with no creases.

Looking down on this plot, as in the right of Figure B1-36, we can see a crossover zone where the probabilities smoothly change from 1 to 0 or vice-versa.

Let's look at the values from `decision_function()` for our curved boundary in Figure B1-33 in the same way. We'll plot these in 3D and then look down on the plot, as in Figure B1-37. The soft regions of changing color show where the confidence values are changing.

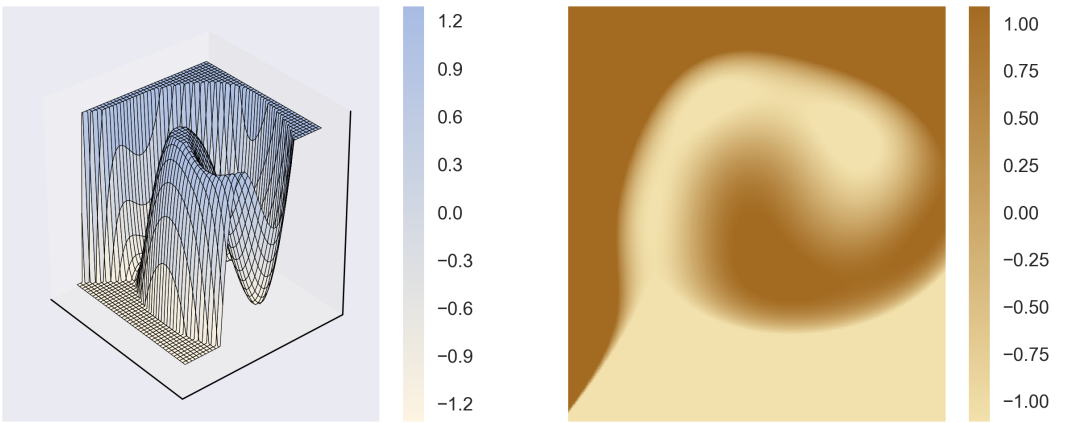



*Figure B1-37: The curved boundary in 3D, and viewed from the top. The flat regions are because we've clamped the values to [–1, 1] to better see the structure near the middle. Left: A 3D view. Notice how smooth the surfaces are. Right: A top-down view of the 3D plot.*

We can see that there's a smooth transition zone between the two categories (we manually clamped the output of `decision_function()` to [–1, 1] so we could focus on what's happening in that range). In the cross-over regions, the probability of a point being in one class or the other varies smoothly.

In some circumstances, getting the most likely class from `predict()` is just what we want. In others, getting the more refined confidences from `decision_function()` (or `predict_proba()`) can be more useful.

## Applying Pipelined Transformations

We promised to return to a gap in Figure B1-31, where we showed the fold going through a transformation, but not where the transformation came from. Let's address that now. This will also pay off our earlier promise to show how to use pipelines to correctly apply a transformation while doing cross-validation.

Let's continue with our previous example of using `PolynomialFeatures` followed by `RidgeClassifier`. The first step in our pipeline adds new polynomials to each sample, so it counts as a *transformation* of our training data, since it changes the samples that go into the classifier. In this example, `PolynomialFeatures` is including new features, rather than changing the features themselves like a scaling transform would. But the samples are changing, so we call it a transformation.

Remember our cardinal rule about transformations: anything we do to the samples in the training data must also be done to all other data.

Since the transformation of our training data (adding new features) is happening inside the pipeline, we have to pull that transformation out so we can apply it to the validation fold. Figure B1-38 shows a close-up of the pipeline in the cross-validation step, where step 1 (our `PolynomialFeatures` object) computes a transformation, and then that transformed data is applied to both the training data used by step 2, and the validation fold used to evaluate the result.

There's no leaked information here, because we're doing everything by the book. For each model, we extract a fold, compute the transformation on the remaining training data, and then apply that transformation to both the training data and the data in the fold.

This application of the transformation is carried out for us automatically by both the regular and random grid search objects, so we don't have to lift a finger, or change our code from the previous section in any way. In other words, scikit-learn applies the steps in the pipeline in just the right way, automatically.

We discussed this detail because it's a great illustration of how we have to always think about information leakage any time we touch our sample data. It also shows us a nice way to handle this issue. And finally, this example shows how our libraries make our lives easier by handling these issues for us.

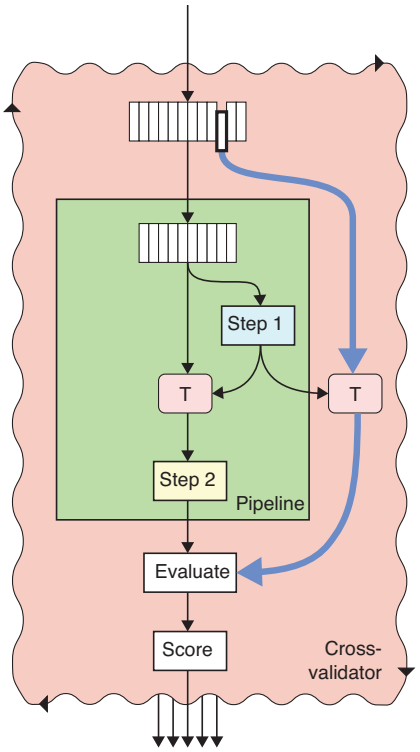



*Figure B1-38: A close-up of the cross-validation step in Figure B1-31, filling in the missing source of the transformation to the fold.*

We can use multiple transformations in our pipeline. For instance, we might add a `MinMaxScaler` after the `PolynomialFeatures` object. We might also then include a PCA object to reduce the dimensionality of the data. We can have as many of these data-transforming steps as we like, and they will all get applied in sequence to both the training and fold data.

## Datasets

*See the Jupyter notebook* Bonus01-Scikit-Learn-8-Datasets.ipynb.

Scikit-learn provides a variety of datasets that are clean and ready for immediate use. Some of these are *real-world* data representing actual field studies, and some are *synthetic* data that is generated procedurally when we ask for it.

For example, we can easily import the famous *Iris* dataset, which describes the lengths of different petals of several species of iris flower. This is a classic database that's used in many discussions of categorization.

Scikit-learn also includes the *Boston housing* dataset that records the prices of homes in the Boston area for a period of years, along with other geographical information. This is often used in discussions of regression.

There are other famous datasets as well. For example, the *20newsgroups* dataset provides text data from online discussions, which is frequently used for text-based learners. The *digits* dataset contains small grayscale images of handwritten digits from 0 to 9. And the *Labeled Faces in the Wild*, or *LFW*, dataset provides labeled photographs of people that are useful for face detection and classifying images.

Most of these datasets are returned in NumPy arrays that are ready for immediate use, but always check the documentation to learn of any idiosyncrasies or exceptions.

To load a dataset, we usually need only to call its loader and save that routine's output in a variable. For example, we can load the Boston data as in Listing B1-35.

```
from sklearn import datasets

house_data = datasets.load_boston()
```

*Listing B1-35: Loading the Boston dataset*

The arguments for synthetic datasets are important because they help us control the size and shape of the data that's made.

Some of the most popular synthetic dataset creators are `make_moons()` which generates the two interlocking arcs we used earlier, `make_circles()` which creates a pair of nested circles, and `make_blobs()` which makes sets of points drawn from Gaussian distributions.

As an example, Listing B1-36 shows how we call `make_moons()`. We tell it how many points to make, and we can optionally add a `noise` parameter to break up the uniformity.

```
from sklearn.datasets import make_moons

(moons_xy, moons_labels) = make_moons(n_samples=800, noise=.08)
```

*Listing B1-36: Using the `make_moons()` routine to generate 800 points of synthetic data.*

Figure B1-39 shows these three types of synthetic data, using their defaults and with noise added. Note that each routine provides labels along with the point locations, so the data is ready for classification.

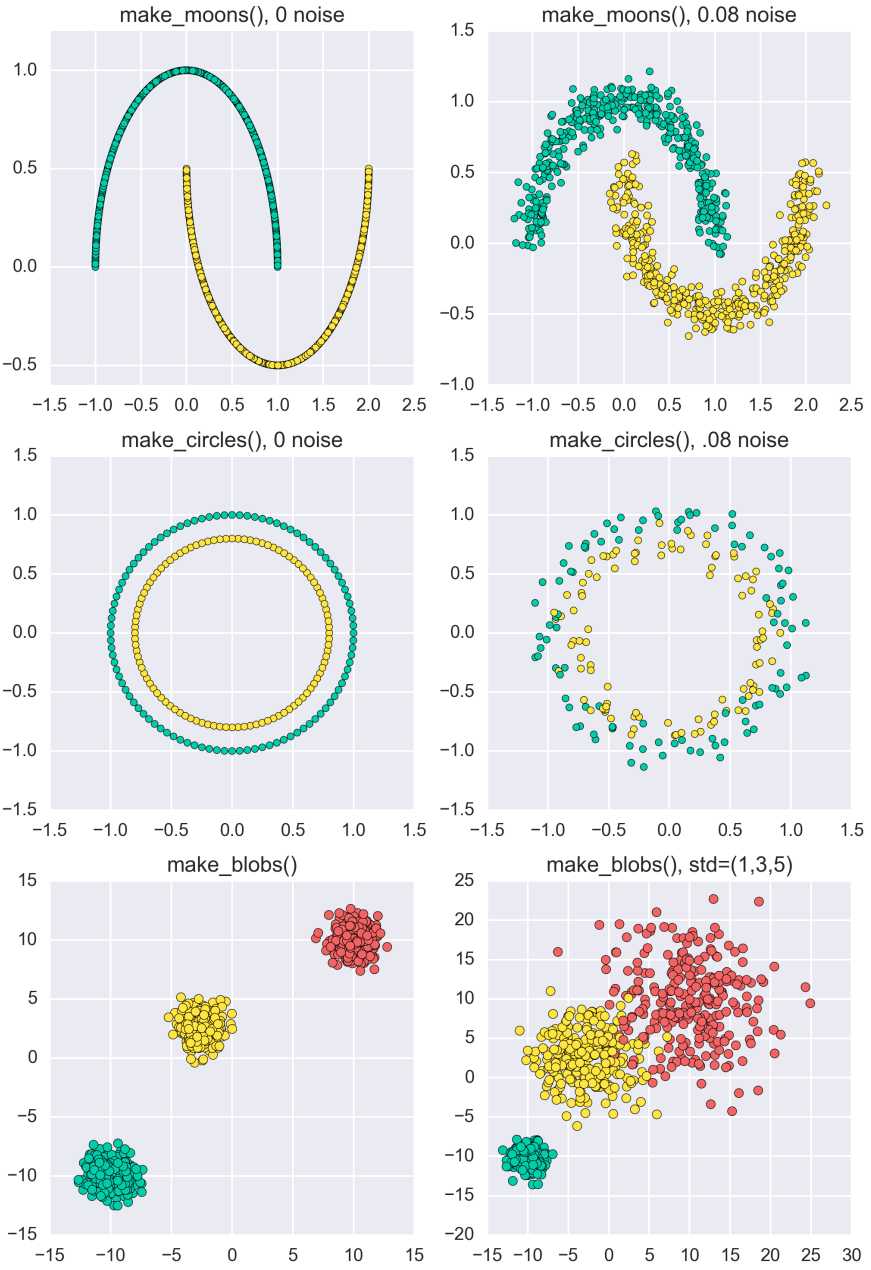



*Figure B1-39: Synthetic datasets provided by scikit-learn. Top row: `make_moons` with 800 points, with no noise and with the `noise` parameter set to 0.08. Middle row: `make_circles` with 200 points, with no noise and with the `noise` parameter set to 0.08. Bottom row: `make_blobs()` with 800 points and 3 blobs, with the default standard deviation (all 1's) and with larger standard deviations to spread out the points more.*

## Utilities

*See the Jupyter notebook* Bonus01-Scikit-Learn-Utilities.ipynb.

Like any library, scikit-learn has its share of utility functions that we can collect together into a kind of grab-bag, or miscellaneous, category.

Perhaps one of the most popular of these is used to split a database into different pieces. The routine `train_test_split()` does just as it says: given a database, it splits it into two pieces that are typically used as a training set and a test (or validation) set. Among other useful arguments, `test_size` is a value from 0 to 1 that specifies the percentage of the database to place into the test set. By default, this has a value of 0.25. Listing B1-37 shows how this works with the nested-circles dataset.

```
from sklearn.model_selection import train_test_split

(circle_xy, circle_labels) = make_circles(n_samples=200, noise=.08)

samples_train, samples_test, labels_train, labels_test = \
    train_test_split(circle_xy, circle_labels, test_size=0.25)
```

*Listing B1-37: Using `train_test_split()` to break up a dataset of nested circles into training and testing sets.*

This produces the datasets shown in Figure B1-40.

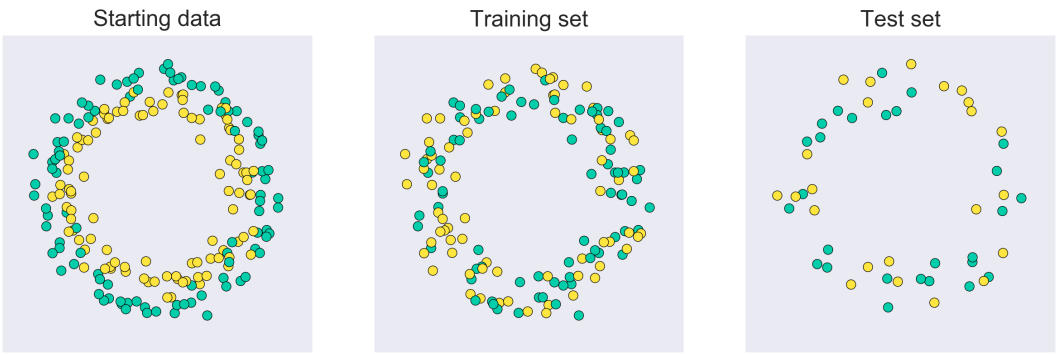



*Figure B1-40: Splitting up our original data into a training and testing set with `train_test_split()`. The two sets don't have any samples in common. Left: The starting data of 200 points. Middle: The training set of 150 points. Right: The test set of 50 points.*

We've split up our starting data into two new sets. Notice that the correct labels have come along with their samples, as we'd like. What's not shown in the figure is that the order of the samples has been shuffled. That is, their order in the training and testing sets are not the same order as in the starting data.

To see why the algorithm takes this step, suppose our circle samples had been generated by starting at 3 o'clock and working clockwise. Then the first 75% of the data would be the samples from 3 o'clock to 12 o'clock, and last 25% would be from 12 to 3, as in Figure B1-41. The test data is

nothing like the training data. This would make a terrible training-test split! For this figure, we generated our own data as described above, starting at 12 o'clock and working clockwise, rather than using `make_circles()`.


*Figure B1-41: We generated this data starting at 3 o'clock and then proceeded clockwise. Left: The original data of 200 points. Middle: The first 150 points for the training set. Right: The final 50 points for the test set.*

We wouldn't have this problem with `make_circles()`, because it generates its samples more randomly, but other programs might not be so careful, and data we get from other sources might have been put into some kind of order before it came to us. To reduce the chance of such datasets producing bad training-test splits like Figure B1-41, `train_test_split()` shuffles the samples before assigning them to the train and test sets. The hope is that this will cause each set to be representative of the totality of the starting data, as it is in Figure B1-40.

## Wrapping Up

As we promised at the start, this chapter has barely hinted at the huge variety of objects and functions offered by scikit-learn.

The scikit-learn online documentation is free and always available, but it's typically aimed at the working programmer who already understands the concepts and just needs to be reminded of syntax or argument names. There are some explanatory and tutorial articles, and some examples of use, but they, too, are often terse. For these reasons, the library's documentation is probably best used for reference, and not for explanations, though the FAQ can sometimes help clear up a question.

An easier way to dig deeper into the mechanics of this versatile library is the book by Müller and Guido [Müller-Guido16]. The book assumes familiarity with machine-learning algorithms, though there is some introductory and review material.

For help understanding a particular algorithm, or its implementations, great explanations in text or video are often just an internet search away.

A much more in-depth approach, with plenty of mathematical details, is offered in the book by Raschka [Raschka15].

## References


[Domingos12]: Pedro Domingos, "A Few Useful Things to Know About Machine Learning," *Communications of the ACM*, Volume 55 Issue 10, October 2012. *https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf*

[Github14]: GitHub contributors, 2014. *https://github.com/mwaskom/seaborn/issues/229/*

[Jupyter17]: The Jupyter authors, "Jupyter," 2017. *http://jupyter.org/*

[Kassambara17]: Alboukadel Kassambara, "Determining the Optimal Number of Clusters: 3 Must Know Methods," STHDA blog [Statistical Tools for High-Throughput Data Analysis], 2017. *http://www.sthda.com/english/articles/29-cluster-validation-essentials/96-determining-the-optimal-number-of-clusters-3-must-know-methods/*

[Müller-Guido16]: Andreas C. Müller and Sarah Guido, "Introduction to Machine Learning with Python," O'Reilly Media, 2016.

[Raschka15]: Sebastian Raschka, "Python Machine Learning," Packt Publishing, 2015.

[scikit-learn20]: Scikit-learn authors, "Machine Learning in Python," version 0.24. *https://scikit-learn.org/stable/index.html*

[VanderPlas16]: Jake VanderPlas, "Python Data Science Handbook," O'Reilly Media, 2016.

[Waskom17]: Michael Waskom, "seaborn: statistical data visualization," Seaborn website, 2017. *https://seaborn.pydata.org/*