Today, I am going to cover classification problems.

So far, we have considered the regression problems. Our targets/labels are continuous variables such as test scores or house prices. These quantities have **infinite many** possible values. It is called a **regression problem**.

On the other hand, there are some problems that their targets have only **finite** many possibilities. For example, doctors may need to determine whether a patient has a heart disease or not. The target has only two cases here. If the target has **only finite many cases**, it is called a **classification problem**.

In this section, we only focus on the target that has only two cases, which is **binary classification problems**.

## Linear model for classification problem fails

Now let’s consider a real world example based on our test scores data set. Suppose we want to forecast the gender of the test takers using Race, MathScore, ReadingScore and WritingScore.

We notice that our target is gender that has only two cases, female and male, in the data set. It is a binary classification problem.

Let’s load the students’ performance data set into memory:

library(readxl)  
StudentsPerformance <- read\_excel("C:/Users/yliu3/OneDrive - Maryville University/Online DSCI502 R Programming/DataSets/StudentsPerformance.xlsx")

There are two types of variables/columns in our model

* **Gender** and **Race** are factors with finite many cases
* **MathScore**, **ReadingScore**, and **WritingScore** are continuous variables with infinite many cases

Typically, we need to convert variables with finite many cases to factors in R using the as.factor() function.

Let’s first convert the Gender and Race to factors:

StudentsPerformance$Gender <- as.factor(StudentsPerformance$Gender)  
StudentsPerformance$Race <- as.factor(StudentsPerformance$Race)

When we called the function as.factor(), it mapped the gender to numbers in the memory by the order of the levels. The levels are unique values of this categorical variable. We can look at the **levels** using the following R code:

levels(StudentsPerformance$Gender)

## [1] "female" "male"

R engine set the **default levels by alphabetical order**. Sometimes, you may want to specify the orders in the levels using the levels parameter in the factor() function. For example, if you want to have the order of “male”, “female” in your levels, you can run the following codes

StudentsPerformance$Gender <- factor(StudentsPerformance$Gender, levels = c("male", "female"))

We have learned the linear regression model. Can we use it to forecast Gender? Let’s try it.

lm.result <- lm(Gender ~ Race + MathScore + ReadingScore + WritingScore, data = StudentsPerformance)  
  
  
summary(lm.result)  
  
R engine gives us an **error message**; factors are not allowed in Residuals.

The main reason that linear model fails is because of classification problems:

* There are only finite many cases in the target, but the linear model could produce infinite many cases based on the right hand size of formula

Instead of predicting the gender directly, we can predict the probability of the gender.

Therefore, we would like to forecast:

But the probability of Gender is a continuous variable from 0 to 1. It has infinite many cases.

Our original target, Gender, has only two levels, female/0 and male/1, by looking at “levels(StudentsPerformance$Gender)” . We need to map the interval to , and that is the default levels in R memory.

We can choose a threshold, for example, , This threshold split the interval into two sub intervals and we can map these two intervals to two levels, 0/female and 1/male by the following map

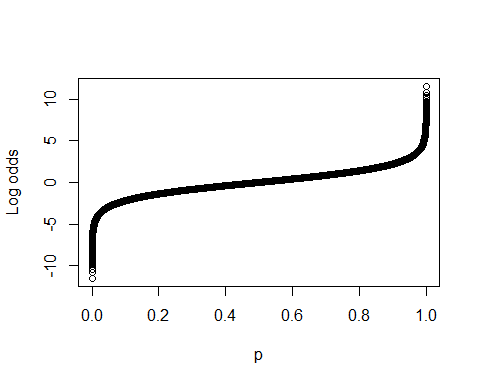
* if , then the gender is 1/male
* if , then the gender is 0/female

Let’s look at the logistic regression model. Let denote the probability of female that is **the first level of the default levels**. We consider the following model:

By taking the logarithm of the odds ratio (), we obtain a continuous variable on the right hand size of the equation.

Let’s graph the log of the odds ratio in R. To graph it we need to compute and plot many points on the curve. They are based on the same formula. We can use a **user defined function** to achieve it. We can graph it using the following R codes:

#define a function in R  
logodds <- function (p){  
  
 return(log(p/(1-p)))  
}  
#vectorize the user defined function  
vlogodds <- Vectorize(logodds)  
#plot it  
plot(x = seq(0.00001,0.99999, 0.00001), y = vlogodds(seq(0.00001,0.99999, 0.00001)), ylab = "Log odds", xlab = "p")



To define a function in R, we use the following syntax:

UserDefinedFunctionName <- function (arguments){  
 #perform computation, for example  
 result <- arguments^2  
 #return values using return keyword  
 return(result)  
}

We need to specify the function name in the left side and provide argument in the parenthis. Typically we need to return values in the function body enclosed by {}.

A function may have multiple parameters/arguments. For example, if we want to define a function to compute the area of a rectangle, we can use the following function

Area <- function(l, w= 10){  
 myarea <- l\*w  
 return(myarea)  
}

We notice that width has a **default argument** of 10 by assigning it using w =10 in the parenthesis. If we don’t specify the value for width, the R engine will use 10. If we specify it, the R engine will use the new value. For example:

Area(5,7)

## [1] 35

#This is equivalent to Area(5,10)  
Area(5)

## [1] 50

**All the default parameters must appear at the end of the declaration**. For example, the following function definition is not correct since the default parameter appeared on the left side and not at the end of the argument lists.

#There is a bug in the definition  
Area <- function(l=5, w){  
 myarea <- l\*w  
 return(myarea)  
}

The R code, **vlogodds <- Vectorize(logodds),** vectorize the scaler function logodds and get the corresponding vectorized version. Next we call **y = vlogodds(seq(0.00001,0.99999, 0.00001))** by providing a vector from the smallest number, 0.0001, to the largest number, 0.9999 with a step size = 0.00001. We exclude 0 and 1 since the domain of this function excludes 0 and 1 respectively. These codes are more efficient than the typical R loops.