6
Learning Best Practices for Model Evaluation and Hyperparameter Tuning

In the previous chapters, you learned about the essential machine learning algorithms for classification and how to get our data into shape before we feed it into those algorithms. Now, it's time to learn about the best practices of building good machine learning models by fine-tuning the algorithms and evaluating the model's performance! In this chapter, we will learn how to:

- Obtain unbiased estimates of a model's performance
- Diagnose the common problems of machine learning algorithms
- Fine-tune machine learning models
- Evaluate predictive models using different performance metrics

Streamlining workflows with pipelines

When we applied different preprocessing techniques in the previous chapters, such as **standardization** for feature scaling in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, or **principal component analysis** for data compression in *Chapter 5*, *Compressing Data via Dimensionality Reduction*, you learned that we have to reuse the parameters that were obtained during the fitting of the training data to scale and compress any new data, for example, the samples in the separate test dataset.
In this section, you will learn about an extremely handy tool, the `Pipeline` class in scikit-learn. It allows us to fit a model including an arbitrary number of transformation steps and apply it to make predictions about new data.

Loading the Breast Cancer Wisconsin dataset

In this chapter, we will be working with the **Breast Cancer Wisconsin** dataset, which contains 569 samples of **malignant** and **benign** tumor cells. The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnosis (*M=malignant*, *B=benign*), respectively. The columns 3-32 contain 30 real-value features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant. The Breast Cancer Wisconsin dataset has been deposited on the *UCI machine learning repository* and more detailed information about this dataset can be found at `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`.

In this section we will read in the dataset, and split it into training and test datasets in three simple steps:

1. We will start by reading in the dataset directly from the UCI website using pandas:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/breast-cancer-wisconsin/wdbc.data',
header=None)
```

2. Next, we assign the 30 features to a NumPy array `X`. Using `LabelEncoder`, we transform the class labels from their original string representation (`M` and `B`) into integers:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

   After encoding the class labels (diagnosis) in an array `y`, the malignant tumors are now represented as class `1`, and the benign tumors are represented as class `0`, respectively, which we can illustrate by calling the `transform` method of `LabelEncoder` on two dummy class labels:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3. Before we construct our first model pipeline in the following subsection, let's divide the dataset into a separate training dataset (80 percent of the data) and a separate test dataset (20 percent of the data):

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = \
...      train_test_split(X, y, test_size=0.20, random_state=1)
```

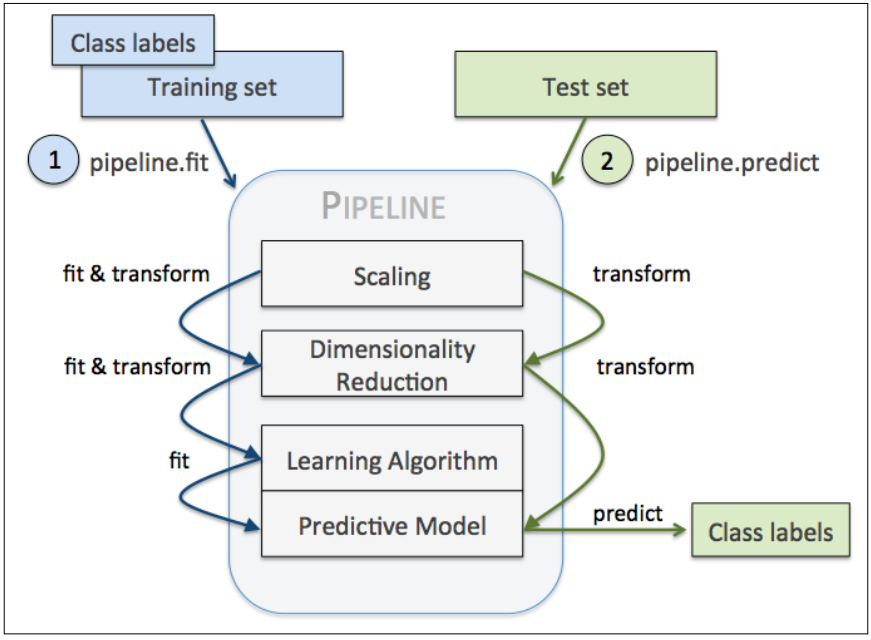
Combining transformers and estimators in a pipeline

In the previous chapter, you learned that many learning algorithms require input features on the same scale for optimal performance. Thus, we need to standardize the columns in the Breast Cancer Wisconsin dataset before we can feed them to a linear classifier, such as logistic regression. Furthermore, let's assume that we want to compress our data from the initial 30 dimensions onto a lower two-dimensional subspace via **principal component analysis** (**PCA**), a feature extraction technique for dimensionality reduction that we introduced in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. Instead of going through the fitting and transformation steps for the training and test dataset separately, we can chain the `StandardScaler`, `PCA`, and `LogisticRegression` objects in a pipeline:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import Pipeline
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...             ('pca', PCA(n_components=2)),
...             ('clf', LogisticRegression(random_state=1))])
>>> pipe_lr.fit(X_train, y_train)
>>> print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
Test Accuracy: 0.947
```

The `Pipeline` object takes a list of tuples as input, where the first value in each tuple is an arbitrary identifier string that we can use to access the individual elements in the pipeline, as we will see later in this chapter, and the second element in every tuple is a scikit-learn transformer or estimator.

The intermediate steps in a pipeline constitute scikit-learn transformers, and the last step is an estimator. In the preceding code example, we built a pipeline that consisted of two intermediate steps, a `StandardScaler` and a `PCA` transformer, and a logistic regression classifier as a final estimator. When we executed the `fit` method on the pipeline `pipe_lr`, the `StandardScaler` performed `fit` and `transform` on the training data, and the transformed training data was then passed onto the next object in the pipeline, the `PCA`. Similar to the previous step, `PCA` also executed `fit` and `transform` on the scaled input data and passed it to the final element of the pipeline, the estimator. We should note that there is no limit to the number of intermediate steps in this pipeline. The concept of how pipelines work is summarized in the following figure:



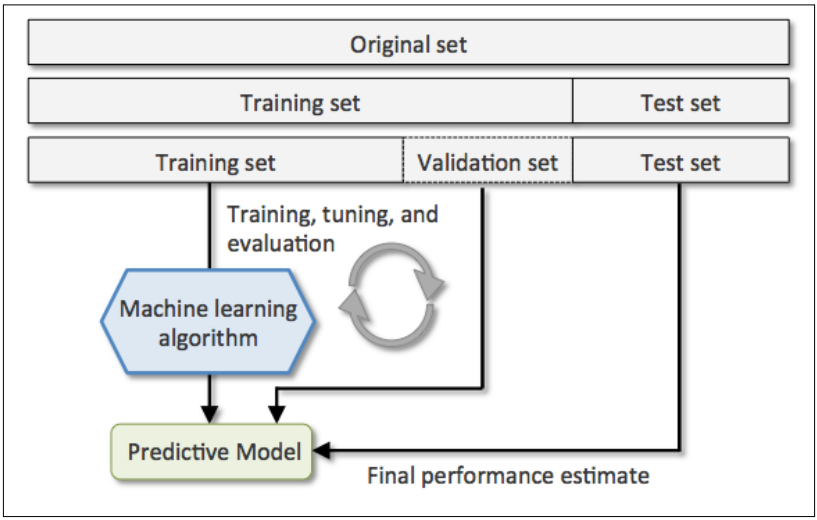
Using k-fold cross-validation to assess model performance

One of the key steps in building a machine learning model is to estimate its performance on data that the model hasn't seen before. Let's assume that we fit our model on a training dataset and use the same data to estimate how well it performs in practice. We remember from the *Tackling overfitting via regularization* section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, that a model can either suffer from underfitting (high bias) if the model is too simple, or it can overfit the training data (high variance) if the model is too complex for the underlying training data. To find an acceptable bias-variance trade-off, we need to evaluate our model carefully. In this section, you will learn about the useful cross-validation techniques **holdout cross-validation** and **k-fold cross-validation**, which can help us to obtain reliable estimates of the model's generalization error, that is, how well the model performs on unseen data.

The holdout method

A classic and popular approach for estimating the generalization performance of machine learning models is holdout cross-validation. Using the holdout method, we split our initial dataset into a separate training and test dataset—the former is used for model training, and the latter is used to estimate its performance. However, in typical machine learning applications, we are also interested in tuning and comparing different parameter settings to further improve the performance for making predictions on unseen data. This process is called **model selection**, where the term model selection refers to a given classification problem for which we want to select the *optimal* values of tuning parameters (also called **hyperparameters**). However, if we reuse the same test dataset over and over again during model selection, it will become part of our training data and thus the model will be more likely to overfit. Despite this issue, many people still use the test set for model selection, which is not a good machine learning practice.

A better way of using the holdout method for model selection is to separate the data into three parts: a training set, a validation set, and a test set. The training set is used to fit the different models, and the performance on the validation set is then used for the model selection. The advantage of having a test set that the model hasn't seen before during the training and model selection steps is that we can obtain a less biased estimate of its ability to generalize to new data. The following figure illustrates the concept of holdout cross-validation where we use a validation set to repeatedly evaluate the performance of the model after training using different parameter values. Once we are satisfied with the tuning of parameter values, we estimate the models' generalization error on the test dataset:



A disadvantage of the holdout method is that the performance estimate is sensitive to how we partition the training set into the training and validation subsets; the estimate will vary for different samples of the data. In the next subsection, we will take a look at a more robust technique for performance estimation, k-fold cross-validation, where we repeat the holdout method *k* times on *k* subsets of the training data.

K-fold cross-validation

In k-fold cross-validation, we randomly split the training dataset into *k* folds without replacement, where $k-1$ folds are used for the model training and one fold is used for testing. This procedure is repeated *k* times so that we obtain *k* models and performance estimates.

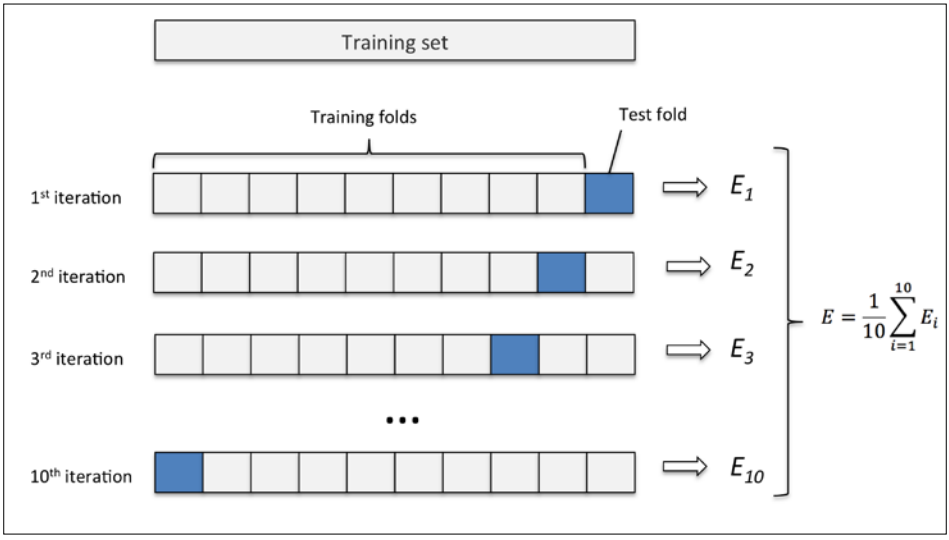
In case you are not familiar with the terms sampling *with* and *without* replacement, let's walk through a simple thought experiment. Let's assume we are playing a lottery game where we randomly draw numbers from an urn. We start with an urn that holds five unique numbers 0, 1, 2, 3, and 4, and we draw exactly one number each turn. In the first round, the chance of drawing a particular number from the urn would be 1/5. Now, in sampling without replacement, we do not put the number back into the urn after each turn. Consequently, the probability of drawing a particular number from the set of remaining numbers in the next round depends on the previous round. For example, if we have a remaining set of numbers 0, 1, 2, and 4, the chance of drawing number 0 would become 1/4 in the next turn.

However, in random sampling with replacement, we always return the drawn number to the urn so that the probabilities of drawing a particular number at each turn does not change; we can draw the same number more than once. In other words, in sampling with replacement, the samples (numbers) are independent and have a covariance zero. For example, the results from five rounds of drawing random numbers could look like this:

- Random sampling without replacement: 2, 1, 3, 4, 0
- Random sampling with replacement: 1, 3, 3, 4, 1

We then calculate the average performance of the models based on the different, independent folds to obtain a performance estimate that is less sensitive to the subpartitioning of the training data compared to the holdout method. Typically, we use k-fold cross-validation for model tuning, that is, finding the optimal hyperparameter values that yield a satisfying generalization performance. Once we have found satisfactory hyperparameter values, we can retrain the model on the complete training set and obtain a final performance estimate using the independent test set.

Since k-fold cross-validation is a resampling technique without replacement, the advantage of this approach is that each sample point will be part of a training and test dataset exactly once, which yields a lower-variance estimate of the model performance than the holdout method. The following figure summarizes the concept behind k-fold cross-validation with $k = 10$. The training data set is divided into 10 folds, and during the 10 iterations, 9 folds are used for training, and 1 fold will be used as the test set for the model evaluation. Also, the estimated performances E_i (for example, classification accuracy or error) for each fold are then used to calculate the estimated average performance E of the model:



The standard value for *k* in k-fold cross-validation is 10, which is typically a reasonable choice for most applications. However, if we are working with relatively small training sets, it can be useful to increase the number of folds. If we increase the value of *k*, more training data will be used in each iteration, which results in a lower bias towards estimating the generalization performance by averaging the individual model estimates. However, large values of *k* will also increase the runtime of the cross-validation algorithm and yield estimates with higher variance since the training folds will be more similar to each other. On the other hand, if we are working with large datasets, we can choose a smaller value for *k*, for example, $k = 5$, and still obtain an accurate estimate of the average performance of the model while reducing the computational cost of refitting and evaluating the model on the different folds.

A special case of k-fold cross validation is the **leave-one-out** (**LOO**) cross-validation method. In LOO, we set the number of folds equal to the number of training samples ($k = n$) so that only one training sample is used for testing during each iteration. This is a recommended approach for working with very small datasets.

A slight improvement over the standard k-fold cross-validation approach is stratified k-fold cross-validation, which can yield better bias and variance estimates, especially in cases of unequal class proportions, as it has been shown in a study by R. Kohavi et al. (R. Kohavi et al. *A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection*. In Ijcai, volume 14, pages 1137–1145, 1995). In stratified cross-validation, the class proportions are preserved in each fold to ensure that each fold is representative of the class proportions in the training dataset, which we will illustrate by using the `StratifiedKFold` iterator in scikit-learn:

```
>>> import numpy as np
>>> from sklearn.cross_validation import StratifiedKFold
>>> kfold = StratifiedKFold(y=y_train,
...                         n_folds=10,
...                         random_state=1)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print('Fold: %s, Class dist.: %s, Acc: %.3f' % (k+1,
...                 np.bincount(y_train[train]), score))
Fold: 1, Class dist.: [256 153], Acc: 0.891
Fold: 2, Class dist.: [256 153], Acc: 0.978
Fold: 3, Class dist.: [256 153], Acc: 0.978
Fold: 4, Class dist.: [256 153], Acc: 0.913
Fold: 5, Class dist.: [256 153], Acc: 0.935
Fold: 6, Class dist.: [257 153], Acc: 0.978
Fold: 7, Class dist.: [257 153], Acc: 0.933
Fold: 8, Class dist.: [257 153], Acc: 0.956
Fold: 9, Class dist.: [257 153], Acc: 0.978
Fold: 10, Class dist.: [257 153], Acc: 0.956
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                 np.mean(scores), np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

First, we initialized the `StratifiedKfold` iterator from the `sklearn.cross_validation` module with the class labels `y_train` in the training set, and specified the number of folds via the `n_folds` parameter. When we used the `kfold` iterator to loop through the `k` folds, we used the returned indices in `train` to fit the logistic regression pipeline that we set up at the beginning of this chapter. Using the `pile_lr` pipeline, we ensured that the samples were scaled properly (for instance, standardized) in each iteration. We then used the `test` indices to calculate the accuracy score of the model, which we collected in the `scores` list to calculate the average accuracy and the standard deviation of the estimate.

Although the previous code example was useful to illustrate how k-fold cross-validation works, scikit-learn also implements a k-fold cross-validation scorer, which allows us to evaluate our model using stratified k-fold cross-validation more efficiently:

```
>>> from sklearn.cross_validation import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                          X=X_train,
...                          y=y_train,
...                          cv=10,
...                          n_jobs=1)
>>> print('CV accuracy scores: %s' % scores)
CV accuracy scores: [ 0.89130435  0.97826087  0.97826087
                      0.91304348  0.93478261  0.97777778
                      0.93333333  0.95555556  0.97777778
                         0.95555556]
>>> print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

An extremely useful feature of the `cross_val_score` approach is that we can distribute the evaluation of the different folds across multiple CPUs on our machine. If we set the `n_jobs` parameter to `1`, only one CPU will be used to evaluate the performances just like in our `StratifiedKFold` example previously. However, by setting `n_jobs=2` we could distribute the 10 rounds of cross-validation to two CPUs (if available on our machine), and by setting `n_jobs=-1`, we can use all available CPUs on our machine to do the computation in parallel.

Please note that a detailed discussion of how the variance of the generalization performance is estimated in cross-validation is beyond the scope of this book, but you can find a detailed discussion in this excellent article by M. Markatou et al (M. Markatou, H. Tian, S. Biswas, and G. M. Hripcsak. *Analysis of Variance of Cross-validation Estimators of the Generalization Error. Journal of Machine Learning Research*, 6:1127–1168, 2005).

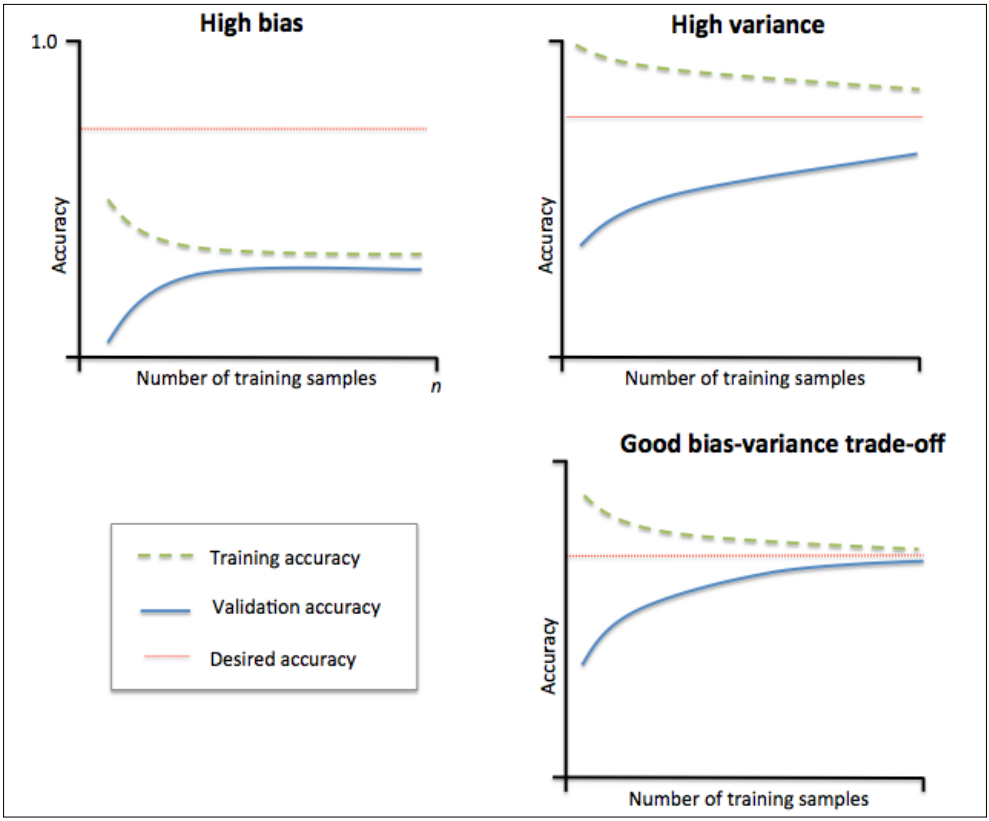
You can also read about alternative cross-validation techniques, such as the .632 Bootstrap cross-validation method (B. Efron and R. Tibshirani. *Improvements on Cross-validation: The 632+ Bootstrap Method. Journal of the American Statistical Association*, 92(438):548–560, 1997).

Debugging algorithms with learning and validation curves

In this section, we will take a look at two very simple yet powerful diagnostic tools that can help us to improve the performance of a learning algorithm: **learning curves** and **validation curves**. In the next subsections, we will discuss how we can use learning curves to diagnose if a learning algorithm has a problem with overfitting (high variance) or underfitting (high bias). Furthermore, we will take a look at validation curves that can help us address the common issues of a learning algorithm.

Diagnosing bias and variance problems with learning curves

If a model is too complex for a given training dataset—there are too many degrees of freedom or parameters in this model—the model tends to overfit the training data and does not generalize well to unseen data. Often, it can help to collect more training samples to reduce the degree of overfitting. However, in practice, it can often be very expensive or simply not feasible to collect more data. By plotting the model training and validation accuracies as functions of the training set size, we can easily detect whether the model suffers from high variance or high bias, and whether the collection of more data could help to address this problem. But before we discuss how to plot learning curves in sckit-learn, let's discuss those two common model issues by walking through the following illustration:



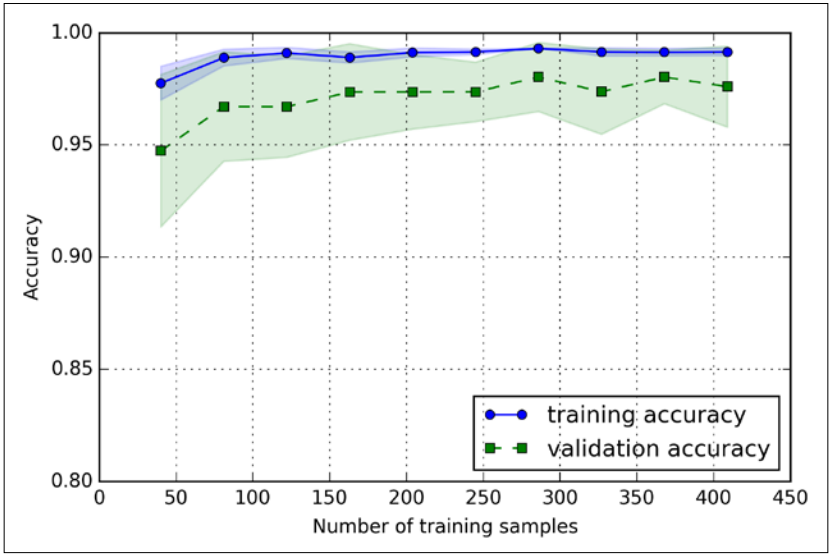
The graph in the upper-left shows a model with high bias. This model has both low training and cross-validation accuracy, which indicates that it underfits the training data. Common ways to address this issue are to increase the number of parameters of the model, for example, by collecting or constructing additional features, or by decreasing the degree of regularization, for example, in SVM or logistic regression classifiers. The graph in the upper-right shows a model that suffers from high variance, which is indicated by the large gap between the training and cross-validation accuracy. To address this problem of overfitting, we can collect more training data or reduce the complexity of the model, for example, by increasing the regularization parameter; for unregularized models, it can also help to decrease the number of features via feature selection (*Chapter 4*, *Building Good Training Sets – Data Preprocessing*) or feature extraction (*Chapter 5*, *Compressing Data via Dimensionality Reduction*). We shall note that collecting more training data decreases the chance of overfitting. However, it may not always help, for example, when the training data is extremely noisy or the model is already very close to optimal.

In the next subsection, we will see how to address those model issues using validation curves, but let's first see how we can use the learning curve function from scikit-learn to evaluate the model:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.learning_curve import learning_curve
>>> pipe_lr = Pipeline([
...           ('scl', StandardScaler()),
...           ('clf', LogisticRegression(
...                         penalty='l2', random_state=0))])
>>> train_sizes, train_scores, test_scores =\
...        learning_curve(estimator=pipe_lr,
...                       X=X_train,
...                       y=y_train,
...                       train_sizes=np.linspace(0.1, 1.0, 10),
...                       cv=10,
...                       n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...          color='blue', marker='o',
...          markersize=5,
...          label='training accuracy')
>>> plt.fill_between(train_sizes,
...                  train_mean + train_std,
...                  train_mean - train_std,
```

```
...                      alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(train_sizes,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Number of training samples')
>>> plt.ylabel('Accuracy')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

After we have successfully executed the preceding code, we will obtain the following learning curve plot:



Via the `train_sizes` parameter in the `learning_curve` function, we can control the absolute or relative number of training samples that are used to generate the learning curves. Here, we set `train_sizes=np.linspace(0.1, 1.0, 10)` to use 10 evenly spaced relative intervals for the training set sizes. By default, the `learning_curve` function uses stratified k-fold cross-validation to calculate the cross-validation accuracy, and we set $k = 10$ via the `cv` parameter. Then, we simply calculate the average accuracies from the returned cross-validated training and test scores for the different sizes of the training set, which we plotted using matplotlib's `plot` function. Furthermore, we add the standard deviation of the average accuracies to the plot using the `fill_between` function to indicate the variance of the estimate.

As we can see in the preceding learning curve plot, our model performs quite well on the test dataset. However, it may be slightly overfitting the training data indicated by a relatively small, but visible, gap between the training and cross-validation accuracy curves.

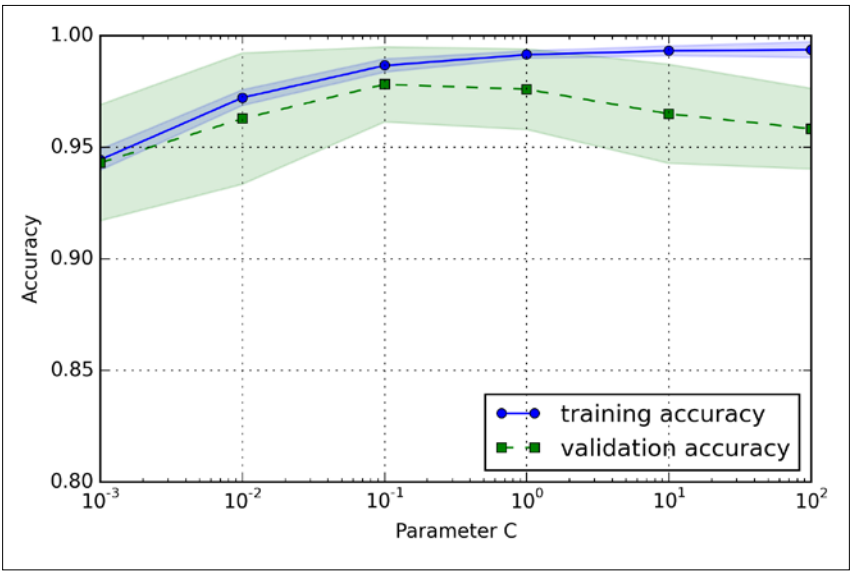
Addressing overfitting and underfitting with validation curves

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting. Validation curves are related to learning curves, but instead of plotting the training and test accuracies as functions of the sample size, we vary the values of the model parameters, for example, the inverse regularization parameter `C` in logistic regression. Let's go ahead and see how we create validation curves via sckit-learn:

```
>>> from sklearn.learning_curve import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...                 estimator=pipe_lr,
...                  X=X_train,
...                  y=y_train,
...                  param_name='clf__C',
...                  param_range=param_range,
...                  cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
```

```
>>> plt.plot(param_range, train_mean,
...          color='blue', marker='o',
...          markersize=5,
...          label='training accuracy')
>>> plt.fill_between(param_range, train_mean + train_std,
...                  train_mean - train_std, alpha=0.15,
...                  color='blue')
>>> plt.plot(param_range, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(param_range,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parameter C')
>>> plt.ylabel('Accuracy')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Using the preceding code, we obtained the validation curve plot for the parameter `C`:



Similar to the `learning_curve` function, the `validation_curve` function uses stratified k-fold cross-validation by default to estimate the performance of the model if we are using algorithms for classification. Inside the `validation_curve` function, we specified the parameter that we wanted to evaluate. In this case, it is `C`, the inverse regularization parameter of the `LogisticRegression` classifier, which we wrote as `'clf__C'` to access the `LogisticRegression` object inside the scikit-learn pipeline for a specified value range that we set via the `param_range` parameter. Similar to the learning curve example in the previous section, we plotted the average training and cross-validation accuracies and the corresponding standard deviations.

Although the differences in the accuracy for varying values of `C` are subtle, we can see that the model slightly underfits the data when we increase the regularization strength (small values of `C`). However, for large values of `C`, it means lowering the strength of regularization, so the model tends to slightly overfit the data. In this case, the sweet spot appears to be around `C=0.1`.

Fine-tuning machine learning models via grid search

In machine learning, we have two types of parameters: those that are learned from the training data, for example, the weights in logistic regression, and the parameters of a learning algorithm that are optimized separately. The latter are the tuning parameters, also called hyperparameters, of a model, for example, the **regularization** parameter in logistic regression or the **depth** parameter of a decision tree.

In the previous section, we used validation curves to improve the performance of a model by tuning one of its hyperparameters. In this section, we will take a look at a powerful hyperparameter optimization technique called **grid search** that can further help to improve the performance of a model by finding the *optimal* combination of hyperparameter values.

Tuning hyperparameters via grid search

The approach of grid search is quite simple, it's a brute-force exhaustive search paradigm where we specify a list of values for different hyperparameters, and the computer evaluates the model performance for each combination of those to obtain the optimal set:

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import SVC
>>> pipe_svc = Pipeline([('scl', StandardScaler()),
...                      ('clf', SVC(random_state=1))])
>>> param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'clf__C': param_range,
...                'clf__kernel': ['linear']},
...               {'clf__C': param_range,
...                'clf__gamma': param_range,
...                'clf__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=10,
...                   n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.978021978022
>>> print(gs.best_params_)
{'clf__C': 0.1, 'clf__kernel': 'linear'}
```

Using the preceding code, we initialized a `GridSearchCV` object from the `sklearn.grid_search` module to train and tune a **support vector machine** (**SVM**) pipeline. We set the `param_grid` parameter of `GridSearchCV` to a list of dictionaries to specify the parameters that we'd want to tune. For the linear SVM, we only evaluated the inverse regularization parameter `C`; for the RBF kernel SVM, we tuned both the `C` and `gamma` parameter. Note that the `gamma` parameter is specific to kernel SVMs. After we used the training data to perform the grid search, we obtained the score of the best-performing model via the `best_score_` attribute and looked at its parameters, that can be accessed via the `best_params_` attribute. In this particular case, the linear SVM model with `'clf__C'= 0.1'` yielded the best k-fold cross-validation accuracy: 97.8 percent.

Finally, we will use the independent test dataset to estimate the performance of the best selected model, which is available via the `best_estimator_` attribute of the `GridSearchCV` object:

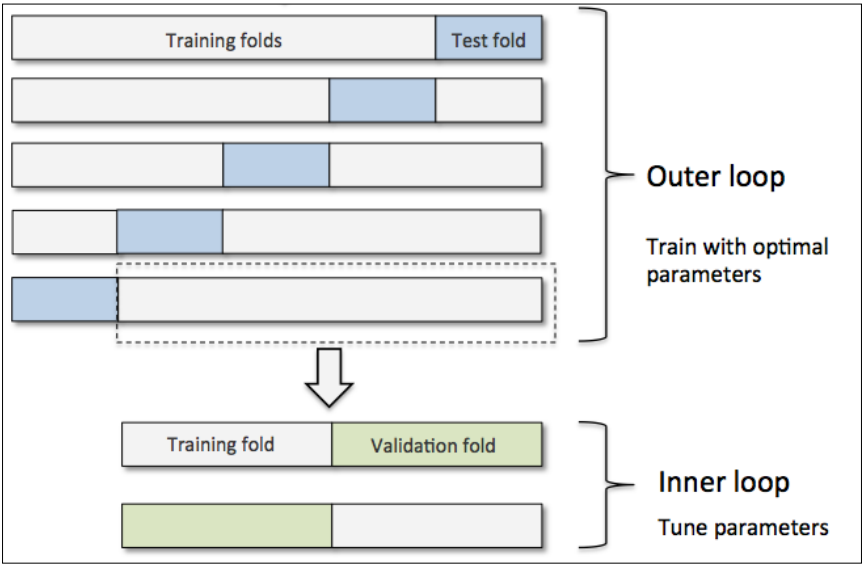
```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print('Test accuracy: %.3f' % clf.score(X_test, y_test))
Test accuracy: 0.965
```

Although grid search is a powerful approach for finding the optimal set of parameters, the evaluation of all possible parameter combinations is also computationally very expensive. An alternative approach to sampling different parameter combinations using scikit-learn is randomized search. Using the `RandomizedSearchCV` class in scikit-learn, we can draw random parameter combinations from sampling distributions with a specified budget. More details and examples for its usage can be found at `http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization`.

Algorithm selection with nested cross-validation

Using k-fold cross-validation in combination with grid search is a useful approach for fine-tuning the performance of a machine learning model by varying its hyperparameters values as we saw in the previous subsection. If we want to select among different machine learning algorithms though, another recommended approach is nested cross-validation, and in a nice study on the bias in error estimation, Varma and Simon concluded that the true error of the estimate is almost unbiased relative to the test set when nested cross-validation is used (S. Varma and R. Simon. *Bias in Error Estimation When Using Cross-validation for Model Selection*. BMC bioinformatics, 7(1):91, 2006).

In nested cross-validation, we have an outer k-fold cross-validation loop to split the data into training and test folds, and an inner loop is used to select the model using k-fold cross-validation on the training fold. After model selection, the test fold is then used to evaluate the model performance. The following figure explains the concept of nested cross-validation with five outer and two inner folds, which can be useful for large data sets where computational performance is important; this particular type of nested cross-validation is also known as **5x2 cross-validation**:



In scikit-learn, we can perform nested cross-validation as follows:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=5,
...                   n_jobs=-1)
>>> scores = cross_val_score(gs, X, y, scoring='accuracy', cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...               np.mean(scores), np.std(scores)))
CV accuracy: 0.978 +/- 0.012
```

The returned average cross-validation accuracy gives us a good estimate of what to expect if we tune the hyperparameters of a model and then use it on unseen data. For example, we can use the nested cross-validation approach to compare an SVM model to a simple decision tree classifier; for simplicity, we will only tune its depth parameter:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...       estimator=DecisionTreeClassifier(random_state=0),
...       param_grid=[
...            {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...       scoring='accuracy',
...       cv=5)
>>> scores = cross_val_score(gs,
...                          X_train,
...                          y_train,
...                          scoring='accuracy',
...                          cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                      np.mean(scores), np.std(scores)))
CV accuracy: 0.908 +/- 0.045
```

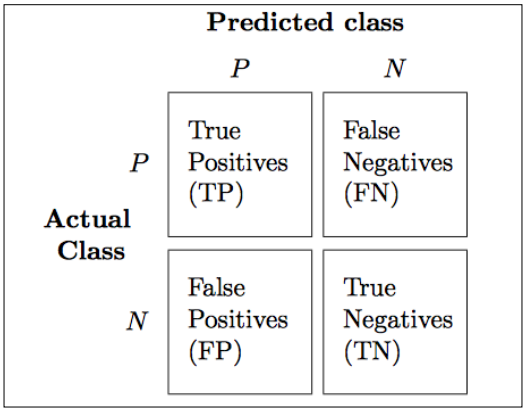
As we can see here, the nested cross-validation performance of the SVM model (97.8 percent) is notably better than the performance of the decision tree (90.8 percent). Thus, we'd expect that it might be the better choice for classifying new data that comes from the same population as this particular dataset.

Looking at different performance evaluation metrics

In the previous sections and chapters, we evaluated our models using the model accuracy, which is a useful metric to quantify the performance of a model in general. However, there are several other performance metrics that can be used to measure a model's relevance, such as **precision**, **recall**, and the **F1-score**.

Reading a confusion matrix

Before we get into the details of different scoring metrics, let's print a so-called **confusion matrix**, a matrix that lays out the performance of a learning algorithm. The confusion matrix is simply a square matrix that reports the counts of the **true positive**, **true negative**, **false positive**, and **false negative** predictions of a classifier, as shown in the following figure:



Although these metrics can be easily computed manually by comparing the true and predicted class labels, scikit-learn provides a convenient `confusion_matrix` function that we can use as follows:

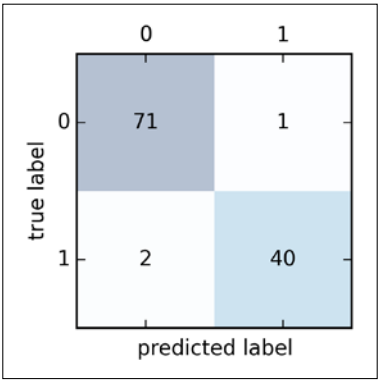
```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
 [ 2 40]]
```

The array that was returned after executing the preceding code provides us with information about the different types of errors the classifier made on the test dataset that we can map onto the confusion matrix illustration in the previous figure using matplotlib's `matshow` function:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
```

```
>>> plt.xlabel('predicted label')
>>> plt.ylabel('true label')
>>> plt.show()
```

Now, the confusion matrix plot as shown here should make the results a little bit easier to interpret:



Assuming that class 1 (malignant) is the positive class in this example, our model correctly classified 71 of the samples that belong to class 0 (false negatives) and 40 samples that belong to class 1 (true positives), respectively. However, our model also incorrectly misclassified 2 samples from class 0 as class 1 (false negatives), and it predicted that 1 sample is benign although it is a malignant tumor (false positive). In the next section, we will learn how we can use this information to calculate various different error metrics.

Optimizing the precision and recall of a classification model

Both the prediction **error** (**ERR**) and **accuracy** (**ACC**) provide general information about how many samples are misclassified. The error can be understood as the sum of all false predictions divided by the number of total predications, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions, respectively:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

The prediction accuracy can then be calculated directly from the error:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

The **true positive rate** (**TPR**) and **false positive rate** (**FPR**) are performance metrics that are especially useful for imbalanced class problems:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In tumor diagnosis, for example, we are more concerned about the detection of malignant tumors in order to help a patient with the appropriate treatment. However, it is also important to decrease the number of benign tumors that were incorrectly classified as malignant (false positives) to not unnecessarily concern a patient. In contrast to the FPR, the true positive rate provides useful information about the fraction of positive (or relevant) samples that were correctly identified out of the total pool of positives (**P**).

Precision (**PRE**) and **recall** (**REC**) are performance metrics that are related to those true positive and true negative rates, and in fact, recall is synonymous to the true positive rate:

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In practice, often a combination of precision and recall is used, the so-called **F1-score**:

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

These scoring metrics are all implemented in scikit-learn and can be imported from the `sklearn.metrics` module, as shown in the following snippet:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics  import recall_score, f1_score
>>> print('Precision: %.3f' % precision_score(
...              y_true=y_test, y_pred=y_pred))
Precision: 0.976
>>> print('Recall: %.3f' % recall_score(
...              y_true=y_test, y_pred=y_pred))
Recall: 0.952
>>> print('F1: %.3f' % f1_score(
...              y_true=y_test, y_pred=y_pred))
F1: 0.964
```

Furthermore, we can use a different scoring metric other than accuracy in `GridSearch` via the scoring parameter. A complete list of the different values that are accepted by the scoring parameter can be found at `http://scikit-learn.org/stable/modules/model_evaluation.html`.

Remember that the positive class in scikit-learn is the class that is labeled as class 1. If we want to specify a different *positive label*, we can construct our own scorer via the `make_scorer` function, which we can then directly provide as an argument to the scoring parameter in `GridSearchCV`:

```
>>> from sklearn.metrics import make_scorer, f1_score
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring=scorer,
...                   cv=10)
```

Plotting a receiver operating characteristic

Receiver operator characteristic (**ROC**) graphs are useful tools for selecting models for classification based on their performance with respect to the false positive and true positive rates, which are computed by shifting the decision threshold of the classifier. The diagonal of an ROC graph can be interpreted as random guessing, and classification models that fall below the diagonal are considered as worse than random guessing. A perfect classifier would fall into the top-left corner of the graph with a true positive rate of 1 and a false positive rate of 0. Based on the ROC curve, we can then compute the so-called **area under the curve** (**AUC**) to characterize the performance of a classification model.

Similar to ROC curves, we can compute **precision-recall curves** for the different probability thresholds of a classifier. A function for plotting those precision-recall curves is also implemented in scikit-learn and is documented at `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html`.

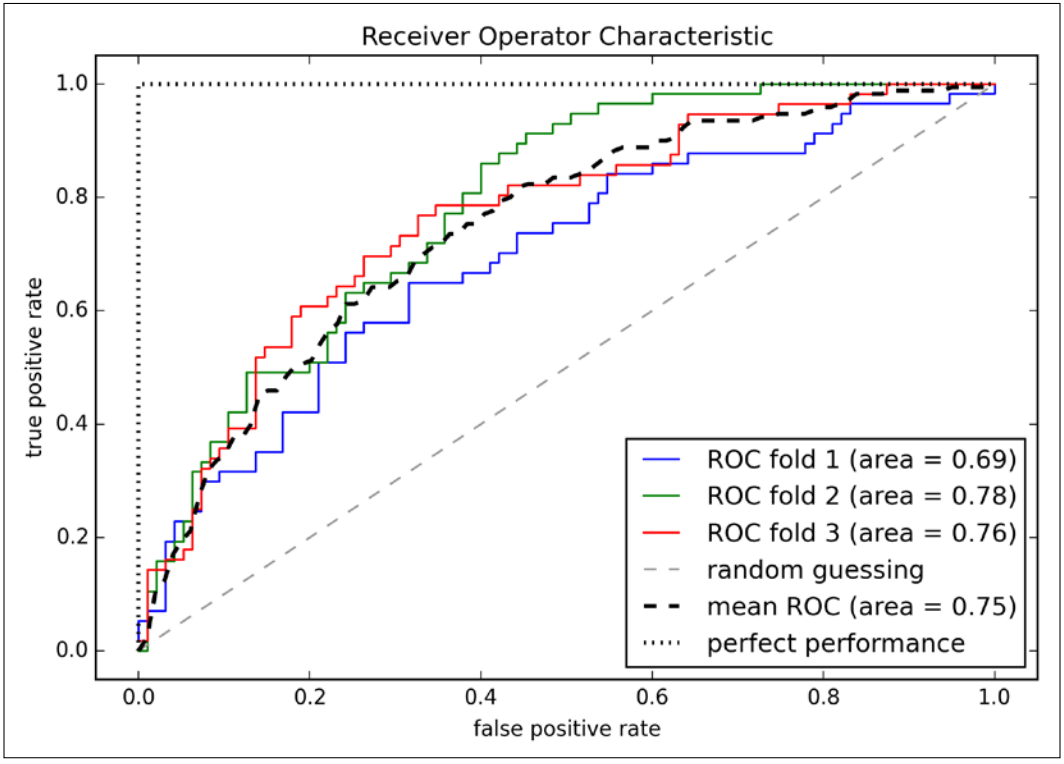
By executing the following code example, we will plot an ROC curve of a classifier that only uses two features from the Breast Cancer Wisconsin dataset to predict whether a tumor is benign or malignant. Although we are going to use the same logistic regression pipeline that we defined previously, we are making the classification task more challenging for the classifier so that the resulting ROC curve becomes visually more interesting. For similar reasons, we are also reducing the number of folds in the `StratifiedKFold` validator to three. The code is as follows:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = StratifiedKFold(y_train,
...                      n_folds=3,
...                      random_state=1)
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []

>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(X_train2[train],
>>> y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
```

```
...                                     probas[:, 1],
...                                     pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
...              tpr,
...              lw=1,
...              label='ROC fold %d (area = %0.2f)'
...                    % (i+1, roc_auc))
>>> plt.plot([0, 1],
...          [0, 1],
...          linestyle='--',
...          color=(0.6, 0.6, 0.6),
...          label='random guessing')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...          label='mean ROC (area = %0.2f)' % mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...          [0, 1, 1],
...          lw=2,
...          linestyle=':',
...          color='black',
...          label='perfect performance')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('false positive rate')
>>> plt.ylabel('true positive rate')
>>> plt.title('Receiver Operator Characteristic')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

In the preceding code example, we used the already familiar `StratifiedKFold` class from scikit-learn and calculated the ROC performance of the `LogisticRegression` classifier in our `pipe_lr` pipeline using the `roc_curve` function from the `sklearn.metrics` module separately for each iteration. Furthermore, we interpolated the average ROC curve from the three folds via the `interp` function that we imported from SciPy and calculated the area under the curve via the `auc` function. The resulting ROC curve indicates that there is a certain degree of variance between the different folds, and the average ROC AUC (0.75) falls between a perfect score (1.0) and random guessing (0.5):



If we are just interested in the ROC AUC score, we could also directly import the `roc_auc_score` function from the `sklearn.metrics` submodule. The following code calculates the classifier's ROC AUC score on the independent test dataset after fitting it on the two-feature training set:

```
>>> pipe_svc = pipe_svc.fit(X_train2, y_train)
>>> y_pred2 = pipe_svc.predict(X_test[:, [4, 14]])
```

```
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.metrics import accuracy_score
>>> print('ROC AUC: %.3f' % roc_auc_score(
...         y_true=y_test, y_score=y_pred2))
ROC AUC: 0.671

>>> print('Accuracy: %.3f' % accuracy_score(
...         y_true=y_test, y_pred=y_pred2))
Accuracy: 0.728
```

Reporting the performance of a classifier as the ROC AUC can yield further insights in a classifier's performance with respect to imbalanced samples. However, while the accuracy score can be interpreted as a single cut-off point on a ROC curve, A. P. Bradley showed that the ROC AUC and accuracy metrics mostly agree with each other (A. P. Bradley. *The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms*. Pattern recognition, 30(7):1145–1159, 1997).

The scoring metrics for multiclass classification

The scoring metrics that we discussed in this section are specific to binary classification systems. However, scikit-learn also implements **macro** and **micro** averaging methods to extend those scoring metrics to multiclass problems via **One vs. All** (**OvA**) classification. The micro-average is calculated from the individual true positives, true negatives, false positives, and false negatives of the system. For example, the micro-average of the precision score in a k-class system can be calculated as follows:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

The macro-average is simply calculated as the average scores of the different systems:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Micro-averaging is useful if we want to weight each instance or prediction equally, whereas macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels.

If we are using binary performance metrics to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default. The weighted macro-average is calculated by weighting the score of each class label by the number of true instances when calculating the average. The weighted macro-average is useful if we are dealing with class imbalances, that is, different numbers of instances for each label.

While the weighted macro-average is the default for multiclass problems in scikit-learn, we can specify the averaging method via the `average` parameter inside the different scoring functions that we import from the `sklean.metrics` module, for example, the `precision_score` or `make_scorer` functions:

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
...                          greater_is_better=True,
...                          average='micro')
```

Summary

In the beginning of this chapter, we discussed how to chain different transformation techniques and classifiers in convenient model pipelines that helped us to train and evaluate machine learning models more efficiently. We then used those pipelines to perform k-fold cross-validation, one of the essential techniques for model selection and evaluation. Using k-fold cross-validation, we plotted learning and validation curves to diagnose the common problems of learning algorithms, such as overfitting and underfitting. Using grid search, we further fine-tuned our model. We concluded this chapter by looking at a confusion matrix and various different performance metrics that can be useful to further optimize a model's performance for a specific problem task. Now, we should be well-equipped with the essential techniques to build supervised machine learning models for classification successfully.

In the next chapter, we will take a look at ensemble methods, methods that allow us to combine multiple models and classification algorithms to boost the predictive performance of a machine learning system even further.