Chapter 6: Online Appendices

Table of contents

Online Appendix 6.A: Equivalence of 2 and 4 parameter models

Here is the demonstration to show that the 4-parameter model can be reduced to the two-parameter model in Eqn. (6.2). Open the file **software.Rproj** corresponding to this chapter and under the **Files** tab (lower right quadrant window in **RStudio**), find the **R**-code file named **Main4ParameterVs2Parameter.R** and **Source** it. You should see two identical ROC plots, the only way to tell is to use the arrows under the **Plots** menu to go back and forth between them; one plot is labeled **TPF** vs. **FPF**, and the other is labeled **TPF1** vs. **FPF1**, and they are identical. The code listing follows:

Online Appendix 6.A.1: Code listing

```
# Main4ParameterVs2Parameter.R
rm(list = ls())
source("OpPtsFromZsamples.R")
seed <- 1;set.seed(seed)

# define 4-parameter binormal model
Mu1 <- 3.1;Sigma1 <- 2.3
Mu2 <- 4.7;Sigma2 <- 4
K1 <- 10;K2 <- 12

# get K1/K2 samples from the normal distributions
z1 <- rnorm(K1,mean = Mu1, sd = Sigma1)
z2 <- rnorm(K2,mean = Mu2, sd = Sigma2)

OP <- OpPtsFromZsamples ( z1, z2 )
FPF <- OP$FPF
TPF <- OP$TPF

plot(FPF, TPF)

seed <- 1;set.seed(seed) # if you do not reset the seed, the data will be different

# define equivalent 2-parameter binormal model
mu <- (Mu2-Mu1)/Sigma1
sigma <- Sigma2 / Sigma1

# get K1/K2 samples from the normal distributions
z1 <- rnorm(K1) # mu = 0 and sd = 1 is implicit
z2 <- rnorm(K2,mean = mu, sd = sigma)

OP <- OpPtsFromZsamples ( z1, z2 )
FPF1 <- OP$FPF
TPF1 <- OP$TPF

plot(FPF1, TPF1)
```

Line 2 is the cleanup function and line 3 sources the **OpPtsFromZsamples()** function. Line 4 sets the seed to unity to ensure repeatable "random" samples. Lines 6-9 define the 4-parameter model and the numbers of non-diseased and diseased cases. Line 11-13 realizes **K1** samples from the non-diseased and **K2** from the diseased distributions and saves them in variables **z1** and **z2**. Lines 15-17 uses the function **OpPtsFromZsamples(z1,z2)** to calculate the corresponding operating points and these are extracted from the list variable **OP** (for operating point) to **FPF** and **TPF**. This function will be explained shortly, but for now let us move on. Line 19 plots the data points defined by **FPF** and **TPF**. Line 21 resets the seed to 1 (since we wish to recreate the same random samples we got so far). Lines 24-25 define the equivalent two-parameter model, and line 28-29 obtains **K1** and **K2** samples from the $N(0,1)$ non-diseased and $N(\mu,\sigma^2)$ diseased distributions, respectively, and saves them to variables **z11** and **z22**. Lines 31-33 converts these to operating points, extracted to arrays **FPF1** and **TPF1**, and line 34 plots them. Both plots can be viewed in the **Plots** window (the appropriate arrow key becomes active when there are multiple plots hiding behind each other). To delete or clear all plots, click on the "broom" **Clear All** button. The plots window will go blank.

Online Appendix 6.A.2: Debugging your code

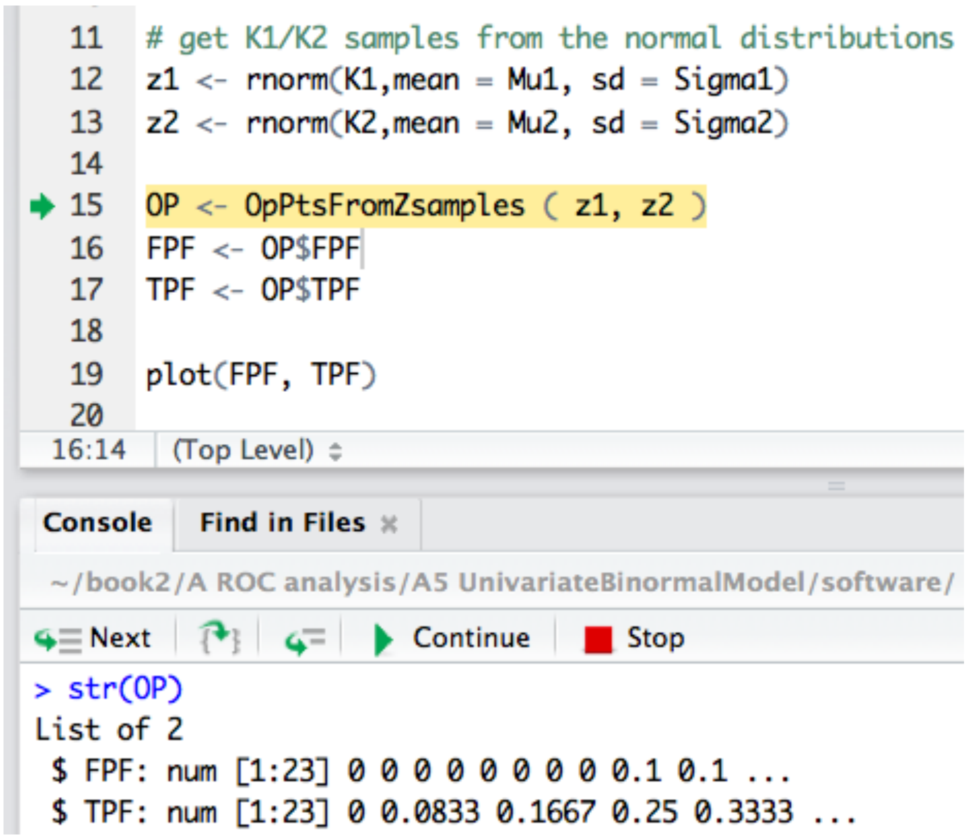
[The debug functions of **RStudio** have improved considerably over the period that I have been writing this book.] It is rare that one writes code that does exactly what one wishes it to do. I am going to show you simple methods of checking your code. The simplest is to position the cursor at line 1 and start clicking on the **Run** button (not **Source**). Start with deleting all plots (broom symbol). As each line is executed the cursor will move down one line. Look at the **environment** window when Line 2 is executed – it should read: “Environment is empty”. As it is executed each line is “echoed” in the **Console** window; you can examine the values of variables either by looking at the **Environment** window or by entering them at the **Console >** prompt. For example, from glancing at the **Environment** window, you can confirm that the 4 parameters indeed have the desired values, and you can view the first few values of the operating point defining arrays **FPF** and **TPF**. Continue clicking and observing the effect on the environment window. When you get to line 15 stop clicking; this line calls the **OpPtsFromZsamples()** function. You have two choices: (i) click on **Run** and simply accept the function as a black box or (ii) step into the function to see how it works. Let us exercise the first option for now: click on **Run**. The **environment** window should show a new variable **OP**, clicking on which shows that it is a list with two members **FPF** and **TPF**. The code snippets below show how to examine the contents of **OP** in the console window.

Online Appendix 6.A.2.1: Code snippet

```
> str(OP)
List of 2
 $ FPF: num [1:23] 0 0 0 0 0 0 0 0 0.1 0.1 ...
 $ TPF: num [1:23] 0 0.0833 0.1667 0.25 0.3333 ...
> OP$FPF
 [1] 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.1 0.2 0.2 0.2 0.3 0.4 0.5 0.6 0.7 0.7 0.8 0.9 1.0 1.0
> OP$TPF
 [1] 0.0000 0.0833 0.1667 0.2500 0.3333 0.4167 0.5000 0.5833 0.5833 0.6667 0.6667 0.7500 0.8333
0.8333
[15] 0.8333 0.8333 0.8333 0.8333 0.9167 0.9167 0.9167 0.9167 1.0000
```

[**RStudio** tries to help out by making suggestions and completing parenthesis. Try the **tab** key to bring up relevant help.]

To exercise the second option, debugging the **OpPtsFromZsamples()** function, place a break-point at line 15. Simply click on the side gray panel to the left of the line number. A red dot should appear. Your screen should look like Fig. 6.A.1. Now, click on **Source**. Your screen should look like Fig. 6.A.2.

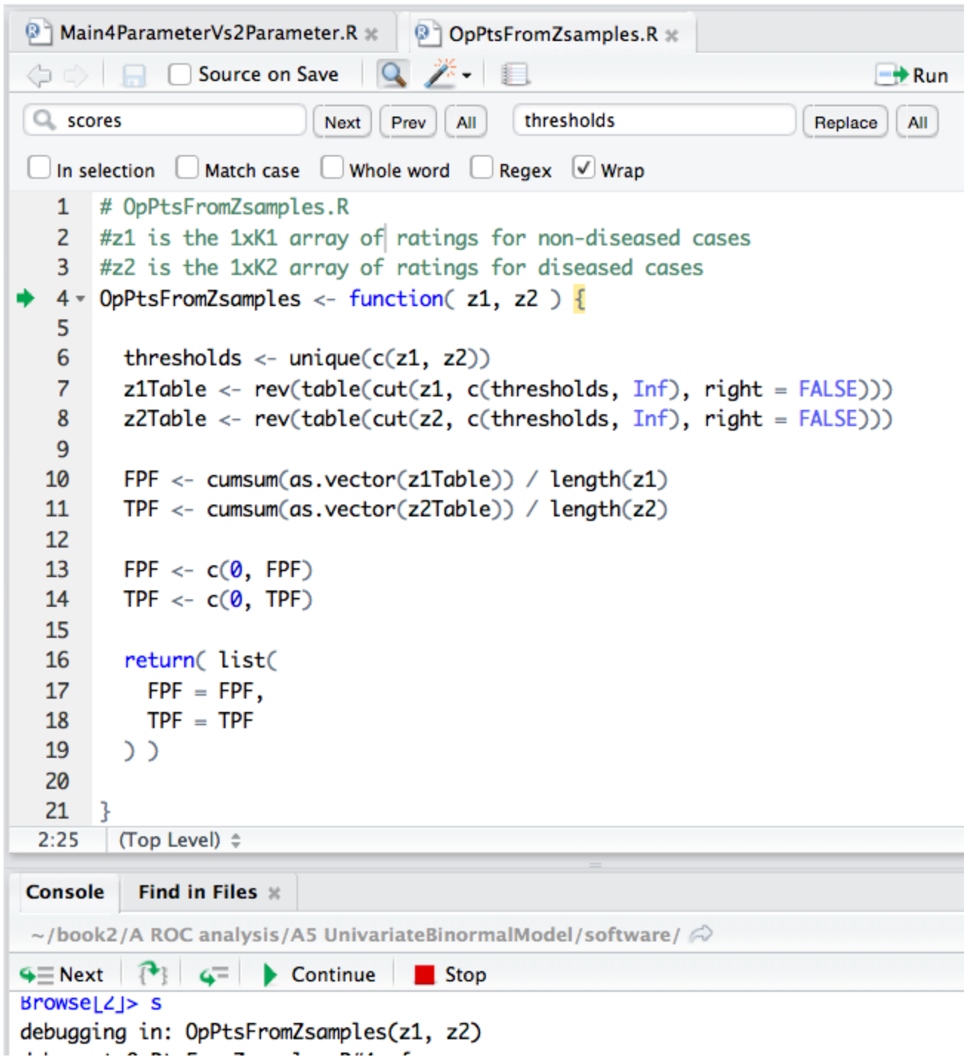


```
# get K1/K2 samples from the normal distributions
z1 <- rnorm(K1,mean = Mu1, sd = Sigma1)
z2 <- rnorm(K2,mean = Mu2, sd = Sigma2)

OP <- OpPtsFromZsamples ( z1, z2 )
FPF <- OP$FPF
TPF <- OP$TPF

plot(FPF, TPF)

```

```
> str(OP)
List of 2
 $ FPF: num [1:23] 0 0 0 0 0 0 0 0 0.1 0.1 ...
 $ TPF: num [1:23] 0 0.0833 0.1667 0.25 0.3333 ...
```

Fig. 6.A.1: Debugging R code. The green arrow is at the break point. It is arrived at by first inserting a break point by clicking on the portion of the window to the left of the line number and then clicking on source. Additional controls appear in the Console window.

The green arrow shows the next statement to be executed and new debug menu items have appeared in the Console window. Clicking on **`Next`** would simply execute the statement; as we have already seen the effect of choosing that option, click on the symbol to its immediate right, which could be described as an arrow stepping into code contained in braces {}. The next right button steps out of a function (or from inside a for-loop) to the calling statement or the outer layer, the button labeled **`Continue`** executes all code until the next break point is encountered, if any, and finally the **`Stop`** button gets you out of debug mode. Stepping into the function yields Fig. 6.A.2.



```
 1  # OpPtsFromZsamples.R
 2  #z1 is the 1xK1 array of ratings for non-diseased cases
 3  #z2 is the 1xK2 array of ratings for diseased cases
 4  OpPtsFromZsamples <- function( z1, z2 ) {
 5
 6    thresholds <- unique(c(z1, z2))
 7    z1Table <- rev(table(cut(z1, c(thresholds, Inf), right = FALSE)))
 8    z2Table <- rev(table(cut(z2, c(thresholds, Inf), right = FALSE)))
 9
10    FPF <- cumsum(as.vector(z1Table)) / length(z1)
11    TPF <- cumsum(as.vector(z2Table)) / length(z2)
12
13    FPF <- c(0, FPF)
14    TPF <- c(0, TPF)
15
16    return( list(
17      FPF = FPF,
18      TPF = TPF
19    ) )
20
21  }
2:25    (Top Level)
```

```
~/book2/A ROC analysis/A5 UnivariateBinormalModel/software/
Browse[2]> s
debugging in: OpPtsFromZsamples(z1, z2)
```

Fig. 6.A.2: debugging inside a function.

Online Appendix 6.A.2.2: Debugging inside a function

We are already inside the function. To see how the function **`OpPtsFromZsamples(z1,z2)`** works, keep clicking on the **`Next`** button and use judicious console window commands. For example, line 6 concatenates the two arrays **`z1`** and **`z2`**, containing the z-samples of the non-diseased and the diseased cases, respectively, and applies the **`unique()`** function, which has the effect of discarding any duplicated values. Since these arrays were generated by floating point random number generators, it is highly unlikely that they will have any common values and this function has no effect; with binned data the situation is completely different; in either case, the results are assigned to the variable **`thresholds`**, which is the ζ vector. Lines 7 and 8 are complex constructs, but by applying the rule of starting from the innermost level and working your way out, you should be able to figure out that they have the effect of computing the ROC counts tables, separately for false positives and true positives. The necessary division implied by Eqn. (5.7) is implemented in lines 10 – 11. Finally, lines 13 – 14 adds the (0,0) trivial point to complete the arrays. The two arrays are returned as a **`list`** variable.

Online Appendix 6.A.2.3: Code listing

```
# OpPtsFromZsamples.R
#z1 is the 1xK1 array of ratings for non-diseased cases
#z2 is the 1xK2 array of ratings for diseased cases
OpPtsFromZsamples <- function( z1, z2 ) {

  thresholds <- unique(c(z1, z2))
  z1Table <- rev(table(cut(z1, c(thresholds, Inf), right = FALSE)))
  z2Table <- rev(table(cut(z2, c(thresholds, Inf), right = FALSE)))

  FPF <- cumsum(as.vector(z1Table)) / length(z1)
  TPF <- cumsum(as.vector(z2Table)) / length(z2)

  FPF <- c(0, FPF)
  TPF <- c(0, TPF)

  return( list(
    FPF = FPF,
    TPF = TPF
  ) )

}
```

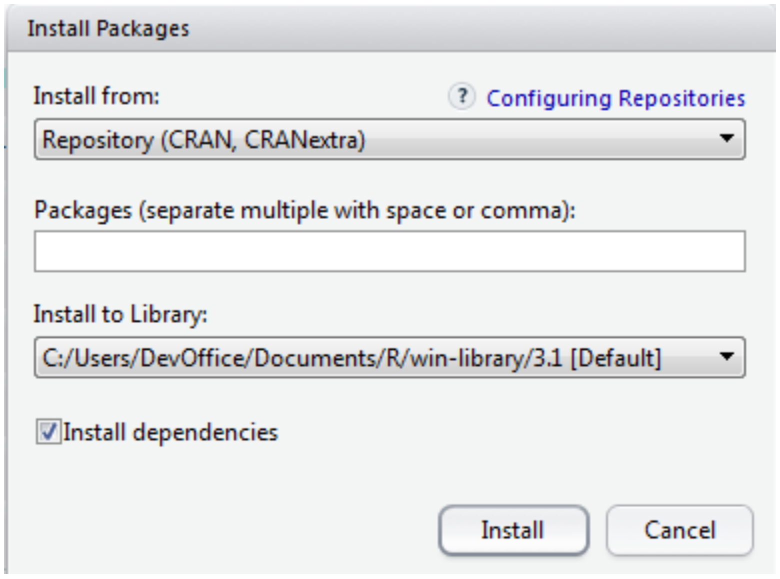
Online Appendix 6.A.3: R code for displaying the pdfs of the binormal model

By *pdf* we mean the probability density function, not the probability distribution function (*PDF*), which is another name for the cumulative distribution function (*CDF*). To minimize confusion, in this book we will stick to the *pdf / CDF* notation. Open the file named **`MainRocPdfs.R`**, a listing of which follows:

Online Appendix 6.A.3.1: Code listing

```
# MainRocPdfs.R
rm( list = ls())
require(ggplot2)

mu <- 1.5
sigma <- 1.5

z1 <- seq(-3, 3, by = 0.01)
z2 <- seq(-3, 7, by = 0.01)

pdf1 <- dnorm(z1)
pdf2 <- dnorm(z2, mu, sd = sigma)

df <- data.frame(z = c(z1, z2), pdf = c(pdf1, pdf2),
  truth = c(rep('non-diseased', length(pdf1)),
  rep('diseased', length(pdf2))))

rocPdfs <- ggplot(df, aes(x = z, y = pdf, color = truth)) +
  geom_line() +
  scale_colour_manual(values=c("red","green")) +
  theme(legend.title = element_blank(), legend.position = c(0.9, 0.9))

print(rocPdfs)
```

Explanation: Line 2 introduces the **`require()`** function with argument **`ggplot2`**. The former function is used to load additional software, termed **`packages`**, that are not part of the standard **`R`** installation. **`Packages`** are a way of extending the capabilities of **`R`**. It so happens that someone (Dr. Hadley Wickham to be specific[1]) has developed software to allow complex plotting capabilities in **`R`**. This software was then put into a standardized form, documented, etc., and uploaded as a **`package`** to the central **`R`** website or repository, thereby making it available to others. This is a tremendous advantage with the open source aspect of **`R`**. There are literally thousands of talented persons contributing code extending the capabilities of **`R`**. Now you have probably not added this particular package to your **`R`** installation. If so, click on **`Packages`** (lower right quadrant window in **`RStudio`**) and then on **`Install`**; a window titled **`Install Packages`** pops up, Fig. A3. In the blank line start typing in **`ggplot2`**: before you get too far, it shows you the correct choice; select it and then click on **`Install`**. Some download activity occurs and if all goes well, a success message appears. Just because you have downloaded the **`ggplot2`** package does not mean it is available to your code. It needs to be compiled (i.e., converted to instructions that **`R`** can understand). This is where is **`require()`** function comes in. If **`ggplot2`** is not available to the code you are about to execute, this function ensures that it is loaded, provided it exists on your computer; if it does not an error occurs, which means you need to download the package (you should not see this error if have already downloaded the **`ggplot2`** package).



Repository (CRAN, CRANextra)

Packages (separate multiple with space or comma):

Install to Library:

C:/Users/DevOffice/Documents/R/win-library/3.1 [Default]

[✓] Install dependencies

Fig. 6.A.3: installing a package

Lines 5 and 6 define the μ, σ parameters of the model. Lines 8 and 9 create two arrays, **`z1`** and **`z2`**, the first extending from -3 to 3 and the second from -5 to 7, the spacing is 0.01 in each case; the last argument was chosen to be sufficiently small so that the plots appear continuous. [Incidentally, these z's should not be confused with z-samples. They define points at which to evaluate the *pdf*s.] This is a common use of the **`seq()`** (for sequence) function to create an equally spaced array of numbers. I choose a larger range for **`z2`** because σ is generally greater than unity: if the plot does not come out right we can always adjust the range and re-run the code. Line 11 evaluates the pdf of $N(0,1)$ at the values specified by the **`z1`** array and saves the result to the array **`pdf1`**. Line 12 evaluates the pdf of $N(\mu,\sigma^2)$ at the values specified by the **`z2`** array and saves the result to the array **`pdf2`**. I could have written the right hand side of line 12 as **`dnorm(z2, mean = mu, sd = sigma)`**, or **`dnorm(z2, sd = sigma, mean = mu)`** or **`dnorm(z2, mu, sigma)`**, but **`dnorm(z2, sigma, mu)`** would not work as, without the explicit use of the keywords **`mean`** and **`sd`**, **`dnorm`** interprets the 2nd argument as the mean and the 3rd argument as the standard deviation. If you choose to write brief code, you had better keep track of the order of the arguments!

Lines 14-16 define a *data frame* variable **`df`**, which is defined in the official help file as a "*tightly coupled collection of variables which share many of the properties of matrices and of lists, used as the fundamental data structure by most of R's modeling software*". You can think of a data frame as a generalized array. In a conventional array all elements have to be of the same type, e.g., all have to be numbers or all have to be strings, but a *data frame* can have members of different types, as long as all members have the same length. The way it is constructed in this example, the data frame **`df`** consists of a member called **`z`** (for z-sample) which contains the two arrays **`z1`** and **`z2`** concatenated into one bigger array called **`z`**; the next member of this data frame is the concatenation of the two arrays **`pdf1`** and **`pdf2`** into one bigger array called **`pdf`**. The third member of the data frame is a categorical variable named **`truth`** with two values, **`non-diseased`** and **`diseased`**: the two **`rep()`** (for replicate) functions create two string arrays, the 1st containing the string **`'non-diseased'`**, repeated as many times as determined by the **`length`** of the **`pdf1`** array and the 2nd containing the string **`'diseased'`**, repeated as many times as determined by the **`length`** of the **`pdf2`** array. The **`c()`** (for concatenate) function combines these two arrays into one bigger array, and looking at the big picture, the function **`data.frame()`** combines the *z*-values, the *pdf* values, and the truth types, all into one big object of type data frame named **`df`**. Lines 18-21 creates the plot and saves it to the variable **`rocPdfs`** and line 23 **`prints`** it, i.e., shows it. If you **`source`** the code, you should see the following plot, Fig. 6.A.4.

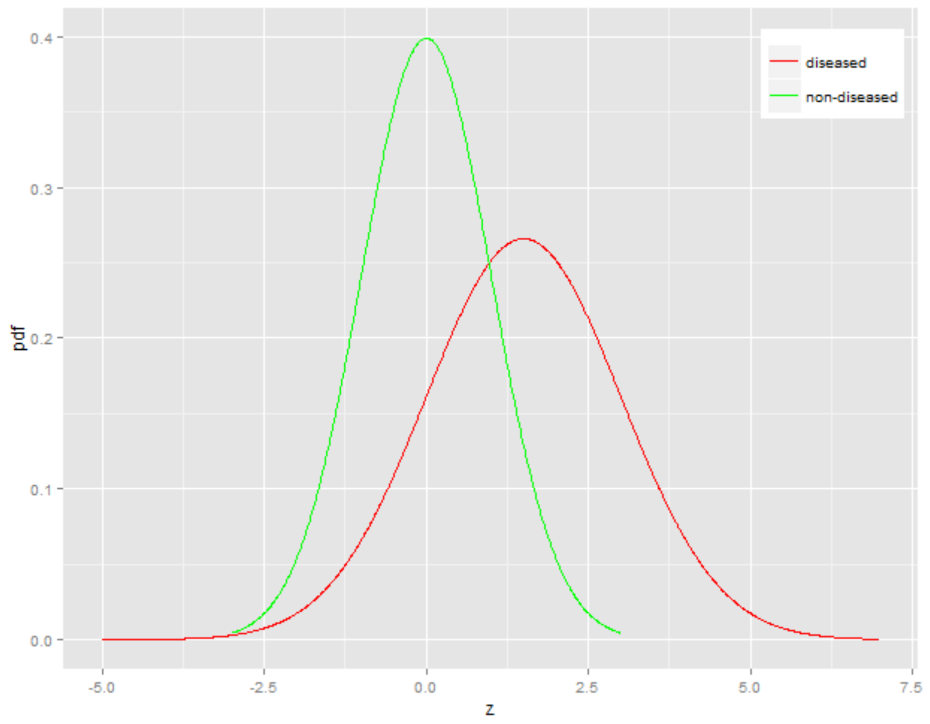


Fig. 6.A.4: The two probability density functions (pdfs) corresponding to the non-diseased cases (green line) and the diseased cases (red line). The plots are for $\mu = 1.5$ and $\sigma = 1.5$. The area under each individual pdf is unity; since the diseased pdf has variance greater than unity, its peak height, at z = 1.5 is lower than that of the non-diseased distribution.

Here is how to examine the structure of the data frame **df**:

Online Appendix 6.A.3.2: Code snippet

```
> str(df)
'data.frame':1602 obs. of  3 variables:
 $ z    : num  -3 -2.99 -2.98 -2.97 -2.96 -2.95 -2.94 -2.93 -2.92 -2.91 ...
 $ pdf  : num  0.00443 0.00457 0.0047 0.00485 0.00499 ...
 $ truth: Factor w/ 2 levels "diseased","non-diseased": 2 2 2 2 2 2 2 2 2 2 ...
> str(df$z)
 num [1:1602] -3 -2.99 -2.98 -2.97 -2.96 -2.95 -2.94 -2.93 -2.92 -2.91 ...
> str(df$pdf)
 num [1:1602] 0.00443 0.00457 0.0047 0.00485 0.00499 ...
> str(df$truth)
 Factor w/ 2 levels "diseased","non-diseased": 2 2 2 2 2 2 2 2 2 2 ...
```

Online Appendix 6.A.4: Code to generate Fig. 6.1

Again, this is more for practice with **R** than new science. Load the file **MainRocPdfsWithCutoffs.R** and **Source** it; you should see Fig. 10. The listing follows:

Online Appendix 6.A.4.1: Code listing

```
rm( list = ls())
require(ggplot2)
require(grid)

mu <- 1.5
sigma <- 1.5

z1 <- seq(-3, 4, by = 0.01)
z2 <- seq(-3, 6, by = 0.01)

pdf1 <- dnorm(z1)
pdf2 <- dnorm(z2, mu, sd = sigma)

df <- data.frame(z = c(z1, z2), pdfs = c(pdf1, pdf2),
  truth = c(rep('non-diseased', length(pdf1)), rep('diseased', length(pdf2))), stringsAsFactors =
FALSE)

cut_point <- data.frame(z = c(-2.0, -0.5, 1, 2.5))
```

```
rocPdfs <- ggplot(df, aes(x = z, y = pdfs, color = truth)) +
  geom_line() +
  scale_colour_manual(values=c("red","green")) +
  theme(legend.title = element_blank(), legend.position = c(0.9, 0.9),
        axis.title.x = element_text(hjust = 0.8, size = 20),
        axis.title.y = element_text(vjust = 0.8, hjust = 0.8, size = 20)) +
  geom_vline(data = cut_point, aes(xintercept = z), linetype = 2)

for (i in 1 : length(cut_point$z)){
  rocPdfs <- rocPdfs +
    annotation_custom(grob = textGrob(bquote(zeta[.(i)])),
                      xmin = cut_point$z[i], xmax = cut_point$z[i],
                      ymin = -0.05, ymax = -0.05)
}

gt <- ggplot_gtable(ggplot_build(rocPdfs))
gt$layout$clip[gt$layout$name == "panel"] <- "off"
grid.draw(gt)
```

Online Appendix 6.B: Output of Eng website software

This is the complete output of the Eng website software for the dataset in Table 6.1

```
JROCFIT:
Maximum likelihood estimation of a binormal ROC curve from
categorical rating data.

Java translation by John Eng, M.D.
The Russell H. Morgan Department of Radiology and
  Radiological Science
Johns Hopkins University, Baltimore, Maryland, USA
Version 1.0.2, March 2004

Original Fortran program ROCFIT by Charles Metz and colleagues
Department of Radiology, University of Chicago
January 1994
------------------------------------------------------------

DATA CHARACTERISTICS:
  Data collected in 5 categories with category 5 representing
    strongest evidence of positivity (e.g., that abnormality is
    present).
  Number of actually negative cases = 60
  Number of actually positive cases = 50

RESPONSE DATA:
  Category                     1     2     3     4     5
  Actually negative cases     30    19     8     2     1
  Actually positive cases      5     6     5    12    22

OBSERVED OPERATING POINTS:
  FPF:  0.0000  0.0167  0.0500  0.1833  0.5000  1.0000
  TPF:  0.0000  0.4400  0.6800  0.7800  0.9000  1.0000

INITIAL VALUES OF PARAMETERS:
  A = 1.3281
  B = 0.6291
  Z(K): -0.0000  0.9026  1.6452  2.1285
  LOGL = -143.8057

FINAL VALUES OF PARAMETERS:
  Procedure converges after 5 iterations.
  A = 1.3204
  B = 0.6075
  Z(K):  0.0077  0.8963  1.5157  2.3967
  LOGL = -141.4354

VARIANCE-COVARIANCE MATRIX:

  A     0.0656  0.0259  0.0150  0.0128  0.0070 -0.0135
  B     0.0259  0.0254  0.0053 -0.0022 -0.0141 -0.0458
  Z(1)  0.0150  0.0053  0.0260  0.0153  0.0109  0.0042
  Z(2)  0.0128 -0.0022  0.0153  0.0317  0.0276  0.0285
  Z(3)  0.0070 -0.0141  0.0109  0.0276  0.0539  0.0660
  Z(4) -0.0135 -0.0458  0.0042  0.0285  0.0660  0.1664

CORRELATION MATRIX:
  A     1.0000  0.6362  0.3626  0.2799  0.1179 -0.1293
  B     0.6362  1.0000  0.2046 -0.0767 -0.3814 -0.7048
  Z(1)  0.3626  0.2046  1.0000  0.5340  0.2906  0.0644
  Z(2)  0.2799 -0.0767  0.5340  1.0000  0.6678  0.3925
  Z(3)  0.1179 -0.3814  0.2906  0.6678  1.0000  0.6968
  Z(4) -0.1293 -0.7048  0.0644  0.3925  0.6968  1.0000

SUMMARY OF ROC CURVE:
  Area = 0.8705
  Std. Dev. (Area) = 0.0378

ESTIMATED BINORMAL ROC CURVE WITH ASYMMETRIC 95% CONFIDENCE
INTERVAL:
   FPF     TPF    95% Conf. Interv.
  0.005  0.4034  (0.1935, 0.6465)
  0.010  0.4629  (0.2563, 0.6804)
  0.020  0.5289  (0.3332, 0.7177)
  0.030  0.5705  (0.3847, 0.7417)
  0.040  0.6013  (0.4238, 0.7598)
  0.050  0.6259  (0.4553, 0.7746)
  0.060  0.6464  (0.4818, 0.7873)
  0.070  0.6641  (0.5046, 0.7984)
  0.080  0.6797  (0.5245, 0.8084)
  0.090  0.6935  (0.5422, 0.8174)
  0.100  0.7060  (0.5581, 0.8257)
  0.110  0.7174  (0.5726, 0.8334)
  0.120  0.7279  (0.5857, 0.8405)
  0.130  0.7377  (0.5978, 0.8472)
  0.140  0.7467  (0.6089, 0.8535)
  0.150  0.7552  (0.6193, 0.8595)
  0.200  0.7908  (0.6620, 0.8850)
  0.250  0.8188  (0.6945, 0.9054)
  0.300  0.8419  (0.7207, 0.9221)
  0.400  0.8784  (0.7615, 0.9477)
  0.500  0.9067  (0.7935, 0.9658)
  0.600  0.9298  (0.8209, 0.9788)
  0.700  0.9494  (0.8461, 0.9880)
  0.800  0.9665  (0.8713, 0.9943)
  0.900  0.9821  (0.9001, 0.9982)
  0.950  0.9898  (0.9195, 0.9994)

ESTIMATES OF EXPECTED OPERATING POINTS ON FITTED ROC CURVE:
      Expected           95% C.I. of        95% C.I. of
   Operating Point       Lower Bound        Upper Bound
  (  FPF ,   TPF )   (  FPF ,   TPF )   (  FPF ,   TPF )
  (0.0083, 0.4461)   (0.0007, 0.2672)   (0.0551, 0.6369)
  (0.0648, 0.6553)   (0.0244, 0.5490)   (0.1444, 0.7505)
  (0.1851, 0.7811)   (0.1065, 0.7136)   (0.2921, 0.8384)
  (0.4969, 0.9059)   (0.3731, 0.8695)   (0.6211, 0.9342)

WARNINGS AND ERROR MESSAGES:
Chi-square goodness of fit not calculated because
some expected cell frequencies are less than 5.
Chi-square goodness of fit not calculated because
some expected cell frequencies are less than 5.
```

Online Appendix 6.C: Maximizing log likelihood

A brute force way would be to try different values for all parameters of the model until the function reaches a maximum. Various techniques exist for *minimizing* a function, see for example Chapter 10 in my Bible, i.e., Numerical Recipes [7]. That presents no problem because minimizing $-LL\left(a,b,\vec{\zeta}\right)$ is equivalent to maximizing $LL\left(a,b,\vec{\zeta}\right)$. The technique used in the available software is termed the Newton-Raphson algorithm [7], which is an iterative algorithm, i.e., starting with an initial educated guess for all parameters, it will find a set of better estimates, and the procedure is repeated until the minimum does not change appreciably (termed *convergence*). Newton-Raphson requires analytic expressions for the partial first derivative of the $-LL$ function with respect to the individual model parameters. This was challenging in 1969, and Dorfman and Alf made one mistake, which was later corrected [17] (the referenced paper is a particularly readable description of the binormal model estimation process). Nowadays, using symbolic mathematical software such as **MAPLE** or **MATHEMATICA** and even **R** it is trivial to calculate the derivatives. However, considerable algebra is involved, and long unwieldy formulae result, which we do not propose to show. Instead, we will use one of the packages in **R**, namely **nlm**, which according to the official **R** help page is a non-linear minimization algorithm that "… carries out a minimization of the function using a Newton-type algorithm". It does not require specification of derivative functions (these are estimated numerically). The results of this simple implementation will be compared to that of the website, which is a Java implementation (version 1.0.2, March 2004) of the original Fortran program ROCFIT by Charles Metz and colleagues at the Department of Radiology, University of Chicago.

Open the project file **software.Rproj** and open the file "**MainRocfitR.R**". The listing follows:

Online Appendix 6.C.1: Code listing

```
rm(list = ls()) # mainRocFitR.R
library("numDeriv")

source("Transforms.R");source("LL.R")
source("RocOperatingPointsFromRatingsTable.R")
source("VarianceAz.R");source("ChisqrGoodnessOfFit.R")

a_min <<- 0.001;a_max <<- 6 # clamps on range of allowed values
b_min <<- 0.001;b_max <<- 6#do:

K1 <- c(30,19,8,2,1) # this is the observed data!
K2 <- c(5,6,5,12,22) # this is the observed data!
#K1 <- K1*10;K2 <- K2*10
# initial estimates of a and b parameters
ret <- RocOperatingPointsFromRatingsTable (K1, K2)
FPF <- ret$FPF; TPF <- ret$TPF

phiInvFpf <- qnorm(FPF); phiInvTpf <- qnorm(TPF)
fit <- lm(phiInvTpf~phiInvFpf) # straight line fit method of estimating a and b
a <- fit$coefficients[[1]] # these is the initial estimate of a
b <- fit$coefficients[[2]] # these is the initial estimate of b

# thresholds can be estimated by by applying inverse function to Eqn. xx and solving to zeta
zetaIniFpf <- -phiInvFpf # see Eqn. xx
zetaIniTpf <- (a - phiInvTpf)/b # see Eqn. xx
zetaIni <- (zetaIniFpf + zetaIniTpf)/2 # average the two estimates
zetaIni <- rev(zetaIni) # apply reverse order to correct the ordering of the cutoffs
zetaIniGuess <- seq(-b, a + 1, length.out = length(K1)-1) # to test stability of alg. to guess
choice

paramIni <- c(a, b, zetaIni)
#paramIni <- c(1, 1, zetaIniGuess) # to test stability of alg. to other choices

paramIniPrime <- ThetaPrime(paramIni)# use this method to test variation of -LL with parameters
LLvalIni <- LL(paramIniPrime, K1, K2) # use this method to test variation of -LL with parameters

retNlm <- nlm(LL, paramIniPrime, K1 = K1, K2 = K2, stepmax = 0.1) # this does the actual
minimization of -LL
paramFinal <- Theta(retNlm$estimate)

hess <- hessian(LL_theta, paramFinal, method="Richardson", K1 = K1, K2 = K2)
Cov <- solve(hess)

Az <- pnorm(a/sqrt(1+b^2))
StdAz <- sqrt(VarianceAz (a, b,Cov))

cat("initial parameters = ", paramIni, "\n")
cat("final parameters = ", paramFinal, "\n")
cat("-LL values, initial, final", LLvalIni, retNlm$minimum,"\n")
cat("covariance matrix = \n")
print(Cov)
cat("Az = ", Az, "StdAz = ", StdAz, "\n")

retChisqInitial <- ChisqrGoodnessOfFit(paramIni,K1,K2)
retChisqFinal <- ChisqrGoodnessOfFit(paramFinal,K1,K2)
if (!is.na(retChisqInitial)) cat("retChisqInitial = ", retChisqInitial$pVal,"\n")
if (!is.na(retChisqFinal)) cat("retChisqFinal = ", retChisqFinal$pVal, "\n")
```

The file **`Transforms.R`** implements two transformations, the function **`ThetaPrime()`** that transforms its argument $\vec{\boldsymbol{\theta}} \rightarrow \vec{\boldsymbol{\theta}}'$ and the function **`Theta()`** that transforms its argument $\vec{\boldsymbol{\theta}}' \rightarrow \vec{\boldsymbol{\theta}}$. These maintain positivity of the σ sigma parameter and correct ordering of the thresholds. Otherwise the minimization algorithm might supply negative values for σ or put, for example, the second threshold at a lower value than the first, which would cause the algorithm to stop with an error because taking the logarithm of a negative quantity is being attempted. The file **`LL.R`** implements the (negative) logarithm of the likelihood function. You should inspect it to confirm the implementation of Eqn. **Error! Reference source not found.**. The file **`OpPtsFromZsamplesTable.R`** implements the calculation of operating points from a ratings table, such as in the body of Table 1, by cumulating counts. Lines 8-9, which are part of the transforms, ensure that the searched space for *a*, *b* stay within reasonable bounds.

Lines 11-12 are the observed data, lifted from the body of Table 6.1; the arrays **`K1`** and **`K2`** are actually the vectors $\overrightarrow{K_1}$ and $\overrightarrow{K_2}$ defined in connection with Eqn. (5.1) and (5.2). Line 15-16 calculates **`FPF`** and **`TPF`** from the input ratings table. These are fitted by the least-squares method described earlier to obtain initial estimates of the *a* and *b* parameters, lines 18-21. To obtain an initial estimate of $\vec{\zeta}$, lines 24-25, one inverts Eqn. (6.19) and Eqn. (6.20):

$$\left.\begin{aligned} \Phi^{-1}(FPF) &= -\zeta \\ \Phi^{-1}(TPF) &= a - b\zeta \end{aligned}\right\} \quad . \qquad \textbf{(0.1)}$$

This yields two estimates of $\vec{\zeta}$, one from the FPF values and one from the TPF values.

$$\left.\begin{aligned} \zeta_{FPF} &= -\Phi^{-1}(FPF) \\ \zeta_{TPF} &= \frac{a - \Phi^{-1}(TPF)}{b} \end{aligned}\right\} \quad . \qquad \textbf{(0.2)}$$

The final step is to average the two estimates, Line 26. Line 27 reverses the ordering of the thresholds (because the steps described above yield the correct estimates, but in the reverse order). Line 30 combines the *a*, *b* and **`zetaIni`** threshold parameters into a single array called **`paramIni`**. Line 31, which is commented out, is a guess at the threshold values. If you uncomment it, then the program uses "bad" initial values. This type of de-tuning is useful to test the stability of the results to the starting values. If the results are sensitive to the initial values, then one might wish to choose an alternate minimization routine, such as simulated annealing, which is available in **R**. Line 33 implements the forward transformation of **`paramIni`** to values **`paramIniPrime`**, which can be positive or negative, with no special ordering requirement, and hence suitable as arguments for the minimization algorithm. Line 34 calculates the initial value of the (negative) log-likelihood function. Notice that it needs arguments, **`paramIniPrime`**, and the data vectors $\overrightarrow{K_1}$ and $\overrightarrow{K_2}$. Line 36 uses the **R** non-linear minimization function **`nlm()`** whose first argument is the initial estimates of the transformed parameters **`paramIniPrime`**, the second argument is the name of the likelihood function **`LL`**, no quotes needed, and the following arguments pass additional parameters needed by the likelihood function. The **`stepmax`** argument is set to unity; this determines the initial size of the steps the **`nlm`** algorithm will take in its search for a minimum (the default value of **`stepmax`** is 1000, which overwhelms the Φ function and causes a nuisance warning message to be generated, although the final results are unaffected – experiment with removing this argument). The result of the minimization is saved in the variable **`retNlm`**. You can print it out (highlight it and click on **`Run`**) or type it in the **`Console`** window and confirm that **`retNlm$estimate`** contains the final estimates of the transformed parameters. Line 37 applies the inverse transformation to convert from primed variables to regular variables (a, b, followed by the threshold vector without the infinity padding) and the result is saved in the vector **`paramFinal`**.

Line 39 evaluates the Hessian of the negative of the log likelihood function **`LL_theta`**, expressed as a function of $\boldsymbol{\theta}$ not $\boldsymbol{\theta}'$. It follows that the inverse of the Hessian matrix returned by the function **`hessian`** is the covariance matrix, whose diagonal elements are the variances of the parameters.

$$Cov(\boldsymbol{\theta}) = H^{-1} \quad . \qquad (0.3)$$

The function **`solve()`**, line 40, calculates the inverse of the Hessian matrix, using the **R** function **`solve()`**, and stores the inverse in the variable **`Cov`**. Line 42-43 calculates A_z and the square root of the variance of A_z (code to be shown below) and prints the initial parameters, the final parameters, the initial and final (negative) log-likelihood values (the numeric value should *decrease*, corresponding to *maximization* of log-likelihood), and the covariance matrix (whose diagonal elements are the corresponding variances). Line 52-55 calculate and print the chi-square statistic, the degrees of freedom and the p-value for the initial parameters and the final parameters. Table 2 summarizes the results (only variances shown) and compares them to those obtained by the Eng website program.

Online Appendix 6.D: Validating the fitting model

We proceed now to the **R** implementation of the Pearson goodness of fit statistic. Open the file **`ChisqrGoodnessOfFit.R`**, a listing of which follows:

```
# parameters are ordered as a,b,zeta1,..., zetaLast
ChisqrGoodnessOfFit <- function (parameters,K1,K2)
{
  R <- length(K1);L <- length(parameters)

  zeta <- c(-Inf, parameters[3:L],Inf)
  a <- parameters[1]
  b <- parameters[2]
  k1 <- sum(K1);k2 <- sum(K2) # total number of non-diseased and diseased cases

  C2 <- 0
  for (r in 1:R) {
    K1Exp <- k1*(pnorm(zeta[r+1]) - pnorm(zeta[r]));if (K1Exp < 5){C2 <- NA;break}
    K2Exp <- k2*(pnorm(b*zeta[r+1]-a) - pnorm(b*zeta[r]-a));if (K2Exp < 5){C2 <- NA;break}
    C2 <- C2 + (K1[r]-K1Exp)^2/K1Exp + (K2[r]-K2Exp)^2/K2Exp
  }

  if(!is.na(C2)) {
    df <- 2*R -(2+R-1) - 2
    pVal <- 1- pchisq(C2,df)
    return( list(
      C2 = C2,
      df = df,
      pVal = pVal
    ))
  } else return (NA)
}
```

Line 4 extracts the number of ratings bins (**`R`**) and the length (**`L`**) of the parameter array. Line 6 infinity-pads the thresholds array, lines 7 and 8 extracts the **`a`** and **`b`** parameters of the binormal model. Line 9 calculates the total number of non-diseased (**`k1`**) and diseased (**`k2`**) cases, respectively. Line 11-16 implements the calculation of the goodness of fit statistic C^2; line 13 calculates the expected number of counts in the non-diseased cell **`r`**, and line 14 calculates the corresponding number in the diseased cell **`r`**. Lines 13 and 14 also test if any of the expected number of counts is smaller than 5, and if so, the function returns **`NA`**. Otherwise the function calculates the degrees of freedom at line 19, the p-value at line 20, and returns, as a list variable, the value of C^2, the degrees of freedom and the p-value.

Online Appendix 6.E: Variance of Az

Open the file "**VarianceAz.R**" from the **File** menu of **RStudio**. The listing follows:

Online Appendix 6.E.1: Code listing

```
VarianceAz <- function (a, b,Cov)
{
  derivWrtA <- dnorm (a/sqrt(1+b^2))/sqrt(1+b^2)
  derivWrtB <- dnorm (a/sqrt(1+b^2))*(-a*b*(1+b^2)^(-1.5))
  VarAz <- (derivWrtA)^2*Cov[1,1]+(derivWrtB)^2*Cov[2,2] +
    2 * derivWrtA*derivWrtB*Cov[1,2]
  return (VarAz)
}
```

Note that we do not have the **rm(list = ls())** statement that begins any file whose name begins with **Main**. Functions, or subroutines, depending on your programming language preference, are meant to be called by other program units, i.e., other functions or a **source**-able file. The last thing we want to do inside a function is to delete all variables! The function named **VarianceAz()** takes three arguments, **a**, **b** and the **Cov** matrix and returns the variance of A_z. Study line 1 carefully as you will have to follow this template to define your own functions. A basic function that does nothing is:

```
DoNothingFunction  <- function ()
{
}
```

It is shown merely to show you the essential elements that every function must have. Coming back to our function that does something, lines 3 and 4 are straightforward implementations of the two required derivatives, and lines 5-6 implements Eqn. **Error! Reference source not found.**. The **return** statement literally returns the value of the variable contained within the parenthesis.

Functions are to be thought of as "black-boxes". The black box takes one or more input values and returns one or more output values. The nice thing about the black-box approach is that you cannot easily alter the values of the input parameters (supplied by the calling code) or any other variable in the calling code from within the called-function. Try it: insert the line **a <- 0** just before the **return** statement in the function **VarianceAz**. Save everything and **Source** the file **MainRocfitR.R**. Type **a** in the Console window and hit **return**. You should see **a = 1.328148**. In other words, setting **a <- 0** within the function did not alter its value in the calling code. *The only exception to this is if the variable is not declared in the argument list and is declared at the global level and you make a global level assignment inside the function.* I should not tell you this, because it can lead to bad programming habits, but you can declare a global variable as follows: type the following on line 10 of the code in **MainRocfitR.R**:

```
a <<- 0
```

Notice the double arrows: this is the construct in **R** to designate a *global* variable. Make necessary changes to the function argument list and the calling statements (basically take out **a**), insert a global level assignment of **a <<- 0** in the function, just before the **return** statement, save everything and then **Source** file **MainRocfitR.R**. Now you should see, in the **Global Environment** window, that **a** is zero. As you can see, you have to try quite hard inside a function to modify a variable in the calling code. Now that I have shown you how to do it, don't ever do it again. So undo all the changes you just made, and **Source** the file **MainRocfitR.R**. You should see the following output:

Online Appendix 6.E.2: Code output

```
> source('~/.active-rstudio-document')
initial parameters =  1.328148 0.6292443 0.03702537 0.8931309 1.506143 2.239335
```

```
final parameters =  1.320453 0.607497 0.007675259 0.8962713 1.515645 2.39671
-LL values, initial, final 141.6644 141.4354
covariance matrix =
             [,1]         [,2]        [,3]         [,4]         [,5]        [,6]
[1,]  0.065222451  0.025075693 0.014901301  0.012449852  0.007256201 -0.01067237
[2,]  0.025075693  0.024258552 0.005134077 -0.002321735 -0.014328108 -0.04286043
[3,]  0.014901301  0.005134077 0.025927199  0.015224065  0.010850407  0.00464582
[4,]  0.012449852 -0.002321735 0.015224065  0.031623191  0.027831802  0.02855593
[5,]  0.007256201 -0.014328108 0.010850407  0.027831802  0.055993439  0.06697530
[6,] -0.010672373 -0.042860433 0.004645820  0.028555929  0.066975300  0.15851205
Az =  0.8695184 StdAz =  0.0376075
[1] NA
[1] NA
```

The estimates using the Eng program are reproduced in the Appendix. You can confirm that your estimates are close to those of the Metz software ROCFIT, as implemented on the web by Eng.

Online Appendix 6.F: Transformations

The following transformations maintain positivity of sigma and proper ordering of thresholds. Let $\boldsymbol{\theta}$ denote the parameters in the model, i.e., it is a vector containing the elements $a,b,\zeta_2,\zeta_3,...,\zeta_{R-1}$. We perform the following transformation to $\boldsymbol{\theta}'$, termed the *forward transform*: $\vec{\boldsymbol{\theta}} \rightarrow \vec{\boldsymbol{\theta}'}$:

$$\left.\begin{aligned} \theta_1' &= \log\left(-\log\left(\frac{\theta_1 - a_{\min}}{a_{\max} - a_{\min}}\right)\right) \\ \theta_2' &= \log\left(-\log\left(\frac{\theta_2 - b_{\min}}{b_{\max} - b_{\min}}\right)\right) \\ \theta_3' &= \theta_3 \\ \theta_r' &= \log\left(\theta_r - \theta_{r-1}\right) \qquad r = 4,5,...,R+1 \end{aligned}\right\} . \tag{0.4}$$

The *inverse transform* ($\vec{\boldsymbol{\theta}'} \rightarrow \vec{\boldsymbol{\theta}}$) is:

$$\left.\begin{aligned} \theta_1 &= a_{\min} + \left(a_{\max} - a_{\min}\right)\exp\left(-\exp\left(\theta_1'\right)\right) \\ \theta_2 &= b_{\min} + \left(b_{\max} - b_{\min}\right)\exp\left(-\exp\left(\theta_2'\right)\right) \\ \theta_r &= \theta_{r-1} + \exp\left(\theta_r'\right) \qquad r = 4,5,...,R+1 \end{aligned}\right\} . \tag{0.5}$$

The variables subscripted with "min" indicate minimum values and those with "max" indicate maximum values. In the code they are set to 0.001 and 6, respectively.

6.6: References

1. Wickham H. *ggplot2: elegant graphics for data analysis.* Springer Science & Business Media; 2009.
2. Chang W. *R graphics cookbook.* " O'Reilly Media, Inc."; 2012.
3. Dorfman DD, Alf E. Maximum-Likelihood Estimation of Parameters of Signal-Detection Theory and Determination of Confidence Intervals - Rating-Method Data. *Journal of Mathematical Psychology.* 1969;6:487-496.
4. Green DM, Swets JA. *Signal Detection Theory and Psychophysics.* New York: John Wiley & Sons; 1966.
5. Hanley JA. The Robustness of the "Binormal" Assumptions Used in Fitting ROC Curves. *Med. Decis. Making.* 1988;8(3):197-203.

6. Dorfman DD, Berbaum KS, Metz CE, Lenth RV, Hanley JA, Abu Dagga H. Proper Receiving Operating Characteristic Analysis: The Bigamma model. *Acad. Radiol.* 1997;4(2):138-149.
7. Press WH, Teukolsky SA, Vetterling WT, Flannery BP. *Numerical Recipes: The Art of Scientific Computing.* 3 ed. Cambridge: Cambridge University Press; 2007.
8. McCullagh P, Nelder JA. *Generalized linear models.* Vol 37: CRC press; 1989.
9. Pearson K. On the Criterion that a Given System of Deviations from the Probable in the Case of a Correlated System of Variables is Such that it Can be Reasonably Supposed to have Arisen from Random Sampling. *Breakthroughs in Statistics*: Springer; 1992:11-28.
10. Larsen RJ, Marx ML. *An Introduction to Mathematical Statistics and Its Applications.* 3rd ed. Upper Saddle River, NJ: Prentice-Hall Inc; 2001.
11. Grey DR, Morgan BJT. Some Aspects of ROC Curve-Fitting: Normal and Logistic Models. *Journal of Mathematical Psychology.* 1972;9:128-1390.
12. Brandt S. *Data analysis: statistical and computational methods for scientists and engineers.* Springer Science & Business Media; 2012.
13. Thompson ML, Zucchini W. On the statistical analysis of ROC curves. *Stat Med.* 1989;8(10):1277-1290.
14. Pisano ED, Gatsonis C, Hendrick E, et al. Diagnostic performance of digital versus film mammography for breast-cancer screening. *N Engl J Med.* 2005;353(17):1-11.
15. Pisano ED, Gatsonis CA, Yaffe MJ, et al. American College of Radiology Imaging Network Digital Mammographic Imaging Screening Trial: Objectives and Methodology. *Radiology.* 2005;236(2):404-412.
16. Dorfman DD, Berbaum KS, Brandser EA. A contaminated binormal model for ROC data: Part I. Some interesting examples of binormal degeneracy. *Acad Radiol.* 2000;7(6):420-426.
17. Grey D, Morgan B. Some aspects of ROC curve-fitting: normal and logistic models. *Journal of Mathematical Psychology.* 1972;9(1):128-139.
18. Stein SK, Barcellos A. *Calculus and analytic geometry.* 5 ed: McGraw-Hill; 1992.