The first panel is an unscaled two-dimensional dataset, with the training set shown as circles and the test set shown as triangles. The second panel is the same data, but scaled using the `MinMaxScaler`. Here, we called `fit` on the training set, and then called `transform` on the training and test sets. You can see that the dataset in the second panel looks identical to the first; only the ticks on the axes have changed. Now all the features are between 0 and 1. You can also see that the minimum and maximum feature values for the test data (the triangles) are not 0 and 1.

The third panel shows what would happen if we scaled the training set and test set separately. In this case, the minimum and maximum feature values for both the training and the test set are 0 and 1. But now the dataset looks different. The test points moved incongruously to the training set, as they were scaled differently. We changed the arrangement of the data in an arbitrary way. Clearly this is not what we want to do.

As another way to think about this, imagine your test set is a single point. There is no way to scale a single point correctly, to fulfill the minimum and maximum requirements of the `MinMaxScaler`. But the size of your test set should not change your processing.

Shortcuts and Efficient Alternatives

Often, you want to `fit` a model on some dataset, and then `transform` it. This is a very common task, which can often be computed more efficiently than by simply calling `fit` and then `transform`. For this use case, all models that have a `transform` method also have a `fit_transform` method. Here is an example using `StandardScaler`:

`In[9]:`

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# calling fit and transform in sequence (using method chaining)
X_scaled = scaler.fit(X).transform(X)
# same result, but more efficient computation
X_scaled_d = scaler.fit_transform(X)
```

While `fit_transform` is not necessarily more efficient for all models, it is still good practice to use this method when trying to transform the training set.

The Effect of Preprocessing on Supervised Learning

Now let's go back to the `cancer` dataset and see the effect of using the `MinMaxScaler` on learning the SVC (this is a different way of doing the same scaling we did in Chapter 2). First, let's fit the SVC on the original data again for comparison:

In[10]:

```
from sklearn.svm import SVC

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

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Test set accuracy: {:.2f}".format(svm.score(X_test, y_test)))
```

Out[10]:

```
Test set accuracy: 0.63
```

Now, let's scale the data using `MinMaxScaler` before fitting the `SVC`:

In[11]:

```
# preprocessing using 0-1 scaling
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# learning an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)

# scoring on the scaled test set
print("Scaled test set accuracy: {:.2f}".format(
    svm.score(X_test_scaled, y_test)))
```

Out[11]:

```
Scaled test set accuracy: 0.97
```

As we saw before, the effect of scaling the data is quite significant. Even though scaling the data doesn't involve any complicated math, it is good practice to use the scaling mechanisms provided by `scikit-learn` instead of reimplementing them yourself, as it's easy to make mistakes even in these simple computations.

You can also easily replace one preprocessing algorithm with another by changing the class you use, as all of the preprocessing classes have the same interface, consisting of the `fit` and `transform` methods:

In[12]:

```
# preprocessing using zero mean and unit variance scaling
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# learning an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)

# scoring on the scaled test set
print("SVM test accuracy: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

Out[12]:

```
SVM test accuracy: 0.96
```

Now that we've seen how simple data transformations for preprocessing work, let's move on to more interesting transformations using unsupervised learning.

Dimensionality Reduction, Feature Extraction, and Manifold Learning

As we discussed earlier, transforming data using unsupervised learning can have many motivations. The most common motivations are visualization, compressing the data, and finding a representation that is more informative for further processing.

One of the simplest and most widely used algorithms for all of these is principal component analysis. We'll also look at two other algorithms: non-negative matrix factorization (NMF), which is commonly used for feature extraction, and t-SNE, which is commonly used for visualization using two-dimensional scatter plots.

Principal Component Analysis (PCA)

Principal component analysis is a method that rotates the dataset in a way such that the rotated features are statistically uncorrelated. This rotation is often followed by selecting only a subset of the new features, according to how important they are for explaining the data. The following example (Figure 3-3) illustrates the effect of PCA on a synthetic two-dimensional dataset:

In[13]:

```
mglearn.plots.plot_pca_illustration()
```

The first plot (top left) shows the original data points, colored to distinguish among them. The algorithm proceeds by first finding the direction of maximum variance, labeled "Component 1." This is the direction (or vector) in the data that contains most of the information, or in other words, the direction along which the features are most correlated with each other. Then, the algorithm finds the direction that contains the most information while being orthogonal (at a right angle) to the first direction. In two dimensions, there is only one possible orientation that is at a right angle, but in higher-dimensional spaces there would be (infinitely) many orthogonal directions. Although the two components are drawn as arrows, it doesn't really matter where the head and the tail are; we could have drawn the first component from the center up to