```
    # rebuild a model on the combined training and validation set,
    # and evaluate it on the test set
    svm = SVC(**best_parameters)
    svm.fit(X_trainval, y_trainval)
    test_score = svm.score(X_test, y_test)
    print("Best score on validation set: {:.2f}".format(best_score))
    print("Best parameters: ", best_parameters)
    print("Test set score with best parameters: {:.2f}".format(test_score))
```

Out[20]:

```
    Size of training set: 84   size of validation set: 28   size of test set: 38

    Best score on validation set: 0.96
    Best parameters:  {'C': 10, 'gamma': 0.001}
    Test set score with best parameters: 0.92
```

The best score on the validation set is 96%: slightly lower than before, probably because we used less data to train the model (`X_train` is smaller now because we split our dataset twice). However, the score on the test set—the score that actually tells us how well we generalize—is even lower, at 92%. So we can only claim to classify new data 92% correctly, not 97% correctly as we thought before!

The distinction between the training set, validation set, and test set is fundamentally important to applying machine learning methods in practice. Any choices made based on the test set accuracy "leak" information from the test set into the model. Therefore, it is important to keep a separate test set, which is only used for the final evaluation. It is good practice to do all exploratory analysis and model selection using the combination of a training and a validation set, and reserve the test set for a final evaluation—this is even true for exploratory visualization. Strictly speaking, evaluating more than one model on the test set and choosing the better of the two will result in an overly optimistic estimate of how accurate the model is.

Grid Search with Cross-Validation

While the method of splitting the data into a training, a validation, and a test set that we just saw is workable, and relatively commonly used, it is quite sensitive to how exactly the data is split. From the output of the previous code snippet we can see that `GridSearchCV` selects `'C': 10, 'gamma': 0.001` as the best parameters, while the output of the code in the previous section selects `'C': 100, 'gamma': 0.001` as the best parameters. For a better estimate of the generalization performance, instead of using a single split into a training and a validation set, we can use cross-validation to evaluate the performance of each parameter combination. This method can be coded up as follows:

In[21]:

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters,
        # train an SVC
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
# rebuild a model on the combined training and validation set
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

To evaluate the accuracy of the SVM using a particular setting of `C` and `gamma` using five-fold cross-validation, we need to train 36 * 5 = 180 models. As you can imagine, the main downside of the use of cross-validation is the time it takes to train all these models.

The following visualization (Figure 5-6) illustrates how the best parameter setting is selected in the preceding code:

In[22]:

```
mglearn.plots.plot_cross_val_selection()
```

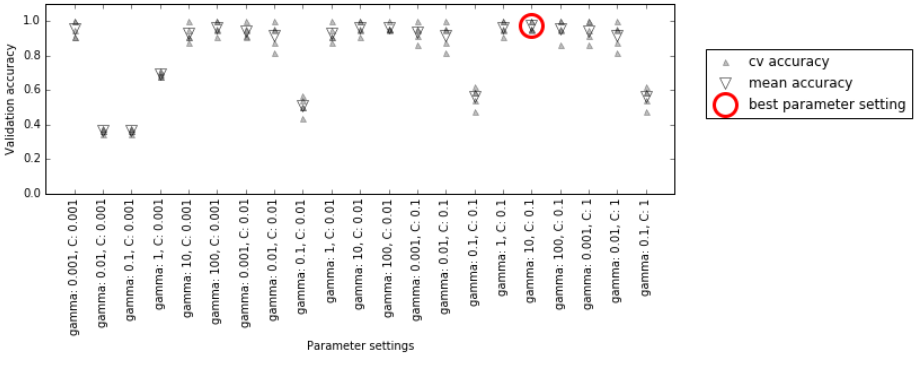

Figure 5-6. Results of grid search with cross-validation

For each parameter setting (only a subset is shown), five accuracy values are computed, one for each split in the cross-validation. Then the mean validation accuracy is computed for each parameter setting. The parameters with the highest mean validation accuracy are chosen, marked by the circle.

As we said earlier, cross-validation is a way to evaluate a given algorithm on a specific dataset. However, it is often used in conjunction with parameter search methods like grid search. For this reason, many people use the term *cross-validation* colloquially to refer to grid search with cross-validation.

The overall process of splitting the data, running the grid search, and evaluating the final parameters is illustrated in Figure 5-7:

In[23]:

```
mglearn.plots.plot_grid_search_overview()
```

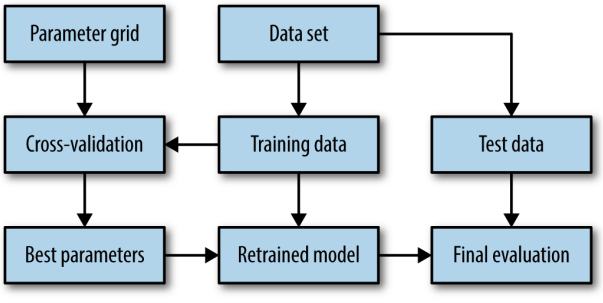

Figure 5-7. Overview of the process of parameter selection and model evaluation with GridSearchCV

Because grid search with cross-validation is such a commonly used method to adjust parameters, `scikit-learn` provides the `GridSearchCV` class, which implements it in the form of an estimator. To use the `GridSearchCV` class, you first need to specify the parameters you want to search over using a dictionary. `GridSearchCV` will then perform all the necessary model fits. The keys of the dictionary are the names of parameters we want to adjust (as given when constructing the model—in this case, `C` and `gamma`), and the values are the parameter settings we want to try out. Trying the values `0.001`, `0.01`, `0.1`, `1`, `10`, and `100` for `C` and `gamma` translates to the following dictionary:

In[24]:

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))
```

Out[24]:

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

We can now instantiate the `GridSearchCV` class with the model (`SVC`), the parameter grid to search (`param_grid`), and the cross-validation strategy we want to use (say, five-fold stratified cross-validation):

In[25]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

`GridSearchCV` will use cross-validation in place of the split into a training and validation set that we used before. However, we still need to split the data into a training and a test set, to avoid overfitting the parameters:

In[26]:

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

The `grid_search` object that we created behaves just like a classifier; we can call the standard methods `fit`, `predict`, and `score` on it.[1] However, when we call `fit`, it will run cross-validation for each combination of parameters we specified in `param_grid`:

In[27]:

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

Fitting the `GridSearchCV` object not only searches for the best parameters, but also automatically fits a new model on the whole training dataset with the parameters that yielded the best cross-validation performance. What happens in `fit` is therefore equivalent to the result of the In[21] code we saw at the beginning of this section. The `GridSearchCV` class provides a very convenient interface to access the retrained model using the `predict` and `score` methods. To evaluate how well the best found parameters generalize, we can call `score` on the test set:

In[28]:

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

Out[28]:

```
Test set score: 0.97
```

Choosing the parameters using cross-validation, we actually found a model that achieves 97% accuracy on the test set. The important thing here is that we *did not use the test set* to choose the parameters. The parameters that were found are scored in the

1 A `scikit-learn` estimator that is created using another estimator is called a *meta-estimator*. `GridSearchCV` is the most commonly used meta-estimator, but we will see more later.

best_params_ attribute, and the best cross-validation accuracy (the mean accuracy over the different splits for this parameter setting) is stored in best_score_:

In[29]:

```
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

Out[29]:

```
Best parameters: {'C': 100, 'gamma': 0.01}
Best cross-validation score: 0.97
```

Again, be careful not to confuse best_score_ with the generalization performance of the model as computed by the score method on the test set. Using the score method (or evaluating the output of the predict method) employs a model *trained on the whole training set*. The best_score_ attribute stores the mean cross-validation accuracy, with *cross-validation performed on the training set*.

Sometimes it is helpful to have access to the actual model that was found—for example, to look at coefficients or feature importances. You can access the model with the best parameters trained on the whole training set using the best_estimator_ attribute:

In[30]:

```
print("Best estimator:\n{}".format(grid_search.best_estimator_))
```

Out[30]:

```
Best estimator:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
   decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',
   max_iter=-1, probability=False, random_state=None, shrinking=True,
   tol=0.001, verbose=False)
```

Because grid_search itself has predict and score methods, using best_estimator_ is not needed to make predictions or evaluate the model.

Analyzing the result of cross-validation

It is often helpful to visualize the results of cross-validation, to understand how the model generalization depends on the parameters we are searching. As grid searches are quite computationally expensive to run, often it is a good idea to start with a relatively coarse and small grid. We can then inspect the results of the cross-validated grid search, and possibly expand our search. The results of a grid search can be found in the cv_results_ attribute, which is a dictionary storing all aspects of the search. It

contains a lot of details, as you can see in the following output, and is best looked at after converting it to a pandas DataFrame:

In[31]:

```
import pandas as pd
# convert to DataFrame
results = pd.DataFrame(grid_search.cv_results_)
# show the first 5 rows
display(results.head())
```

Out[31]:

```
    param_C  param_gamma  params                        mean_test_score
0   0.001    0.001        {'C': 0.001, 'gamma': 0.001}  0.366
1   0.001    0.01         {'C': 0.001, 'gamma': 0.01}   0.366
2   0.001    0.1          {'C': 0.001, 'gamma': 0.1}    0.366
3   0.001    1            {'C': 0.001, 'gamma': 1}      0.366
4   0.001    10           {'C': 0.001, 'gamma': 10}     0.366

    rank_test_score  split0_test_score  split1_test_score  split2_test_score
0   22               0.375              0.347              0.363
1   22               0.375              0.347              0.363
2   22               0.375              0.347              0.363
3   22               0.375              0.347              0.363
4   22               0.375              0.347              0.363

    split3_test_score  split4_test_score  std_test_score
0   0.363              0.380              0.011
1   0.363              0.380              0.011
2   0.363              0.380              0.011
3   0.363              0.380              0.011
4   0.363              0.380              0.011
```

Each row in `results` corresponds to one particular parameter setting. For each setting, the results of all cross-validation splits are recorded, as well as the mean and standard deviation over all splits. As we were searching a two-dimensional grid of parameters (`C` and `gamma`), this is best visualized as a heat map (Figure 5-8). First we extract the mean validation scores, then we reshape the scores so that the axes correspond to `C` and `gamma`:

In[32]:

```
scores = np.array(results.mean_test_score).reshape(6, 6)

# plot the mean cross-validation scores
mglearn.tools.heatmap(scores, xlabel='gamma', xticklabels=param_grid['gamma'],
                      ylabel='C', yticklabels=param_grid['C'], cmap="viridis")
```

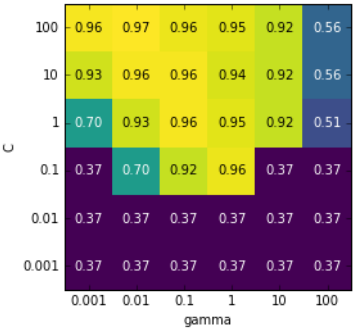

Figure 5-8. Heat map of mean cross-validation score as a function of C and gamma

Each point in the heat map corresponds to one run of cross-validation, with a particular parameter setting. The color encodes the cross-validation accuracy, with light colors meaning high accuracy and dark colors meaning low accuracy. You can see that `SVC` is very sensitive to the setting of the parameters. For many of the parameter settings, the accuracy is around 40%, which is quite bad; for other settings the accuracy is around 96%. We can take away from this plot several things. First, the parameters we adjusted are *very important* for obtaining good performance. Both parameters (`C` and `gamma`) matter a lot, as adjusting them can change the accuracy from 40% to 96%. Additionally, the ranges we picked for the parameters are ranges in which we see significant changes in the outcome. It's also important to note that the ranges for the parameters are large enough: the optimum values for each parameter are not on the edges of the plot.

Now let's look at some plots (shown in Figure 5-9) where the result is less ideal, because the search ranges were not chosen properly:

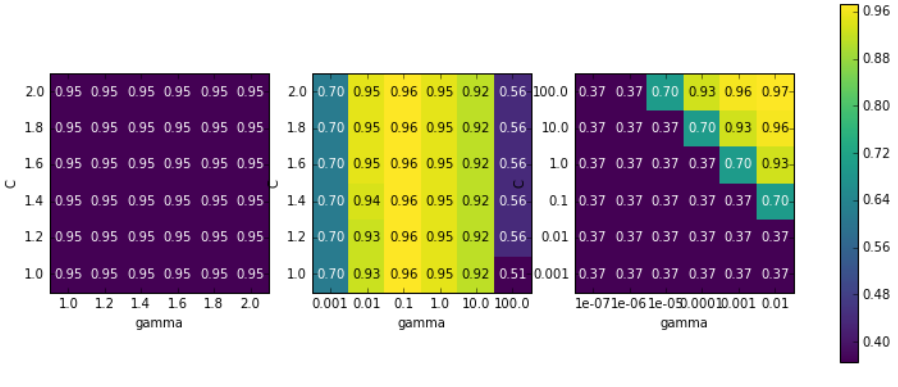

Figure 5-9. Heat map visualizations of misspecified search grids

In[33]:

```
fig, axes = plt.subplots(1, 3, figsize=(13, 5))

param_grid_linear = {'C': np.linspace(1, 2, 6),
                     'gamma':  np.linspace(1, 2, 6)}

param_grid_one_log = {'C': np.linspace(1, 2, 6),
                      'gamma':  np.logspace(-3, 2, 6)}

param_grid_range = {'C': np.logspace(-3, 2, 6),
                    'gamma':  np.logspace(-7, -2, 6)}

for param_grid, ax in zip([param_grid_linear, param_grid_one_log,
                           param_grid_range], axes):
    grid_search = GridSearchCV(SVC(), param_grid, cv=5)
    grid_search.fit(X_train, y_train)
    scores = grid_search.cv_results_['mean_test_score'].reshape(6, 6)

    # plot the mean cross-validation scores
    scores_image = mglearn.tools.heatmap(
        scores, xlabel='gamma', ylabel='C', xticklabels=param_grid['gamma'],
        yticklabels=param_grid['C'], cmap="viridis", ax=ax)

plt.colorbar(scores_image, ax=axes.tolist())
```

The first panel shows no changes at all, with a constant color over the whole parameter grid. In this case, this is caused by improper scaling and range of the parameters `C` and `gamma`. However, if no change in accuracy is visible over the different parameter settings, it could also be that a parameter is just not important at all. It is usually good to try very extreme values first, to see if there are any changes in the accuracy as a result of changing a parameter.

The second panel shows a vertical stripe pattern. This indicates that only the setting of the `gamma` parameter makes any difference. This could mean that the `gamma` parameter is searching over interesting values but the `C` parameter is not—or it could mean the `C` parameter is not important.

The third panel shows changes in both `C` and `gamma`. However, we can see that in the entire bottom left of the plot, nothing interesting is happening. We can probably exclude the very small values from future grid searches. The optimum parameter setting is at the top right. As the optimum is in the border of the plot, we can expect that there might be even better values beyond this border, and we might want to change our search range to include more parameters in this region.

Tuning the parameter grid based on the cross-validation scores is perfectly fine, and a good way to explore the importance of different parameters. However, you should not test different parameter ranges on the final test set—as we discussed earlier, eval-

uation of the test set should happen only once we know exactly what model we want to use.

Search over spaces that are not grids

In some cases, trying all possible combinations of all parameters as `GridSearchCV` usually does, is not a good idea. For example, `SVC` has a `kernel` parameter, and depending on which kernel is chosen, other parameters will be relevant. If `kernel='linear'`, the model is linear, and only the `C` parameter is used. If `kernel='rbf'`, both the `C` and `gamma` parameters are used (but not other parameters like `degree`). In this case, searching over all possible combinations of `C`, `gamma`, and `kernel` wouldn't make sense: if `kernel='linear'`, gamma is not used, and trying different values for `gamma` would be a waste of time. To deal with these kinds of "conditional" parameters, `GridSearchCV` allows the `param_grid` to be a *list of dictionaries*. Each dictionary in the list is expanded into an independent grid. A possible grid search involving kernel and parameters could look like this:

`In[34]:`

```
param_grid = [{'kernel': ['rbf'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100],
               'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
              {'kernel': ['linear'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
print("List of grids:\n{}".format(param_grid))
```

`Out[34]:`

```
List of grids:
[{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100],
  'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
 {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

In the first grid, the `kernel` parameter is always set to `'rbf'` (not that the entry for `kernel` is a list of length one), and both the `C` and `gamma` parameters are varied. In the second grid, the `kernel` parameter is always set to `linear`, and only `C` is varied. Now let's apply this more complex parameter search:

`In[35]:`

```
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

`Out[35]:`

```
Best parameters: {'C': 100, 'kernel': 'rbf', 'gamma': 0.01}
Best cross-validation score: 0.97
```

Let's look at the `cv_results_` again. As expected, if `kernel` is `'linear'`, then only `C` is varied:

In[36]:

```
results = pd.DataFrame(grid_search.cv_results_)
# we display the transposed table so that it better fits on the page:
display(results.T)
```

Out[36]:

	0	1	2	3	...	38	39	40	41
param_C	0.001	0.001	0.001	0.001	...	0.1	1	10	100
param_gamma	0.001	0.01	0.1	1	...	NaN	NaN	NaN	NaN
param_kernel	rbf	rbf	rbf	rbf	...	linear	linear	linear	linear
params	{*C*: 0.001, *kernel*: *rbf*, *gamma*: 0.001}	{*C*: 0.001, *kernel*: *rbf*, *gamma*: 0.01}	{*C*: 0.001, *kernel*: *rbf*, *gamma*: 0.1}	{*C*: 0.001, *kernel*: *rbf*, *gamma*: 1}	...	{*C*: 0.1, *kernel*: *linear*}	{*C*: 1, *kernel*: *linear*}	{*C*: 10, *kernel*: *linear*}	{*C*: 100, *kernel*: *linear*}
mean_test_score	0.37	0.37	0.37	0.37	...	0.95	0.97	0.96	0.96
rank_test_score	27	27	27	27	...	11	1	3	3
split0_test_score	0.38	0.38	0.38	0.38	...	0.96	1	0.96	0.96
split1_test_score	0.35	0.35	0.35	0.35	...	0.91	0.96	1	1
split2_test_score	0.36	0.36	0.36	0.36	...	1	1	1	1
split3_test_score	0.36	0.36	0.36	0.36	...	0.91	0.95	0.91	0.91
split4_test_score	0.38	0.38	0.38	0.38	...	0.95	0.95	0.95	0.95
std_test_score	0.011	0.011	0.011	0.011	...	0.033	0.022	0.034	0.034

```
12 rows × 42 columns
```

Using different cross-validation strategies with grid search

Similarly to `cross_val_score`, `GridSearchCV` uses stratified *k*-fold cross-validation by default for classification, and *k*-fold cross-validation for regression. However, you can also pass any cross-validation splitter, as described in "More control over cross-validation" on page 256, as the `cv` parameter in `GridSearchCV`. In particular, to get only a single split into a training and a validation set, you can use `ShuffleSplit` or `StratifiedShuffleSplit` with `n_iter=1`. This might be helpful for very large datasets, or very slow models.

Nested cross-validation

In the preceding examples, we went from using a single split of the data into training, validation, and test sets to splitting the data into training and test sets and then performing cross-validation on the training set. But when using `GridSearchCV` as

described earlier, we still have a single split of the data into training and test sets, which might make our results unstable and make us depend too much on this single split of the data. We can go a step further, and instead of splitting the original data into training and test sets once, use multiple splits of cross-validation. This will result in what is called *nested cross-validation*. In nested cross-validation, there is an outer loop over splits of the data into training and test sets. For each of them, a grid search is run (which might result in different best parameters for each split in the outer loop). Then, for each outer split, the test set score using the best settings is reported.

The result of this procedure is a list of scores—not a model, and not a parameter setting. The scores tell us how well a model generalizes, given the best parameters found by the grid. As it doesn't provide a model that can be used on new data, nested cross-validation is rarely used when looking for a predictive model to apply to future data. However, it can be useful for evaluating how well a given model works on a particular dataset.

Implementing nested cross-validation in `scikit-learn` is straightforward. We call `cross_val_score` with an instance of `GridSearchCV` as the model:

`In[34]:`

```
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                         iris.data, iris.target, cv=5)
print("Cross-validation scores: ", scores)
print("Mean cross-validation score: ", scores.mean())
```

`Out[34]:`

```
Cross-validation scores:  [ 0.967  1.     0.967  0.967  1.   ]
Mean cross-validation score:  0.98
```

The result of our nested cross-validation can be summarized as "`SVC` can achieve 98% mean cross-validation accuracy on the `iris` dataset"—nothing more and nothing less.

Here, we used stratified five-fold cross-validation in both the inner and the outer loop. As our `param_grid` contains 36 combinations of parameters, this results in a whopping 36 * 5 * 5 = 900 models being built, making nested cross-validation a very expensive procedure. Here, we used the same cross-validation splitter in the inner and the outer loop; however, this is not necessary and you can use any combination of cross-validation strategies in the inner and outer loops. It can be a bit tricky to understand what is happening in the single line given above, and it can be helpful to visualize it as `for` loops, as done in the following simplified implementation:

In[35]:

```
def nested_cv(X, y, inner_cv, outer_cv, Classifier, parameter_grid):
    outer_scores = []
    # for each split of the data in the outer cross-validation
    # (split method returns indices)
    for training_samples, test_samples in outer_cv.split(X, y):
        # find best parameter using inner cross-validation
        best_parms = {}
        best_score = -np.inf
        # iterate over parameters
        for parameters in parameter_grid:
            # accumulate score over inner splits
            cv_scores = []
            # iterate over inner cross-validation
            for inner_train, inner_test in inner_cv.split(
                    X[training_samples], y[training_samples]):
                # build classifier given parameters and training data
                clf = Classifier(**parameters)
                clf.fit(X[inner_train], y[inner_train])
                # evaluate on inner test set
                score = clf.score(X[inner_test], y[inner_test])
                cv_scores.append(score)
            # compute mean score over inner folds
            mean_score = np.mean(cv_scores)
            if mean_score > best_score:
                # if better than so far, remember parameters
                best_score = mean_score
                best_params = parameters
        # build classifier on best parameters using outer training set
        clf = Classifier(**best_params)
        clf.fit(X[training_samples], y[training_samples])
        # evaluate
        outer_scores.append(clf.score(X[test_samples], y[test_samples]))
    return np.array(outer_scores)
```

Now, let's run this function on the iris dataset:

In[36]:

```
from sklearn.model_selection import ParameterGrid, StratifiedKFold
scores = nested_cv(iris.data, iris.target, StratifiedKFold(5),
          StratifiedKFold(5), SVC, ParameterGrid(param_grid))
print("Cross-validation scores: {}".format(scores))
```

Out[36]:

```
Cross-validation scores: [ 0.967  1.     0.967  0.967  1.   ]
```

Parallelizing cross-validation and grid search

While running a grid search over many parameters and on large datasets can be computationally challenging, it is also *embarrassingly parallel*. This means that building a

model using a particular parameter setting on a particular cross-validation split can be done completely independently from the other parameter settings and models. This makes grid search and cross-validation ideal candidates for parallelization over multiple CPU cores or over a cluster. You can make use of multiple cores in `GridSearchCV` and `cross_val_score` by setting the `n_jobs` parameter to the number of CPU cores you want to use. You can set `n_jobs=-1` to use all available cores.

You should be aware that `scikit-learn` *does not allow nesting of parallel operations*. So, if you are using the `n_jobs` option on your model (for example, a random forest), you cannot use it in `GridSearchCV` to search over this model. If your dataset and model are very large, it might be that using many cores uses up too much memory, and you should monitor your memory usage when building large models in parallel.

It is also possible to parallelize grid search and cross-validation over multiple machines in a cluster, although at the time of writing this is not supported within `scikit-learn`. It is, however, possible to use the IPython parallel framework for parallel grid searches, if you don't mind writing the `for` loop over parameters as we did in "Simple Grid Search" on page 261.

For Spark users, there is also the recently developed `spark-sklearn` package, which allows running a grid search over an already established Spark cluster.

Evaluation Metrics and Scoring

So far, we have evaluated classification performance using accuracy (the fraction of correctly classified samples) and regression performance using R^2. However, these are only two of the many possible ways to summarize how well a supervised model performs on a given dataset. In practice, these evaluation metrics might not be appropriate for your application, and it is important to choose the right metric when selecting between models and adjusting parameters.

Keep the End Goal in Mind

When selecting a metric, you should always have the end goal of the machine learning application in mind. In practice, we are usually interested not just in making accurate predictions, but in using these predictions as part of a larger decision-making process. Before picking a machine learning metric, you should think about the high-level goal of the application, often called the *business metric*. The consequences of choosing a particular algorithm for a machine learning application are