CHAPTER 7
# Algorithm Chains and Pipelines

For many machine learning algorithms, the particular representation of the data that you provide is very important, as we discussed in Chapter 5. This starts with scaling the data and combining features by hand and goes all the way to learning features using unsupervised machine learning as we saw in Chapter 3.

Consequently, most machine learning applications require not only the application of a single algorithm, but the chaining together of many different processing steps and machine learning models.

For example we noticed that we can greatly improve the performance of a kernel SVM on the `cancer` dataset by using the `MinMaxScaler` for preprocessing. The code for splitting the data, computing minimum and maximum, scaling the data, and training the SVM is shown below:

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# load and split the data
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

# compute minimum and maximum on the training data
scaler = MinMaxScaler().fit(X_train)
# rescale 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 test data and score the scaled data
X_test_scaled = scaler.transform(X_test)
svm.score(X_test_scaled, y_test)
```

```
0.95104895104895104
```

## Parameter Selection with Preprocessing

Now let's say we want to find better parameters for `SVC` using `GridSearchCV`, as discussed in Chapter 6.

How should we go about doing this? A naive approach might look like this:

```
from sklearn.model_selection import GridSearchCV
# 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:", grid.best_score_)
print("test set score: ", grid.score(X_test_scaled, y_test))
print("best parameters: ", grid.best_params_)
```

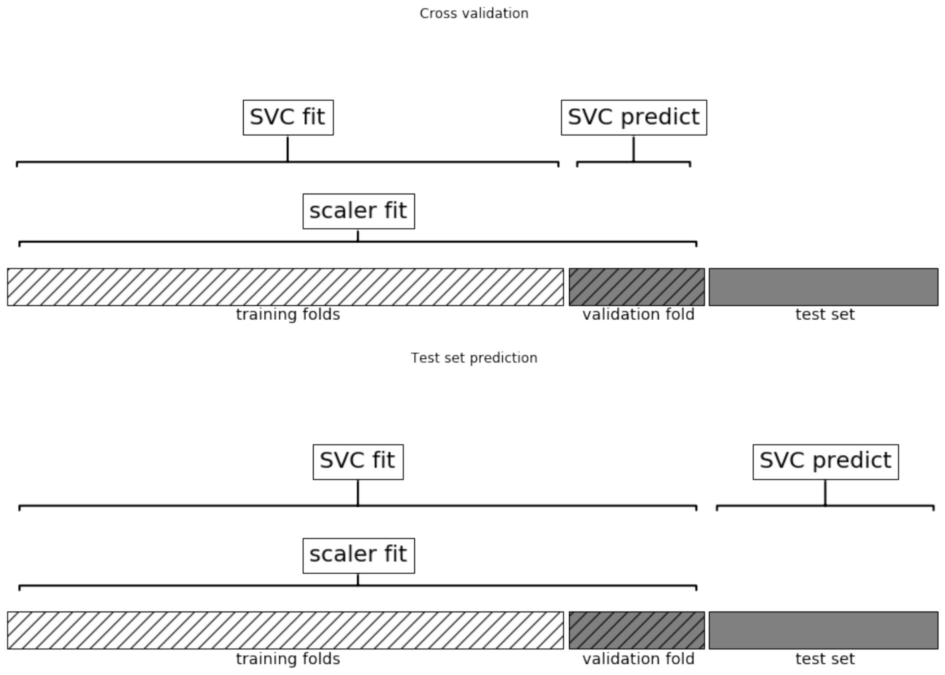
```
best cross-validation accuracy: 0.981220657277

test set score:  0.972027972028

best parameters:  {'gamma': 1, 'C': 1}
```

Here, we ran the grid-search over the parameters of the `SVC` using the scaled data. However, there is a subtle catch in what we just did. When scaling the data, we used *all 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 this split, and some the test part of the split. The test part is used to measure how 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 whole training data* to find the right scaling of the data. *This is fundamentally different to 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. This data might have a different minimum and maximum than the training data.

The illustration below shows how the data processing during cross-validation and the final evaluation differ:

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



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 data set 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 data set, 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 `GridSearchCV` 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 estimator. 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` to express the work-flow for training an SVM after scaling the data `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 choosing [Footnote: With one exception: the name may not contain a double underscore "__".]) and an instance of an estimator:

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

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

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

```
Pipeline(steps=[('scaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('svm', SVC(C=1.0, cach

  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',

  max_iter=-1, probability=False, random_state=None, shrinking=True,

  tol=0.001, verbose=False))])
```

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:

```
pipe.score(X_test, y_test)
```

```
0.95104895104895104
```

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 above, 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.

## Using Pipelines in Grid-searches

Using a pipeline in a grid-search works the same way as using any other estimator. We define a parameter grid to search over, and construct a GridSearchCV from the pipeline and the parameter grid. When specifying the parameter grid, there is a slight change, though. We need to specify for each parameter which step of the pipeline it belongs to.

Both parameters that we want to adjust, C and gamma are parameters of SVC, the second step. We gave this step the name "svm". The syntax to define the a parameter grid for a pipeline is to specify for each parameter the step name, followed by "__" (double underscore), followed by the parameter name. To search over the C parameter of the SVC we therefore have to use "svm__C" as the key in the parameter grid dictionary, and similarly for gamma:

```
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

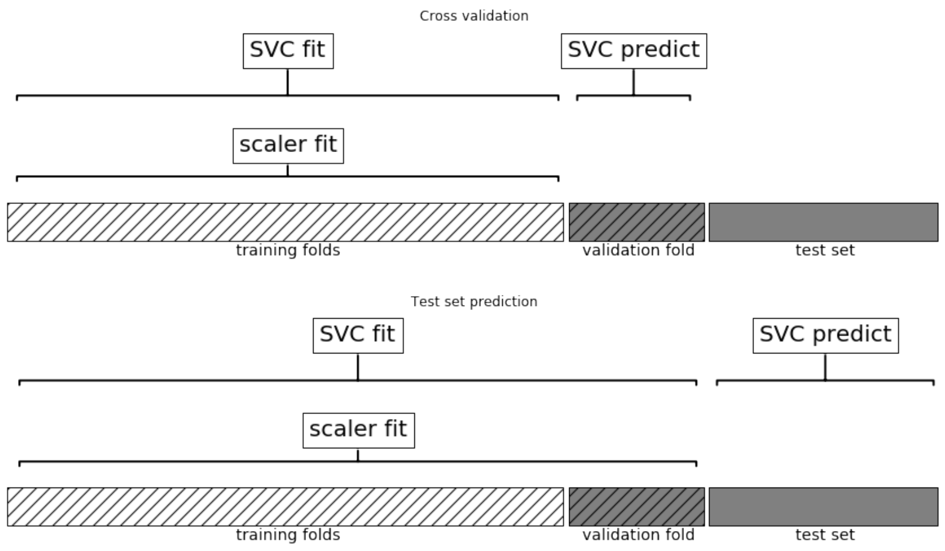
Using this parameter grid we can use GridSearchCV as usual:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
print("best cross-validation accuracy:", grid.best_score_)
print("test set score: ", grid.score(X_test, y_test))
print("best parameters: ", grid.best_params_)
```

```
best cross-validation accuracy: 0.981220657277

test set score:  0.972027972028

best parameters:  {'svm__C': 1, 'svm__gamma': 1}
```

In contrast to the grid-search we did before, now for each split in the cross-validation, the MinMaxScaler is refit with only the training splits, not leaking any information of the test split into the parameter search, as illustrated below. Compare this with Figure improper_preprocessing above.

```
mglearn.plots.plot_proper_processing()
```



The impact of leaking information in the cross-validation varies depending on the nature of the preprocessing step. Estimating the scale of the data using the test fold usually doesn't have a terrible impact, while using the test fold in feature extraction and feature selection can lead to substantial differences in outcomes.

[FIXME info box] Illustrating information leakage

A great example of leaking information in cross-validation is given in Hastie et al. (FIXME insert cite) and we reproduce an adapted version here.

Let us consider a synthetic regression task with 100 samples and 1000 features that are sampled independently from a Gaussian distribution. We also sample the response from a Gaussian distribution:

```
rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))
```

Given the way we created the dataset, there is no relation between the data X and the target y (they are independent), so it should not be possible to learn anything from this data set.

We will now do the following: First select the most informative of the ten features using `SelectPercentile` feature selection, and then evaluate a `Ridge` regressor using cross-validation:

```
from sklearn.feature_selection import SelectPercentile, f_regression

select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
X_selected = select.transform(X)
print(X_selected.shape)
```

```
(100, 500)
```

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge
np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))
```

```
0.90579530652398221
```

The mean $R^2$ computed by cross-validation is 0.9, indicating a very good model. This can clearly not be right, as our data is entirely random. What happened here is that our feature selection picked out some features among the 10000 random features that are (by chance) very well correlated with the target. Because we fit the feature selection *outside* of the cross-validation, it could find features that are correlated both on the training and the test folds. The information we leaked from the test-folds was very informative, leading to highly unrealistic results.

Let's compare this to a proper cross-validation using a pipeline:

```
pipe = Pipeline([("select", SelectPercentile(score_func=f_regression, percentile=5)), ("ridge", Ri
np.mean(cross_val_score(pipe, X, y, cv=5))
```

```
-0.24655422384952805
```

This time, we get a *negative* $R^2$ score, indicating a very poor model.

Using the pipeline, the feature selection is now *inside* the cross-validation loop. This means features can only be selected using the training folds of the data, not the test fold. The feature selection finds features that are correlated with the target on the training set. But because the data is entirely random, these features are not correlated with the target on the test set.

In this example, rectifying the data leakage issue in the feature selection makes the difference between concluding that a model works very well and concluding that a model works not at all.

[end infobox]

## The General Pipeline Interface

The `Pipeline` class is not restricted to preprocessing and classification, but can in fact join any number of estimators together.

For example, you could build a pipeline containing feature extraction, feature selection, scaling and classification, for a total of four steps. Similarly the last step could be regression or clustering instead of classification.

The only requirement for estimators in a pipeline is that all but the last step need to have a `transform` method, so they can produce a new representation of the data that can be used in the next step.

Internally, during the call to `Pipeline.fit`, the pipeline calls first `fit` and then `transform` on each step in turn [Footnote: or just `fit_transform`], with the input given by the output of the transform method of the previous step. For the last step in the pipeline, just `fit` is called. Brushing over some finer details, this is implemented as follows. Remember that `pipeline.steps` is a list of tuples, so `pipeline.steps[0][1]` is the first estimator, `pipeline.steps[1][1]` is the second estimator, and so on.

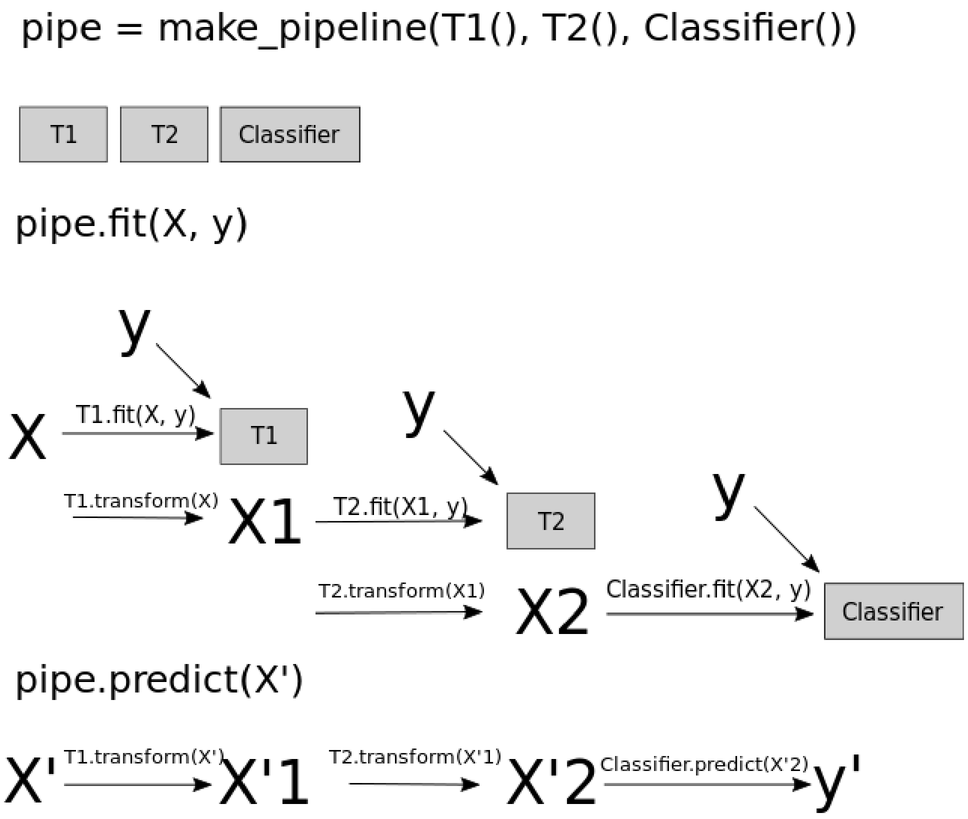
```
def fit(self, X, y):
    X_transformed = X
    for step in self.steps[:-1]:
        # iterate over all but the final step
        # fit and transform the data
        X_transformed = step[1].fit_transform(X_transformed, y)
    # fit the last step
    self.steps[-1][1].fit(X_transformed, y)
    return self
```

When predicting using `Pipeline`, similarly we `transform` the data using all but the last step, and then call `predict` on the last step:

```
def predict(self, X):
    X_transformed = X
```

```
        for step in self.steps[:-1]:
            # iterate over all but the final step
            # transform the data
            X_transformed = step[1].transform(X_transformed)
        # fit the last step
        return self.steps[-1][1].predict(X_transformed)
```

The process is illustrated below for two transformers T1 and T2 and a classifier Clf:



The pipeline is actually even more general than this. There is no requirement for the last step in a pipeline to have a `predict` function, and we could create a pipeline just containing, for example, a scaler and `PCA`. Then, because the last step `PCA` has a `trans form` method, we could call `transform` on the pipeline to get the output of `PCA.trans form` applied to the data that was processed by the previous step. The last step of a pipeline is only required to have a `fit` method.

## Convenient Pipeline creation with `make_pipeline`

Creating a Pipeline using the syntax described above is sometimes a bit cumbersome, and we often don't need user-specified names for each step. There is a convenience function `make_pipeline` that will create a pipeline for us and automatically name each step based on its class. The syntax for `make_pipeline` is as follows:

```
from sklearn.pipeline import make_pipeline
# standard syntax
pipe_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])
# abbreviated syntax
pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
```

The pipeline objects `pipe_long` and `pipe_short` do exactly the same, only that `pipe_short` has steps that were automatically named. We can see the name of the steps by looking at the `steps` attribute:

```
pipe_short.steps
```

```
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))),

 ('svc', SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,

    decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',

    max_iter=-1, probability=False, random_state=None, shrinking=True,

    tol=0.001, verbose=False))]
```

The steps are named `minmaxscaler` and `svc`. In general the step names are just lowercase version of the class names. If multiple steps have the same class, a number is appended:

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())
pipe.steps
```

```
[('standardscaler-1',

  StandardScaler(copy=True, with_mean=True, with_std=True)),

 ('pca', PCA(copy=True, iterated_power=4, n_components=2, random_state=None,

    svd_solver='auto', tol=0.0, whiten=False)),

 ('standardscaler-2',

  StandardScaler(copy=True, with_mean=True, with_std=True))]
```

As you can see, the first `StandardScaler` was named `"standardscaler-1"` and the second `"standardscaler-2"`. However, in such settings it might be better to use the `Pipeline` construction with explicit names, to give more semantic names to each step.

### Accessing step attributes

Often you might want to inspect attributes of one of the steps of the pipeline, say the coefficients of a linear model or the components extracted by `PCA`. The easiest way to access the step in a pipeline is the `named_steps` attribute, which is a dictionary from step names to the estimators:

```
# fit the pipeline defined above to the cancer dataset
pipe.fit(cancer.data)
# extract the first two principal components from the "pca" step
components = pipe.named_steps["pca"].components_
print(components.shape)
```

```
/home/andy/checkout/scikit-learn/sklearn/utils/extmath.py:368: UserWarning: The number of power it

  warnings.warn("The number of power iterations is increased to "
```

### Accessing attributes in grid-searched pipeline.

As we discussed above, one of the main reasons to use pipelines is for doing grid-searches. A common task then is to access some of the steps of a pipeline inside a grid-search.

Let's grid-search a `LogisticRegression` classifier on the `cancer` dataset, using `Pipeline` and `StandardScaler` to scale the data before passing it to the `LogisticRegression` classifier.

First we create a pipeline using the `make_pipeline` function:

```
from sklearn.linear_model import LogisticRegression

pipe = make_pipeline(StandardScaler(), LogisticRegression())
```

Next, we create a parameter grid. The regularization parameter to tune for `LogisticRegression` is the parameter `C` as explained in Chapter 2. We use a logarithmic grid for this parameter, searching between 0.01 and 100. Because we used the `make_pipeline` function, the name of the `LogisticRegression` step in the pipeline is the lower-cased class-name `"logisticregression"`. To tune the parameter C, we therefore have to specify a parameter grid for `"logisticregression__C"`:

```
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}
```

We split the cancer dataset into training and test set, and fit a grid-search as usual:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=4)
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, error_score='raise',

       estimator=Pipeline(steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False))]),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'logisticregression__C': [0.01, 0.1, 1, 10, 100]},
       pre_dispatch='2*n_jobs', refit=True, scoring=None, verbose=0)
```

So how do we access the coefficients of the best `LogisticRegression` model that was found by `GridSearchCV`? From Chapter 6 we know that the best model found by `GridSearchCV`, trained on all the training data, is stored in `grid.best_estimator_`:

```
print(grid.best_estimator_)
Pipeline(steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('lo
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False))])
```

This `best_estimator_` in our case is a pipeline with two steps, `"standardscaler"` and `"logisticregression"`. To access the `logisticregression` step, we can use the `named_steps` attribute of the pipeline that we explained above:

```
print(grid.best_estimator_.named_steps["logisticregression"])
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

Now that we have the trained `LogisticRegression` instance, we can access the coefficients (weights) associated with each input feature:

```
print(grid.best_estimator_.named_steps["logisticregression"].coef_)
[[-0.389 -0.375 -0.376 -0.396 -0.115  0.017 -0.355 -0.39  -0.058  0.209
  -0.495 -0.004 -0.371 -0.383 -0.045  0.198  0.004 -0.049  0.21   0.224
  -0.547 -0.525 -0.499 -0.515 -0.393 -0.123 -0.388 -0.417 -0.325 -0.139]]
```

This might be a somewhat lengthy expression, but often comes in handy in understanding your models.

## Grid-searching preprocessing steps and model parameters

Using pipelines, we can encapsulate all processing steps in our machine learning work flow in a single scikit-learn estimator. Another benefit of doing this is that we can now *adjust the parameters of the preprocessing* using the outcome of a supervised task like regression or classification.

In previous chapters, we used polynomial features on the `boston` dataset before applying the ridge regressor. Let's model that using a pipeline. The pipeline contains three steps: scaling the data, computing polynomial features, and ridge regression:

```
from sklearn.datasets import load_boston
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target, random_state=0)

from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(),
    Ridge())
```

But how do we know which degree of polynomials to choose, or whether to choose any polynomials or interactions at all? Ideally we want to select the `degree` parameter based on the outcome of the classification.

Using our pipeline, we can search over the `degree` parameter together with the parameter `alpha` of `Ridge`. To do this, we define a `param_grid` that contains both, appropriately prefixed by the step names:

```
param_grid = {'polynomialfeatures__degree': [1, 2, 3],
              'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

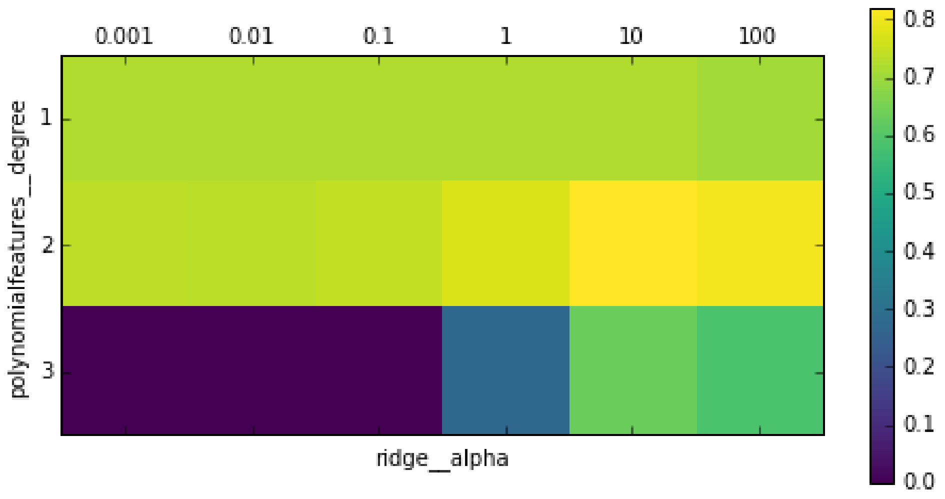
Now we can run our grid-search again:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, error_score='raise',

       estimator=Pipeline(steps=[('standardscaler', StandardScaler(copy=True, with_mean=True, with

   normalize=False, random_state=None, solver='auto', tol=0.001))]),

       fit_params={}, iid=True, n_jobs=-1,

       param_grid={'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100], 'polynomialfeatures__degree': [

       pre_dispatch='2*n_jobs', refit=True, scoring=None, verbose=0)
```

We can visualize the outcome of the cross-validation using a heatmap, as we did in Chapter 6:

```
plt.matshow(np.array([s.mean_validation_score for s in grid.grid_scores_]).reshape(3, -1),
            vmin=0, cmap="viridis")
plt.xlabel("ridge__alpha")
plt.ylabel("polynomialfeatures__degree")
plt.xticks(range(len(param_grid['ridge__alpha'])), param_grid['ridge__alpha'])
plt.yticks(range(len(param_grid['polynomialfeatures__degree'])), param_grid['polynomialfeatures__c

plt.colorbar()
```



Looking at the results produced by the cross-validation, we can see that using polynomials of degree two helps, but that degree three polynomials are much worse than either degree one or two.

This is reflected in the best parameters that were found:

```
print(grid.best_params_)
```

```
{'ridge__alpha': 10, 'polynomialfeatures__degree': 2}
```

Which lead to the following score:

```
grid.score(X_test, y_test)
```

```
0.76735803503061784
```

Let’s run a grid-search without polynomial features for comparison:

```
param_grid = {'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = make_pipeline(StandardScaler(), Ridge())
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
grid.score(X_test, y_test)
```

```
0.62717803817745799
```

As we had expected from the grid-search results visualized above, using no polynomial features leads to decidedly worse results. Searching over preprocessing parameters together with model parameters is a very powerful strategy. However, keep in mind that `GridSearchCV` tries *all possible combinations* of the specified parameters. Adding more parameters to your grid therefore increases the number of models that need to be built exponentially.

## Summary and Outlook

In this chapter we introduced the `Pipeline` class a general purpose tool to chain together multiple processing steps in a machine learning work flow. Real-world applications of machine learning are rarely an isolated use of a model, and instead a sequence of processing steps. Using pipelines allows us to encapsulate multiple steps into a single python object that adheres to the familiar scikit-learn interface of `fit`, `predict` and `transform`.

In particular when doing model evaluation using cross-validation and parameter selection using grid-search, using the `Pipeline` class to capture all processing steps is essential for proper evaluation.

The `Pipeline` class also allows writing more succinct code, and reduces the likelihood of mistakes that can happen when building processing chains without the pipeline class (like forgetting to apply all transformers on the test set, or maybe not applying them in the right order).

Choosing the right combination of feature extraction, preprocessing and models is somewhat of an art, that often requires some trial-and-error. However, using pipelines, this "trying out" of many different processing steps is quite simple. When experimenting, be careful not to over-complicate your processes, and make sure to evaluate whether every component your are including in your model is necessary.

With this chapter, we complete our survey of general purpose tools and algorithms provided by scikit-learn. You now possess all the required skills and know the necessary mechanisms to apply machine learning in practice. In the next chapter, we will dive in more detail into one particular type of data that is commonly seen in practice, and that requires some special expertise to handle correctly: text data.