Data 621 Homework Assignment # 2

Group 2 - James Topor, Jeff Nieman, Scott Karr, Armenoush Aslanian-Persico

June 23, 2016

1.

Download the classification output data set (attached in Blackboard to the assignment).

```
library(knitr)
library(caret)
library(pROC)

hw2 <- read.csv("https://raw.githubusercontent.com/jtopor/CUNY-MSDA-621/master/HW-2/classification-outpu
```

2.

Use the table() function to get the raw confusion matrix for this scored dataset. Make sure you understand the output. In particular, do the rows represent the actual or predicted class? The columns?

```
# extract the three relevant columns
hw2.d <- hw2[,c(9,10,11)]

##########################################################
# extracted columns MUST be converted to matrix format

# actual = class
actual <- as.matrix(hw2.d[,1])

# predicted = scored.class
predicted <- as.matrix(hw2.d[,2])

########################################
# rows = actual
# cols = predicted

# left column = NEGATIVES
# right column = POSITIVES
########################################


# upper left = True Negative  | Upper right = False Positive
# lower left = False Negative | lower right = True Positive


table(actual, predicted)
```

```
##       predicted
## actual   0   1
##      0 119   5
##      1  30  27
```

The rows of the table represent the actual class while the columns represent the predicted class. This is because we supplied the *actual* class as the first argument to the **table()** function.

We can verify this is the case by converting the contingency table to a dataframe, which has the effect of separating the various counts from the underlying negative and positive indicators. As shown in the dataframe below, we have:

Column 1 Metric	Count	Column 2 Metric	Count
True Negatives (TN)	119	False Postives (FP)	5
False Negatives (FN)	30	True Positives (TP)	27

```
# display an interpretable version of the matrix
tmp <- data.frame(table(actual, predicted))
tmp
```

```
##   actual predicted Freq
## 1      0         0  119
## 2      1         0   30
## 3      0         1    5
## 4      1         1   27
```

Furthermore, it is important to note here that R's **table()** function is treating zeroes as ***negative*** outcomes while 1's are treated as ***positive*** outcomes. This fact will become relevant when we compare the output of the functions we build here against the output of R's **caret** package.

3.

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the accuracy of the predictions.

The R code for our "accuracy" function is provided below:

```
accuracy <- function(actual, predicted){

  # Equation to be modeled: (TP + TN) / (TP + FP + TN + FN)

  # derive confusion matrix cell values
  c.mat <- data.frame(table(actual, predicted))

  # extract all four confusion matrix values from the data frame
  TN <- as.numeric(as.character(c.mat[1,3]))
  FN <- as.numeric(as.character(c.mat[2,3]))
  FP <- as.numeric(as.character(c.mat[3,3]))
  TP <- as.numeric(as.character(c.mat[4,3]))

  # now calculate the required metric
```

```
  return( (TP + TN) / (TP + FP + TN + FN) )
}
```

4.

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the classification error rate of the predictions.

The R code for our classification error rate function is provided below:

```
classif.err.rate <- function(actual, predicted) {

  # Equation to be modeled: (FP + FN) / (TP + FP + TN + FN)

  # derive confusion matrix cell values
  c.mat <- data.frame(table(actual, predicted))

  # extract all four confusion matrix values from the data frame
  TN <- as.numeric(as.character(c.mat[1,3]))
  FN <- as.numeric(as.character(c.mat[2,3]))
  FP <- as.numeric(as.character(c.mat[3,3]))
  TP <- as.numeric(as.character(c.mat[4,3]))

  # now calculate the required metric
  return( (FP + FN) / (TP + FP + TN + FN) )
}
```

Now verify that you get an accuracy and an error rate that sums to one:

```
(acc.1 <- accuracy(actual, predicted) )
```

```
## [1] 0.8066298
```

```
(cer.1 <- classif.err.rate(actual, predicted) )
```

```
## [1] 0.1933702
```

```
acc.1 + cer.1
```

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

The accuracy and error rate do, in fact, sum to one.

5.

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the precision of the predictions.

```r
precision <- function(actual, predicted) {

  # Equation to be modeled: TP / (TP + FP)

  # derive confusion matrix cell values
  c.mat <- data.frame(table(actual, predicted))

  # extract all four confusion matrix values from the data frame
  TN <- as.numeric(as.character(c.mat[1,3]))
  FN <- as.numeric(as.character(c.mat[2,3]))
  FP <- as.numeric(as.character(c.mat[3,3]))
  TP <- as.numeric(as.character(c.mat[4,3]))

  # now calculate the required metric
  return( TP / (TP + FP) )
}
```

6.

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the sensitivity of the predictions. Sensitivity is also known as recall.

```r
sensitivity <- function(actual, predicted) {

  # Equation to be modeled: TP / (TP + FN)

  # derive confusion matrix cell values
  c.mat <- data.frame(table(actual, predicted))

  # extract all four confusion matrix values from the data frame
  TN <- as.numeric(as.character(c.mat[1,3]))
  FN <- as.numeric(as.character(c.mat[2,3]))
  FP <- as.numeric(as.character(c.mat[3,3]))
  TP <- as.numeric(as.character(c.mat[4,3]))

  # now calculate the required metric
  return( TP / (TP + FN) )
}
```

7.

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the specificity of the predictions.

```r
specificity <- function(actual, predicted) {

  # Equation to be modeled: TN / (TN + FP)
```

```
  # derive confusion matrix cell values
  c.mat <- data.frame(table(actual, predicted))

  # extract all four confusion matrix values from the data frame
  TN <- as.numeric(as.character(c.mat[1,3]))
  FN <- as.numeric(as.character(c.mat[2,3]))
  FP <- as.numeric(as.character(c.mat[3,3]))
  TP <- as.numeric(as.character(c.mat[4,3]))

  # now calculate the required metric
  return( TN / (TN + FP) )
}
```

8.

Write a function that takes the data set as a dataframe, with actual and predicted classifications identified, and returns the F1 score of the predictions.

```
F1.Score <- function(actual, predicted) {

  # Equation to be modeled: ( 2 * precision * sensitivity) / (precision + sensitivity)

  # now calculate the required metric
  return( ( 2 * precision(actual, predicted) * sensitivity(actual, predicted))
          / (precision(actual, predicted) + sensitivity(actual, predicted)) )
}
```

```
# Now test F1.score
F1.Score(actual, predicted)
```

9.

What are the bounds on the F1 score? Show that the F1 score will always be between 0 and 1. (Hint: If 0 < alpha < 1 and 0 < beta < 1 then (alpha x beta) < alpha)

The bounds of the F1 score can be assessed by evaluating the bounds of each of the individual component functions that comprise the F1 Score formula. Recall that the F1 Score formula is:

F1 Score = (2 * precision * sensitivity) / (precision + sensitivity)

We can therefore start by assessing the limits of the **precision** and **sensitivity** functions.

- $precision = TP/(TP + FP)$. It's obvious from that equation that precision is maximized when $TP = 1$ and $FP = 0$, yielding ($precision = 1$), and minimized when $TP = 0$ which yields ($precision = 0$), so it follows that:
$$0 <= precision <= 1$$
- $sensitivity = TP/(TP + FN)$. Once again we have a function that is maximized when $TP = 1$ (and $FN = 0$), which yields ($sensitivity = 1$). The function is minimized when $TP = 0$ which yields $sensivity = 0$, so it follows that:
$$0 < sensitivity < 1$$

Therefore, if we let $a = precision$ and $b = sensitivity$ we have:

- If $0 < a < 1$ and $0 < b < 1$, then $0 < ab < a$ and $0 < ab < b$
- If $0 < ab < a$ and $0 < ab < b$ then $0 < 2 * ab < a + b$ (by substitution)
- Therefore, $0 < 2 * ab/(a + b) < 1$ (using algebra (mulitplicative inverse))
- and $0 < 2 * precision * sensitivity/(precision + sensitivity) < 1$ (by substitution)

Therefore $0 < F1 < 1$ are the bounds of the F1 score range.

10.

Write a function to plot ROC and find AUC.

The function to plot ROC and calculate AUC is defined below.

Please note that the AUC calculation we use is implemented via an approximation algorithm based on a Riemann sum, where for each Y-value the corresponding X distance used to define the base of the rectangle is calculated as the difference between the current and previous $(1 - Specificity)$ values.

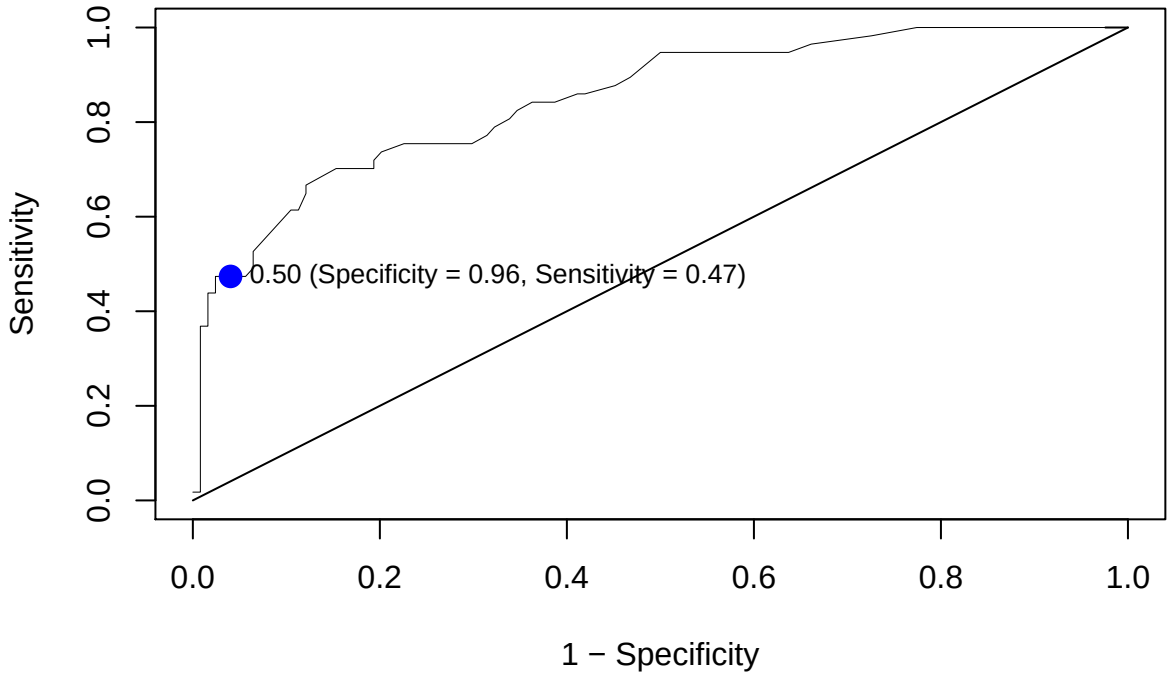
```
rocplot <- function(hw2.d){

  # Function generates roc plot and aoc calculation
  # accepts data frame
  # returns list of plot output

# Generate a vector of values beteen 0 and 1 by .01
  thresh <- seq(0.01, 1, by = .01)

 # initialize a vector to be used for accumulating the sensitivities and specificities

  speci.i <- rep(0.0, 100)
  sensi.i <- rep(0.0, 100)

  # loop through all 100 threshold values
  for(k in 1:100) {

    # initialize vector for storing thresholded scores
    thresh.pred <- rep(0, nrow(hw2.d))

    # use threshhold to set predicted value to 1 if scored.probability > threshhold
    thresh.pred[hw2.d$scored.probability > thresh[k] ] <- 1

    # now run specificity and sensitivity and store results
    speci.i[k] <- 1 - specificity(hw2.d$class, as.matrix(thresh.pred) )
    sensi.i[k] <- sensitivity(hw2.d$class, as.matrix(thresh.pred) )

    # record 0.50 spot for plotting
    if (thresh[k] == 0.5) {
      speci.MID <- speci.i[k]
      sensi.MID <- sensi.i[k]
    }
```

```
} # end for k

# now create ROC plot
par(mfrow = c(1,1))
p <- plot(sensi.i ~ speci.i, xlim=c(0,1), ylim=c(0,1), type = 'l', lty=1,lwd=0.5,
     xlab = "1 - Specificity", ylab = "Sensitivity")

# get max points for extending horizontal line to diagonal
xm <- max(speci.i, na.rm = TRUE)
ym <- max(sensi.i, na.rm = TRUE)

# plot horizontal line to diagonal
segments(x0 = xm, y0 = ym, x1 = 1, y1 = ym )

# add diagonal line
lines(x = c(0,1), y = c(0,1))

# add point for 0.50 location
points(speci.MID, sensi.MID, type = 'p', pch = 19, col = "blue", cex = 1.5)

t.speci <- as.character(round(1 - speci.MID, 2))
t.sensi <- as.character(round(sensi.MID, 2))

# add text label for 0.50 point in plot
p.label <- sprintf("0.50 (Specificity = %s, Sensitivity = %s)", t.speci, t.sensi )
text(speci.MID, sensi.MID, labels= p.label, pos = 4, cex = .8)

# ------------------------------------------------
# calcualte area under curve (auc)

speci.i[is.na(speci.i)] <- 0
sensi.i[is.na(sensi.i)] <- 0

auc <- 0
speci.prev <- 0

#reverse orders of vectors to enable more straigthforward area calculation
speci.i <- rev(speci.i)
sensi.i <- rev(sensi.i)

# aggregate incremental distance under curve of kth point
# multiplied by the incremental distance between the current and previous speci.i values
for(k in 1:100) {
  auc <- auc + ( sensi.i[k] * (speci.i[k] - speci.prev) )

  # update p-speci
  speci.prev <- speci.i[k]

} # end for

# now add final rectangle beyond max value of speci.i to ensure coverage of entire AUC
auc <- auc + (1 - max(speci.i))
```

```
  # return results in a list
  l <- list(p,auc)
  return(l)

} # end function
```

Now run the **rocplot()** function we've built:

```
roclist <- rocplot(hw2.d)
```



The AUC indicated by our custom ROC function is:

```
# display area under curve calculation
roclist[2]
```

```
## [[1]]
## [1] 0.8539898
```

11.

Use your created R functions and the provided classification output data set to produce all of the classification metrics discussed above.

```r
accuracy(hw2$class, hw2$scored.class)
```

```
## [1] 0.8066298
```

```r
classif.err.rate(hw2$class, hw2$scored.class)
```

```
## [1] 0.1933702
```

```r
precision(hw2$class, hw2$scored.class)
```

```
## [1] 0.84375
```

```r
sensitivity(hw2$class, hw2$scored.class)
```

```
## [1] 0.4736842
```

```r
specificity(hw2$class, hw2$scored.class)
```

```
## [1] 0.9596774
```

```r
F1.Score(hw2$class, hw2$scored.class)
```

```
## [1] 0.6067416
```

Metric	Value
Accuracy	0.8066
Classification Error Rate	0.1934
Precision	0.8438
Sensitivity	0.4737
Specificity	0.9597
F1 Score	0.6067

12.

Investigate the caret package. In particular, consider the functions confusionMatrix, sensitivity and specificity. Apply the functions to the data set. How do the results compare with your own functions?

```r
confusionMatrix(hw2$scored.class, hw2$class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
```

```
##          0 119  30
##          1   5  27
##
##                Accuracy : 0.8066
##                  95% CI : (0.7415, 0.8615)
##     No Information Rate : 0.6851
##     P-Value [Acc > NIR] : 0.0001712
##
##                   Kappa : 0.4916
##  Mcnemar's Test P-Value : 4.976e-05
##
##             Sensitivity : 0.9597
##             Specificity : 0.4737
##          Pos Pred Value : 0.7987
##          Neg Pred Value : 0.8438
##              Prevalence : 0.6851
##          Detection Rate : 0.6575
##    Detection Prevalence : 0.8232
##       Balanced Accuracy : 0.7167
##
##        'Positive' Class : 0
##
```

The table below summarizes the sensitivity, specificity, and accuracy metrics produced by our functions and those of the **caret** package:

Approach	Sensitivity	Specificity	Accuracy
Our Functions	0.4737	0.9597	0.8066
caret package	0.9597	0.4737	0.8066

As the table indicates, the accuracy metrics are identical. However, the sensitivity and specificity values provided by the **caret** package are switched relative to the sensitivity and specificity values generated by our functions. This is because the **caret** package treats zeroes as ***positive*** outcomes and 1's as ***negative*** outcomes, while the **table()** function we were required to use in building up our functions treats zeroes as ***negative*** outcomes and 1's as ***positive*** outcomes. Switching the meaning of the zeroes and 1's causes the values for specificity and sensitivity to flip.

13.

Investigate the pROC package. Use it to generate an ROC curve for the data set. How do the results compare with your own functions?

R's pROC package provides tools for visualizing, smoothing and comparing Receiver Operating Characteristic (ROC) curves. Within the package are specific function for calculating full and partial areas under the ROC curve and a method for comparing curves, the "roc.test" function. Confidence intervals can also be computed using the "ci" functions.

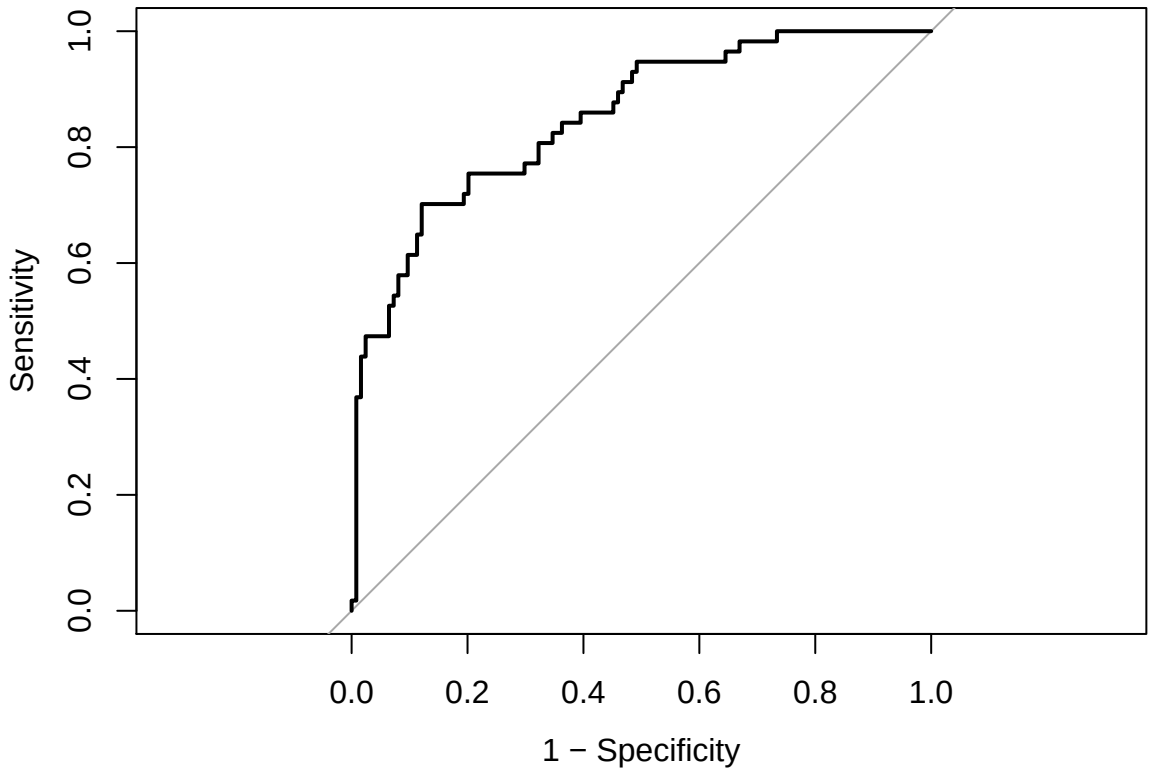
The key function provided by the pROC package is **roc()**, which builds an ROC "list" object containing two vectors (response, predictor) and can be plotted or tested against other ROC curves using the roc.test function. Finally, the **auc()** function is used to calculate the area under the ROC curve using a trapezoidal algorithm.

To mimic the behavior of the **rocplot()** function we constructed for **Question #10**, we've wrapped a function around calls to the pROC package's **roc()** and **auc()** functions:

```
roclist2 <- function(actual,predicted) {

  # Function generates roc plot and AUC calculation
  # accepts a response list and a predictor probablity list
  # returns list of plot output

  rocCurve <- roc(
      response=actual, predictor=predicted
  )

  r <- plot(rocCurve, legacy.axes = TRUE)

  rocAuc <- auc(rocCurve)
  list(r,rocAuc)
}
```

Now test the roclist2() function:

```
x <- hw2.d$class
y <- hw2.d$scored.probability
roclist2(x,y)
```




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

```
##
## Call:
## roc.default(response = actual, predictor = predicted)
##
## Data: predicted in 124 controls (actual 0) < 57 cases (actual 1).
## Area under the curve: 0.8503
##
## [[2]]
## Area under the curve: 0.8503
```

The plot generated by the pROC package looks very similar to the plot we generated using the **rocplot()** function we created for **Question # 10**. Both plots show a similar dynamic: sensitivity rises drastically relative to $(1 - specificity)$ until approximately the point where the 0.50 probability threshold is surpassed. From that point onward sensitivity continues to increase but at a lower rate until flattening out completely.

The comparision of AUC between our custom function and the pROC package is as follows:

Approach	AUC
Custom Function	0.8539
pROC package	0.8503

As the table shows, the AUC values are nearly identical. The small difference may be accounted for by differences in the ways the two functions approximate the area under the curve. The pROC package makes use of the ***trapezoidal*** rule for approximating the AUC while the **rocplot()** function we've written here makes use of a Riemann / ***rectangular*** rule for approximating the AUC.