Using binning is one way to expand a continuous feature. Another one is to use *polynomials* of the original features. For a given feature `x`, we might want to consider `x ** 2`, `x ** 3`, `x ** 4`, and so on. This is implemented in `PolynomialFeatures` in the `preprocessing` module:

In[21]:

```
from sklearn.preprocessing import PolynomialFeatures

# include polynomials up to x ** 10:
# the default "include_bias=True" adds a feature that's constantly 1
poly = PolynomialFeatures(degree=10, include_bias=False)
poly.fit(X)
X_poly = poly.transform(X)
```

Using a degree of 10 yields 10 features:

In[22]:

```
print("X_poly.shape: {}".format(X_poly.shape))
```

Out[22]:

```
X_poly.shape: (100, 10)
```

Let's compare the entries of `X_poly` to those of `X`:

In[23]:

```
print("Entries of X:\n{}".format(X[:5]))
print("Entries of X_poly:\n{}".format(X_poly[:5]))
```

Out[23]:

```
Entries of X:
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]
Entries of X_poly:
[[    -0.753      0.567     -0.427      0.321     -0.242      0.182
      -0.137      0.103     -0.078      0.058]
 [     2.704      7.313     19.777     53.482    144.632    391.125
    1057.714   2860.360   7735.232  20918.278]
 [     1.392      1.938      2.697      3.754      5.226      7.274
      10.125     14.094     19.618     27.307]
 [     0.592      0.350      0.207      0.123      0.073      0.043
       0.025      0.015      0.009      0.005]
 [    -2.064      4.260     -8.791     18.144    -37.448     77.289
    -159.516    329.222   -679.478   1402.367]]
```

You can obtain the semantics of the features by calling the `get_feature_names` method, which provides the exponent for each feature:

In[24]:

```
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

Out[24]:

```
Polynomial feature names:
['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9', 'x0^10']
```

You can see that the first column of `X_poly` corresponds exactly to `X`, while the other columns are the powers of the first entry. It's interesting to see how large some of the values can get. The second column has entries above 20,000, orders of magnitude different from the rest.

Using polynomial features together with a linear regression model yields the classical model of *polynomial regression* (see Figure 4-5):

In[26]:

```
reg = LinearRegression().fit(X_poly, y)

line_poly = poly.transform(line)
plt.plot(line, reg.predict(line_poly), label='polynomial linear regression')
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

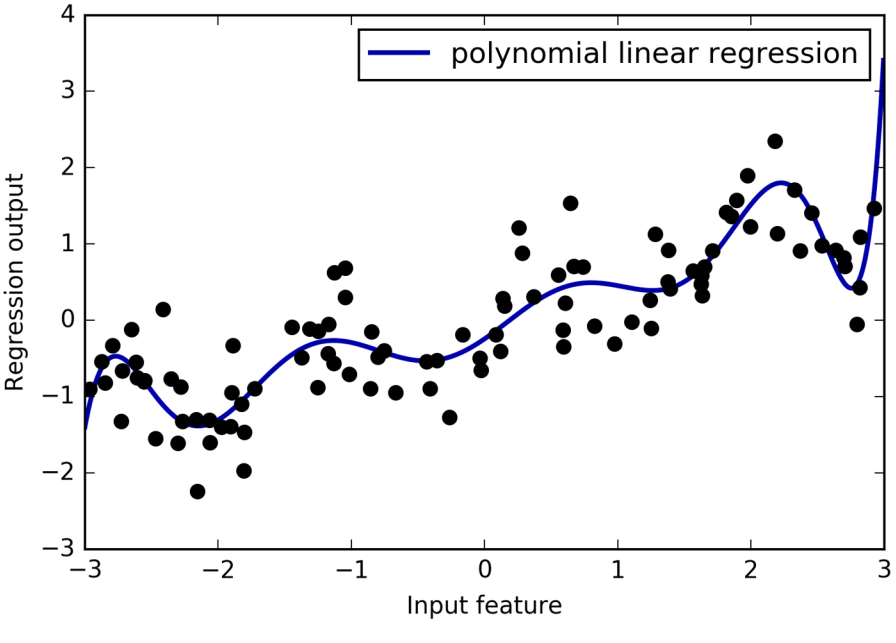

Figure 4-5. Linear regression with tenth-degree polynomial features

As you can see, polynomial features yield a very smooth fit on this one-dimensional data. However, polynomials of high degree tend to behave in extreme ways on the boundaries or in regions with little data.

As a comparison, here is a kernel SVM model learned on the original data, without any transformation (see Figure 4-6):

In[26]:

```
from sklearn.svm import SVR

for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line), label='SVR gamma={}'.format(gamma))

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

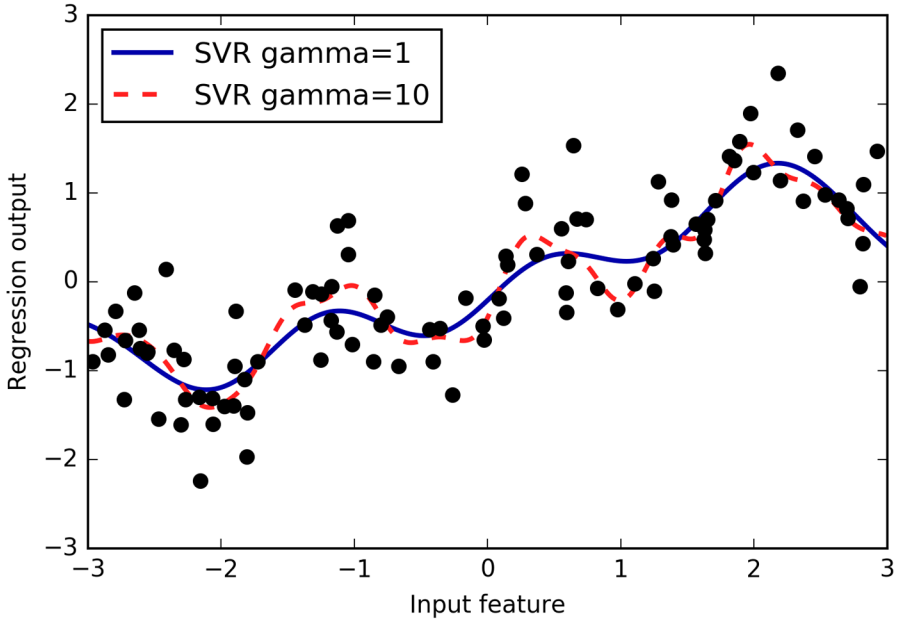

Figure 4-6. Comparison of different gamma parameters for an SVM with RBF kernel

Using a more complex model, a kernel SVM, we are able to learn a similarly complex prediction to the polynomial regression without an explicit transformation of the features.

As a more realistic application of interactions and polynomials, let's look again at the Boston Housing dataset. We already used polynomial features on this dataset in Chapter 2. Now let's have a look at how these features were constructed, and at how much the polynomial features help. First we load the data, and rescale it to be between 0 and 1 using `MinMaxScaler`:

In[27]:

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split
    (boston.data, boston.target, random_state=0)

# rescale data
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Now, we extract polynomial features and interactions up to a degree of 2:

In[28]:

```
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_poly.shape: {}".format(X_train_poly.shape))
```

Out[28]:

```
X_train.shape: (379, 13)
X_train_poly.shape: (379, 105)
```

The data originally had 13 features, which were expanded into 105 interaction features. These new features represent all possible interactions between two different original features, as well as the square of each original feature. `degree=2` here means that we look at all features that are the product of up to two original features. The exact correspondence between input and output features can be found using the `get_feature_names` method:

In[29]:

```
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

Out[29]:

```
Polynomial feature names:
['1', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10',
'x11', 'x12', 'x0^2', 'x0 x1', 'x0 x2', 'x0 x3', 'x0 x4', 'x0 x5', 'x0 x6',
'x0 x7', 'x0 x8', 'x0 x9', 'x0 x10', 'x0 x11', 'x0 x12', 'x1^2', 'x1 x2',
```

```
'x1 x3', 'x1 x4', 'x1 x5', 'x1 x6', 'x1 x7', 'x1 x8', 'x1 x9', 'x1 x10',
'x1 x11', 'x1 x12', 'x2^2', 'x2 x3', 'x2 x4', 'x2 x5', 'x2 x6', 'x2 x7',
'x2 x8', 'x2 x9', 'x2 x10', 'x2 x11', 'x2 x12', 'x3^2', 'x3 x4', 'x3 x5',
'x3 x6', 'x3 x7', 'x3 x8', 'x3 x9', 'x3 x10', 'x3 x11', 'x3 x12', 'x4^2',
'x4 x5', 'x4 x6', 'x4 x7', 'x4 x8', 'x4 x9', 'x4 x10', 'x4 x11', 'x4 x12',
'x5^2', 'x5 x6', 'x5 x7', 'x5 x8', 'x5 x9', 'x5 x10', 'x5 x11', 'x5 x12',
'x6^2', 'x6 x7', 'x6 x8', 'x6 x9', 'x6 x10', 'x6 x11', 'x6 x12', 'x7^2',
'x7 x8', 'x7 x9', 'x7 x10', 'x7 x11', 'x7 x12', 'x8^2', 'x8 x9', 'x8 x10',
'x8 x11', 'x8 x12', 'x9^2', 'x9 x10', 'x9 x11', 'x9 x12', 'x10^2', 'x10 x11',
'x10 x12', 'x11^2', 'x11 x12', 'x12^2']
```

The first new feature is a constant feature, called `"1"` here. The next 13 features are the original features (called `"x0"` to `"x12"`). Then follows the first feature squared (`"x0^2"`) and combinations of the first and the other features.

Let's compare the performance using `Ridge` on the data with and without interactions:

In[30]:

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(
    ridge.score(X_test_scaled, y_test)))
ridge = Ridge().fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(
    ridge.score(X_test_poly, y_test)))
```

Out[30]:

```
Score without interactions: 0.621
Score with interactions: 0.753
```

Clearly, the interactions and polynomial features gave us a good boost in performance when using `Ridge`. When using a more complex model like a random forest, the story is a bit different, though:

In[31]:

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100).fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(
    rf.score(X_test_scaled, y_test)))
rf = RandomForestRegressor(n_estimators=100).fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(rf.score(X_test_poly, y_test)))
```

Out[31]:

```
Score without interactions: 0.799
Score with interactions: 0.763
```

You can see that even without additional features, the random forest beats the performance of `Ridge`. Adding interactions and polynomials actually decreases performance slightly.

Univariate Nonlinear Transformations

We just saw that adding squared or cubed features can help linear models for regression. There are other transformations that often prove useful for transforming certain features: in particular, applying mathematical functions like `log`, `exp`, or `sin`. While tree-based models only care about the ordering of the features, linear models and neural networks are very tied to the scale and distribution of each feature, and if there is a nonlinear relation between the feature and the target, that becomes hard to model—particularly in regression. The functions `log` and `exp` can help by adjusting the relative scales in the data so that they can be captured better by a linear model or neural network. We saw an application of that in Chapter 2 with the memory price data. The `sin` and `cos` functions can come in handy when dealing with data that encodes periodic patterns.

Most models work best when each feature (and in regression also the target) is loosely Gaussian distributed—that is, a histogram of each feature should have something resembling the familiar "bell curve" shape. Using transformations like `log` and `exp` is a hacky but simple and efficient way to achieve this. A particularly common case when such a transformation can be helpful is when dealing with integer count data. By count data, we mean features like "how often did user A log in?" Counts are never negative, and often follow particular statistical patterns. We are using a synthetic dataset of counts here that has properties similar to those you can find in the wild. The features are all integer-valued, while the response is continuous:

`In[32]:`

```
rnd = np.random.RandomState(0)
X_org = rnd.normal(size=(1000, 3))
w = rnd.normal(size=3)

X = rnd.poisson(10 * np.exp(X_org))
y = np.dot(X_org, w)
```

Let's look at the first 10 entries of the first feature. All are integer values and positive, but apart from that it's hard to make out a particular pattern.

If we count the appearance of each value, the distribution of values becomes clearer:

In[33]:

```
print("Number of feature appearances:\n{}".format(np.bincount(X[:, 0])))
```

Out[33]:

```
Number of feature appearances:
[28 38 68 48 61 59 45 56 37 40 35 34 36 26 23 26 27 21 23 23 18 21 10  9 17
  9  7 14 12  7  3  8  4  5  5  3  4  2  4  1  1  3  2  5  3  8  2  5  2  1
  2  3  3  2  2  3  3  0  1  2  1  0  0  3  1  0  0  0  1  3  0  1  0  2  0
  1  1  0  0  0  0  1  0  0  2  2  0  1  1  0  0  0  0  1  1  0  0  0  0  0
  0  0  1  0  0  0  0  0  1  1  0  0  1  0  0  0  0  0  0  1  0  0  0  0
  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1]
```

The value 2 seems to be the most common, with 62 appearances (`bincount` always starts at 0), and the counts for higher values fall quickly. However, there are some very high values, like 134 appearing twice. We visualize the counts in Figure 4-7:

In[34]:

```
bins = np.bincount(X[:, 0])
plt.bar(range(len(bins)), bins, color='w')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```

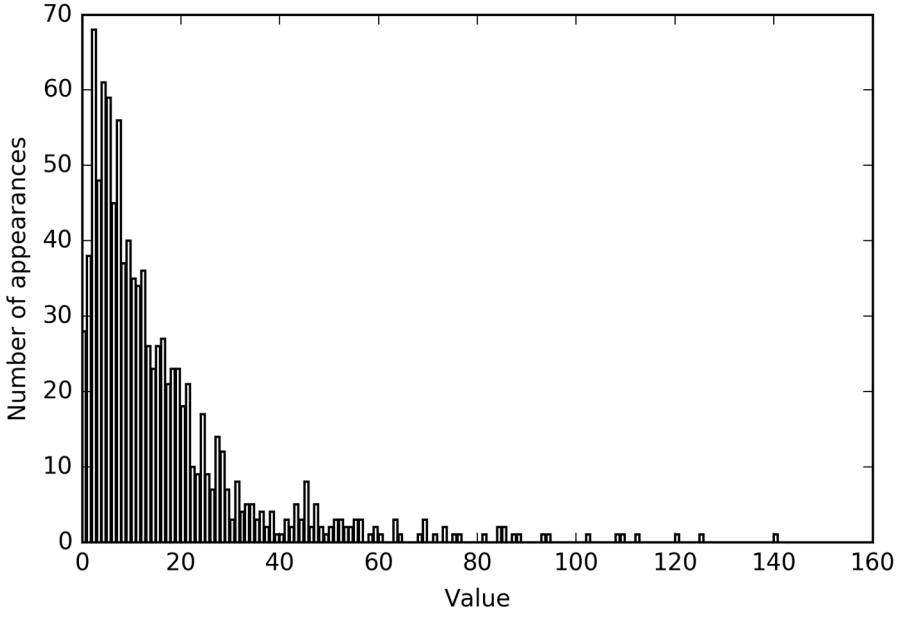

Figure 4-7. Histogram of feature values for X[0]

Features `X[:, 1]` and `X[:, 2]` have similar properties. This kind of distribution of values (many small ones and a few very large ones) is very common in practice.[1] However, it is something most linear models can't handle very well. Let's try to fit a ridge regression to this model:

In[35]:

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
score = Ridge().fit(X_train, y_train).score(X_test, y_test)
print("Test score: {:.3f}".format(score))
```

Out[35]:

```
Test score: 0.622
```

As you can see from the relatively low R^2 score, `Ridge` was not able to really capture the relationship between `X` and `y`. Applying a logarithmic transformation can help, though. Because the value 0 appears in the data (and the logarithm is not defined at 0), we can't actually just apply `log`, but we have to compute `log(X + 1)`:

In[36]:

```
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

After the transformation, the distribution of the data is less asymmetrical and doesn't have very large outliers anymore (see Figure 4-8):

In[37]:

```
plt.hist(np.log(X_train_log[:, 0] + 1), bins=25, color='gray')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```

1 This is a Poisson distribution, which is quite fundamental to count data.

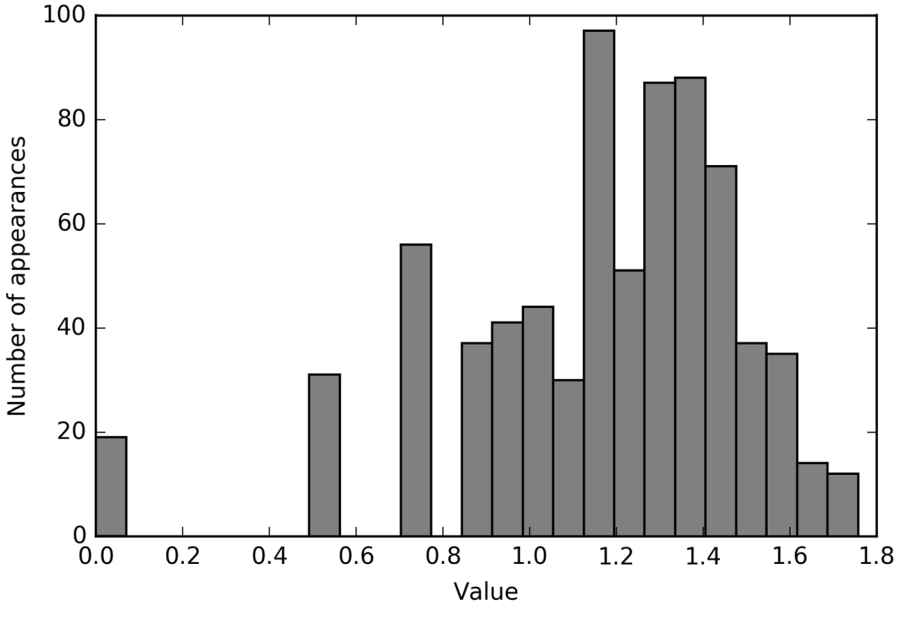

Figure 4-8. Histogram of feature values for X[0] after logarithmic transformation

Building a ridge model on the new data provides a much better fit:

In[38]:

```
score = Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
print("Test score: {:.3f}".format(score))
```

Out[38]:

```
Test score: 0.875
```

Finding the transformation that works best for each combination of dataset and model is somewhat of an art. In this example, all the features had the same properties. This is rarely the case in practice, and usually only a subset of the features should be transformed, or sometimes each feature needs to be transformed in a different way. As we mentioned earlier, these kinds of transformations are irrelevant for tree-based models but might be essential for linear models. Sometimes it is also a good idea to transform the target variable `y` in regression. Trying to predict counts (say, number of orders) is a fairly common task, and using the `log(y + 1)` transformation often helps.[2]

2 This is a very crude approximation of using Poisson regression, which would be the proper solution from a probabilistic standpoint.

As you saw in the previous examples, binning, polynomials, and interactions can have a huge influence on how models perform on a given dataset. This is particularly true for less complex models like linear models and naive Bayes models. Tree-based models, on the other hand, are often able to discover important interactions themselves, and don't require transforming the data explicitly most of the time. Other models, like SVMs, nearest neighbors, and neural networks, might sometimes benefit from using binning, interactions, or polynomials, but the implications there are usually much less clear than in the case of linear models.

Automatic Feature Selection

With so many ways to create new features, you might get tempted to increase the dimensionality of the data way beyond the number of original features. However, adding more features makes all models more complex, and so increases the chance of overfitting. When adding new features, or with high-dimensional datasets in general, it can be a good idea to reduce the number of features to only the most useful ones, and discard the rest. This can lead to simpler models that generalize better. But how can you know how good each feature is? There are three basic strategies: *univariate statistics*, *model-based selection*, and *iterative selection*. We will discuss all three of them in detail. All of these methods are supervised methods, meaning they need the target for fitting the model. This means we need to split the data into training and test sets, and fit the feature selection only on the training part of the data.

Univariate Statistics

In univariate statistics, we compute whether there is a statistically significant relationship between each feature and the target. Then the features that are related with the highest confidence are selected. In the case of classification, this is also known as *analysis of variance* (ANOVA). A key property of these tests is that they are *univariate*, meaning that they only consider each feature individually. Consequently, a feature will be discarded if it is only informative when combined with another feature. Univariate tests are often very fast to compute, and don't require building a model. On the other hand, they are completely independent of the model that you might want to apply after the feature selection.

To use univariate feature selection in `scikit-learn`, you need to choose a test, usually either `f_classif` (the default) for classification or `f_regression` for regression, and a method to discard features based on the *p*-values determined in the test. All methods for discarding parameters use a threshold to discard all features with too high a *p*-value (which means they are unlikely to be related to the target). The methods differ in how they compute this threshold, with the simplest ones being `SelectKBest`, which selects a fixed number `k` of features, and `SelectPercentile`, which selects a fixed percentage of features. Let's apply the feature selection for classification on the

`cancer` dataset. To make the task a bit harder, we'll add some noninformative noise features to the data. We expect the feature selection to be able to identify the features that are noninformative and remove them:

In[39]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()

# get deterministic random numbers
rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
# add noise features to the data
# the first 30 features are from the dataset, the next 50 are noise
X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(
    X_w_noise, cancer.target, random_state=0, test_size=.5)
# use f_classif (the default) and SelectPercentile to select 50% of features
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
# transform training set
X_train_selected = select.transform(X_train)

print("X_train.shape: {}".format(X_train.shape))
print("X_train_selected.shape: {}".format(X_train_selected.shape))
```

Out[39]:

```
X_train.shape: (284, 80)
X_train_selected.shape: (284, 40)
```

As you can see, the number of features was reduced from 80 to 40 (50 percent of the original number of features). We can find out which features have been selected using the `get_support` method, which returns a Boolean mask of the selected features (visualized in Figure 4-9):

In[40]:

```
mask = select.get_support()
print(mask)
# visualize the mask -- black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

Out[40]:

```
[ True  True  True  True  True  True  True  True  True False  True False
  True  True  True  True  True  True False False  True  True  True  True
  True  True  True  True  True  True False False False  True False  True
```

```
 False False  True False False False False  True False False  True False
 False  True False  True False False False False False False  True False
  True False False False False  True False  True False False False False
  True  True False  True False False False False]
```

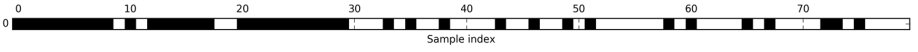

Figure 4-9. Features selected by `SelectPercentile`

As you can see from the visualization of the mask, most of the selected features are the original features, and most of the noise features were removed. However, the recovery of the original features is not perfect. Let's compare the performance of logistic regression on all features against the performance using only the selected features:

`In[41]:`

```
from sklearn.linear_model import LogisticRegression

# transform test data
X_test_selected = select.transform(X_test)

lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Score with all features: {:.3f}".format(lr.score(X_test, y_test)))
lr.fit(X_train_selected, y_train)
print("Score with only selected features: {:.3f}".format(
    lr.score(X_test_selected, y_test)))
```

`Out[41]:`

```
Score with all features: 0.930
Score with only selected features: 0.940
```

In this case, removing the noise features improved performance, even though some of the original features were lost. This was a very simple synthetic example, and outcomes on real data are usually mixed. Univariate feature selection can still be very helpful, though, if there is such a large number of features that building a model on them is infeasible, or if you suspect that many features are completely uninformative.

Model-Based Feature Selection

Model-based feature selection uses a supervised machine learning model to judge the importance of each feature, and keeps only the most important ones. The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling. The feature selection model needs to provide some measure of importance for each feature, so that they can be ranked by this measure. Decision trees and decision tree–based models provide a `feature_importances_`

attribute, which directly encodes the importance of each feature. Linear models have coefficients, which can also be used to capture feature importances by considering the absolute values. As we saw in Chapter 2, linear models with L1 penalty learn sparse coefficients, which only use a small subset of features. This can be viewed as a form of feature selection for the model itself, but can also be used as a preprocessing step to select features for another model. In contrast to univariate selection, model-based selection considers all features at once, and so can capture interactions (if the model can capture them). To use model-based feature selection, we need to use the `SelectFromModel` transformer:

In[42]:

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold="median")
```

The `SelectFromModel` class selects all features that have an importance measure of the feature (as provided by the supervised model) greater than the provided threshold. To get a comparable result to what we got with univariate feature selection, we used the median as a threshold, so that half of the features will be selected. We use a random forest classifier with 100 trees to compute the feature importances. This is a quite complex model and much more powerful than using univariate tests. Now let's actually fit the model:

In[43]:

```
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_l1.shape: {}".format(X_train_l1.shape))
```

Out[43]:

```
X_train.shape: (284, 80)
X_train_l1.shape: (284, 40)
```

Again, we can have a look at the features that were selected (Figure 4-10):

In[44]:

```
mask = select.get_support()
# visualize the mask -- black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

Figure 4-10. Features selected by SelectFromModel using the RandomForestClassifier

This time, all but two of the original features were selected. Because we specified to select 40 features, some of the noise features are also selected. Let's take a look at the performance:

In[45]:

```
X_test_l1 = select.transform(X_test)
score = LogisticRegression().fit(X_train_l1, y_train).score(X_test_l1, y_test)
print("Test score: {:.3f}".format(score))
```

Out[45]:

```
Test score: 0.951
```

With the better feature selection, we also gained some improvements here.

Iterative Feature Selection

In univariate testing we used no model, while in model-based selection we used a single model to select features. In iterative feature selection, a series of models are built, with varying numbers of features. There are two basic methods: starting with no features and adding features one by one until some stopping criterion is reached, or starting with all features and removing features one by one until some stopping criterion is reached. Because a series of models are built, these methods are much more computationally expensive than the methods we discussed previously. One particular method of this kind is *recursive feature elimination* (RFE), which starts with all features, builds a model, and discards the least important feature according to the model. Then a new model is built using all but the discarded feature, and so on until only a prespecified number of features are left. For this to work, the model used for selection needs to provide some way to determine feature importance, as was the case for the model-based selection. Here, we use the same random forest model that we used earlier, and get the results shown in Figure 4-11:

In[46]:

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
             n_features_to_select=40)

select.fit(X_train, y_train)
# visualize the selected features:
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

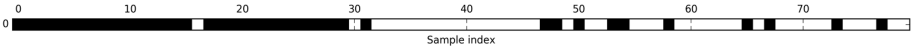

Figure 4-11. Features selected by recursive feature elimination with the random forest classifier model

The feature selection got better compared to the univariate and model-based selection, but one feature was still missed. Running this code also takes significantly longer than that for the model-based selection, because a random forest model is trained 40 times, once for each feature that is dropped. Let's test the accuracy of the logistic regression model when using RFE for feature selection:

In[47]:

```
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)

score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("Test score: {:.3f}".format(score))
```

Out[47]:

```
Test score: 0.951
```

We can also use the model used inside the RFE to make predictions. This uses only the feature set that was selected:

In[48]:

```
print("Test score: {:.3f}".format(select.score(X_test, y_test)))
```

Out[48]:

```
Test score: 0.951
```

Here, the performance of the random forest used inside the RFE is the same as that achieved by training a logistic regression model on top of the selected features. In other words, once we've selected the right features, the linear model performs as well as the random forest.

If you are unsure when selecting what to use as input to your machine learning algorithms, automatic feature selection can be quite helpful. It is also great for reducing the amount of features needed—for example, to speed up prediction or to allow for more interpretable models. In most real-world cases, applying feature selection is unlikely to provide large gains in performance. However, it is still a valuable tool in the toolbox of the feature engineer.

Utilizing Expert Knowledge

Feature engineering is often an important place to use *expert knowledge* for a particular application. While the purpose of machine learning in many cases is to avoid having to create a set of expert-designed rules, that doesn't mean that prior knowledge of the application or domain should be discarded. Often, domain experts can help in identifying useful features that are much more informative than the initial representation of the data. Imagine you work for a travel agency and want to predict flight prices. Let's say you have a record of prices together with dates, airlines, start locations, and destinations. A machine learning model might be able to build a decent model from that. Some important factors in flight prices, however, cannot be learned. For example, flights are usually more expensive during peak vacation months and around holidays. While the dates of some holidays (like Christmas) are fixed, and their effect can therefore be learned from the date, others might depend on the phases of the moon (like Hanukkah and Easter) or be set by authorities (like school holidays). These events cannot be learned from the data if each flight is only recorded using the (Gregorian) date. However, it is easy to add a feature that encodes whether a flight was on, preceding, or following a public or school holiday. In this way, prior knowledge about the nature of the task can be encoded in the features to aid a machine learning algorithm. Adding a feature does not force a machine learning algorithm to use it, and even if the holiday information turns out to be noninformative for flight prices, augmenting the data with this information doesn't hurt.

We'll now look at one particular case of using expert knowledge—though in this case it might be more rightfully called "common sense." The task is predicting bicycle rentals in front of Andreas's house.

In New York, Citi Bike operates a network of bicycle rental stations with a subscription system. The stations are all over the city and provide a convenient way to get around. Bike rental data is made public in an anonymized form and has been analyzed in various ways. The task we want to solve is to predict for a given time and day how many people will rent a bike in front of Andreas's house—so he knows if any bikes will be left for him.

We first load the data for August 2015 for this particular station as a `pandas DataFrame`. We resample the data into three-hour intervals to obtain the main trends for each day:

`In[49]:`

```
citibike = mglearn.datasets.load_citibike()
```

In[50]:

```
print("Citi Bike data:\n{}".format(citibike.head()))
```

Out[50]:

```
Citi Bike data:
starttime
2015-08-01 00:00:00     3.0
2015-08-01 03:00:00     0.0
2015-08-01 06:00:00     9.0
2015-08-01 09:00:00    41.0
2015-08-01 12:00:00    39.0
Freq: 3H, Name: one, dtype: float64
```

The following example shows a visualization of the rental frequencies for the whole month (Figure 4-12):

In[51]:

```
plt.figure(figsize=(10, 3))
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(),
                       freq='D')
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")
plt.plot(citibike, linewidth=1)
plt.xlabel("Date")
plt.ylabel("Rentals")
```

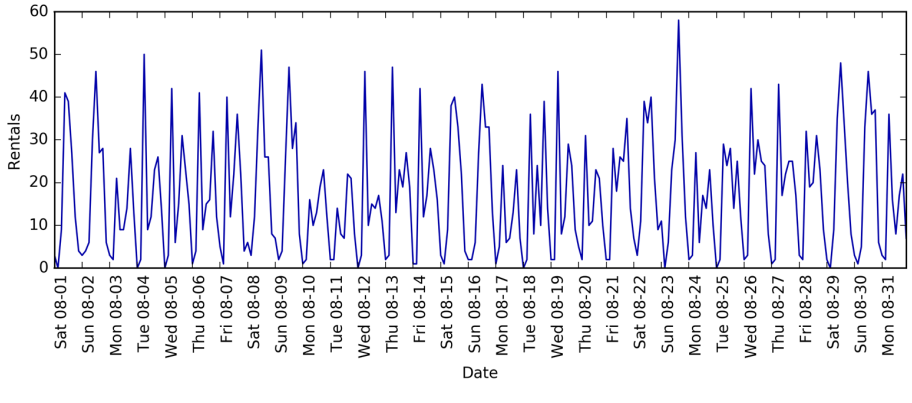

Figure 4-12. Number of bike rentals over time for a selected Citi Bike station

Looking at the data, we can clearly distinguish day and night for each 24-hour interval. The patterns for weekdays and weekends also seem to be quite different. When evaluating a prediction task on a time series like this, we usually want to *learn from the past* and *predict for the future*. This means when doing a split into a training and a test set, we want to use all the data up to a certain date as the training set and all the data past that date as the test set. This is how we would usually use time series prediction: given everything that we know about rentals in the past, what do we think will

happen tomorrow? We will use the first 184 data points, corresponding to the first 23 days, as our training set, and the remaining 64 data points, corresponding to the remaining 8 days, as our test set.

The only feature that we are using in our prediction task is the date and time when a particular number of rentals occurred. So, the input feature is the date and time—say, `2015-08-01 00:00:00`—and the output is the number of rentals in the following three hours (three in this case, according to our `DataFrame`).

A (surprisingly) common way that dates are stored on computers is using POSIX time, which is the number of seconds since January 1970 00:00:00 (aka the beginning of Unix time). As a first try, we can use this single integer feature as our data representation:

In[52]:

```
# extract the target values (number of rentals)
y = citibike.values
# convert the time to POSIX time using "%s"
X = citibike.index.strftime("%s").astype("int").reshape(-1, 1)
```

We first define a function to split the data into training and test sets, build the model, and visualize the result:

In[54]:

```
# use the first 184 data points for training, and the rest for testing
n_train = 184

# function to evaluate and plot a regressor on a given feature set
def eval_on_features(features, target, regressor):
    # split the given features into a training and a test set
    X_train, X_test = features[:n_train], features[n_train:]
    # also split the target array
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)))
    y_pred = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)
    plt.figure(figsize=(10, 3))

    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90,
               ha="left")

    plt.plot(range(n_train), y_train, label="train")
    plt.plot(range(n_train, len(y_test) + n_train), y_test, '-', label="test")
    plt.plot(range(n_train), y_pred_train, '--', label="prediction train")

    plt.plot(range(n_train, len(y_test) + n_train), y_pred, '--',
             label="prediction test")
    plt.legend(loc=(1.01, 0))
    plt.xlabel("Date")
    plt.ylabel("Rentals")
```

We saw earlier that random forests require very little preprocessing of the data, which makes this seem like a good model to start with. We use the POSIX time feature `X` and pass a random forest regressor to our `eval_on_features` function. Figure 4-13 shows the result:

In[55]:

```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators=100, random_state=0)
plt.figure()
eval_on_features(X, y, regressor)
```

Out[55]:

```
Test-set R^2: -0.04
```

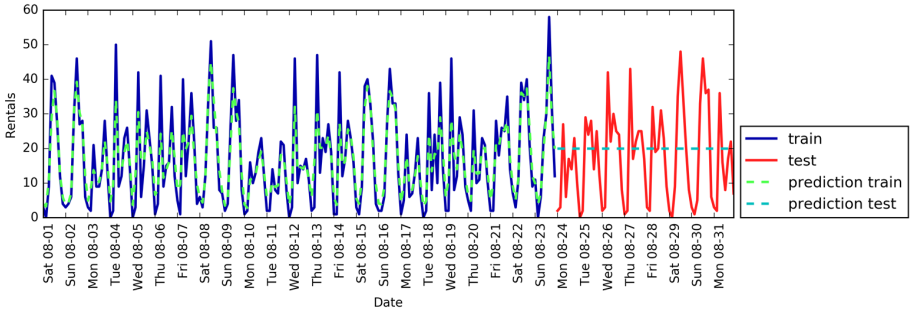

Figure 4-13. Predictions made by a random forest using only the POSIX time

The predictions on the training set are quite good, as is usual for random forests. However, for the test set, a constant line is predicted. The R^2 is –0.03, which means that we learned nothing. What happened?

The problem lies in the combination of our feature and the random forest. The value of the POSIX time feature for the test set is outside of the range of the feature values in the training set: the points in the test set have timestamps that are later than all the points in the training set. Trees, and therefore random forests, cannot *extrapolate* to feature ranges outside the training set. The result is that the model simply predicts the target value of the closest point in the training set—which is the last time it observed any data.

Clearly we can do better than this. This is where our "expert knowledge" comes in. From looking at the rental figures in the training data, two factors seem to be very important: the time of day and the day of the week. So, let's add these two features. We can't really learn anything from the POSIX time, so we drop that feature. First, let's use only the hour of the day. As Figure 4-14 shows, now the predictions have the same pattern for each day of the week:

In[56]:

```
X_hour = citibike.index.hour.reshape(-1, 1)
eval_on_features(X_hour, y, regressor)
```

Out[56]:

```
Test-set R^2: 0.60
```

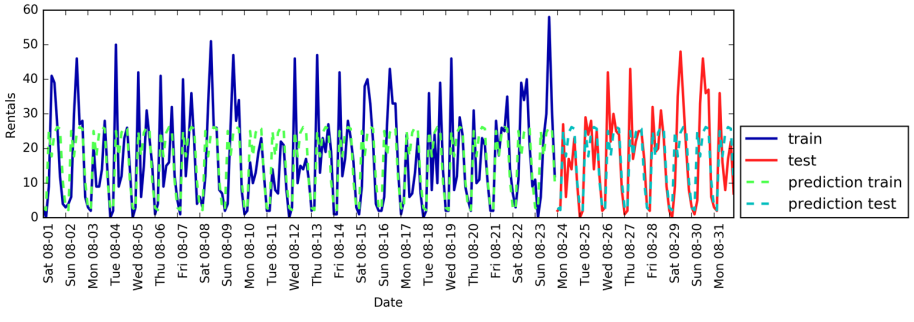

Figure 4-14. Predictions made by a random forest using only the hour of the day

The R^2 is already much better, but the predictions clearly miss the weekly pattern. Now let's also add the day of the week (see Figure 4-15):

In[57]:

```
X_hour_week = np.hstack([citibike.index.dayofweek.reshape(-1, 1),
                         citibike.index.hour.reshape(-1, 1)])
eval_on_features(X_hour_week, y, regressor)
```

Out[57]:

```
Test-set R^2: 0.84
```

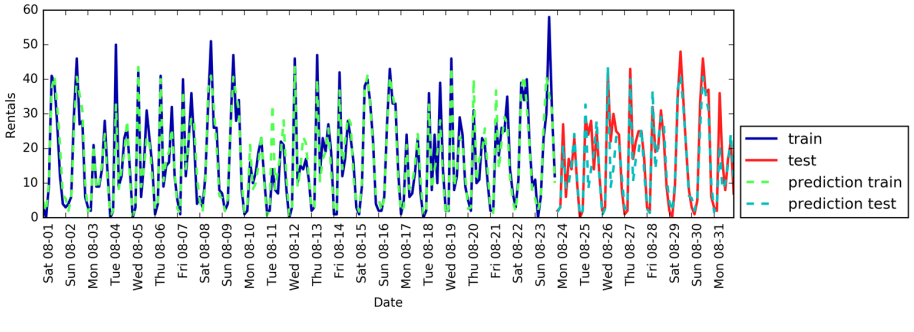

Figure 4-15. Predictions with a random forest using day of week and hour of day features

Now we have a model that captures the periodic behavior by considering the day of week and time of day. It has an R^2 of 0.84, and shows pretty good predictive performance. What this model likely is learning is the mean number of rentals for each combination of weekday and time of day from the first 23 days of August. This actually does not require a complex model like a random forest, so let's try with a simpler model, `LinearRegression` (see Figure 4-16):

In[58]:

```
from sklearn.linear_model import LinearRegression
eval_on_features(X_hour_week, y, LinearRegression())
```

Out[58]:

```
Test-set R^2: 0.13
```

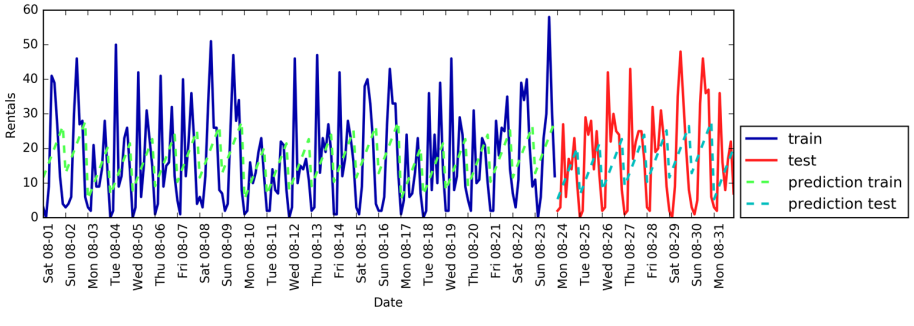

Figure 4-16. Predictions made by linear regression using day of week and hour of day as features

`LinearRegression` works much worse, and the periodic pattern looks odd. The reason for this is that we encoded day of week and time of day using integers, which are interpreted as categorical variables. Therefore, the linear model can only learn a linear function of the time of day—and it learned that later in the day, there are more rentals. However, the patterns are much more complex than that. We can capture this by interpreting the integers as categorical variables, by transforming them using `OneHotEncoder` (see Figure 4-17):

In[59]:

```
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
```

In[60]:

```
eval_on_features(X_hour_week_onehot, y, Ridge())
```

Out[60]:

```
Test-set R^2: 0.62
```

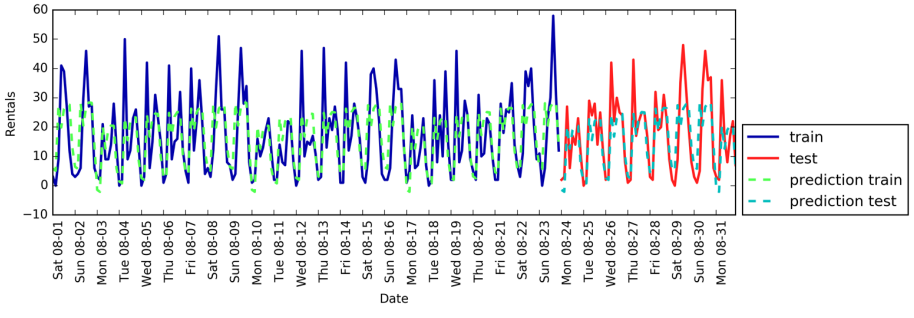

Figure 4-17. Predictions made by linear regression using a one-hot encoding of hour of day and day of week

This gives us a much better match than the continuous feature encoding. Now the linear model learns one coefficient for each day of the week, and one coefficient for each time of the day. That means that the "time of day" pattern is shared over all days of the week, though.

Using interaction features, we can allow the model to learn one coefficient for each combination of day and time of day (see Figure 4-18):

In[61]:

```
poly_transformer = PolynomialFeatures(degree=2, interaction_only=True,
                                      include_bias=False)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

Out[61]:

```
Test-set R^2: 0.85
```

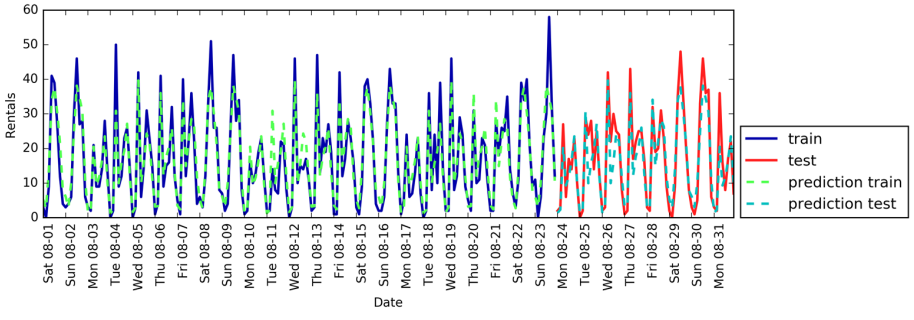

Figure 4-18. Predictions made by linear regression using a product of the day of week and hour of day features

This transformation finally yields a model that performs similarly well to the random forest. A big benefit of this model is that it is very clear what is learned: one coefficient for each day and time. We can simply plot the coefficients learned by the model, something that would not be possible for the random forest.

First, we create feature names for the hour and day features:

In[62]:

```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features =  day + hour
```

Then we name all the interaction features extracted by `PolynomialFeatures`, using the `get_feature_names` method, and keep only the features with nonzero coefficients:

In[63]:

```
features_poly = poly_transformer.get_feature_names(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

Now we can visualize the coefficients learned by the linear model, as seen in Figure 4-19:

In[64]:

```
plt.figure(figsize=(15, 2))
plt.plot(coef_nonzero, 'o')
plt.xticks(np.arange(len(coef_nonzero)), features_nonzero, rotation=90)
plt.xlabel("Feature magnitude")
plt.ylabel("Feature")
```

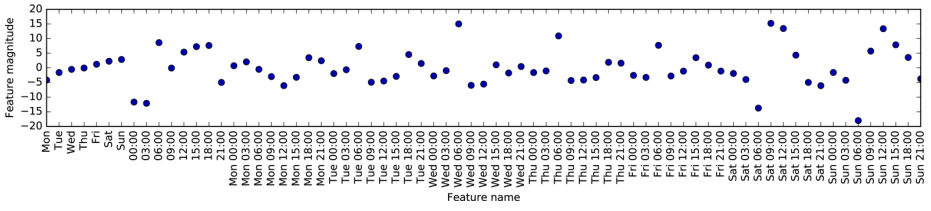

Figure 4-19. Coefficients of the linear regression model using a product of hour and day

Summary and Outlook

In this chapter, we discussed how to deal with different data types (in particular, with categorical variables). We emphasized the importance of representing data in a way that is suitable for the machine learning algorithm—for example, by one-hot-encoding categorical variables. We also discussed the importance of engineering new features, and the possibility of utilizing expert knowledge in creating derived features from your data. In particular, linear models might benefit greatly from generating new features via binning and adding polynomials and interactions, while more complex, nonlinear models like random forests and SVMs might be able to learn more complex tasks without explicitly expanding the feature space. In practice, the features that are used (and the match between features and method) is often the most important piece in making a machine learning approach work well.

Now that you have a good idea of how to represent your data in an appropriate way and which algorithm to use for which task, the next chapter will focus on evaluating the performance of machine learning models and selecting the right parameter settings.

CHAPTER 5

Model Evaluation and Improvement

Having discussed the fundamentals of supervised and unsupervised learning, and having explored a variety of machine learning algorithms, we will now dive more deeply into evaluating models and selecting parameters.

We will focus on the supervised methods, regression and classification, as evaluating and selecting models in unsupervised learning is often a very qualitative process (as we saw in Chapter 3).

To evaluate our supervised models, so far we have split our dataset into a training set and a test set using the `train_test_split` function, built a model on the training set by calling the `fit` method, and evaluated it on the test set using the `score` method, which for classification computes the fraction of correctly classified samples. Here's an example of that process:

In[2]:

```
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# create a synthetic dataset
X, y = make_blobs(random_state=0)
# split data and labels into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# instantiate a model and fit it to the training set
logreg = LogisticRegression().fit(X_train, y_train)
# evaluate the model on the test set
print("Test set score: {:.2f}".format(logreg.score(X_test, y_test)))
```

Out[2]:

```
Test set score: 0.88
```

Remember, the reason we split our data into training and test sets is that we are interested in measuring how well our model *generalizes* to new, previously unseen data. We are not interested in how well our model fit the training set, but rather in how well it can make predictions for data that was not observed during training.

In this chapter, we will expand on two aspects of this evaluation. We will first introduce cross-validation, a more robust way to assess generalization performance, and discuss methods to evaluate classification and regression performance that go beyond the default measures of accuracy and R^2 provided by the `score` method.

We will also discuss grid search, an effective method for adjusting the parameters in supervised models for the best generalization performance.

Cross-Validation

Cross-validation is a statistical method of evaluating generalization performance that is more stable and thorough than using a split into a training and a test set. In cross-validation, the data is instead split repeatedly and multiple models are trained. The most commonly used version of cross-validation is *k-fold cross-validation*, where *k* is a user-specified number, usually 5 or 10. When performing five-fold cross-validation, the data is first partitioned into five parts of (approximately) equal size, called *folds*. Next, a sequence of models is trained. The first model is trained using the first fold as the test set, and the remaining folds (2–5) are used as the training set. The model is built using the data in folds 2–5, and then the accuracy is evaluated on fold 1. Then another model is built, this time using fold 2 as the test set and the data in folds 1, 3, 4, and 5 as the training set. This process is repeated using folds 3, 4, and 5 as test sets. For each of these five *splits* of the data into training and test sets, we compute the accuracy. In the end, we have collected five accuracy values. The process is illustrated in Figure 5-1:

`In[3]:`

```
mglearn.plots.plot_cross_validation()
```

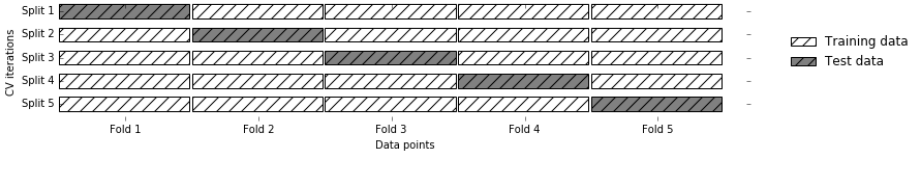

Figure 5-1. Data splitting in five-fold cross-validation

Usually, the first fifth of the data is the first fold, the second fifth of the data is the second fold, and so on.

Cross-Validation in scikit-learn

Cross-validation is implemented in `scikit-learn` using the `cross_val_score` function from the `model_selection` module. The parameters of the `cross_val_score` function are the model we want to evaluate, the training data, and the ground-truth labels. Let's evaluate `LogisticRegression` on the `iris` dataset:

In[4]:

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()

scores = cross_val_score(logreg, iris.data, iris.target)
print("Cross-validation scores: {}".format(scores))
```

Out[4]:

```
Cross-validation scores: [ 0.961  0.922  0.958]
```

By default, `cross_val_score` performs three-fold cross-validation, returning three accuracy values. We can change the number of folds used by changing the `cv` parameter:

In[5]:

```
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
print("Cross-validation scores: {}".format(scores))
```

Out[5]:

```
Cross-validation scores: [ 1.     0.967  0.933  0.9    1.   ]
```

A common way to summarize the cross-validation accuracy is to compute the mean:

In[6]:

```
print("Average cross-validation score: {:.2f}".format(scores.mean()))
```

Out[6]:

```
Average cross-validation score: 0.96
```

Using the mean cross-validation we can conclude that we expect the model to be around 96% accurate on average. Looking at all five scores produced by the five-fold cross-validation, we can also conclude that there is a relatively high variance in the accuracy between folds, ranging from 100% accuracy to 90% accuracy. This could imply that the model is very dependent on the particular folds used for training, but it could also just be a consequence of the small size of the dataset.

Benefits of Cross-Validation

There are several benefits to using cross-validation instead of a single split into a training and a test set. First, remember that `train_test_split` performs a random split of the data. Imagine that we are "lucky" when randomly splitting the data, and all examples that are hard to classify end up in the training set. In that case, the test set will only contain "easy" examples, and our test set accuracy will be unrealistically high. Conversely, if we are "unlucky," we might have randomly put all the hard-to-classify examples in the test set and consequently obtain an unrealistically low score. However, when using cross-validation, each example will be in the training set exactly once: each example is in one of the folds, and each fold is the test set once. Therefore, the model needs to generalize well to all of the samples in the dataset for all of the cross-validation scores (and their mean) to be high.

Having multiple splits of the data also provides some information about how sensitive our model is to the selection of the training dataset. For the `iris` dataset, we saw accuracies between 90% and 100%. This is quite a range, and it provides us with an idea about how the model might perform in the worst case and best case scenarios when applied to new data.

Another benefit of cross-validation as compared to using a single split of the data is that we use our data more effectively. When using `train_test_split`, we usually use 75% of the data for training and 25% of the data for evaluation. When using five-fold cross-validation, in each iteration we can use four-fifths of the data (80%) to fit the model. When using 10-fold cross-validation, we can use nine-tenths of the data (90%) to fit the model. More data will usually result in more accurate models.

The main disadvantage of cross-validation is increased computational cost. As we are now training k models instead of a single model, cross-validation will be roughly k times slower than doing a single split of the data.

It is important to keep in mind that cross-validation is not a way to build a model that can be applied to new data. Cross-validation does not return a model. When calling `cross_val_score`, multiple models are built internally, but the purpose of cross-validation is only to evaluate how well a given algorithm will generalize when trained on a specific dataset.

Stratified k-Fold Cross-Validation and Other Strategies

Splitting the dataset into k folds by starting with the first one-k-th part of the data, as described in the previous section, might not always be a good idea. For example, let's have a look at the `iris` dataset:

In[7]:

```
from sklearn.datasets import load_iris
iris = load_iris()
print("Iris labels:\n{}".format(iris.target))
```

Out[7]:

```
Iris labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

As you can see, the first third of the data is the class 0, the second third is the class 1, and the last third is the class 2. Imagine doing three-fold cross-validation on this dataset. The first fold would be only class 0, so in the first split of the data, the test set would be only class 0, and the training set would be only classes 1 and 2. As the classes in training and test sets would be different for all three splits, the three-fold cross-validation accuracy would be zero on this dataset. That is not very helpful, as we can do much better than 0% accuracy on `iris`.

As the simple *k*-fold strategy fails here, `scikit-learn` does not use it for classification, but rather uses *stratified k-fold cross-validation*. In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset, as illustrated in Figure 5-2:

In[8]:

```
mglearn.plots.plot_stratified_cross_validation()
```

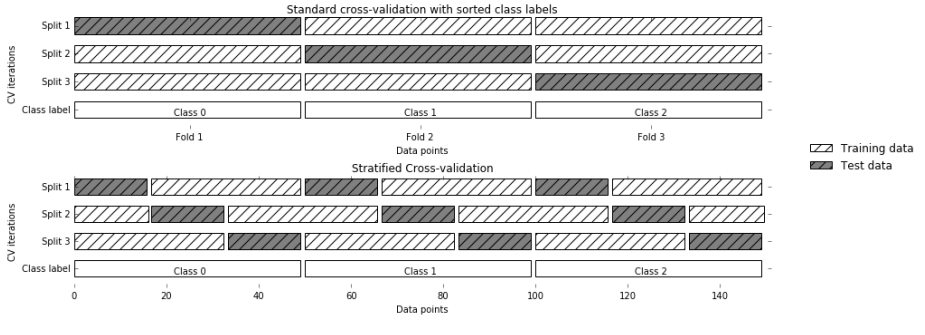

Figure 5-2. Comparison of standard cross-validation and stratified cross-validation when the data is ordered by class label

For example, if 90% of your samples belong to class A and 10% of your samples belong to class B, then stratified cross-validation ensures that in each fold, 90% of samples belong to class A and 10% of samples belong to class B.

It is usually a good idea to use stratified *k*-fold cross-validation instead of *k*-fold cross-validation to evaluate a classifier, because it results in more reliable estimates of generalization performance. In the case of only 10% of samples belonging to class B, using standard *k*-fold cross-validation it might easily happen that one fold only contains samples of class A. Using this fold as a test set would not be very informative about the overall performance of the classifier.

For regression, `scikit-learn` uses the standard *k*-fold cross-validation by default. It would be possible to also try to make each fold representative of the different values the regression target has, but this is not a commonly used strategy and would be surprising to most users.

More control over cross-validation

We saw earlier that we can adjust the number of folds that are used in `cross_val_score` using the `cv` parameter. However, `scikit-learn` allows for much finer control over what happens during the splitting of the data by providing a *cross-validation splitter* as the `cv` parameter. For most use cases, the defaults of *k*-fold cross-validation for regression and stratified *k*-fold for classification work well, but there are some cases where you might want to use a different strategy. Say, for example, we want to use the standard *k*-fold cross-validation on a classification dataset to reproduce someone else's results. To do this, we first have to import the `KFold` splitter class from the `model_selection` module and instantiate it with the number of folds we want to use:

`In[9]:`

```
from sklearn.model_selection import KFold
kfold = KFold(n_splits=5)
```

Then, we can pass the `kfold` splitter object as the `cv` parameter to `cross_val_score`:

`In[10]:`

```
print("Cross-validation scores:\n{}".format(
      cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

`Out[10]:`

```
Cross-validation scores:
[ 1.     0.933  0.433  0.967  0.433]
```

This way, we can verify that it is indeed a really bad idea to use three-fold (nonstratified) cross-validation on the `iris` dataset:

In[11]:

```
kfold = KFold(n_splits=3)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[11]:

```
Cross-validation scores:
[ 0.  0.  0.]
```

Remember: each fold corresponds to one of the classes in the `iris` dataset, and so nothing can be learned. Another way to resolve this problem is to shuffle the data instead of stratifying the folds, to remove the ordering of the samples by label. We can do that by setting the `shuffle` parameter of `KFold` to `True`. If we shuffle the data, we also need to fix the `random_state` to get a reproducible shuffling. Otherwise, each run of `cross_val_score` would yield a different result, as each time a different split would be used (this might not be a problem, but can be surprising). Shuffling the data before splitting it yields a much better result:

In[12]:

```
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[12]:

```
Cross-validation scores:
[ 0.9   0.96  0.96]
```

Leave-one-out cross-validation

Another frequently used cross-validation method is *leave-one-out*. You can think of leave-one-out cross-validation as *k*-fold cross-validation where each fold is a single sample. For each split, you pick a single data point to be the test set. This can be very time consuming, particularly for large datasets, but sometimes provides better estimates on small datasets:

In[13]:

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Number of cv iterations: ", len(scores))
print("Mean accuracy: {:.2f}".format(scores.mean()))
```

Out[13]:

```
Number of cv iterations:  150
Mean accuracy: 0.95
```

Shuffle-split cross-validation

Another, very flexible strategy for cross-validation is *shuffle-split cross-validation*. In shuffle-split cross-validation, each split samples `train_size` many points for the training set and `test_size` many (disjoint) point for the test set. This splitting is repeated `n_iter` times. Figure 5-3 illustrates running four iterations of splitting a dataset consisting of 10 points, with a training set of 5 points and test sets of 2 points each (you can use integers for `train_size` and `test_size` to use absolute sizes for these sets, or floating-point numbers to use fractions of the whole dataset):

`In[14]:`

```
mglearn.plots.plot_shuffle_split()
```

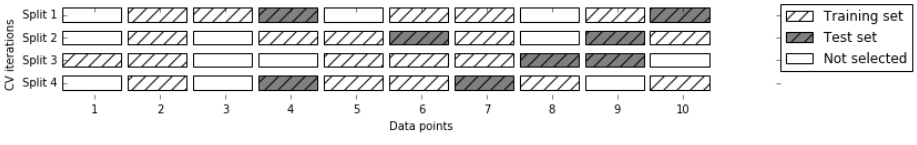

Figure 5-3. ShuffleSplit with 10 points, train_size=5, test_size=2, and n_iter=4

The following code splits the dataset into 50% training set and 50% test set for 10 iterations:

`In[15]:`

```
from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Cross-validation scores:\n{}".format(scores))
```

`Out[15]:`

```
Cross-validation scores:
[ 0.96   0.907  0.947  0.96   0.96   0.907  0.893  0.907  0.92   0.973]
```

Shuffle-split cross-validation allows for control over the number of iterations independently of the training and test sizes, which can sometimes be helpful. It also allows for using only part of the data in each iteration, by providing `train_size` and `test_size` settings that don't add up to one. Subsampling the data in this way can be useful for experimenting with large datasets.

There is also a stratified variant of `ShuffleSplit`, aptly named `StratifiedShuffleSplit`, which can provide more reliable results for classification tasks.

Cross-validation with groups

Another very common setting for cross-validation is when there are groups in the data that are highly related. Say you want to build a system to recognize emotions from pictures of faces, and you collect a dataset of pictures of 100 people where each person is captured multiple times, showing various emotions. The goal is to build a classifier that can correctly identify emotions of people not in the dataset. You could use the default stratified cross-validation to measure the performance of a classifier here. However, it is likely that pictures of the same person will be in both the training and the test set. It will be much easier for a classifier to detect emotions in a face that is part of the training set, compared to a completely new face. To accurately evaluate the generalization to new faces, we must therefore ensure that the training and test sets contain images of different people.

To achieve this, we can use `GroupKFold`, which takes an array of `groups` as argument that we can use to indicate which person is in the image. The `groups` array here indicates groups in the data that should not be split when creating the training and test sets, and should not be confused with the class label.

This example of groups in the data is common in medical applications, where you might have multiple samples from the same patient, but are interested in generalizing to new patients. Similarly, in speech recognition, you might have multiple recordings of the same speaker in your dataset, but are interested in recognizing speech of new speakers.

The following is an example of using a synthetic dataset with a grouping given by the `groups` array. The dataset consists of 12 data points, and for each of the data points, `groups` specifies which group (think patient) the point belongs to. The groups specify that there are four groups, and the first three samples belong to the first group, the next four samples belong to the second group, and so on:

`In[17]:`

```
from sklearn.model_selection import GroupKFold
# create synthetic dataset
X, y = make_blobs(n_samples=12, random_state=0)
# assume the first three samples belong to the same group,
# then the next four, etc.
groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Cross-validation scores:\n{}".format(scores))
```

`Out[17]:`

```
Cross-validation scores:
[ 0.75   0.8    0.667]
```

The samples don't need to be ordered by group; we just did this for illustration purposes. The splits that are calculated based on these labels are visualized in Figure 5-4.

As you can see, for each split, each group is either entirely in the training set or entirely in the test set:

In[16]:

```
mglearn.plots.plot_label_kfold()
```

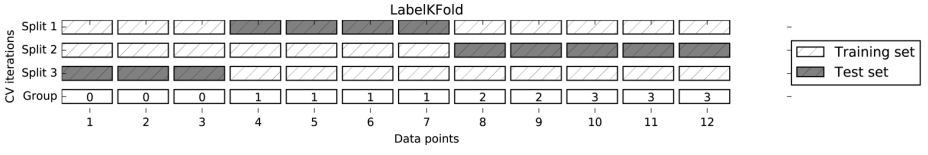

Figure 5-4. Label-dependent splitting with GroupKFold

There are more splitting strategies for cross-validation in `scikit-learn`, which allow for an even greater variety of use cases (you can find these in the `scikit-learn` user guide). However, the standard `KFold`, `StratifiedKFold`, and `GroupKFold` are by far the most commonly used ones.

Grid Search

Now that we know how to evaluate how well a model generalizes, we can take the next step and improve the model's generalization performance by tuning its parameters. We discussed the parameter settings of many of the algorithms in `scikit-learn` in Chapters 2 and 3, and it is important to understand what the parameters mean before trying to adjust them. Finding the values of the important parameters of a model (the ones that provide the best generalization performance) is a tricky task, but necessary for almost all models and datasets. Because it is such a common task, there are standard methods in `scikit-learn` to help you with it. The most commonly used method is *grid search*, which basically means trying all possible combinations of the parameters of interest.

Consider the case of a kernel SVM with an RBF (radial basis function) kernel, as implemented in the `SVC` class. As we discussed in Chapter 2, there are two important parameters: the kernel bandwidth, `gamma`, and the regularization parameter, `C`. Say we want to try the values `0.001`, `0.01`, `0.1`, `1`, `10`, and `100` for the parameter `C`, and the same for `gamma`. Because we have six different settings for `C` and `gamma` that we want to try, we have 36 combinations of parameters in total. Looking at all possible combinations creates a table (or grid) of parameter settings for the SVM, as shown here:

	C = 0.001	C = 0.01	...	C = 10
gamma=0.001	SVC(C=0.001, gamma=0.001)	SVC(C=0.01, gamma=0.001)	...	SVC(C=10, gamma=0.001)
gamma=0.01	SVC(C=0.001, gamma=0.01)	SVC(C=0.01, gamma=0.01)	...	SVC(C=10, gamma=0.01)
...	...	...	...	...
gamma=100	SVC(C=0.001, gamma=100)	SVC(C=0.01, gamma=100)	...	SVC(C=10, gamma=100)

Simple Grid Search

We can implement a simple grid search just as `for` loops over the two parameters, training and evaluating a classifier for each combination:

In[18]:

```
# naive grid search implementation
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
print("Size of training set: {}   size of test set: {}".format(
      X_train.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the test set
        score = svm.score(X_test, y_test)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("Best score: {:.2f}".format(best_score))
print("Best parameters: {}".format(best_parameters))
```

Out[18]:

```
Size of training set: 112   size of test set: 38
Best score: 0.97
Best parameters: {'C': 100, 'gamma': 0.001}
```

The Danger of Overfitting the Parameters and the Validation Set

Given this result, we might be tempted to report that we found a model that performs with 97% accuracy on our dataset. However, this claim could be overly optimistic (or just wrong), for the following reason: we tried many different parameters and

selected the one with best accuracy on the test set, but this accuracy won't necessarily carry over to new data. Because we used the test data to adjust the parameters, we can no longer use it to assess how good the model is. This is the same reason we needed to split the data into training and test sets in the first place; we need an independent dataset to evaluate, one that was not used to create the model.

One way to resolve this problem is to split the data again, so we have three sets: the training set to build the model, the validation (or development) set to select the parameters of the model, and the test set to evaluate the performance of the selected parameters. Figure 5-5 shows what this looks like:

In[19]:

```
mglearn.plots.plot_threefold_split()
```

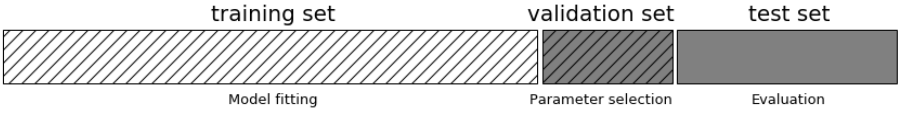

Figure 5-5. A threefold split of data into training set, validation set, and test set

After selecting the best parameters using the validation set, we can rebuild a model using the parameter settings we found, but now training on both the training data and the validation data. This way, we can use as much data as possible to build our model. This leads to the following implementation:

In[20]:

```
from sklearn.svm import SVC
# split data into train+validation set and test set
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
# split train+validation set into training and validation sets
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
print("Size of training set: {}   size of validation set: {}   size of test set:"
      " {}\n".format(X_train.shape[0], X_valid.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the test set
        score = svm.score(X_valid, y_valid)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
```

```
    # rebuild a model on the combined training and validation set,
    # and evaluate it on the test set
    svm = SVC(**best_parameters)
    svm.fit(X_trainval, y_trainval)
    test_score = svm.score(X_test, y_test)
    print("Best score on validation set: {:.2f}".format(best_score))
    print("Best parameters: ", best_parameters)
    print("Test set score with best parameters: {:.2f}".format(test_score))
```

Out[20]:

```
    Size of training set: 84   size of validation set: 28   size of test set: 38

    Best score on validation set: 0.96
    Best parameters:  {'C': 10, 'gamma': 0.001}
    Test set score with best parameters: 0.92
```

The best score on the validation set is 96%: slightly lower than before, probably because we used less data to train the model (`X_train` is smaller now because we split our dataset twice). However, the score on the test set—the score that actually tells us how well we generalize—is even lower, at 92%. So we can only claim to classify new data 92% correctly, not 97% correctly as we thought before!

The distinction between the training set, validation set, and test set is fundamentally important to applying machine learning methods in practice. Any choices made based on the test set accuracy "leak" information from the test set into the model. Therefore, it is important to keep a separate test set, which is only used for the final evaluation. It is good practice to do all exploratory analysis and model selection using the combination of a training and a validation set, and reserve the test set for a final evaluation—this is even true for exploratory visualization. Strictly speaking, evaluating more than one model on the test set and choosing the better of the two will result in an overly optimistic estimate of how accurate the model is.

Grid Search with Cross-Validation

While the method of splitting the data into a training, a validation, and a test set that we just saw is workable, and relatively commonly used, it is quite sensitive to how exactly the data is split. From the output of the previous code snippet we can see that `GridSearchCV` selects `'C': 10, 'gamma': 0.001` as the best parameters, while the output of the code in the previous section selects `'C': 100, 'gamma': 0.001` as the best parameters. For a better estimate of the generalization performance, instead of using a single split into a training and a validation set, we can use cross-validation to evaluate the performance of each parameter combination. This method can be coded up as follows:

In[21]:

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters,
        # train an SVC
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
# rebuild a model on the combined training and validation set
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

To evaluate the accuracy of the SVM using a particular setting of `C` and `gamma` using five-fold cross-validation, we need to train 36 * 5 = 180 models. As you can imagine, the main downside of the use of cross-validation is the time it takes to train all these models.

The following visualization (Figure 5-6) illustrates how the best parameter setting is selected in the preceding code:

In[22]:

```
mglearn.plots.plot_cross_val_selection()
```

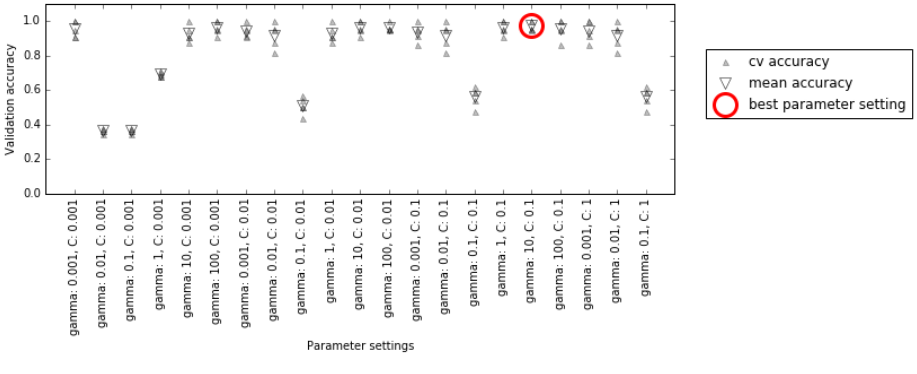

Figure 5-6. Results of grid search with cross-validation

For each parameter setting (only a subset is shown), five accuracy values are computed, one for each split in the cross-validation. Then the mean validation accuracy is computed for each parameter setting. The parameters with the highest mean validation accuracy are chosen, marked by the circle.

As we said earlier, cross-validation is a way to evaluate a given algorithm on a specific dataset. However, it is often used in conjunction with parameter search methods like grid search. For this reason, many people use the term *cross-validation* colloquially to refer to grid search with cross-validation.

The overall process of splitting the data, running the grid search, and evaluating the final parameters is illustrated in Figure 5-7:

In[23]:

```
mglearn.plots.plot_grid_search_overview()
```

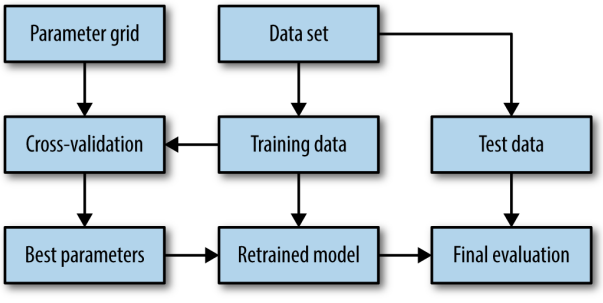

Figure 5-7. Overview of the process of parameter selection and model evaluation with GridSearchCV

Because grid search with cross-validation is such a commonly used method to adjust parameters, `scikit-learn` provides the `GridSearchCV` class, which implements it in the form of an estimator. To use the `GridSearchCV` class, you first need to specify the parameters you want to search over using a dictionary. `GridSearchCV` will then perform all the necessary model fits. The keys of the dictionary are the names of parameters we want to adjust (as given when constructing the model—in this case, `C` and `gamma`), and the values are the parameter settings we want to try out. Trying the values `0.001`, `0.01`, `0.1`, `1`, `10`, and `100` for `C` and `gamma` translates to the following dictionary:

In[24]:

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))
```

Out[24]:

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

We can now instantiate the `GridSearchCV` class with the model (`SVC`), the parameter grid to search (`param_grid`), and the cross-validation strategy we want to use (say, five-fold stratified cross-validation):

In[25]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

`GridSearchCV` will use cross-validation in place of the split into a training and validation set that we used before. However, we still need to split the data into a training and a test set, to avoid overfitting the parameters:

In[26]:

```
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
```

The `grid_search` object that we created behaves just like a classifier; we can call the standard methods `fit`, `predict`, and `score` on it.[1] However, when we call `fit`, it will run cross-validation for each combination of parameters we specified in `param_grid`:

In[27]:

```
grid_search.fit(X_train, y_train)
```

Fitting the `GridSearchCV` object not only searches for the best parameters, but also automatically fits a new model on the whole training dataset with the parameters that yielded the best cross-validation performance. What happens in `fit` is therefore equivalent to the result of the In[21] code we saw at the beginning of this section. The `GridSearchCV` class provides a very convenient interface to access the retrained model using the `predict` and `score` methods. To evaluate how well the best found parameters generalize, we can call `score` on the test set:

In[28]:

```
print("Test set score: {:.2f}".format(grid_search.score(X_test, y_test)))
```

Out[28]:

```
Test set score: 0.97
```

Choosing the parameters using cross-validation, we actually found a model that achieves 97% accuracy on the test set. The important thing here is that we *did not use the test set* to choose the parameters. The parameters that were found are scored in the

1 A `scikit-learn` estimator that is created using another estimator is called a *meta-estimator*. `GridSearchCV` is the most commonly used meta-estimator, but we will see more later.