FIGURE 5.5. *A schematic display of* 5*-fold CV. A set of n observations is randomly split into five non-overlapping groups. Each of these fifths acts as a validation set (shown in beige), and the remainder as a training set (shown in blue). The test error is estimated by averaging the five resulting MSE estimates.*

It is not hard to see that LOOCV is a special case of k-fold CV in which k is set to equal n. In practice, one typically performs k-fold CV using $k = 5$ or $k = 10$. What is the advantage of using $k = 5$ or $k = 10$ rather than $k = n$? The most obvious advantage is computational. LOOCV requires fitting the statistical learning method n times. This has the potential to be computationally expensive (except for linear models fit by least squares, in which case formula (5.2) can be used). But cross-validation is a very general approach that can be applied to almost any statistical learning method. Some statistical learning methods have computationally intensive fitting procedures, and so performing LOOCV may pose computational problems, especially if n is extremely large. In contrast, performing 10-fold CV requires fitting the learning procedure only ten times, which may be much more feasible. As we see in Section 5.1.4, there also can be other non-computational advantages to performing 5-fold or 10-fold CV, which involve the bias-variance trade-off.

The right-hand panel of Figure 5.4 displays nine different 10-fold CV estimates for the `Auto` data set, each resulting from a different random split of the observations into ten folds. As we can see from the figure, there is some variability in the CV estimates as a result of the variability in how the observations are divided into ten folds. But this variability is typically much lower than the variability in the test error estimates that results from the validation set approach (right-hand panel of Figure 5.2).

When we examine real data, we do not know the *true* test MSE, and so it is difficult to determine the accuracy of the cross-validation estimate. However, if we examine simulated data, then we can compute the true test MSE, and can thereby evaluate the accuracy of our cross-validation results. In Figure 5.6, we plot the cross-validation estimates and true test error rates that result from applying smoothing splines to the simulated data sets illustrated in Figures 2.9–2.11 of Chapter 2. The true test MSE is displayed in blue. The black dashed and orange solid lines respectively show the estimated LOOCV and 10-fold CV estimates. In all three plots, the two cross-validation estimates are very similar. In the right-hand panel
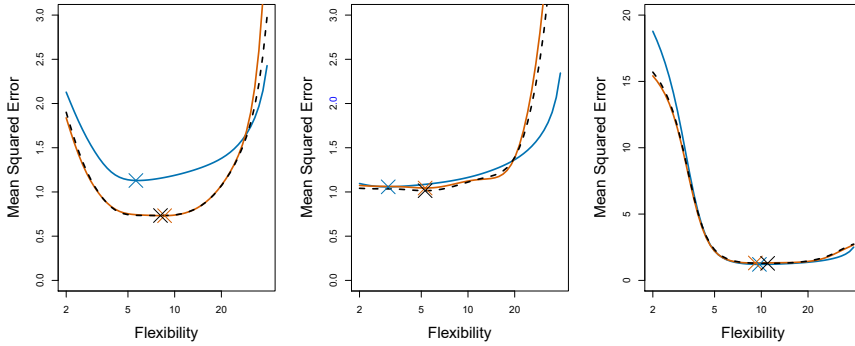
FIGURE 5.6. *True and estimated test MSE for the simulated data sets in Figures 2.9 (*left*), 2.10 (*center*), and 2.11 (*right*). The true test MSE is shown in blue, the LOOCV estimate is shown as a black dashed line, and the* 10-*fold CV estimate is shown in orange. The crosses indicate the minimum of each of the MSE curves.*

of Figure 5.6, the true test MSE and the cross-validation curves are almost identical. In the center panel of Figure 5.6, the two sets of curves are similar at the lower degrees of flexibility, while the CV curves overestimate the test set MSE for higher degrees of flexibility. In the left-hand panel of Figure 5.6, the CV curves have the correct general shape, but they underestimate the true test MSE.

When we perform cross-validation, our goal might be to determine how well a given statistical learning procedure can be expected to perform on independent data; in this case, the actual estimate of the test MSE is of interest. But at other times we are interested only in the location of the *minimum point in the estimated test MSE curve.* This is because we might be performing cross-validation on a number of statistical learning methods, or on a single method using different levels of flexibility, in order to identify the method that results in the lowest test error. For this purpose, the location of the minimum point in the estimated test MSE curve is important, but the actual value of the estimated test MSE is not. We find in Figure 5.6 that despite the fact that they sometimes underestimate the true test MSE, all of the CV curves come close to identifying the correct level of flexibility—that is, the flexibility level corresponding to the smallest test MSE.

5.1.4   Bias-Variance Trade-Off for k-Fold Cross-Validation

We mentioned in Section 5.1.3 that k-fold CV with $k < n$ has a computational advantage to LOOCV. But putting computational issues aside, a less obvious but potentially more important advantage of k-fold CV is that it often gives more accurate estimates of the test error rate than does LOOCV. This has to do with a bias-variance trade-off.

It was mentioned in Section 5.1.1 that the validation set approach can lead to overestimates of the test error rate, since in this approach the training set used to fit the statistical learning method contains only half the observations of the entire data set. Using this logic, it is not hard to see

that LOOCV will give approximately unbiased estimates of the test error, since each training set contains $n-1$ observations, which is almost as many as the number of observations in the full data set. And performing k-fold CV for, say, $k = 5$ or $k = 10$ will lead to an intermediate level of bias, since each training set contains approximately $(k-1)n/k$ observations— fewer than in the LOOCV approach, but substantially more than in the validation set approach. Therefore, from the perspective of bias reduction, it is clear that LOOCV is to be preferred to k-fold CV.

However, we know that bias is not the only source for concern in an estimating procedure; we must also consider the procedure's variance. It turns out that LOOCV has higher variance than does k-fold CV with $k < n$. Why is this the case? When we perform LOOCV, we are in effect averaging the outputs of n fitted models, each of which is trained on an almost identical set of observations; therefore, these outputs are highly (positively) correlated with each other. In contrast, when we perform k-fold CV with $k < n$, we are averaging the outputs of k fitted models that are somewhat less correlated with each other, since the overlap between the training sets in each model is smaller. Since the mean of many highly correlated quantities has higher variance than does the mean of many quantities that are not as highly correlated, the test error estimate resulting from LOOCV tends to have higher variance than does the test error estimate resulting from k-fold CV.

To summarize, there is a bias-variance trade-off associated with the choice of k in k-fold cross-validation. Typically, given these considerations, one performs k-fold cross-validation using $k = 5$ or $k = 10$, as these values have been shown empirically to yield test error rate estimates that suffer neither from excessively high bias nor from very high variance.

5.1.5 *Cross-Validation on Classification Problems*

In this chapter so far, we have illustrated the use of cross-validation in the regression setting where the outcome Y is quantitative, and so have used MSE to quantify test error. But cross-validation can also be a very useful approach in the classification setting when Y is qualitative. In this setting, cross-validation works just as described earlier in this chapter, except that rather than using MSE to quantify test error, we instead use the number of misclassified observations. For instance, in the classification setting, the LOOCV error rate takes the form

$$\text{CV}_{(n)} = \frac{1}{n} \sum_{i=1}^{n} \text{Err}_i, \qquad (5.4)$$

where $\text{Err}_i = I(y_i \neq \hat{y}_i)$. The k-fold CV error rate and validation set error rates are defined analogously.

As an example, we fit various logistic regression models on the two-dimensional classification data displayed in Figure 2.13. In the top-left panel of Figure 5.7, the black solid line shows the estimated decision boundary resulting from fitting a standard logistic regression model to this data set. Since this is simulated data, we can compute the *true* test error rate, which takes a value of 0.201 and so is substantially larger than the Bayes
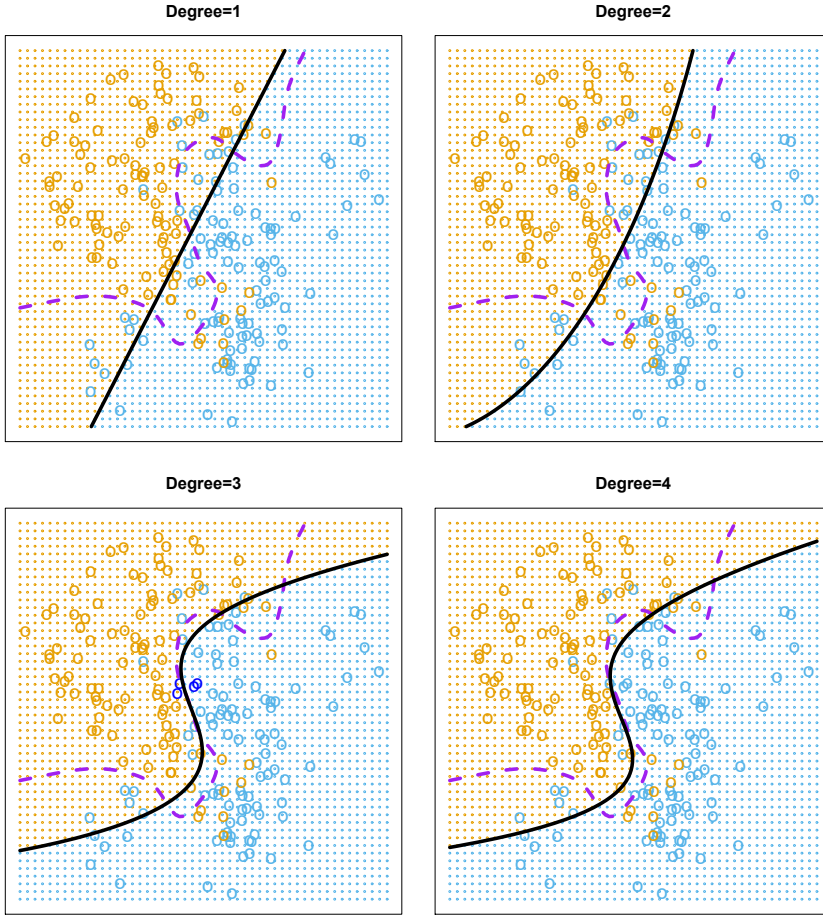
FIGURE 5.7. *Logistic regression fits on the two-dimensional classification data displayed in Figure 2.13. The Bayes decision boundary is represented using a purple dashed line. Estimated decision boundaries from linear, quadratic, cubic and quartic (degrees 1–4) logistic regressions are displayed in black. The test error rates for the four logistic regression fits are respectively* 0.201, 0.197, 0.160, *and* 0.162, *while the Bayes error rate is* 0.133.

error rate of 0.133. Clearly logistic regression does not have enough flexibility to model the Bayes decision boundary in this setting. We can easily extend logistic regression to obtain a non-linear decision boundary by using polynomial functions of the predictors, as we did in the regression setting in Section 3.3.2. For example, we can fit a *quadratic* logistic regression model, given by

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_1^2 + \beta_3 X_2 + \beta_4 X_2^2. \tag{5.5}$$

The top-right panel of Figure 5.7 displays the resulting decision boundary, which is now curved. However, the test error rate has improved only slightly, to 0.197. A much larger improvement is apparent in the bottom-left panel
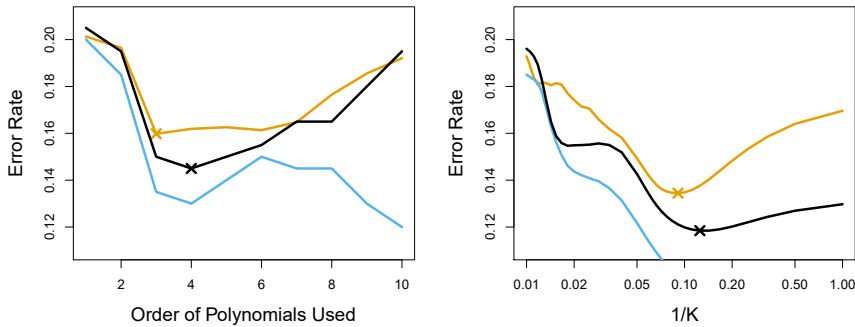
FIGURE 5.8. *Test error (brown), training error (blue), and* 10*-fold CV error (black) on the two-dimensional classification data displayed in Figure 5.7.* Left: *Logistic regression using polynomial functions of the predictors. The order of the polynomials used is displayed on the x-axis.* Right: *The KNN classifier with different values of K, the number of neighbors used in the KNN classifier.*

of Figure 5.7, in which we have fit a logistic regression model involving cubic polynomials of the predictors. Now the test error rate has decreased to 0.160. Going to a quartic polynomial (bottom-right) slightly increases the test error.

In practice, for real data, the Bayes decision boundary and the test error rates are unknown. So how might we decide between the four logistic regression models displayed in Figure 5.7? We can use cross-validation in order to make this decision. The left-hand panel of Figure 5.8 displays in black the 10-fold CV error rates that result from fitting ten logistic regression models to the data, using polynomial functions of the predictors up to tenth order. The true test errors are shown in brown, and the training errors are shown in blue. As we have seen previously, the training error tends to decrease as the flexibility of the fit increases. (The figure indicates that though the training error rate doesn't quite decrease monotonically, it tends to decrease on the whole as the model complexity increases.) In contrast, the test error displays a characteristic U-shape. The 10-fold CV error rate provides a pretty good approximation to the test error rate. While it somewhat underestimates the error rate, it reaches a minimum when fourth-order polynomials are used, which is very close to the minimum of the test curve, which occurs when third-order polynomials are used. In fact, using fourth-order polynomials would likely lead to good test set performance, as the true test error rate is approximately the same for third, fourth, fifth, and sixth-order polynomials.

The right-hand panel of Figure 5.8 displays the same three curves using the KNN approach for classification, as a function of the value of K (which in this context indicates the number of neighbors used in the KNN classifier, rather than the number of CV folds used). Again the training error rate declines as the method becomes more flexible, and so we see that the training error rate cannot be used to select the optimal value for K. Though the cross-validation error curve slightly underestimates the test error rate, it takes on a minimum very close to the best value for K.

5.2   The Bootstrap

The *bootstrap* is a widely applicable and extremely powerful statistical tool   bootstrap
that can be used to quantify the uncertainty associated with a given esti-
mator or statistical learning method. As a simple example, the bootstrap
can be used to estimate the standard errors of the coefficients from a linear
regression fit. In the specific case of linear regression, this is not particularly
useful, since we saw in Chapter 3 that standard statistical software such as
R outputs such standard errors automatically. However, the power of the
bootstrap lies in the fact that it can be easily applied to a wide range of
statistical learning methods, including some for which a measure of vari-
ability is otherwise difficult to obtain and is not automatically output by
statistical software.

In this section we illustrate the bootstrap on a toy example in which we
wish to determine the best investment allocation under a simple model.
In Section 5.3 we explore the use of the bootstrap to assess the variability
associated with the regression coefficients in a linear model fit.

Suppose that we wish to invest a fixed sum of money in two financial
assets that yield returns of X and Y, respectively, where X and Y are
random quantities. We will invest a fraction α of our money in X, and will
invest the remaining $1 - \alpha$ in Y. Since there is variability associated with
the returns on these two assets, we wish to choose α to minimize the total
risk, or variance, of our investment. In other words, we want to minimize
$\text{Var}(\alpha X + (1-\alpha)Y)$. One can show that the value that minimizes the risk
is given by

$$\alpha = \frac{\sigma_Y^2 - \sigma_{XY}}{\sigma_X^2 + \sigma_Y^2 - 2\sigma_{XY}}, \tag{5.6}$$

where $\sigma_X^2 = \text{Var}(X), \sigma_Y^2 = \text{Var}(Y)$, and $\sigma_{XY} = \text{Cov}(X, Y)$.

In reality, the quantities σ_X^2, σ_Y^2, and σ_{XY} are unknown. We can compute
estimates for these quantities, $\hat{\sigma}_X^2$, $\hat{\sigma}_Y^2$, and $\hat{\sigma}_{XY}$, using a data set that
contains past measurements for X and Y. We can then estimate the value
of α that minimizes the variance of our investment using

$$\hat{\alpha} = \frac{\hat{\sigma}_Y^2 - \hat{\sigma}_{XY}}{\hat{\sigma}_X^2 + \hat{\sigma}_Y^2 - 2\hat{\sigma}_{XY}}. \tag{5.7}$$

Figure 5.9 illustrates this approach for estimating α on a simulated data
set. In each panel, we simulated 100 pairs of returns for the investments
X and Y. We used these returns to estimate σ_X^2, σ_Y^2, and σ_{XY}, which we
then substituted into (5.7) in order to obtain estimates for α. The value of
$\hat{\alpha}$ resulting from each simulated data set ranges from 0.532 to 0.657.

It is natural to wish to quantify the accuracy of our estimate of α. To
estimate the standard deviation of $\hat{\alpha}$, we repeated the process of simu-
lating 100 paired observations of X and Y, and estimating α using (5.7),
1,000 times. We thereby obtained 1,000 estimates for α, which we can call
$\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_{1,000}$. The left-hand panel of Figure 5.10 displays a histogram
of the resulting estimates. For these simulations the parameters were set to
$\sigma_X^2 = 1, \sigma_Y^2 = 1.25$, and $\sigma_{XY} = 0.5$, and so we know that the true value of
α is 0.6. We indicated this value using a solid vertical line on the histogram.
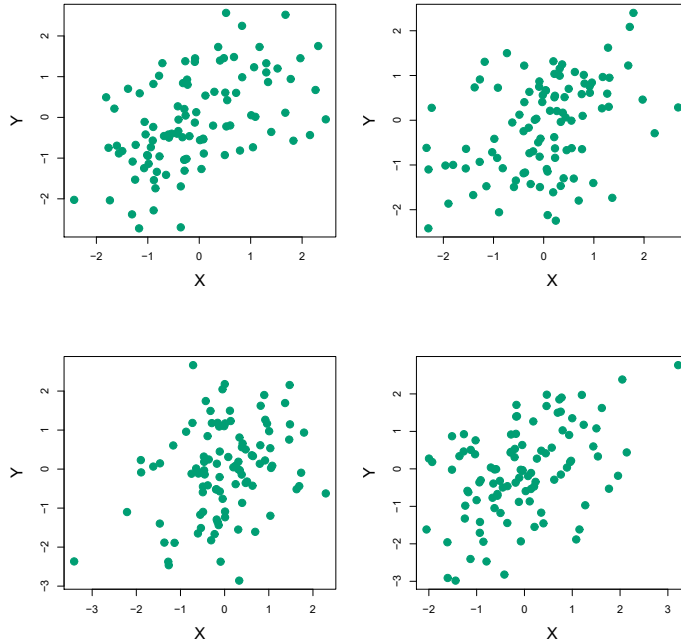
FIGURE 5.9.   *Each panel displays* 100 *simulated returns for investments* X *and* Y*. From left to right and top to bottom, the resulting estimates for* α *are* 0.576*,* 0.532*,* 0.657*, and* 0.651*.*

The mean over all 1,000 estimates for α is

$$\bar{\alpha} = \frac{1}{1000} \sum_{r=1}^{1000} \hat{\alpha}_r = 0.5996,$$

very close to $\alpha = 0.6$, and the standard deviation of the estimates is

$$\sqrt{\frac{1}{1000 - 1} \sum_{r=1}^{1000} (\hat{\alpha}_r - \bar{\alpha})^2} = 0.083.$$

This gives us a very good idea of the accuracy of $\hat{\alpha}$: $\text{SE}(\hat{\alpha}) \approx 0.083$. So roughly speaking, for a random sample from the population, we would expect $\hat{\alpha}$ to differ from α by approximately 0.08, on average.

In practice, however, the procedure for estimating $\text{SE}(\hat{\alpha})$ outlined above cannot be applied, because for real data we cannot generate new samples from the original population. However, the bootstrap approach allows us to use a computer to emulate the process of obtaining new sample sets, so that we can estimate the variability of $\hat{\alpha}$ without generating additional samples. Rather than repeatedly obtaining independent data sets from the population, we instead obtain distinct data sets by repeatedly sampling observations *from the original data set.*

This approach is illustrated in Figure 5.11 on a simple data set, which we call Z, that contains only $n = 3$ observations. We randomly select n observations from the data set in order to produce a bootstrap data set,
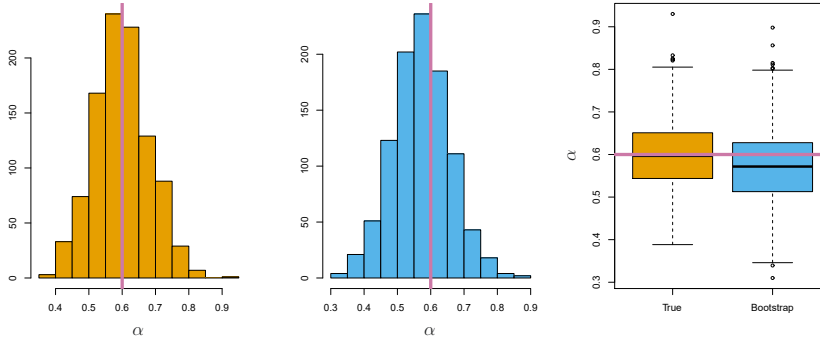
FIGURE 5.10. Left: *A histogram of the estimates of α obtained by generating 1,000 simulated data sets from the true population.* Center: *A histogram of the estimates of α obtained from 1,000 bootstrap samples from a single data set.* Right: *The estimates of α displayed in the left and center panels are shown as boxplots. In each panel, the pink line indicates the true value of α.*

Z^{*1}. The sampling is performed *with replacement*, which means that the same observation can occur more than once in the bootstrap data set. In this example, Z^{*1} contains the third observation twice, the first observation once, and no instances of the second observation. Note that if an observation is contained in Z^{*1}, then both its X and Y values are included. We can use Z^{*1} to produce a new bootstrap estimate for α, which we call $\hat{\alpha}^{*1}$. This procedure is repeated B times for some large value of B, in order to produce B different bootstrap data sets, $Z^{*1}, Z^{*2}, \ldots, Z^{*B}$, and B corresponding α estimates, $\hat{\alpha}^{*1}, \hat{\alpha}^{*2}, \ldots, \hat{\alpha}^{*B}$. We can compute the standard error of these bootstrap estimates using the formula

$$\mathrm{SE}_B(\hat{\alpha}) = \sqrt{\frac{1}{B-1} \sum_{r=1}^{B} \left(\hat{\alpha}^{*r} - \frac{1}{B} \sum_{r'=1}^{B} \hat{\alpha}^{*r'} \right)^2}. \qquad (5.8)$$

with replacement

This serves as an estimate of the standard error of $\hat{\alpha}$ estimated from the original data set.

The bootstrap approach is illustrated in the center panel of Figure 5.10, which displays a histogram of 1,000 bootstrap estimates of α, each computed using a distinct bootstrap data set. This panel was constructed on the basis of a single data set, and hence could be created using real data. Note that the histogram looks very similar to the left-hand panel, which displays the idealized histogram of the estimates of α obtained by generating 1,000 simulated data sets from the true population. In particular the bootstrap estimate $\mathrm{SE}(\hat{\alpha})$ from (5.8) is 0.087, very close to the estimate of 0.083 obtained using 1,000 simulated data sets. The right-hand panel displays the information in the center and left panels in a different way, via boxplots of the estimates for α obtained by generating 1,000 simulated data sets from the true population and using the bootstrap approach. Again, the boxplots have similar spreads, indicating that the bootstrap approach can be used to effectively estimate the variability associated with $\hat{\alpha}$.
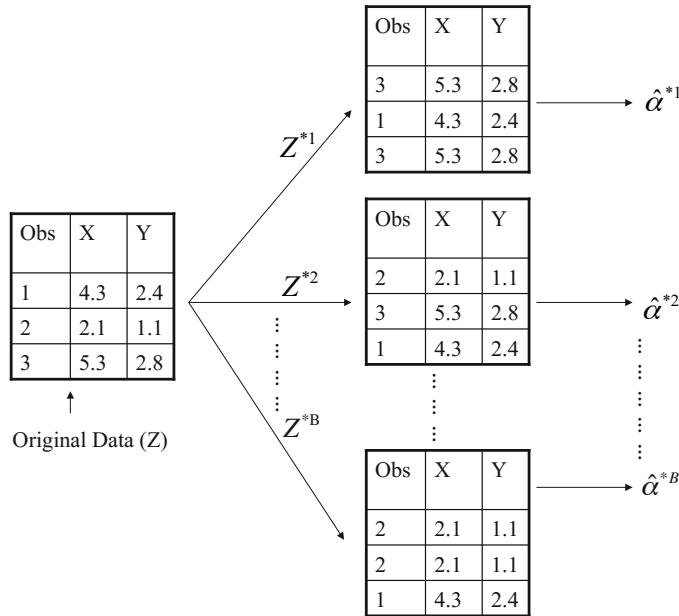
FIGURE 5.11. *A graphical illustration of the bootstrap approach on a small sample containing $n = 3$ observations. Each bootstrap data set contains n observations, sampled with replacement from the original data set. Each bootstrap data set is used to obtain an estimate of α.*

5.3   Lab: Cross-Validation and the Bootstrap

In this lab, we explore the resampling techniques covered in this chapter. Some of the commands in this lab may take a while to run on your computer.

We again begin by placing most of our imports at this top level.

```
In [1]: import numpy as np
        import statsmodels.api as sm
        from ISLP import load_data
        from ISLP.models import (ModelSpec as MS,
                                 summarize,
                                 poly)
        from sklearn.model_selection import train_test_split
```

There are several new imports needed for this lab.

```
In [2]: from functools import partial
        from sklearn.model_selection import \
             (cross_validate,
              KFold,
              ShuffleSplit)
        from sklearn.base import clone
        from ISLP.models import sklearn_sm
```

5.3.1   The Validation Set Approach

We explore the use of the validation set approach in order to estimate the test error rates that result from fitting various linear models on the `Auto` data set.

We use the function `train_test_split()` to split the data into training and validation sets. As there are 392 observations, we split into two equal sets of size 196 using the argument `test_size=196`. It is generally a good idea to set a random seed when performing operations like this that contain an element of randomness, so that the results obtained can be reproduced precisely at a later time. We set the random seed of the splitter with the argument `random_state=0`.

`train_test_split()`

```
In [3]: Auto = load_data('Auto')
        Auto_train, Auto_valid = train_test_split(Auto,
                                                  test_size=196,
                                                  random_state=0)
```

Now we can fit a linear regression using only the observations corresponding to the training set `Auto_train`.

```
In [4]: hp_mm = MS(['horsepower'])
        X_train = hp_mm.fit_transform(Auto_train)
        y_train = Auto_train['mpg']
        model = sm.OLS(y_train, X_train)
        results = model.fit()
```

We now use the `predict()` method of `results` evaluated on the model matrix for this model created using the validation data set. We also calculate the validation MSE of our model.

```
In [5]: X_valid = hp_mm.transform(Auto_valid)
        y_valid = Auto_valid['mpg']
        valid_pred = results.predict(X_valid)
        np.mean((y_valid - valid_pred)**2)
```

```
Out[5]: 23.6166
```

Hence our estimate for the validation MSE of the linear regression fit is 23.62.

We can also estimate the validation error for higher-degree polynomial regressions. We first provide a function `evalMSE()` that takes a model string as well as a training and test set and returns the MSE on the test set.

```
In [6]: def evalMSE(terms,
                     response,
                     train,
                     test):

            mm = MS(terms)
            X_train = mm.fit_transform(train)
            y_train = train[response]

            X_test = mm.transform(test)
            y_test = test[response]
```

```
    results = sm.OLS(y_train, X_train).fit()
    test_pred = results.predict(X_test)

    return np.mean((y_test - test_pred)**2)
```

Let's use this function to estimate the validation MSE using linear, quadratic and cubic fits. We use the `enumerate()` function here, which gives both the values and indices of objects as one iterates over a for loop.

`enumerate()`

```
In [7]: MSE = np.zeros(3)
        for idx, degree in enumerate(range(1, 4)):
            MSE[idx] = evalMSE([poly('horsepower', degree)],
                                'mpg',
                                Auto_train,
                                Auto_valid)
        MSE
```

```
Out[7]: array([23.62, 18.76, 18.80])
```

These error rates are $23.62, 18.76$, and 18.80, respectively. If we choose a different training/validation split instead, then we can expect somewhat different errors on the validation set.

```
In [8]: Auto_train, Auto_valid = train_test_split(Auto,
                                                    test_size=196,
                                                    random_state=3)
        MSE = np.zeros(3)
        for idx, degree in enumerate(range(1, 4)):
            MSE[idx] = evalMSE([poly('horsepower', degree)],
                                'mpg',
                                Auto_train,
                                Auto_valid)
        MSE
```

```
Out[8]: array([20.76, 16.95, 16.97])
```

Using this split of the observations into a training set and a validation set, we find that the validation set error rates for the models with linear, quadratic, and cubic terms are $20.76, 16.95$, and 16.97, respectively.

These results are consistent with our previous findings: a model that predicts `mpg` using a quadratic function of `horsepower` performs better than a model that involves only a linear function of `horsepower`, and there is no evidence of an improvement in using a cubic function of `horsepower`.

5.3.2 Cross-Validation

In theory, the cross-validation estimate can be computed for any generalized linear model. In practice, however, the simplest way to cross-validate in Python is to use `sklearn`, which has a different interface or API than `statsmodels`, the code we have been using to fit GLMs.

This is a problem which often confronts data scientists: "I have a function to do task A, and need to feed it into something that performs task B, so that I can compute $B(A(D))$, where D is my data." When A and B don't naturally speak to each other, this requires the use of a *wrapper*. In the ISLP

wrapper

package, we provide a wrapper, `sklearn_sm()`, that enables us to easily use
the cross-validation tools of `sklearn` with models fit by `statsmodels`.

The class `sklearn_sm()` has as its first argument a model from `statsmodels`.
It can take two additional optional arguments: `model_str` which can be used
to specify a formula, and `model_args` which should be a dictionary of addi-
tional arguments used when fitting the model. For example, to fit a logistic
regression model we have to specify a `family` argument. This is passed as
`model_args={'family':sm.families.Binomial()}`.

Here is our wrapper in action:

```
In [9]: hp_model = sklearn_sm(sm.OLS,
                              MS(['horsepower']))
        X, Y = Auto.drop(columns=['mpg']), Auto['mpg']
        cv_results = cross_validate(hp_model,
                                    X,
                                    Y,
                                    cv=Auto.shape[0])
        cv_err = np.mean(cv_results['test_score'])
        cv_err
```

```
Out[9]: 24.2315
```

The arguments to `cross_validate()` are as follows: an object with the ap-
propriate `fit()`, `predict()`, and `score()` methods, an array of features `X` and
a response `Y`. We also included an additional argument `cv` to `cross_validate()`;
specifying an integer K results in K-fold cross-validation. We have provided
a value corresponding to the total number of observations, which results
in leave-one-out cross-validation (LOOCV). The `cross_validate()` func-

tion produces a dictionary with several components; we simply want the
cross-validated test score here (MSE), which is estimated to be 24.23.

We can repeat this procedure for increasingly complex polynomial fits.
To automate the process, we again use a for loop which iteratively fits
polynomial regressions of degree 1 to 5, computes the associated cross-
validation error, and stores it in the ith element of the vector `cv_error`.
The variable `d` in the for loop corresponds to the degree of the polynomial.
We begin by initializing the vector. This command may take a couple of
seconds to run.

```
In [10]: cv_error = np.zeros(5)
         H = np.array(Auto['horsepower'])
         M = sklearn_sm(sm.OLS)
         for i, d in enumerate(range(1,6)):
             X = np.power.outer(H, np.arange(d+1))
             M_CV = cross_validate(M,
                                   X,
                                   Y,
                                   cv=Auto.shape[0])
             cv_error[i] = np.mean(M_CV['test_score'])
         cv_error
```

```
Out[10]: array([24.2315, 19.2482, 19.3350, 19.4244, 19.0332])
```

As in Figure 5.4, we see a sharp drop in the estimated test MSE between
the linear and quadratic fits, but then no clear improvement from using
higher-degree polynomials.

Above we introduced the `outer()` method of the `np.power()` function. The `outer()` method is applied to an operation that has two arguments, such as `add()`, `min()`, or `power()`. It has two arrays as arguments, and then forms a larger array where the operation is applied to each pair of elements of the two arrays.

`.outer()`
`np.power()`

```
In [11]: A = np.array([3, 5, 9])
         B = np.array([2, 4])
         np.add.outer(A, B)
```

```
Out[11]: array([[ 5,  7],
                [ 7,  9],
                [11, 13]])
```

In the CV example above, we used $K = n$, but of course we can also use $K < n$. The code is very similar to the above (and is significantly faster). Here we use `KFold()` to partition the data into $K = 10$ random groups. We use `random_state` to set a random seed and initialize a vector `cv_error` in which we will store the CV errors corresponding to the polynomial fits of degrees one to five.

`KFold()`

```
In [12]: cv_error = np.zeros(5)
         cv = KFold(n_splits=10,
                    shuffle=True,
                    random_state=0) # use same splits for each degree
         for i, d in enumerate(range(1,6)):
             X = np.power.outer(H, np.arange(d+1))
             M_CV = cross_validate(M,
                                   X,
                                   Y,
                                   cv=cv)
             cv_error[i] = np.mean(M_CV['test_score'])
         cv_error
```

```
Out[12]: array([24.2077, 19.1853, 19.2763, 19.4785, 19.1372])
```

Notice that the computation time is much shorter than that of LOOCV. (In principle, the computation time for LOOCV for a least squares linear model should be faster than for K-fold CV, due to the availability of the formula (5.2) for LOOCV; however, the generic `cross_validate()` function does not make use of this formula.) We still see little evidence that using cubic or higher-degree polynomial terms leads to a lower test error than simply using a quadratic fit.

The `cross_validate()` function is flexible and can take different splitting mechanisms as an argument. For instance, one can use the `ShuffleSplit()` funtion to implement the validation set approach just as easily as K-fold cross-validation.

`Shuffle Split()`

```
In [13]: validation = ShuffleSplit(n_splits=1,
                                    test_size=196,
                                    random_state=0)
         results = cross_validate(hp_model,
                                  Auto.drop(['mpg'], axis=1),
                                  Auto['mpg'],
                                  cv=validation);
         results['test_score']
```