logistic-regression

Joseph Plummer

GitHub Documents

This is an R Markdown format used for publishing markdown documents to GitHub. When you click the **Knit** button all R code chunks are run and a markdown file (.md) suitable for publishing to GitHub is generated.

Logistic regression

In this document, we use logistic regression to model the probability of having disease given a biomarker.

It is inspired by the well written works of:

https://daviddalpiaz.github.io/r4sl/logistic-regression.html

Load libraries

You can install a library if necessary by using `install.packages()`.

```
library(caret)
library(pROC)
library(ggplot2)
```

Load data

We will be using anonymous ventilation defect percentage (VDP) and disease classification data.

```
# Generate random data.
data <- data.frame(VDP = runif(100, min=0, max=15), Disease = rbinom(100, 1, 0.5))

# Or supply your own.
# Import data:
data <- read.csv("data/vdp.csv")
x <- data$VDP
y <- data$Disease
```

Understand data

```
print(paste("num_subj = ",length(x)))
```

```
## [1] "num_subj =  67"
```

```
head(data)
```

```
##    VDP Disease
## 1  8.3       0
## 2  4.6       0
## 3  9.7       0
## 4 16.0       1
## 5  4.8       0
## 6  4.4       0
```

```
summary(data)
```

```
##       VDP           Disease
##  Min.   : 0.50   Min.   :0.0000
##  1st Qu.: 2.95   1st Qu.:0.0000
##  Median : 5.20   Median :0.0000
##  Mean   :11.56   Mean   :0.3582
##  3rd Qu.:18.30   3rd Qu.:1.0000
##  Max.   :45.30   Max.   :1.0000
```

Split data into testing and training

```
set.seed(42)
data_idx = sample(nrow(data), 20)
data_trn = data[data_idx, ]
data_tst = data[-data_idx, ]
```

This next line is illegal, but for learning purposes: let's use the same data for both testing and training the model. For real statistics, remove this section.

```
# data_trn = data
# data_tst = data
```

Logistic Regression with `glm()`

The probability of an outcome $Y = 1$ given a list of independent variables $X = x$ is mathematically represented by:

$$p(x) = P(Y = 1 \mid X = x)$$

In this case, x is a vector containing our independent variables data (in our case, we only have VDP). Logistic regression takes the log of the probability $p(x)$ over the anti-probability $1 - p(x)$, and fits a linear regression model to x:

$$\log\left(\frac{p(x)}{1 - p(x)}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

where β_i are the model coefficients for each x_i. In this form, this model does not provide any use to us. However, by fitting the model and working out the coefficients, we can then rearrange for the probability again. This is given by:

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)}} = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)$$

Notice, we use the sigmoid function as shorthand notation, which appears often in deep learning literature. It takes any real input, and outputs a number between 0 and 1. How useful! (This is actualy a particular sigmoid function called the logistic function, but since it is by far the most popular sigmoid function, often sigmoid function is used to refer to the logistic function).

$$\sigma(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}$$

The model is fit by numerically maximizing the likelihood, which we will let `R` take care of.

In this case, we have a single predictor x (our VDP measurements). We fit a generalized linear model in `R` using `glm`:

```
model_glm = glm(Disease ~ VDP, data = data_trn, family = "binomial")
```

Fitting this model looks very similar to fitting a simple linear regression. Instead of `lm()` we use `glm()`. The only other difference is the use of `family = "binomial"` which indicates that we have a two-class categorical response. Using `glm()` with `family = "gaussian"` would perform the usual linear regression.

We can obtain the fitted coefficients:

```
coef(model_glm)
```

```
## (Intercept)         VDP
##  -3.6650095   0.2061658
```

The next thing we should understand is how the `predict()` function works with `glm()`. So, let's look at some predictions.

```
head(predict(model_glm))
```

```
##          49          65          25          10          36          18
##  0.04597515 -2.92281259 -3.14959499 -3.41761055 -3.04651208 -2.59294729
```

By default, `predict.glm()` uses `type = "link"`.

```
head(predict(model_glm, type = "link"))
```

```
##          49          65          25          10          36          18
##  0.04597515 -2.92281259 -3.14959499 -3.41761055 -3.04651208 -2.59294729
```

That is, `R` is returning

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p$$

for each observation.

Importantly, these are **not** predicted probabilities. To obtain the predicted probabilities

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x)$$

we need to use `type = "response"`

```
head(predict(model_glm, type = "response"))
```

```
##         49         65         25         10         36         18
## 0.51149176 0.05103731 0.04110724 0.03174960 0.04536830 0.06959370
```

Note that these are probabilities, **not** classifications. To obtain classifications, we will need to compare to the correct cutoff value with an `ifelse()` statement.

```
model_glm_pred = ifelse(predict(model_glm, type = "link") > 0, "1", "0")
# model_glm_pred = ifelse(predict(model_glm, type = "response") > 0.5, "1", "0")
```

The line that is run is performing

$$\hat{C}(x) = \begin{cases} 1 & \hat{f}(x) > 0 \\ 0 & \hat{f}(x) \leq 0 \end{cases}$$

where

$$\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p.$$

The commented line, which would give the same results, is performing

$$\hat{C}(x) = \begin{cases} 1 & \hat{p}(x) > 0.5 \\ 0 & \hat{p}(x) \leq 0.5 \end{cases}$$

where

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x).$$

Once we have classifications, we can calculate metrics such as the training classification error rate.

```
calc_class_err = function(actual, predicted) {
  mean(actual != predicted)
}
```

```
calc_class_err(actual = data_trn$Disease, predicted = model_glm_pred)
```

```
## [1] 0.15
```

The `table()` and `confusionMatrix()` functions can be used to quickly obtain many more metrics.

```
train_tab = table(predicted = model_glm_pred, actual = data_trn$Disease)
train_tab
```

```
##          actual
## predicted  0  1
##         0 12  2
##         1  1  5
```

```
train_con_mat = confusionMatrix(train_tab, positive = "1")
c(train_con_mat$overall["Accuracy"],
  train_con_mat$byClass["Sensitivity"],
  train_con_mat$byClass["Specificity"])
```

```
##    Accuracy Sensitivity Specificity
##   0.8500000   0.7142857   0.9230769
```

ROC Curves

We write a function which allows use to make predictions based on different probability cutoffs.

```
get_logistic_pred = function(mod, data, res = "y", pos = 1, neg = 0, cut = 0.5) {
  probs = predict(mod, newdata = data, type = "response")
  ifelse(probs > cut, pos, neg)
}
```

$$\hat{C}(x) = \begin{cases} 1 & \hat{p}(x) > c \\ 0 & \hat{p}(x) \leq c \end{cases}$$

Let's use this to obtain predictions using a low, medium, and high cutoff. (0.1, 0.5, and 0.9)

```
test_pred_10 = get_logistic_pred(model_glm, data = data_tst, res = "Disease",
                                 pos = "1", neg = "0", cut = 0.1)
test_pred_50 = get_logistic_pred(model_glm, data = data_tst, res = "Disease",
                                 pos = "1", neg = "0", cut = 0.5)
test_pred_90 = get_logistic_pred(model_glm, data = data_tst, res = "Disease",
                                 pos = "1", neg = "0", cut = 0.9)
```

Now we evaluate accuracy, sensitivity, and specificity for these classifiers.

```
# Make robust to zero frequency
test_pred_10 <- factor(test_pred_10,levels=c(0,1))
test_pred_50 <- factor(test_pred_50,levels=c(0,1))
test_pred_90 <- factor(test_pred_90,levels=c(0,1))

# Write tables
test_tab_10 = table(predicted = test_pred_10, actual = data_tst$Disease)
test_tab_50 = table(predicted = test_pred_50, actual = data_tst$Disease)
test_tab_90 = table(predicted = test_pred_90, actual = data_tst$Disease)

# Generate confusion matrices
test_con_mat_10 = confusionMatrix(test_tab_10, positive = "1")
test_con_mat_50 = confusionMatrix(test_tab_50, positive = "1")
test_con_mat_90 = confusionMatrix(test_tab_90, positive = "1")
```

```r
metrics = rbind(

  c(test_con_mat_10$overall["Accuracy"],
    test_con_mat_10$byClass["Sensitivity"],
    test_con_mat_10$byClass["Specificity"]),

  c(test_con_mat_50$overall["Accuracy"],
    test_con_mat_50$byClass["Sensitivity"],
    test_con_mat_50$byClass["Specificity"]),

  c(test_con_mat_90$overall["Accuracy"],
    test_con_mat_90$byClass["Sensitivity"],
    test_con_mat_90$byClass["Specificity"])

)

rownames(metrics) = c("c = 0.10", "c = 0.50", "c = 0.90")
metrics
```

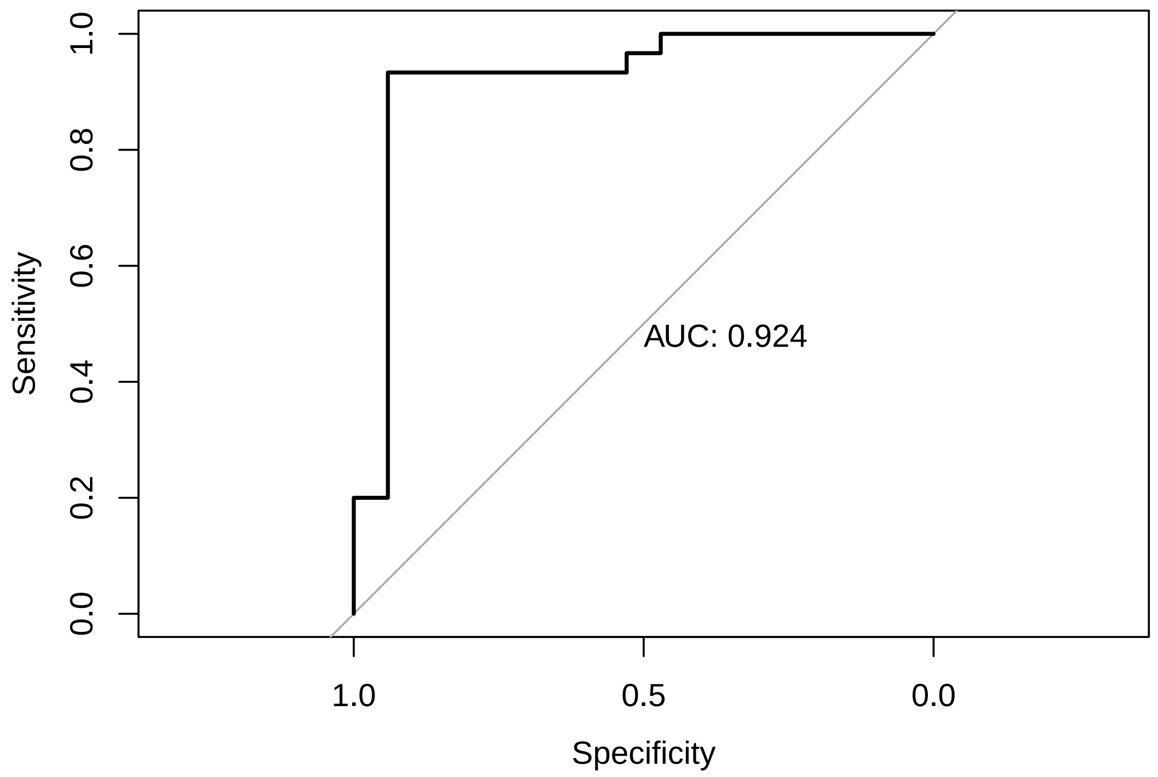
```
##          Accuracy Sensitivity Specificity
## c = 0.10 0.8723404   0.9411765   0.8333333
## c = 0.50 0.8085106   0.5882353   0.9333333
## c = 0.90 0.7021277   0.1764706   1.0000000
```

We see then sensitivity decreases as the cutoff is increased. Conversely, specificity increases as the cutoff increases. This is useful if we are more interested in a particular error, instead of giving them equal weight.

Note that usually the best accuracy will be seen near $c = 0.50$.

Instead of manually checking cutoffs, we can create an ROC curve (receiver operating characteristic curve) which will sweep through all possible cutoffs, and plot the sensitivity and specificity.

```r
test_prob = predict(model_glm, newdata = data_tst, type = "response")
test_roc = roc(data_tst$Disease ~ test_prob,
               levels = c(1,0),
               direction = ">",
               plot = TRUE, print.auc = TRUE)
```

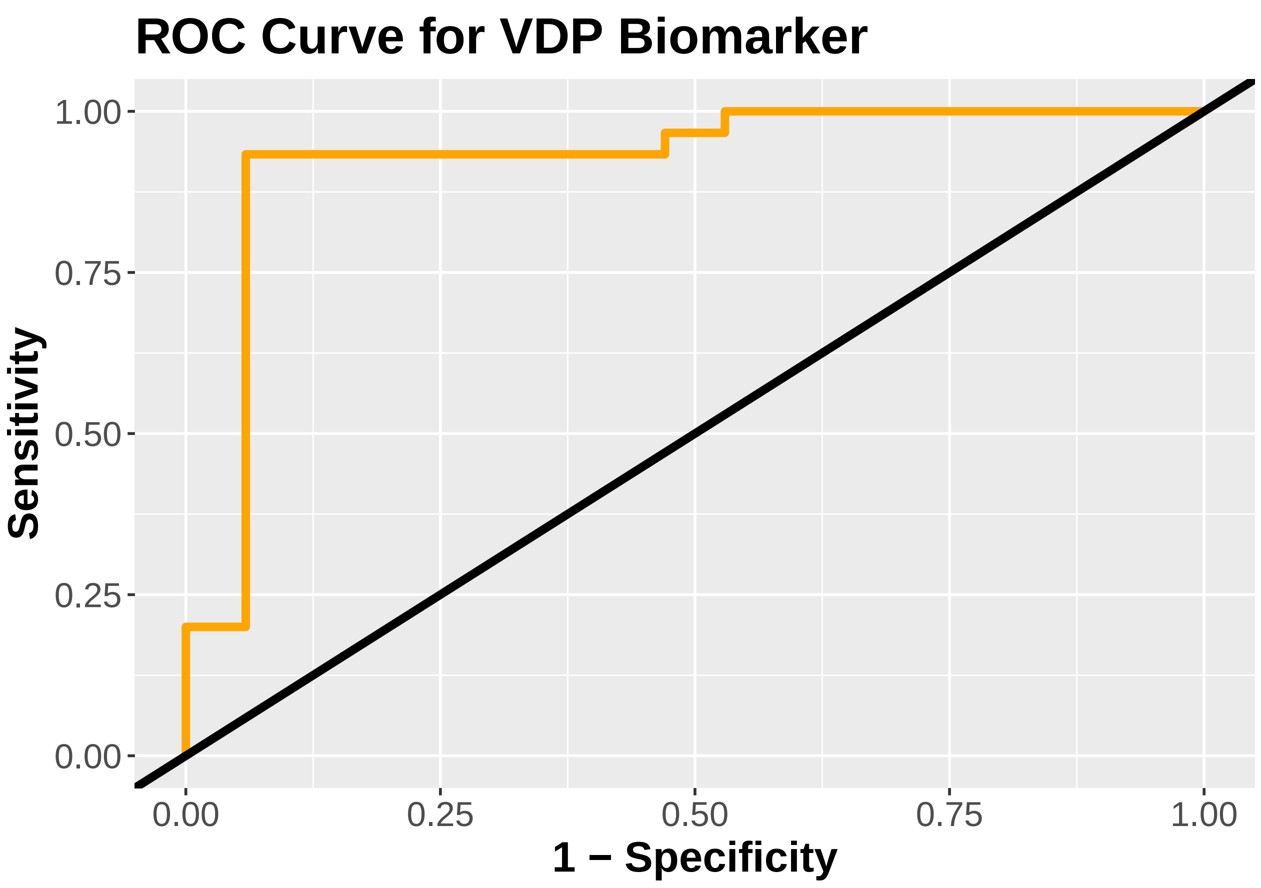


```
as.numeric(test_roc$auc)
```

```
## [1] 0.9235294
```

A good model will have a high AUC, that is as often as possible a high sensitivity and specificity.

We can also make a much prettier plot using `ggplot2`.



Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.