BTRY4830_Lab9

Olivia Lang

3/28/2019

0. Overview

1. Reading in Data

   Review of data types

   What to do when bad data strikes

2. Calculate p-value with covariates

   Compare F-statitstic calculation with and without covariates

   Simulate some phenotype data with covariates

3. Logistic Regression

   GWAS model for catgorical phenotypes (binary, take values 0 or 1)

4. Exercise–Implementing the IRLS Algorithm to conduct a Logistic Regression GWAS

1. Reading in Data

We have been reading in data from genotype and phenotype files for the past few labs, but now let's slow down and actualy discuss the different arguments that can help with this process. The basic arguments are header which is a boolean argument deciding whether the first row should become the rownames, and row.names which is a numeric or F

```
geno <- read.table("example.tsv", header = T, row.names = 1)
```

In order to remove these row names set the argument to null. If the length of the header line was the same as the first row we could set header = F to also remove the column names, but because the header line is shorter we cannot. You can test this out.

```
geno <- read.table("example.tsv", header = T, row.names = NULL)
head(geno)
```

```
##   row.names rs791607 rs791608 rs4236625 rs4731427 rs10244329
## 1   NA18486        0        0         1         1         -1
## 2   NA18487       -1       -1         0         0          0
## 3   NA18488        0        0         1         1          0
## 4   NA18489        1        1         0         0          0
## 5   NA18498        1        1         0         0          1
## 6   NA18499        1        1         0         0          1
```

However this process would not work if you tried to read in a comma separated file:

```
geno <- read.table("example.csv", header = T, row.names = 1)
head(geno)
```

```
## data frame with 0 columns and 6 rows
```

We have to specify the delimiter or switch to using the read.csv() function:

```
geno <- read.table("example.csv", header = T, row.names = 1, sep=",")
geno <- read.csv("example.csv", header = T, row.names = 1)
head(geno)
```

```
##         rs791607 rs791608 rs4236625 rs4731427 rs10244329
## NA18486       TT       TT        AA        AA         CC
## NA18487       CC       CC        TT        TT         TT
## NA18488       TT       TT        AA        AA         TT
## NA18489       AA       AA        TT        TT         TT
## NA18498       AA       AA        TT        TT         AA
## NA18499       AA       AA        TT        TT         AA
```

Look at the data types of each column, they are factors. Factors are a data type, just like numeric, character, or integers. There are two parts of a factor, the data and the levels. To a computer the data is a basic integer 1,2,3, ... and the levels are the translation 1="a", 2="b", 3="c", ...

```
str(geno[,1])
```

```
##  Factor w/ 3 levels "AA","CC","TT": 3 2 3 1 1 1 3 3 1 3 ...
```

Factors can be great for two reasons: you have catagorical data or you have lots of data.

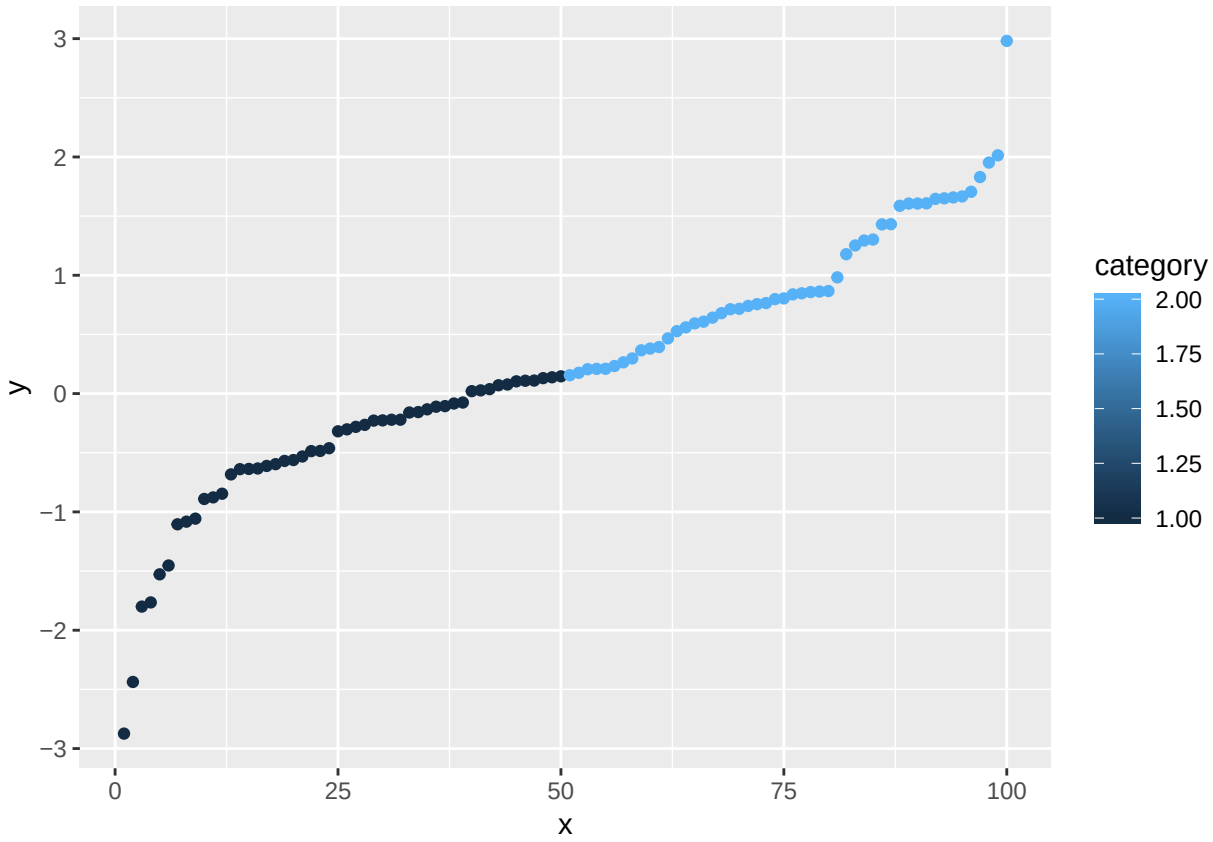
```
x <- sample(c("AA","CC"),10000,replace = T)
object.size(x)
```

```
## 80160 bytes
```

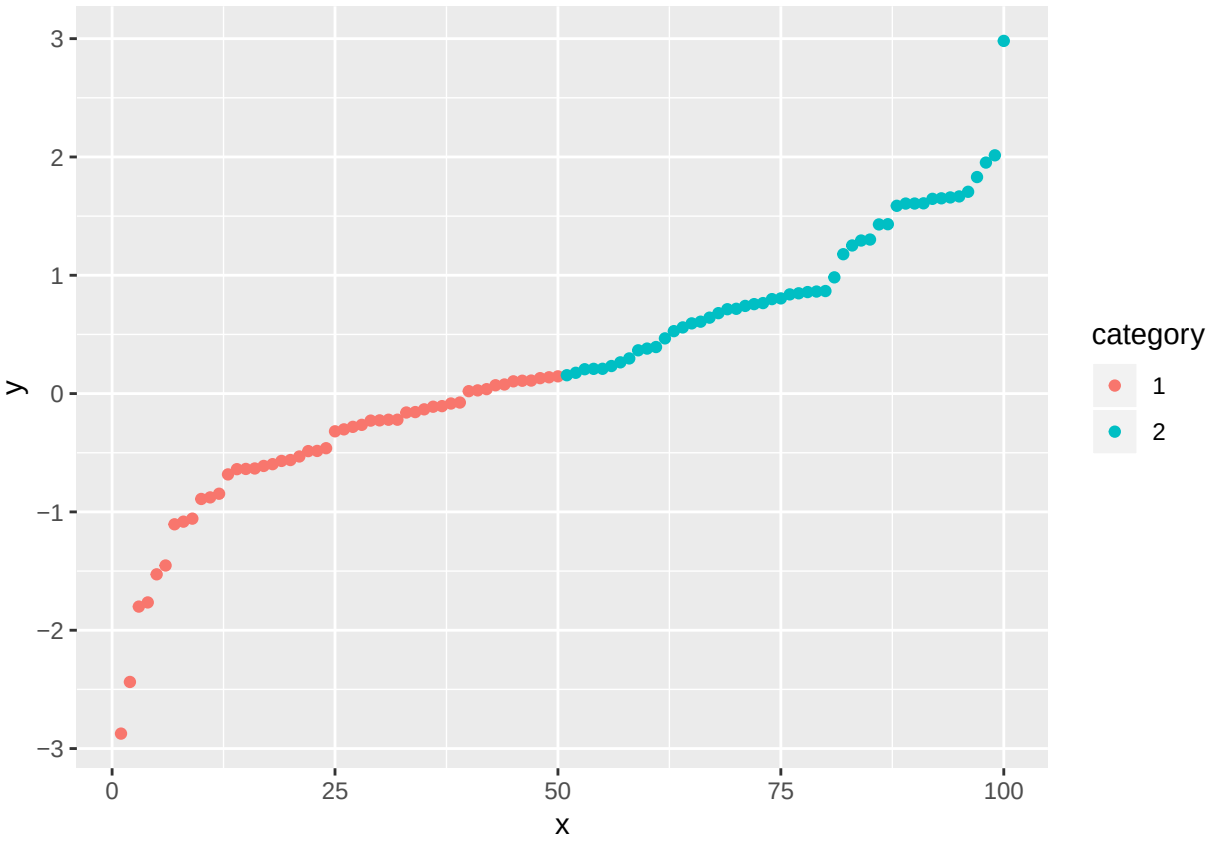
```
y <- as.factor(x)
object.size(y)
```

```
## 40560 bytes
```

```
df <- data.frame(x=1:100,y=sort(rnorm(100)),
                 category=c(rep(1,50),rep(2,50)))

ggplot(df,aes(x,y))+geom_point(aes(color=category))
```




```
df$category <- as.factor(df$category)
ggplot(df,aes(x,y))+geom_point(aes(color=category))
```



Factors can also come with plenty of problems, as you cannot do any calculations with factors and they do not sort as you would expect:

```
x <- as.factor(1:10)
mean(x)
```

```
## Warning in mean.default(x): argument is not numeric or logical: returning
## NA
```

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

Therefore, it is likely a good idea to just steer clear of factors unless we know we want them for a particular purpose. To do this we simply set stringsAsFactors=F.

```
geno <- read.csv("example.csv", header = T, row.names = 1, stringsAsFactors = F)
head(geno)
```

```
##         rs791607 rs791608 rs4236625 rs4731427 rs10244329
## NA18486       TT       TT        AA        AA         CC
## NA18487       CC       CC        TT        TT         TT
## NA18488       TT       TT        AA        AA         TT
## NA18489       AA       AA        TT        TT         TT
## NA18498       AA       AA        TT        TT         AA
## NA18499       AA       AA        TT        TT         AA
```

There are plenty of other useful read.table arguments, and even other functions altogether. However what I would consider to be the 4 major arguments of read.table have just been covered. If messing with these arguments is still not getting the data into R, try to open the data file in vi, emacs, NotePad, or textEdit. Check out the delimiter, does it change throughout the data, or those spaces or are there tabs, do some lines have more elements than others? Try to tidy up the data so each row has the same number of elements

accoring to one consistent delimiter and the data should be able to be read into R. Further command line tools such as head, tail, cut, and tr can also be very helpful.

1. Covariates

Recall in Lab 7 that the p-value calculator used an F-statistic ratio that was calculated using the variance between the estimates and the mean along with the variance between the estimates and the real phenotype values. We have seen this in lecture as:

$$SSM = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2 \qquad SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \qquad F_{[2,n-3]} = \frac{\frac{SSM}{df(M)}}{\frac{SSE}{df(E)}} = \frac{\frac{SSM}{2}}{\frac{SSE}{n-3}} \tag{1}$$

And in previous labs within the pval_calculator:

```
# Lab 7 code calculating Fstatistic according to Lecture 12 equations

# Function to calculate the pval given a set of individuals' phenotype, and genotype encodings.
pval_calculator_lab7 <- function(pheno_input, xa_input, xd_input){
    n_samples <- length(xa_input)

    X_mx <- cbind(1,xa_input,xd_input)

    MLE_beta <- ginv(t(X_mx) %*% X_mx) %*% t(X_mx) %*% pheno_input
    y_hat <- X_mx %*% MLE_beta

    SSM <- sum((y_hat - mean(pheno_input))^2)
    SSE <- sum((pheno_input - y_hat)^2)

    df_M <- 2
    df_E <- n_samples - 3

    MSM <- SSM / df_M
    MSE <- SSE / df_E

    Fstatistic <- MSM / MSE

    pval <- pf(Fstatistic, df_M, df_E,lower.tail = FALSE)

    return(pval)
}
```

We have since learned and shown with the lm() function that it is often important to model covariates that might be skewing our data. While the lm() function is a quick covariate method, it is important to understand what is going on "under the hood".

Below is a function for calculating the F-statistic based on the equations from Lecture 15. It is important to remember that this pvalue is controlling for covariates by including the covariates in the null hypothesis.

By looking at the equation below we see that the process is nearly exactly the same as above, all we are doing is changing the form of the F-statistic slightly.

$$SSE(\hat{\theta_0}) = \sum_{i=1}^{n}(y_i - \hat{y}_{i,\hat{\theta_0}})^2 \qquad SSE(\hat{\theta_1}) = \sum_{i=1}^{n}(y_i - \hat{y}_{i,\hat{\theta_1}})^2 \qquad F_{[2,n-3]} = \frac{\frac{SSE(\hat{\theta_0}) - SSE(\hat{\theta_1})}{2}}{\frac{SSE(\hat{\theta_1})}{n-3}} \tag{2}$$

```r
# New function to calculate the pval given a set of individuals' phenotype, and genotype encodings, adj
pval_calculator_lab10 <- function(pheno_input, xa_input, xd_input, z_input){
    n_samples <- length(xa_input)

    # Set up random variables for null (Z_mx) and with genotypes (XZ_mx)
    Z_mx <- cbind(1,z_input)                                    # H0 (w/ covariates)
    XZ_mx <- cbind(1,xa_input,xd_input,z_input)                 # w/ genotypes too

    # Calculate MLE betas for both null model and model with genotypes and covariates
    MLE_beta_theta0 <- ginv(t(Z_mx)  %*% Z_mx)  %*% t(Z_mx)  %*% pheno_input
    MLE_beta_theta1 <- ginv(t(XZ_mx) %*% XZ_mx) %*% t(XZ_mx) %*% pheno_input

    # Get Y estimates using the betas calculated above to give each hypothesis its best chance
    y_hat_theta0 <- Z_mx  %*% MLE_beta_theta0
    y_hat_theta1 <- XZ_mx %*% MLE_beta_theta1

    # Get the variance between the true phenotype values and our estimates under each hypothesis
    SSE_theta0 <- sum((pheno_input - y_hat_theta0)^2)
    SSE_theta1 <- sum((pheno_input - y_hat_theta1)^2)

    # Set degrees of freedom
    df_M <- 2
    df_E <- n_samples - 3

    # Put together calculated terms to get Fstatistic
    Fstatistic <- ((SSE_theta0-SSE_theta1)/df_M) / (SSE_theta1/df_E)

    # Determine pval of the Fstatistic
    pval <- pf(Fstatistic, df_M, df_E,lower.tail = FALSE)
    return(pval)
}
```

This code looks great, but that's no proof it works. Let's try it out with actual data where we generate a bunch of genotypes as Xa and Xd encodings, and then generate the phenotypes based on these encodings.

First comes the encodings and a covariate, each of which are independent:

```r
set.seed(2019)

# Set the dimensions of the data (Number of individuals and polymorphic sites to obtain genotypes for)
n_individuals = 1500
n_polymorphic_sites = 1000
# simulate genotypes as Xa and Xd encodings using HW frequencies with each allele freq=0.5
xa_sim <- matrix( sample(c(1,0,-1),
                         n_individuals*n_polymorphic_sites,
                         replace=T,
                         prob=c(0.25, 0.5, 0.25)),
                  nrow = n_individuals,
                  ncol = n_polymorphic_sites)
xd_sim <- 2*abs(xa_sim) -1
# simulate two covariates
z_sim <- cbind( sample(c(0, 1), n_individuals, replace=T, prob = c(0.5,0.5)),
                sample(c(0, 1), n_individuals, replace=T, prob = c(0.5,0.5)) )
```

Second we pick the index of the causal site, the betas that go into an equation that produces the phenotypes.

We must also include some random error in the phenotypes.

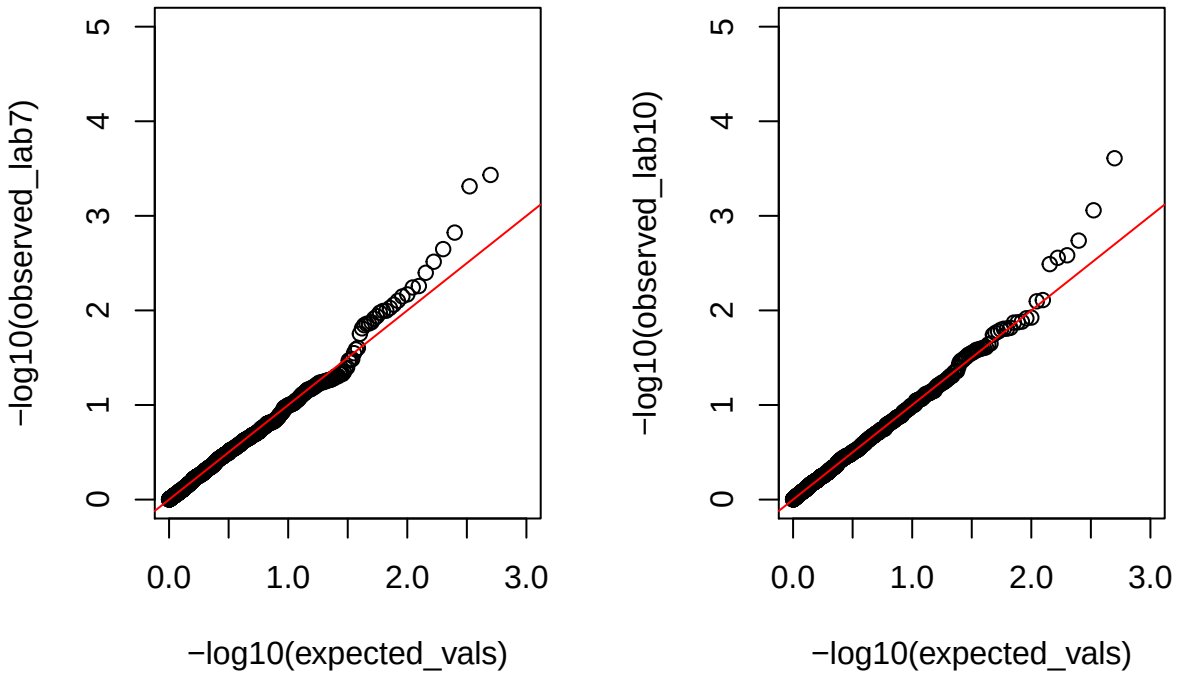
```
# simulate phenotypes based on the following true parameters
causal_site <- 150
true_betas <- c(  0.3,  0.2, -0.1,  0.9,  0.7 )  # Bmu, Ba, Bd, Bz1, Bz2, ...
epsion_sigmasq <- 1                  # sigma_sq in \epsilon=N(0,sigma_sq)
pheno <- cbind( 1, xa_sim[,causal_site], xd_sim[,causal_site], z_sim ) %*% true_betas + rnorm(n_individ
```

Below we used the simulated data to calculate pvalues for each polymorphic site using the calculators from lab7 and the new one we are introducing in this lab now.

```
# Initialize some variables and constants
pval_mx <- matrix(NA, nrow = n_polymorphic_sites, ncol=2)

# Calculate and save pvals for each phenotype-genotype pair
for (i in 1 : n_polymorphic_sites){
  pval_mx[i,1] <- pval_calculator_lab7(pheno_input = pheno,
                                       xa_input = xa_sim[,i],
                                       xd_input = xd_sim[,i])
  pval_mx[i,2] <- pval_calculator_lab10(pheno_input = pheno,
                                        xa_input = xa_sim[,i],
                                        xd_input = xd_sim[,i],
                                        z_input = z_sim)
}
```

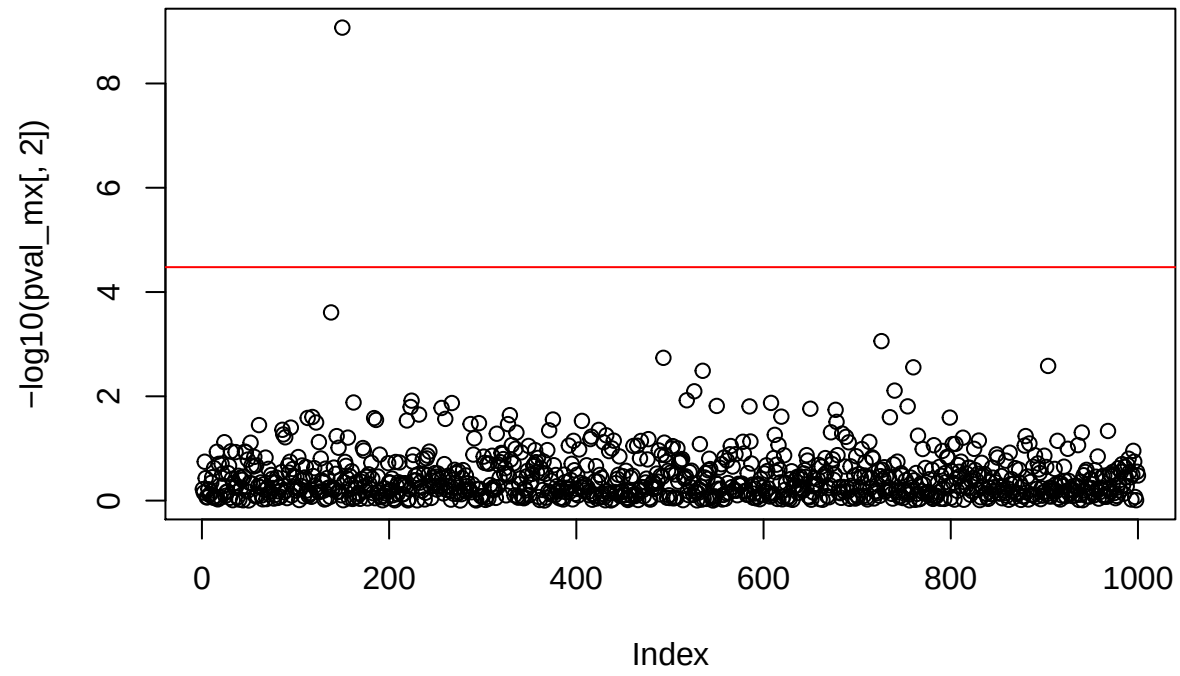
We can then compare the QQ-plots side-by-side. You can see that controlling for covariates using the new pvalue calculator tightens the data around the y=x red line, where out pvalues ideally would reside.

```
par(mfrow=c(1,2))

# Compare QQ plots
expected_vals <- seq(1/n_polymorphic_sites, 1, by=1/n_polymorphic_sites)
observed_lab7 <- sort(pval_mx[,1])
observed_lab10<- sort(pval_mx[,2])

plot(-log10(expected_vals), -log10(observed_lab7),
     ylim=c(0,5))
abline(a=0,b=1,col='red')

plot(-log10(expected_vals), -log10(observed_lab10),
     ylim=c(0,5))
abline(a=0,b=1,col='red')
```



And finally, from the manhattan plot of the new pvalue calculator, we can identify the correct causal site.

```
par(mfrow=c(1,1))
plot(-log10(pval_mx[,2]) )
abline(h=-log10(0.05/n_individuals), col='red')
```



2. Logistic Regression

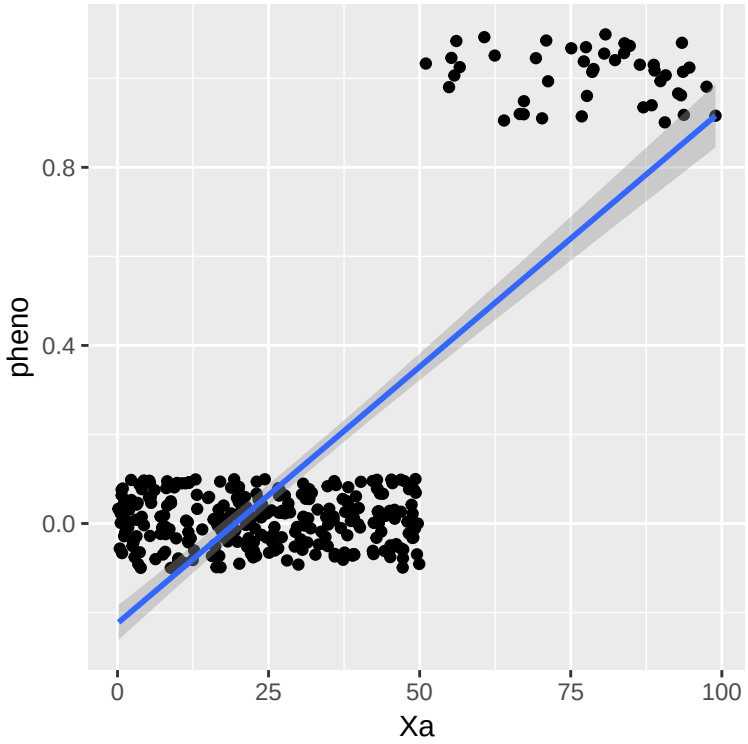
Next we will be talking about **logistic regression.** In a logistic regression, the dependent variable y (in our case phenotype) is **categorical** instead of continuous. Specifically, we are going to use logistic regression to

deal with binary phenotypes coded as 0 and 1. For example, in genome-wide association studies (GWAS) a healthy or normal control phenotype would be 0 and a disease phenotype (ex. diabetes, alzheimers, etc ...) would be 1, and the goal is to identify genomic variations that increase the probability of belonging to the disease category.

You might be wondering why we need this in the first place. So let's try to use linear regression for a binary phenotype and see what happens. First we need to generate some data, similar to before we'll randomly make some x's and generate y's from those values. The x's are coming from two distinct groups here, those below and above 50.

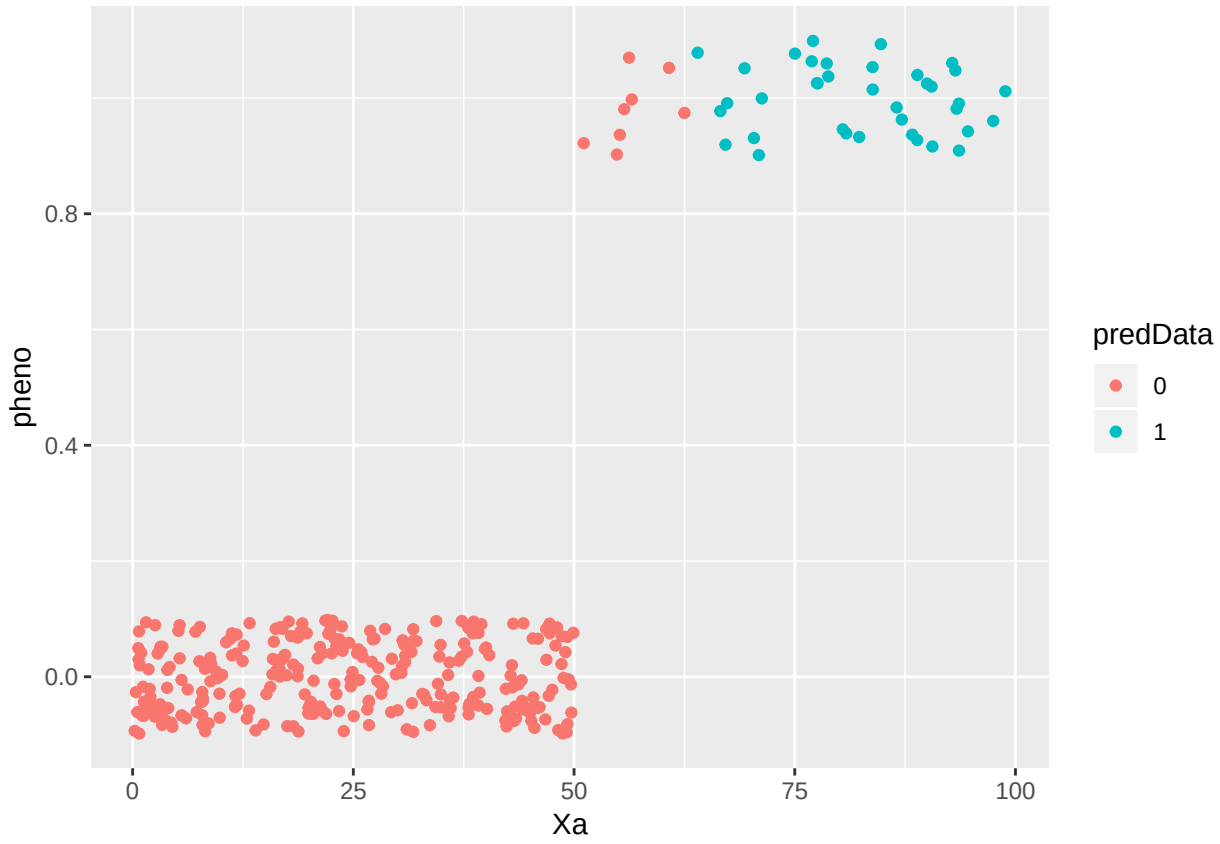
```
xa <- c(runif(275,min=0,max=50), runif(45,50,100))
y <- round(xa/100)
test_data <- data.frame("pheno" = y, "Xa" = xa)
```

Now we can calculate a linear (non-logistic model), and plot the predictions. Switching to ggplot we can do this in one line using the geom_smooth function. Also note the geom_jitter(), these values are actually on either 0 or 1.



The line looks a bit poorly fit. To test we can predict the values from each point. To do this we use the lm() function to form a linear model, just as we learned in last lab with covariats (although now there are not any covariates). Since we are using the same data to form the line and make the prediction the answer should be perfect. Let's see what we get:

```
ourModel <- lm(pheno ~ Xa, data=test_data)
new <- data.frame(Xa = test_data$Xa)
test_data$predData <- factor(round(predict(ourModel, newdata=new)))
ggplot(test_data,aes(Xa,pheno))+geom_jitter(width=0.1,height=0.1,aes(color=predData))
```



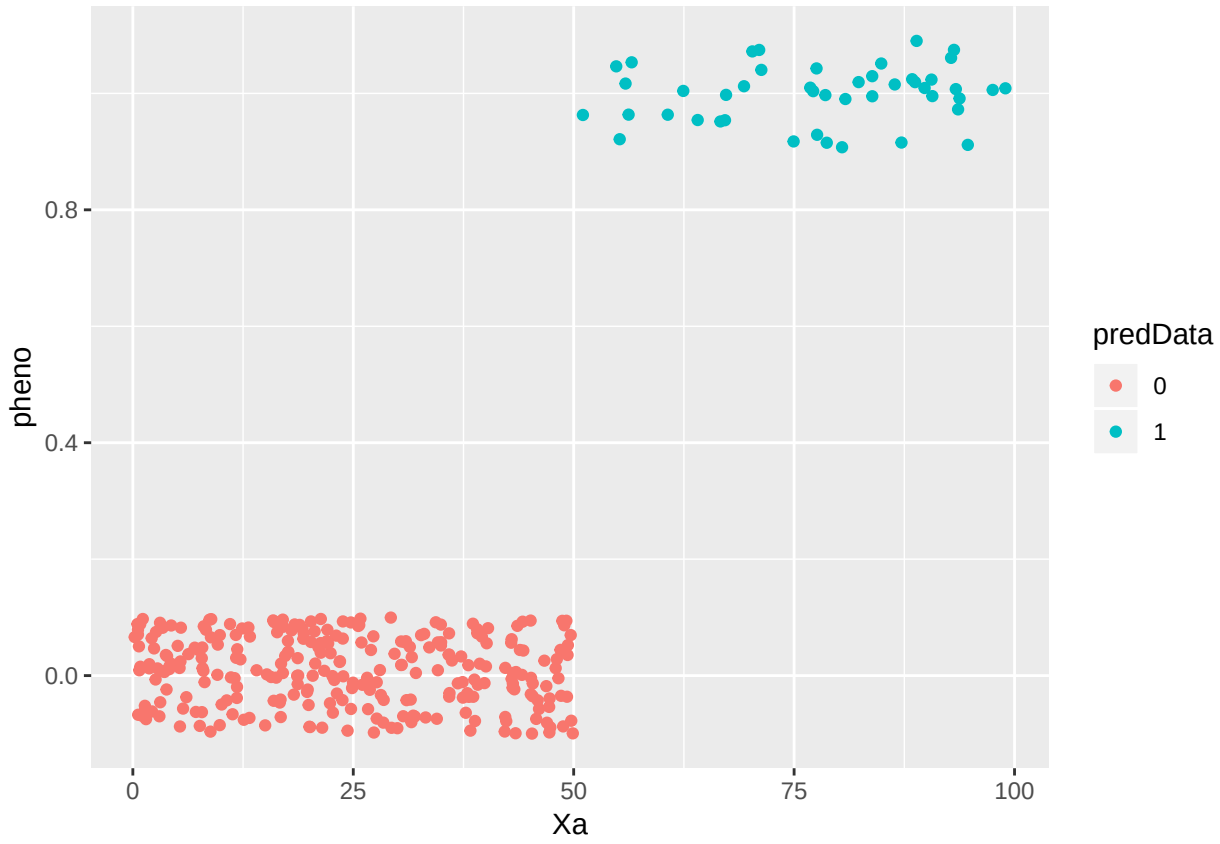
Some of the data points that should be 1 are being called as 0, that's not good. Clearly the linear model is failing us, so now we can switch over to the logistic model. In R this can easily be done by changing the linear model to a general linear model, glm(), and then specify we are looking at binomial data. The model that is created can then be treated as if it is any other model, so the rest of the prediction steps are the same.

```
ourModel <- glm(pheno ~ Xa, data=test_data, family="binomial")
```

```
## Warning: glm.fit: algorithm did not converge
```

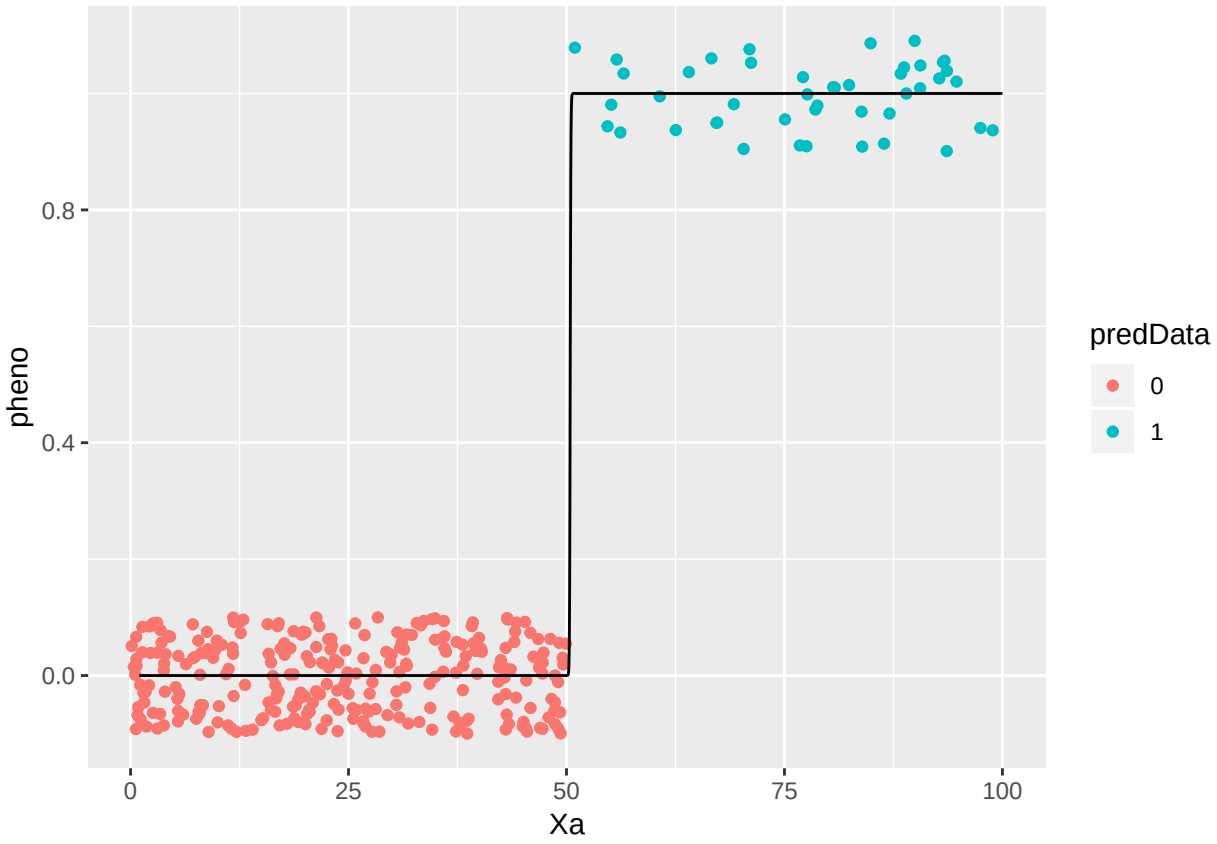
```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
new <- data.frame(Xa = test_data$Xa)
test_data$predData <- factor(round(predict(ourModel, newdata=new, type="response")))
ggplot(test_data,aes(Xa,pheno))+geom_jitter(width=0.1,height=0.1,aes(color=predData))
```



Nice! All of the data points are now correctly predicted. We can plot a line over the data to show exactly what the glm() is calculating at each point. The black outline clearly does not look like a normal line, or something that could be output from a linear regression. However, it can easily be produced with just the glm function.

```
test_data$evenSpace <- c(1,seq(45,55,length.out = nrow(test_data)-2),100)
new <- data.frame(Xa = test_data$evenSpace)
test_data$prob <- predict(ourModel, newdata=new, type="response")
ggplot(test_data,aes(Xa,pheno))+geom_jitter(width=0.1,height=0.1,aes(color=predData))+
  geom_line(aes(evenSpace,prob))
```



How does logistic regression exactly work? Simply put, logistic regression transforms the linear model that we use to "fit" the binary phenotypes. The logistic function takes the form as follows:

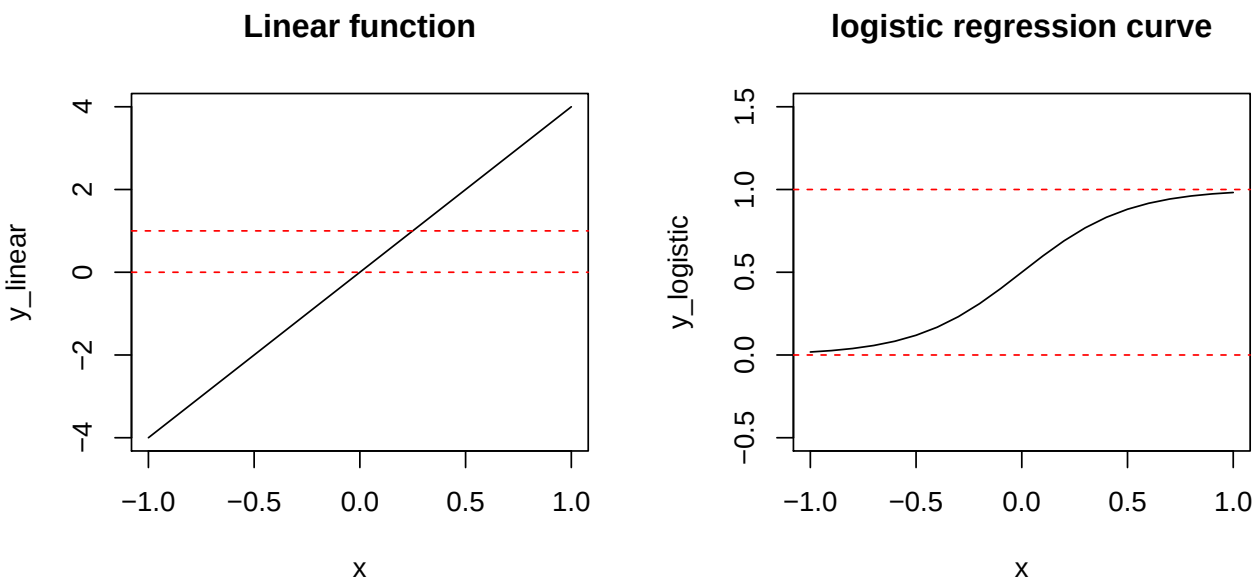
$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

So if we substitute t with the linear function that depends on x and beta values, the logistic function that we use becomes

$$F(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

A simple visualization in R might give us a better idea of how the data is transformed. Basically, logistic regression confines the original dependent values within the range of 0 and 1.

```
x <- seq(-1,1,by = 0.1)
y_linear <- x * 4
y_logistic <- 1 / ( 1 + exp(-y_linear))

par(mfrow = c(1,2))
plot(x, y_linear, main = "Linear function", type = "l")
abline(h = 0, col = "red", lty = 2)
abline(h = 1, col = "red", lty = 2)
plot(x,y_logistic, main = "logistic regression curve", type ="l", ylim = c(-0.5,1.5))
abline(h = 0, col = "red", lty = 2)
abline(h = 1, col = "red", lty = 2)
```



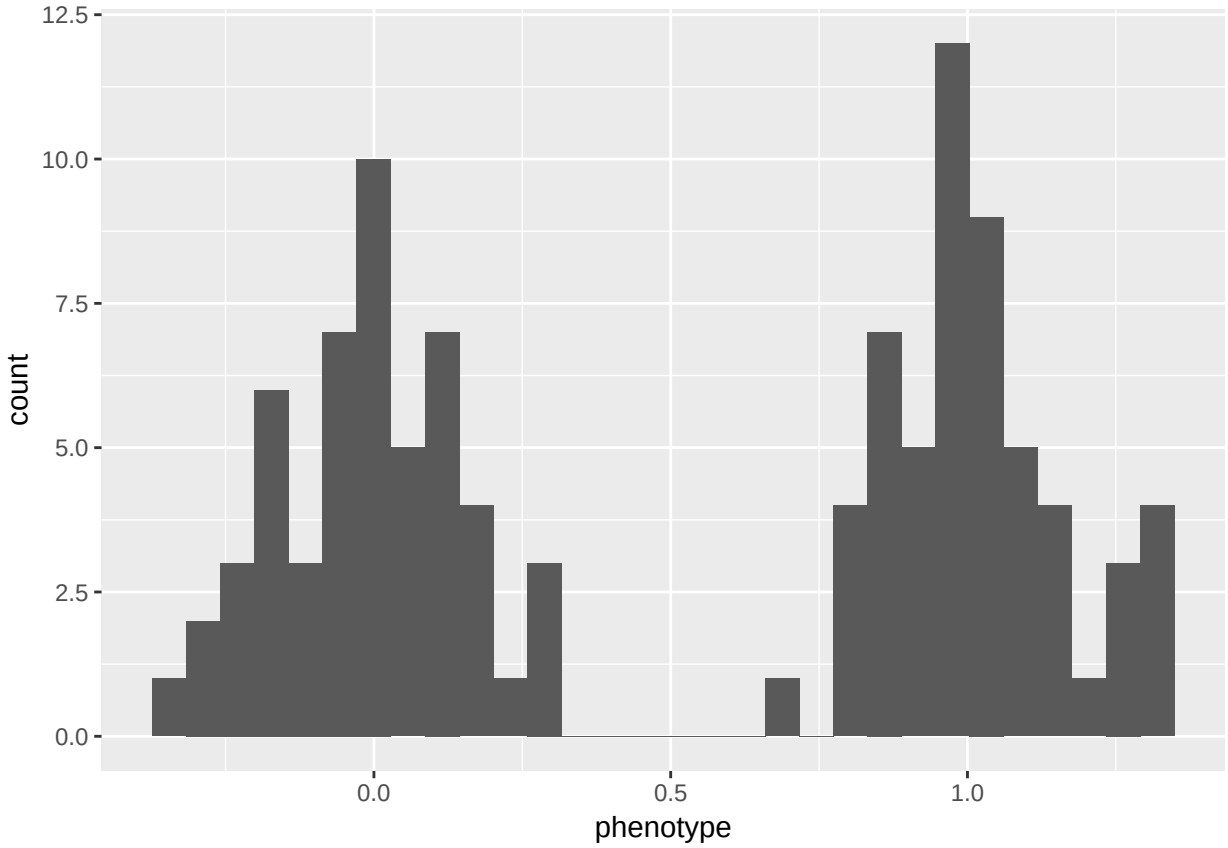
Let's project the settings in our problem onto the above equation and clarify our goal. We have a given phenotype that takes either a value of 1 or 0, and two matrices for genotypes in the form of Xa(-1,0,1) coding and Xd(-1,1) coding. Just like in linear regression, the goal is to find the values for β_μ, β_a and β_d for each genotype that best explain the relationship between the genotype and phenotype.

If the relationship was error free and the genotype value directly predicts the phenotype, we would not need logistic regression (For example, if A2A2 indicates phenotype = 1 with 100% certainty). However, that is more than often not the case in real world genetics/genomics so we would have to "soft" model the relationship between genotypes and phenotypes by using probabilities, (In other words, A2A2 has a higher chance of having phenotype=1 than phenotype=0) and that is what the transformation given in the above equation is doing.

Exercise

Let's dig into some read data, and find some genetic associations. While typically we can just jump into a regression analysis, now that we now about logistic regression, we should check if our phenotype is normally distributed, and therefore should be analyzed with linear regrssion, or binary, and should be analyzed with logistic regression.

```
pheno <- read.table("phenotypes-lab10.tsv")
geno <- read.table("genotypes-lab10.tsv")

ggplot(pheno,aes(phenotype))+geom_histogram(bins=30)
```



The phenotype sure looks binary to me. To finish off the analysis conduct a GWAS using logistic regression. The genotype file has rows of samples and columns of SNPs, like previous Xa matricies. You only need to use the Xa codings, no need to generate the Xd codings. The process is very similar to what was done 2 labs ago with the lm() function, although now use glm(). Once you have tested each variant generate a quick manhattan plot. To get everyone started, here is the code we used for the lm() example.

```
#using a for loop
#pval <- rep(0,ncol(Xa))
#for(i in 1:ncol(Xa)){
#  model <- lm(Y ~ Xa[,i])
#  pval[i] <- summary(model)$coefficients[2,4]
#}

#or in one line
#pvals <- apply(Xa.all, 2, function(x,y) summary(lm(y ~ x))$coefficients[2,4], hapmap.pheno.mx[,1])
```