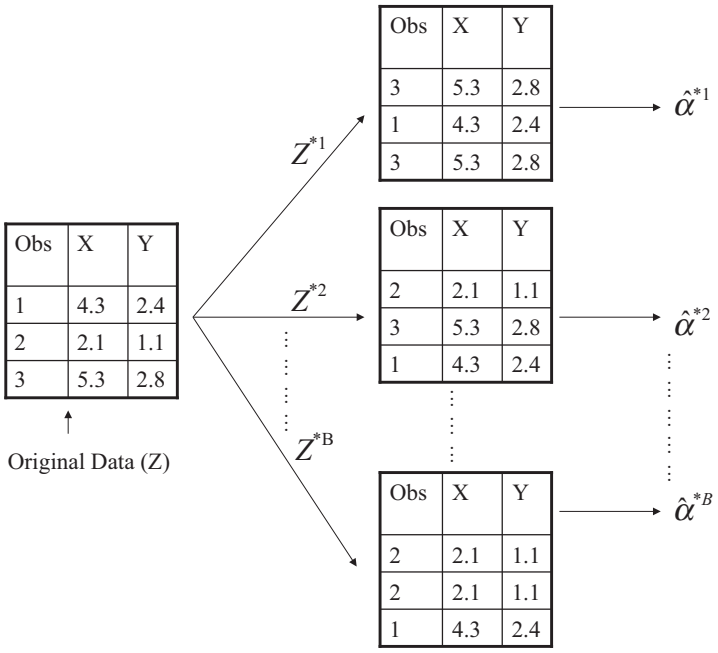

FIGURE 5.11. *A graphical illustration of the bootstrap approach on a small sample containing $n = 3$ observations. Each bootstrap data set contains n observations, sampled with replacement from the original data set. Each bootstrap data set is used to obtain an estimate of α.*

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 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 are quite similar to each other, indicating that the bootstrap approach can be used to effectively estimate the variability associated with $\hat{\alpha}$.

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.

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.

Before we begin, we use the `set.seed()` function in order to set a *seed* for `R`'s random number generator, so that the reader of this book will obtain precisely the same results as those shown below. It is generally a good idea to set a random seed when performing an analysis such as cross-validation that contains an element of randomness, so that the results obtained can be reproduced precisely at a later time.

seed

We begin by using the `sample()` function to split the set of observations into two halves, by selecting a random subset of 196 observations out of the original 392 observations. We refer to these observations as the training set.

sample()

```
> library(ISLR)
> set.seed(1)
> train=sample(392,196)
```

(Here we use a shortcut in the sample command; see `?sample` for details.) We then use the `subset` option in `lm()` to fit a linear regression using only the observations corresponding to the training set.

```
> lm.fit=lm(mpg~horsepower,data=Auto,subset=train)
```

We now use the `predict()` function to estimate the response for all 392 observations, and we use the `mean()` function to calculate the MSE of the 196 observations in the validation set. Note that the `-train` index below selects only the observations that are not in the training set.

```
> attach(Auto)
> mean((mpg-predict(lm.fit,Auto))[-train]^2)
[1] 26.14
```

Therefore, the estimated test MSE for the linear regression fit is 26.14. We can use the `poly()` function to estimate the test error for the quadratic and cubic regressions.

```
> lm.fit2=lm(mpg~poly(horsepower,2),data=Auto,subset=train)
> mean((mpg-predict(lm.fit2,Auto))[-train]^2)
[1] 19.82
> lm.fit3=lm(mpg~poly(horsepower,3),data=Auto,subset=train)
> mean((mpg-predict(lm.fit3,Auto))[-train]^2)
[1] 19.78
```

These error rates are 19.82 and 19.78, respectively. If we choose a different training set instead, then we will obtain somewhat different errors on the validation set.

```
> set.seed(2)
> train=sample(392,196)
> lm.fit=lm(mpg~horsepower,subset=train)
```

```
> mean((mpg-predict(lm.fit,Auto))[-train]^2)
[1] 23.30
> lm.fit2=lm(mpg~poly(horsepower,2),data=Auto,subset=train)
> mean((mpg-predict(lm.fit2,Auto))[-train]^2)
[1] 18.90
> lm.fit3=lm(mpg~poly(horsepower,3),data=Auto,subset=train)
> mean((mpg-predict(lm.fit3,Auto))[-train]^2)
[1] 19.26
```

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 23.30, 18.90, and 19.26, 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 little evidence in favor of a model that uses a cubic function of `horsepower`.

5.3.2 Leave-One-Out Cross-Validation

The LOOCV estimate can be automatically computed for any generalized linear model using the `glm()` and `cv.glm()` functions. In the lab for Chapter 4, we used the `glm()` function to perform logistic regression by passing in the `family="binomial"` argument. But if we use `glm()` to fit a model without passing in the `family` argument, then it performs linear regression, just like the `lm()` function. So for instance,

`cv.glm()`

```
> glm.fit=glm(mpg~horsepower,data=Auto)
> coef(glm.fit)
(Intercept)  horsepower
     39.936      -0.158
```

and

```
> lm.fit=lm(mpg~horsepower,data=Auto)
> coef(lm.fit)
(Intercept)  horsepower
     39.936      -0.158
```

yield identical linear regression models. In this lab, we will perform linear regression using the `glm()` function rather than the `lm()` function because the former can be used together with `cv.glm()`. The `cv.glm()` function is part of the `boot` library.

```
> library(boot)
> glm.fit=glm(mpg~horsepower,data=Auto)
> cv.err=cv.glm(Auto,glm.fit)
> cv.err$delta
    1     1
24.23 24.23
```

The `cv.glm()` function produces a list with several components. The two numbers in the `delta` vector contain the cross-validation results. In this

case the numbers are identical (up to two decimal places) and correspond to the LOOCV statistic given in (5.1). Below, we discuss a situation in which the two numbers differ. Our cross-validation estimate for the test error is approximately 24.23.

We can repeat this procedure for increasingly complex polynomial fits. To automate the process, we use the `for()` function to initiate a *for loop* which iteratively fits polynomial regressions for polynomials of order $i = 1$ to $i = 5$, computes the associated cross-validation error, and stores it in the ith element of the vector `cv.error`. We begin by initializing the vector. This command will likely take a couple of minutes to run.

`for()`
for loop

```
> cv.error=rep(0,5)
> for (i in 1:5){
+ glm.fit=glm(mpg~poly(horsepower,i),data=Auto)
+ cv.error[i]=cv.glm(Auto,glm.fit)$delta[1]
+ }
> cv.error
[1] 24.23 19.25 19.33 19.42 19.03
```

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-order polynomials.

5.3.3 *k-Fold Cross-Validation*

The `cv.glm()` function can also be used to implement k-fold CV. Below we use $k = 10$, a common choice for k, on the `Auto` data set. We once again set a random seed and initialize a vector in which we will store the CV errors corresponding to the polynomial fits of orders one to ten.

```
> set.seed(17)
> cv.error.10=rep(0,10)
> for (i in 1:10){
+ glm.fit=glm(mpg~poly(horsepower,i),data=Auto)
+ cv.error.10[i]=cv.glm(Auto,glm.fit,K=10)$delta[1]
+ }
> cv.error.10
[1] 24.21 19.19 19.31 19.34 18.88 19.02 18.90 19.71 18.95 19.50
```

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, unfortunately the `cv.glm()` function does not make use of this formula.) We still see little evidence that using cubic or higher-order polynomial terms leads to lower test error than simply using a quadratic fit.

We saw in Section 5.3.2 that the two numbers associated with `delta` are essentially the same when LOOCV is performed. When we instead perform k-fold CV, then the two numbers associated with `delta` differ slightly. The

first is the standard k-fold CV estimate, as in (5.3). The second is a bias-corrected version. On this data set, the two estimates are very similar to each other.

5.3.4 The Bootstrap

We illustrate the use of the bootstrap in the simple example of Section 5.2, as well as on an example involving estimating the accuracy of the linear regression model on the `Auto` data set.

Estimating the Accuracy of a Statistic of Interest

One of the great advantages of the bootstrap approach is that it can be applied in almost all situations. No complicated mathematical calculations are required. Performing a bootstrap analysis in `R` entails only two steps. First, we must create a function that computes the statistic of interest. Second, we use the `boot()` function, which is part of the `boot` library, to perform the bootstrap by repeatedly sampling observations from the data set with replacement.

`boot()`

The `Portfolio` data set in the `ISLR` package is described in Section 5.2. To illustrate the use of the bootstrap on this data, we must first create a function, `alpha.fn()`, which takes as input the (X, Y) data as well as a vector indicating which observations should be used to estimate α. The function then outputs the estimate for α based on the selected observations.

```
> alpha.fn=function(data,index){
+ X=data$X[index]
+ Y=data$Y[index]
+ return((var(Y)-cov(X,Y))/(var(X)+var(Y)-2*cov(X,Y)))
+ }
```

This function *returns*, or outputs, an estimate for α based on applying (5.7) to the observations indexed by the argument `index`. For instance, the following command tells `R` to estimate α using all 100 observations.

```
> alpha.fn(Portfolio,1:100)
[1] 0.576
```

The next command uses the `sample()` function to randomly select 100 observations from the range 1 to 100, with replacement. This is equivalent to constructing a new bootstrap data set and recomputing $\hat{\alpha}$ based on the new data set.

```
> set.seed(1)
> alpha.fn(Portfolio,sample(100,100,replace=T))
[1] 0.596
```

We can implement a bootstrap analysis by performing this command many times, recording all of the corresponding estimates for α, and computing