**Title of relevant chapter: Sources of variability in AUC**

# Online Appendix 7.A: The bootstrap method

### Online Appendix 7.A.1: Code Listing

rm( list = ls()) # mainBootstrapSd.R

source("Transforms.R")

source("LL.R")

source("RocfitR.R")

source("RocOperatingPoints.R")

source("FixRocCountsTable.R")

source("WilcoxonCountsTable.R")

FOM <- "Az"

#FOM <- "Wilcoxon"

B <- 2000;seed <- 1;set.seed(seed)

cat("FOM = ", FOM, ", seed = ", seed, ", B = ", B, "\n")

RocCountsTable = array(dim = c(2,5))

RocCountsTable[1,] <- c(30,19,8,2,1)

RocCountsTable[2,] <- c(5,6,5,12,22)

K <- c(sum(RocCountsTable[1,]), sum(RocCountsTable[2,])) # this is the K vector

if (FOM == "Az") { # AUC for observed data

ret <- RocfitR(RocCountsTable)

if (ret$Az == -1) stop("RocfitR did not converge on original data")

} else {

ret <- WilcoxonCountsTable(RocCountsTable)

}

Az <- ret$Az

# ready to bootstrap; first put the counts data into a linear form

z1 <- rep(1:length(RocCountsTable[1,]),RocCountsTable[1,])#convert counts table to array

z2 <- rep(1:length(RocCountsTable[2,]),RocCountsTable[2,])#do:

AUC <- array(dim = B)#to save the bs AUC values

for ( b in 1 : B){

while (1) {

RocCountsTable\_bs <- array(dim = c(2,length(RocCountsTable[1,])))

k1\_b <- ceiling( runif( K[ 1 ] ) \* K[ 1 ] ) # bs indices for non-diseased

k2\_b <- ceiling( runif( K[ 2 ] ) \* K[ 2 ] ) # bs indices for diseased

bsTable <- table(z1[k1\_b])

RocCountsTable\_bs[1, as.numeric(names(bsTable))] <- bsTable#convert array to frequency table

bsTable <- table(z2[k2\_b])

RocCountsTable\_bs[2, as.numeric(names(bsTable))] <- bsTable #do:

RocCountsTable\_bs[is.na(RocCountsTable\_bs )] <- 0 #replace NAs with zeroes

if (FOM == "Az") {

temp <- RocfitR(RocCountsTable\_bs) # AUC for observed data

} else {

temp <- WilcoxonCountsTable(RocCountsTable\_bs)

}

AUC[b] <- temp$Az

if (AUC[b] != -1) break # a return of -1 means Az did not converge

}

}

Var <- var(AUC)

stdAUC <- sqrt(Var)

cat("OrigAUC = ", Az, ", meanAUCbs = ", mean(AUC), ", stdAUC = ", stdAUC, "\n")

Line 10 defines the FOM (figure of merit) to be used to calculate the figure of merit. Currently the binormal model based AUC, denoted "Az", is selected, but if the commenting were reversed, empirical AUC, denoted "Wilcoxon", would be selected.

Line 12 sets the number of bootstraps B to 200 and for pedagogical purposes the seed variable is set to 1 (in real applications it should be set to NULL). Line 15-16 is the observed ROC counts table, stored in the variable RocCountsTable. One should confirm that it contains the numbers in the body of book Table 4.1. This is the observed data. Line 18 uses the function sum() to add the numbers in the first and second rows of RocCountsTable to obtain the  vector, whose first element is the number of non-diseased cases (i.e., 60) and whose second element is the number of diseased cases (i.e., 50). Try it: insert a break point at line 29 (click on the gray area to the left of the line number whereupon a red dot appears) and click on Source. Look at the Environment panel: one sees that the K vector is (60, 50). Because of the choice of FOM, line 21 uses the function RocfitR() to determine the area under the binormal fitted ROC curve, whose value, 0.87045189, is displayed in the Environment panel. To summarize, lines 1 – 26 calculate the binormal model AUC for the counts data in book Table 4.1. At line 22 a convergence check is made; if RocfitR returns -1, that signals that the algorithm failed to converge on the original dataset and the process comes to a grinding halt. Modern fitting algorithms implemented in the software and described in later book Chapters, are not subject to this problem.

One is now ready to bootstrap the data.

First, it is necessary to convert the counts data (which is like a histogram) to a "linear form", where each case is represented with a single z-sample; this is accomplished by lines 29-30. They use the rep() function. For example, rep(1,5) repeats 1 five times, and the function also works when the second argument is an array. Study and reproduce the following code snippets by carefully selecting the appropriate part of the right hand side of line 29, and clicking Run:

### Online Appendix 7.A.2: Code Snippet

Browse[2]> RocCountsTable[1,]

[1] 30 19 8 2 1

Browse[2]> length(RocCountsTable[1,])

[1] 5

Browse[2]> rep(1:length(RocCountsTable[1,]),RocCountsTable[1,])

[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3

[56] 3 3 4 4 5

Focusing on the last line, one can choose to count the number of occurrences of 1, or take the author's word for it, there are 30 ones followed by 19 twos, 8 threes, two fours and one five, essentially a linearized form of the ratings for non-diseased cases in book Table 4.1. Using the same method, line 30 linearizes the counts data for diseased cases. Line 31 defines an array AUC, of length 200, to hold the bootstrapped AUC values. Click Next enough times to advance the code pointer to line 32.

Now comes the interesting part. The for-loop beginning at line 32 is executed 200 times, once per bootstrap. Line 34 allocates memory for a new bootstrapped counts table, RocCountsTable\_bs. To see this more clearly, click Next enough times to advance the code pointer to line 35. Select the left hand side of line 34 and click Run. The Console window output is:

### Online Appendix 7.A.3: Code Snippet

Browse[2]> array(dim = c(2,length(RocCountsTable[1,])))

[,1] [,2] [,3] [,4] [,5]

[1,] NA NA NA NA NA

[2,] NA NA NA NA NA

This has the same structure as the counts table in book Table 4.1, so this variable is ready to receive data. Click Next enough times to advance the code pointer to line 42. On line 35 select the left hand side and click Run.

### Online Appendix 7.A.4: Code Snippet

Browse[2]> k1\_b

[1] 16 23 35 55 13 54 57 40 38 4 13 11 42 24 47 30 44 60 23 47 57 13 40 8 17 24 1 23 53 21

[31] 29 36 30 12 50 41 48 7 44 25 50 39 47 34 32 48 2 29 44 42 29 52 27 15 5 6 19 32 40 25

As one sees, each integer in the 60 printed values (go ahead and count them) is in the range 1 – 60. They represent the indices of bootstrapped non-diseased cases. Scanning the values one sees that original cases 1 and 2 were each picked once, case 3 was not picked, etc. What happened is this: runif(K[1]) yielded 60 samples from the uniform distribution *U(0,1)*. Multiplying by 60 yields 60 values in the open interval (0,60), e.g., 0.0001 to 59.999. The ceiling() function converts them to integers in the closed interval [1,60].

*The ratings of the cases are obtained by appropriately indexing the original z1 array.* For example, z1[1] = 1, z1[31] = 2, z1[49] = 4 and z1[60] = 5:

### Online Appendix 7.A.5: Code Snippet

Browse[2]> z1

[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

[46] 2 2 2 2 3 3 3 3 3 3 3 3 4 4 5

Browse[2]> z1[1]

[1] 1

Browse[2]> z1[31]

[1] 2

Browse[2]> z1[59]

[1] 4

Browse[2]> z1[60]

[1] 5

Line 37 converts the linear data contained in z1[k1\_b] into a counts table using the table() function, and populates the cells in the 1st row of RocCountsTable\_bs. Carefully select the left hand side of line 38 and click Run.

### Online Appendix 7.A.6: Code Snippet

Browse[2]> RocCountsTable\_bs[1, as.numeric(names(bsTable))]

[1] 30 21 8 0 1

This tells us that non-diseased cases with rating 1 occurred 30 times, those with rating 2 occurred 21 times, those with rating 3 occurred 8 times and those with rating 5 occurred once. No cases with rating 4 are represented in the 1st bootstrapped non-diseased case sample.

Line 40 and 41 perform similar operations on diseased cases. Select RocCountsTable\_bs and click Run.

### Online Appendix 7.A.7: Code Snippet

Browse[2]> RocCountsTable\_bs

[,1] [,2] [,3] [,4] [,5]

[1,] 30 21 8 0 1

[2,] 2 5 5 15 23

Line 43 runs RocfitR() on the current bootstrapped dataset and saves the result to a temporary variable temp. Line 47 saves the current value of AUC to the array AUC[b].

Now here comes a complication, similar to the non-convergence issue discussed earlier. At line 33 there starts a "*do forever or infinite while loop*", while(1), which keeps looping between lines 33-49 unless a break statement is encountered. This is because one needs to allow for the possibility that RocfitR may not converge. If this happens then RocfitR returns -1 and the if clause at line 48 evaluates to FALSE, so control is returned to just inside the while loop, i.e., another dataset is simulated, and one tries again. Eventually, and hopefully, RocfitR will converge and line 48 causes control to break out of the while loop, and the next iteration of the bootstrap index starts.

On successful completion of the 200 values of *b*, control passes to line 51, which uses the var() function to calculate the variance of the 200 bootstrap values. The next line computes the standard deviation and the last line prints out the final values.

Exit debug mode (click on Stop button) and remove the breakpoint (by clicking on the red dot – it disappears) and click source, yielding the following output.

### Online Appendix 7.A.8: Code output

> source(…)

FOM = Az , seed = 1 , B = 200

OrigAUC = 0.8704519 , meanAUCBs = 0.8671713 , stdAUC = 0.04380523

>

The original AUC is a single value; one cannot calculate a standard deviation of a single value. However, one can calculate the standard deviation of 200 bootstrapped AUC samples. Notice that the mean of the bootstrapped samples is not equal to the value for the original dataset. In quoting a confidence interval it is customary to center it on the original value, not the mean of the bootstrapped values.

Try repeating this with 2000 bootstraps and different values of seed. This should yield an idea of the stability of the standard deviation estimate and how many bootstraps are sufficient. In the author's experience 200 is usually enough. Repeat all steps with the Wilcoxon selected as the FOM. One advantage of the Wilcoxon is that there are no convergence issues.

For a greater research-level challenge, use RJafroc to simulate binormal datasets. The package has a binormal model simulator and one can systematically study the effect on standard deviation of varying the two model parameters (*a*,*b*) and the numbers of cases. In this situation one does not need the bootstrap – one simply samples the population. As AUC approaches unity one expects the standard deviation to approach zero, and conversely, one expects the standard deviation to be large when AUC is close to 0.5. The standard deviation depends inversely on the number of cases. There is an interesting paper by Burgess1 that looked into this issue at a theoretical level, but no one has studied it systematically. It is relevant to a practical problem – what is the optimum AUC that the study designer should aim for in order to maximize power. Large AUC might give small standard deviation but the effect size is also expected to be small (if AUC is already close to unity there in not much room for improvement). Section 11.12 has information that is relevant to attacking this problem.

# Online Appendix 7.B: The jackknife method

Here is the listing of the file mainJackknifeSd.R:

### Online Appendix 7.B.1: Code Listing

rm( list = ls()) # MainJackknifeSd.R

source("Transforms.R")

source("LL.R")

source("RocfitR.R")

source("RocOperatingPoints.R")

source("FixRocCountsTable.R")

source("WilcoxonCountsTable.R")

FOM <- "Az"

#FOM <- "Wilcoxon"

cat("FOM = ", FOM, "\n")

RocCountsTable = array(dim = c(2,5))

RocCountsTable[1,] <- c(30,19,8,2,1)

RocCountsTable[2,] <- c(5,6,5,12,22)

K <- c(sum(RocCountsTable[1,]), sum(RocCountsTable[2,])) # this is the K vector

if (FOM == "Az") { # AUC for observed data

Az <- RocfitR(RocCountsTable)

if (Az$Az == -1) stop("RocfitR did not converge on original data")

} else {

Az <- WilcoxonCountsTable(RocCountsTable)

}

Az = Az$Az # AUC for observed data

z1 <- rep(1:length(RocCountsTable[1,]),RocCountsTable[1,])#convert frequency table to array

z2 <- rep(1:length(RocCountsTable[1,]),RocCountsTable[2,])#do:

AUC <- array(dim = sum(K)); Y <- array(dim = sum(K))

z\_jk <- array(dim = sum(K))

for ( k in 1 : sum(K)){

RocCountsTable\_jk <- array(dim = c(2,length(RocCountsTable[1,])))

if ( k <= K[ 1 ]){

z1\_jk <- z1[ -k ]

z2\_jk <- z2

}else{

z1\_jk <- z1

z2\_jk <- z2[ -(k - K[ 1 ]) ]

}

RocCountsTable\_jk[1,1:length(table(z1\_jk))] <- table(z1\_jk)#convert array to frequency table

RocCountsTable\_jk[2,1:length(table(z2\_jk))] <- table(z2\_jk)#do:

RocCountsTable\_jk[is.na(RocCountsTable\_jk)] <- 0#replace NAs with zeroes

if (FOM == "Az") {

temp <- RocfitR(RocCountsTable\_jk) # AUC for observed data

} else {

temp <- WilcoxonCountsTable(RocCountsTable\_jk)

}

AUC[k] <- temp$Az

Y[k] <- sum(K)\*Az - (sum(K)-1)\*AUC[k]

if (AUC[k] == -1) stop("RocfitR did not converge in jackknife loop")

}

Var <- var(AUC) \* ( sum(K) - 1)^2 / sum(K) #Efron and Stein's paper

stdAUC <- sqrt(Var)

cat("OrigAUC = ", Az, "jackknifeMeanAuc = ", mean(AUC), "stdAUC = ", stdAUC, "\n")

Notice that the set.seed statement has been removed, as a random number generator is not needed in the jackknife method. Currently FOM <- "Az" is selected. Lines 30-52 have replaced the previous bootstrap code. Line 32 begins a for-loop in k, the case to be removed. For the non-diseased cases, defined by the if block between lines 34 and 37, the construct z1[-k] removes element k from the array z1. At the Console prompt, type x ­<- seq(1:10) upon which R prints the expected values: [1] 1 2 3 4 5 6 7 8 9 10. Now enter x[-1] at the Console prompt. One should see: [1] 2 3 4 5 6 7 8 9 10. Try with other negative values, in the range 1 through 10, inside the square bracket. R makes it easy to jackknife a case (the negative index also works with higher dimension arrays). Try experimenting with values outside the allowed range: x[-11]; since element 11 does not exist, no element is removed and the original array is returned.

On the first pass through the for-loop with k = 1, the first non-diseased case is removed and the result is assigned at line 35 to z1\_jk, whose length is 59. But z2, which contains the ratings of the 50 diseased cases, is assigned, without alteration, to the variable z2\_jk. Lines 41-42 construct the counts table from the jackknifed data, named RocCountsTable\_jk, line 43 replaces any NAs with 0 and line 45 uses the RocfitR() function to calculate . On the second pass through the for-loop, k = 2, the second non-diseased case is removed, and a new value is returned by RocfitR, namely .

When *k* = 61 the else block of the if statement, namely lines 37-39, is executed. This time the non-diseased ratings are copied, unaltered, to array z1\_jk, whose length is 60, but the first diseased case is removed resulting in the ratings array of length 49, which is assigned to z2\_jk. The RocfitR function yields . On the next pass through the for-loop, k = 62, the 2nd diseased case is removed and the RocfitR function yields , and so on. On exit from the for-loop, all elements of AUC are filled in (they were initialized with NAs at line 30). Line 54 implements the jackknife estimate of the variance; note the explicit presence of the variance inflation factor. The last two lines calculate the standard deviation, which is the square root of the variance, and prints out the results.

Use debugging techniques, involving inserting breakpoints followed by sourcing the code and clicking on Next, to thoroughly understand what is going on. Finally, remove breakpoints, stop debug mode and click on source. [Sometimes break points are not visible; use the Debug – Clear All Breakpoints to remove them.]

### Online Appendix 7.B.2: Code output

> source(...)

FOM = Az

OrigAUC = 0.8704519

jackknifeMeanAuc = 0.8704304

stdAUC = 0.03861591

R has sophisticated bootstrap and jackknife analysis routines. The author leaves it to a user to implement them in the current context.

# Online Appendix 7.C: The DeLong method

This relates to book Section 7.4. A listing of file mainDeLongSd.R follows:

### Online Appendix 7.C.1: Code Listing

rm(list = ls()) # mainDeLongSd.R

source("Wilcoxon.R");source("DeLongVar.R")

seed <- 1;set.seed(seed)

mu <- 1.5;sigma <- 1.3;K1 <- 50;K2 <- 52

cat("seed = ", seed, "\nK1 = ", K1, "\nK2 = ", K2,

"\nmu = ", mu, "\nsigma = ", sigma, "\n")

# brute force method to find the population mean and stdDev

empAuc <- array(dim = 10000)

for (i in 1:length(empAuc)) {

zk1 <- rnorm(K1);zk2 <- rnorm(K2, mean = mu, sd = sigma)

empAuc[i] <- Wilcoxon(zk1, zk2)

}

meanempAuc <- mean(empAuc)

stdDevempAuc <- sqrt(var(empAuc))

cat("population mean empAuc = ", meanempAuc,

"\npopulation stdDev empAuc = ", stdDevempAuc, "\n")

# one more trial

zk1 <- rnorm(K1);zk2 <- rnorm(K2, mean = mu, sd = sigma)

empAuc <- Wilcoxon(zk1, zk2)

ret <- DeLongVar(zk1,zk2)

stdDevDeLong <- sqrt(ret)

cat("1 sample empAuc = ", empAuc,

"\nstdDev DeLong = ", stdDevDeLong, "\n")

This code is discussed in the book. Sourcing it yields:

### Online Appendix 7.C.2: Code output

> source(...)

seed = 1

K1 = 50

K2 = 52

mu = 1.5

sigma = 1.3

population mean empAuc = 0.8192

population stdDev empAuc = 0.04177

1 sample empAuc = 0.8627

stdDev DeLong = 0.03804

# Online Appendix 7.D: A calibrated simulator for a single dataset

This relates to book Section 7.7. A listing of file mainCalSimulator.R follows:

### Online Appendix 7.D.1: Code Listing

rm( list = ls()) # mainCalSimulator.R

source("Transforms.R")

source("LL.R")

source("RocfitR.R")

source("RocOperatingPoints.R")

source("FixRocCountsTable.R")

source("SimulateRocCountsTable.R")

source("WilcoxonCountsTable.R")

FOM <- "Az"

#FOM <- "Wilcoxon"

seed <- 2;set.seed(seed);P <- 2000#number of pop. samples

RocCountsTable = array(dim = c(2,5))

RocCountsTable[1,] <- c(30,19,8,2,1)

RocCountsTable[2,] <- c(5,6,5,12,22)

K <- c(sum(RocCountsTable[1,]), sum(RocCountsTable[2,])) # this is the K vector

# to build the model we have to do a parametric fit first

ret <- RocfitR(RocCountsTable) # AUC for observed data

AUC\_org <- ret$Az;a <- ret$a;b <- ret$b;zeta <- ret$zeta;mu <- a/b; sigma <- 1/b;

zeta <- zeta/b # need to also scale zetas

if (FOM == "Az") { # AUC for observed data

AUC\_org <- RocfitR(RocCountsTable)

} else {

AUC\_org <- WilcoxonCountsTable(RocCountsTable)

}

AUC\_org <- AUC\_org$Az;

cat("Calibrated simulator values: a, b, zetas:", ret$a, ret$b, ret$zeta, "\n")

AUC <- array(dim = P)#to save the pop sample AUC values

a <- array(dim = P);b <- array(dim = P)

for ( p in 1 : P){

while (1) {

RocCountsTableSimPop <- SimulateRocCountsTable(K, mu, sigma, zeta)

RocCountsTableSimPop[is.na(RocCountsTableSimPop )] <- 0#replace NAs with zeroes

if (FOM == "Az") {

temp <- RocfitR(RocCountsTableSimPop) # AUC for observed data

if (temp[1] != -1) {# a return of -1 means RocFitR did not converge

AUC[p] <- temp$Az;a[p] <- temp$a;b[p] <- temp$b

break

}

} else {

AUC[p] <- (WilcoxonCountsTable(RocCountsTableSimPop))$Az

break

}

}

}

Var <- var(AUC)

stdAUC <- sqrt(Var)

cat("seed = ", seed, "OrigAUC = ", AUC\_org, "meanAUC = ", mean(AUC), "stdAUC = ", stdAUC, "\n")

The approach is to fit the counts data, lines 18-19, to the binormal model, lines 23-24, and use the parameter values to simulate new datasets, lines 36 - 51. For each simulated dataset, lines 38-39, the ratings are fitted by RocfitR, line 41, to get the fitted AUC. P is the total number of population samples, set to 2000, and the corresponding index is p. After all is done one calculates the standard deviation of the values. Sourcing the code yields:

### Online Appendix 7.D.2: Code output

> source(...)

seed = 2 , FOM = Az , P = 2000

Calibrated simulator values: a, b, zetas:

1.32 0.6075 0.007675 0.8963 1.516 2.397

seed = 2

OrigAUC = 0.8705

meanAUC = 0.8682

stdAUC = 0.04055

Notice that all three methods, the bootstrap, the jackknife and population sampling using a calibrated simulator, yield about the same estimate for standard deviation of AUC, about 0.04.

### Online Appendix 7.D.3: Code Listing

SimulateRocCountsTable <- function(K,mu,sigma,zeta)

{

z1 <- rnorm(K[1])

z2 <- rnorm(K[2], mean = mu, sd = sigma)

zeta <- c(-Inf,zeta,Inf)

for (k in 1:K[1]) {

for (b in 1:(length(zeta)-1)) {

if ((z1[k] > zeta[b]) && (z1[k] <= zeta[b+1])) {

z1[k] <- b

break

}

}

}

for (k in 1:K[2]) {

for (b in 1:(length(zeta)-1)) {

if ((z2[k] > zeta[b]) && (z2[k] <= zeta[b+1])) {

z2[k] <- b

break

}

}

}

RocCountsTable = array(dim = c(2,length(zeta)-2+1))

RocCountsTable[1,1:length(table(z1))] <- table(z1)

RocCountsTable[2,1:length(table(z2))] <- table(z2)

return(RocCountsTable)

}

The function SimulateRocCountsTable takes three arguments and returns an ROC counts table. The first argument is K, which is the  vector, which tells the simulator code how many non-diseased and diseased cases are desired. The next three arguments are mu, sigma and zeta, which stand for  (*R* is the number of ratings bins). Line 4 realizes K[1] samples from a unit normal distribution, corresponding to the non-diseased cases, and saves the values to the array z1. Line 5 realizes K[2] samples from a normal distribution with mean and standard deviation , corresponding to the diseased cases, and saves the values to the array z2. Line 7 performs the infinity padding of the  vector, which defines the vector with length *R*+1. Lines 9-16 implement the binning rules for the non-diseased cases, and lines 17-24 is the corresponding implementation for the diseased cases. Line 26 allocates the variable RocCountsTable which will hold the ROC counts table and the next line uses the table function to convert the array z1 with K[1] values, each representing the rating of a non-diseased case, to a frequency table with *R*-1 values which sum to K[1]. Line 28 does the same for the diseased cases and the function returns the variable RocCountsTable.

# Online Appendix 7.E: Comparison of different methods of estimating variability

Open the source code file main4EstimatesSd.R.

### Online Appendix 7.E.1: Code Listing

rm( list = ls()) # Main4EstimatesSd.R

source('GenerateCaseSamples.R')

source('VarPopSampling.R')

source('VarBootstrap.R')

source('VarJack.R')

source('Wilcoxon.R')

source("VarDeLong.R")

FinalParameters <- c(1.320455, 0.6074974, 0.007676989, 0.8962713, 1.515645, 2.396711)

a <- FinalParameters[1];b <- FinalParameters[2];zetas <- FinalParameters[3:length((FinalParameters))]

mu <- a/b;sigma <- 1/b;K <- c(600, 500)#;mu <- 1.5;sigma <- 1.3

seed <- 1;cat("K1 = ", K[1], ", K2 = ", K[2], ", mu = ", mu, ", sigma = ", sigma, "\n")

P <- 2000;B <- 2000;P1 <- 20

set.seed( seed );VPS<- array(dim = P1); {for (p in 1 : P1) VPS[p] <- VarPopSampling(K, mu, sigma, zetas, P)}

set.seed( seed );VBS<- array(dim = P1); {for (p in 1 : P1) VBS[p] <- VarBootstrap(K, mu, sigma, zetas, B)}

set.seed( seed );VJK<- array(dim = P1); {for (p in 1 : P1) VJK[p] <- VarJack(K, mu, sigma, zetas)}

set.seed( seed );VDL <- array(dim = P1); {for (p in 1 : P1) VDL[p] <- VarDeLong(K, mu, sigma, zetas)}

cat("Mean Sd Pop Sampling = ", mean(sqrt(VPS)),"\n")

cat("Mean Sd Boot Sampling = ", mean(sqrt(VBS)),"\n")

cat("Mean Sd Jack Sampling = ", mean(sqrt(VJK)),"\n")

cat("Mean Sd DeLong = ", mean(sqrt(VDL)),"\n")

The code is shorter as the different sampling methods have been put into functions. Sourcing this yields (the numbers of cases has been inflated by a factor of 10 for the second run):

### Online Appendix 7.E.2: Code Output

> source(...)

seed = 1 K1 = 60 , K2 = 50 , mu = 2.174 , sigma = 1.646

Mean Sd Pop Sampling = 0.03595

Mean Sd Boot Sampling = 0.03656

Mean Sd Jack Sampling = 0.03349

Mean Sd DeLong = 0.03332

> source(...)

seed = 1 K1 = 600 , K2 = 500 , mu = 2.174 , sigma = 1.646

Mean Sd Pop Sampling = 0.01133

Mean Sd Boot Sampling = 0.01099

Mean Sd Jack Sampling = 0.01131

Mean Sd DeLong = 0.0113

Notice that the methods agree with each other, and the agreement improves as one increases the sample size. Also, the standard deviation scales as the inverse square root of the total number of cases.

# Online Appendix 7.F: Empirical vs. fitted curves and bin variability

This relates to book Section 7.10 and 7.11. Open the file mainEmpVsFit.R.

### Online Appendix 7.F.1: Code Listing

rm( list = ls())#mainEmpVsFit.R

library(RJafroc);library(ggplot2)

seed <- 10;set.seed(seed)

mu <- 2; sigma <- 1.5

cat("Population AUC = ",

pnorm(mu/sqrt(1+sigma^2)), "\n")

K1 <- 500; K2 <- 500

fp <- rnorm(K1);tp <- rnorm(K2, mu, sigma)

zetas <- c(-Inf, 1.5, 2, 2.5, 3, 4, Inf)

fp1 <- as.numeric(cut(fp, zetas))

tp1 <- as.numeric(cut(tp, zetas))

rocData1 <- Df2RJafrocDataset(fp1, tp1)

plotEmp1 <- PlotEmpiricalOperatingCharacteristics(

rocData1, 1, 1)

print(plotEmp1$Plot)

empAuc1 <- UtilFigureOfMerit(

rocData1, FOM = "Wilcoxon")

cat("Emp. AUC bunched data = ", empAuc1, "\n")

Fit1 <- FitCbmRoc(rocData1)

print(Fit1$fittedPlot)

cat("CBM AUC, bunched data =" , Fit1$AUC,"\n")

zetas <- c(-Inf, -0.5, 0, 1, 1.5, 2, Inf)

fp2 <- as.numeric(cut(fp, zetas))

tp2 <- as.numeric(cut(tp, zetas))

rocData2 <- Df2RJafrocDataset(fp2, tp2)

plotEmp2 <- PlotEmpiricalOperatingCharacteristics(

rocData2, 1, 1)

print(plotEmp2$Plot)

empAuc2 <- UtilFigureOfMerit(

rocData2, FOM =

"Wilcoxon")

cat("Emp. AUC well-spaced = ", empAuc2, "\n")

Fit2 <- FitCbmRoc(rocData2)

print(Fit2$fittedPlot)

cat("CBM AUC, well-spaced data =" , Fit2$AUC,"\n")

Sourcing this code yields book Figures 7.1.

Open the file mainBinVariability.R.

### Online Appendix 7.F.2: Code Listing

rm( list = ls())#mainBinVariability.R

library(caTools);library(ggplot2);source("rocY.R")

mu <- 2;sigma <- 1.5 # experiment with other values

a <- mu/sigma; b <- 1/sigma # a and b parameters

cat("true AUC = ", pnorm(mu/sqrt(1+sigma^2)), "\n")

x <- seq(0.0, 1, 0.01)

zeta <- c(3, 2.5, 2)

FPF <- pnorm(-zeta);FPF <- c(0,FPF,1)

TPF <- pnorm((mu-zeta)/sigma);TPF <- c(0,TPF,1)

pointsData <- data.frame(FPF = FPF, TPF = TPF)

AUC <- trapz(FPF,TPF)

cat("empirical AUC, sparse points = ", AUC, "\n")

rocPlot1 <- ggplot(mapping = aes(x = FPF, y = TPF)) +

geom\_line(data = pointsData) +

geom\_point(data = pointsData)

print(rocPlot1)

zeta <- seq(3, -2, -0.5)

FPF <- pnorm(-zeta);FPF <- c(0,FPF,1)

TPF <- pnorm((mu-zeta)/sigma);TPF <- c(0,TPF,1)

pointsData <- data.frame(FPF = FPF, TPF = TPF)

AUC <- trapz(FPF,TPF)

cat("empirical AUC, dense point = ", AUC, "\n")

rocPlot2 <- ggplot(mapping = aes(x = FPF, y = TPF)) +

geom\_line(data = pointsData) +

geom\_point(data = pointsData)

print(rocPlot2)

Sourcing this code yields book Figures 7.1.

# Online Appendix 7.G: Modality effect

This relates to book Section 7.9. Open the file MainModalityEffect.R.

### Online Appendix 7.G.1: Code Listing

rm(list = ls()) #MainModalityEffect.R

library(RJafroc)

# dataset03 contains the Franken dataset

# dataset02 contains the VanDyke dataset

# dataRoc <- ReadDataFile(fileName, format = "MRMC") # this is not needed as dataset already exists in RJafroc

Foms <- UtilFigureOfMerit(dataset02, FOM = "Wilcoxon")

mean1 <- mean(Foms[1,]);mean2 <- mean(Foms[2,])

cat("reader-average FOM in modality 1 =", mean1, "\nreader-average FOM in modality 2 =", mean2,

"\neffect size, i.e., fom modality 1 minus modality 2 =", mean1-mean2, "\n")

### Online Appendix 7.E.2: Code Output

> source(...)

reader-average FOM in modality 1 = 0.897037

reader-average FOM in modality 2 = 0.9408374

effect size, i.e., fom modality 1 minus modality 2 = -0.04380032

# References

1. Burgess AE. Comparison of receiver operating characteristic and forced choice observer performance measurement methods. *Med. Phys.* 1995;22(5):643-655.