Workshop: Introduction to Statistical Learning, Part II- Cross-Validation and Bootstrap for estimating test error

Mohammad Jafari Jozani

February 14, 2021

Contents

1 Test Error

Often, we want an accurate **estimate of the test error** of our method (e.g., linear regression) for

- **Predictive assessment:** to get an absolute understanding of the magnitude of errors we should expect in making future predictions
- **Model/method selection:** to choose among different models/methods, attempting to minimize test error

As we discussed earlier, an important application of statistics concerns prediction. Prediction and estimations are close cousins but they are not twins. The **training error**

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

is not a good estimate of the test error of our method and it is in general **too optimistic** as an estimate of the test error. The reason is that every model is too optimistic about how well it will actually predict. Model parameters are often estimated to make $\hat{Y}_i$ close to Y_i, $i = 1, \ldots, n$, in the first place, hence training error is not a good estimate of the test error.

Also, importantly, the more **complex/adaptive** the method, the more optimistic its training error is as an estimate of test error

Consider a toy example, where the underlying true relationship between X and Y is a quadratic model given by

$$Y = -3 + 3X\sin(2\pi X) - 1.3X^2\cos(X) + \epsilon, \quad \text{with} \quad \epsilon \sim N(0, 2).$$

Suppose $X \sim U(-2, 5)$ and we have generated a training data of size $n = 15$ and an independent test data of size $n = 15$ from the underlying model. Here we show these two data sets

```
set.seed(2020)
n = 15
x = sort(runif(n, -2, 5))
y = -3+ 3*x*sin(2*pi*x)-1.3*x^2*cos(x) + rnorm(n, 0, 2)
x0 = sort(runif(n, -2, 5))
y0 = -3+ 3*x0*sin(2*pi*x0)-1.3*x0^2*cos(x0) + rnorm(n, 0, 2)
par(mfrow=c(1,2))
xlim = range(c(x,x0)); ylim = range(c(y,y0))
plot(x, y, xlim=xlim, ylim=ylim, main="Training data", pch=20)
plot(x0, y0, xlim=xlim, ylim=ylim, main="Test data", pch=20, col="red")
```

We first fit a 3rd degree polynomial regression model to our training data. It is easy to calculate the training error for the model given by

$$\text{Training Error} = \frac{1}{15}\sum_{i=1}^{15}(Y_i - \hat{Y}_i)^2 = 15.768.$$

Also, for the new test data $Y_{0i}, i = 1, \ldots, 15$, we calculate the test error given by

$$\text{Test Error} = \frac{1}{15}\sum_{i=1}^{15}(Y_{0i} - \hat{Y}_{0i})^2 = 53.586.$$

As we observe the model performs relatively good on the test data. However, still the training error is underestimating the test error.

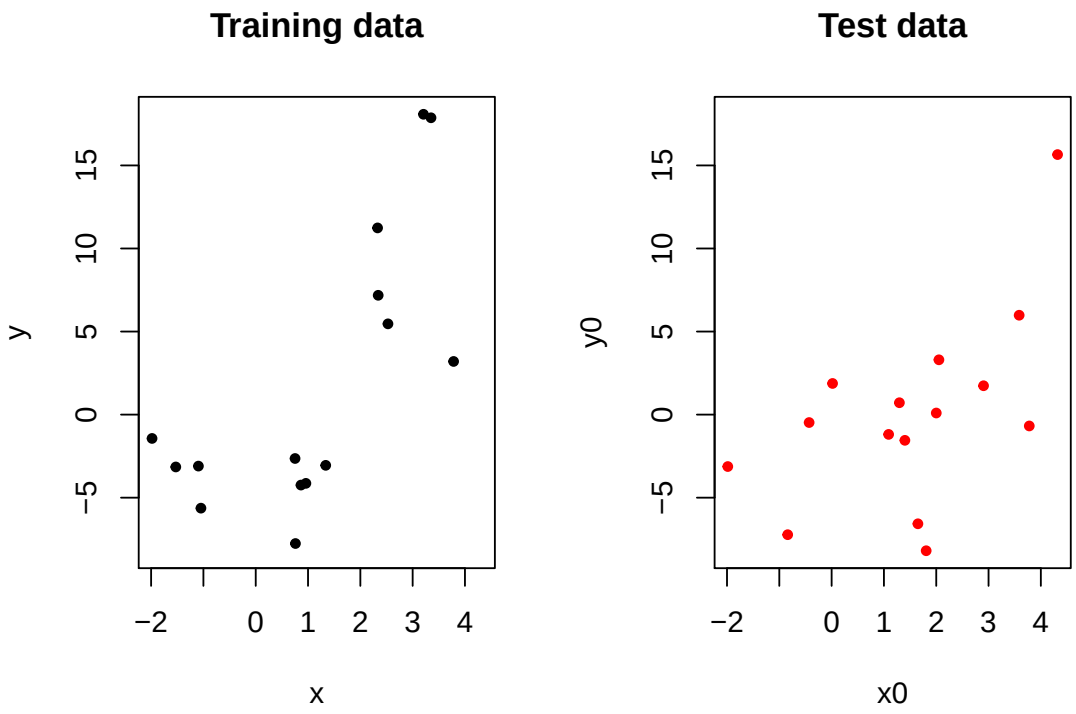


Figure 1: Scatter plots of training and test data of the same size generated from a nonlinear model

```
# Training and test errors for a simple linear model
lm.3 = lm(y ~ poly(x, 3))
yhat.3 = predict(lm.3, data.frame(x=x))
train.err.3 = mean((y-yhat.3)^2)
y0hat.3 = predict(lm.3, data.frame(x=x0))
test.err.3 = mean((y0-y0hat.3)^2)

par(mfrow=c(1,2))
plot(x, y, xlim=xlim, ylim=ylim, main="Training data", pch=20)
lines(x, yhat.3, col=2, lwd=2)
text(1, 15, label=paste("Training error:", round(train.err.3,3)))

plot(x0, y0, xlim=xlim, ylim=ylim, main="Test data", pch=20, col="red")
lines(x, yhat.3, col=3, lwd=2)
text(1,15, label=paste("Test error:", round(test.err.3,3)))
```

Now, we fit a 6th degree polynomial regression to our training data. Here we can easily observe that the more complex polynomial model fits better to our training data but the performance of the fitted model in our training data seems to be very optimistic. This can easily be revealed by using it to predict the response values in the test data. By doing this, we observe that the test error is severely bigger than the training error.

```
# Training and test errors for a 6th order polynomial regression
lm.6 = lm(y ~ poly(x,6))
yhat.6 = predict(lm.6, data.frame(x=x))
train.err.6 = mean((y-yhat.6)^2)
y0hat.6 = predict(lm.6, data.frame(x=x0))
test.err.6 = mean((y0-y0hat.6)^2)

par(mfrow=c(1,2))
```

Figure 2: Scatter plots of training and test data with a fitted polynomial of degree 3 fitted on the training data and depicted on the test data

```
xx = seq(min(xlim), max(xlim), length=100)

plot(x, y, xlim=xlim, ylim=ylim, main="Training data", pch=20)
lines(xx, predict(lm.6, data.frame(x=xx)), col=2, lwd=2)
text(0, 12, label=paste("Training error:", round(train.err.6,3)))

plot(x0, y0, xlim=xlim, ylim=ylim, main="Test data", pch=20, col="red")
lines(xx, predict(lm.6, data.frame(x=xx)), col=3, lwd=2)
text(0, 12, label=paste("Test error:", round(test.err.6,3)))
```

In this toy example, we were doing a simulation study and had access to a test data. In practice, one does not have access to such data and should find a way to estimate the test error. In the absence of a very large designated test set that can be used to directly estimate the test error, a number of techniques can be used to estimate this quantity using the available training error. One approach is to adjust the training error using a penalty term to better estimate the test error. Another approach which is the topic of this chapter estimates the test error rate by holding out a subset of the training observation from the fitting process, and then applying the statistical learning method to those held out observations.

2 Different methods for estimating the Test Error

Here we discuss a number of methods that are often used to estimate the test error. The idea behind most of these methods is based on creating proper hold-out datasets and using the model fitted on the training data (not hold-out) to predict the response of the hold-out dataset.

2.1 Sample-splitting

Given a training data set, an effective approach that has been widely used in statistics is the **sample-splitting** approach. In this approach, one can proceed as follow:

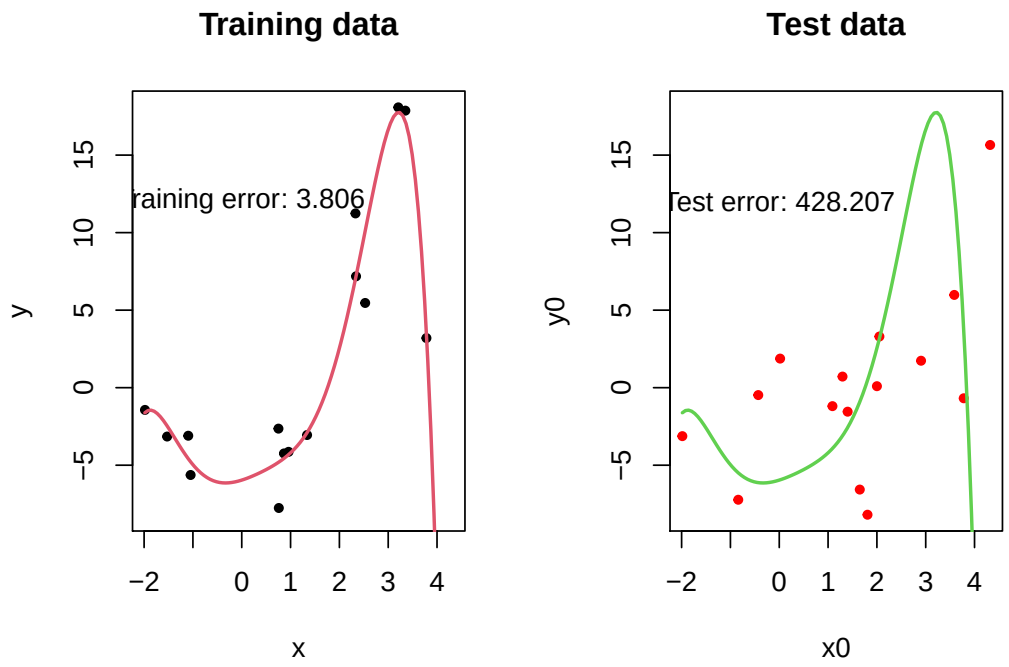


Figure 3: Scatter plots of training and test data with a fitted polynomial of degree 6 fitted on the training data and depicted on the test data

- Split the data set into two parts as training data and hold-out data
- Train the model/method to the training dataset.
- Make predictions on hold-out data
- Evaluate observed test error

As an example, consider again our toy example and suppose we have generated a sample of size $n = 100$ from the underlying population.

```
set.seed(2020)
n = 100
x = sort(runif(n, -2, 5))
y = -3+ 3*x*sin(2*pi*x)-1.3*x^2*cos(x) + rnorm(n, 0, 2)
toy.data <- data.frame(x=x, y=y)
n = nrow(toy.data)

# Split data in half, randomly
inds = sample(rep(1:2, length=n))

train.toy = toy.data[inds==1,] # Training data
test.toy = toy.data[inds==2,] # Test data

plot(toy.data$x, toy.data$y, pch=c(21,19)[inds], main="Sample-splitting", xlab="x", ylab="y")
legend("topleft", legend=c("Training","Test"), pch=c(21,19))
```

Here, we train a cubic well as a 6th degree polynomial regression model to our training data and predict the second half of the data (as hold out) to evaluate the test error.

```
# Train on the first half
lm.3 = lm(y ~ poly(x, 3), data=train.toy)
lm.6 = lm(y ~ poly(x,6), data=train.toy)
```

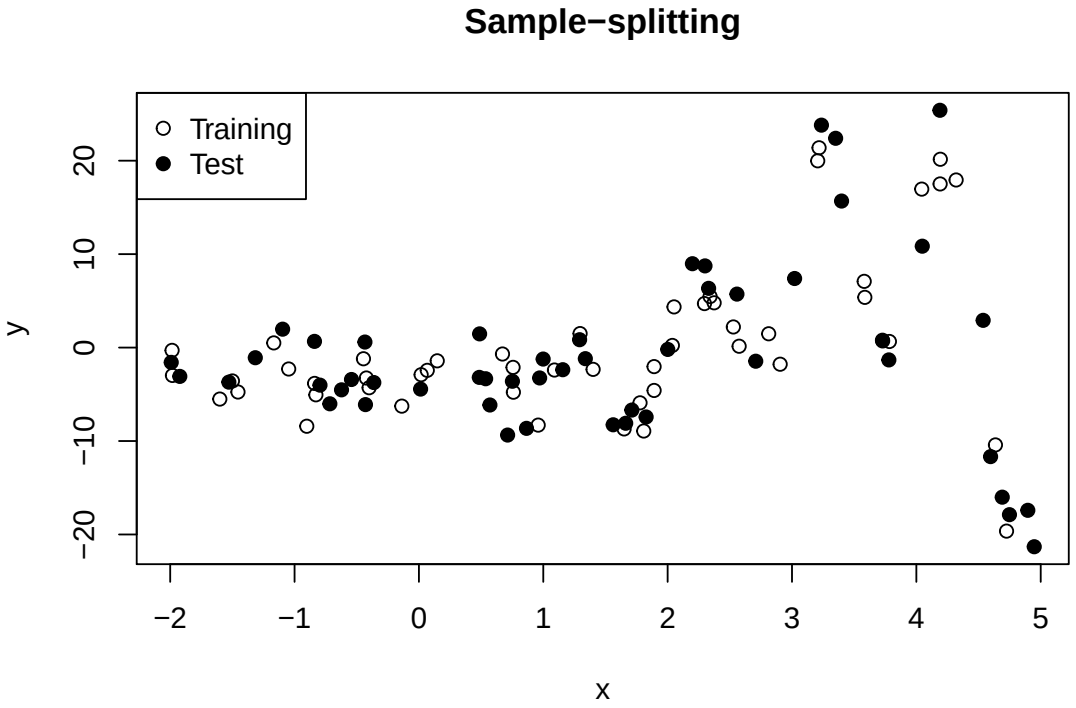


Figure 4: Scatter plots of a sample of size 100 taken from the model and randomly divided into training and test data sets

```r
# Predict on the second half, evaluate test error
pred.3 = predict(lm.3, data.frame(x=test.toy$x))
pred.6 = predict(lm.6, data.frame(x=test.toy$x))

test.err.3 = mean((test.toy$y - pred.3)^2)
test.err.6 = mean((test.toy$y - pred.6)^2)

# Plot the results
par(mfrow=c(1,2))
xx = seq(min(toy.data$x), max(toy.data$x), length=100)

plot(toy.data$x, toy.data$y, pch=c(21,19)[inds], xlab="", ylab="",
     main="Sample-splitting, Degree=3")
lines(xx, predict(lm.3, data.frame(x=xx)), col=2, lwd=2)
legend("topleft", legend=c("Training","Test"), pch=c(21,19))
text(1, -15, label=paste("Test error:", round(test.err.3,3)))

plot(toy.data$x, toy.data$y, pch=c(21,19)[inds], xlab="", ylab="",
     main="Sample-splitting, Degree=6")
lines(xx, predict(lm.6, data.frame(x=xx)), col=3, lwd=2)
legend("topleft", legend=c("Training","Test"), pch=c(21,19))
text(1, -15, label=paste("Test error:", round(test.err.6,3)))
```

We could use this approach to decide the degree of a polynomial regression that best fits the data by choosing a degree that gives the smallest test error estimate. To this end, one can proceed as follow:

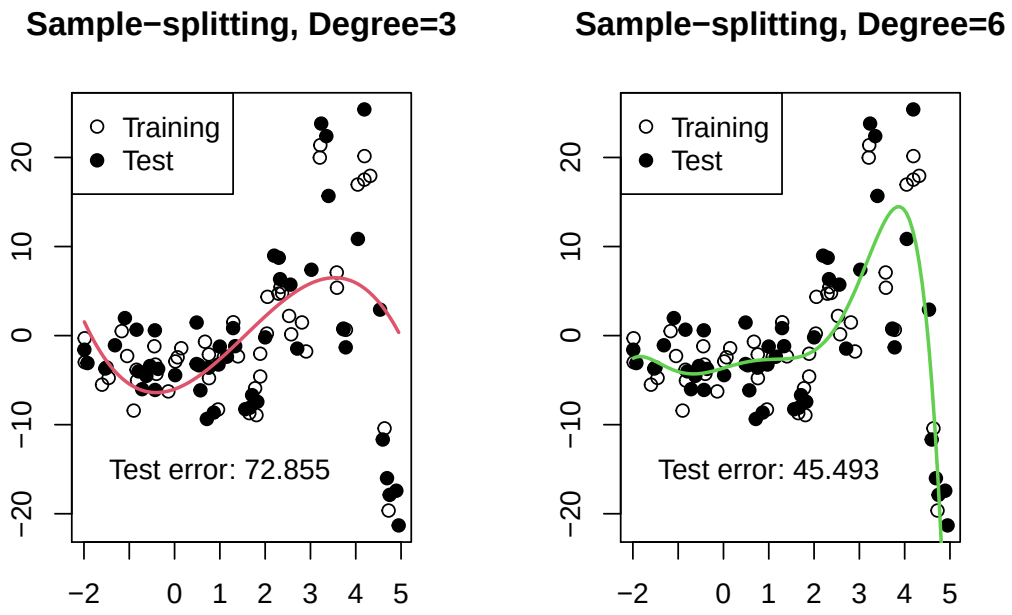


Figure 5: Fitted polynomials of degree 3 and 5 on training portion of our data set and using them to predict the response values associated with the hold-out data

```
p.max <- 8 #the maximum degree of polynomial regression
test.err <- rep(0, p.max)

for(p in 1:p.max){
lm.p = lm(y ~ poly(x, p), data=train.toy)
pred.p = predict(lm.p, data.frame(x=test.toy$x))
test.err[p] = mean((test.toy$y - pred.p)^2)
}

plot(1:p.max, test.err, type="b", pch=20, col="red",
     xlab="Degree of Polynomial",
     ylab="Test Error Estimate")
```

Note that, however, this approach highly depends on how splitting is done. Here we have repeated the data splitting a few times and have calculated the test-error estimates for different values of p.

The splitting set approach is conceptually simple and is easy to implement and might be effective whenever it is possible to perfrom. But it has two potential drawbacks:

1. As shown above, the spiting approach for estimating test error can be highly variable, depending on precisely which observations are included in the training set and which observations are included in the test set.
2. In this approach, only a subset of observations are used to fit the model. Since statistical methods tend to perform worse when trained on fewer observations, this suggests that the estimate we obtain in this approach overestimates the true test error for the model fit on the entire data set.

A refinement of splitting sample approach is cross-validation that addresses these two issues.

2.2 Cross-validation

Ideally, if we had enough data, we would set aside a validation set and use it to assess the performance of our prediction model. But this approach is not often possible and even if possible, it estimates the test error when

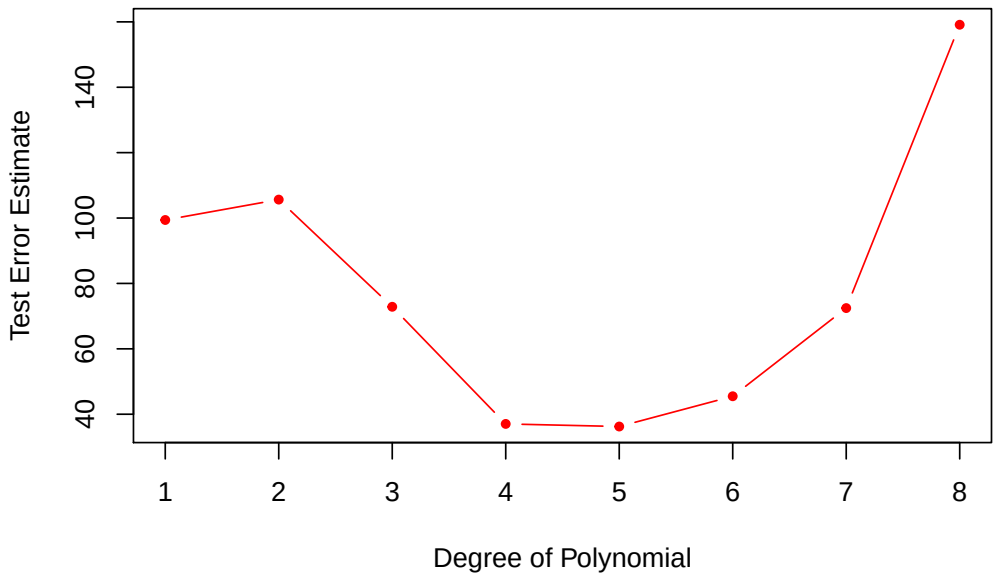


Figure 6: Test error pertinent to fitted polynomials of degrees up to 8 when data are splitted are random into two halves

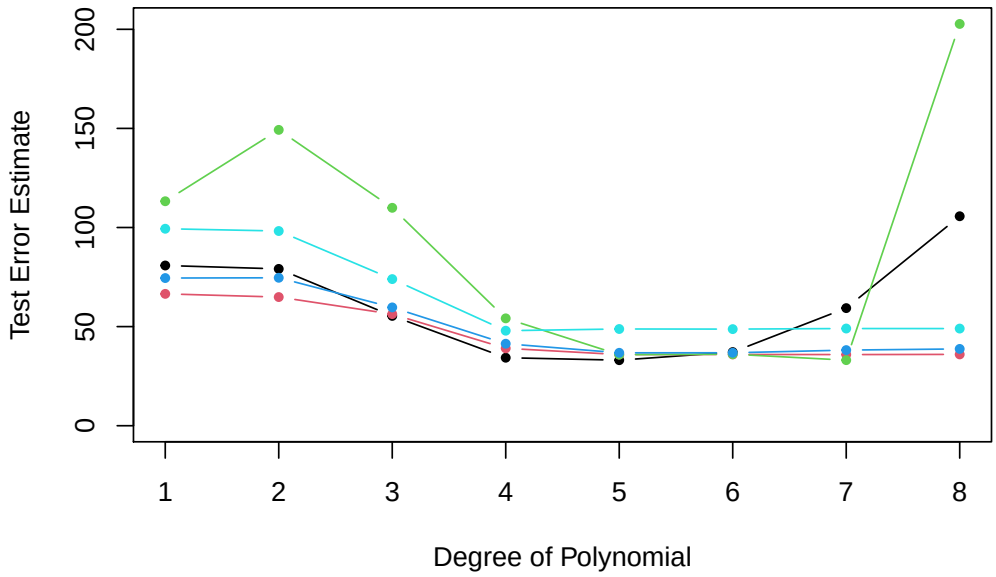


Figure 7: Test error estimates for fitted polynomials of degrees up to 8 when 5 different sample splitting in halves are perfromed on our original dataset

the model/method is trained on **less data** (say, roughly half as much). Probably, the simplest and most widely used method of estimating test error is cross-validation. Consider a model $m(\mathbf{x}) = \mathbb{E}[Y|\mathbf{X} = \mathbf{x}]$ which we would like to estimate using our training data

$$\mathcal{T} = \{(\mathbf{X}_i, Y_i) \ : \ i = 1, \ldots, n\}.$$

Suppose $\hat{Y}_i = \hat{m}(\mathbf{X}_i)$ be the estimated function using the training sample.

2.2.1 Leave-One-Out Cross Validation (LOOCV)

The leave-one-out cross validation starts by removing the ith observation from the training sample and uses

$$\mathcal{T}^{(-i)} = \mathcal{T} - \{(\mathbf{X}_i, Y_i)\}, \quad i = 1, \ldots, n,$$

to obtain the fitted value for data point i given by $\hat{m}^{(-i)}(\mathbf{X}_i) = \hat{m}_i^{(-i)} = \hat{Y}_i^{(-i)}$, where the subscript $(-i)$ indicates that point i was left out in calculating this fit. The LOOCV score of the model is

$$\text{LOOCV} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i^{(-i)})^2 \tag{1}$$

Many more old-fashioned regression textbooks look at nLOOCV, and call it PRESS, *predictive residual sum of squares*.

The idea of LOOCV is very simple. We want to know if our model can generalize to new data. Leaving out each point in turn ensures that the set of points on which we try to make predictions is just as representative of the whole population as the original sample was. LOOCV is an unbiased estimate of the generalization error. However, its variance is high, as by changing the sample, the LOOCV score will change. One of the reason is that $\mathcal{T}^{(-i)}$'s are almost the same when calculating LOOCV score. So, the variability of the samples does not exist in its calculation and LOOCV does not account for that. In the K-fold cross validation as we will see shortly, this problem is relatively solved, however we need trade-off a little bit of bias to reduce the variability of LOOCV.

Re-estimating the model n times using

$$\mathcal{T}^{(-1)}, \mathcal{T}^{(-2)}, \ldots, \mathcal{T}^{(-n)},$$

would be seriously time-consuming, especially when n is large and $\mathbf{X}$ is high-dimensional. Fortunately, for linear smoother of the form

$$\hat{Y} = H\mathbf{Y},$$

there is a shortcut formula that requires only fitting the model to the whole data set $\mathcal{T}$ once and calculate the LOOCV score by re-weighting its individual errors. In other words, for linear smoothers, we have

$$\text{LOOCV} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{Y_i - \hat{Y}_i}{1 - H_{ii}} \right)^2 \tag{2}$$

Note that using the whole training data set $\mathcal{T}$, we have

$$\begin{pmatrix} \hat{Y}_1 \\ \hat{Y}_2 \\ \vdots \\ \hat{Y}_n \end{pmatrix} = \begin{pmatrix} H_{11} & H_{12} & \ldots & H_{1n} \\ H_{21} & H_{22} & \ldots & H_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ H_{n1} & H_{n2} & \ldots & H_{nn} \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix},$$

and

$$\hat{Y}_i = \sum_{j=1}^{n} H_{ij} Y_j.$$

Now, let $\hat{Y}_i^{(-i)}$ be the fitted value using $T^{(-i)}$. It is easy to see that $\mathbf{Y}^{(-i)}$ is also the fitted model to

$$\mathcal{T}_i^* = \{(\mathbf{X}_j, Z_j^i) \ : \ j = 1, \ldots, n\},$$

with

$$Z_j^i = \begin{cases} Y_j & j \neq i \\ \hat{Y}_i^{(-i)} & j = i. \end{cases}$$

To see this, note that

$$\sum_{j \neq i} (Y_j - \hat{Y}_j^{(-i)})^2 = \sum_{j=1}^{n} (Z_j^i - \hat{Y}_j^{(-i)})^2.$$

So, one can write

$$\hat{Y}_i^{(-i)} = \sum_{j=1}^{n} H_{ij} Z_j^i.$$

Now, we can write

$$\begin{aligned} \hat{Y}_i - \hat{Y}_i^{(-i)} &= \sum_{j=1}^{n} H_{ij}(Y_j - Z_j) \\ &= H_{ii}(Y_i - \hat{Y}_i^{(-i)}), \end{aligned}$$

and so

$$\hat{Y}_i = H_{ii} Y_i + (1 - H_{ii}) \hat{Y}_i^{(-i)}.$$

Now, it is easy to see that

$$Y_i - \hat{Y}_i = (1 - H_{ii})(Y_i - \hat{Y}_i^{(-i)}),$$

or equivalently

$$Y_i - \hat{Y}_i^{(-i)} = \frac{Y_i - \hat{Y}_i}{1 - H_{ii}}.$$

This results in

$$\text{LOOCV} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i^{(-i)})^2 = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{Y_i - \hat{Y}_i}{1 - H_{ii}} \right)^2.$$

In this shortcut formula for LOOCV, the numerator inside the square is the residual of the model fit to the full data. This gets divided by $1 - H_{ii}$, which is obtained after we fit the model to the whole data set. In this formula, the denominator forces the residuals for high-leverage points count more, and those for low-leverage points count less. If the model is going out of its way to match Y_i (high leverage H_{ii}) and still can not fit it, that is worse than the same sized residual at a point the model does not really care about (low leverage).

Note that, the gap between LOOCV and the MSE can be thought of as a penalty or an estimate of the optimism of the model.
In many cases, such as the multiple linear regression with p predictors Trace(H) $= \sum_{i=1}^{n} H_{ii} = p + 1$. In such cases, one can estimate H_{ii}'s with their average given by

$$\lambda = \frac{1}{n} \sum_{i=1}^{n} H_{ii} = \frac{p+1}{n}.$$

Using a Taylor series expansion, we get

$$\frac{1}{(1 - H_{ii})^2} \approx \frac{1}{(1 - \lambda)^2} \approx 1 + 2\lambda,$$

and accordingly

$$\begin{aligned}\text{LOOCV} &= \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Y_i - \hat{Y}_i}{1-H_{ii}}\right)^2 \\ &\approx \frac{1}{n}\sum_{j=1}^{n}(Y_j - \hat{Y}_j)^2(1+2\lambda) \\ &= \text{MSE} + \frac{2(p+1)}{n}MSE \\ &= \text{MSE} + \frac{2(p+1)}{n}\hat{\sigma}^2.\end{aligned}$$

This is inline with the expression we obtained in the Chapter pertinent to the Degree of Freedom, i.e.,

$$\widehat{\text{Test Error}} = \text{MSE} + \text{Optimisim of the Model}.$$

Remark 1. *The idea of LOOCV can also be applied for any model that we might use for probabilistic prediction. In such situations, instead of calculating the mean squared error, one can measure the negative log-probability density the model assigns to the actual left-out data point. When the underlying model is normally distributed, this reduces to using MSE and results will be similar to what we explained earlier.*

It is known that as $n \to \infty$ the expected out of sample MSE of the model picked by LOOCV is close to that of the best model considered. This is a very general result that is valid under very mild conditions and does not need normality of the noise term, etc. On the other hand, it is also shown that as $n \to \infty$, if the true model is among those being compared, LOOCV will tend to pick a strictly larger model than the truth. In other words, LOOCV is not consistent for model selection, as it tends to prefer more complex models even if the true model is among the pool of models to be studies. This is partly due to the fact that LOOCV gives unbiased estimates of the generalization error, but it has nothing to do with the variance of the estimates. Models with more parameters have higher variance, and the penalty applied by LOOCV is not strong enough to overcome the chance of capitalizing on that variance. Note that, cross-validation does not estimate the conditional error, with the training set $\mathcal{T}$ held fixed. Instead, they provide good estimates of the expected prediction error.

Remark 2. *As in LOOCV the training datasets $\mathcal{T}^{(-1)}, \mathcal{T}^{(-2)}, \ldots,$ and $\mathcal{T}^{(-n)}$ are almost identical, then the corresponding fitted models are highly (positively) correlated, hence making the LOOCV score to have a high variance. So, the test error estimate obtained via LOOCV has a high variance.*

Let us look at our toy example and see how LOOCV can be used to perform model selection by estimating the test error. To this end, we fit polynomial regression models up to 8th degree to our data set generated from

$$Y = -3 + 3X\sin(2\pi X) - 1.3X^2\cos(X) + \epsilon, \quad \text{with} \quad \epsilon \sim N(0,2),$$

and where $X \sim U(-2,5)$. We generate a sample of size $n = 80$ from the population.

```
set.seed(2020)
n  <-  100
p.max <- 8
x  <-  sort(runif(n, -2, 5))
y = -3+ 3*x*sin(2*pi*x)-1.3*x^2*cos(x) + rnorm(n, 0, 2)
train <- data.frame(x=x, y=y)

LOOCV <- rep(0, p.max)

for(p in 1:p.max){
```

```
  err<- 0
  for( i in 1:n){
  fitte.model <- lm(y~poly(x, p), data = train[-i, ])
  pred.i <- predict(fitte.model, data.frame(x=train[i, 1]))
  err <- err + (pred.i - train[i, 2])^2
  }
LOOCV[p] <- err/n
}

plot(1:p, LOOCV, type="b", pch=20, col="red", xlab="Degree of Polynomial")
```

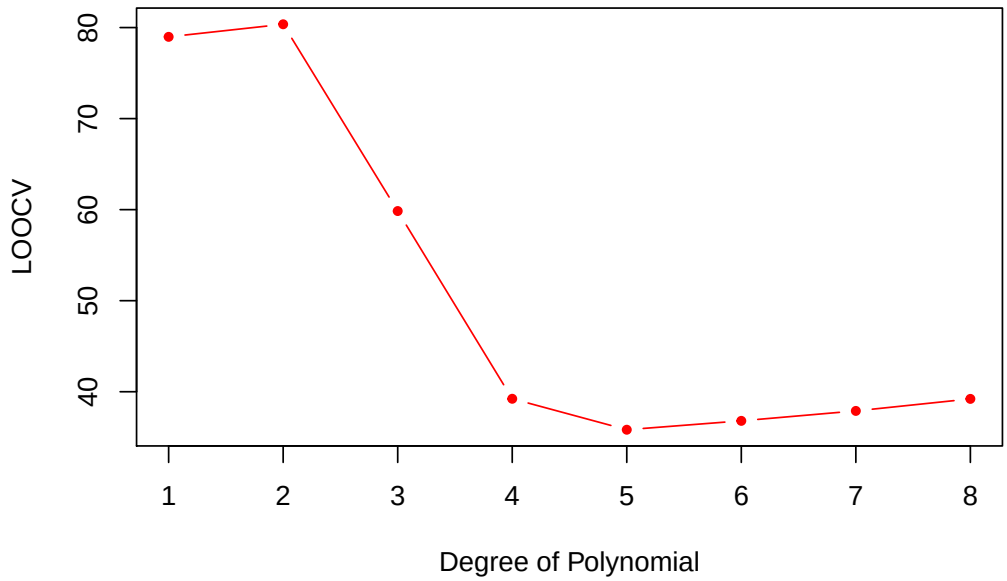


Figure 8: Test error estimates for fitted polynomials of degrees up to 8 using LOOCV

It seems that a polynomial regression with degree $p = 5$ is selected here as our best model using LOOCV with an estimate of test error given by 35.8257909.

In LOOCV we only observe the mean test error by averaging over n numbers corresponding to prediction error for n data points that are held out one by one. It is interesting to see the variability of the components of the LOOCV test error estimates:

```
set.seed(2020)
n  <-  100
p.max <- 8

x  <-  sort(runif(n, -2, 5))
y = -3+ 3*x*sin(2*pi*x)-1.3*x^2*cos(x) + rnorm(n, 0, 2)
toy.data <- data.frame(x=x, y=y)

LOOCV <- matrix(0, nrow=p.max,ncol=n)

for(p in 1:p.max){
   for (k in 1:n) {

      train.data = toy.data[-k,] # Training data
      test.data = toy.data[k,] # Test data
```

```
      # Train our models
      lm.p.minus.k = lm(y ~ poly(x,p),  data=train.data)

      # Record predictions
      pred.mat.k = predict(lm.p.minus.k, data.frame(x=test.data$x))

      #Calculate pred Error
      LOOCV[p, k] <- round(mean((test.data$y - pred.mat.k)^2),3)
  }
}

y.max <- max(LOOCV)

plot(1:p.max, LOOCV[,1], type="b", pch=20, col=1 , ylim=c(0, y.max),
     xlab="Degree of Polynomial",
     ylab="Test error estimates  for each hold-out data")
for(k in 2:n){
  points(1:p.max, LOOCV[, k], col=k, type="b", pch=20)
}
```

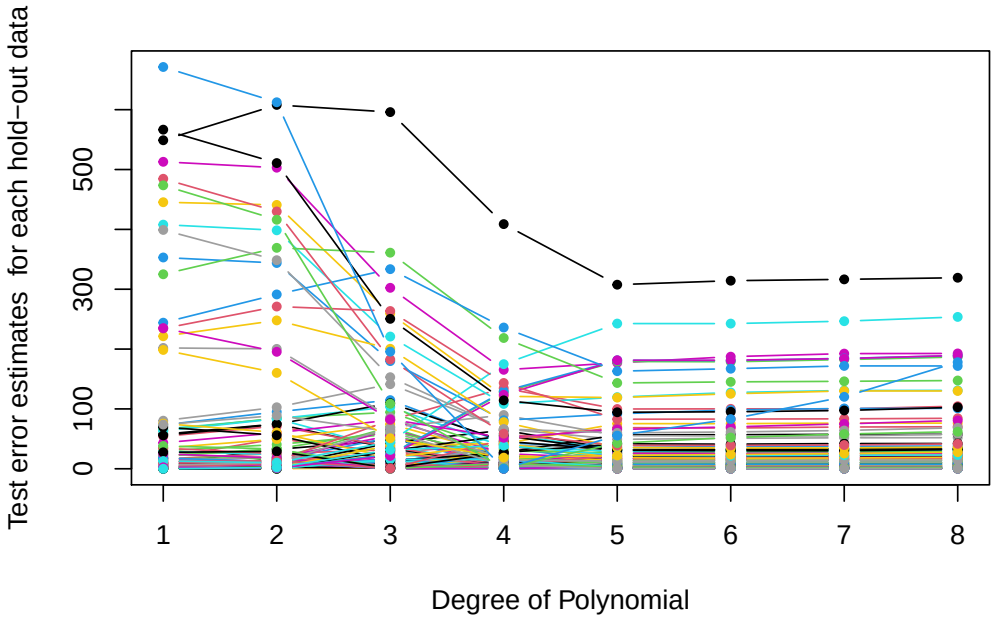


Figure 9: Variability of LOOCV test error estimate over training data when observationa left out one each at the time

Now, one might wonder, what happens to LOOCV error curve if we take different samples of size $n = 100$ from the underlying population. How does LOOCV behave and what is the distribution of the best selected model? To this end, we generate $B = 10$ different samples of size $n = 80$ and perform LOOCV to investigate its variability over repeated samples.

```
library("boot")
set.seed(2020)
n  <-  100
p.max <- 8
B <- 10
```

```
LOOCV <- matrix(0, nrow=p.max,ncol=B)

for (b in 1:B){
x  <-  sort(runif(n, -2, 5))
y = -3+ 3*x*sin(2*pi*x)-1.3*x^2*cos(x) + rnorm(n, 0, 2)
train <- data.frame(x=x, y=y)

for(p in 1:p.max){
   fitte.model <- glm(y~poly(x, p), data = train)
  LOOCV[p, b] <- cv.glm(train, fitte.model, K=n)$delta[1]
  }
}

y.max <- max(LOOCV)

plot(1:p, LOOCV[,1], type="b", pch=20, col=1 , ylim=c(0, y.max),
     xlab="Degree of Polynomial",
     ylab="LOOCV over repeated samples")
for(b in 2:B){
  points(1:p, LOOCV[, b], col=b, type="b", pch=20)
}
```

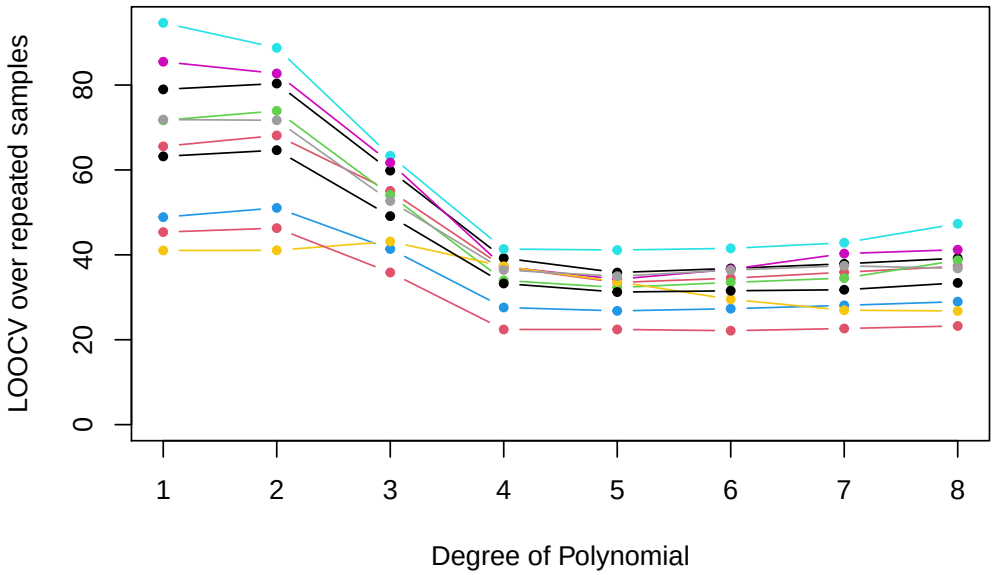


Figure 10: LOOCV over repeated samples from the same population for estimating the test error of polynomial regression models

The distribution of the best model over repeated samples is given by

```
table(apply(LOOCV, 2, which.min))/B
```

```
##
##   5   6   8
## 0.8 0.1 0.1
```

Also, we can get the variance of LOOCV test error for fitted polynomial regression models as a function of p over our repeated sampling:

```r
LOOCV.var <- apply(LOOCV, 1, var)
par(mfrow=c(1, 2))
plot(1:p.max, LOOCV.var, type="b",
     xlab="Degree of Polynomial",
     ylab="Variance of Test Error Estimate" )

boxplot(t(LOOCV),
        main="Test error distribution",
        xlab="Degree of Polynomial",
        col="orange",
        norder="brown"
        )
```

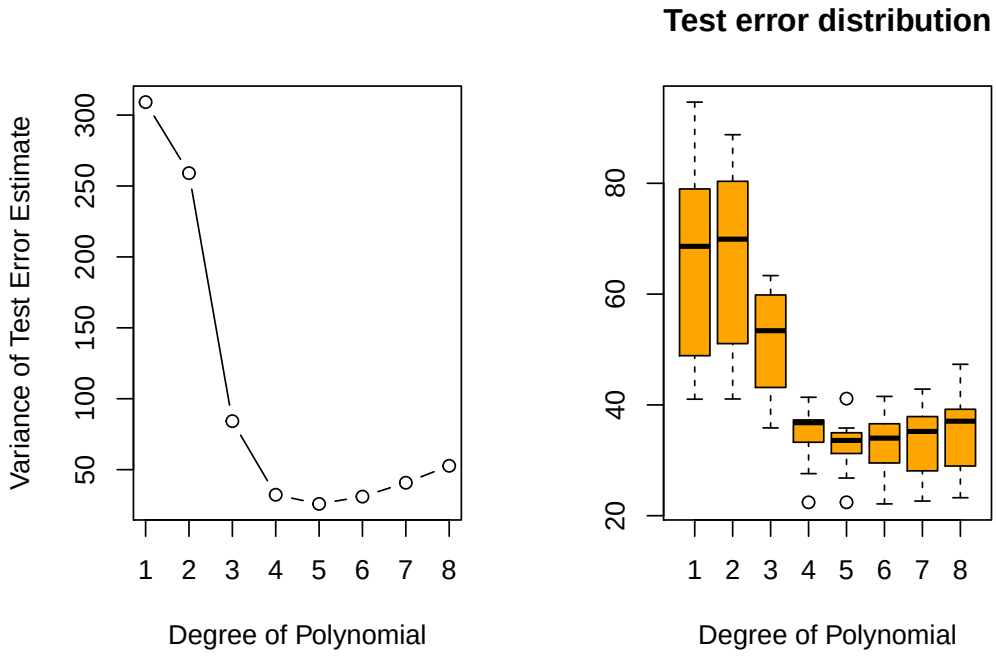


Figure 11: Variance of the LOOCV test error estimates as well as the box plot of error estimates obver each fold for polynomial regressions of degree up to 8

2.2.2 K-fold cross-validation

As we mentioned earlier, a drawback of LOOCV is that since the model has to be fit n times, this can be very time consuming when n is large, and/or each individual model is slow to fit. Also, the shortcut formula is not always applicable and it is only valid for linear smoothers. An alternative to LOOCV is K-fold cross-validation. This approach involves randomly dividing the set of observations into K groups (folds) of roughly equal size observations. Then, we proceed as follow:

- Use all but one fold to train your model/method
- Use the left out folds to make predictions
- Rotate around the roles of folds, K rounds total
- Compute squared error of all predictions, in the end

A common choice is $k = 5$ or $k = 10$ (note that $k = n$, reduces to LOOCV).

Let $\kappa : \{1, 2, \ldots, n\} \to \{1, \ldots, K\}$ be an indexing function that indicates the partition (fold) which observation i is allocated by the randomiation in the K-fold CV. Let $\hat{m}^{[-k]}(\mathbf{X})$ be the fitted function computed with the kth part of the data removed. Then, the K-fold CV estimate of test error is

$$CV_K(\hat{m}) = \frac{1}{n}\sum_{i=1}^{n} L(Y_I, \hat{m}^{[-\kappa(i)]}(\mathbf{X}_i)).$$

This indeed provides estimates of the average test error $Err_{\mathcal{T}}$ and not the conditional one. With $K = n$, the cross-validation estimator is approximately unbiased for the true (expected) test error, but can have high variance because the n training sets are similar to each other. On the other hand for K-fold CV, we will have only $K << n$ training data sets $\mathcal{T}_1^*, \ldots, \mathcal{T}_K^*$, where

$$\mathcal{T}_i^* = \mathcal{T} - \text{Fold}_i, \quad i = 1, 2, \ldots, K.$$

Note that, as each $\mathcal{T}_i^*$ could be significantly smaller than $\mathcal{T}^{(-i)}$, then $\mathcal{T}_i^*$'s are less correlated, hence K-fold CV scores are less variable. Also, this approach is computationally less intensive than LOOCV.

Let us consider our toy example again, where we fit polynomial regression models up to 8th degree to our data set generated from

$$Y = -3 + 3X\sin(2\pi X) - 1.3X^2\cos(X) + \epsilon, \quad \text{with} \quad \epsilon \sim N(0, 2),$$

and where $X \sim U(-2, 5)$. We generate a sample of size $n = 100$ from the population.

We use a 5-fold CV to estimate the test error associated with polynomial regression of degrees 3 and 6. To this end, we first create our folds:

```
# Split data in 5 parts, randomly
set.seed(2020)
K = 5
inds = sample(rep(1:K, length=n))
table(inds)
```

```
## inds
##  1  2  3  4  5
## 20 20 20 20 20
```

To perform 5-fold CV, we calculate predicted values when models are trained to data by removing each fold and treating it as a hold-out data:

```
pred.mat = matrix(0, n, 2) # Empty matrix to store predictions
for (k in 1:K) {
  #cat(paste("Fold",k,"... "))

  train.data = toy.data[inds!=k,] # Training data
  test.data = toy.data[inds==k,] # Test data

  # Train our models
  lm.3.minus.k = lm(y ~ poly(x,3),  data=train.data)
  lm.6.minus.k = lm(y ~ poly(x,6), data=train.data)

  # Record predictions
  pred.mat[inds==k,1] = predict(lm.3.minus.k, data.frame(x=test.data$x))
  pred.mat[inds==k,2] = predict(lm.6.minus.k, data.frame(x=test.data$x))
}
```

Now, we need to calculate the prediction errors over folds that are held out and take their average to obtain the corresponding test error estimates:

```
# Compute cross-validation error, one for each model
cv.errs = colMeans((pred.mat - toy.data$y)^2)

# Plot the results
par(mfrow=c(1,2))
xx = seq(min(toy.data$x), max(toy.data$x), length=100)

plot(toy.data$x, toy.data$y, xlab="y", ylab="y", pch=20, col=inds+1, main="Degree 3 Polynomial")
lines(xx, predict(lm.3, data.frame(x=xx)), # Note: model trained on FULL data!
      lwd=2, lty=2)
legend("topleft", legend=paste("Fold",1:k), pch=20, col=2:(k+1))
text(1, -15, label=paste("CV error:", round(cv.errs[1],3)))

plot(toy.data$x, toy.data$y, xlab="x", ylab="y", pch=20, col=inds+1, main="Degree 6 Polynomial")
lines(xx, predict(lm.6, data.frame(x=xx)), # Note: model trained on FULL data!
      lwd=2, lty=2)
legend("topleft", legend=paste("Fold",1:k), pch=20, col=2:(k+1))
text(1, -15, label=paste("CV error:", round(cv.errs[2],3)))
```

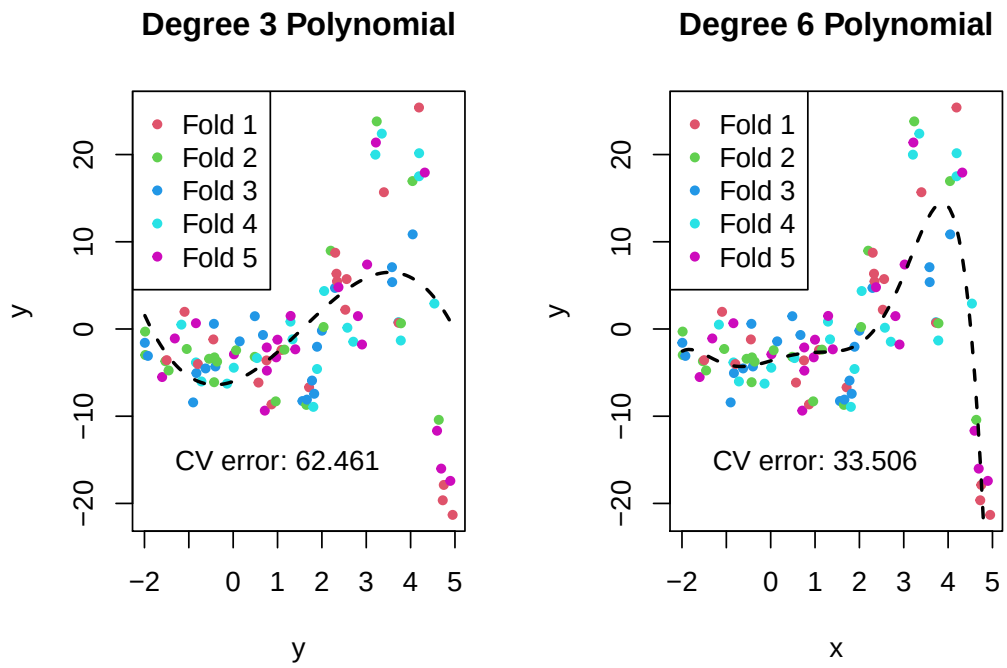
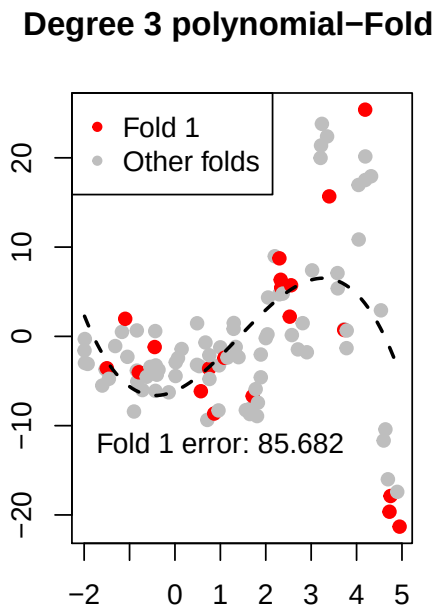
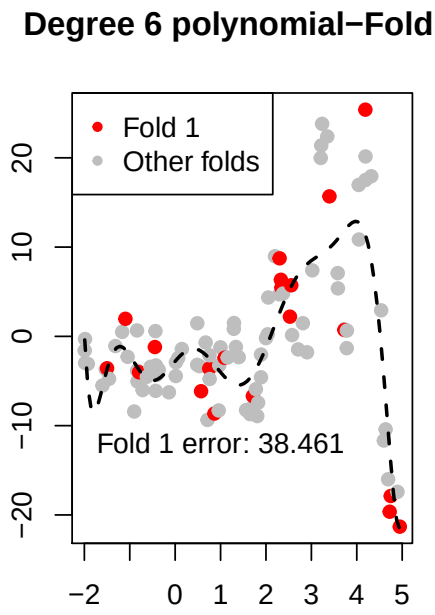


Figure 12: Observations in each fold of a 5-fold CV and corresponding test error estimates for polynomilas of degrees 3 and 6

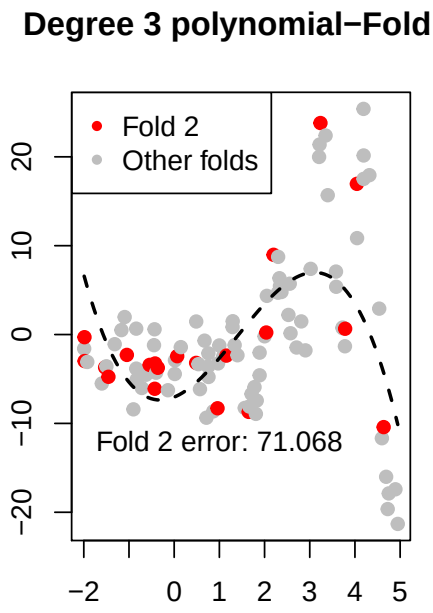
We can also visualize the different models that we fitted to each $\mathcal{T}_i^* = \mathcal{T} - \text{Fold}_i$, for $i = 1, 2, \ldots, 5$.

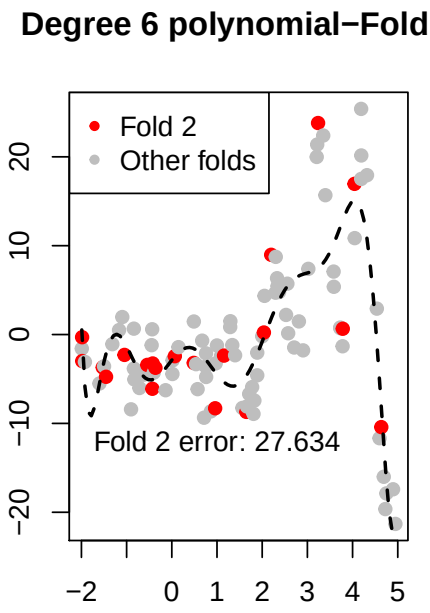
1. Fitted models to $\mathcal{T}_1^*$ by holding out the first fold as test data:



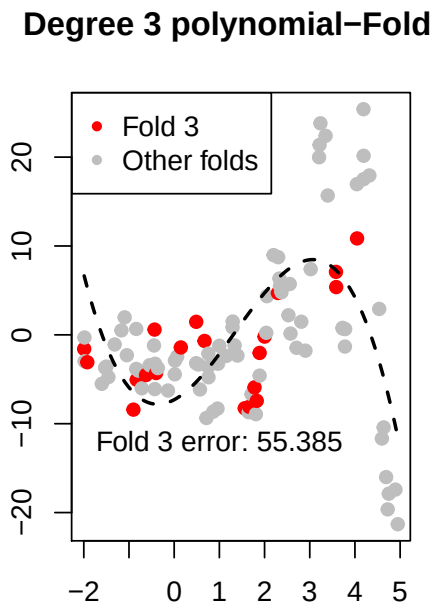


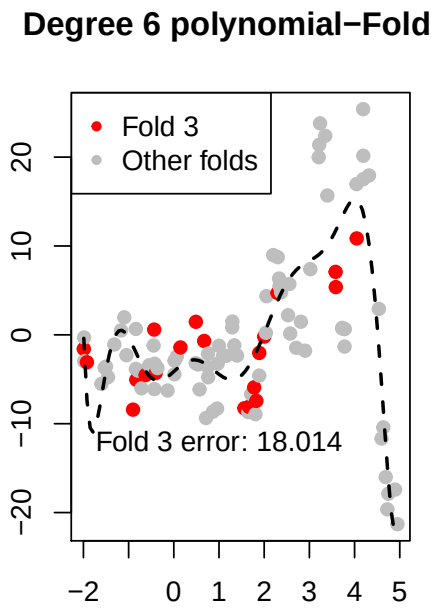
2. Fitted models to $\mathcal{T}_2^*$ by holding out the second fold as test data:



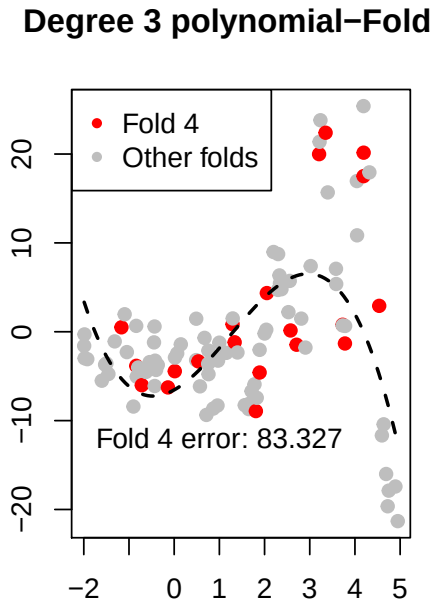


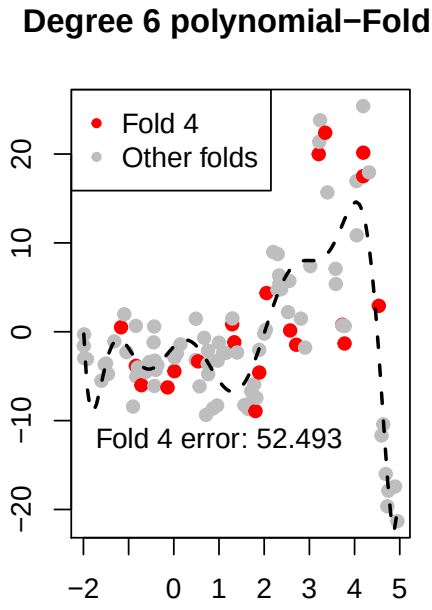
3. Fitted models to $\mathcal{T}_3^*$ by holding out the third fold as test data:



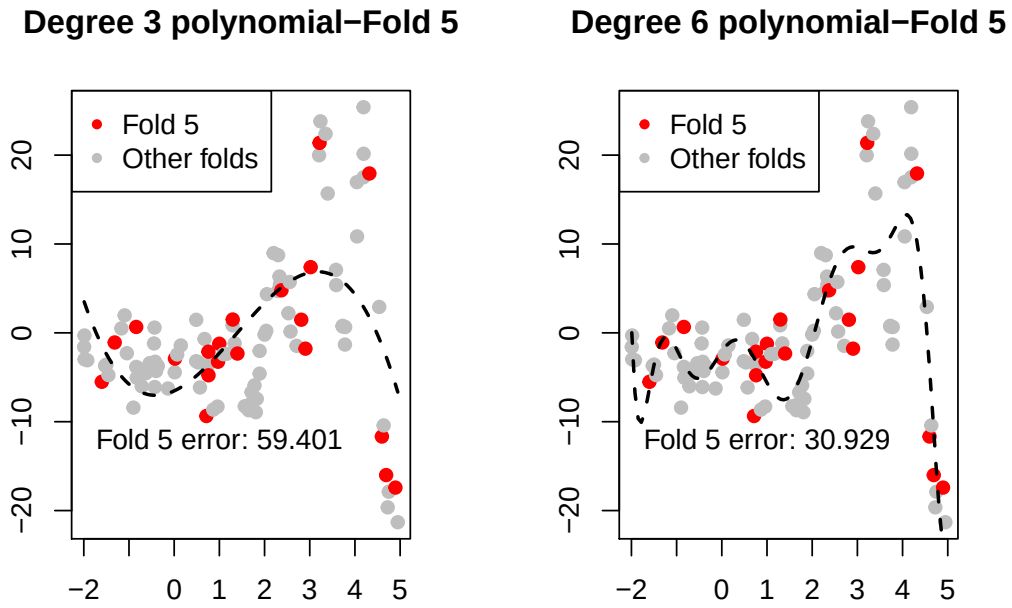


4. Fitted models to $\mathcal{T}_4^*$ by holding out the 4th fold as test data:





5. Fitted models to $\mathcal{T}_5^*$ by holding out the last fold as test data:



Now, to see how the test error estimates behave using K-fold CV, we look at our toy example again and fit polynomials of degrees up to 8 on a data set of size 100. Note that, here one can easily obtain the variance of the test-error as the byproduct of the K-fold CV as we have K estimates of the test error that we can use and these estimates are not that variables compared to what one can obtain in LOOCV which is n of them but very variable.

```
set.seed(2020)
library("boot")
n  <-  100
p.max <- 8
K <- 5

x  <-  sort(runif(n, -2, 5))
y = -3+ 3*x*sin(2*pi*x)-1.3*x^2*cos(x) + rnorm(n, 0, 2)
toy.data <- data.frame(x=x, y=y)

KfoldCV <- matrix(0, nrow=p.max,ncol=K)

for(p in 1:p.max){
   for (k in 1:K) {
     # cat(paste("Fold",k,"... "))

      train.data = toy.data[inds!=k,] # Training data
      test.data = toy.data[inds==k,] # Test data

      # Train our models
      lm.p.minus.k = lm(y ~ poly(x,p),  data=train.data)

      # Record predictions
      pred.mat[inds==k,1] = predict(lm.p.minus.k, data.frame(x=test.data$x))

      #Calculate pred Error
      KfoldCV[p, k] <- round(mean((test.data$y - pred.mat[inds==k,1])^2),3)
  }
}
```

```
KfoldCV.average <- apply(KfoldCV, 1, mean)

y.max <- max(KfoldCV)

par(mfrow=c(1, 2))
plot(1:p.max, KfoldCV.average, type="b", pch=20, col="darkgreen",
     ylab="Kfold CV test error estimates", xlab=" Degree of Polynomial", ylim=c(0, y.max))
plot(1:p.max, KfoldCV[,1], type="b", pch=20, col=1 , ylim=c(0, y.max),
     xlab="Degree of Polynomial",
     ylab="Test error estimates  over folds")
for(k in 2:K){
  points(1:p.max, KfoldCV[, k], col=b, type="b", pch=20)
}
```

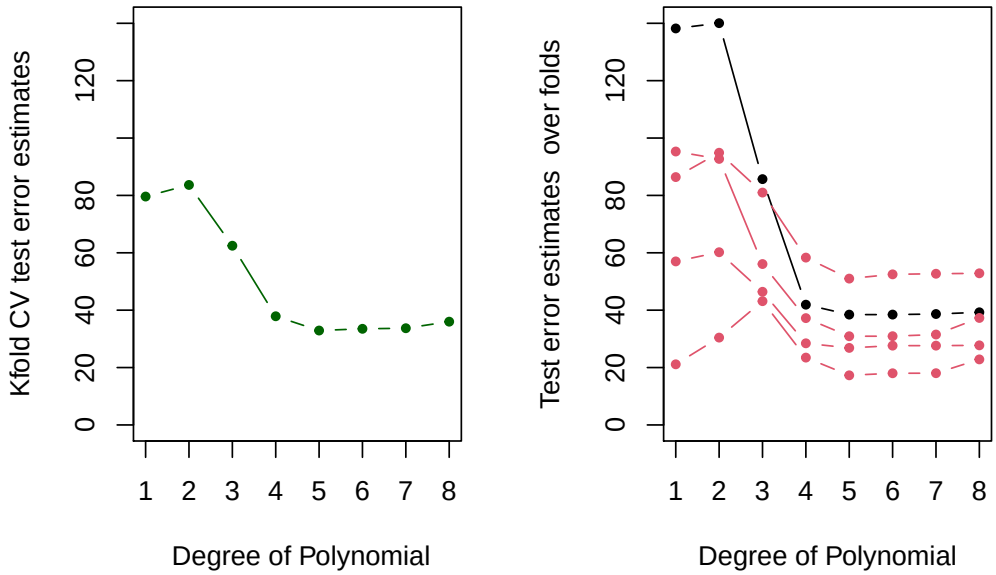


Figure 13: 5-fold CV test error estimates as well as test error estimates over different folds for polynomila regressions of degrees up to 8

We can also use the resampling approach (not practical in real situations) to get an estimate of the variance of the K-fold CV test error estimates. This can be done as follow:

```
set.seed(2020)
n  <-  100
p.max <- 8
B <- 10
K <- 5

KfoldCV <- matrix(0, nrow=p.max,ncol=B)

for (b in 1:B){
x  <-  sort(runif(n, -2, 5))
y = -3+ 3*x*sin(2*pi*x)-1.3*x^2*cos(x) + rnorm(n, 0, 2)
train <- data.frame(x=x, y=y)

for(p in 1:p.max){
```

```
  fitte.model <- glm(y~poly(x, p), data = train)
  KfoldCV[p, b] <- cv.glm(train, fitte.model, K=K)$delta[1]
 }
}

y.max <- max(KfoldCV)

plot(1:p.max, KfoldCV[,1], type="b", pch=20, col=1 , ylim=c(0, y.max),
     xlab="Degree of Polynomial",
     ylab="K-fold test error curves over repeated samples")
for(b in 2:B){
  points(1:p, KfoldCV[, b], col=b, type="b", pch=20)
}
```

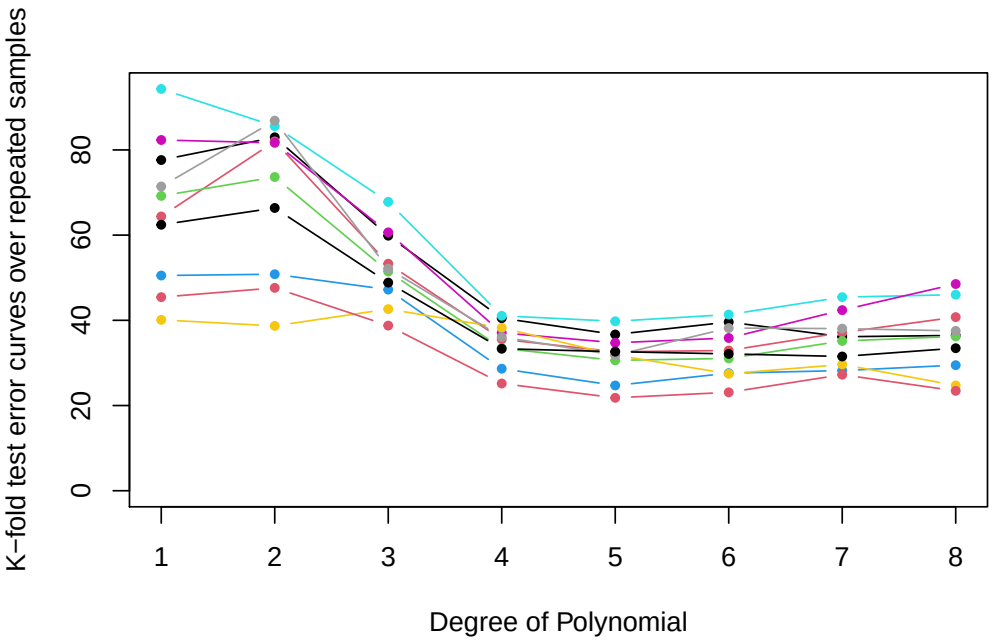


Figure 14: K-fold CV test error estimates over repeated samples from the population

As we see here, the K-fold CV error curves are less variable than LOOCV curves.

```
KfoldCV.var <- apply(KfoldCV, 1, var)
par(mfrow=c(1, 2))
y.max <- max(KfoldCV.var, LOOCV.var)
plot(1:p.max,KfoldCV.var, type="b", pch=20,  ylim=c(0, y.max),
     xlab="Degree of Polynomial",
     ylab="Variance of Test Error Estimate", col="green" )

points(1:p.max, LOOCV.var, type="b", pch=10, col="red" )
legend("topright", legend=c("K-fold CV", "LOOCV"), col=c("green", "red"), pch=c(20, 10))


boxplot(t(KfoldCV.var),
        main="Test error distribution",
        xlab="Degree of Polynomial",
        col="orange",
```

```
        norder="brown"
        )
```

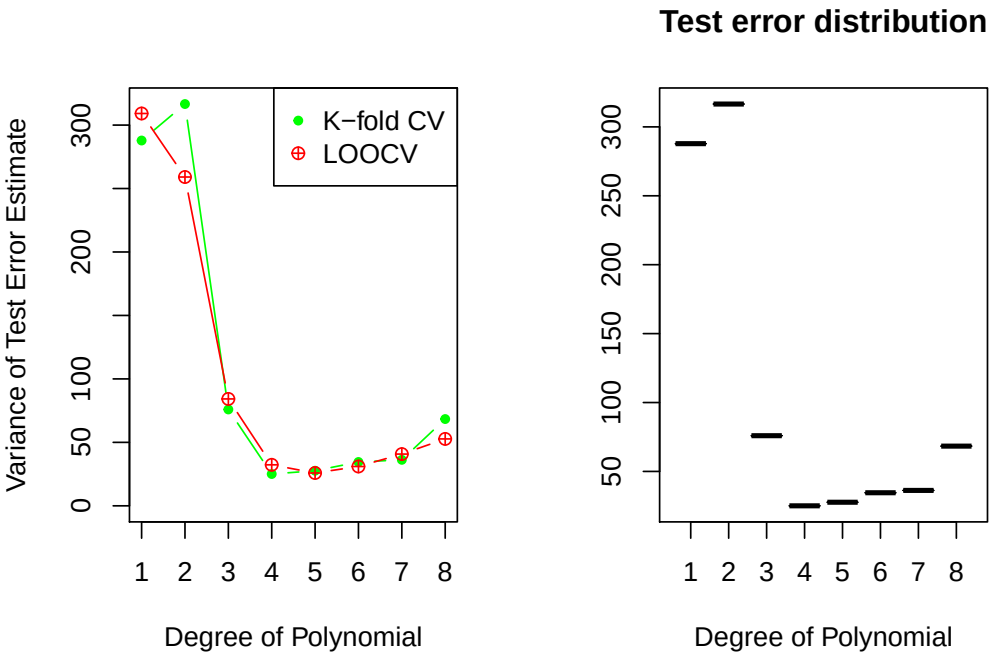


Figure 15: The variance of 5-fold CV test error estimates as well as the box plot og test error estimates over different folds for polynomila regressions of degrees up to 8

#The right and wrong ways of doing cross-validation Consider a classification problem with a large number of predictors, such as those in genomic applications. A typical strategy for analysis might be as follows:

1. Screen the predictors to find a subset of good predictors that show strong correlation with the class labels.
2. Using just this subset of predictors, build a multivariate classifier.
3. Use cross-validation to choose among classifiers and accordingly estimate the prediction error of the final model.

To investigate if such a method is correct or not, we will work with an example where under the null hypothesis there are $n = 50$ samples in two equal-sized classes, and $p = 5000$ standard normal predictors that are independent of class labels (response variable). The true test error rate of any classifier for this setting is 50%. We carry out the above recipe. We first generate the data as follows. We also present the histogram of the correlation coefficients of X_i's with Y in our data set and as we can see the mean of the correlations is zero as one would also expect to see:

```
set.seed(2020)
n = 50              # Sample size
p = 5000            # Dimension of feature space
best.size = 100     # Dimension after screening
B = 100             # Number of simulation
K = 5               # K in Knn approach
#Generate population
get.data <- function(n, p){
X = as.data.frame(matrix(rnorm(n*p),n, p))
Y = sample(rep(0:1, n/2), n, replace=FALSE)
X$Y = Y
```

```
return(X)
}
Data <- get.data(n, p)
cor.data <- cor(Data[, -ncol(Data)], Data$Y)
hist(cor.data , col="pink", xlab="Correlation of X's with Y", ylab="Frequency",  main="")
```

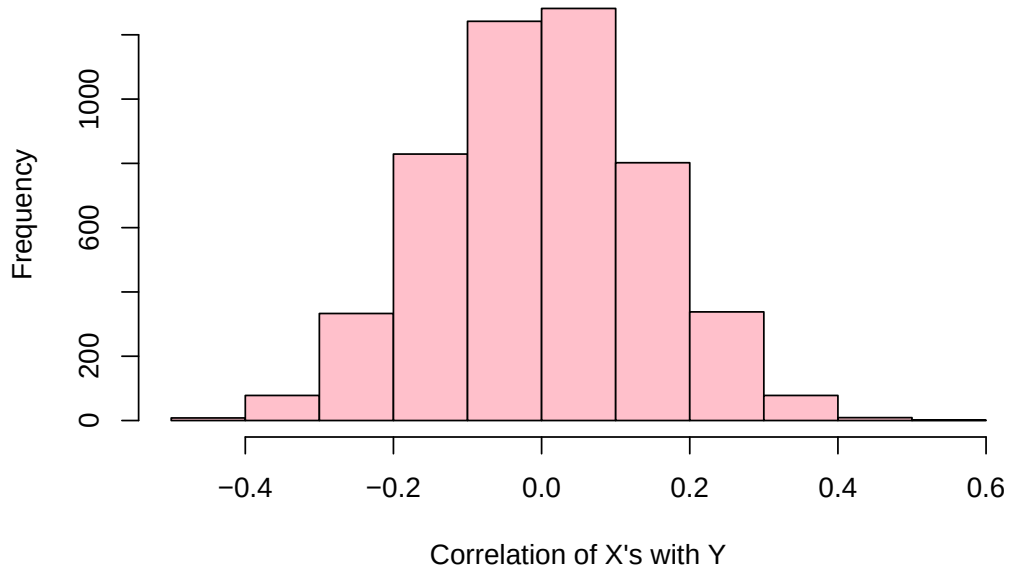


Figure 16: Histogram of correlation coefficients of features with class labels

```
mean(cor.data)
```

```
## [1] 1.421884e-05
```

We use the above mentioned algorithm and first perform a screening and select 100 predictors having the highest correlations with the class labels. This can be done as shown below, where we also provide the histogram of the correlation coefficients of selected features with class labels.

```
get.high.cor <- function(Data,p, best.size) {
  correlations = as.vector(cor(Data[,-ncol(Data)],Data[, ncol(Data)]))
  sorted = sort((correlations),index.return=T,decreasing=T)
  best.index = sorted$ix[1:best.size]
  return(list(high.cor= correlations[best.index], index.best=best.index))
 }

ind.best <- get.high.cor(Data,p, best.size)$index.best

hist(get.high.cor(Data,p, best.size)$high.cor,
     col="pink", main="Wrong way",
     xlab="", ylab="Frequency")
```

Now, we are ready to define functions to perform cross-validation the way that was described above.

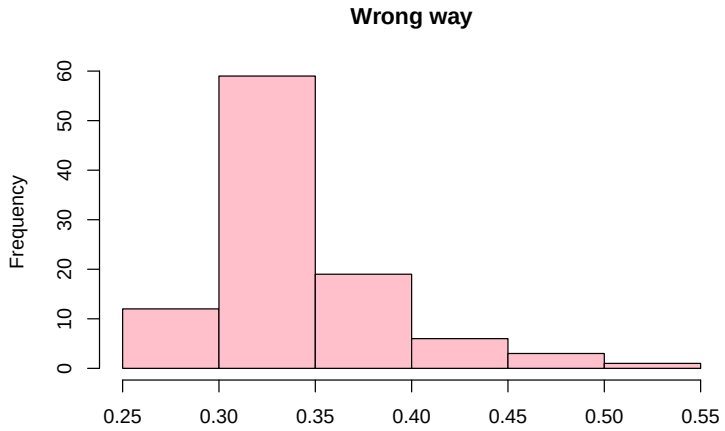


Figure 17: Histogram of the largest 100 correlation coefficients of features with class labels

2.3 1-NN classifier

For the data that we have generated, we perform a cross-validation and provide the estimate test error when a 1-nearest neighbor classifier, based on just 100 selected predictors are performed in each fold. We observe that 1-nearest neighbor classifier performs extremely well on the test data generate samples out after the variables have been selected does not correctly mimic the application of the classifier to a completely independent test set, since these predictors have already see the left out samples.

```
set.seed(2020)
WrongCV <- function(Data,p, best.size, K) {
 n <-nrow(Data)
 best = get.high.cor(Data,p, best.size)$index.best
 D.best <- Data[, c(best, ncol(Data))]

 inds = sample(rep(1:K, length=n))
 errors = c()
 corrr <- matrix(NA,  best.size, K)

  for (i in 1:K) {
    test = D.best[inds==i, ]
    train = D.best[inds != i, ]
    knn.train <- class::knn(train = train[, -ncol(train)],
                  test = train[, -ncol(train)],
                  cl   = train$Y,
                  k = 1, prob = FALSE)

    knn.test <- class:: knn(train =train[, -ncol(train)],  # train x
                 test = test[, -ncol(train)],  # test x
                 cl = train$Y,   #train y
                 k = 1, prob = FALSE)
    test.error.knn <- mean(test$Y != knn.test )
    # training error rate
    train.error.knn <- mean(train$Y != knn.train)
    # error rate
    errors = c(errors, test.error.knn)
   # corrr[ , i] <- cor(test[,-ncol(test)], test[, ncol(test)])
    corrr[, i] <- sapply(test[,-ncol(test)], function(x){cor(x, test[,ncol(test)])} )
}
```

```
 return(list(test.error = mean(errors), cor.with.y = as.vector(corrr)))
}

WrongCV(Data, p, best.size, 5)$test.error
```

```
## [1] 0
```

```
mean(WrongCV(Data, p, best.size, 5)$cor.with.y)
```

```
## [1] 0.3054754
```

To get more insight, after we select the best 100 predictors having highest correlations with the class labels, we select a random sample of size 10 from the population and calculate the correlation of the class labels with the selected feature for this sample.

```
index.10 <- sample(1:nrow(Data), 10, replace=FALSE)
Data.new <- Data[index.10, c(get.high.cor(Data, p, best.size)$index.best, ncol(Data))]

cor.in.10 <- cor(Data.new[, -ncol(Data.new)], Data.new[, ncol(Data.new)])
hist(cor.in.10, col="pink", xlab="Correlations of Selected Predictors with Outcome",
     main="")
```

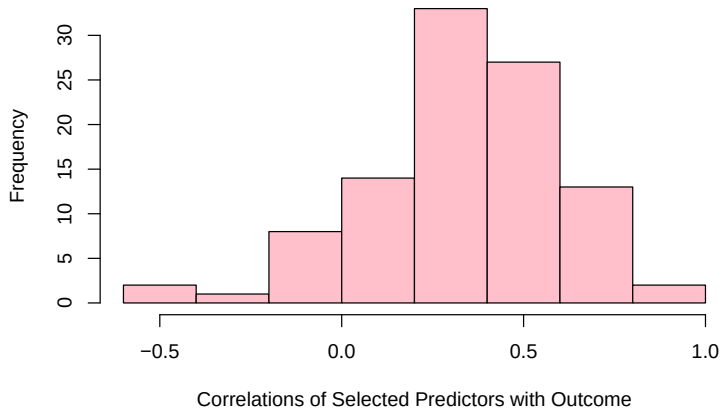


Figure 18: Histogram of the correlation coefficients of class labels and selected features for 10 observations randomly selected from the population

```
mean(cor.in.10)
```

```
## [1] 0.3435443
```

The average correlations is about 0.3435443, rather than 0, as one might expect.

Now, let us repeat this process 50 times. We provide the histogram of the correlation coefficients of class labels and selected features for test data in each fold over the simulation as well as the mean and boxplot of test errors.

```
B = 50
K = 5
wErrors = c()
wCorr = c()
for (i in 1:B) {
  X <- get.data(n, p)
  wCorr = c(wCorr, WrongCV(X,p, best.size,  K)$cor.with.y)
  wErrors = c(wErrors,WrongCV(X,p, best.size,  K)$test.error)

}

hist(wCorr, col="pink", main="", xlab="Correlation Coefficients")
```

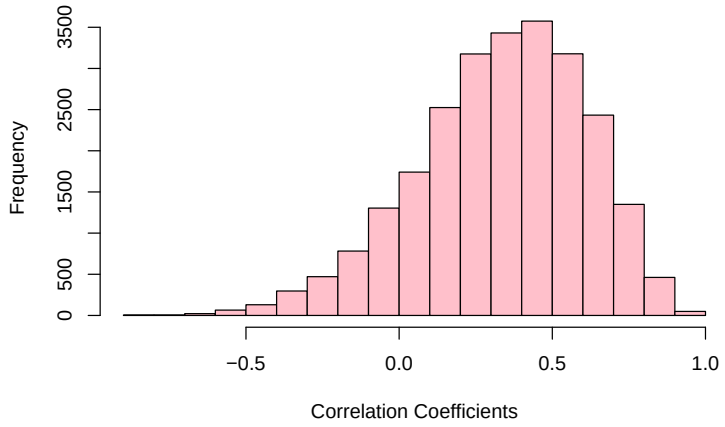


Figure 19: Histogram of the correlation coefficients of class labels and selected features for test data in each fold over the simulation

```
mean(wErrors)
```

```
## [1] 0.0336
```

```
boxplot(wErrors, col="red")
```

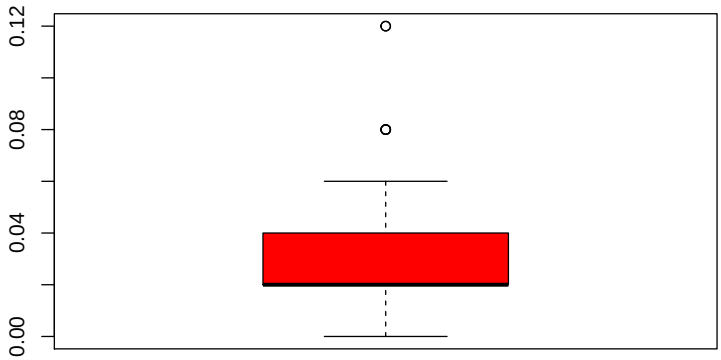


Figure 20: Histogram of the correlation coefficients of class labels and selected features for test data in each fold over the simulation

Here is the correct way to carry out cross-validation in this example:

1. Divide the samples into K cross-validation folds(group) at random.
2. For each fold $i = 1, 2, \ldots, K$
    - Find a subset of good predictors that show fairly strong correlation with the class labels, using all of the samples except those in fold i.
    - Using just this subset of predictors, build a multivariate classier, using all of the samples except those in fold i.
    - Use the classifier to predict the class labels for the samples in fold i.

The error estimates are then accumulated over all K folds to produce cross-validation estimate of prediction error.

```
RightCV <- function(Data,n,p, K) {
 n <-nrow(Data)
 inds = sample(rep(1:K, length=n))
 errors = c()
 corrr <- matrix(NA,  best.size, K)

  for (i in 1:K) {
    best = get.high.cor(Data[inds!=i, ],p, best.size)$index.best
    D.best <- Data[, c(best, ncol(Data))]
    test = D.best[inds==i, ]
    train = D.best[inds!= i, ]
    knn.train <- class::knn(train = train[, -ncol(train)],
                  test = train[, -ncol(train)],
                  cl   = train$Y,
                  k = 5, prob = FALSE)
    knn.test <- class:: knn(train =train[, -ncol(train)],  # train x
                 test = test[, -ncol(train)],  # test x
                 cl = train$Y,   #train y
                 k = 5, prob = FALSE)
    test.error.knn <- mean(test$Y != knn.test )
    # training error rate
    train.error.knn <- mean(train$Y != knn.train)

    errors = c(errors, test.error.knn)
    corrr[ , i] <- cor(D.best[inds==i,-ncol(D.best)], D.best[inds==i, ncol(D.best)])
  }

 return(list(test.error = mean(errors), cor.with.y = as.vector(corrr)))
}

RightCV(Data, p, best.size, 5)$test.error
```

```
## [1] 0.44
```

```
mean(RightCV(Data, p, best.size, 5)$cor.with.y)
```

```
## [1] -0.00179424
```

```
hist(RightCV(Data, p, best.size, 5)$cor.with.y, col="green",
     main="Right Way", xlab="Correlation of X's with Y in test")
```

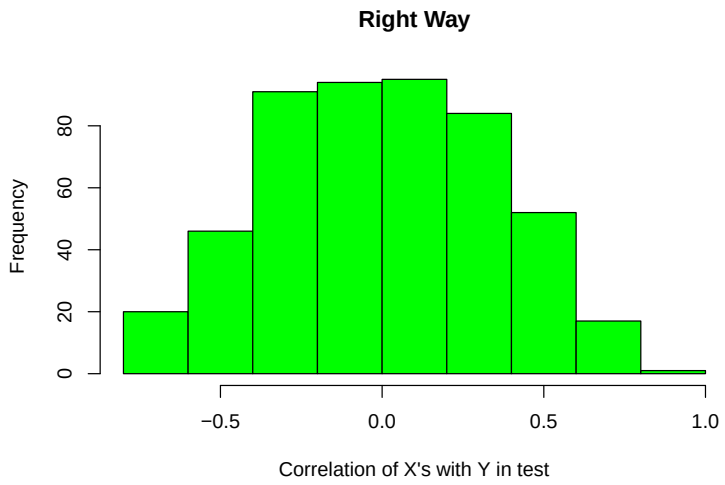


Figure 21: Histogram of the correlation coefficients of class labels and selected features for test data in each fold over the simulation using the right way of cross-validation

By repeating this process 50 times, we obtain the average test error as well as the histogram of the correlation coefficients between selected features in each fold with the class labels in left out samples.

```
cErrors = c()
cCorr = c()
for (i in 1:B) {
  X <- get.data(n, p)
  cCorr = c(cCorr, RightCV(X,p, best.size,  K)$cor.with.y)
  cErrors = c(cErrors,RightCV(X,p, best.size,  K)$test.error)

}

hist(cCorr, col="green", main="", xlab="Correlation Coefficients for Right Way of CV")
```

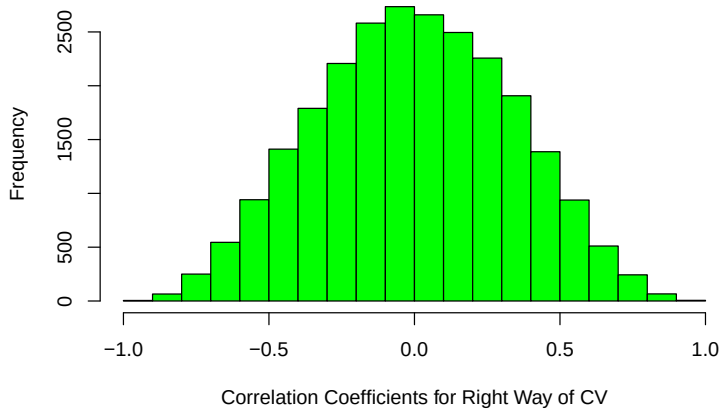


Figure 22: Histogram of the correlation coefficients of class labels and selected features for test data in each fold over 50 simulation

```
mean(cErrors)
```

```
## [1] 0.5432
```

By comparing the box-plot of the test errors of the right and wrong methods one can observe that the wrong cross-validation results in severely wrong and misleading results.

```
boxplot(cbind(cErrors, wErrors), col=c("green", "pink"), label=c("Correct Way", "Wromg Way" ) )
```

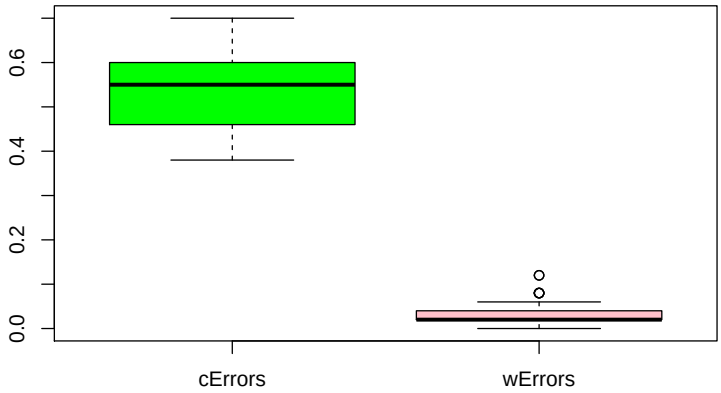


Figure 23: Boxplots oftest error estimates using the right and wrong cross-validation methods

2.4 Logistic Regression

Here we repeat the same process but this time we use logistic regression to perform classification. The details of the code are left to students to figure out by themselve by modifying previous codes accordingly.

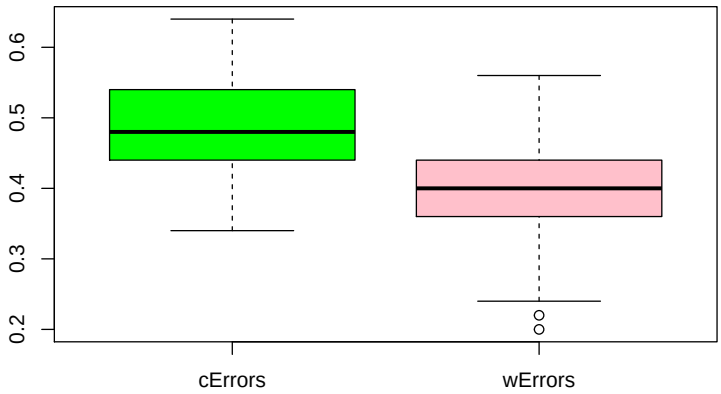


Figure 24: Test error estimates of logist regression models for classifications using the right and wrong way of cross-validations

The box-plots of the test errors associated with the right and wrong ways are given in Figure 24:

Remark 3. *In general, with a multistage modeling procedures, cross-validation must be applied to the entire sequence of modeling process steps. In particular, samples must be left out before any selection or filtering steps are applied. The only exception is when feature selection is done in an unsupervised way. For example, one might select 1100 predictors with highest variance across all 50 samples, before starting cross validation. Since the filtering doe snot involve the class labels, it does not give the predictors an unfair advantage.*

3 Cross-validation with unsupervised screening

Here we perform our previous study but this time instead of keeping those variables with the highest correlation with class labels, we retain 100 predictors that have the highest variance. We then fit 1-nearest neighbor classifiers using the right and wring way and run our simulation 50 times. The box-plot of errors with the two methods described above are then presented. As we observe, as variable selection is done independent of class labels in an unsupervised method, these two approaches give you similar test errors.

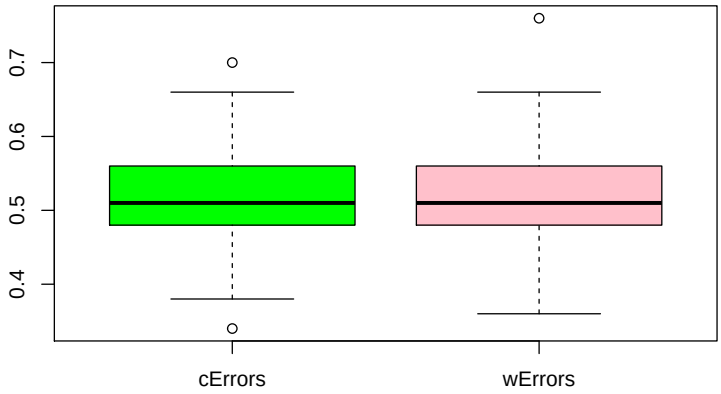


Figure 25: Boxplots of test error estimates using th eright and wrong way of classification methods when screening is done in an unsupervised way

4 Bootstrap

The bootstrap is a general tool for assessing statistical accuracy. In statistics, bootstrapping is any test or metric that relies on random sampling with replacement. Bootstrapping allows assigning measures of accuracy (defined in terms of bias, variance, confidence intervals, prediction error or some other such measure) to sample estimates. Although the method is nonparametric in nature, it can also be used for inference about parameters in parametric models. Bootstrapping is the practice of estimating properties of an estimator (such as its variance) by measuring those properties when sampling from an approximating distribution. One standard choice for an approximating distribution is the empirical distribution function of the observed data. In the case where a set of observations can be assumed to be from an independent and identically distributed population, this can be implemented by constructing a number of re-samples with replacement, of the observed dataset (and of equal size to the observed dataset).

##Introduction The bootstrap is a general data based computational methodology for determining the variance and the distribution of estimators of an unknown parameter θ. In general, parameters of a population are statistical functionals defined on a class of distributions $\mathcal{F}$, i.e., $\theta = T(F)$ for $F \in \mathcal{F}$, where $T : \mathcal{F} \to \mathbb{R}^d$. This is a very general definition of a parameter and allows one to cover bootstrap for both parametric and nonparametric studies. Here are some examples:

1. Suppose $\mathbf{X} = (X_1, \ldots, X_n)$ is an i.i.d. sample from a distribution with a known cdf $F(\cdot;\theta)$ where θ is the vector of unknown parameters of interest with $\theta \in \Theta \subseteq \mathbb{R}^d$, $d \geq 1$. This is the usual framework in parametric setting where the functional form of the underlying distribution is known but it is indexed by a set of unknown parameters. Knowing the values of θ will characterize the underlying population. Let $\widehat{\gamma}_n = \widehat{\gamma}_n(\mathbf{X})$ be an estimator of $\gamma(\theta)$, e.g., the maximum likelihood or the method of moments estimator. We would like to estimate the variance of $\widehat{\gamma}_n$ and construct a $100(1-\alpha)\%$ confidence interval for $\gamma(\theta)$. Note that in this case, we know the distribution of X_i's. However, there are many cases where finding the distribution and/or the variance of γ_n is very hard. In such cases, one needs to rely on resampling techniques, such as the bootstrap, to find an approximate sampling distribution for γ_n and make approximate inference about $\gamma(\theta)$.

2. Suppose $\mathbf{X} = (X_1, \ldots, X_n)$ is a random sample from an unknown distribution F and let $\theta = T(F)$ denote the mean of F. Here $\theta = E(X_i) = \int x dF(x)$. Let $\widehat{\theta}_n = T(F_n) = \frac{1}{n}\sum_{i=1}^n X_i$ be the plug-in estimator of θ where F_n is the edf of $\mathbf{X}$. Again, we would like to estimate the variance of $\widehat{\theta}_n$ and construct a $100(1-\alpha)\%$ confidence interval for θ. This is a nonparametric example where we do not know the distribution of X_i's. As another example, we might be interested in making inference about the population median $\theta = F^{-1}(\frac{1}{2})$, where $P(X_i \leq \theta) = \frac{1}{2}$.

In the first example θ denotes the parameter of a parametric model, while in the second example, we are in a nonparametric situation. In nonparametric cases, we think of a parameter as a function of the underlying distribution F and we simply write $\theta = T(F)$. One way to estimate $T(F)$ is to use the plug-in method by replacing F with the edf F_n and obtaining $\widehat{\theta}_n = T(F_n)$. The bootstrap can help us to make inference about the distribution of $\widehat{\theta}_n - \theta$. In order to do this, suppose an i.i.d. random sample $X_1, \ldots, X_n$ is available from F. A sample of size n from F_n is called a bootstrap sample, denoted by

$$X_1^*, \ldots, X_n^* \sim F_n.$$

Bootstrap samples play an important role in what follows. We should note that drawing i.i.d. samples $X_1^*, \ldots, X_n^*$ from F_n is equivalent to drawing n observations, with replacement, from the original sample $\mathbf{X} = (X_1, \ldots, X_n)$. This is why the bootstrap is also called resampling the data in the literature.

After we obtain a bootstrap sample $X_1^*, \ldots, X_n^*$ from F_n, we construct an estimate $\widehat{\theta}_n^* = T(F_n^*)$ for $\widehat{\theta}_n$ where F_n^* is the edf of $\mathbf{X}^* = (X_1^*, \ldots, X_n^*)$.

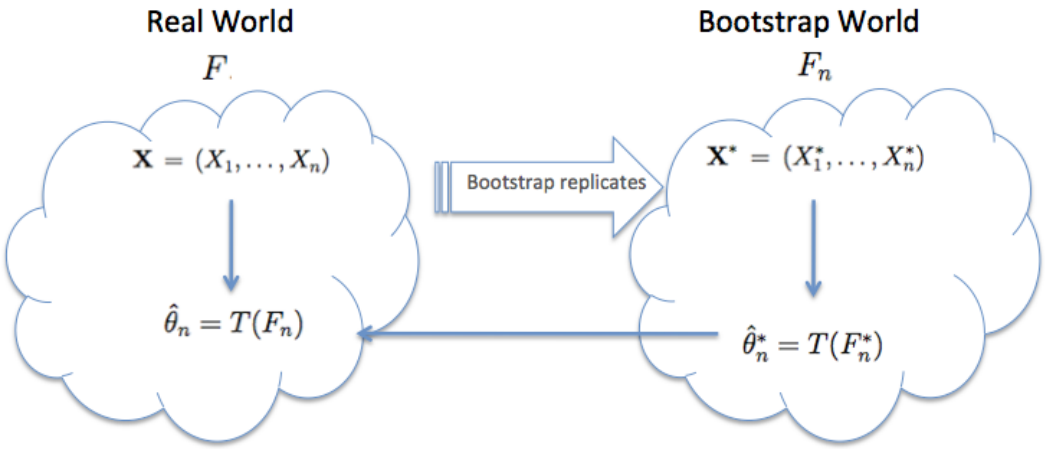


Figure 26: Bootstrap for approximating the sampling distribution of an estimator

The bootstrap idea is simply the following distributional approximation

$$(\widehat{\theta}_n - \theta) \approx (\widehat{\theta}_n^* - \widehat{\theta}_n),$$

which means the the distributions of two random variables $(\widehat{\theta}_n - \theta)$ and $(\widehat{\theta}_n^* - \widehat{\theta}_n)$ are approximately the same. This implies that, for example,

$$\text{Bias}_F(\widehat{\theta}_n) = \mathbb{E}_F(\widehat{\theta}_n - \theta) \approx \mathbb{E}_{F_n}(\widehat{\theta}_n^* - \widehat{\theta}_n),$$

and

$$\mathbb{V}ar_F(\widehat{\theta}_n) = \mathbb{V}ar_F(\widehat{\theta}_n - \theta) \approx \mathbb{V}ar_{F_n}(\widehat{\theta}_n^* - \widehat{\theta}_n).$$

Since F_n is known, we can often find theoretical properties of the distribution of $\widehat{\theta}_n^* - \widehat{\theta}_n$ reasonably easy, or using Monte Carlo methods. So, bootstrapping is an approach to statistical inference based on building a sampling distribution for a statistic by resampling from the data at hand. The term *bootstrapping* due to Efron (1979), is an allusion to the expression "pulling oneself up by one's bootstraps", in this case using the sample data as a population from which repeated samples are drawn.

4.1 Bootstrap for Bias correction

We can use bootstrapping to correct for a bias estimator. Since the sampling distribution of $\widehat{\theta}_n^*$ is close to that of $\widehat{\theta}_n$, and $\widehat{\theta}_n$ itself is close to θ, as we showed earlier

$$\mathbb{E}_F(\widehat{\theta}_n) - \theta \approx \mathbb{E}_{F_n}(\widehat{\theta}_n^*) - \widehat{\theta}_n. \tag{3}$$

The left hand side is the bias that we are looking for and the right-hand side the one that we can calculate using bootstrap. After estimating the bias of $\widehat{\theta}_n$, one can correct for the bias of the estimator. To this end, since

$$\mathbb{E}_F(\widehat{\theta}_n) = \theta + \text{bias}_F(\widehat{\theta}_n),$$

and

$$\widehat{\text{bias}_F(\widehat{\theta}_n)} \approx \mathbb{E}_{F_n}(\widehat{\theta}_n^*) - \widehat{\theta}_n,$$

we have

$$\mathbb{E}_F\left(\widehat{\theta}_n - \widehat{\text{bias}_F(\widehat{\theta}_n)}\right) \approx \theta.$$

This results in a bootstrap-bias corrected estimator

$$2\widehat{\theta}_n - \mathbb{E}_{F_n}(\widehat{\theta}_n^*).$$

If simulations are used to estimate $\mathbb{E}_{F_n}(\widehat{\theta}_n^*)$, then in the formula for the bootstrap estimate of the bias and the bias corrected estimator, $\mathbb{E}_{F_n}(\widehat{\theta}_n^*)$ would be replaced by $\frac{1}{B}\sum_{b=1}^{B}\widehat{\theta}_{n,b}^*$ (see next Section). Note that (3) remains valid as long as the sampling distribution of $\widehat{\theta}_n - \theta$ is close to that of $\widehat{\theta}_n^* - \widehat{\theta}_n$. This is of course a weaker requirement than asking for $\widehat{\theta}_n$ and $\widehat{\theta}_n^*$ themselves have similar distributions, or asking for $\widehat{\theta}_n$ be close to θ. A sufficient but not necessary condition for (3) to hold is that $\widehat{\theta}_n - \theta$ be a pivot, or approximately pivotal.

##Bootstrap algorithm for estimating the variance

Here we give a general algorithm for estimating the variance of the estimator $\widehat{\theta}_n$ using the bootstrap method. To this end, suppose $\widehat{\theta}_n = T(F_n)$ denotes an estimator of θ. To find the bootstrap variance estimator of $\widehat{\theta}_n$ we proceed as follow:

1. Using $\mathbf{X} = (X_1, \ldots, X_n)$ we obtain the edf $F_n(t) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(X_i \le t)$, $t \in \mathbb{R}$.
2. For $b = 1, 2, \ldots, B$, with B being large,
   - 2.1 Draw a simple random sample with replacement $X_b^* = (X_{1,b}^*, \ldots, X_{n,b}^*)$ from F_n (or equivalently from $\mathbf{X}$).
   - 2.2 Calculate the bootstrap edf $F_{n,b}^*(t) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(X_{i,b}^* \le t)$.
   - 2.2 Calculate the bootstrap replicates of $\widehat{\theta}_n$ given by $\widehat{\theta}_{n,b}^* = T(F_{n,b}^*)$.
3. After calculating $\widehat{\theta}_{n,1}^*, \ldots, \widehat{\theta}_{n,B}^*$ compute

$$\widehat{\mathbb{V}ar}_B(\widehat{\theta}_n) = \frac{1}{B-1}\sum_{i=1}^{B}(\widehat{\theta}_{n,i}^* - \overline{\theta^*})^2 \approx \mathbb{V}ar_{F_n}(\widehat{\theta}_n^*),$$

   where

$$\overline{\theta^*} = \frac{1}{B}\sum_{b=1}^{B}\widehat{\theta}_{n,b}^* \approx \mathbb{E}_{F_n}(\widehat{\theta}_n^*).$$

Note that the random selection of bootstrap samples is not an essential aspect of the (nonparametric) bootstrap. At least in principal, we could enumerate all bootstraps of samples of size n corresponding to all simple random samples with replacement from $\mathbf{X} = (X_1, \ldots, X_n)$ and calculate the exact values of $\mathbb{E}_{F_n}(\widehat{\theta}_n)$ and $\mathbb{V}ar_{F_n}(\widehat{\theta}_n)$ rather than estimating them. One can easily observe that there are two sources of errors when we perform bootstrapping:

1. The error induced by using F_n instead of F or essentially $X_1, \ldots, X_n$ instead of the whole population. This error happens because n is finite.

2. The simulation error produced by failing to enumerate all bootstrap samples.

After we observed our data we do not have any further control on the first error, while the second source of error can be controlled by making B sufficiently large. In practice, it usually suffices to take $B = 10,000$. So, we can ignore the second source of error.

$$\mathbb{V}ar_F(\widehat{\theta}_n) \overset{O(1/\sqrt{n})}{\approx} \mathbb{V}ar_{F_n}(\widehat{\theta}_n) \overset{O(1/\sqrt{B})}{\approx} \widehat{\mathbb{V}ar}_B(\widehat{\theta}_n).$$

In the following sections we further explain parametric and nonparametric bootstrap methods and show that the way that we approach these problems are slightly different.

4.2 Parametric Bootstrap

Suppose F belongs to a parametric family indexed by a parameter λ. That is, $F(\cdot) = F_\lambda(\cdot)$ with pdf $f_\lambda(\cdot)$. Let $\theta = T(F_\lambda)$ be the parameter of interest. Since we have the likelihood function, we can estimate θ using its MLE. Suppose $\widehat{\lambda}$ is the MLE of λ, then by the invariance property of the MLE, F_λ will be estimated by $F_{\widehat{\lambda}}$ and accordingly θ by $\widehat{\theta} = T(F_{\widehat{\lambda}})$. However, there are many cases where finding the sampling distribution of $\widehat{\theta}$ might be difficult or the sample size is small such that one can not rely on the asymptotic theory to obtain the variance of $\widehat{\theta}$. In such cases we can simply use a parametric bootstrap. Suppose $X_1^*, \ldots, X_n^*$ are i.i.d. samples from $F_{\widehat{\lambda}}$ and let $\widehat{\lambda}^*$ be its corresponding MLE. The parametric bootstrap approximates the distribution of

$$\widehat{\theta} - \theta = T(F_{\widehat{\lambda}}) - T(F_\lambda),$$

with the distribution of

$$\widehat{\theta}^* - \widehat{\theta} = T(F_{\widehat{\lambda}^*}) - T(F_{\widehat{\lambda}})$$

However, as we show later, if one needs to make sure, as the testing hypothesis is done under the null hypothesis, the bootstrap should reflect the null hypothesis correctly.

Example 1. *Suppose $X_1, \ldots, X_n$ be i.i.d. samples from a $N(\mu, \sigma^2)$ distribution where $\mu \in \mathbb{R}$, $\sigma > 0$ and the sample size is small. Assume that $\lambda = (\mu, \sigma^2)^\top$ is unknown and we are interested in $\theta = \mu$. The MLE of λ is $\widehat{\lambda} = (\bar{X}, \frac{1}{n}\sum_{i=1}^n (X_i - \bar{X})^2)$, and so $\widehat{\theta} = \bar{X}$. We know that*

$$\widehat{\theta} - \theta \sim N(0, \frac{\sigma^2}{n}).$$

Since σ^2 is unknown we can not use this distribution to make inference about θ. In addition, the sample size is small to get a good estimator of σ^2 and use the t-distribution to make inference about θ. However, we can use the bootstrap method. To this end, let

$$\widehat{F} = F_{\widehat{\lambda}} = N(\bar{X}, \widehat{\sigma}^2),$$

as a new distribution with parameters $\bar{X}$ and $\widehat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^n (X_i - \bar{X})^2$. We now draw bootstrap samples $X_1^*, \ldots, X_n^*$ from $F_{\widehat{\lambda}}$. Let

$$\bar{X}^* = \frac{1}{n}\sum_{i=1}^n X_i^* \quad \text{and} \quad \widehat{\sigma}^{2*} = \frac{1}{n}\sum_{i=1}^n (X_i - \bar{X})^2.$$

So $\widehat{\theta}^* = \bar{X}^*$ and

$$\widehat{\theta}^* - \widehat{\theta} \sim N(0, \frac{\widehat{\sigma}^2}{n}).$$

Thus, the bootstrap variance estimator of $\widehat{\theta}$ is given by

$$\begin{aligned}\mathbb{V}ar_{boot}(\widehat{\theta}) &= \mathbb{V}ar_F(\widehat{\theta} - \theta) \\ &\approx \mathbb{V}ar_{\widehat{F}}(\widehat{\theta}^* - \widehat{\theta}) \\ &= \frac{\widehat{\sigma}^2}{n} = \frac{1}{n^2}\sum_{i=1}^{n}(X_i - \bar{X})^2.\end{aligned}$$

In Example 1 we were able to obtain analytical solutions. In general, when analytical solutions are not possible or very hard to obtain, we simulate B bootstrap samples

$$X^*_{1,b}, X^*_{2,b}, \ldots, X^*_{n,b} \sim F_{\widehat{\lambda}}, \quad b = 1, \ldots, B,$$

and calculate

$$\widehat{\theta}^*_1, \widehat{\theta}^*_2, \ldots, \widehat{\theta}^*_B,$$

where each of which are estimates of $\widehat{\theta}$. So, we can use the empirical bias and variance of these replicates to estimate the bias and the variance of $\widehat{\theta}$. In other words

$$\text{Bias}_{\text{B}}(\widehat{\theta}) \approx \frac{1}{B}\sum_{b=1}^{B}(\widehat{\theta}^*_b - \widehat{\theta}),$$

and

$$\mathbb{V}ar_{\text{B}}(\widehat{\theta}) \approx \frac{1}{B-1}\sum_{b=1}^{B}(\widehat{\theta}^*_b - \overline{\theta^*})^2,$$

where $\overline{\theta^*} = \frac{1}{B}\sum_{b=1}^{B}\widehat{\theta}^*_b$.

4.3 Nonparametric Bootstrap

In nonparametric bootstrap, which is very popular in practice, we do not impose any parametric assumption on F. Suppose F belongs to a nonparametric family $\mathcal{F}$ of distributions and $X_1, \ldots, X_n$ is a random sample of size n from F. The usual estimator of F is the edf

$$F_n(t) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(X_i \leq t) = \frac{\#\{X_i \leq t\}}{n}, t \in \mathbb{R}.$$

We note that sampling from F_n is equivalent to resampling from $X_1, \ldots, X_n$ with replacement. As in parametric bootstrap, we can sometimes perform the bootstrap analytically but most often we need to use the Monte Carlo approach. As an example, consider estimating the population mean $\theta = T(F) = \int x dF(x)$. The plug-in estimator of θ is

$$\widehat{\theta} = T(F_n) = \frac{1}{n}\sum_{i=1}^{n}X_i = \bar{X}.$$

Consider a bootstrap (i.i.d.) sample

$$X^*_1, \ldots, X^*_n \sim F_n,$$

and let $\bar{X}^* = \frac{1}{n}\sum_{i=1}^{n}X^*_i$ be the bootstrap sample mean. Hence,

$$\begin{aligned}\mathbb{E}_{F_n}(\bar{X}^*) &= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}_{F_n}(X^*_i) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left\{\frac{X_1}{n} + \ldots + \frac{X_n}{n}\right\} \\ &= \bar{X}.\end{aligned}$$

Also

$$\begin{aligned}\mathbb{V}ar_{F_n}(\bar{X}^*) &= \frac{1}{n^2}\sum_{i=1}^{n}\mathbb{V}ar_{F_n}(X_i^*)\\ &= \frac{1}{n}\mathbb{E}_{F_n}(X_i^* - \bar{X})^2\\ &= \frac{1}{n}\left\{\frac{(X_1-\bar{X})^2}{n} + \ldots + \frac{(X_n-\bar{X})^2}{n}\right\}\\ &= \frac{1}{n^2}\sum_{i=1}^{n}(X_i-\bar{X})^2.\end{aligned}$$

This is a very simple but general function which can be used to bootstrap a `Statistic` by repeated sampling from a `Population` when the sampling is repeated `B` times with sample size `n`.

```
boot.approx <- function(Population, Statistic, B, n) {
out <- numeric(B)
for(b in 1:B) out[b] <- Statistic(sample(Population, n, replace=TRUE))
return(out)
}
```

4.4 Example 1: Gamma distribution

Suppose we have a population which is generated from a $Gamma(\alpha = 2, \beta = 1/3)$ distribution with pdf

$$f(x) = \frac{1}{\Gamma(\alpha)\beta^\alpha}x^{\alpha-1}e^{-\frac{x}{\beta}}, \quad x, \alpha, \beta > 0.$$

We are going to take bootstrap samples to see the variability of the sample variance as an estimator of the population variance $Var(X) = \frac{\alpha}{\beta^2} = 18$.

```
set.seed(1)
U <- rgamma(100, 2, 1/3)
par(mfrow=c(1,3), mgp=c(1.8, 1, 0), mar=c(3,3,2,1), family="Times", cex.lab=1,cex=1)
out1 <- boot.approx(U, var, B=5000, n=5)
out2 <- boot.approx(U, median, B=5000, n=5)
out3 <- boot.approx(U, mean, B=5000, n=5)
hist(out1, freq=FALSE, xlab=expression(S[boot]^2), col="orange", border="white",main="")
abline(v=var(U), lwd=2, lty=2)
hist(out2, freq=FALSE, xlab=expression(Median[boot]), col="orange", border="white",main="")
abline(v=median(U), lwd=2, lty=2)
hist(out3, freq=FALSE, xlab=expression(Mean[boot]), col="orange", border="white",main="")
abline(v=mean(U), lwd=2, lty=2)
```

```
print(quantile(out1, c(0.05, 0.95)))
```

```
##        5%       95%
##  1.901586 25.814964
```

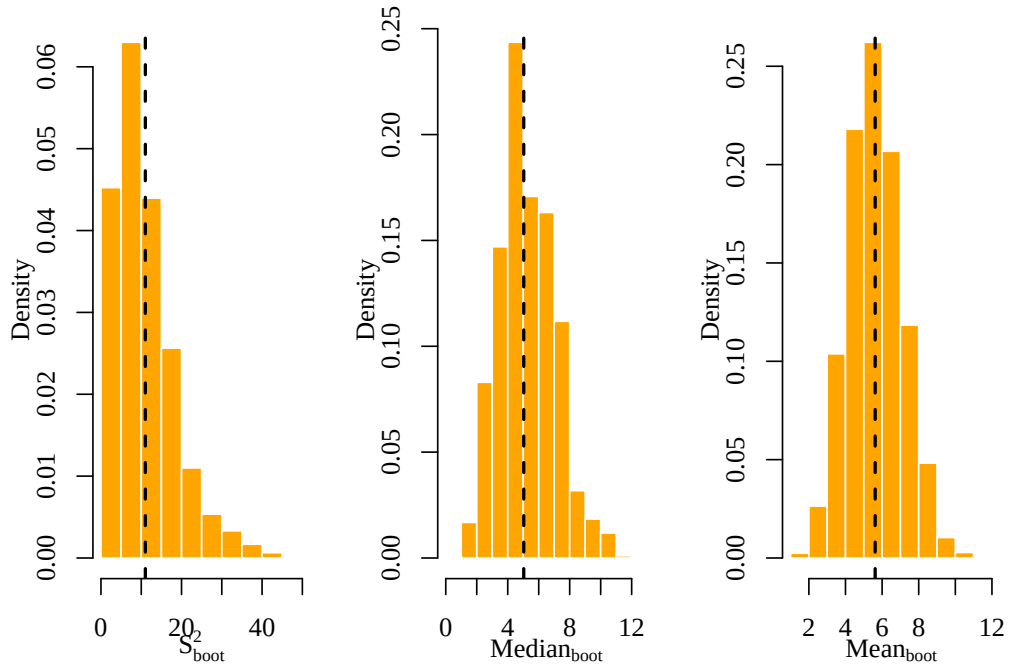


Figure 27: Bootstrap estimates of the sampling distributions of the variance, median and mean in a Gamma distribution

```
print(quantile(out2, c(0.05, 0.95)))
```

```
##       5%      95%
## 2.467276 8.465507
```

```
print(quantile(out3, c(0.05, 0.95)))
```

```
##       5%      95%
## 3.338046 8.180469
```

4.5 Example 2 : Asset Management

This is based on Section 5.2 of the book **An Introduction to Statistical Learning: with applications in R**. Suppose 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. We wish to choose α that minimizes the total risk. In other words, we want to minimize

$$Var(\alpha X + (1-\alpha)Y).$$

It can easily be shown that the minimum value of α is given by

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

where $\sigma_X^2 = Var(X)$, $\sigma_Y^2 = Var(Y)$ and $\sigma_{XY} = Cor(X,Y)$. To perform a bootstrap analysis for this example we first write a function to generate an (X,Y) pair as the underlying population.

```
gen.xy = function(n, rho=0.4, seed=NULL) {
  if (!is.null(seed)) set.seed(seed)
  z1 = rnorm(n)
  z2 = rnorm(n)
  x = z1
  y = rho*z1 + sqrt(1-rho^2)*z2
  return(list(x=x,y=y))
}
```

Here we write the function needed to estimate α.

```
alpha.fn = function(data, index=1:length(data$x)) {
  x = data$x[index]
  y = data$y[index]
  return((var(y)-cov(x,y))/(var(x)+var(y)-2*cov(x,y)))
}
```

The true value of α is given by

```
alpha.true = alpha.fn(gen.xy(1e7))
```

Each panel obtained display 100 simulated returns for investments X and Y.

```
n = 100
par(mfrow=c(2,2), mar=c(4, 3, 1, 1), cex=0.7,cex.lab=0.75, family="Times")
data = gen.xy(n,seed=1)
plot(data$x,data$y, pch=20, col="darkgreen", xlab="X", ylab="Y")

data2 = gen.xy(n,seed=2)
plot(data2$x,data2$y, pch=20, col="darkgreen", xlab="X", ylab="Y")

data3 = gen.xy(n,seed=3)
plot(data3$x,data3$y, pch=20, col="darkgreen", xlab="X", ylab="Y")

data4 = gen.xy(n,seed=4)
plot(data4$x,data4$y, pch=20, col="darkgreen", xlab="X", ylab="Y")
```

From left to right and top to bottom, the resulting estimates for α are 0.5435079, 0.3680172, 0.629502 and 0.527159, respectively

Let us see how variable is this $\hat{\alpha}$ over 1000 simulations (data sets) from the true population. Also, we can obtain the histogram of estimates of α based on 1000 bootstrap samples from a single data set when the we used seed=1.

```
R = 1000
alpha.hat.real = numeric(R)
for (r in 1:R) {
  data.r = gen.xy(n)
  alpha.hat.real[r] = alpha.fn(data.r)
}

par(mfrow=c(1,2), family="Times")
```

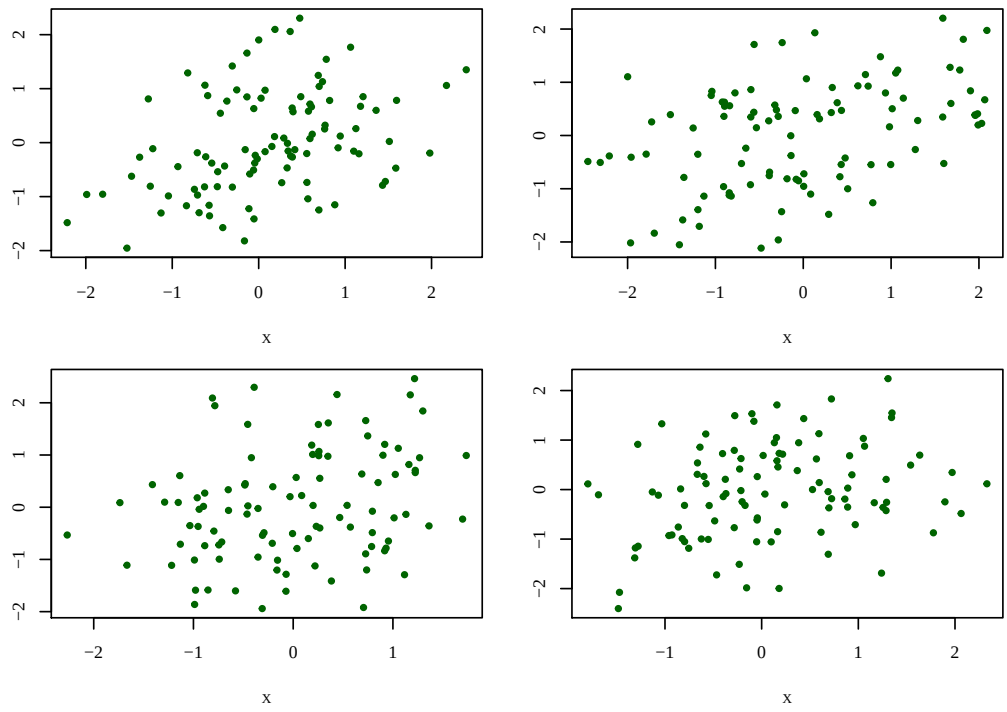


Figure 28: Four different 100 simulated returns for investemnets X and Y

```
hist(alpha.hat.real, col="orange",
     main="Samples from true population",
     xlab=expression(hat(alpha)))
abline(v=alpha.true, lwd=3, lty=2)


B = 1000
alpha.hat.boot = numeric(B)
for (b in 1:B) {
  index.b = sample(1:n,replace=TRUE)
  alpha.hat.boot[b] = alpha.fn(data,index.b)
}

hist(alpha.hat.boot, col="lightblue",
     main="Bootstrap samples from data with seed=1",
     xlab=expression(hat(alpha)))
abline(v=alpha.true, lwd=3, lty=2)
```

It is easy to see that the simulated standard error for $\hat{\alpha}$ is 0.0754701. Here the bootstrap standard estimates is 0.071239. Also, the bootstrap bias estimate and a basic bootstrap confidence interval for α are given by

```
alpha.hat = alpha.fn(data)

mean(alpha.hat.boot) - alpha.hat
```

```
## [1] -0.001993025
```

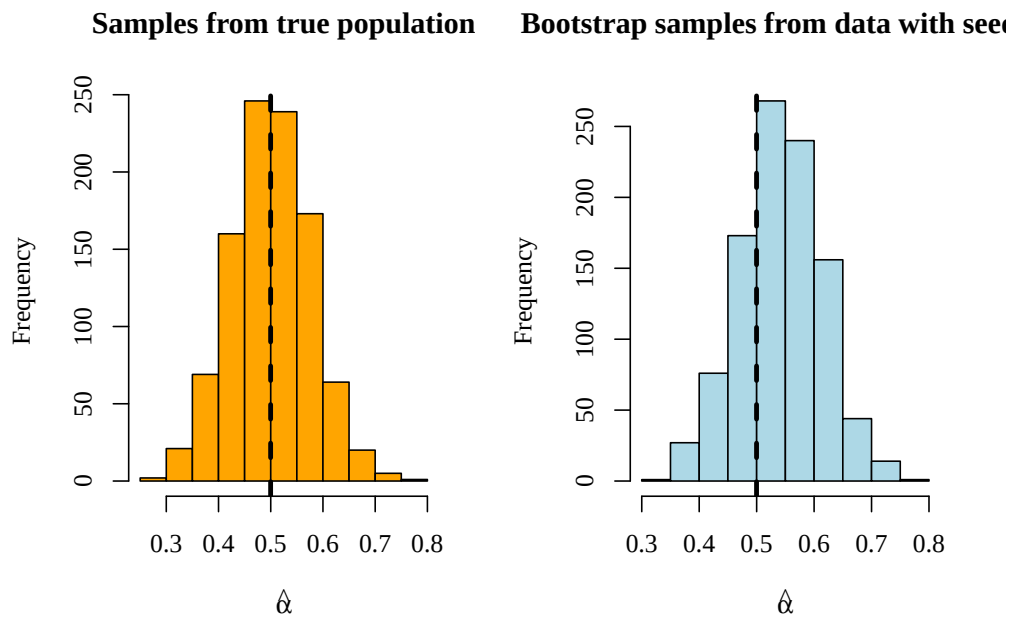


Figure 29: Histogram of the estimtes of the proportion of investment of X using bootstrap method and corresponding one when samples are directly obtained from the underlying population

```
alpha = 0.05
q.lo = quantile(alpha.hat.boot, prob=alpha/2)
q.hi = quantile(alpha.hat.boot, prob=1-alpha/2)
basic.boot.int = c(2*alpha.hat-q.hi, 2*alpha.hat-q.lo)
basic.boot.int
```

```
##     97.5%      2.5%
## 0.4071947 0.6901904
```

4.6 Example 3. Using R packages

Although writing R codes for implementing bootstrap ideas is simple but there are R packages that could also be do bootstrapping. One of the famous one is the package called `boot` which you can easily download and install in `RStudio`. After you installed the package `boot` then one can use the function `boot()` which takes several argument. The first three arguments that are required are as follow `boot(data, statistic, R)`

- **data**: A data vector, matrix, or data frame to which bootstrap re-sampling is to be applied.
- **Statistic**: A function that returns the (vector-valued) statistic to be bootstrapped. This function must take two inputs as **(data, indices)** where the data will be put in order by "indices".
- **R**: The number of bootstrap replications (We used B in the notes).

Now, I describe how to use the package to implement some bootstrap examples.

Again, we use the data set *LawSchool.txt* which gives the average LSAT and GPA scores for the 1973 entering classes of 15 American Law schools.

```
dir <- getwd()
LawSchool<-read.table(file= paste(dir, "/LawSchool.txt", sep=""))
attach(LawSchool)
```

The data set are plotted in Figure 1.

```
library(calibrate)
```

```
## Loading required package: MASS
```

```
plot(GPA, LSAT, pch=20, cex=1, xlim=c(2.7, 3.5), ylim=c(540, 670), col="red")
textxy(GPA, LSAT,  1:length(LSAT), offset=0.75, cex=0.75)
```

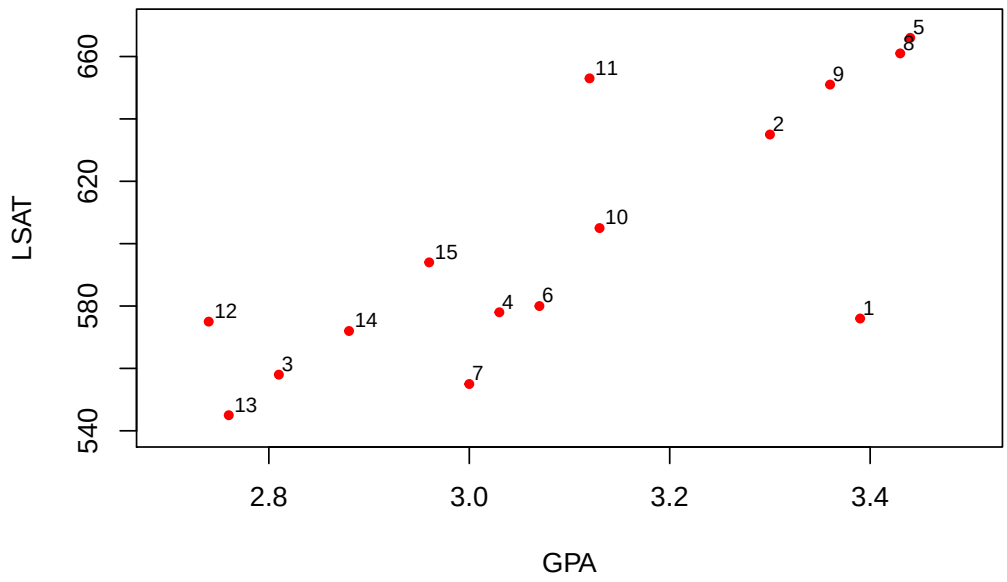


Figure 30: The scatter plot of the GPA and LSAT scores

We first define a function to compute the statistic that we are interested in bootstrapping, which is here the correlation coefficient. Remember that the correlation coefficient between LSAT and GPA is $r = 0.7763745$.

```
r.statistic<-function(Pop, indices){
  Pop<-Pop[indices,]
  r<-cor(Pop$LSAT, Pop$GPA)
  return(r)
  }
```

Now, to get a bootstrap confidence interval from say B=500 bootstrap samples using `boot()` and following the percentile method.

```
library(boot)
boot.results<-boot(LawSchool, r.statistic, R=500)
boot.ci(boot.results, conf=0.9, type="perc")
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 500 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = boot.results, conf = 0.9, type = "perc")
##
```

```
## Intervals :
## Level     Percentile
## 90%   ( 0.5285,  0.9526 )
## Calculations and Intervals on Original Scale
```

We can also get the histogram of bootstrap estimate of the correlation coefficient.

```
hist(boot.results$t, col="brown", border="white", xlab="r", main="")
abline(v=cor(LSAT, GPA), lwd=2, lty=3)
```

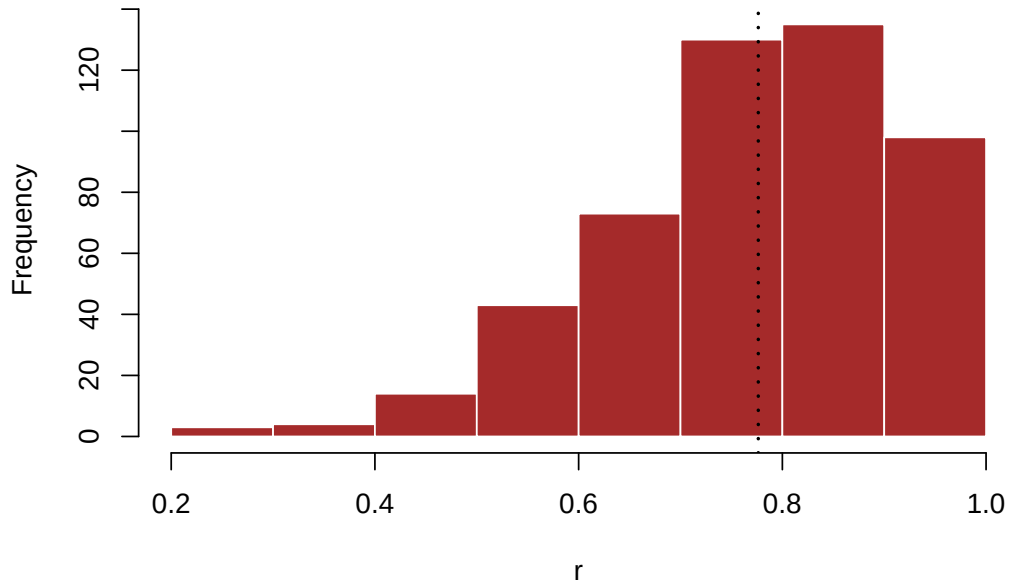


Figure 31: Histogram of bootstrap sampling distribution of correlation coefficients

5 Bootstrap for estimating prediction error

There are several approaches that can be used to estimate the prediction error pertinent to a prediction (statistical learning) model.

5.1 Using bootstrap data for model fitting and original data for prediction

One approach is to fit the model in question on a set of bootstrap samples, and then keep track of how well it predicts the original training set. Suppose we have B bootstrap datasets B.Data$_b$, $b = 1, \ldots, B$ generated from the training data $\mathcal{T}$. Let $\hat{f}^{*b}(\mathbf{x}_i)$ be the predicted value at $\mathbf{x}_i$, from the model fitted to the bth bootstrap dataset B.Data$_b$. Then, one can estimate the prediction error as follows:

$$\widehat{\text{Err}}_{boot} = \frac{1}{B}\frac{1}{n}\sum_{b=1}^{B}\sum_{i=1}^{n} L(y_i, \hat{f}^{*b}(\mathbf{x}_i)). \tag{4}$$

In this approach, the bootstrap datasets are used for the training purpose, while the original training set is used for testing. As bootstrap samples share a lot of information with the original sample, one can easily see that this method will not work well in general. The overlap between the training and test error can make overfit predictions look unrealistically good.

Consider for example a 1-nearest neighbor classifier applied to a two-class classification problem with the same number of observations in each class. Suppose that predictors and class labels are independent of each other. The true test error is 0.5. But contribution to the bootstrap estimate $\widehat{\text{Err}}_{boot}$ will be zero unless the observation i does not appear in the bootstrap sample b. Here we perform a very simple simulation study where we generate $p = 10$ independent features from normal distribution independent of class labels. We use 1NN as classifier on bootstrapped data and find the prediction error using the original sample as our test data.

```
set.seed(2021)
n<-100
p<-10
B<- 1000

X <- matrix(rnorm(n*p), n, p)
y<- sample(rep(0:1, n/2), n, replace=FALSE)

Data <- data.frame(X, y)

errors <- c()
for (b in 1:B){
  inds<- sample(1:nrow(Data), n, replace=TRUE)
  B.data <- Data[inds, ]

  knn.train <- class::knn(train = B.data[, -ncol(B.data)],
                  test = B.data[, -ncol(B.data)],
                  cl   = B.data$y,
                  k = 1, prob = FALSE)


  knn.test <- class:: knn(train=B.data[, -ncol(B.data)],  # train x
                 test = Data[, -ncol(Data)],  # test x
                 cl = B.data$y,   #train y
                 k = 1, prob = FALSE)


  test.error.knn <- mean(Data$y != knn.test )
  errors = c(errors, test.error.knn)
}

mean(errors)
```

```
## [1] 0.19983
```

We observe that estimated test error is 0.19983 which is far below the test error. Note that,

$$\begin{aligned}\Pr(\text{observation } i \in \text{ bootstrap } b) &= 1 - (1 - \frac{1}{n})^n \\ &\approx 1 - e^{-1} \\ &= 0.632.\end{aligned}$$

So, the expectation of $\widehat{\text{Err}}_{boot}$ is about $0.5 \times 0.368 = 0.184$, similar to the number we observe in our simulation study, which is far below the correct test error rate 0.5.

5.2 Using bootstrap data for model fitting and left out data for prediction

As we showed earlier there is about 63 percent chance that any observation being selected in the bootstrap sample b. That means from a sample of size n on average around 37 percent of the observations will be left out. So, mimicking the idea of cross-validation, a better approach is to keep track of predictions from bootstrap samples not containing that observation. The leave-one-out bootstrap estimate of prediction (test) error can be obtained by

$$\widehat{\text{Err}}^{(1)} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{|C^{-i}|}\sum_{b\in C^{(-i)}} L(y_i, \hat{f}^{*b}(\mathbf{x}_i)) \tag{5}$$

Here

$$C^{(-i)} = \{b \in \{1,\ldots,B\} : (\mathbf{x}_i, y_i) \notin \text{B.Data}_b\},$$

is the set of indices of the bootstrap samples b that do not contain observation i, and $|C^{(-i)}|$ is the number of such samples. To calculate (5) we either have to choose B large enough to ensure that all the $|C^{-i}|$'s are greater than zero, or we can leave out the terms corresponding to those with $|C^{-i}| = 0$.

Lets us use this approach in our toy example.

```
B <- 500
C<-matrix(FALSE, n,B)
L<-matrix(0, n, B)
for (b in 1:B){
  inds<- sample(1:nrow(Data), n, replace=TRUE)
  ind.unique<- unique(sort(inds, decreasing=FALSE))
  C[-ind.unique,b] <- TRUE

  B.data <- Data[ind.unique, ]
  T.data <- Data[-ind.unique, ]
  knn.train <- class::knn(train = B.data[, -ncol(B.data)],
                  test = B.data[, -ncol(B.data)],
                  cl   = B.data$y,
                  k = 1, prob = FALSE)


  knn.test <- class:: knn(train=B.data[, -ncol(B.data)],  # train x
                 test = T.data[, -ncol(Data)],  # test x
                 cl = B.data$y,   #train y
                 k = 1, prob = FALSE)

 L[-ind.unique, b] <- (T.data$y != knn.test)
}

error <- mean(apply(L, 1, sum)/apply(C, 1, sum))
error
```

```
## [1] 0.5467914
```

As we see here test error estimate here is not very bad 0.5467914. In other words, $\widehat{\text{Err}}^{(1)}$ solved the overfitting problem suffered by $\widehat{\text{Err}}_{boot}$. But it still has the training set size bias. The average number of distinct observations in each bootstrap sample is about $0.632 \times n$, so its bias will roughly behave like that of twofold cross-validation.

Thus, if the learning curve has considerable slope at sample size $n/2$, the leave-one-out bootstrap will be biased upward as an estimate of the true error.

One might also want to consider the following approach to estimate the prediction error. For each bootstrap sample B.Data_b denote the left out observations by $\text{T.Data}_b = \mathcal{T} - \text{B.Data}_b$, $b = 1, \ldots, B$. Let

$$\mathbb{I}_i^b = \begin{cases} 1 & \text{if } (\mathbf{x}_i, y_i) \in \text{B.Data}_b, \\ 0 & \text{if } (\mathbf{x}_i, y_i) \notin \text{B.Data}_b. \end{cases}$$

The leave-one-out bootstrap estimate of prediction (test) error can then be obtained by

$$\widehat{\text{Err}}^* = \frac{\sum_{i=1}^n \sum_{b=1}^B \mathbb{I}_i^B L(y_i, \hat{f}^{*b}(\mathbf{x}_i))}{\sum_{i=1}^n \sum_{b=1}^B \mathbb{I}_i^B} \tag{6}$$

```
errors <- c()
for (b in 1:B){
  inds<- sample(1:nrow(Data), n, replace=TRUE)
  B.data <- Data[inds, ]
  T.data <- Data[-inds, ]
  knn.train <- class::knn(train = B.data[, -ncol(B.data)],
                  test = B.data[, -ncol(B.data)],
                  cl   = B.data$y,
                  k = 1, prob = FALSE)


  knn.test <- class:: knn(train=B.data[, -ncol(B.data)],  # train x
                test = T.data[, -ncol(T.data)],  # test x
                cl = B.data$y,   #train y
                k = 1, prob = FALSE)


  test.error.knn <- mean(T.data$y != knn.test )
  errors = c(errors, test.error.knn)
}

mean(errors)
```

```
## [1] 0.5487562
```

This approach also results in a relatively good estimate of the test error, i.e., 0.5487562.

5.3 0.632 estimator

The 0.632 estimator is designed to alleviate the issue with the bias mentioned in calculating leave-one-out bootstrap approach. This estimator is defined as

$$\widehat{\text{Err}}^{(0.632)} = 0.368 \cdot \overline{\text{err}} + 0.632 \cdot \widehat{\text{Err}}^{(1)}. \tag{7}$$

Here $\overline{\text{err}}$ is the training error

$$\overline{\text{err}} = \frac{1}{n} \sum_{i=1}^n L(y_i, \hat{f}(\mathbf{x}_i)),$$

using the whole training dataset. Note that, the deviation of the 0.632 estimator is complex; intuitively it pulls the leave-one-out bootstrap estimate down toward the training error rate, and hence reduces its upward bias.

The 0.632 estimator works well in light fitting situations, but can break down in overall ones. To see this, let us consider our toy example again, using a 1NN classifier and as we see below, the training error is 0.

```
knn.train <- class::knn(train = Data[, -ncol(Data)],
                        test = Data[, -ncol(Data)],
                        cl   = Data$y,
                        k = 1, prob = FALSE)
(err.bar <- sum(Data$y != knn.train))
```

```
## [1] 0
```

So, the 0.632 estimator gives $\widehat{\text{Err}}^{(0.632)} = 0.368 \cdot \overline{\text{err}} + 0.632 \cdot \widehat{\text{Err}}^{(1)} = 0.3455721$. However, the true error is 0.5.

5.4 Adjusted 0.632 estimate

One can improve the 0.632 estimator by taking into account the amount of overfitting. Define γ as the *non-information error rate*, to represent the error rate of our prediction rule if the inputs and class labels were independent. An estimate of γ can be obtained by averaging the prediction rule error over all possible combinations of targets y_i, and predictors $\mathbf{x}_{i'}$:

$$\hat{\gamma} = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{i'=1}^{n} L(y_i, \hat{f}(\mathbf{x}_{i'})).$$

Using this, the *relative overfitting rate* is defined by

$$\hat{R} = \frac{\widehat{\text{Err}}^{(1)} - \overline{\text{err}}}{\hat{\gamma} - \overline{\text{err}}},$$

a quantity that ranges from 0 if there is no overfitting (i.e., $\widehat{\text{Err}}^{(1)} - \overline{\text{err}}$) to 1 if the overfitting equals the no-information value $\hat{\gamma} - \overline{\text{err}}$. Finally, the adjusted 0.632 rule is defined as

$$\widehat{\text{Err}}^{(0.632+)} = (1 - \hat{w}) \cdot \overline{\text{err}} + \hat{w} \cdot \widehat{\text{Err}}^{(1)}, \tag{8}$$

with

$$\hat{w} = \frac{0.632}{1 - 0.368\hat{R}}.$$

Note that here, the weight $\hat{w}$ ranges from 0.632 if $\hat{R} = 0$ to 1 if $\hat{R} = 1$. In other words, $\widehat{\text{Err}}^{(0.632+)}$ ranges from $\widehat{\text{Err}}^{(0.632)}$ to $\widehat{\text{Err}}^{(1)}$.

Again, the derivation of $\widehat{\text{Err}}^{(0.632+)}$ is technical. But, roughly speaking, it produces a compromise between the leave-one-out bootstrap and the training error rate that depends on the amount of overfitting. For the 1NN approach with features that are independent of class labels, $\hat{w} = \hat{R} = 1$, and so $\widehat{\text{Err}}^{(0.632+)} = \widehat{\text{Err}}^{(1)}$, which has the correct expectation of 0.5. In other problems with less overfitting, $\widehat{\text{Err}}^{(0.632+)}$ will lie somewhere between $\overline{\text{err}}$ and $\widehat{\text{Err}}^{(1)}$.