title: "Lab 7"

author: "Asher Katz"

output: pdf_document

#YARF

For the next labs, I want you to make some use of my package. Make sure you have a JDK installed first

https://www.oracle.com/java/technologies/downloads/

Then try to install rJava

```{r}
options(java.parameters = "-Xmx4000m")
pacman::p_load(rJava)
.jinit()
```

If you have error, messages, try to google them. Everyone has trouble with rJava!

If you made it past that, please try to run the following:

```{r}
if (!pacman::p_isinstalled(YARF)){
  pacman::p_install_gh("kapelner/YARF/YARFJARs", ref = "dev")
  pacman::p_install_gh("kapelner/YARF/YARF", ref = "dev", force = TRUE)
}
pacman::p_load(YARF)
```

Please try to fix the error messages (if they exist) as best as you can. I can help on slack.

#Rcpp

We will get some experience with speeding up R code using C++ via the `Rcpp` package.

First, clear the workspace and load the `Rcpp` package.

```{r}
#TO-DO
rm(list = ls())
pacman::p_load(Rcpp)
```

Create a variable `n` to be 10 and a vaiable `Nvec` to be 100 initially. Create a random vector via `rnorm` `Nvec` times and load it into a `Nvec` x `n` dimensional matrix.

```{r}
#TO-DO
n = 10
Nvec = 100
X = c()
for (i in 1:n){
  x = rnorm(Nvec)
  X = cbind(X, x)
}
dim(X)
```

Write a function `all_angles` that measures the angle between each of the pairs of vectors. You should measure the vector on a scale of 0 to 180 degrees with negative angles coerced to be positive.

```{r}
#TO-DO
```

```
all_angles = function(X){
  n = nrow(X)
  D =matrix(NA, nrow = n, ncol = n)
  for(i in 1:(n-1)){
    for(j in (i-1):n){
      x_i = X[i,]
      x_j = X[j,]
      D[i,j] = abs( ( acos( sum(x_i * x_j) / sqrt( sum(x_i^2) * sum(x_j^2) ) )) * (180/pi) )

    }
  }
  D
}
```

Plot the density of these angles.

```{r}
#TO-DO
D = all_angles(X)
pacman::p_load(ggplot2)
ggplot(data.frame(angles = c(D) ))+
  geom_density(aes(x =angles))
```

Write an Rcpp function `all_angles_cpp` that does the same thing. Use an IDE if you want, but write it below in-line.

```{r}
#TO-DO
cppFunction('
  NumericMatrix all_angles_cpp(NumericMatrix X){
    int n = X.nrow();
```

```
   int p = X.ncol();

   NumericMatrix D(n,n);

   std::fill(D.begin(), D.end, NA_REAL);


   for (int i = 0; i <(n-1); i++){

     for (int j = i + 1; j < n; j ++){


       double dot_prod = 0;

       double length_x_i_sq = 0;

       double length_x_j_sq = 0;


       for (int k = 0; k < pl j++){

         dot_prod += X(i,k) * X(j,k);

       length_x_i_sq += pow(X(i,k),2) ;

       length_x_j_sq += pow(X(j,k),2);

       }

       D(i, j) = abs(acos(dot_prod/ sqrt(length_x_i_sq * length_x_j_sq )) * (180/M_PI));

   }

  }

return D;

}

')

#go to the prac lec and find the Numeric Matrix func
```

Test the time difference between these functions for `n = 1000` and `Nvec = 100, 500, 1000, 5000` using the package `microbenchmark`. Store the results in a matrix with rows representing `Nvec` and two columns for base R and Rcpp.

```{r}

Nvecs = c(100, 500, 1000, 5000)

results_for_time = data.frame(

  Nvec = Nvecs,
```

~~~
  time_for_base_R = numeric(),
  time_for_cpp = numeric()
)
for (i in 1 : length(Nvecs)){
  X = matrix(rnorm(n * Nvecs[i]), nrow = Nvec)
  results_for_time$time_for_base_R[i] = all_angles(X)
  results_for_time$time_for_cpp[i] = all_angles_cpp(X)
}
ggplot(results_for_time) +
  geom_line(aes(x = Nvec, y = time_for_base_R), col = "red") +
  geom_line(aes(x = Nvec, y = time_for_cpp), col = "blue")
```
~~~

Plot the divergence of performance (in log seconds) over n using a line geometry. Use two different colors for the R and CPP functions. Make sure there's a color legend on your plot. We wil see later how to create "long" matrices that make such plots easier.

~~~
```{r}
#TO-DO
```
~~~

Let `Nvec = 10000` and vary `n` to be 10, 100, 1000. Plot the density of angles for all three values of `n` on one plot using color to signify `n`. Make sure you have a color legend. This is not easy.

~~~
```{r}
#TO-DO
```
~~~

Write an R function `nth_fibonnaci` that finds the nth Fibonnaci number via recursion but allows you to specify the starting number. For instance, if the sequence started at 1, you get the familiar 1, 1, 2, 3, 5, etc. But if it started at 0.01, you would get 0.01, 0.01, 0.02, 0.03, 0.05, etc.

~~~
```{r}
#TO-DO
~~~

~~~~
#TO-DO n-2 + n-1 is the next in fibonacci sequence

nthfibonacci = function(n, s= 1)

  if (n<=2){

    s

  }else{

    nthfibonacci(n-1, s) + nthfibonacci(n-2,s)

  }

nthfibonacci(6)

```


Write an Rcpp function `nth_fibonnaci_cpp` that does the same thing. Use an IDE if you want, but write it below in-line.


```{r}

#TO-DO

```


Time the difference in these functions for n = 100, 200, ...., 1500 while starting the sequence at the smallest possible floating point value in R. Store the results in a matrix.


```{r}

#TO-DO

```


Plot the divergence of performance (in log seconds) over n using a line geometry. Use two different colors for the R and CPP functions. Make sure there's a color legend on your plot.


```{r}

#TO-DO

```




# Trees, bagged trees and random forests
~~~~

You can use the `YARF` package if it works, otherwise, use the `randomForest` package (the standard).

Let's take a look at a simulated sine curve. Below is the code for the data generating process:

~~~
```{r}
pacman::p_load(caret)
pacman::p_load(randomForest)
rm(list = ls())
n = 500
sigma = 0.3
x_min = 0
x_max = 10
f_x = function(x){sin(x)}
y_x = function(x, sigma){f_x(x) + rnorm(n, 0, sigma)}
x_train = runif(n, x_min, x_max)
y_train = y_x(x_train, sigma)
```
~~~

Plot an example dataset of size 500:

~~~
```{r}
pacman::p_load(ggplot2)
#TO-DO
ggplot(data.frame(x=x_train, y=y_train))+
  geom_point(aes(x=x, y=y))
```
~~~

Create a test set of size 500 as well

~~~
```{r}
#TO-DO
~~~

```
x_test = runif(500, x_min, x_max)

y_test = y_x(x_test,sigma)

ggplot(data.frame(x=x_train, y=y_train))+

  geom_point(aes(x=x, y=y))
```

Locate the optimal node size hyperparameter for the regression tree model. I believe you can use `randomForest` here by setting `ntree = 1`, `replace = FALSE`, `sampsize = n` (`mtry` is already set to be 1 because there is only one feature) and then you can set `nodesize`. Plot nodesize by out of sample s_e. Plot.

```{r}
#TO-DO


nodeSizes = 1:n # 500 length vec


results = matrix(NA, nrow = n, ncol = 2) # the results of a tree with every nodesize from 1 - 500, a length(nodeSizes) X 2
matrix


for (i in 1:n){

  nodeSize = nodeSizes[i]

  g = randomForest(data.frame(x = x_train), y = y_train, ntree = 1, replace = FALSE, sampsize = n, nodesize = nodeSize) #
make a tree with

  yhat_test = predict(g, data.frame(x = x_test))

  results[i,] = c(nodeSize, sd(y_test - yhat_test)) # results are node size, real outcomes -  tree with node[i]  test results

}

results[order(results[,2]),][1,] # once ordered, print the first result


ggplot(data.frame(nodesize= results[,1] , oos_error=results[,2]))+

  geom_point(aes(x=nodesize, y=oos_error))
```

Plot the regression tree model g(x) with the optimal node size.

```{r}

#TO-DO

g_optimal = randomForest(data.frame(x = x_train), y = y_train, ntree = 1, replace = FALSE, sampsize = n, nodesize = results[order(results[,2]),][1,1] )


x_predict = data.frame(x = seq(0, x_max, length.out = n))

g = predict(g_optimal, x_predict)

ggplot(data.frame(x = x_predict, y = g), aes(x, y)) +

  geom_point(lwd = 3) +

  geom_point(aes(x, y), data.frame(x = x_predict, y = g), col = "blue")

```

Provide the bias-variance decomposition of this DGP fit with this model. It is a lot of code, but it is in the practice lectures. If your three numbers don't add up within two significant digits, increase your resolution.

```{r}

#TO-DO

xmin = 0

xmax = 5

n_train = 20

n_test = 1000

sigma = 1 ###### this squared is the irreducible error component through all of the bias-variance decomp formulas

f = function(x){predict(g_optimal, data.frame(x=x))}

Nsim = 1000

training_gs = matrix(NA, nrow = Nsim, ncol = 2)

x_trains = matrix(NA, nrow = Nsim, ncol = n_train)

y_trains = matrix(NA, nrow = Nsim, ncol = n_train)

all_oos_residuals = matrix(NA, nrow = Nsim, ncol = n_test)

for (nsim in 1 : Nsim){

  #simulate dataset $\mathbb{D}$

  x_train = runif(n_train, xmin, xmax)

  delta_train = rnorm(n_train, 0, sigma) #Assumption I: mean zero and Assumption II: homoskedastic
```

~~~
  y_train = f(x_train) + delta_train
  x_trains[nsim, ] = x_train
  y_trains[nsim, ] = y_train

  #fit a model g | x's, delta's and save it
  g_model = lm(y_train ~ ., data.frame(x = x_train))
  training_gs[nsim, ] = coef(g_model)

  #generate oos dataset according to the same data generating process (DGP)
  x_test = runif(n_test, xmin, xmax)
  delta_test = rnorm(n_test, 0, sigma)
  y_test = f(x_test) + delta_test
  #predict oos using the model and save the oos residuals
  y_hat_test = predict(g_model, data.frame(x = x_test))
  all_oos_residuals[nsim, ] = y_test - y_hat_test
}



resolution = 10000 #how many xstars to test
x = seq(xmin, xmax, length.out = resolution)
f_x_df = data.frame(x = x, f = f(x))
ggplot(f_x_df, aes(x, f)) +
  geom_line(col = "green") +
  geom_point(aes(x, y), data = data.frame(x = x_trains[1, ], y = y_trains[1, ]))
```


```{r}
rm(list = ls())
```
~~~

Take a sample of n = 2000 observations from the diamonds data.

```{r}

#TO-DO

n_train = 2000

training_indices = sample(1 : nrow(diamonds), n_train)

diamonds_train = diamonds[training_indices, ]

y_train = diamonds_train$price

X_train = diamonds_train

X_train$price = NULL

```

Find the bootstrap s_e for a RF model using 1, 2, 5, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500, 1000 trees. If you are using the `randomForest` package, you can calculate oob residuals via `e_oob = y_train - rf_mod$predicted`. Plot.

```{r}

#TO-DO


num_trees = cbind(1, 2, 5, 10, 20, 30, 40, 50, 100, 200)

oob_s_e = 1:length(num_trees)

results = matrix(NA, nrow = length(num_trees), ncol = 2)

for (i in 1:length(num_trees)){

  rf_mod = randomForest(data.frame( x = X_train), y = y_train, ntree = num_trees[i], replace = FALSE)

  #e_oob[i] = y_train - rf_mod$predicted

  #YARFBAG(data.frame(x = x_train), y_train, num_trees = i)

  oob_s_e[i] = sqrt(rf_mod$mse)

  results[i,] = c(num_trees[i], oob_s_e[i])


  g = predict(rf_mod, data.frame(X_train))

ggplot(data.frame(x = X_trains, y = g), aes(x, y)) +

  geom_point(lwd = 3) +

  geom_point(aes(x, y), data.frame(x = X_train, y = g), col = "blue")
```

~~~
}
results

```

Using the diamonds data, find the oob s_e for a bagged-tree model using 1, 2, 5, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500, 1000 trees. If you are using the `randomForest` package, you can create the bagged tree model via setting an argument within the RF constructor function. Plot.

```{r}
#TO-DO
num_trees = cbind(1, 2, 5, 10, 20, 30, 40, 50, 100, 200)
oob_s_e = 1:length(num_trees)
results2 = matrix(NA, nrow = length(num_trees), ncol = 2)
for (i in 1:length(num_trees)){
  bagged_tree_mod = randomForest(data.frame(x = X_train), y = y_train, ntree = num_trees[i], replace = TRUE)
  #e_oob[i] = y_train - bagged_tree_mod$predicted
  #YARFBAG(data.frame(x = x_train), y_train, num_trees = i)
  oob_s_e[i] = sqrt(bagged_tree_mod$mse)
  results2[i,] = c(num_trees[i], oob_s_e[i])

  g = predict(rf_mod, data.frame(X_train))
ggplot(data.frame(x = X_trains, y = g), aes(x, y)) +
  geom_point(lwd = 3) +
  geom_point(aes(x, y), data.frame(x = X_train, y = g), col = "blue")
}
results2
```

What is the percentage gain / loss in performance of the RF model vs bagged trees model?

```{r}
~~~

```
#TO-DO

100 * (results2[1,] - results[1,])/ results[1,]

```

Why was this the result?

#

because bagged trees have a little benefit for free

Plot oob s_e by number of trees for both RF and bagged trees.

```{r}
  g = predict(rf_mod, data.frame(X_train))
ggplot(data.frame(x = results2[1,] , y = num_trees), aes(x, y)) +
  geom_point(lwd = 3) +
  geom_point(aes(x, y), data.frame(x = result[1,], y = num_trees), col = "blue")



  g = predict(bagged_tree_mod, data.frame(X_train))
ggplot(data.frame(x = results2[1,] , y = num_trees), aes(x, y)) +
  geom_point(lwd = 3) +
  geom_point(aes(x, y), data.frame(x = result[1,], y = num_trees), col = "blue")
}
```

Build RF models for 500 trees using different `mtry` values: 1, 2, ... the maximum. That maximum will be the number of features assuming that we do not binarize categorical features if you are using `randomForest` or the number of features assuming binarization of the categorical features if you are using `YARF`. Calculate oob s_e for all mtry values. Plot.

```{r}
#TO-DO
```

```
for (i in 1:length(x_test){
 rf_mod2 = randomForest(data.frame( x = X_train), y = y_train, ntree = 500, replace = FALSE, mtry = [i] )

  #e_oob[i] = y_train - rf_mod$predicted
  #YARFBAG(data.frame(x = x_train), y_train, num_trees = i)
  oob_s_e[i] = sqrt(rf_mod2$mse)
  results[i,] = c(num_trees[i], oob_s_e[i])

  g = predict(rf_mod, data.frame(X_train))
ggplot(data.frame(x = X_trains, y = g), aes(x, y)) +
  geom_point(lwd = 3) +
  geom_point(aes(x, y), data.frame(x = X_train, y = g), col = "blue")
}
results


```

Plot oob s_e by mtry.

```{r}
#TO-DO
  g = predict(rf_mod, data.frame(X_train))
ggplot(data.frame(x = oob_s_e, y = rf_mod$mtry), aes(x, y)) +
  geom_point(lwd = 3) +
  geom_point(aes(x, y), data.frame(x = oob_s_e, y = rf_mod$mtry), col = "blue")
```

```{r}
rm(list = ls())
```

Take a sample of n = 2000 observations from the adult data.

```{r}
#TO-DO
```

Using the adult data, find the bootstrap misclassification error for an RF model using 1, 2, 5, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500, 1000 trees.

```{r}
#TO-DO
```

Using the adult data, find the bootstrap misclassification error for a bagged-tree model using 1, 2, 5, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500, 1000 trees. Plot.

```{r}
#TO-DO
```

What is the percentage gain / loss in performance of the RF model vs bagged trees model?

```{r}
#TO-DO
```

Plot bootstrap misclassification error by number of trees for both RF and bagged trees.

```{r}
#TO-DO
```

Build RF models for 500 trees using different `mtry` values: 1, 2, ... the maximum (see above as maximum is defined by the specific RF algorithm implementation). Plot.

~~~
```{r}
#TO-DO
```
~~~

Plot bootstrap misclassification error by `mtry`.

~~~
```{r}
#TO-DO
```
~~~

~~~
```{r}
rm(list = ls())
~~~