Resampling Methods

Laura Cline

24/10/2021

Contents


```
set.seed(1)
library(ISLR) #Auto, Portfolio datasets
library(boot) #cv.glm()
```

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` dataset.

Before we begin, we use the `set.seed()` function in order to set a *seed* for `R`'s random number generator. 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.

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.

```
# We randomly generate a vector of integers to tell us how to divide up our data
attach(Auto)
train <- sample(392,196)
```

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

```
# Fit the model with the training set, and find the calculate the mean squared error
lm.fit <- lm(mpg ~ horsepower, data=Auto, subset=train)
mean((mpg - predict(lm.fit, Auto))[-train]^2)
```

```
## [1] 23.26601
```

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 above selects only the observations that are not in the training set.

Therefore, the estimated test MSE for the linear regression is 23.27. We can use the `poly()` function to estimate the test error for the quadratic and cubic regressions.

```
# Refit the model using higher order terms
lm.fit2 <- lm(mpg ~ poly(horsepower,2), data=Auto, subset=train)
mean((mpg - predict(lm.fit2, Auto))[-train]^2)
```

```
## [1] 18.71646
```

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

```
## [1] 18.79401
```

The error rates are 18.72 and 18.79 respectively. If we choose a different training set instead, then we will obtain somewhat different errors on the validation set.

```
# If we want to find a different training set, simply change the seed and repeat this process
set.seed(2)
train <- sample(392,196)
```

```
lm.fit <- lm(mpg ~ horsepower, data=Auto, subset=train)
mean((mpg - predict(lm.fit, Auto))[-train]^2)
```

```
## [1] 25.72651
```

```
lm.fit2 <- lm(mpg ~ poly(horsepower,2), data=Auto, subset=train)
mean((mpg - predict(lm.fit2, Auto))[-train]^2)
```

```
## [1] 20.43036
```

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

```
## [1] 20.38533
```

The error terms are consistent across training sets and fits, we can also see that the quadratic term results in the least MSE for each sample.

Using this split of the observations into a training set and validation set, we find that the validation set error rates for the models with linear, quadratic and cubic terms are 25.73, 20.43 and 20.38 respectively.

These results are consistent with our previous findings: a model that predicts `mpg` using a quadratic function of `horsepower` performs better than amodel that involves only a linear function of `horsepower`, and there is little evidence in fabour of a model that uses a cubic function of `horsepower`.

Leave-One-Out Cross-Validation (LOOCV)

The LOOCV estimate can be automatically computed for any generalized linear model using the `glm()` and `cv.glm()` functions. We previously 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.

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.

```
# The LOOCV estimate can be automatically estimated using the cv.glm() function, which is found in the
glm.fit <- glm(mpg ~ horsepower, data=Auto)
cv.err <- cv.glm(Auto, glm.fit)
cv.err$delta
```

```
## [1] 24.23151 24.23114
```

`cv.glm()` provides us a list with several components: The `delta` vector contains the CV results - the average of the n test error rates. The two numbers of 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. 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 compute minutes to run.

```
# Let's repeat this process for increasingly large polynomial terms and see how delta changes
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] 24.23151 19.24821 19.33498 19.42443 19.03321
```

Notice how we create the `cv.error` vector to accommodate the loop. Before the loop begins, we create an empty vector that has the same length as the number of iterations of the loop (5). With each iteration of the loop, a new model is created and a new MSE estimate is found. The indexing variable, i, tells us which iteration we are on and allows us to throw the newest estimate into the correct position in the `cv.error` vector.

We can se e that the average MSE drops off when we change to a quadratic term, then it doesn't really improve with higher degrees.

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` dataset. We once again set a random seed and initialize a vector in which we will store the CV errors corresponing to the polynomial fits of orders one to ten.

```
# The cv.glm() function we used earlier allows for k-fold CV, so we can repeat pretty much the exact sa
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] 24.27207 19.26909 19.34805 19.29496 19.03198 18.89781 19.12061 19.14666
##  [9] 18.87013 20.95520
```

Notice that the computation time is much shorter than that of LOOCV. 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 earlier 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-gold CV estimate. the second is the bias-corrected version. On this dataset, the two estimates are similair to each other.

The Bootstrap

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 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 dataset with replacement.

We will use the `Portfolio` dataset in the `ISLR` library. 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.

```
# Boostraping in R requires the computation of a statistic of interest
# In this example, we write a function alpha.fn() to find the proportion of our money to invest in X (a
alpha.fn <- function(data, index){
  X <- data$X[index]
  Y <- data$Y[index]
  return((var(Y)-cov(X,Y))/(var(Y)-2*cov(X,Y)))
}
```

The function *returns* or outputs an estimate for α based on applying the formula to the observations to the argument `index`. For instance, the following command tells `R` to estimate α using all 100 observations.

```
# Now we randomly select 100 observations (with replacement) from the portfolio to find the bootstraped
set.seed(1)
alpha.fn(Portfolio, sample(100,100, replace=T))
```

```
## [1] 4.201554
```

We can implement a bootstrap analysis by performing this command many times, recording all of the corresponding estimates for α, and computing the resulting standard deviation. However, the `boot()` function automates this approach. Below we produce R = 1,000 bootstrap estimates for α.

```
# To conduct a boostrap analysis we would want to repeat this process, record the estimates for alpha,
#Luckily, boot() does this for us
boot(Portfolio, alpha.fn, R=1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Portfolio, statistic = alpha.fn, R = 1000)
##
##
## Bootstrap Statistics :
##     original    bias    std. error
## t1* 12.28149 -13.24492    124.4232
```

The final output shows that using the original data, $\hat{\alpha} = 12.28149$ and that the bootstrap estimate for $SE(\hat{\alpha})$ is 124.4232.

Estimating the Accuracy of a Linear Regression Model

The bootstrap approach can be used to assess the variability of the coefficient estimates and predictions from a statistical learning method. Here we use the bootstrap approach in order to assess the variability of the estimates for β_0 and β_1, the intercept and slope terms for the linear regression model that uses `horsepower` to predict `mpg` in the `Auto` dataset. We will compare the estimates obtained using the bootstrap to those obtained using the formulas for $SE(\hat{\beta}_0)$ and $SE(\hat{\beta}_1)$.

We first create a simple function, `boot.fn()`, which takes in the `Auto` dataset as well as a set of indices for the observations, and returns the intercept and slope estimates for the linear regression model. We then apply this function to the full set of 392 observations in order to compute the estimates of β_0 and β_1 on the entire dataset using the usual linear regression coefficient estimate formulas. Note that we do not need the `{` and `}` at the beginning and end of the function because it is only one line long.

Bootstraping can be used to assess the variability of coefficient estimates and predictions from a specific learning method. here, we use it to assess the variability of the estimate of the coefficients in a linear regression model that predicts `mpg` using `horsepower`. We first define a function that takes in the `Auto` data, runs a regression and return the estimated coefficients. Then, we randomly generate bootstrapped samples and iterate as many times as we would like.

```
boot.fn <- function(data, index){
  return(coef(lm(mpg~horsepower, data=data, subset=index)))
}
```

```
# Return the coefficients of the model that uses all the observations
set.seed(1)
boot.fn(Auto, 1:392)
```

```
## (Intercept)  horsepower
##  39.9358610  -0.1578447
```

The `boot.fn()` function can also be used to create bootstrap estimates for the intercept and slope terms by randomly sampling from among the observations with replacement. Here, we give an example.

```
# Return the coefficients of the model that samples 392 observations (with replacement) from the origin
boot.fn(Auto, sample(392,392, replace=T))
```

```
## (Intercept)  horsepower
##  40.3404517  -0.1634868
```

Next, we use the `boot()` function to compute the standard slopes of 1,000 bootstrap estimates for the intercept and slope terms.

```
boot(Auto, boot.fn, 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Auto, statistic = boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##       original        bias    std. error
## t1* 39.9358610  0.0549915227 0.841925746
## t2* -0.1578447 -0.0006210818 0.007348956
```

```
summary(lm(mpg~horsepower, data=Auto))$coef # Show this output so we can analyze different types of SEs
```

```
##                Estimate  Std. Error   t value      Pr(>|t|)
## (Intercept) 39.9358610 0.717498656  55.65984 1.220362e-187
## horsepower  -0.1578447 0.006445501 -24.48914  7.031989e-81
```

From this output, we can see that the standard error for estimated intercept and slope terms are 0.8419 and 0.0073 respectively (in the bootstrap). Furthermore, why might there be a difference in the bootstraped SEs and the single linear model? It is because the estimate for variance is biased upwards.

This indicates that the bootstrap estimate for $SE(\hat{\beta}_0)$ is 0.8419, and that the bootstrap estimate for $SE(\hat{\beta}_1)$ is 0.0073. Standard formulas can be used to compute the standard errors for the regression coefficients in a linear model. These can be obtained using the `summary()` function.

```
summary(lm(mpg~horsepower, data=Auto))$coef
```

```
##                Estimate  Std. Error   t value      Pr(>|t|)
## (Intercept) 39.9358610 0.717498656  55.65984 1.220362e-187
## horsepower  -0.1578447 0.006445501 -24.48914  7.031989e-81
```

The standard error estimates for $\hat{\beta}_0$ and$\hat{\beta}_1$ obtained using the formulas are 0.717 for the intercept and 0.0064 for the slope. Interestingly, these are somewhat different from the estimates obtained using bootstrap. Does this indicate a problem with bootstrap. In fact, it suggests the opposite. Recall the standard formulas rely on certain assumptions. For example, they depend on the unknown parameter σ^2, the noise variance. We then estimate σ^2 using the RSS. Now although the formula for the standard errors do not rely on the linear model being correct, the estimate for σ^2 does. We see that there is a non-linear relationship in the data, and so the residuals from a linear fit will be inflated, and so will σ^2. Secondly, the standard formulas assume that the x_i are fixed, and all the variability comes from the variation in the errors ϵ_i. The bootstrap approach does not rely on any of these assumptions, and so it is likely giving a more accurate estimate of the standard errors of $\hat{\beta}_0$ and$\hat{\beta}_1$ than is the `summary()` function.

Below we compute the bootstrap standard error estimates and the standard linear regression estimates that result from fitting the quadratic model to the data. Since the model provides a good fit to the data, there is now a better correspondence between the bootstrap estimates and the standard estimates of $SE(\hat{\beta}_0)$, $SE(\hat{\beta}_1)$ and $SE(\hat{\beta}_2)$.

```
# Let's repeat what we did above using a quadratic term
boot.fn <- function(data, index){
  return(coef(lm(mpg ~ horsepower + I(horsepower^2), data=data, subset=index)))
}
```

```
set.seed(1)
boot(Auto, boot.fn, 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Auto, statistic = boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##         original        bias     std. error
## t1* 56.900099702  3.511640e-02 2.0300222526
## t2* -0.466189630 -7.080834e-04 0.0324241984
## t3*  0.001230536  2.840324e-06 0.0001172164
```

```
summary(lm(mpg~horsepower + I(horsepower^2), data=Auto))$coef
```

```
##                       Estimate   Std. Error   t value      Pr(>|t|)
## (Intercept)      56.900099702 1.8004268063  31.60367 1.740911e-109
## horsepower       -0.466189630 0.0311246171 -14.97816  2.289429e-40
## I(horsepower^2)   0.001230536 0.0001220759  10.08009  2.196340e-21
```