BIOS707 PS5

Code

Kuei-Yueh Ko

About the notebook

This is the notebook of problem set 5 in BIOS707

Hide

```
### toolbox
library(tidyverse)
### running parallel
library(doParallel)
library(foreach)
### lasso/ridge model
library(glmnet)
### tree model
library(rpart)
library(randomForest)
### GAM
library(splines)
library(mgcv)
### forward/backward selection
library(MASS)
### plotting
library(gridExtra)
library(RColorBrewer)
```

Simulation

Q1. Optimal Functional Forms:

Different statistical learning models/algorithms solve different problems with modeling data. Therefore each algorithm will perform optimally in different scenarios. A famous paper from the early 90s referred to this as the “No Free Lunch Theorem.”

We are going to compare LASSO, GAM and CART. Consider two predictors, X_1, X_2 and an outcome Y. Simulate a relationship between X and Y where:

a. LASSO performs the best
b. GAM performs the best
c. CART performs the best

First, different outcome (y) were generated using different models. $X1$ and $X2$ were first generated from , including adding higher order terms.

Hide

```
### initialization
set.seed(0)
N  = 1000
### create x1 and x2
x1   = rnorm(N, mean = 0, sd = 10)
x2   = rnorm(N, mean = 0, sd = 10)
X    = cbind(x1, x2)
### interaction and transformation
x1x2 = x1 * x2
q = quantile(x1, probs = c(0.1, 0.5, 0.9))
x1steps = cut(x1,
    breaks = c(-Inf, q, Inf),
    labels = c(1, 2, 3, 4))
x1steps = as.numeric(xsteps)
x2steps = as.numeric(x2 < median(x2))
### modeling outcomes
lst_y = list()
lst_y$y11 = x1 + x2
lst_y$y12 = x1 + x1^2        + x2 + x2^2
lst_y$y13 = x1 + x1^2 + x1^7 + x2 + x2^2 + x2^5
lst_y$y21 = x1 + x2                             + x1x2
lst_y$y22 = x1 + x1^2        + x2 + x2^2        + x1x2
lst_y$y23 = x1 + x1^2 + x1^7 + x2 + x2^2 + x2^5 + x1x2
lst_y$y31 = x1 + x2                             + x1x2 + (x1x2)^(2) + (exp(x1x2))^(0.5)
lst_y$y32 = x1 + x1^2        + x2 + x2^2        + x1x2 + (x1x2)^(2) + (exp(x1x2))^(0.5)
lst_y$y33 = x1 + x1^2 + x1^7 + x2 + x2^2 + x2^5 + x1x2 + (x1x2)^(2) + (exp(x1x2))^(0.5)
lst_y$y41 = x1 + x2                             + x1x2 * (x2steps + x1steps)
lst_y$y42 = x1 + x1^2        + x2 + x2^2        + x1x2 * (x2steps + x1steps)
lst_y$y43 = x1 + x1^2 + x1^7 + x2 + x2^2 + x2^5 + x1x2 * (x2steps + x1steps)
### add noise to each outcome
lst_y = lapply(lst_y, function(y){
    return(y + rnorm(N, mean = 0, sd = 1))
})
### rename the outcome based on how it is constructed
names(lst_y) = c(
    "linear",
    "quadratic",
    "high_deg",

    "linear    + interact",
    "quadratic + interact",
    "high_deg  + interact",

    "linear    + complex interact",
    "quadratic + complex interact",
    "high_deg  + complex interact",

    "linear    + interact + step func",
    "quadratic + interact + step func",
    "high_deg  + interact + step func"
)
```

helper functions to perform CV

Hide

```r
### helper function
fit_models_q1 = function(x, y, method){
    # function that fit model using GAM, LASSO, and CART(TREE)
    ################################################
    if (method == "gam") {
        ### rearrange data
        dat = cbind(y, x) %>% as.data.frame
        ### fit the data using GAM
        fit = gam(y ~ s(x1) + s(x2), data = dat)
        return(fit)
    } # end if

    if (method == "lasso") {
        ### fit the data using LASSO
        fit = cv.glmnet(x = x, y = y, family = "gaussian", alpha = 1, nfolds = 10)
        return(fit)
    } # end if

    if (method == "cart") {
        ### rearrange data
        dat = cbind(y, x) %>% as.data.frame
        ### fit the data using CART
        fit = rpart(y ~ ., data = dat, control = rpart.control(cp = 0, minsplit = 2))
        return(fit)
    } # end if

    ### stop the process if wrong method is given
    stop("No match methods (should be one of 'ridge', 'lasso', and 'cart')")
} # end function fit_models_q1
fit_predict_q1 = function(fit, x, method){
    # Predict the new data x based on the given method and model
    #####################################################
    if (method == "lasso") {
        yhat = predict(fit, newx = x, s = "lambda.1se")
    } else {
        yhat = predict(fit, newdata = data.frame(x))
    } # end if-else
    return(yhat)
} # end func
fit_cv_q1 = function(X, y, K, fun_fit, fun_predict, methods){
    # Perform cross validation
    #####################################################
    ### initialization
    sp = split(1:nrow(X), 1:K)
    rss = matrix(NA, nrow = length(y), ncol = length(methods))

    ### perform cross validation (cv)
    for (k in 1:K) {

        ### split data into train & test data for cv
        x_train = X[-sp[[k]], ]
        x_test  = X[ sp[[k]], ]
        y_train = y[-sp[[k]]]
        y_test  = y[ sp[[k]]]

        ### try different methods
        for (idx in 1:length(methods)) {
            ### get the methods
            method = methods[idx]

            ### fit the model based on the specified method on k-1 fold of data
            fit  = fun_fit(x_train, y_train, method)
            yhat = fun_predict(fit, x_test, method)

            ### predict the remain fold of the data and calculate the squared error
            rss[sp[[k]], idx] = (y_test - yhat)^2
        } # end inner for loop
    } # end outer for loop

    ### return the rss matrix
    return(rss)
}
```

Perform cross validation of LASSO, GAM, and CART

Hide

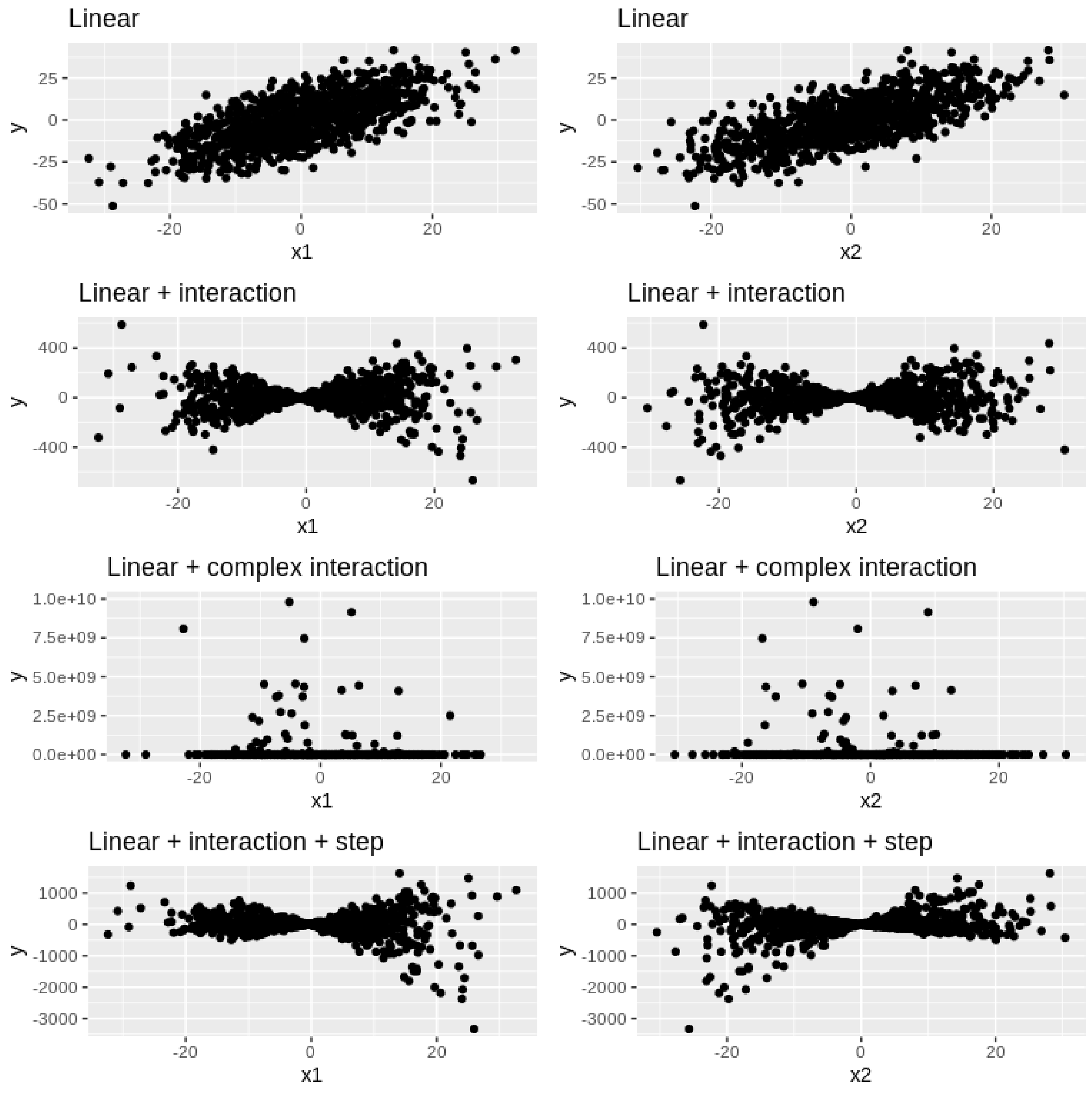
```
### initialization
methods = c("lasso","gam", "cart")
K = 5
### perform cv
lst_rss = lapply(lst_y, function(y){
    rss = fit_cv(X, y, K, fit_models_q1, fit_predict_q1, methods)
    rss = apply(rss, 2, mean)
    return(rss)
})
### arrange the results and find the
res = do.call(rbind, lst_rss) %>% data.frame
colnames(res) = methods
idx = apply(res, 1, which.min)
res$best = methods[idx]
print.data.frame(res)
```

```
                                       lasso          gam         cart  best
linear                          9.354524e-01 8.877098e-01 4.986064e+00   gam
quadratic                       3.837269e+04 1.095128e+01 2.996507e+03   gam
high_deg                        5.363863e+18 6.685844e+17 1.139389e+18   gam
linear    + interact            1.089517e+04 1.211607e+04 1.690204e+03  cart
quadratic + interact            4.744655e+04 1.226445e+04 4.741149e+03  cart
high_deg  + interact            5.363863e+18 6.685845e+17 1.024459e+18   gam
linear    + complex interact    4.947327e+17 4.958346e+17 1.155324e+18 lasso
quadratic + complex interact    4.947327e+17 4.958346e+17 1.148064e+18 lasso
high_deg  + complex interact    2.517976e+18 3.097309e+18 4.605596e+18 lasso
linear    + interact + step func 1.312650e+05 1.291051e+05 1.092410e+04  cart
quadratic + interact + step func 1.571844e+05 1.278236e+05 1.529162e+04  cart
high_deg  + interact + step func 5.363863e+18 6.685846e+17 1.059658e+18   gam
```

plot the relationship between X and Y

Hide

```
### combine the values
df = do.call(cbind, lst_y) %>% as.data.frame
df$X1 = x1
df$X2 = x2
### plot each model
g11 = ggplot(df, aes(x = x1, y = linear)) +
    geom_point() +
    labs(title = "Linear", y = "y")
g12 = ggplot(df, aes(x = x2, y = linear)) +
    geom_point() +
    labs(title = "Linear", y = "y")
g21 = ggplot(df, aes(x = x1, y = `linear    + interact`)) +
    geom_point() +
    labs(title = "Linear + interaction", y = "y")
g22 = ggplot(df, aes(x = x2, y = `linear    + interact`)) +
    geom_point() +
    labs(title = "Linear + interaction", y = "y")
g31 = ggplot(df, aes(x = x1, y = `linear    + complex interact`)) +
    geom_point() +
    ylim(c(0, 10^10)) +
    labs(title = "Linear + complex interaction", y = "y")
g32 = ggplot(df, aes(x = x2, y = `linear    + complex interact`)) +
    geom_point() +
    ylim(c(0, 10^10)) +
    labs(title = "Linear + complex interaction", y = "y")
g41 = ggplot(df, aes(x = x1, y = `linear    + interact + step func`)) +
    geom_point() +
    labs(title = "Linear + interaction + step", y = "y")
g42 = ggplot(df, aes(x = x2, y = `linear    + interact + step func`)) +
    geom_point() +
    labs(title = "Linear + interaction + step", y = "y")
### visualization
grid.arrange(g11, g12,
             g21, g22,
             g31, g32,
             g41, g42,
             nrow = 4, ncol = 2)
```



What were observed in the resutls:

- For simple polynomial function without interaction, gam works the best in cross validation
- when complex interactions are included, lasso outperform others. However, from the plot, extreme values are observed and therefore, this may affect the best methods for fitting the model.
- when introducing step function, the tree outperform others most of the time. Only when higher degree of terms were introduced did gam become better than tree.

Working with Data

For the following questions we will work off the mouse data set and use the proteins to develop a prediction model for genotype.

Preprocess: Remove NA > 10% and impuate the remained NA values using single mean imputation.

Hide

```
### import
dat_mice = read_csv("~/GitRepo/Duke_BIOS707_ML/hw/PS3/Data_Cortex_Nuclear.csv")
```

```
Parsed with column specification:
cols(
  .default = col_double(),
  MouseID = col_character(),
  Genotype = col_character(),
  Treatment = col_character(),
  Behavior = col_character(),
  class = col_character()
)
See spec(...) for full column specifications.
```

Hide

```
### seperate phenodata and exprs
idx = c("MouseID", "Genotype", "Treatment", "Behavior", "class")
dat_pheno = dat_mice %>% dplyr::select( idx)
dat_exprs = dat_mice %>% dplyr::select(-idx)
### proportion of NA values in each row
prop_na = apply(dat_exprs, 2, function(x){mean(is.na(x))})
### remove NA > 10%
idx = prop_na > 0.1
dat_exprs = dat_exprs[, !idx]
### helper function: single mean imputation
single_mean_impute = function(dat){
    dat_imputed = lapply(dat, function(x){
        mu = mean(x, na.rm = TRUE)
        x[is.na(x)] = mu
        return(x)
    })
    return(do.call(cbind, dat_imputed))
} # end func
### simple mean imputation
dat_exprs = dat_exprs %>% single_mean_impute %>% as.tibble
print(dim(dat_exprs))
```

```
[1] 1080   72
```

construct model matrix and outcome

Hide

```
model_mat = bind_cols(
    dat_pheno %>% dplyr::select(Genotype),
    dat_exprs) %>%
    as.data.frame
model_mat$Genotype = factor(
    model_mat$Genotype,
    levels = c("Control", "Ts65Dn"))
model_mat$Genotype = as.numeric(model_mat$Genotype) - 1
genotype   = model_mat$Genotype
```

Q2 Variable Selection Algorithms

(a) Create a single plot showing the coefficient estimates from the following approaches: - (i) Forward Selection - (ii) Backward Selection - (iii) Ridge Regression - (iv) LASSO Regression

fit all kinds of methods

Hide

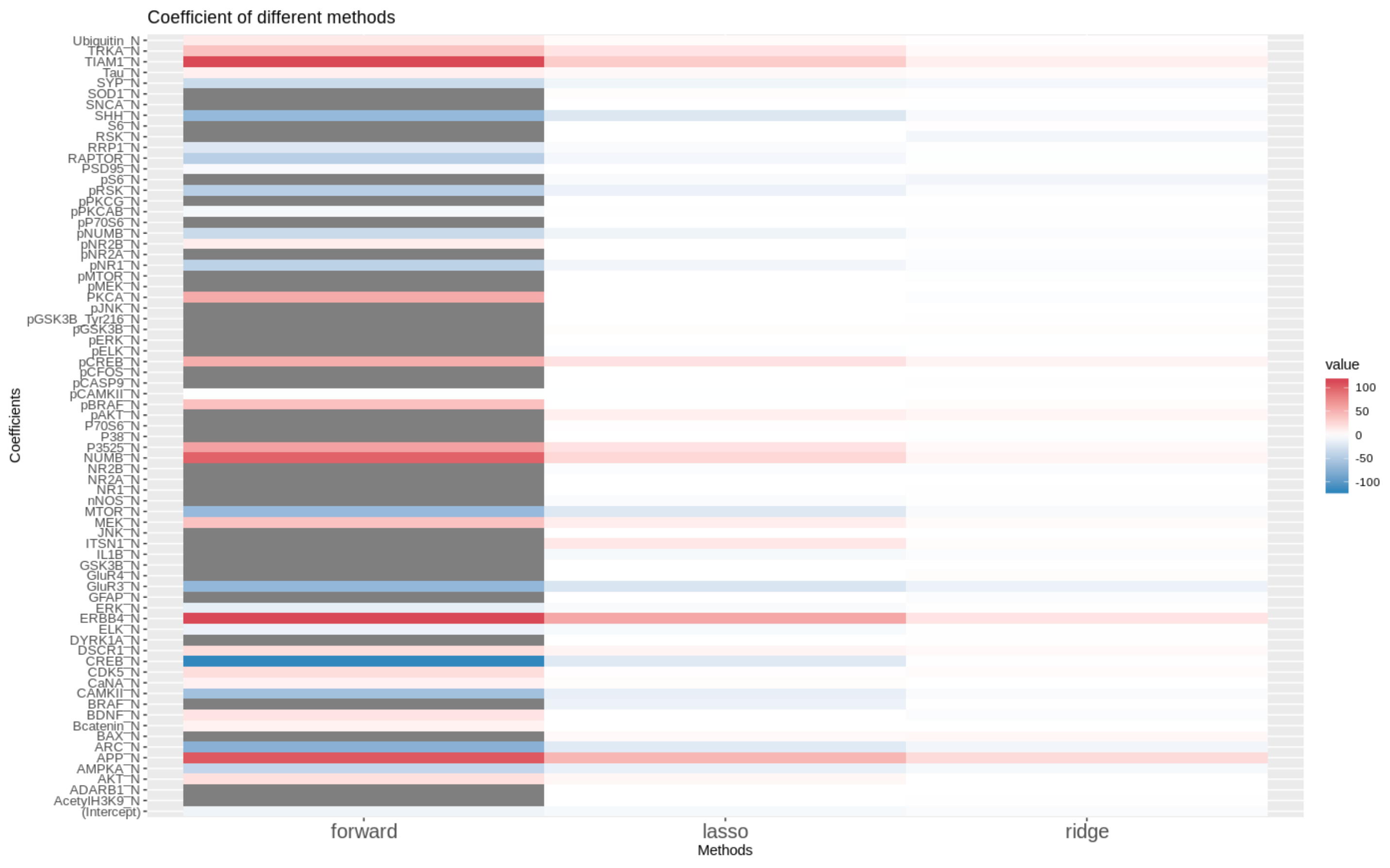
```r
### initialization for forward / backward selection
fitIntercept = glm(
    Genotype ~ 1,
    family = binomial(link = "logit"),
    data = model_mat)
fitFull      = glm(
    Genotype ~ .,
    family = binomial(link = "logit"),
    data = model_mat)
# forward selection start from the null model
sFor <- stepAIC(
    fitIntercept,
    direction = "forward",
    trace = F,
    scope = list(
        lower = fitIntercept,
        upper = fitFull))
```

arrange the coefficient of each methods

Hide

```r
### beta coef of forward selection
tmp = coef(sFor)
beta_for = data.frame(
    beta_name  = names(tmp),
    forward = as.numeric(tmp))
### beta coef of backward selection
tmp = coef(sBack)
beta_back = data.frame(
    beta_name = names(tmp),
    backward  = as.numeric(tmp))
### beta coef of lasso regression
tmp = coef(lasFit_cv, s = "lambda.1se")
beta_las = data.frame(
    beta_name = rownames(tmp),
    lasso     = as.numeric(tmp))
### beta coef of ridge regression
tmp = coef(rdgFit_cv, s = "lambda.1se")
beta_rdg = data.frame(
    beta_name = rownames(tmp),
    ridge     = as.numeric(tmp))
### combine all the coefficients
beta_all = reduce(list(beta_for, beta_las, beta_rdg), full_join, by = "beta_name")
### visualize
df = beta_all
df = df %>% gather(method, value, -beta_name)
gp = df %>% ggplot(., aes(x = method, y = beta_name, fill = value)) +
    geom_tile() +
    scale_fill_gradient2(
        low = "#3288bd", mid = "white", high = "#d53e4f",
        midpoint = 0,
        na.value = "grey50") +
    labs(title = "Coefficient of different methods", x = "Methods", y = "Coefficients") +
    theme(axis.text.x = element_text(size = 15),
          axis.text.y = element_text(size = 10))

print(gp)
```



(b) Use Cross-Validation to assess which approach performs best

Hide

```
fitIntercept = glm(
    Genotype ~ 1,
    family = binomial(link = "logit"),
    data = model_mat)

fitFull      = glm(
    Genotype ~ .,
    family = binomial(link = "logit"),
    data = model_mat)

# forward selection start from the null model
sFor <- stepAIC(
    fitIntercept,
    direction = "forward",
    trace = F,
    scope = list(
        lower = fitIntercept,
        upper = fitFull))

# backward selection start from the full model
sBack <- stepAIC(
    fitFull,
    direction = "backward",
    trace = F,
    scope = list(
        lower = fitIntercept,
        upper = fitFull))

lasFit_cv = cv.glmnet(x = as.matrix(dat_exprs), y = genotype, family = "binomial", alpha = 1, nfolds = 10)
rdgFit_cv = cv.glmnet(x = as.matrix(dat_exprs), y = genotype, family = "binomial", alpha = 0, nfolds = 10)
```

helper function to perform cross validation

Hide

```
fit_predict_q2 = function(fit, x, method) {
    # function to perform prediction in question 2
    ##########################################
    if (method == "forward" | method == "backward") {
        yhat = predict(fit, newdata = data.frame(x))
    } else {
        yhat = predict(fit, newx = as.matrix(x), s = "lambda.1se")
    } # end if-else
    return(yhat)
} # end func
fit_models_q2 = function(x, y, method, random_state){
    # function to fit the model in question 2
    ###########################################
    ### initialization
    set.seed(random_state)

    if (method == "forward") {
        ### rearrange data
        dat = cbind(y, x) %>% as.data.frame

        ### initialization
        fitIntercept = glm(
            y ~ 1,
            family = binomial(link = "logit"),
            data = dat)
        fitFull      = glm(
            y ~ .,
            family = binomial(link = "logit"),
            data = dat)

        # forward selection start from the null model
        fit <- stepAIC(
            fitIntercept,
            direction = "forward",
            trace = F,
            scope = list(
                lower = fitIntercept,
                upper = fitFull))

        return(fit)
    } # end if

    if (method == "backward") {
        ### rearrange data
        dat = cbind(y, x) %>% as.data.frame

        ### initialization
        fitIntercept = glm(
            y ~ 1,
            family = binomial(link = "logit"),
            data = dat)
        fitFull      = glm(
            y ~ .,
            family = binomial(link = "logit"),
            data = dat)

        # forward selection start from the null model
        fit <- stepAIC(
            fitFull,
            direction = "backward",
            trace = F,
            scope = list(
                lower = fitIntercept,
                upper = fitFull))

        return(fit)
    } # end if
    if (method == "lasso") {
        fit = cv.glmnet(x = as.matrix(x), y = y, family = "binomial", alpha = 1, nfolds = 10)
        return(fit)
    } # end if

    if (method == "ridge") {
        #print(method)
        fit = cv.glmnet(x = as.matrix(x), y = y, family = "binomial", alpha = 0, nfolds = 10)
        return(fit)
    } # end if

    stop("No match methods (should be one of 'forward', 'backward', 'lasso', and 'ridge'")
}
fit_cv_q2 = function(X, y, K, fun_fit, fun_predict, methods, random_state = 123){
    ### split teh data
    sp = split(1:nrow(X), 1:K)
    res = matrix(NA, nrow = length(y), ncol = length(methods))
    colnames(res) = methods
    ### perform cross validation (cv)
    for (k in 1:K) {
        cat("CV:", k, "\n")

        ### get the train test data for cv
        x_train = X[-sp[[k]], ]
        x_test  = X[ sp[[k]], ]
        y_train = y[-sp[[k]]]
        y_test  = y[ sp[[k]]]

        for (idx in 1:length(methods)) {
            ### specify the method
            method = methods[idx]

            ### fit and predict based on specified methods
            fit  = fun_fit(x_train, y_train, method, random_state)
            yhat = fun_predict(fit, x_test, method)

            ### record the squared errors
            res[sp[[k]], idx] = (y_test - yhat)^2
        } # end inner for loop
    } # end outer for loop

    return(res)
} # end func
```

perform CV on method: “forward”, “backward”, “lasso”, and “ridge”

Hide

```
methods = c("forward", "backward", "lasso", "ridge")
#rss = fit_cv_q2(dat_exprs, genotype, 5, fit_models_q2, fit_predict_q2, methods)
```

show the residual sum of squared and find the best method

Hide

```
res = apply(rss, 2, mean)
print(res)
```

```
     forward     backward        lasso        ridge
2.429229e+07 5.737296e+07 2.564134e+01 3.883799e+00
```

Hide

```
print(methods[which.min(res)])
```

```
[1] "ridge"
```

According to the results, the ridge perform the best. Note that this result may be inclucenced by the k chosen in k-fold cross validation.

Q3 Random Forests

Random Forests has two main tuning parameter: the number of variables randomly selected (mtry) and terminal node size (nodesize). Using the mouse data vary the tuning parameters to assess the impact on performance. Use the Out of Bag (OOB) Error Rate to assess performance. Grow enough trees so that the OOB-ER stabilizes.

a. fit across a grid of mtry values
b. fit across a grid of nodesize values

here I fit across mtry values and nodesize values at the same time

Hide

```
### intialization
MTRY_ALL     = seq(1, 71, by = 10)
NODESIZE_ALL = c(5, 10, 15, 20)

### combination of mtry and nodesize
params = expand.grid(mtry = MTRY_ALL, nodesize = NODESIZE_ALL)
params$error = NA

### fit random forest
rf_all = list()
for (idx in seq_len(nrow(params))){
    param  = params[idx, ]
    mtry   = param$mtry
    ndsize = param$nodesize

    rf <- randomForest(
            factor(Genotype) ~ .,
            data       = model_mat,
            importance = T,
            mtry       = mtry,
            nodesize   = ndsize,
            ntree      = 200)

    rf_all[[idx]] = rf
    params$error[idx] = mean((model_mat$Genotype - rf$votes[, 2])^2)
} # end for loop
```

show the best combination of mtry and nodesize. the best combination is mtry = 30 and nodesize = 5

Hide

```
idx = which.min(params$error)
params[idx, ]
```

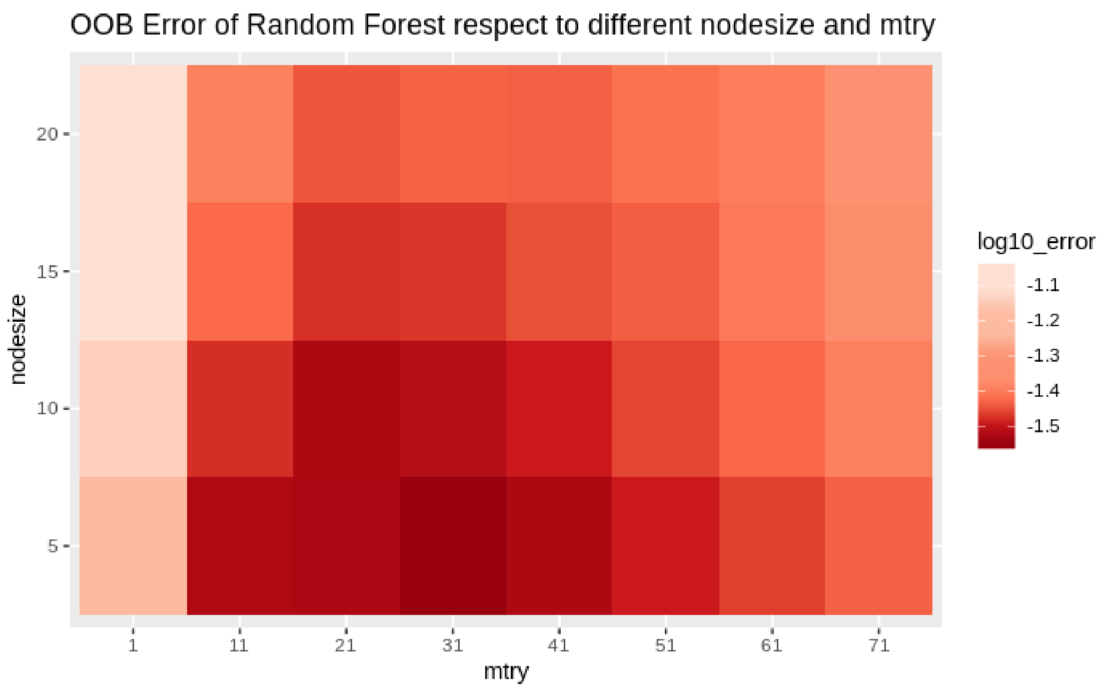
	mtry <dbl>	nodesize <dbl>	error <dbl>
4	31	5	0.02814125

1 row

visualize the out of bag error

Hide

```
df = params
df$mtry = factor(df$mtry, levels = sort(unique(df$mtry)))
df$nodesize = factor(df$nodesize, levels = sort(unique(df$nodesize)))
df$log10_error = log10(df$error)
col = brewer.pal(n = 8, name = "Reds")[c(2, 2, 3, 3, 4, 4, 5, 7, 8)]
df %>% ggplot(., aes(x = mtry, y = nodesize, fill = log10_error)) +
    geom_tile() +
    scale_fill_gradientn(colours = rev(col)) +
    labs(title = "OOB Error of Random Forest respect to different nodesize and mtry")
```



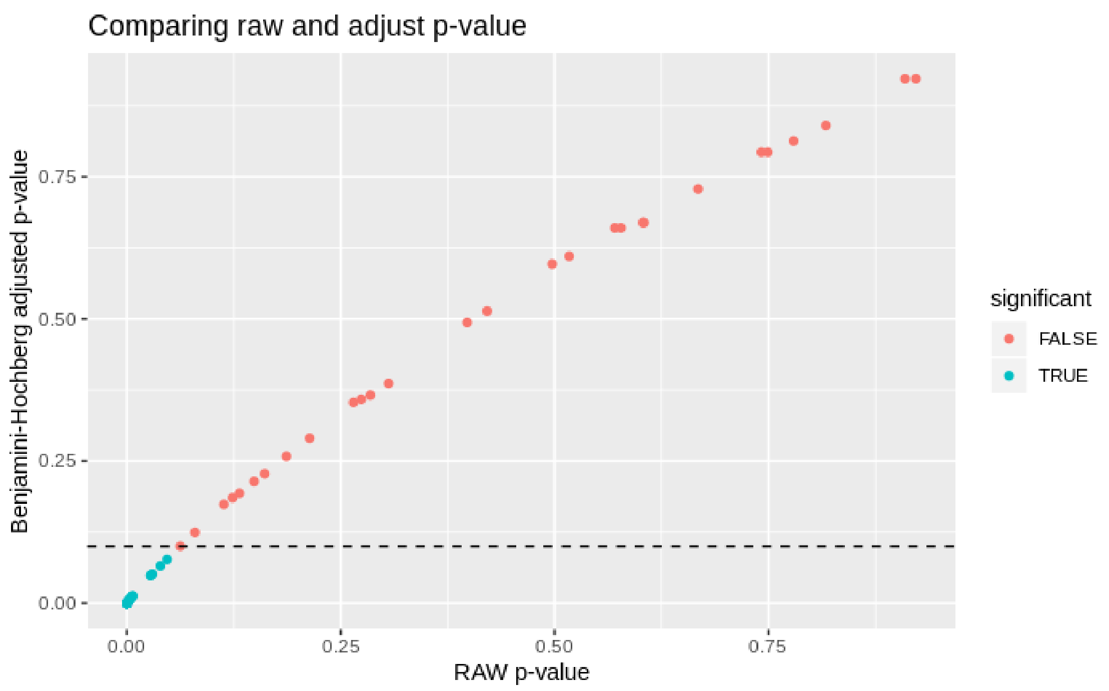
Q4 Variable Selection

An important part of ML algorithms is identifying important predictors

(a) Perform a T-test to identify variables with the top marginal association. How many are significant are if we use the Benjamini-Hochberg procedure to control the FDR at 10%

Hide

```
### get pvalue from t-test
pval = apply(dat_exprs, 2, function(x){
    tmp = split(x, genotype)
    res = t.test(tmp[[1]], tmp[[2]], alternative = "two.sided")
    return(res$p.value)
})
### combine pvalues and perform BH methods
df = data.frame(protein = colnames(dat_exprs),
                pval_raw = pval,
                pval_BH  = p.adjust(pval, method = "BH"))
df$significant = df$pval_BH < 0.1
### visualize
gp = df %>%
    ggplot(., aes(x = pval_raw, y = pval_BH, color = significant)) +
    geom_point() +
    geom_hline(yintercept = 0.1, lty = 2) +
    labs(title = "Comparing raw and adjust p-value",
         x = "RAW p-value",
         y = "Benjamini-Hochberg adjusted p-value")
print(gp)
```



choose the top 5 variables based on p adjusted value

Hide

```
#top_x_bh = df %>% arrange(pval_BH) #%>% .$protein
#print(length(top_x_bh))
#print(top_x_bh)
top_x_bh = df %>% arrange(pval_BH) %>% dplyr::top_n(5, pval_BH)
top_x_bh = as.character(top_x_bh$protein)
print(top_x_bh)
```

```
[1] "GluR4_N"    "SHH_N"      "RSK_N"      "Bcatenin_N" "PKCA_N"     "pERK_N"
```

(b) Use the results from question (2) to identify the top variables. Decide if you want to scale the predictors ahead of time

In question 02, the best one I got is ridge regression. Since I need to identify the top influenced variables, I need to scale the predictors ahead of time.

Hide

```
rdgFit_cv_scale   = cv.glmnet(
    x = scale(dat_exprs), y = genotype,
    family = "binomial", alpha = 0, nfolds = 10)
```

observe the coefficient and select top 5 variables based ont eh coefficient

Hide

```
tmp         = coef(rdgFit_cv_scale, s = "lambda.1se")
beta        = as.numeric(tmp)
names(beta) = rownames(tmp)
top_x_rdg   = names(sort(beta, decreasing = TRUE))[1:5]
print(top_x_rdg)
```

```
[1] "APP_N"   "TIAM1_N" "ITSN1_N" "TRKA_N"  "DSCR1_N"
```

(c) Using Random Forests, choose the optimal mtry and node size from (3). Calculate the both the gini and permutation importance.

fit the data using random forests

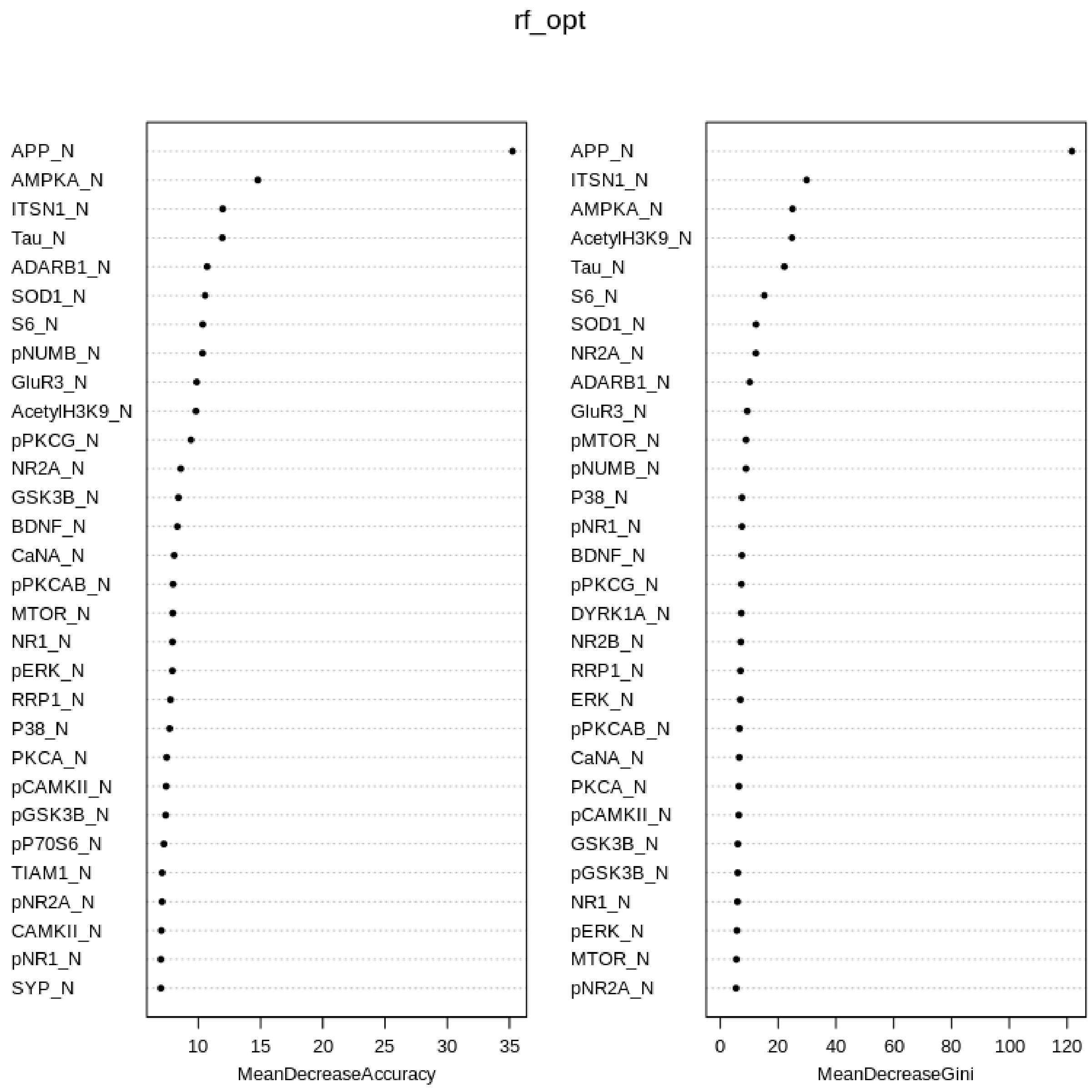
Hide

```
rf_opt <- randomForest(
            factor(Genotype) ~ .,
            data       = model_mat,
            importance = T,
            mtry       = 30,
            nodesize   = 5,
            ntree      = 200)
```

plot the importance of variables

Hide

```
varImpPlot(rf_opt, pch = 20, cex = 0.8)
```



choose the top 5 variables using the plot

Hide

```
top_x_rf = c("APP_N", "AMPKA_N", "ITSN1_N", "Tau_N", "AcetylH3K9_N")
```

(d) Do different procedures give the same or different results?

No, three methods provide different results. The top variables from ridge and random forest are more similar. Proteins APP_N and ITSN1_N are chosen from both methods.

Hide

```
cat("",
    "top protein from BH:   ", sort(top_x_bh),  "\n",
    "top protein from Ridge:", sort(top_x_rdg), "\n",
    "top protein from RF:   ", sort(top_x_rf),  "\n")
```

```
 top protein from BH:    Bcatenin_N GluR4_N pERK_N PKCA_N RSK_N SHH_N 
 top protein from Ridge: APP_N DSCR1_N ITSN1_N TIAM1_N TRKA_N 
 top protein from RF:    AcetylH3K9_N AMPKA_N APP_N ITSN1_N Tau_N 
```

Q5. Develop A Prediction Model

Develop the best prediction model for genotype

(a) Define a test set. So that everyone gets the same test set do

Hide

```
set.seed(1234)
MouseData = read_csv("~/GitRepo/Duke_BIOS707_ML/hw/PS3/Data_Cortex_Nuclear.csv")
```

```
Parsed with column specification:
cols(
  .default = col_double(),
  MouseID = col_character(),
  Genotype = col_character(),
  Treatment = col_character(),
  Behavior = col_character(),
  class = col_character()
)
See spec(...) for full column specifications.
```

Hide

```
testIndex = sample(c(1:nrow(MouseData)), trunc(.2*nrow(MouseData)),replace = F)
MouseData.Train = MouseData[-testIndex,]
MouseData.Test  = MouseData[testIndex,]
```

(b) Decide how to handle missing data I decide to use the default method:
=> Remove NA > 10% and impuate the remained NA values using single mean imputation.

Hide

```
### helper function: single mean imputation
single_mean_impute = function(dat){
    dat_imputed = lapply(dat, function(x){
        mu = mean(x, na.rm = TRUE)
        x[is.na(x)] = mu
        return(x)
    })
    return(do.call(cbind, dat_imputed))
} # end func
### help function to preprocess the data
preprocess_mouse_exprs = function(dat_mice){
    ### seperate phenodata and exprs
    idx = c("MouseID", "Genotype", "Treatment", "Behavior", "class")
    dat_pheno = dat_mice %>% dplyr::select( idx)
    dat_exprs = dat_mice %>% dplyr::select(-idx)
    ### proportion of NA values in each row
    prop_na = apply(dat_exprs, 2, function(x){mean(is.na(x))})

    ### remove NA > 10%
    idx = prop_na > 0.1
    dat_exprs = dat_exprs[, !idx]
    dat_exprs = dat_exprs %>% single_mean_impute %>% as.tibble
    return(dat_exprs)
}
### preprocess the train and test data separately
dat_train = preprocess_mouse_exprs(MouseData.Train)
dat_test  = preprocess_mouse_exprs(MouseData.Test)
y_train = factor(
    MouseData.Train$Genotype,
    levels = c("Control", "Ts65Dn"))
y_test = factor(
    MouseData.Test$Genotype,
    levels = c("Control", "Ts65Dn"))
y_train = as.numeric(y_train) - 1
y_test  = as.numeric(y_test) - 1
```

(c) Develop a prediction model. You can use any algorithm(s) you like (d) You should select your algorithm by doing cross-validation. Don’t use the test data to select your algorithm

Hide

```
fit_predict_q5 = function(fit, x, method) {
    # function to perform prediction in question 2
    ##########################################
    if (method == "forward" | method == "backward") {
        yhat = predict(fit, newdata = data.frame(x))
    }
    if (method == "randomforest"){
        yhat = predict(fit, newdata = data.frame(x))
        yhat = as.numeric(yhat) - 1
    }
    if (method == "lasso" | method == "ridge") {
        yhat = predict(fit, newx = as.matrix(x), s = "lambda.1se")
    } # end if-else
    return(yhat)
} # end func
fit_models_q5 = function(x, y, method, random_state){
    # function to fit the model in question 2
    ###########################################
    ### initialization
    set.seed(random_state)

    if (method == "forward") {
        ### rearrange data
        dat = cbind(y, x) %>% as.data.frame

        ### initialization
        fitIntercept = glm(
            y ~ 1,
            family = binomial(link = "logit"),
            data = dat)
        fitFull      = glm(
            y ~ .,
            family = binomial(link = "logit"),
            data = dat)

        # forward selection start from the null model
        fit <- stepAIC(
            fitIntercept,
            direction = "forward",
            trace = F,
            scope = list(
                lower = fitIntercept,
                upper = fitFull))

        return(fit)
    } # end if

    if (method == "backward") {
        ### rearrange data
        dat = cbind(y, x) %>% as.data.frame

        ### initialization
        fitIntercept = glm(
            y ~ 1,
            family = binomial(link = "logit"),
            data = dat)
        fitFull      = glm(
            y ~ .,
            family = binomial(link = "logit"),
            data = dat)

        # forward selection start from the null model
        fit <- stepAIC(
            fitFull,
            direction = "backward",
            trace = F,
            scope = list(
                lower = fitIntercept,
                upper = fitFull))

        return(fit)
    } # end if

    if (method == "lasso") {
        fit = cv.glmnet(x = as.matrix(x), y = y, family = "binomial", alpha = 1, nfolds = 10)
        return(fit)
    } # end if

    if (method == "ridge") {
        fit = cv.glmnet(x = as.matrix(x), y = y, family = "binomial", alpha = 0, nfolds = 10)
        return(fit)
    } # end if

    if (method == "randomforest") {
        ### rearrange data
        dat = cbind(y, x) %>% as.data.frame

        ### fit data using random forest
        fit <- randomForest(
            factor(y) ~ .,
            data       = dat,
            importance = T,
            mtry       = mtry,
            nodesize   = ndsize,
            ntree      = 500)
        return(fit)
    } # end if

    stop("No match methods (should be one of 'forward', 'backward', 'lasso', and 'ridge'")
}
fit_cv_q5 = function(X, y, K, fun_fit, fun_predict, methods, random_state = 123){
    ### split teh data
    sp = split(1:nrow(X), 1:K)
    res = matrix(NA, nrow = length(y), ncol = length(methods))
    colnames(res) = methods
    ### perform cross validation (cv)
    for (k in 1:K) {
        ### get the train test data for cv
        x_train = X[-sp[[k]], ]
        x_test  = X[ sp[[k]], ]
        y_train = y[-sp[[k]]]
        y_test  = y[ sp[[k]]]

        for (idx in 1:length(methods)) {
            ### specify the method
            method = methods[idx]

            ### fit and predict based on specified methods
            fit  = fun_fit(x_train, y_train, method, random_state)
            yhat = fun_predict(fit, x_test, method)

            ### record the squared errors
            res[sp[[k]], idx] = (y_test - yhat)^2
        } # end inner for loop
    } # end outer for loop

    return(res)
} # end func
```

perform cross validation

Hide

```
methods = c("forward", "backward", "lasso", "ridge", "randomforest")
rss = fit_cv_q5(dat_train, y_train, 5, fit_models_q5, fit_predict_q5, methods)
```

show the residual sum of squared and find the best method

Hide

```
res = apply(rss, 2, mean)
print(res)
```

```
     forward     backward        lasso        ridge randomforest
1.408438e+07 1.004864e+07 2.210675e+01 3.635289e+00 5.208333e-02
```

Hide

```
print(methods[which.min(res)])
```

```
[1] "randomforest"
```

I decided to use random forest for final model

(e) After deciding on your best algorithm, use squared-error loss to evaluate your model on the test data. Report its performance

Hide

```
### perform random forest on whole train data
x   = dat_train
y   = y_train
dat = as.data.frame(cbind(y, x))
fit_train = randomForest(
            factor(y) ~ .,
            data       = dat,
            importance = T,
            mtry       = mtry,
            nodesize   = ndsize,
            ntree      = 500)
### predict test data using the fitted model
x   = dat_test
yhat = predict(fit_train, newdata = data.frame(x))
yhat = as.numeric(yhat) - 1
### calcualte the squared error loss to evaluate the model
print(sum((y_test - yhat)^2))
```

```
[1] 9
```

There are 9 mis-classification.

(f) I will evaluate your prediction model with the test data. Submit with your write-up an R-Script and .RData object. The .RData object should contain the test data and the prediction model. The R Script should call the .RData object, perform any necessary data manipulation (i.e. transformation, imputation etc.), generate predictions on the test data and evaluate the predictions.

make sure the code works correctly

Hide

```
### helper function: single mean imputation
single_mean_impute = function(dat){
    dat_imputed = lapply(dat, function(x){
        mu = mean(x, na.rm = TRUE)
        x[is.na(x)] = mu
        return(x)
    })
    return(do.call(cbind, dat_imputed))
} # end func
### help function to preprocess the data
preprocess_mouse_exprs = function(dat_mice){
    ### seperate phenodata and exprs
    idx = c("MouseID", "Genotype", "Treatment", "Behavior", "class")
    dat_pheno = dat_mice %>% dplyr::select( idx)
    dat_exprs = dat_mice %>% dplyr::select(-idx)
    ### proportion of NA values in each row
    prop_na = apply(dat_exprs, 2, function(x){mean(is.na(x))})

    ### remove NA > 10%
    idx = prop_na > 0.1
    dat_exprs = dat_exprs[, !idx]
    dat_exprs = dat_exprs %>% single_mean_impute %>% as.data.frame
    return(dat_exprs)
}
### help function to perform transformation, imputation and prediction
fun_predict_kk319 = function(dat_test) {
    require(tidyverse)
    require(randomForest)

    dat = preprocess_mouse_exprs(dat_test)
    yhat = predict(fit_train, newdata = data.frame(dat))
    yhat = as.numeric(yhat) - 1
   return(yhat)
} # end func
### preprocess the train and test data separately
sum((y_test - fun_predict_kk319(MouseData.Test))^2)
```

```
[1] 9
```

save the objects

Hide

```
save(MouseData.Test, single_mean_impute, preprocess_mouse_exprs, fun_predict_kk319, file = "fun_predict_kk319.RData")
```