tained within the rectangle created by the x-axis between 0 and 1 and the y-axis between 0 and 1.

Finally, the `Normalizer` does a very different kind of rescaling. It scales each data point such that the feature vector has a Euclidean length of 1. In other words, it projects a data point on the circle (or sphere, in the case of higher dimensions) with a radius of 1. This means every data point is scaled by a different number (by the inverse of its length). This normalization is often used when only the direction (or angle) of the data matters, not the length of the feature vector.

Applying Data Transformations

Now that we've seen what the different kinds of transformations do, let's apply them using `scikit-learn`. We will use the `cancer` dataset that we saw in Chapter 2. Preprocessing methods like the scalers are usually applied before applying a supervised machine learning algorithm. As an example, say we want to apply the kernel SVM (`SVC`) to the `cancer` dataset, and use `MinMaxScaler` for preprocessing the data. We start by loading our dataset and splitting it into a training set and a test set (we need separate training and test sets to evaluate the supervised model we will build after the preprocessing):

`In[3]:`

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=1)
print(X_train.shape)
print(X_test.shape)
```

`Out[3]:`

```
(426, 30)
(143, 30)
```

As a reminder, the dataset contains 569 data points, each represented by 30 measurements. We split the dataset into 426 samples for the training set and 143 samples for the test set.

As with the supervised models we built earlier, we first import the class that implements the preprocessing, and then instantiate it:

`In[4]:`

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

We then fit the scaler using the `fit` method, applied to the training data. For the `MinMaxScaler`, the `fit` method computes the minimum and maximum value of each feature on the training set. In contrast to the classifiers and regressors of Chapter 2, the scaler is only provided with the data (`X_train`) when `fit` is called, and `y_train` is not used:

In[5]:

```
scaler.fit(X_train)
```

Out[5]:

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

To apply the transformation that we just learned—that is, to actually *scale* the training data—we use the `transform` method of the scaler. The `transform` method is used in `scikit-learn` whenever a model returns a new representation of the data:

In[6]:

```
# transform data
X_train_scaled = scaler.transform(X_train)
# print dataset properties before and after scaling
print("transformed shape: {}".format(X_train_scaled.shape))
print("per-feature minimum before scaling:\n {}".format(X_train.min(axis=0)))
print("per-feature maximum before scaling:\n {}".format(X_train.max(axis=0)))
print("per-feature minimum after scaling:\n {}".format(
    X_train_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n {}".format(
    X_train_scaled.max(axis=0)))
```

Out[6]:

```
transformed shape: (426, 30)
per-feature minimum before scaling:
 [   6.98    9.71   43.79  143.50    0.05    0.02    0.      0.      0.11
     0.05    0.12    0.36    0.76    6.80    0.      0.      0.      0.
     0.01    0.      7.93   12.02   50.41  185.20    0.07    0.03    0.
     0.      0.16    0.06]
per-feature maximum before scaling:
 [   28.11    39.28   188.5   2501.0     0.16     0.29     0.43     0.2
     0.300    0.100    2.87     4.88    21.98   542.20     0.03     0.14
     0.400    0.050    0.06     0.03    36.04    49.54   251.20  4254.00
     0.220    0.940    1.17     0.29     0.58     0.15]
per-feature minimum after scaling:
 [ 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.]
per-feature maximum after scaling:
 [ 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.]
```

The transformed data has the same shape as the original data—the features are simply shifted and scaled. You can see that all of the features are now between 0 and 1, as desired.

To apply the SVM to the scaled data, we also need to transform the test set. This is again done by calling the `transform` method, this time on `X_test`:

In[7]:

```
# transform test data
X_test_scaled = scaler.transform(X_test)
# print test data properties after scaling
print("per-feature minimum after scaling:\n{}".format(X_test_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n{}".format(X_test_scaled.max(axis=0)))
```

Out[7]:

```
per-feature minimum after scaling:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.     0.     0.154 -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.     0.    -0.032  0.007
  0.027  0.058  0.02   0.009  0.109  0.026  0.     0.    -0.    -0.002]
per-feature maximum after scaling:
[ 0.958  0.815  0.956  0.894  0.811  1.22   0.88   0.933  0.932  1.037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337  0.391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07   0.924  1.205  1.631]
```

Maybe somewhat surprisingly, you can see that for the test set, after scaling, the minimum and maximum are not 0 and 1. Some of the features are even outside the 0–1 range! The explanation is that the `MinMaxScaler` (and all the other scalers) always applies exactly the same transformation to the training and the test set. This means the `transform` method always subtracts the training set minimum and divides by the training set range, which might be different from the minimum and range for the test set.

Scaling Training and Test Data the Same Way

It is important to apply exactly the same transformation to the training set and the test set for the supervised model to work on the test set. The following example (Figure 3-2) illustrates what would happen if we were to use the minimum and range of the test set instead:

In[8]:

```
from sklearn.datasets import make_blobs
# make synthetic data
X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
# split it into training and test sets
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)

# plot the training and test sets
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
```