Calculating AUC: the area under a ROC Curve

November 22, 2016

by Bob Horton, Microsoft Senior Data Scientist

Receiver Operating Characteristic (ROC) curves are a popular way to visualize the tradeoffs between sensitivitiy and specificity in a binary classifier. In an earlier post, I described a simple "turtle's eye view" of these plots: a classifier is used to sort cases in order from most to least likely to be positive, and a Logo-like turtle marches along this string of cases. The turtle considers all the cases it has passed as having tested positive. Depending on their actual class they are either false positives (FP) or true positives (TP); this is equivalent to adjusting a score threshold. When the turtle passes a TP it takes a step upward on the y-axis, and when it passes a FP it takes a step rightward on the x-axis. The step sizes are inversely proportional to the number of actual positives (in the y-direction) or negatives (in the x-direction), so the path always ends at coordinates (1, 1). The result is a plot of true positive rate (TPR, or specificity) against false positive rate (FPR, or 1 – sensitivity), which is all an ROC curve is.

Computing the area under the curve is one way to summarize it in a single value; this metric is so common that if data scientists say "area under the curve" or "AUC", you can generally assume they mean an ROC curve unless otherwise specified.

Probably the most straightforward and intuitive metric for classifier performance is accuracy. Unfortunately, there are circumstances where simple accuracy does not work well. For example, with a disease that only affects 1 in a million people a completely bogus screening test that always reports "negative" will be 99.9999% accurate. Unlike accuracy, ROC curves are insensitive to class imbalance; the bogus screening test would have an AUC of 0.5, which is like not having a test at all.

In this post I'll work through the geometry exercise of computing the area, and develop a concise vectorized function that uses this approach. Then we'll look at another way of viewing AUC which leads to a probabilistic interpretation.

Yes, it really is the area under the curve

Let's start with a simple artificial data set:

```
category <- c(1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0)
prediction <- rev(seq_along(category))
prediction[9:10] <- mean(prediction[9:10])
```

Here the vector `prediction` holds ersatz scores; these normally would be assigned by a classifier, but here we've just assigned numbers so that the decreasing order of the scores matches the given order of the category labels. Scores 9 and 10, one representing a positive case and the other a negative case, are replaced by their average so that the data will contain ties without otherwise disturbing the order.

To plot an ROC curve, we'll need to compute the true positive and false positive rates. In the earlier article we did this using cumulative sums of positives (or negatives) along the sorted binary labels. But here we'll use the `pROC` package to make it official:

```
library(pROC)
roc_obj <- roc(category, prediction)
auc(roc_obj)
```

```
## Area under the curve: 0.825
```

```
roc_df <- data.frame(
  TPR=rev(roc_obj$sensitivities),
  FPR=rev(1 - roc_obj$specificities),
  labels=roc_obj$response,
  scores=roc_obj$predictor)
```

The `roc` function returns an object with plot methods and other conveniences, but for our purposes all we want from it is vectors of TPR and FPR values. TPR is the same as sensitivity, and FPR is 1 – specificity (see "confusion matrix" in Wikipedia). Unfortunately, the `roc` function reports these values sorted in the order of ascending score; we want to start in the lower left hand corner, so I reverse the order. According to the `auc` function from the pROC package, our simulated category and prediction data gives an AUC of 0.825; we'll compare other attempts at computing AUC to this value.

Plotting the approach

If the ROC curve were a perfect step function, we could find the area under it by adding a set of vertical bars with widths equal to the spaces between points on the FPR axis, and heights equal to the step height on the TPR axis. Since actual ROC curves can also include portions representing sets of values with tied scores which are not square steps, we need to adjust the area for these segments. In the figure below we use green bars to represent the areas under the steps. Adjustments for sets of tied values will be shown as blue rectangles; half the area of each of these blue rectagles is below a sloped segment of the curve.

The function for drawing polygons in base R takes vectors of x and y values; we'll start by defining a `rectangle` function that uses a simpler and more specialized syntax; it takes x and y coordinates for the lower left corner of the rectangle, and a height and width. It sets some default display options, and passes along any other parameters we might specify (like color) to the `polygon` function.

```
rectangle <- function(x, y, width, height, density=12, angle=-45, ...)
  polygon(c(x,x,x+width,x+width), c(y,y+height,y+height,y),
          density=density, angle=angle, ...)
```

The spaces between TPR (or FPR) values can be calculated by `diff`. Since this results in a vector one position shorter than the original data, we pad each difference vector with a zero at the end:

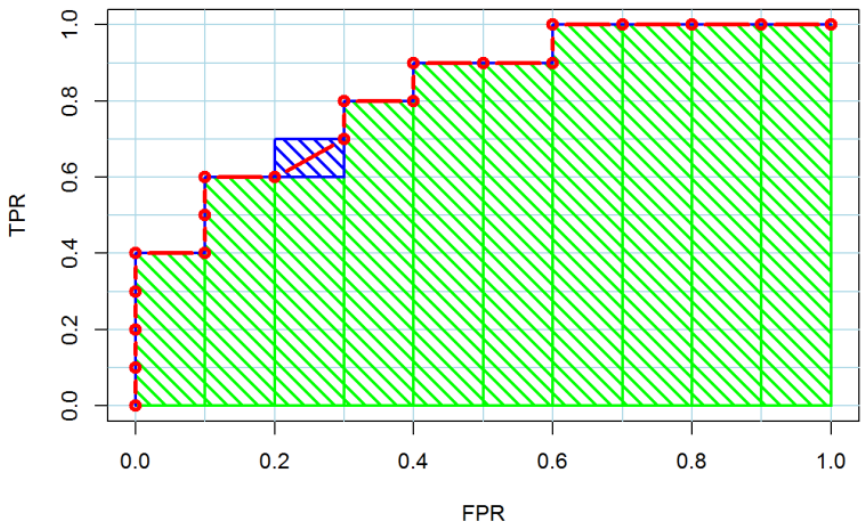
```
roc_df <- transform(roc_df,
  dFPR = c(diff(FPR), 0),
  dTPR = c(diff(TPR), 0))
```

For this figure, we'll draw the ROC curve last to place it on top of the other elements, so we start by drawing an empty graph (`type='n'`) spanning from 0 to 1 on each axis. Since the data set has exactly ten positive and ten negative cases, the TPR and FPR values will all be multiples of 1/10, and the points of the ROC curve will all fall on a regularly spaced grid. We draw the grid using light blue horizontal and vertical lines spaced one tenth of a unit apart. Now we can pass the values we calculated above to the rectangle function, using `mapply` (the multi-variate version of `sapply`) to iterate over all the cases and draw all the green and blue rectangles. Finally we plot the ROC curve (that is, we plot TPR against FPR) on top of everything in red.

```
plot(0:10/10, 0:10/10, type='n', xlab="FPR", ylab="TPR")
abline(h=0:10/10, col="lightblue")
abline(v=0:10/10, col="lightblue")

with(roc_df, {
  mapply(rectangle, x=FPR, y=0,
         width=dFPR, height=TPR, col="green", lwd=2)
  mapply(rectangle, x=FPR, y=TPR,
         width=dFPR, height=dTPR, col="blue", lwd=2)

  lines(FPR, TPR, type='b', lwd=3, col="red")
})
```



Adding up the area

The area under the red curve is all of the green area plus half of the blue area. For adding areas we only care about the height and width of each rectangle, not its (x,y) position. The heights of the green rectangles, which all start from 0, are in the TPR column and widths are in the dFPR column, so the total area of all the green rectangles is the dot product of TPR and dFPR. Note that the vectored approach computes a rectangle for each data point, even when the height or width is zero (in which case it doesn't hurt to add them). Similarly, the heights and widths of the blue rectangles (if there are any) are in columns dTPR and dFPR, so their total area is the dot product of these vectors. For regions of the graph that form square steps, one or the other of these values will be zero, so you only get blue rectangles (of non-zero area) if both TPR and FPR change in the same step. Only half the area of each blue rectangle is below its segment of the ROC curve (which is a diagonal of a blue rectangle). Remember the 'real' `auc` function gave us an AUC of 0.825, so that is the answer we're looking for.

```
simple_auc <- function(TPR, FPR){
  # inputs already sorted, best scores first
  dFPR <- c(diff(FPR), 0)
  dTPR <- c(diff(TPR), 0)
  sum(TPR * dFPR) + sum(dTPR * dFPR)/2
}

with(roc_df, simple_auc(TPR, FPR))
```

```
## [1] 0.825
```

Enumerating rank comparisons

Now let's try a completely different approach. Here we generate a matrix representing all possible combinations of a positive case with a negative case. Each row represents a positive case, in order from the highest-scoring positive case at the bottom to the lowest-scoring positive case at the top. Similarly, the columns represent the negative cases, sorted with the highest scores at the left. Each cell represents a comparison between a particular positive case and a particular negative case, and we mark the cell by whether its positive case has a higher score (or higher overall rank) than its negative case. If your classifier is any good, most of the positive cases will outrank most of the negative cases, and any exceptions will be in the upper left corner, where low-ranking positives are being compared to high-ranking negatives.

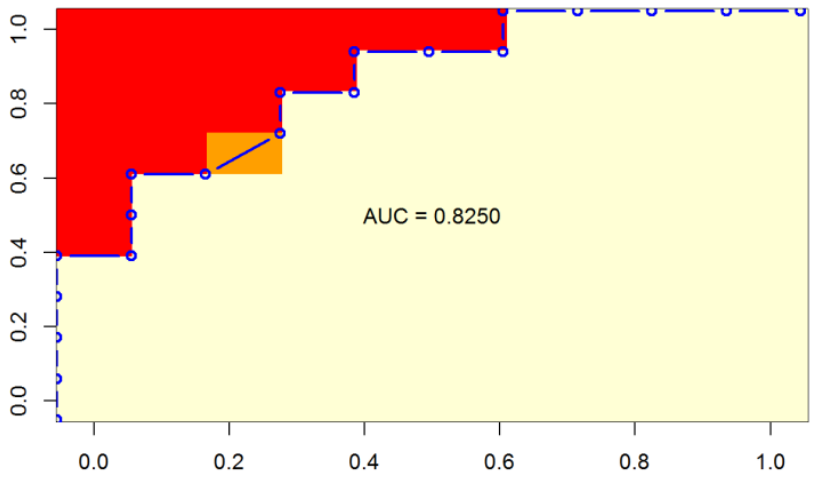
```
rank_comparison_auc <- function(labels, scores, plot_image=TRUE, ...){
  score_order <- order(scores, decreasing=TRUE)
  labels <- as.logical(labels[score_order])
  scores <- scores[score_order]
  pos_scores <- scores[labels]
  neg_scores <- scores[!labels]
  n_pos <- sum(labels)
```

```
  n_neg <- sum(!labels)
  M <- outer(sum(labels):1, 1:sum(!labels),
             function(i, j) (1 + sign(pos_scores[i] - neg_scores[j]))/2)

  AUC <- mean (M)
  if (plot_image){
    image(t(M[nrow(M):1,]), ...)
    library(pROC)
    with( roc(labels, scores),
          lines((1 + 1/n_neg)*((1 - specificities) - 0.5/n_neg),
                (1 + 1/n_pos)*sensitivities - 0.5/n_pos,
                col="blue", lwd=2, type='b'))
    text(0.5, 0.5, sprintf("AUC = %0.4f", AUC))
  }

  return(AUC)
}

rank_comparison_auc(labels=as.logical(category), scores=prediction)
```




```
## [1] 0.825
```

The blue line is an ROC curve computed in the conventional manner (slid and stretched a bit to get the coordinates to line up with the corners of the matrix cells). This makes it evident that the ROC curve marks the boundary of the area where the positive cases outrank the negative cases. The AUC can be computed by adjusting the values in

the matrix so that cells where the positive case outranks the negative case receive a `1`, cells where the negative case has higher rank receive a `0`, and cells with ties get `0.5` (since applying the `sign` function to the difference in scores gives values of 1, -1, and 0 to these cases, we put them in the range we want by adding one and dividing by two.) We find the AUC by averaging these values.

AUC as probability

The probabilistic interpretation is that if you randomly choose a positive case and a negative case, the probability that the positive case outranks the negative case according to the classifier is given by the AUC. This is evident from the figure, where the total area of the plot is normalized to one, the cells of the matrix enumerate all possible combinations of positive and negative cases, and the fraction under the curve comprises the cells where the positive case outranks the negative one.

We can use this observation to approximate AUC:

```
auc_probability <- function(labels, scores, N=1e7){
  pos <- sample(scores[labels], N, replace=TRUE)
  neg <- sample(scores[!labels], N, replace=TRUE)
  # sum( (1 + sign(pos - neg))/2)/N # does the same thing
  (sum(pos > neg) + sum(pos == neg)/2) / N # give partial credit for ties
}

auc_probability(as.logical(category), prediction)
```

```
## [1] 0.8249989
```

Testing on a bigger example

Now let's try our new AUC functions on a bigger dataset. I'll use the simulated dataset from the earlier blog post, where the labels are in the `bad_widget` column of the test set dataframe, and the scores are in a vector called `glm_response_scores`.

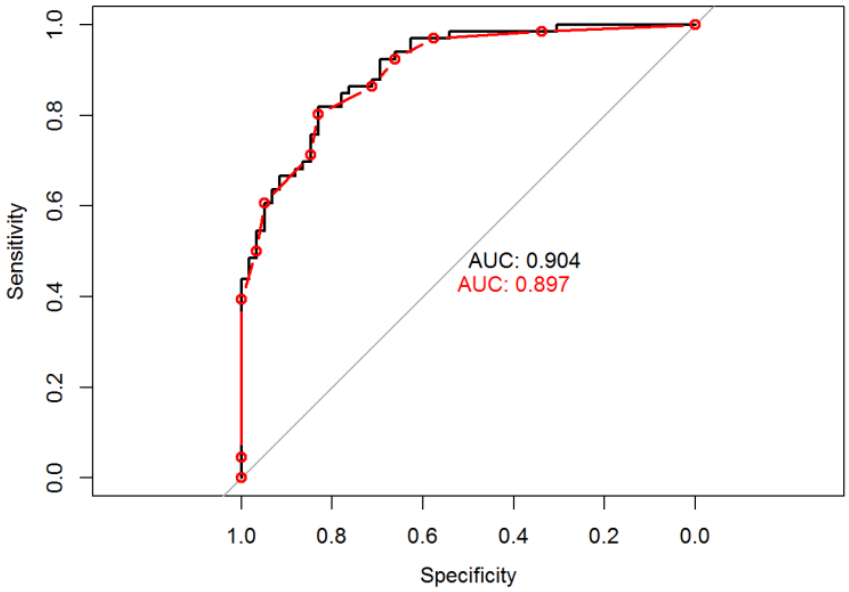
This data has no tied scores, so for testing let's make a modified version that has ties. We'll plot a black line representing the original data; since each point has a unique score, the ROC curve is a step function. Then we'll generate tied scores by rounding the score values, and plot the rounded ROC in red. Note that we are using "response" scores from a `glm` model, so they all fall in the range from 0 to 1. When we round these scores to one decimal place, there are 11 possible rounded scores, from 0.0 to 1.0. The AUC values calculated with the `pROC` package are indicated on the figure.

```
roc_full_resolution <- roc(test_set$bad_widget, glm_response_scores)
rounded_scores <- round(glm_response_scores, digits=1)
```

```
roc_rounded <- roc(test_set$bad_widget, rounded_scores)
plot(roc_full_resolution, print.auc=TRUE)
```

```
##
## Call:
## roc.default(response = test_set$bad_widget, predictor = glm_response_scores)
##
## Data: glm_response_scores in 59 controls (test_set$bad_widget FALSE) < 66 cases
(test_set$bad_widget TRUE).
## Area under the curve: 0.9037
```

```
lines(roc_rounded, col="red", type='b')
text(0.4, 0.43, labels=sprintf("AUC: %0.3f", auc(roc_rounded)), col="red")
```

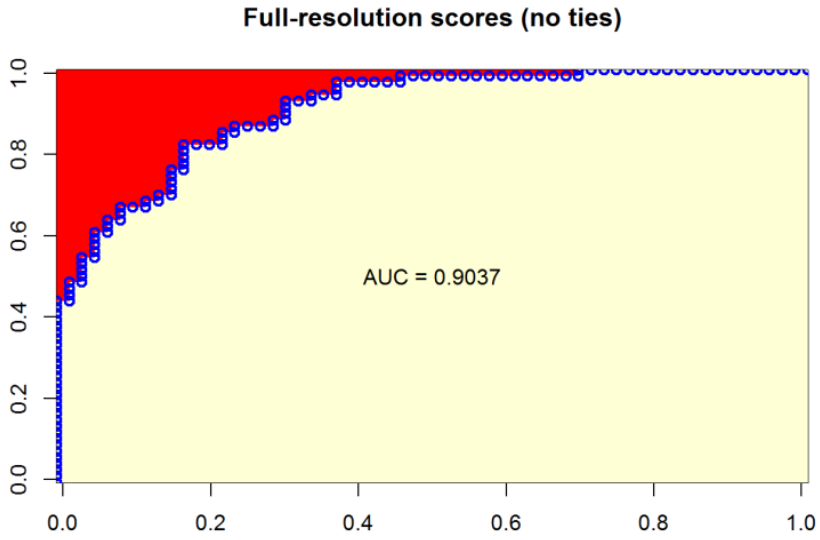


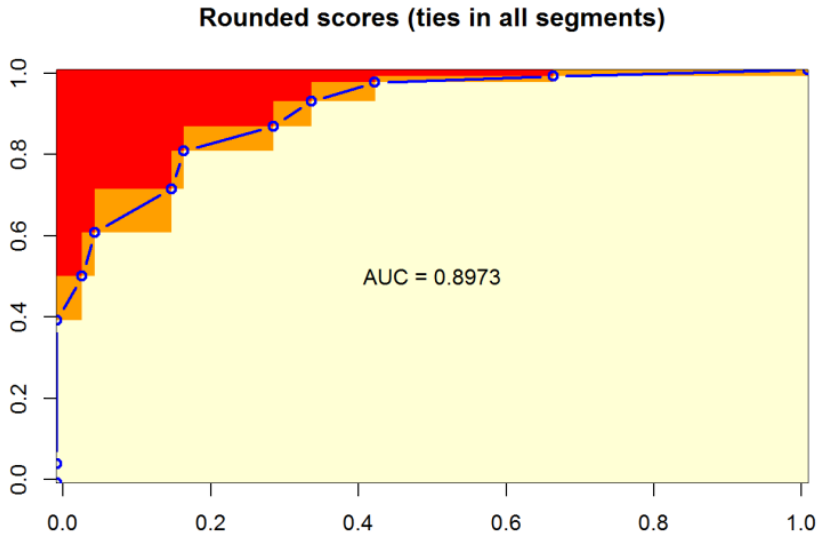
Now we can try our AUC functions on both sets to check that they can handle both step functions and segments with intermediate slopes.

```
options(digits=22)
set.seed(1234)

results <- data.frame(
  `Full Resolution` = c(
    auc = as.numeric(auc(roc_full_resolution)),
    simple_auc = simple_auc(rev(roc_full_resolution$sensitivities), rev(1 -
roc_full_resolution$specificities)),
    rank_comparison_auc = rank_comparison_auc(test_set$bad_widget, glm_response_scores,
                          main="Full-resolution scores (no ties)"),
    auc_probability = auc_probability(test_set$bad_widget, glm_response_scores)
```

```
  ),
  `Rounded Scores` = c(
    auc = as.numeric(auc(roc_rounded)),
    simple_auc = simple_auc(rev(roc_rounded$sensitivities), rev(1 -
roc_rounded$specificities)),
    rank_comparison_auc = rank_comparison_auc(test_set$bad_widget, rounded_scores,
                          main="Rounded scores (ties in all segments)"),
    auc_probability = auc_probability(test_set$bad_widget, rounded_scores)
  )
)
```





	Full.Resolution	Rounded.Scores
auc	0.90369799691833586	0.89727786337955828
simple_auc	0.90369799691833586	0.89727786337955828
rank_comparison_auc	0.90369799691833586	0.89727786337955828
auc_probability	0.90371970000000001	0.89716879999999999

So we have two new functions that give exactly the same results as the function from the `pROC` package, and our probabilistic function is pretty close. Of course, these functions are intended as demonstrations; you should normally use standard packages such as `pROC` or `ROCR` for actual work.

Here we've focused on calculating AUC and understanding the probabilistic interpretation. The probability associated with AUC is somewhat arcane, and is not likely to be exactly what you are looking for in practice (unless you actually will be randomly selecting a positive and a negative case, and you really want to know the probability that the classifier will score the positive case higher.) While AUC gives a single-number summary of classifier performance that is suitable in some circumstances, other metrics are often more appropriate. In many applications, overall behavior of a classifier across all possible score thresholds is of less interest than the behavior in a specific range. For example, in marketing the goal is often to identify a highly enriched target group with a low false positive rate. In other applications it may be more important to clearly identify a group of cases likely to be negative. For example, when pre-screening for a disease or defect you may want to rule out as many cases as you can before you start running expensive confirmatory tests. More generally, evaluation metrics that take into account the actual costs of false positive and false negative errors may be much more appropriate than AUC. If you know these costs, you should probably use them. A good introduction relating ROC curves to economic utility functions, complete with story and characters, is given in the excellent blog post "ML Meets Economics."