# Group Assignment 1: Breast Cancer Analysis using KNN

*Aylin Kosar, Surya Aenuganti Ushasri, Salma Olmai, Nicolas Romero, Viraj Prasad Sapre*

*October 16, 2018*

### Upload Data

*Surya*:

```r
#setwd("~/Fall 2018 R Data Files")
#Load the data from a csv file
cancer = read.csv("wisc_bc_data.csv", na.strings = NULL)
```

### Check Data

*Surya*:

```r
#Check the structure of the data
str(cancer)
```

```
## 'data.frame':    569 obs. of  32 variables:
##  $ id               : int  87139402 8910251 905520 868871 9012568 906539 925291 87880 862989 89827 .
##  $ diagnosis        : Factor w/ 2 levels "B","M": 1 1 1 1 1 1 1 2 1 1 ...
##  $ radius_mean      : num  12.3 10.6 11 11.3 15.2 ...
##  $ texture_mean     : num  12.4 18.9 16.8 13.4 13.2 ...
##  $ perimeter_mean   : num  78.8 69.3 70.9 73 97.7 ...
##  $ area_mean        : num  464 346 373 385 712 ...
##  $ smoothness_mean  : num  0.1028 0.0969 0.1077 0.1164 0.0796 ...
##  $ compactness_mean : num  0.0698 0.1147 0.078 0.1136 0.0693 ...
##  $ concavity_mean   : num  0.0399 0.0639 0.0305 0.0464 0.0339 ...
##  $ points_mean      : num  0.037 0.0264 0.0248 0.048 0.0266 ...
##  $ symmetry_mean    : num  0.196 0.192 0.171 0.177 0.172 ...
##  $ dimension_mean   : num  0.0595 0.0649 0.0634 0.0607 0.0554 ...
##  $ radius_se        : num  0.236 0.451 0.197 0.338 0.178 ...
##  $ texture_se       : num  0.666 1.197 1.387 1.343 0.412 ...
##  $ perimeter_se     : num  1.67 3.43 1.34 1.85 1.34 ...
##  $ area_se          : num  17.4 27.1 13.5 26.3 17.7 ...
##  $ smoothness_se    : num  0.00805 0.00747 0.00516 0.01127 0.00501 ...
##  $ compactness_se   : num  0.0118 0.03581 0.00936 0.03498 0.01485 ...
##  $ concavity_se     : num  0.0168 0.0335 0.0106 0.0219 0.0155 ...
##  $ points_se        : num  0.01241 0.01365 0.00748 0.01965 0.00915 ...
##  $ symmetry_se      : num  0.0192 0.035 0.0172 0.0158 0.0165 ...
##  $ dimension_se     : num  0.00225 0.00332 0.0022 0.00344 0.00177 ...
##  $ radius_worst     : num  13.5 11.9 12.4 11.9 16.2 ...
##  $ texture_worst    : num  15.6 22.9 26.4 15.8 15.7 ...
##  $ perimeter_worst  : num  87 78.3 79.9 76.5 104.5 ...
##  $ area_worst       : num  549 425 471 434 819 ...
##  $ smoothness_worst : num  0.139 0.121 0.137 0.137 0.113 ...
##  $ compactness_worst: num  0.127 0.252 0.148 0.182 0.174 ...
##  $ concavity_worst  : num  0.1242 0.1916 0.1067 0.0867 0.1362 ...
##  $ points_worst     : num  0.0939 0.0793 0.0743 0.0861 0.0818 ...
##  $ symmetry_worst   : num  0.283 0.294 0.3 0.21 0.249 ...
```

```
##  $ dimension_worst  : num  0.0677 0.0759 0.0788 0.0678 0.0677 ...
```

### Organize Data

*Surya*:

```r
table(cancer$diagnosis)
```

```
## 
##   B   M 
## 357 212
```

```r
#Check whether there is any missing data
sum(is.na(cancer$diagnosis))
```

```
## [1] 0
```

```r
#Clearly this should be a factor hence converting it to a factor and labeling the levels to benign or m
cancer$diagnosis <- factor(cancer$diagnosis, levels = c("B", "M"), labels = c("benign","malignant"))
levels(cancer$diagnosis)
```

```
## [1] "benign"    "malignant"
```

*Surya*: Since we need to measure distances to classify them according to knn, we need the variables to have numerical values on a same scale. So we normalize the variables. Here we are creating a function so that we can apply the normalization to all columns using this single function.

```r
normalize = function(x) {
    y = (x - min(x))/(max(x) - min(x))
    y
}
```

```r
#Applying the above normalization function to all columns except the first 2
#so lapply is a function in r where can specify the function to apply and the columns on which we have
can_n_L <- lapply(cancer[, 3:32], normalize)
#converting the data to a data frame[Since wbcd_n_L consist of only data from #3- 32 columns]
can_n <- data.frame(can_n_L)

can_n[1:3, 1:4]
```

```
##   radius_mean texture_mean perimeter_mean  area_mean
## 1   0.2526859    0.0906324      0.2422777 0.13599152
## 2   0.1712812    0.3124789      0.1761454 0.08606575
## 3   0.1921056    0.2407846      0.1874784 0.09743372
```

```r
rownames(can_n) <- cancer$id
```

```r
#Isolate the class labels and name them accordingly
BM_class <- cancer[, 2]
#names(BM_class)-> this would give null because there are no labels yet because #there #are no attribut
names(BM_class) <- cancer$id

BM_class[1:3]
```

```
## 87139402  8910251   905520 
##   benign   benign   benign 
## Levels: benign malignant
```

```r
#so now each label comes under an attribute which is actually the row name/id
#imagine a single row but 569 of attributes
```

### Creating training and test (validation) datasets

*Surya*:

```
nrow(cancer)
```

```
## [1] 569
```

```
rand_permute <- sample(x = 1:569, size = 569)

rand_permute[1:5]
```

```
## [1] 500 164 100 168 414
```

```
# save(rand_permute, file='rand_permute.RData')
```

```
#load("rand_permute.RData")
```

```
#randomly permute the rows of data
all_id_random = cancer[rand_permute, "id"]

# Select the first third of these for validation

569/3
```

```
## [1] 189.6667
```

```
#Get the first 1/3 ids of the data and keep it for validation
validate_id <- as.character(all_id_random[1:189])
#Get the next 2/3 ids of the data and keep it for training
training_id <- as.character(all_id_random[190:569])
```

*Surya*: Subset the data by taking the data of the respective ids

```
can_train <- can_n[training_id, ]

can_val <- can_n[validate_id, ]

BM_class_train <- BM_class[training_id]

BM_class_val <- BM_class[validate_id]

table(BM_class_train)
```

```
## BM_class_train
##    benign malignant
##       240       140
```

```
table(BM_class_val)
```

```
## BM_class_val
##    benign malignant
##       117        72
```

### Executing knn

*Surya*:

```
library(class)
```

```r
`?`(knn)
```

```
## starting httpd help server ... done
```

```r
sqrt(nrow(can_train))
```

```
## [1] 19.49359
```

```r
k = 19
```

```r
#Fitting the model and validating against test set
knn_predict = knn(can_train, can_val, BM_class_train, k = 19)

knn_predict[1:3]
```

```
## [1] malignant benign    malignant
## Levels: benign malignant
```

```r
#Check the confusion matrix for true positives and true negatives
table(knn_predict, BM_class_val)
```

```
##            BM_class_val
## knn_predict benign malignant
##   benign       116         7
##   malignant      1        65
```

```r
prop.table(table(knn_predict, BM_class_val))
```

```
##            BM_class_val
## knn_predict      benign   malignant
##   benign    0.613756614 0.037037037
##   malignant 0.005291005 0.343915344
```

It really depends on the randomly selectd data for testing and validating

### Testing different values of k

**Aylin**: The knn numerical value are given random variables to predict the outcome of the train set. The first is considered the best.

```r
knn_predict_3 = knn(can_train, can_val, BM_class_train, k = 3)

knn_predict_7 = knn(can_train, can_val, BM_class_train, k = 7)

knn_predict_11 = knn(can_train, can_val, BM_class_train, k = 11)

knn_predict_31 = knn(can_train, can_val, BM_class_train, k = 31)
```

```r
table(knn_predict_3, BM_class_val)
```

```
##              BM_class_val
## knn_predict_3 benign malignant
##     benign       114         3
##     malignant      3        69
```

```r
table(knn_predict_7, BM_class_val)
```

```
##              BM_class_val
## knn_predict_7 benign malignant
##     benign       115         2
```

```
##     malignant      2        70
```

```r
table(knn_predict_11, BM_class_val)
```

```
##               BM_class_val
## knn_predict_11 benign malignant
##      benign       116         4
##      malignant      1        68
```

```r
table(knn_predict_31, BM_class_val)
```

```
##               BM_class_val
## knn_predict_31 benign malignant
##      benign       116         8
##      malignant      1        64
```

### Study significance of the variables

**Aylin**: Below the names of the data set are listed. A model is binomial regression model is created. The original name of the model was lm_1 but I renamed it g1 to make it easier. "can" is also substituted for "wbcd". The names of the model is created in the second code after the g1 model is created. Then the F statitic is created.

```r
names(can_train)
```

```
##  [1] "radius_mean"       "texture_mean"      "perimeter_mean"
##  [4] "area_mean"         "smoothness_mean"   "compactness_mean"
##  [7] "concavity_mean"    "points_mean"       "symmetry_mean"
## [10] "dimension_mean"    "radius_se"         "texture_se"
## [13] "perimeter_se"      "area_se"           "smoothness_se"
## [16] "compactness_se"    "concavity_se"      "points_se"
## [19] "symmetry_se"       "dimension_se"      "radius_worst"
## [22] "texture_worst"     "perimeter_worst"   "area_worst"
## [25] "smoothness_worst"  "compactness_worst" "concavity_worst"
## [28] "points_worst"      "symmetry_worst"    "dimension_worst"
```

```r
g1 = lm(radius_mean ~ BM_class_train, data = can_train)
summary(g1)
```

```
##
## Call:
## lm(formula = radius_mean ~ BM_class_train, data = can_train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.30429 -0.07569  0.00025  0.07136  0.50786
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
## (Intercept)               0.245812   0.007394   33.24   <2e-16 ***
## BM_class_trainmalignant 0.246324   0.012182   20.22   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1146 on 378 degrees of freedom
## Multiple R-squared:  0.5196, Adjusted R-squared:  0.5183
## F-statistic: 408.9 on 1 and 378 DF,  p-value: < 2.2e-16
```

```r
names(summary(g1))
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

```r
summary(g1)$fstatistic
```

```
##    value    numdf    dendf
## 408.8577   1.0000 378.0000
```

```r
# The significance measure we want:
summary(g1)$fstatistic[1]
```

```
##    value
## 408.8577
```

**Aylin**: The first chunk of code, a vector is created in order to keep all the outputs together. The next code the variables are run through to try to get a linear fit and have the F- statistic stored. The first code also asks NA to be repeated 30 times. The first three variables in the first row has the f statistic value.

```r
exp_var_fstat <- as.numeric(rep(NA, times = 30))

names(exp_var_fstat) <- names(can_train)

exp_var_fstat["radius_mean"] <- summary(lm(radius_mean ~ BM_class_train, data = can_train))$fstatistic[

exp_var_fstat["texture_mean"] <- summary(lm(texture_mean ~ BM_class_train, data = can_train))$fstatisti

exp_var_fstat["perimeter_mean"] <- summary(lm(perimeter_mean ~ BM_class_train, data = can_train))$fstat

exp_var_fstat
```

```
##       radius_mean      texture_mean    perimeter_mean         area_mean
##         408.85767          78.54968         444.85665                NA
##   smoothness_mean  compactness_mean    concavity_mean       points_mean
##                NA                NA                NA                NA
##     symmetry_mean    dimension_mean         radius_se        texture_se
##                NA                NA                NA                NA
##      perimeter_se           area_se     smoothness_se    compactness_se
##                NA                NA                NA                NA
##      concavity_se         points_se       symmetry_se      dimension_se
##                NA                NA                NA                NA
##      radius_worst     texture_worst   perimeter_worst        area_worst
##                NA                NA                NA                NA
##  smoothness_worst compactness_worst   concavity_worst      points_worst
##                NA                NA                NA                NA
##    symmetry_worst   dimension_worst
##                NA                NA
```

### Looping over variable names

**Aylin**: The last step is repeated again to create a vector in order to hold the siginficance measures. The code commented out produces an error since there is no variable with the name form exp_vars[j]. The named variabe needs to be stored in the variable named "slot", so a variable is created and a formula is created in order for this to happen in the next code snippet below the code with the error. Again the variable exp_var_fstat is called and a table is outputted with the variables in the data set and the f statistic.

```
exp_vars = names(can_train)

exp_var_fstat = as.numeric(rep(NA, times = 30))

names(exp_var_fstat) = exp_vars

# Code snippet commented out creates an error.

#for (j in 1:length(exp_vars)) {
 #    exp_var_fstat[exp_vars[j]] = summary(lm(exp_vars[j] ~ BM_class_train, data = can_train))$fstatist
#  }

for (j in 1:length(exp_vars)) {

   exp_var_fstat[exp_vars[j]] =

      summary(lm(as.formula(paste(exp_vars[j], " ~ BM_class_train")), data = can_train))$fstatistic[1]

   }

exp_var_fstat
```

```
##       radius_mean      texture_mean    perimeter_mean         area_mean
##      4.088577e+02      7.854968e+01      4.448566e+02      3.523234e+02
##   smoothness_mean  compactness_mean    concavity_mean       points_mean
##      5.206519e+01      2.024908e+02      4.308096e+02      6.096495e+02
##     symmetry_mean    dimension_mean         radius_se        texture_se
##      5.392311e+01      2.317485e-02      1.704814e+02      4.948036e-03
##      perimeter_se           area_se     smoothness_se    compactness_se
##      1.575166e+02      1.436926e+02      1.198831e+00      3.769899e+01
##      concavity_se         points_se       symmetry_se      dimension_se
##      5.853678e+01      9.741701e+01      5.988748e-02      2.060474e+00
##      radius_worst     texture_worst   perimeter_worst        area_worst
##      5.567013e+02      1.037929e+02      5.907298e+02      4.119974e+02
##  smoothness_worst compactness_worst   concavity_worst      points_worst
##      9.010266e+01      1.993949e+02      3.540642e+02      6.870139e+02
##    symmetry_worst   dimension_worst
##      9.029549e+01      4.150027e+01
```

**Aylin**: The function lapply or sapply is used to avoid initializing the variables. This gets all stored in a second variable "exp_var_fstat2".

```
exp_var_fstat2 = sapply(exp_vars, function(x) {

    summary(lm(as.formula(paste(x, " ~ BM_class_train")), data = can_train))$fstatistic[1]

})

exp_var_fstat2
```

```
##       radius_mean.value      texture_mean.value    perimeter_mean.value
##            4.088577e+02            7.854968e+01            4.448566e+02
##         area_mean.value   smoothness_mean.value  compactness_mean.value
##            3.523234e+02            5.206519e+01            2.024908e+02
##    concavity_mean.value       points_mean.value     symmetry_mean.value
```

```
##            4.308096e+02            6.096495e+02            5.392311e+01
##    dimension_mean.value         radius_se.value        texture_se.value
##            2.317485e-02            1.704814e+02            4.948036e-03
##      perimeter_se.value            area_se.value     smoothness_se.value
##            1.575166e+02            1.436926e+02            1.198831e+00
##    compactness_se.value       concavity_se.value          points_se.value
##            3.769899e+01            5.853678e+01            9.741701e+01
##       symmetry_se.value       dimension_se.value      radius_worst.value
##            5.988748e-02            2.060474e+00            5.567013e+02
##     texture_worst.value   perimeter_worst.value        area_worst.value
##            1.037929e+02            5.907298e+02            4.119974e+02
##  smoothness_worst.value compactness_worst.value   concavity_worst.value
##            9.010266e+01            1.993949e+02            3.540642e+02
##      points_worst.value    symmetry_worst.value   dimension_worst.value
##            6.870139e+02            9.029549e+01            4.150027e+01
```

```r
names(exp_var_fstat2) = exp_vars
```

### plyr version of the fit

**Aylin**: The data is now combined together by creating a list of data.frames with one category for each variable. The BM class variable is packaged into the data.frames so all the variables are all in one location. When you get the output you will see numerical values with four categories:sample, variable, value, and the variable's class.

```r
can_L = lapply(exp_vars, function(x) {

      df = data.frame(sample = rownames(can_train), variable = x, value = can_train[,
        x], class = BM_class_train)
    df
})

head(can_L[[1]])
```

```
##              sample    variable     value     class
## 874373       874373 radius_mean 0.2238156    benign
## 8510653     8510653 radius_mean 0.2886554    benign
## 864033       864033 radius_mean 0.1323300    benign
## 924632       924632 radius_mean 0.2791897    benign
## 874217       874217 radius_mean 0.5361825 malignant
## 901034302 901034302 radius_mean 0.2630981    benign
```

```r
head(can_L[[5]])
```

```
##              sample        variable     value     class
## 874373       874373 smoothness_mean 0.4072402    benign
## 8510653     8510653 smoothness_mean 0.4953507    benign
## 864033       864033 smoothness_mean 0.4610454    benign
## 924632       924632 smoothness_mean 0.2581927    benign
## 874217       874217 smoothness_mean 0.3001715 malignant
## 901034302 901034302 smoothness_mean 0.1961722    benign
```

```r
names(can_L) = exp_vars
```

**Aylin**: The function laply in the plyr library. The function sapply can also be the same since they are basically the same function. There are three different types of mean( radius_mean, texture_mean, perimeter_mean) along with the f statistic values created.

```
library(plyr)

var_sig_fstats = laply(can_L, function(df) {
    fit = lm(value ~ class, data = df)
    f = summary(fit)$fstatistic[1]
    f
})

names(var_sig_fstats) = names(can_L)

var_sig_fstats[1:3]
```

```
##    radius_mean   texture_mean perimeter_mean
##      408.85767       78.54968      444.85665
```

### Conclusions about significance of the variables

**Aylin**: The first code snippet is asking for the data for variables ordered 1 to 5 which is points_worst, perimeter_worst, points_mean, radius_worst, and area_worst. It then prints out for each of these variables the significant f stats for each.The same goes for the second code snippet except for data variables ordered 25 to 30. Below, the last code snippet, the variables in the training set named data.frame are reordered by significance in order to prepare to do the kNN.

```
most_sig_stats = sort(var_sig_fstats, decreasing = T)

most_sig_stats[1:5]
```

```
##    points_worst     points_mean perimeter_worst    radius_worst
##        687.0139        609.6495        590.7298        556.7013
##  perimeter_mean
##        444.8566
```

```
most_sig_stats[25:30]
```

```
## compactness_se   dimension_se  smoothness_se    symmetry_se dimension_mean
##   37.698991861    2.060474129    1.198830786    0.059887480    0.023174852
##     texture_se
##    0.004948036
```

```
can_train_ord = can_train[, names(most_sig_stats)]
```

### Monte Carlo Cross-Validation

### Selection of the family of training sets

**Aylin**: The data below is subsetted. The first code takes the length of the training set, the second takes the length of the training set and multiplies it by 2/3, the third takes the length of the training set and subtracts it from 253. The value 253 is the size of the new training set. The training data gets loaded and is named training_family_L.Data.

```
length(training_id)
```

```
## [1] 380
```

```
(2/3) * length(training_id)
```

```
## [1] 253.3333
```

```r
length(training_id) - 253
```

```
## [1] 127
```

```r
# Use 253 as the training set size.

training_family_L = lapply(1:1000, function(j) {
    perm = sample(1:380, size = 380, replace = F)
    shuffle = training_id[perm]
    trn = shuffle[1:253]
    trn
})

# save(training_family_L, file='training_family_L.RData')

#load("training_family_L.RData")

validation_family_L = lapply(training_family_L, function(x) setdiff(training_id, x))
```

### Finding an optimal set of variables and optimal k

**Aylin**: The code below calculates the distributions of errors over the 1000 training-validation pairs for different subsets of the variables. The square root of the reference set size is taken in order to test options for k. The value will vary from 3 to 19 and the last code, for each training - validation subset, number of variables, and k, the error of the kNN prediction in the validation set is calculated.

```r
N = seq(from = 3, to = 29, by = 2)

sqrt(length(training_family_L[[1]]))
```

```
## [1] 15.90597
```

```r
K = seq(from = 3, to = 19, by = 2)

1000 * length(N) * length(K)
```

```
## [1] 126000
```

### Execution of the test with loops

**Aylin**: The data frame will be initialized to store 126,000 entries. A new function for the core kNN error is created and stored in the knn_test, n = 5 and k = 7.

```r
paramter_errors_df = data.frame(mc_index = as.integer(rep(NA, times = 126000)),
    var_num = as.integer(rep(NA, times = 126000)), k = as.integer(rep(NA, times = 126000)),
    error = as.numeric(rep(NA, times = 126000)))

knn_test = knn(train = can_train_ord[training_family_L[[1]], 1:5], test = can_train_ord[validation_famil
    1:5], cl = BM_class_train[training_family_L[[1]]], k = 7)

knn_test[1:3]
```

```
## [1] benign benign benign
## Levels: benign malignant
```

```r
tbl_test = table(knn_test, BM_class_train[validation_family_L[[1]]])
```

```
tbl_test
```

```
##
## knn_test    benign malignant
##   benign        80         3
##   malignant      1        43
```

```
err_rate = (tbl_test[1, 2] + tbl_test[2,1])/length(validation_family_L[[1]])

err_rate
```

```
## [1] 0.03149606
```

**Aylin**: Another function is created to run the code snippet and return back the error rate along with other parameters. Then below after "sample" commented out, a nested for loop is created.

```
# j = index, n = length of range of variables, k=k

core_knn = function(j, n, k) {
    knn_predict = knn(train = can_train_ord[training_family_L[[j]], 1:n],
        test = can_train_ord[validation_family_L[[j]], 1:n], cl = BM_class_train[training_family_L[[j]]]

    tbl = table(knn_predict, BM_class_train[validation_family_L[[j]]])
    err = (tbl[1, 2] + tbl[2, 1])/length(validation_family_L[[j]])
    err
}

# sample

core_knn(1, 5, 7)
```

```
## [1] 0.03149606
```

```
iter = 1

str_time = Sys.time()

for (j in 1:1000) {
    for (n in 1:length(N)) {
        for (m in 1:length(K)) {
            err = core_knn(j, N[n], K[m])
            paramter_errors_df[iter, ] <- c(j, N[n], K[m], err)
            iter = iter + 1
        }
    }
}

time_lapsed_for = Sys.time() - str_time

save(paramter_errors_df, time_lapsed_for, file = "for_loop_paramter_errors.RData")

load("for_loop_paramter_errors.RData")
time_lapsed_for
```

```
## Time difference of 8.082501 mins
```

### Execution with plyr

**Aylin**: A data frame of all possible parameter combinations is created.Then they will be nested inside the several **ply functions. Below is just a trst using 20 choices of parameters and then you do a full run.

```
param_df1 = merge(data.frame(mc_index = 1:1000), data.frame(var_num = N))

param_df = merge(param_df1, data.frame(k = K))

str(param_df)
```

```
## 'data.frame':    126000 obs. of  3 variables:
##  $ mc_index: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ var_num : num  3 3 3 3 3 3 3 3 3 3 ...
##  $ k       : num  3 3 3 3 3 3 3 3 3 3 ...
```

```
knn_err_est_df_test_test = ddply(param_df[1:20, ], .(mc_index, var_num, k), function(df) {

   err = core_knn(df$mc_index[1], df$var_num[1], df$k[1])

     err
})

head(knn_err_est_df_test_test)
```

```
##   mc_index var_num k         V1
## 1        1       3 3 0.05511811
## 2        2       3 3 0.07874016
## 3        3       3 3 0.04724409
## 4        4       3 3 0.09448819
## 5        5       3 3 0.07086614
## 6        6       3 3 0.07874016
```

```
str_time = Sys.time()

knn_err_est_df_test_test = ddply(param_df, .(mc_index, var_num, k), function(df) {
    err = core_knn(df$mc_index[1], df$var_num[1], df$k[1])

    err
})

time_lapsed = Sys.time() - str_time

save(knn_err_est_df_test_test, time_lapsed, file = "knn_err_est_df_test_test")

load("knn_err_est_df_test_test")

time_lapsed
```

```
## Time difference of 4.032573 mins
```

```
head(knn_err_est_df_test_test)
```

```
##   mc_index var_num  k         V1
## 1        1       3  3 0.05511811
## 2        1       3  5 0.05511811
## 3        1       3  7 0.05511811
## 4        1       3  9 0.03937008
## 5        1       3 11 0.03149606
```

```
## 6        1       3 13 0.03149606
```

```r
names(knn_err_est_df_test_test)[4] = "error"
```

### Getting summary performance statistics

**Aylin**: The mean is taken for each of the parameters.Then the mean error is taken for the parameters.

```r
mean_ex_df = subset(knn_err_est_df_test_test, var_num == 5 & k == 7)

head(mean_ex_df)
```

```
##     mc_index var_num k      error
## 12         1       5 7 0.03149606
## 138        2       5 7 0.06299213
## 264        3       5 7 0.03937008
## 390        4       5 7 0.06299213
## 516        5       5 7 0.04724409
## 642        6       5 7 0.07874016
```

```r
mean(mean_ex_df$error)
```

```
## [1] 0.05548819
```

```r
mean_errs_df = ddply(knn_err_est_df_test_test, .(var_num, k), function(df)
mean(df$error))

head(mean_errs_df)
```

```
##   var_num  k         V1
## 1       3  3 0.06101575
## 2       3  5 0.06074803
## 3       3  7 0.06195276
## 4       3  9 0.05985827
## 5       3 11 0.05826772
## 6       3 13 0.05629921
```

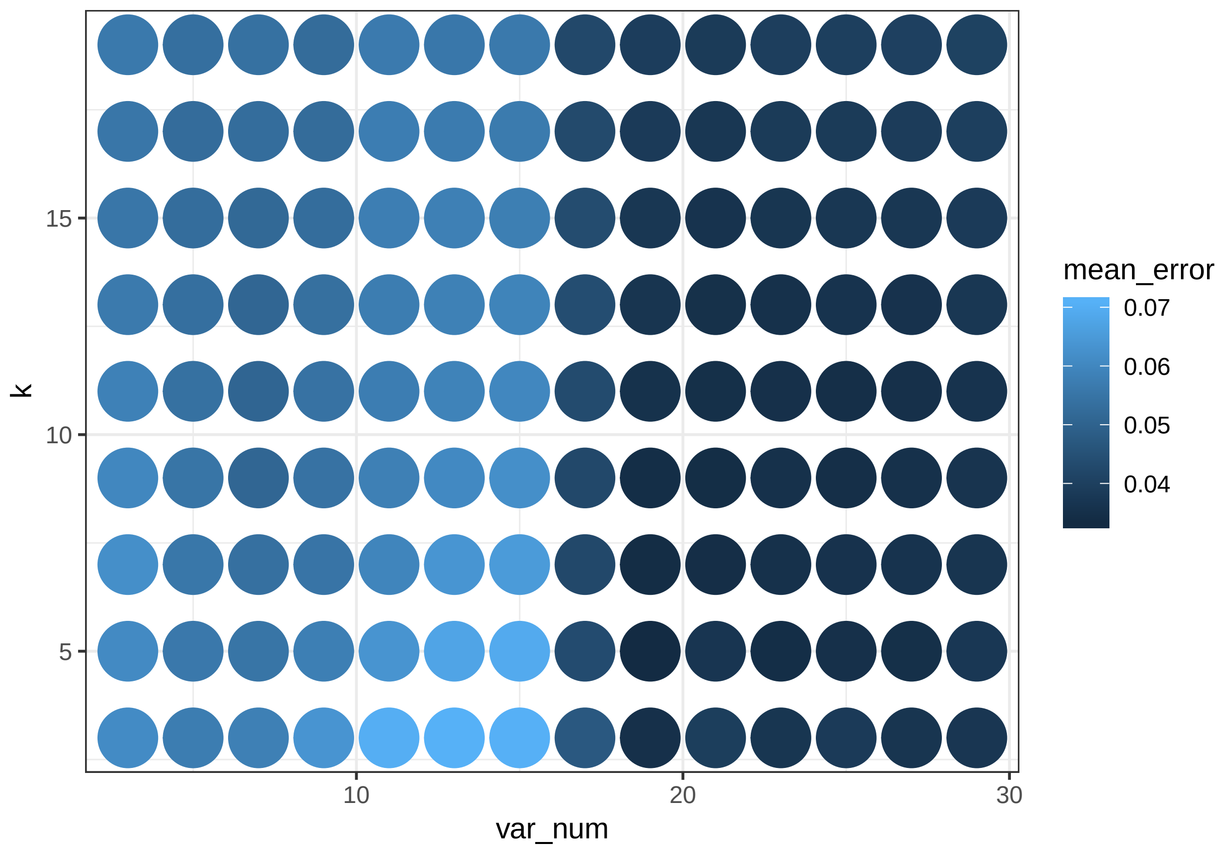
```r
names(mean_errs_df)[3] = "mean_error"
```

### Survey of parameter performance

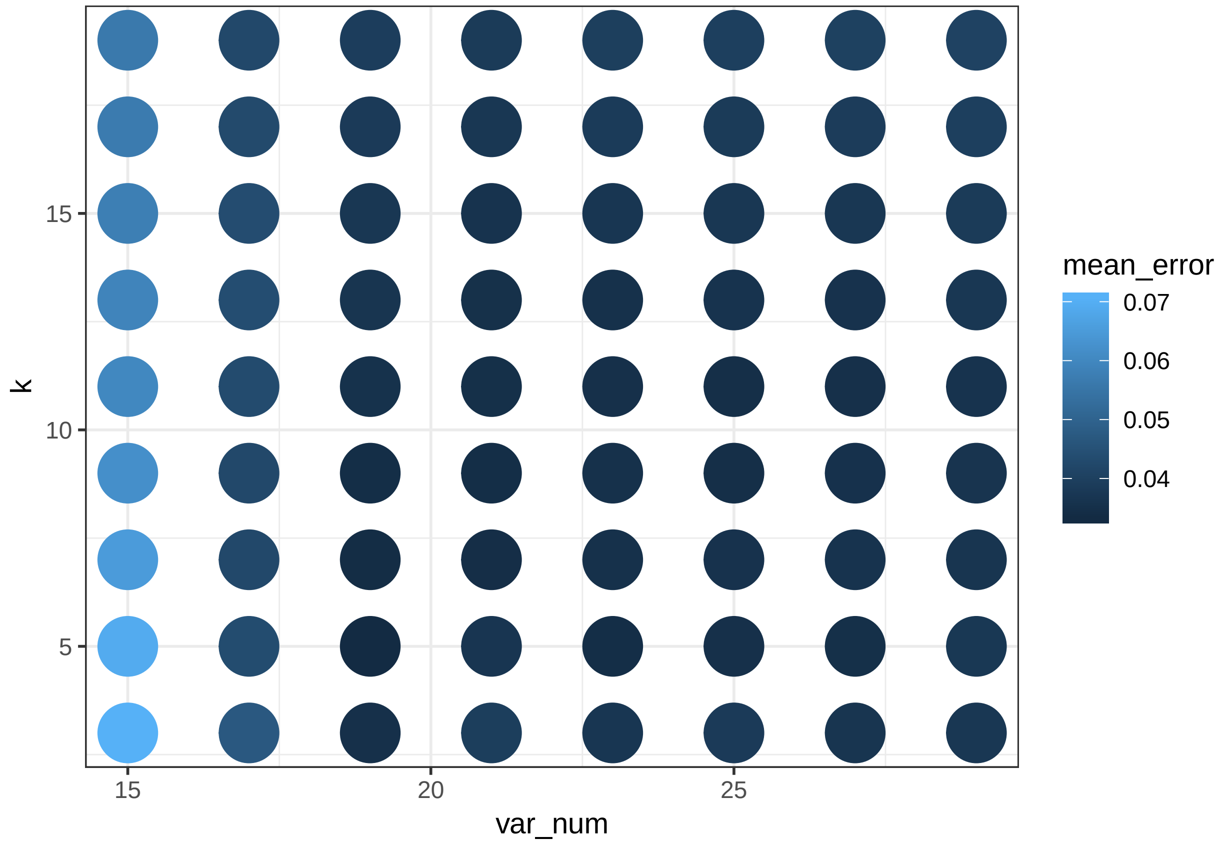
**Aylin**: The performance gets visualized using the library ggplot.

```r
library(ggplot2)

ggplot(data = mean_errs_df, aes(x = var_num, y = k, color = mean_error)) + geom_point(size = 10) +
```




```r
ggplot(data = subset(mean_errs_df, var_num >= 15), aes(x = var_num, y = k, color = mean_error)) +
    geom_point(size = 10) + theme_bw()
```




**Aylin**: Variables with a low k works the best.

```
subset(mean_errs_df, var_num == 17)
```

```
##    var_num  k mean_error
## 64      17  3 0.04700787
## 65      17  5 0.04331496
## 66      17  7 0.04236220
## 67      17  9 0.04225984
## 68      17 11 0.04317323
## 69      17 13 0.04371654
## 70      17 15 0.04346457
## 71      17 17 0.04278740
## 72      17 19 0.04233858
```

```
subset(mean_errs_df, var_num == 19)
```

```
##    var_num  k mean_error
## 73      19  3 0.03497638
## 74      19  5 0.03340157
## 75      19  7 0.03388189
## 76      19  9 0.03425197
## 77      19 11 0.03548819
## 78      19 13 0.03650394
## 79      19 15 0.03722835
## 80      19 17 0.03821260
## 81      19 19 0.03914173
```

```r
subset(mean_errs_df, var_num == 21)
```

```
##    var_num  k mean_error
## 82      21  3 0.03916535
## 83      21  5 0.03667717
## 84      21  7 0.03436220
## 85      21  9 0.03422835
## 86      21 11 0.03485827
## 87      21 13 0.03515748
## 88      21 15 0.03604724
## 89      21 17 0.03710236
## 90      21 19 0.03833071
```

```r
subset(mean_errs_df, var_num == 25)
```

```
##     var_num  k mean_error
## 100      25  3 0.03818898
## 101      25  5 0.03507874
## 102      25  7 0.03570866
## 103      25  9 0.03469291
## 104      25 11 0.03465354
## 105      25 13 0.03588189
## 106      25 15 0.03707874
## 107      25 17 0.03825197
## 108      25 19 0.03969291
```

```r
mean_errs_df[which.min(mean_errs_df$mean_error), ]
```

```
##    var_num k mean_error
## 74      19 5 0.03340157
```

```r
names(can_train_ord)
```

```
##  [1] "points_worst"      "points_mean"       "perimeter_worst"
##  [4] "radius_worst"      "perimeter_mean"    "concavity_mean"
##  [7] "area_worst"        "radius_mean"       "concavity_worst"
## [10] "area_mean"         "compactness_mean"  "compactness_worst"
## [13] "radius_se"         "perimeter_se"      "area_se"
## [16] "texture_worst"     "points_se"         "symmetry_worst"
## [19] "smoothness_worst"  "texture_mean"      "concavity_se"
## [22] "symmetry_mean"     "smoothness_mean"   "dimension_worst"
## [25] "compactness_se"    "dimension_se"      "smoothness_se"
## [28] "symmetry_se"       "dimension_mean"    "texture_se"
```

### Validation of the final test

*VIRAJ*

```r
bcd_val_ord = can_val[, names(can_train_ord)] #Here the 189 observations are taken from both the tables

bm_val_pred <- knn(train = can_train_ord[, 1:27], can_val[, 1:27], BM_class_train,
    k = 3) #training both these variables with k=3 for the class train table

tbl_bm_val <- table(bm_val_pred, BM_class_val) # pred table contains the b or m predicated values, clas
tbl_bm_val
```

```
##            BM_class_val
```

```
## bm_val_pred benign malignant
##   benign       113        37
##   malignant      4        35
```

```
(val_error <- tbl_bm_val[1, 2] + tbl_bm_val[2, 1])/length(BM_class_val) #putting these values from tbl_
```

```
## [1] 0.2169312
```