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")
```

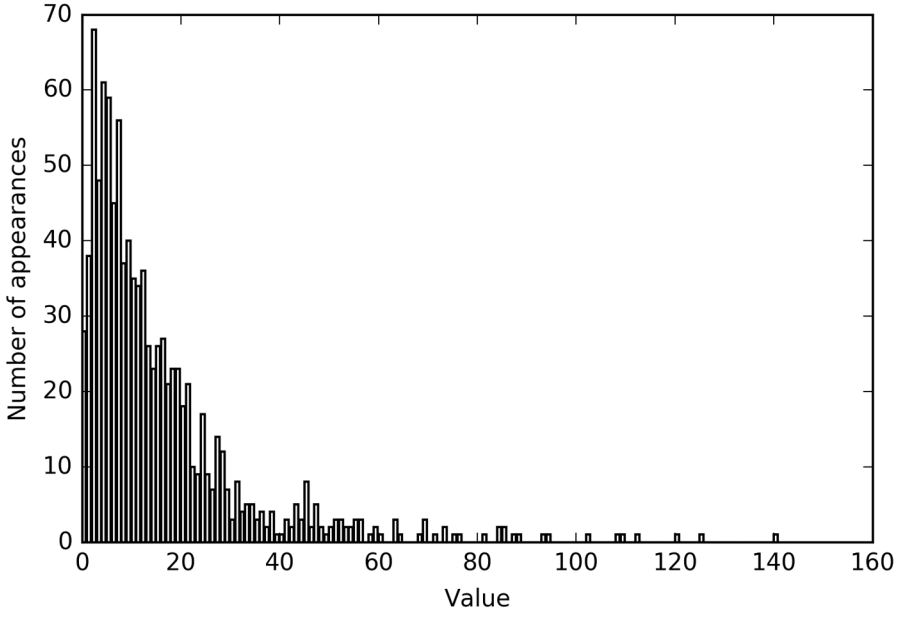

Figure 4-7. Histogram of feature values for X[0]

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.

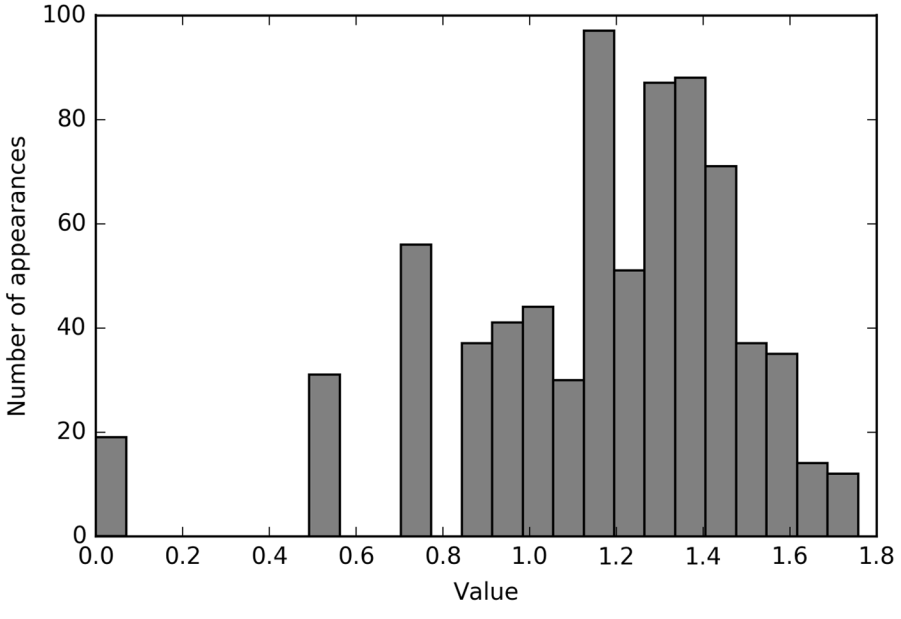

Figure 4-8. Histogram of feature values for X[0] after logarithmic transformation

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