

Revolutions

Milestones in AI, Machine Learning, Data Science, and visualization with R and Python since 2008

August 02, 2016

ROC Curves in Two Lines of R Code

by Bob Horton, Microsoft Data Scientist

ROC curves are commonly used to characterize the sensitivity/specificity tradeoffs for a binary classifier. Most machine learning classifiers produce real-valued scores that correspond with the strength of the prediction that a given case is positive. Turning these real-valued scores into yes or no predictions requires setting a threshold; cases with scores above the threshold are classified as positive, and cases with scores below the threshold are predicted to be negative. Different threshold values give different levels of sensitivity and specificity. A high threshold is more conservative about labelling a case as positive; this makes it less likely to produce false positive results but more likely to miss cases that are in fact positive (lower rate of true positives). A low threshold produces positive labels more liberally, so it is less specific (more false positives) but also more sensitive (more true positives). The ROC curve plots true positive rate against false positive rate, giving a picture of the whole spectrum of such tradeoffs.

There are commonly used packages to plot these curves and to compute metrics from them, but it can still be worthwhile to contemplate how these curves are calculated to try to understand better what they show us. Here I present a simple function to compute an ROC curve from a set of outcomes and associated scores.

The calculation has two steps:

1. Sort the observed outcomes by their predicted scores with the highest scores first.
2. Calculate cumulative True Positive Rate (TPR) and True Negative Rate (TNR) for the ordered observed outcomes.

```
1 simple_roc <- function(labels, scores){  
2   labels <- labels[order(scores, decreasing=TRUE)]  
3   data.frame(TPR=cumsum(labels)/sum(labels), FPR=cumsum(!labels)/sum(!labels), labels)  
4 }
```

simple_roc.R hosted with ♥ by GitHub

[view raw](#)

The function takes two inputs: `labels` is a boolean vector with the actual classification of each case, and `scores` is a vector of real-valued prediction scores assigned by some classifier.

Since this is a binary outcome, the labels vector is a series of TRUE and FALSE values (or ones and zeros if you prefer). You can think of this series of binary values as a sequence of instructions for turtle graphics, only in this case the turtle has a compass and takes instructions in terms of absolute plot directions (North or East) instead of relative left or right. The turtle starts at the origin (as turtles do) and it traces a path across the page dictated by the sequence of instructions. When it sees a one (TRUE) it takes a step Northward (in the positive y direction); when it sees a zero (FALSE) it takes a step to the East (the positive x direction). The sizes of the steps along each axis are scaled so that once the turtle has seen all of the ones it will be at 1.0 on the y-axis, and once it has seen all of the zeros it will be at 1.0 on the x-axis. The path across the page is determined by the order of the ones and zeros, and it always finishes in the upper right corner.

The progress of the turtle along the bits of the instruction string represents adjusting the classification threshold to be less and less stringent. Once the turtle has passed a bit, it has decided to classify that bit as positive. If the bit was in fact positive it is a true positive; otherwise it is a false positive. The y-axis shows the true positive rate (TPR), which is the number of true positives encountered so far divided by the total number of actual positives. The x-axis shows the false positive rate (the number of false positives encountered up to that point divided by total number of true negatives). The vectorized implementation of this logic uses cumulative sums (the `cumsum` function) instead of walking through the values one at a time, though that is what the computer is doing at a lower level.

An ROC “curve” computed in this way is actually a step function. If you had very large numbers of positive and negative cases, these steps would be very small and the curve would appear smooth. (If you actually want to plot ROC curves for large numbers of cases, it

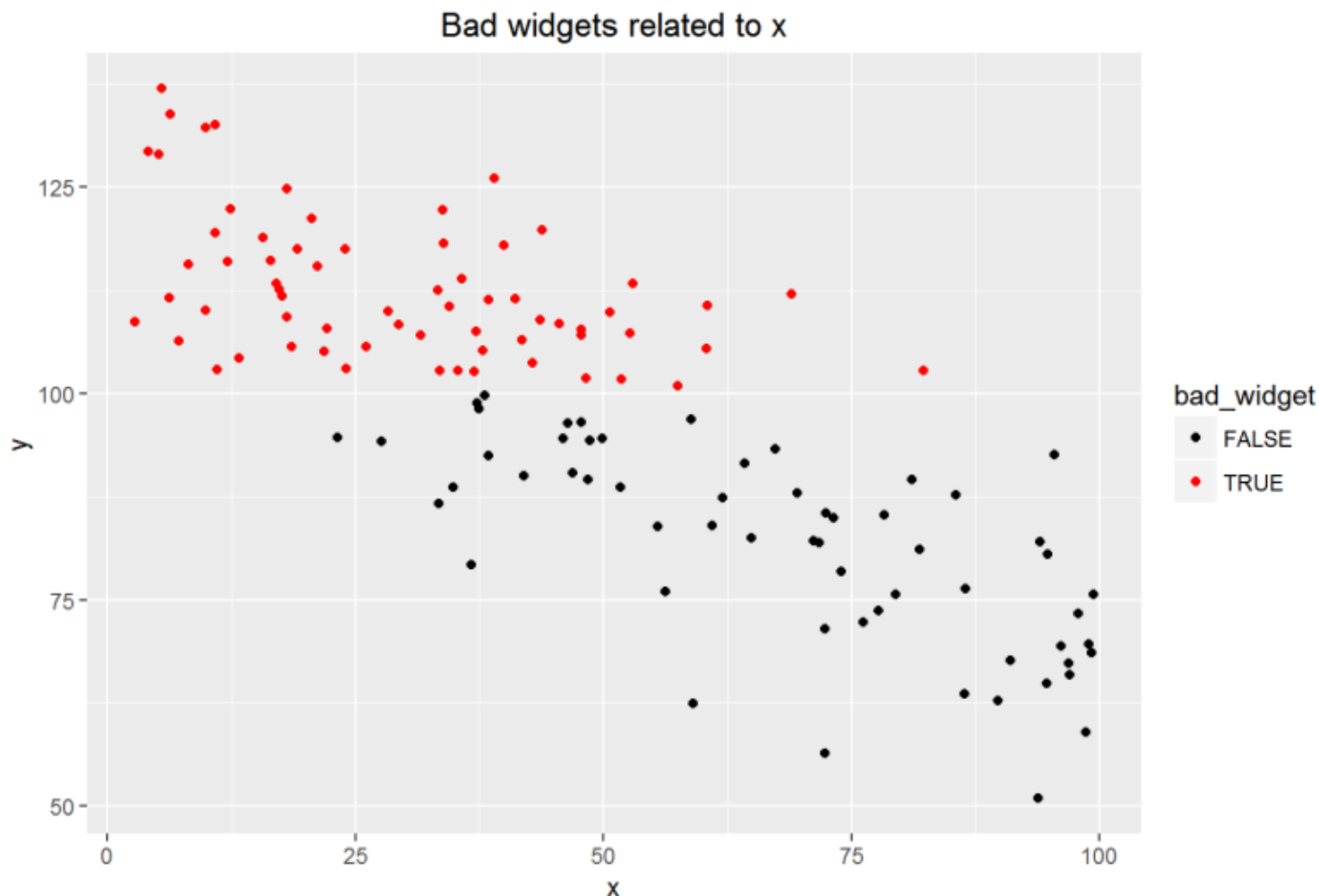
could be problematic to plot every point; this one reason that production-grade ROC functions take more than two lines of code.)

As an example, we will simulate data about widgets. Here we have an input feature x that is linearly related to a latent outcome y which also includes some randomness. The value of y determines whether the widget exceeds tolerance requirements; if it does, it is a bad widget.

```
1  set.seed(1)
2  sim_widget_data <- function(N, noise=100){
3    x <- runif(N, min=0, max=100)
4    y <- 122 - x/2 + rnorm(N, sd=noise)
5    bad_widget <- factor(y > 100)
6    data.frame(x, y, bad_widget)
7  }
8  widget_data <- sim_widget_data(500, 10)
9
10 test_set_idx <- sample(1:nrow(widget_data), size=floor(nrow(widget_data)/4))
11
12 test_set <- widget_data[test_set_idx,]
13 training_set <- widget_data[-test_set_idx,]
14 library(ggplot2)
15 library(dplyr)
16 test_set %>%
17   ggplot(aes(x=x, y=y, col=bad_widget)) +
18   scale_color_manual(values=c("black", "red")) +
19   geom_point() +
20   ggtitle("Bad widgets related to x")
```

simulate_widgets.R hosted with ♥ by GitHub

[view raw](#)



We reserve about 1/4 of cases for test set, and we'll use the rest for training a predictive model. The plot shows the test set, since that is the data we'll use to generate the ROC curves. If x below about 20 all the points are red, and above about 80 they are all black. In between is a region of varying uncertainty with more red at one end and more black at the other.

We use the training set to fit a logistic regression model using the x feature to predict whether a given widget is likely to be bad. This model will be used to generate scores for the test set, which will be used together with the actual labels of the test cases to calculate ROC curves. The values from the `scores` vector will not appear in the plot; they are only used to sort the labels. Two classifiers that put the labels in the same order will have exactly the same ROC curve regardless of the absolute values of the scores. This is shown by comparing the ROC curve you get using either the 'response' or the 'link' predictions from a logistic regression model. The 'response' scores have been mapped into the range between 0 and 1 by a sigmoid function and the 'link' scores have not. But either of these scores will put the points in the same order.

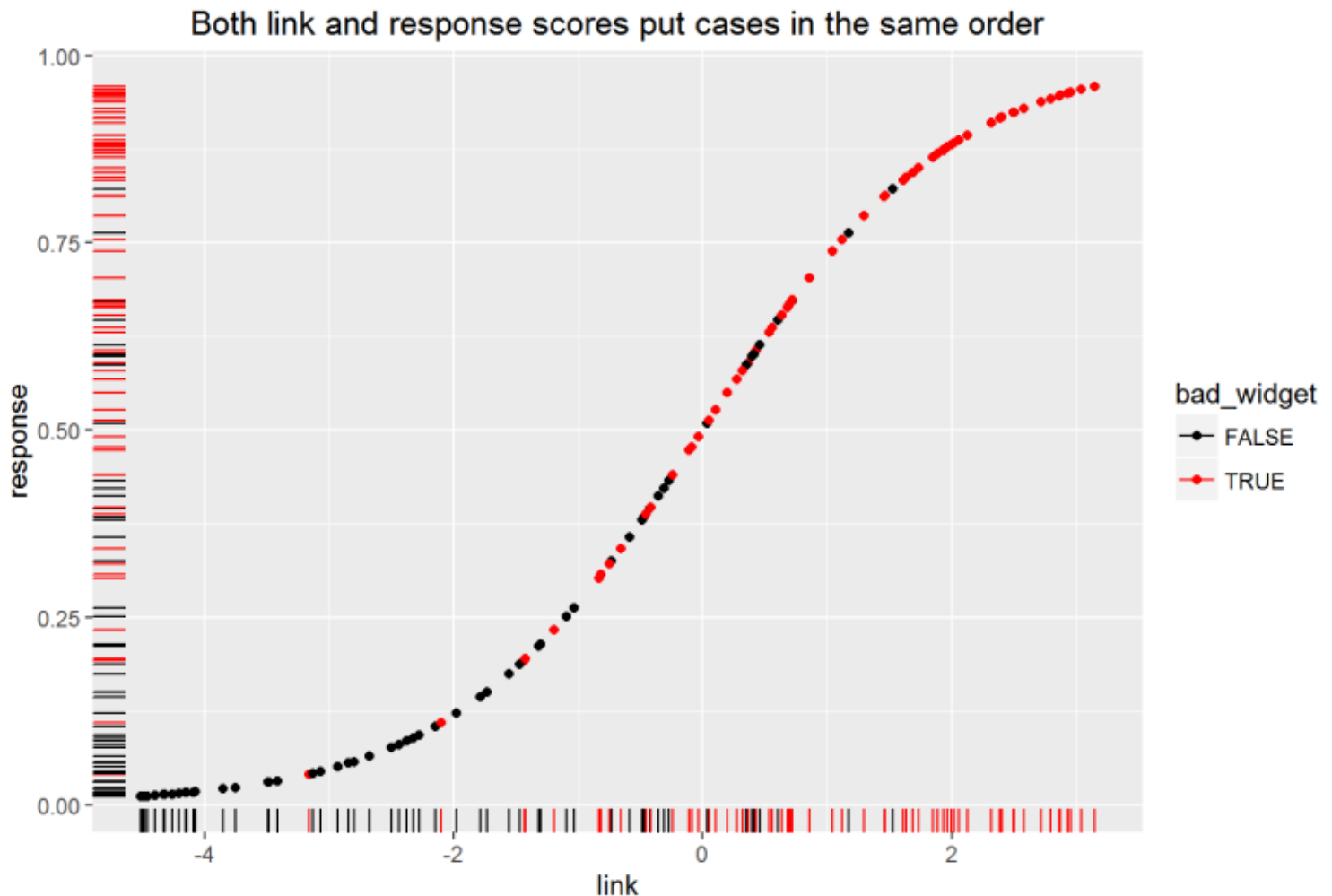
```
1 fit_glm <- glm(bad_widget ~ x, training_set, family=binomial(link="logit"))
2
3 glm_link_scores <- predict(fit_glm, test_set, type="link")
4
5 glm_response_scores <- predict(fit_glm, test_set, type="response")
6
7 score_data <- data.frame(link=glm_link_scores,
8                           response=glm_response_scores,
9                           bad_widget=test_set$bad_widget,
```

```

10         stringsAsFactors=FALSE)
11
12 score_data %>%
13   ggplot(aes(x=link, y=response, col=bad_widget)) +
14     scale_color_manual(values=c("black", "red")) +
15     geom_point() +
16     geom_rug() +
17     ggtitle("Both link and response scores put cases in the same order")

```

glm_widgets.R hosted with ♥ by GitHub

[view raw](#)

```

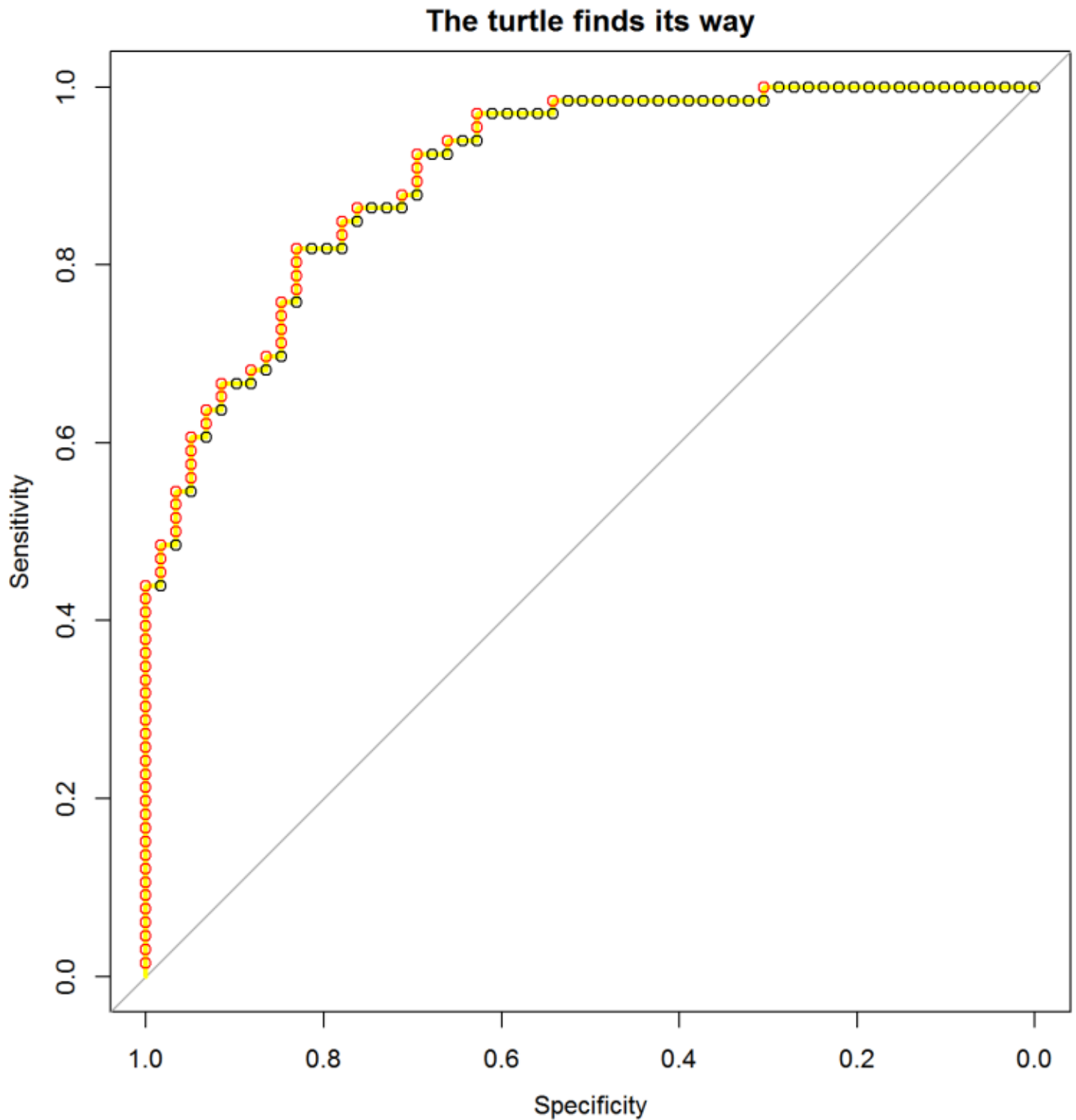
1 library(pROC)
2 plot(roc(test_set$bad_widget, glm_response_scores, direction="<"),
3       col="yellow", lwd=3, main="The turtle finds its way")
4 ##
5 ## Call:
6 ## roc.default(response = test_set$bad_widget, predictor = glm_response_scores, direction = "<")
7 ##
8 ## Data: glm_response_scores in 59 controls (test_set$bad_widget FALSE) < 66 cases (test_set$bad_widget TRUE)

```

```
9  ## Area under the curve: 0.9037
10 glm_simple_roc <- simple_roc(test_set$bad_widget=="TRUE", glm_link_scores)
11 with(glm_simple_roc, points(1 - FPR, TPR, col=1 + labels))
```

proc_widgets.R hosted with ♥ by GitHub

[view raw](#)



Here the ROC curve for the response scores from the logistic regression model is calculated with the widely used `pROC` package and plotted as a yellow line. The `simple_roc` function was also used to calculate an ROC curve, but in this case it is calculated from the link scores. Since both sets of scores put the labels in the same order, and since both functions are doing essentially the same thing, we get the same curve. The points of the `simple_roc` curve are plotted as open circles, which land exactly on top of the yellow line. Each point represents a single case in the test set, and the outline colors of the circles show whether that case was a “bad widget” (red) or not (black). Red circles tell the turtle to go North, and black circles tell it to go East.

Note that the `pROC` package labels the x-axis “Specificity” with 1.0 on the left and 0 on the right (specificity is 1 minus the false positive rate). This means we had to do a similar subtraction to plot the `simple_roc` results the same way. Also, left to its own devices, `roc` decides how to label the cases and controls based on which group has the higher median score, which means it can flip a negative AUC to be positive (basically, a consistently wrong predictor can still be useful if you use reverse psychology). I specify `direction="<"` to prevent this since the `simple_roc` function is not that smart.

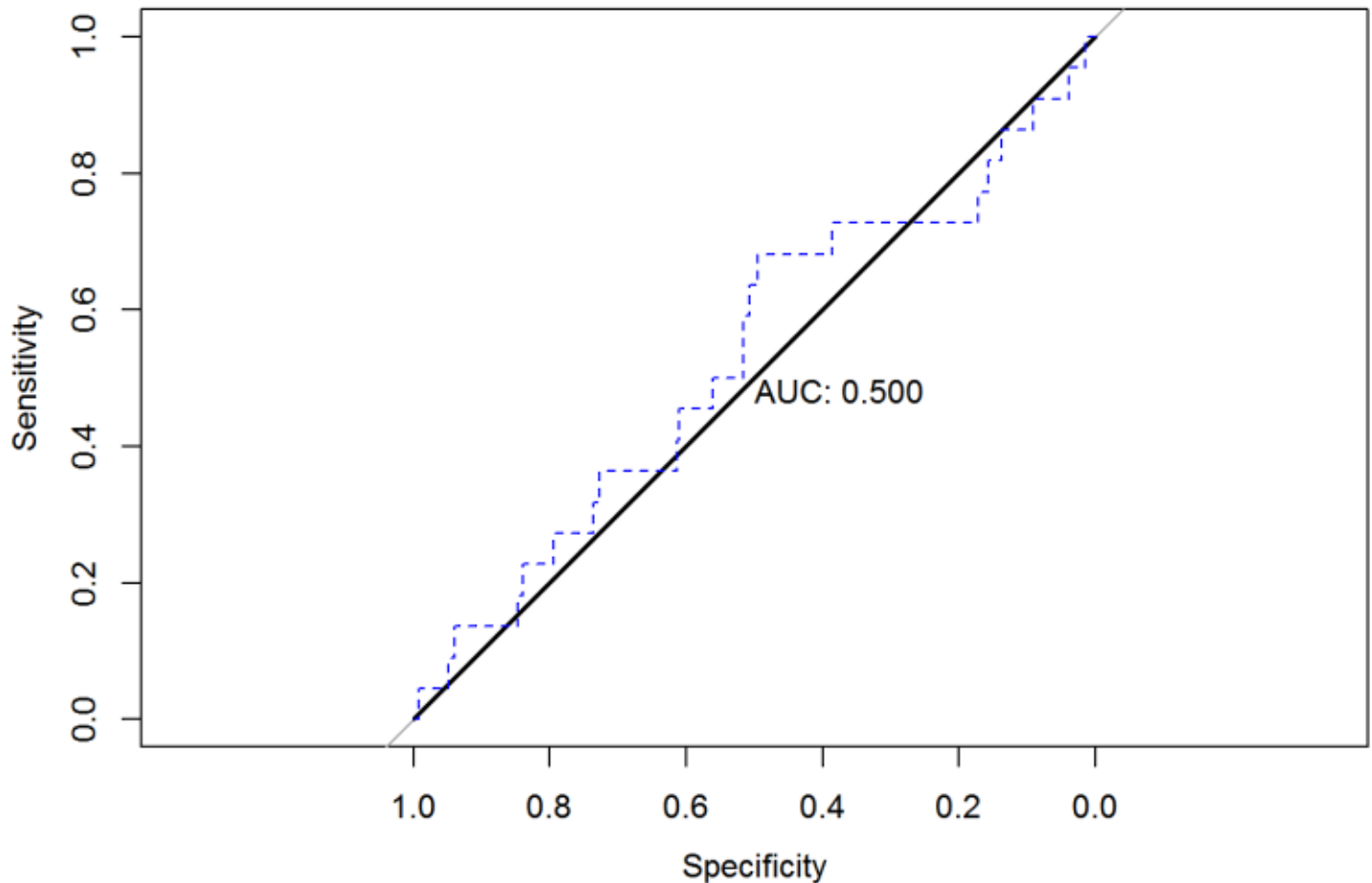
This brings up another limitation of this simple approach; by assuming that the rank order of the outcomes embodies predictive information from the model, it does not properly handle sequences of cases that all have the same score. The turtle assumes that the order of the labels has meaning, but in the situation of identical scores there is no meaningful order. These segments should properly be represented by a diagonal line, while our simple turtle will happily plot meaningless steps.

We’ll show an extreme case by creating an unbalanced dataset that is positive in only about 1% of cases. For prediction, we just always guess that the result will be negative (achieving 99% accuracy). Since all the scores are the same, we don’t really have any basis for sorting the outcomes; `pROC` handles this correctly and draws a diagonal line. The turtle assumes that the order of the cases means something when in fact it does not, and it takes a sort of random walk up to the top right corner.

```
1  set.seed(1)
2  N <- 2000
3  P <- 0.01
4  rare_success <- sample(c(TRUE, FALSE), N, replace=TRUE, prob=c(P, 1-P))
5  guess_not <- rep(0, N)
6  plot(roc(rare_success, guess_not), print.auc=TRUE)
7  ##
8  ## Call:
9  ## roc.default(response = rare_success, predictor = guess_not)
10 ##
11 ## Data: guess_not in 1978 controls (rare_success FALSE) < 22 cases (rare_success TRUE).
12 ## Area under the curve: 0.5
13 simp_roc <- simple_roc(rare_success, guess_not)
14 with(simp_roc, lines(1 - FPR, TPR, col="blue", lty=2))
```

unbalanced_sim.R hosted with ♥ by GitHub

[view raw](#)



```
simp_roc2 <- simple_roc(rare_success, runif(length(guess_not)))
```


If you repeat this simulation with larger value of N you will see that the turtle's path tends to approximate the diagonal more closely, but the more unbalanced the outcomes are, the larger numbers of total cases you will likely need to keep the paths from deviating wildly from the diagonal.

For a less extreme example, you can usually generate diagonal segments in an ordinary ROC curve by rounding scores so that multiple points get identical ranks; this is left as an exercise for the reader.

Because ROC curves are so instructive and commonly used, they deserve some study and contemplation. For further information I recommend this [shiny app](#) showing continuous-valued ROC curves computed from probability distributions, and the excellent paper by Tom Fawcett entitled [An introduction to ROC analysis](#).

Posted by [Joseph Rickert](#) at 08:14 in [data science](#), [R](#), [statistics](#) | [Permalink](#)

Comments

 You can follow this conversation by subscribing to the [comment feed](#) for this post.

The comments to this entry are closed.