

Online Supplementary Material

Online Appendix 7.A: The bootstrap method

Online Appendix 7.B: The jackknife method

Online Appendix 7.C: A calibrated simulator

Online Appendix 7.D: Comparison of different methods of estimating variability

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)
} 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, " meanAUC = ", mean(AUC), " stdAUC = ", stdAUC, "\n")
```

Line 10 defines the **FOM** to be used to calculate the figure of merit. Currently the binormal model based AUC is selected, but if the commenting were reversed, empirical AUC would be selected. Line 12 sets the number of bootstraps **B** to 200 and for ease of explanation the **seed** variable is set to 1 (in real calculations one should set it 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 Table 7.1, i.e., the observed data. Line 18 uses the **R** function **sum()** to add the numbers in the first and second rows of **RocCountsTable** to obtain the \vec{K}

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 28 (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 used the function **RocfitR()** to determine the fitted area under the ROC curve, whose value, 0.87045188, is displayed in the **Environment** panel. To summarize, lines 1 – 25 calculate the binormal model AUC for the counts data in Table 7.1. 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 28-29. They use the **rep()** function of **R**. For example, **rep(1, 5)** repeats 1 five times, and the function also works on arrays. If one highlights the right hand side of line 28, carefully, and clicks on **Run**, one sees:

Online Appendix 7.A.2: Code Snippet

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, etc., essentially a linearized form of the ratings for non-diseased cases in Table 7.1. Using a similar method, line 29 linearizes the counts data for diseased cases. Line 30 defines an array **AUC**, of length 200, to hold the bootstrapped AUC values. Click **Next** enough times to advance the code pointer to line 31.

Now comes the interesting part. The **for**-loop beginning at line 31 is executed 200 times, once per bootstrap. Line 33 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 34. Highlight the left hand side of line 33 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 Table 7.1, so this variable is ready to receive data. Click **Next** enough times to advance the code pointer to line 41. On line 34 highlight the left hand side and click **Run**.

Online Appendix 7.A.4: Code Snippet

```
Browse[2]> ceiling( runif( K[ 1 ] ) * K[ 1 ] )
 [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 29
36 30
[34] 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
Browse[2]>
```

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), i.e., 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[60] = 5:***

Online Appendix 7.A.5: Code Snippet

```
Browse[2]> z1[1]  
[1] 1  
Browse[2]> z1[31]
```

```
[1] 2
Browse[2]> z1[60]
[1] 5
```

Line 36 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 the previously defined variable **RocCountsTable_bs**. Carefully highlight the left hand side of line 37 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 39 and 40 perform similar operations on diseased cases. Highlight **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 42 runs **RocfitR()** on the current bootstrapped dataset. Line 46 saves the current value of AUC to the array **AUC[b]**.

Now here comes a complication. At line 32 there is a "*forever or infinite while loop*", **while(1)**, which keeps looping 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 47 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 47 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 50, 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.

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$Az == -1) stop("RocfitR did not converge, original data")
} else {
  Az <- WilcoxonCountsTable(RocCountsTable)
}
Az = Az$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 no random number generator is 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 in 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 frequency table from the jackknifed data, named **RocCountsTable_jk**, line 43 replaces any **NAs** with 0 and line 45 uses the **RocfitR()** function to calculate $AUC_{(1)}$. 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 $AUC_{(2)}$.

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 $AUC_{(61)}$. On the next pass through the **for**-loop, **k = 62**, the 2nd diseased case is removed and the **RocfitR** function yields $AUC_{(62)}$, 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 52 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.

Online Appendix 7.C: A calibrated simulator

Online Appendix 7.C.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")
```

Online Appendix 7.C.2: 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 \vec{K} 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 $\mu, \sigma, \zeta_2, \zeta_3, \dots, \zeta_{R-1}$ (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 μ and standard deviation σ , corresponding to the diseased cases, and saves the values to the array **z2**. Line 7 performs the infinity padding of the $\zeta_2, \zeta_3, \dots, \zeta_{R-1}$ vector, which defines the vector ζ with length $R+1$ (see chapter xx, eqn. xx). 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.D: Comparison of different methods of estimating variability

Open the source code file **main4EstimatesSd.R** in the software folder corresponding to this chapter.

Online Appendix 7.D.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):

```
> source('~/book2/02 A ROC analysis/A7 Sources of variability in AUC/software/main4EstimatesSd.R')
K1 = 60 , K2 = 50 , mu = 2.173598 , sigma = 1.646098
Mean Sd Pop Sampling = 0.03594515
Mean Sd Boot Sampling = 0.03656486
Mean Sd Jack Sampling = 0.03349281
Mean Sd DeLong = 0.03331918
> source('~/book2/02 A ROC analysis/A7 Sources of variability in AUC/software/main4EstimatesSd.R')
K1 = 600 , K2 = 500 , mu = 2.173598 , sigma = 1.646098
Mean Sd Pop Sampling = 0.01132805
Mean Sd Boot Sampling = 0.01099246
Mean Sd Jack Sampling = 0.01130954
Mean Sd DeLong = 0.01130374
```

Notice that the methods agree with each other, and the agreement improves as one increases the sample size.