Cross-Validation and the Bootstrap

Laura Cline

15/08/2021

The Validation Set Approach

We explore the use of the validation set approach in order to estimate the test error rate that results 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, so another person can get precisely the same results as shown below. It is generally a good idea to set a random seed when performing an anaysis 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.

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

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] 23.26601
```

Therefore, the estimated test MSE for the linear regression fit is 23.27. We 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] 18.71646
```

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

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

These errors 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.

```
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
```

Using this split of the observations into a training set and a validation set, we find that the valiation 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 a model that involves only a linear function of `horsepower`, and there is little evidence in favour of a model that uses a cubic function of `horsepower`.

Leave-One-Out Cross Validation

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

```
glm.fit=glm(mpg~horsepower, data=Auto)
coef(glm.fit)
```

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

and

```
lm.fit=lm(mpg~horsepower, data=Auto)
coef(lm.fit)
```

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

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.git = glm(mpg~horsepower, data=Auto)


cv.err = cv.glm(Auto, glm.fit)
cv.err$delta
```

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

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. 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.

```
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
```

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.

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 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] 24.27207 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. In principle, the computation time for LOOCV for a least squares linear model should be faster than for k-fold CV, due to the avaliability of the formula for LOOCV; however, unfortunately the `cv.glm()` function does not make use of this formula. We still see evidence that using cubic or higher order polynomial terms leads to lower test error than simply using a quadratic fit.

We saw earlier than 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. The second is the bias-corrected version. On this dataset, the two estimates are very similiar 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 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 dataset with replacement.

The `Portfolio` dataset in the `ISLR` package will be used. This simulated dataset contains 100 returns for each of two assets, X and Y. The data is used to estimate the optimal fraction to invest in each asset to minimize investment risk of the combined portfolio. To illustrate the use of bootstrap on this data, we first much 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)-cor(X,Y))/(var(X)+var(Y)-2*cov(X,Y)))
}
```

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

```
alpha.fn(Portfolio,1:100)
```

```
## [1] 0.6694768
```

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 dataset and recomputing $\hat{\alpha}$, and computing the resulting standard deviation. However, the `boot()` function automates this approach. Below we produce R = 1,000 bootstrap estimates for α.

```
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* 0.6694768 0.002523065   0.1105005
```

The final output shows that using the original data, $\hat{\alpha} = 9.6695$, and that the bootstrap estimate for $SE(\hat{\alpha})$ is 0.110.

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 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 nee the "{" and "}" at the beginning and end of the function because it is only one line long.

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

boot.fn(Auto, 1:392)
```

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

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

```
set.seed(1)

boot.fn(Auto, sample(392, 392, replace=T))
```

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

```
boot.fn(Auto, sample(392,292, replace=T))
```

```
## (Intercept)  horsepower
##  40.5735493  -0.1635441
```

Next, we use the `boot()` function to compute the standard errors 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  0.0547462479 0.841402679
## t2* -0.1578447 -0.0006174947 0.007348811
```

This indicates that the bootstrap estimate for β_0 and $SE(\hat{\beta}_0)$ is 0.84 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 the bootstrap. Does this indicate a problem with the bootstrap? In fact, it suggests the opposite. Recall that 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 the linear fit will be inflated, and so will $\hat{\sigma}^2$. Secondly, the standard formulas assume (somewhat unrealistically) 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 $\hat{\beta}_0$ 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 this model provides a good fit to the data there is not a better correspondence between the bootstrap estimates and the standard estimates of $SE(\hat{\beta}_0)$, $SE(\hat{\beta}_1)$ and $SE(\hat{\beta}_2)$.

```
boot.fn = function(data,index)
  coefficients(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
```

Applied Excercises

The Test Error of a Logistic Regression Using a Validation Set

We previously used logistic regression to predict the probability of `default` using `income` and `balance` on the `Default` datasrt. We will now estimate the test error of the logistic regression model using the validation set approach. Do not forget to set a random seed before beginning your analysis.

Fit a logistic regression model that uses `income` and `balance` to predict `default`.

```
set.seed(0)
Default = na.omit(Default)
```

```r
m0 = glm(default ~ income + balance, data=Default, family="binomial")
summary(m0)
```

```
##
## Call:
## glm(formula = default ~ income + balance, family = "binomial",
##     data = Default)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4725  -0.1444  -0.0574  -0.0211   3.7245
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.154e+01  4.348e-01 -26.545  < 2e-16 ***
## income       2.081e-05  4.985e-06   4.174 2.99e-05 ***
## balance      5.647e-03  2.274e-04  24.836  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2920.6  on 9999  degrees of freedom
## Residual deviance: 1579.0  on 9997  degrees of freedom
## AIC: 1585
##
## Number of Fisher Scoring iterations: 8
```

Using the validation set approach, estimate the test error of this model. In order to do this, you must perform the following steps:

1. Split the sample set into a training set and a validation set.
2. Fit a multiple logistic regression model using only the training observations.
3. Obtain a prediction of default status for each individual in the validation set by computing the posterior probability of default for that individual, and classifying the individual to the `default` category if the posterior probability is greater than 0.5.
4. Compute the validation set error, which is the fraction of the observations in the validation set that are misclassified.

```r
validation_error_est = function(){
  # Predictors are income and balance
  n = dim(Default[1])
  training_samples = sample(1:n, floor(n/2))
  validation_samples = (1:n)[-training_samples]

  m = glm(default ~ income + balance, data=Default, family="binomial", subset=training_samples)

  # Results from "predict" are in terms of log odds or the logit transformation of the probabilities
  predictions = predict(m, newata=Default[validation_samples,])
  default = factor(rep("No", length(validation_samples)), c("No", "Yes"))
  default[predictions > 0] = factor("Yes", c("No", "Yes"))

  validatio_error_rate = mean(default != Default[validation_samples,]$default)
}
```

```
print("Three more estimates of the validation set error would give:")
```

```
## [1] "Three more estimates of the validation set error would give:"
```

```
print( validation_error_est())
```

```
## Warning in 1:n: numerical expression has 2 elements: only the first used
```

```
## Warning in 1:n: numerical expression has 2 elements: only the first used
```

```
## [1] 0.0478
```

```
print( validation_error_est())
```

```
## Warning in 1:n: numerical expression has 2 elements: only the first used
```

```
## Warning in 1:n: numerical expression has 2 elements: only the first used
```

```
## [1] 0.046
```

```
print( validation_error_est())
```

```
## Warning in 1:n: numerical expression has 2 elements: only the first used
```

```
## Warning in 1:n: numerical expression has 2 elements: only the first used
```

```
## [1] 0.0436
```

Now consider a logistic regression model that predicts the probability of `default` using `income`, `balance`, and a dummy variable for `student`. Estimate the test error for this model using the validations set approach. Comment on whether or not including a dummy variable for `student` leads to a reduction in the test error rate.

```
validation_error_est = function(){
  # Predictors are income, balance and student
  n = dim(Default)[1]
  training_samples = sample(1:n,floor(n/2))
  validation_samples = (1:n)[-training_samples]

  m = glm(default ~ income + balance + student, data=Default, family="binomial", subset=training_samples

  # Results from "predict" are in terms of log odds or logit transformation of the probabilities
  predictions = predict(m, newdata=Default[validation_samples,])

  default = factor(rep("No", length(validation_samples)), c("No","Yes"))
  default[predictions > 0] = factor("Yes", c("No","Yes"))

  validation_error_rate = mean(default != Default[validation_samples,]$default)
}

print("Using the predictor student our validation set error is:")
```

```
## [1] "Using the predictor student our validation set error is:"
```

```
print(validation_error_est())
```

```
## [1] 0.027
```

Estimating the Probability of Default Using Income and Balance

We continue to consider the use of a logistic regression model to predict the probability of `default` using `income` and `balance` on the `Default` dataset. In particular, we will now compute the estimates for the standard errors of `income` and `balance` logistic regression coefficients in two different ways: (1) using the bootstrap, and (2) using the standard formula for computing the standard errors in the `glm()` function.

```
set.seed(0)

Default = na.omit(Default)
```

```
# Estimate the base model (to get standard errors of the coefficients)
m0 = glm(default ~ income + balance, data=Default, family="binomial")
summary(m0)
```

```
##
## Call:
## glm(formula = default ~ income + balance, family = "binomial",
##     data = Default)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4725  -0.1444  -0.0574  -0.0211   3.7245
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.154e+01  4.348e-01 -26.545  < 2e-16 ***
## income       2.081e-05  4.985e-06   4.174 2.99e-05 ***
## balance      5.647e-03  2.274e-04  24.836  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2920.6  on 9999  degrees of freedom
## Residual deviance: 1579.0  on 9997  degrees of freedom
## AIC: 1585
##
## Number of Fisher Scoring iterations: 8
```

```
boot.fn = function(data, index){
  m = glm(default ~ income + balance, data=data[index,], family="binomial")
  return (coefficients(m))
}

boot.fn(Default, 1:10000) # test our boot function
```

```
##   (Intercept)        income       balance
## -1.154047e+01  2.080898e-05  5.647103e-03
```

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

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
```

```
## boot(data = Default, statistic = boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##          original        bias     std. error
## t1* -1.154047e+01 -4.029654e-02 4.369085e-01
## t2*  2.080898e-05  2.593598e-07 4.723616e-06
## t3*  5.647103e-03  1.627427e-05 2.274881e-04
```

Implementing Leave-One-Out Cross Validation

Earlier, we saw that the `cv.glm()` function cam be used in order to compute the LOOCV test error estimate. Alternatively, one could compute those quantities using just the `glm()` and `predict.glm()` functions, and a for loop. You will now take this approach in order to compute the LOOCV error for a simple logistic regression model on the `Weekly` dataset.

1) Fit a logistic regression model that predictions `Direction` using `Lag1` and `Lag2`

```
set.seed(0)

m_0 = glm(Direction ~ Lag1 + Lag2, data=Weekly, family="binomial")
summary(m_0)
```

```
##
## Call:
## glm(formula = Direction ~ Lag1 + Lag2, family = "binomial", data = Weekly)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -1.623  -1.261   1.001   1.083   1.506
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.22122    0.06147   3.599 0.000319 ***
## Lag1        -0.03872    0.02622  -1.477 0.139672
## Lag2         0.06025    0.02655   2.270 0.023232 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1496.2  on 1088  degrees of freedom
## Residual deviance: 1488.2  on 1086  degrees of freedom
## AIC: 1494.2
##
## Number of Fisher Scoring iterations: 4
```

2) Fit a logistic regression model that predicts `Direction` using `Lag1` and `Lag2` using all but the first observation.

```
m_loocv = glm(Direction ~ Lag1 + Lag2, data=Weekly[-1,], family="binomial")
summary(m_loocv)
```

```
##
## Call:
## glm(formula = Direction ~ Lag1 + Lag2, family = "binomial", data = Weekly[-1,
##     ])
```

```
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.6258  -1.2617   0.9999   1.0819   1.5071
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.22324    0.06150   3.630 0.000283 ***
## Lag1        -0.03843    0.02622  -1.466 0.142683
## Lag2         0.06085    0.02656   2.291 0.021971 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1494.6  on 1087  degrees of freedom
## Residual deviance: 1486.5  on 1085  degrees of freedom
## AIC: 1492.5
##
## Number of Fisher Scoring iterations: 4
```

3) Use the model from (2) to predict the direction of the first observation. You can do this by predicting that the first observation will go up if P(Direction="Up"|Lag1, Lag2) > 0.5. Was this observation correctly classified?

```
print( sprintf( "Prediction on first sample is %d (1=>Up; 0=>Down)", predict( m_loocv, newdata=Weekly[1
```

```
## [1] "Prediction on first sample is 1 (1=>Up; 0=>Down)"
```

```
print(sprintf("First samples true direction is %s", Weekly[1,]$Direction))
```

```
## [1] "First samples true direction is Down"
```

4) Write a for loop from i = 1 to i = n where n is the number of observations in the dataset, that performs the following steps:

a. Fit a logistic regression model using all but the ith observation to predict `Direction` using `Lag1` and `Lag2`.

b. Compute the posterior probability of the market moving up for the ith observation.

c. Use the posterior probability for the ith observation in order to predict whether or not the market moves up

d. Determine whether or not an error was made in predicting the direction of the ith observation. If an error was made, then indicate this as a 1, and otherwise indicate it as a 0.

```
n = dim(Weekly)[1]
number_of_errors = 0

for (i in 1:n){
  m_loocv = glm(Direction ~ Lag1 + Lag2, data=Weekly[-i,], family = "binomial")

  error_1 = (predict(m_loocv, newdata=Weekly[i,]) > 0) & (Weekly[i,]$Direction == "Down")
  error_2 = (predict(m_loocv, newdata=Weekly[i,]) < 0) & (Weekly[i,]$Direction == "Up")

  if (error_1 | error_2){
    number_of_errors = number_of_errors + 1
  }
```

```
}
```

5) Take the average of the n numbers obtained in the (4d) in order to obtain the LOOCV estimate for the test error.

```
print(sprintf("LOOCV test error rate = %10.6f", number_of_errors/n))
```

```
## [1] "LOOCV test error rate =   0.449954"
```

Cross-Valdidation on a Simulated Dataset

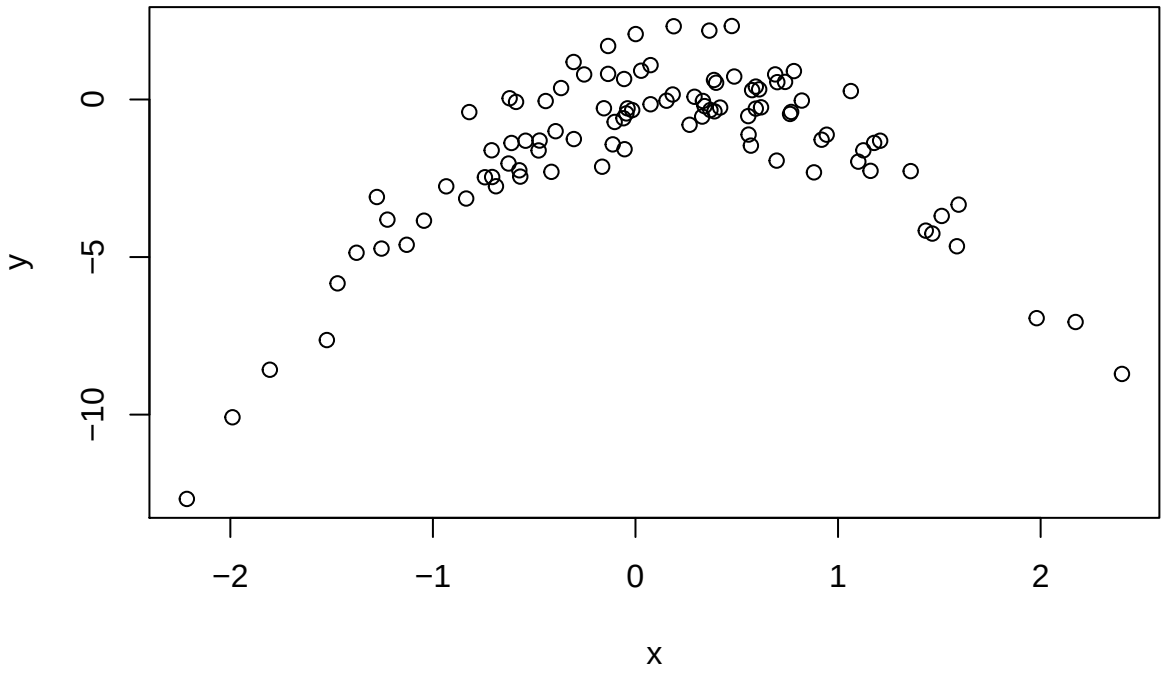
1) Generate a simulated dataset

```
set.seed(1)

x = rnorm(100)
y = x - 2 * x^2 + rnorm(100)
```

2) Create a scatterplot of X against Y

```
plot(x,y)
```



3) Compute the LOOCV errors that result from fitting the following four models using least squares:

a. $Y = \beta_0 + \beta_1 X + \epsilon$
b. $Y = \beta_0 +$
$beta_1 X + beta_2 X^2 + \epsilon$
c. $Y = \beta_0 +$
$beta_1 X + beta_2 X^2 + beta_3 X^3 + \epsilon$
d. $Y = \beta_0 +$
$beta_1 X + beta_2 X^2 + beta_3 X^3 + beta_4 X^4 + \epsilon$

```
DF = data.frame(y=y, x=x)
```

```
m_a = glm(y ~ x, data=DF)
cv.err = cv.glm(DF, m_a)
print(sprintf("Model (a): cv output = %10.6f", cv.err$delta[1]))
```

```
## [1] "Model (a): cv output =   7.288162"
```

```
m_b = glm(y ~ x + I(x^2), data=DF)
cv.err = cv.glm(DF, m_b)
print(sprintf("Model (b): cv output = %10.6f", cv.err$delta[1]))
```

```
## [1] "Model (b): cv output =   0.937424"
```

```
m_c = glm(y ~ x + I(x^2) + I(x^3), data=DF)
cv.err = cv.glm(DF, m_c)
print(sprintf("Model (c): cv output = %10.6f", cv.err$delta[1]))
```

```
## [1] "Model (c): cv output =   0.956622"
```

```
m_d = glm(y ~ x + I(x^2) + I(x^3) + I(x^4), data=DF)
cv.err = cv.glm(DF, m_d)
print(sprintf("Model (d): cv output = %10.6f", cv.err$delta[1]))
```

```
## [1] "Model (d): cv output =   0.953905"
```

Using Bootstrap on the Boston dataset

We will now consider the `Boston` housing dataset, from the `MASS` library.

1) Based on this dataset, provide an estimate for the population mean of `medv`. Call this estimate $\hat{\mu}$

```
library(MASS)
Boston = na.omit(Boston)
mu_hat = mean(Boston$medv)
mu_hat
```

```
## [1] 22.53281
```

2) Provide an estimate of the standard error of $\hat{\mu}$. We compute the standard error of the sample mean by dividing the sample standard deviation by the square root of the number of observations.

```
n = dim(Boston)[1]
mu_se = sd(Boston$medv)/sqrt(n)
mu_se
```

```
## [1] 0.4088611
```

3) Now estimate the standard deviation of $\hat{\mu}$ using the bootstrap.

```
mean_boot.fn = function(data, index){
  mean(data[index])
}

boot(Boston$medv, mean_boot.fn, 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Boston$medv, statistic = mean_boot.fn, R = 1000)
```

```
##
##
## Bootstrap Statistics :
##     original      bias    std. error
## t1* 22.53281 0.007796838   0.4096149
```

4) Based on this dataset, provide an estimate of $\hat{\mu}_{med}$, for the median value of `medv` in the population.

```
median(Boston$medv)
```

```
## [1] 21.2
```

```
median_boot.fn = function(data,index){
  median(data[index])
}

boot(Boston$medv, median_boot.fn, 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Boston$medv, statistic = median_boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##     original  bias    std. error
## t1*     21.2 -0.0396   0.3770862
```

5) Based on this dataset, provide an estimate for the tenth percentile of the `medv` in the Boston suburbs. Call this quantity $\hat{\mu}_{0.1}$.

```
quantile(Boston$medv, probs=c(0.1))
```

```
##   10%
## 12.75
```

```
ten_percent_boot.fn = function(data, index) {
  quantile(data[index], probs=c(0.1))
}

boot(Boston$medv, ten_percent_boot.fn, 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Boston$medv, statistic = ten_percent_boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##     original  bias    std. error
## t1*    12.75 0.01885   0.4927728
```