

In[2]:

```
# rescale the training data
X_train_scaled = scaler.transform(X_train)

svm = SVC()
# learn an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)
# scale the test data and score the scaled data
X_test_scaled = scaler.transform(X_test)
print("Test score: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

Out[2]:

```
Test score: 0.95
```

Parameter Selection with Preprocessing

Now let's say we want to find better parameters for SVC using GridSearchCV, as discussed in [Chapter 5](#). How should we go about doing this? A naive approach might look like this:

In[3]:

```
from sklearn.model_selection import GridSearchCV
# for illustration purposes only, don't use this code!
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=5)
grid.fit(X_train_scaled, y_train)
print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
print("Best set score: {:.2f}".format(grid.score(X_test_scaled, y_test)))
print("Best parameters: ", grid.best_params_)
```

Out[3]:

```
Best cross-validation accuracy: 0.98
Best set score: 0.97
Best parameters: {'gamma': 1, 'C': 1}
```

Here, we ran the grid search over the parameters of SVC using the scaled data. However, there is a subtle catch in what we just did. When scaling the data, we used *all the data in the training set* to find out how to train it. We then use the *scaled training data* to run our grid search using cross-validation. For each split in the cross-validation, some part of the original training set will be declared the training part of the split, and some the test part of the split. The test part is used to measure what new data will look like to a model trained on the training part. However, we already used the information contained in the test part of the split, when scaling the data. Remember that the test part in each split in the cross-validation is part of the training set, and we used the information from the entire training set to find the right scaling of the data.

This is fundamentally different from how new data looks to the model. If we observe new data (say, in form of our test set), this data will not have been used to scale the training data, and it might have a different minimum and maximum than the training data. The following example (Figure 6-1) shows how the data processing during cross-validation and the final evaluation differ:

In[4]:

```
mglearn.plots.plot_improper_processing()
```

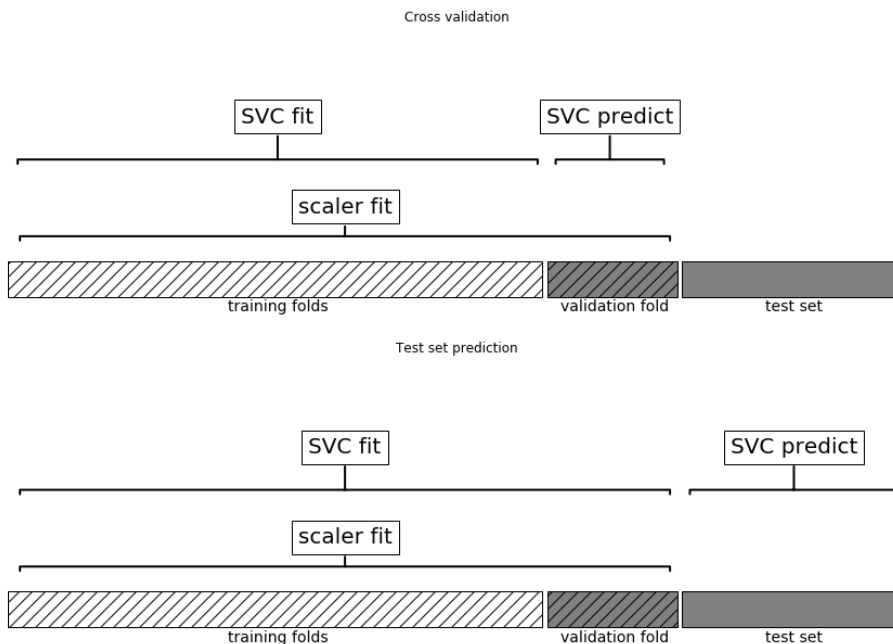


Figure 6-1. Data usage when preprocessing outside the cross-validation loop

So, the splits in the cross-validation no longer correctly mirror how new data will look to the modeling process. We already leaked information from these parts of the data into our modeling process. This will lead to overly optimistic results during cross-validation, and possibly the selection of suboptimal parameters.

To get around this problem, the splitting of the dataset during cross-validation should be done *before doing any preprocessing*. Any process that extracts knowledge from the dataset should only ever be applied to the training portion of the dataset, so any cross-validation should be the “outermost loop” in your processing.

To achieve this in `scikit-learn` with the `cross_val_score` function and the `Grid SearchCV` function, we can use the `Pipeline` class. The `Pipeline` class is a class that allows “gluing” together multiple processing steps into a single `scikit-learn` estima-

tor. The Pipeline class itself has `fit`, `predict`, and `score` methods and behaves just like any other model in `scikit-learn`. The most common use case of the Pipeline class is in chaining preprocessing steps (like scaling of the data) together with a supervised model like a classifier.

Building Pipelines

Let's look at how we can use the Pipeline class to express the workflow for training an SVM after scaling the data with `MinMaxScaler` (for now without the grid search). First, we build a pipeline object by providing it with a list of steps. Each step is a tuple containing a name (any string of your choosing¹) and an instance of an estimator:

In[5]:

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])
```

Here, we created two steps: the first, called "scaler", is an instance of `MinMaxScaler`, and the second, called "svm", is an instance of `SVC`. Now, we can fit the pipeline, like any other `scikit-learn` estimator:

In[6]:

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

Here, `pipe.fit` first calls `fit` on the first step (the scaler), then transforms the training data using the scaler, and finally fits the SVM with the scaled data. To evaluate on the test data, we simply call `pipe.score`:

In[7]:

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

Out[7]:

```
Test score: 0.95
```

Calling the `score` method on the pipeline first transforms the test data using the scaler, and then calls the `score` method on the SVM using the scaled test data. As you can see, the result is identical to the one we got from the code at the beginning of the chapter, when doing the transformations by hand. Using the pipeline, we reduced the code needed for our “preprocessing + classification” process. The main benefit of using the pipeline, however, is that we can now use this single estimator in `cross_val_score` or `GridSearchCV`.

¹ With one exception: the name can't contain a double underscore, `__`.