PQHS 471 FINAL

Youjun Li

May 04, 2018

1 Preparation

```
library("keras")
library(tidyverse)
```

```
## -- Attaching packages --------- tidyverse 1.2.1 --
```

```
## v ggplot2 2.2.1     v purrr   0.2.4
## v tibble  1.4.2     v dplyr   0.7.4
## v tidyr   0.8.0     v stringr 1.2.0
## v readr   1.1.1     v forcats 0.2.0
```

```
## -- Conflicts ------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(dummies)
```

```
## dummies-1.5.6 provided by Decision Patterns
```

```
library(randomForest)
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
library(xgboost)
```

```
##
## Attaching package: 'xgboost'

## The following object is masked from 'package:dplyr':
##
##     slice
```

```r
library(caret)
```

```
## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
##
##     lift
```

```r
trn = read.table("ticdata2000.txt", header = F)
tst = read.table("ticeval2000.txt", header = F)
vnm = read.table("varnames.txt", header = F)
colnames(trn) = vnm$V1
colnames(tst) = vnm$V1[1:85]
tst.y = read.table("tictgts2000.txt", header = F)
colnames(tst.y) = vnm$V1[86]
```

The data contains 86 variables including one binary outcome which indicates if households in one post code would buy the insurance policy. Most of the variables are ordered factors except that `MOSTYPE`(Customer Subtype) and `MOSHOOFD`(Customer main type) are just factors. Additionally, `MAANTHUI`(Number of houses) and `MGEMOMV`(Avg size household) are numerical. Since only Random Forest deals with ordered factors, I will code those level variables as ordered factors only for Random Forest. For the rest of methods (SVM, Boosting and Neural Networks), I will leave them as numerical. In fact, after testing by Neural Network, treating them as numerical gives better results than treating them as factors (using one-hot) in terms of loss (code can be found in the R script file).

First of all, we check if there is any missing.

```r
anyNA(trn)
```

```
## [1] FALSE
```

```r
anyNA(tst)
```

```
## [1] FALSE
```

No missing. Then we want to make sure all variables were loaded as numerical.

```r
mean(sapply(trn, is.numeric))
```

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

```r
mean(sapply(tst, is.numeric))
```

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

All variables are currently numerical. We know the outcome is binary, then what is the proportion for 0?

```r
1 - mean(trn$CARAVAN)  #train
```

```
## [1] 0.9402267
```

```r
1 - mean(tst.y$CARAVAN)  #test
```

```
## [1] 0.9405
```

For both training and testing set, the proportion for 0 is a bit higher than 94%, which means even if we make a prediction of all 0, the accuracy will be at least 0.94. This means our model needs to be really good so that it can out perform the all-zero guessing. This will be challenging.

2 Random Forest

As mentioned above, Random Forest can deal with ordered factors. So we will code the variables that way.

```r
trn2 = trn
tst2 = tst
# convert level variables to ordered factors
trn2[, -c(1, 2, 3, 5, 86)] = (lapply(trn2[, -c(1, 2,
    3, 5, 86)], function(x) factor(x, levels = as.character(sort(unique(x))),
```

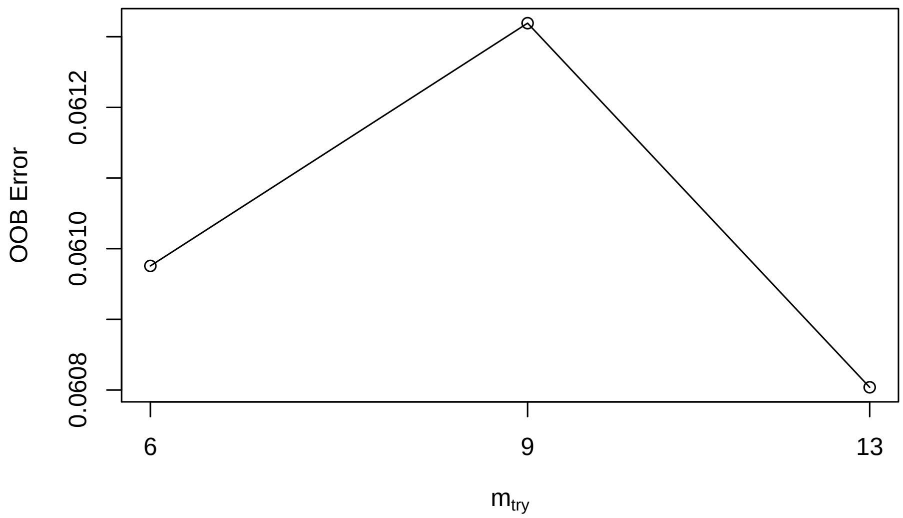
```
    ordered = T)))
tst2[, -c(1, 2, 3, 5)] = (lapply(tst2[, -c(1, 2, 3,
    5)], function(x) factor(x, levels = as.character(sort(unique(x))),
    ordered = T)))

# convert the two customer type variables to
# factors
trn2[, c(1, 5)] = lapply(trn2[, c(1, 5)], function(x) as.factor(x))
tst2[, c(1, 5)] = lapply(tst2[, c(1, 5)], function(x) as.factor(x))
```

I would have shown results from repeated cross validation for a grid search for `mtry`, but it took more than 4 hours to run, so I will just use `tuneRF` to find the optimal `mtry`.

```
set.seed(621)
fit.rf = tuneRF(trn2[, -86], as.factor(trn2$CARAVAN),
    ntreeTry = 1000, stepFactor = 1.5, doBest = T)
```

```
## mtry = 9  OOB error = 6.13%
## Searching left ...
## mtry = 6     OOB error = 6.1%
## 0.005602241 0.05
## Searching right ...
## mtry = 13    OOB error = 6.08%
## 0.008403361 0.05
```




```r
print(fit.rf)
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = res[which.min(res[, 2]), 1])
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##         OOB estimate of  error rate: 6.18%
## Confusion matrix:
##      0  1 class.error
## 0 5461 13 0.002374863
## 1  347  1 0.997126437
```

The results don't look bad (out of bag error rate 6.56%), but considering our goal is to out perform 0.94, I am not so thrilled. Anyway, let's make prediction for the testing set and look at the confusion matrix.

```r
rftest = as.data.frame(cbind(tst2, as.factor(tst.y$CARAVAN)))
yhat.rf = predict(fit.rf, newdata = rftest)
yguess = as.factor(c(rep(0, 4000), 1))[1:4000]
```

```r
# confusionMatrix(yguess,
# rftest$`as.factor(tst.y$CARAVAN)`)
confusionMatrix(data = yhat.rf, rftest$`as.factor(tst.y$CARAVAN)`)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 3758  238
##          1    4    0
##
##                Accuracy : 0.9395
##                  95% CI : (0.9317, 0.9467)
##     No Information Rate : 0.9405
##     P-Value [Acc > NIR] : 0.6216
##
##                   Kappa : -0.002
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.9989
##             Specificity : 0.0000
##          Pos Pred Value : 0.9404
##          Neg Pred Value : 0.0000
##              Prevalence : 0.9405
##          Detection Rate : 0.9395
##    Detection Prevalence : 0.9990
##       Balanced Accuracy : 0.4995
##
##        'Positive' Class : 0
##
```

Still, we failed to out perform 0.94 with accuracy 0.9398.

3 Boosting

I will use xgboost for this section. The data wrangling will be different from it was for Random Forest as I will only convert the two categorical variables to factors without ordering, leaving the rest as numerical, and then to one-hot for the two factors.

```r
trn3 = trn
tst3 = tst

trn3$grp = rep(100, nrow(trn3))
tst3$grp = rep(101, nrow(tst3))

df3 = rbind(trn3[, -86], tst3)
# names(df3)
sum(sapply(df3, is.numeric))
```

```
## [1] 86
```

```r
df3[, c(1, 5)] = lapply(df3[, c(1, 5)], as.factor)

# one hot
df_dum3 = dummy.data.frame(df3)
# colnames(df_dum3)

xtrn3 = dplyr::filter(df_dum3, grp == 100)
mean(sapply(xtrn3, is.numeric))
```

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

```r
xtrn3 = as.data.frame(apply(xtrn3, 2, as.numeric))
x_train3 = data.matrix(xtrn3[, -ncol(xtrn3)])
dim(x_train3)
```

```
## [1] 5822  133
```

```r
# sum(sapply(x_train, is.numeric))
y_train3 = data.matrix((trn3$CARAVAN))
colnames(y_train3) = colnames(tst.y)
xtst3 = filter(df_dum3, grp == 101)
xtst3 = as.data.frame(apply(xtst3, 2, as.numeric))
x_test3 = data.matrix(xtst3[, -ncol(xtst3)])
dim(x_test3)
```

```
## [1] 4000  133
```

```r
y_test3 = data.matrix(tst.y)
```

```r
dtrain = xgb.DMatrix(data = x_train3, label = as.numeric(y_train3))
dtest = xgb.DMatrix(data = x_test3, label = as.numeric(y_test3))
```

Now let's do xgboost by trees and see what test accuracy it will give.

```r
watchlist = list(train = dtrain, test = dtest)
xgb_params = list(objective = "binary:logistic")
set.seed(621)
bst = xgb.train(params = xgb_params, data = dtrain,
    max.depth = 2, eta = 1, nthread = 1, nround = 10,
    watchlist = watchlist, eval.metric = "error", eval.metric = "logloss")
```

```
## [1]  train-error:0.059773    train-logloss:0.255941  test-error:0.059500 test-logl
## [2]  train-error:0.059773    train-logloss:0.210790  test-error:0.059500 test-logl
## [3]  train-error:0.059773    train-logloss:0.201693  test-error:0.059500 test-logl
## [4]  train-error:0.059086    train-logloss:0.197326  test-error:0.060750 test-logl
## [5]  train-error:0.059086    train-logloss:0.194689  test-error:0.061000 test-logl
## [6]  train-error:0.059086    train-logloss:0.192470  test-error:0.060750 test-logl
## [7]  train-error:0.059086    train-logloss:0.190597  test-error:0.060500 test-logl
## [8]  train-error:0.058914    train-logloss:0.189228  test-error:0.060500 test-logl
## [9]  train-error:0.058914    train-logloss:0.187500  test-error:0.060500 test-logl
## [10] train-error:0.058914    train-logloss:0.186402  test-error:0.060500 test-logl
```

```r
# 3 rounds are sufficient
predxg = predict(bst, dtest)
predxg = as.numeric(predxg > 0.5)
confusionMatrix(predxg, tst.y$CARAVAN)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 3755  235
##          1    7    3
##
##                Accuracy : 0.9395
##                  95% CI : (0.9317, 0.9467)
##     No Information Rate : 0.9405
##     P-Value [Acc > NIR] : 0.6216
##
##                   Kappa : 0.0195
```

```
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.99814
##             Specificity : 0.01261
##          Pos Pred Value : 0.94110
##          Neg Pred Value : 0.30000
##              Prevalence : 0.94050
##          Detection Rate : 0.93875
##    Detection Prevalence : 0.99750
##       Balanced Accuracy : 0.50537
##
##        'Positive' Class : 0
##
```

```r
unique(predxg)
```

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

We get an accuracy of 0.9395, still not better than 0.94, but a little improvement over random forest (even Kappa is better, from 0.0539 to 0.0195).

4 Support Vector Machine

The form of the data will be the same as boosting, only with predictors and outcome merged.

```r
svm_train = xtrn3
svm_train$grp = as.factor(y_train3)
class(svm_train$grp)
```

```
## [1] "factor"
```

```r
dim(svm_train)
```

```
## [1] 5822  134
```

```r
colnames(svm_train)[134] = "CARAVAN"
svm_test = xtst3
svm_test$grp = as.factor(y_test3)
dim(svm_test)
```

```
## [1] 4000  134
```

```r
class(svm_test$grp)
```

```
## [1] "factor"
```

```r
colnames(svm_test)[134] = "CARAVAN"
```

After doing cross validation, the best c we found is 0.01, and we do predictions for the testing set and look at the confusion matrix.

```r
grid.c = expand.grid(C = seq(0.01, 10, length.out = 10))
trctrl.svm = trainControl(method = "cv", number = 5)

set.seed(621)
svm_Linear = train(CARAVAN ~ ., data = svm_train, method = "svmLinear",
    trControl = trctrl.svm, tuneGrid = grid.c, tuneLength = 10)

svm_Linear
```

```
## Support Vector Machines with Linear Kernel
##
## 5822 samples
##  133 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 4657, 4658, 4657, 4658, 4658
## Resampling results across tuning parameters:
##
##   C      Accuracy   Kappa
##    0.01  0.9400550  -0.0003393775
##    1.12  0.9398832   0.0041319996
##    2.23  0.9400550   0.0045131990
##    3.34  0.9398832   0.0041319996
##    4.45  0.9398832   0.0041319996
##    5.56  0.9398832   0.0041319996
##    6.67  0.9398832   0.0041319996
##    7.78  0.9398832   0.0041319996
##    8.89  0.9398832   0.0041319996
##   10.00  0.9398832   0.0041319996
##
```

```
## Accuracy was used to select the optimal
##  model using the largest value.
## The final value used for the model was C = 0.01.
```

```
predsvm = predict(svm_Linear, svm_test)
confusionMatrix(predsvm, tst.y$CARAVAN)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 3762  238
##          1    0    0
##
##                Accuracy : 0.9405
##                  95% CI : (0.9327, 0.9476)
##     No Information Rate : 0.9405
##     P-Value [Acc > NIR] : 0.5172
##
##                   Kappa : 0
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 1.0000
##             Specificity : 0.0000
##          Pos Pred Value : 0.9405
##          Neg Pred Value :    NaN
##              Prevalence : 0.9405
##          Detection Rate : 0.9405
##    Detection Prevalence : 1.0000
##       Balanced Accuracy : 0.5000
##
##        'Positive' Class : 0
##
```

```
unique(predsvm)
```

```
## [1] 0
## Levels: 0 1
```

In fact, the SVM model gives a prediction of all 0. Well, technically an improvement, but even a person who doesn't know anything about machine learning can make this prediction.

5 Neural Networks

For Neural Networks, the form of the data will be the same as boosting.

```
use_session_with_seed(621)
```

```
## Set session seed to 621 (disabled GPU, CPU parallelism)
```

```
model = keras_model_sequential() %>% layer_dense(units = 100,
    activation = "relu", input_shape = ncol(x_train3),
    kernel_regularizer = regularizer_l2(l = 0.01)) %>%
    layer_dropout(rate = 0.2) %>% layer_dense(units = 267,
    activation = "relu") %>% layer_dropout(rate = 0.2) %>%
    layer_dense(units = 1, activation = "sigmoid")  #output
summary(model)
```

```
## ___________________________________________________
## Layer (type)           Output Shape        Param #
## ===================================================
## dense_1 (Dense)        (None, 100)         13400
## ___________________________________________________
## dropout_1 (Dropout)    (None, 100)         0
## ___________________________________________________
## dense_2 (Dense)        (None, 267)         26967
## ___________________________________________________
## dropout_2 (Dropout)    (None, 267)         0
## ___________________________________________________
## dense_3 (Dense)        (None, 1)           268
## ===================================================
## Total params: 40,635
## Trainable params: 40,635
## Non-trainable params: 0
## ___________________________________________________
```

```
model %>% compile(loss = "binary_crossentropy", optimizer = optimizer_adam(),
    metrics = c("accuracy"))
history = model %>% fit(x_train3, y_train3, epochs = 50,
    batch_size = 100, verbose = 1, validation_split = 0.3)

model %>% evaluate(x_test3, y_test3, verbose = 0)  ## 0.94075 on test set
```

```
## $loss
```

```
## [1] 0.2295662
##
## $acc
## [1] 0.94075
```

```
y_pred1 = model %>% predict_classes(x_test3)  ## prediction on test set
table(y_test3, y_pred1)  ## test set confusion matrix
```

```
##        y_pred1
## y_test3    0    1
##       0 3761    1
##       1  236    2
```

After several runs, I ended up using three layers, 100 nodes for the first layer with an $l2$ regularizer, $2 * \#ofpredictors - 1$ nodes for the second layer, 50 epochs, batch size of 100. It gives me the best accuracy (0.94075) so far but not from all-zero prediction.

6 Unsupervised Learning

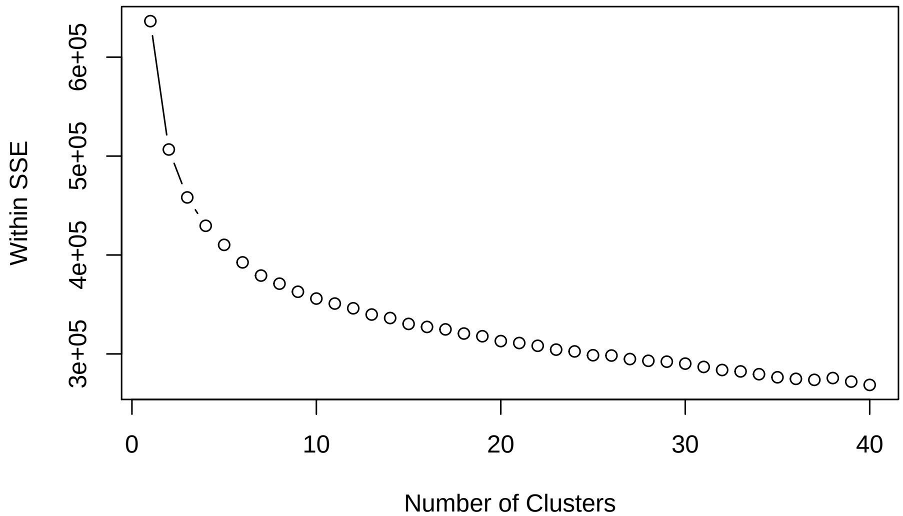
6.1 Kmeans

For kmeans, I use within group sum of squared error to decide the number of clusters.

```
trn5 = trn[, c(6:41)]
# determine how many clusters by within group sum
# of squared error
wss = (nrow(trn5) - 1) * sum(apply(trn5, 2, var))
for (i in 2:40) {
    wss[i] = sum(kmeans(trn5, centers = i, nstart = 10)$withinss)
}
```

```
## Warning: did not converge in 10 iterations
```

```
plot(1:40, wss, type = "b", xlab = "Number of Clusters",
    ylab = "Within SSE")
```



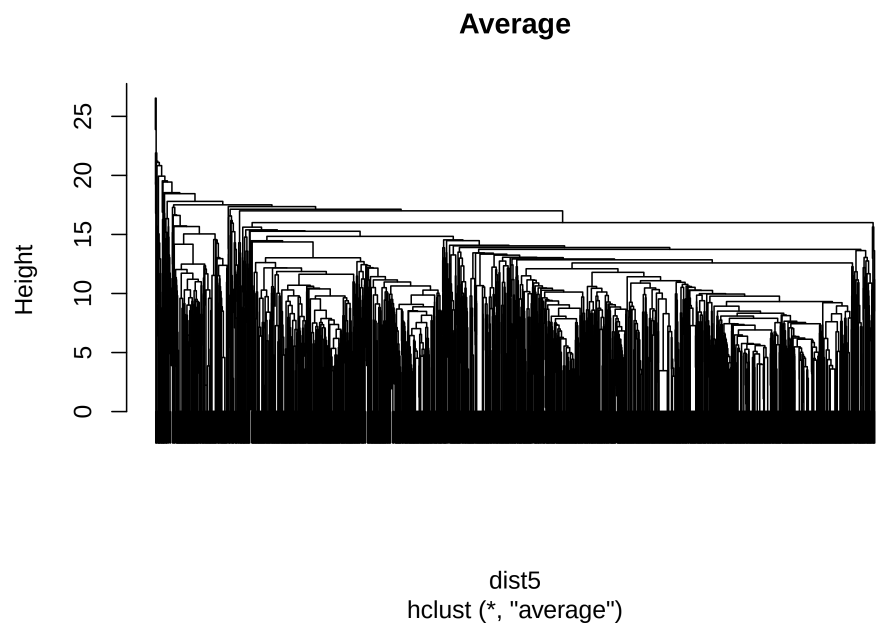

```
# choose 10
cluster.km = kmeans(trn5, 10, nstart = 10)
```

No obvious "elbow" point in the scree plot, but it goes relatively flat after 10, so I choose 10.

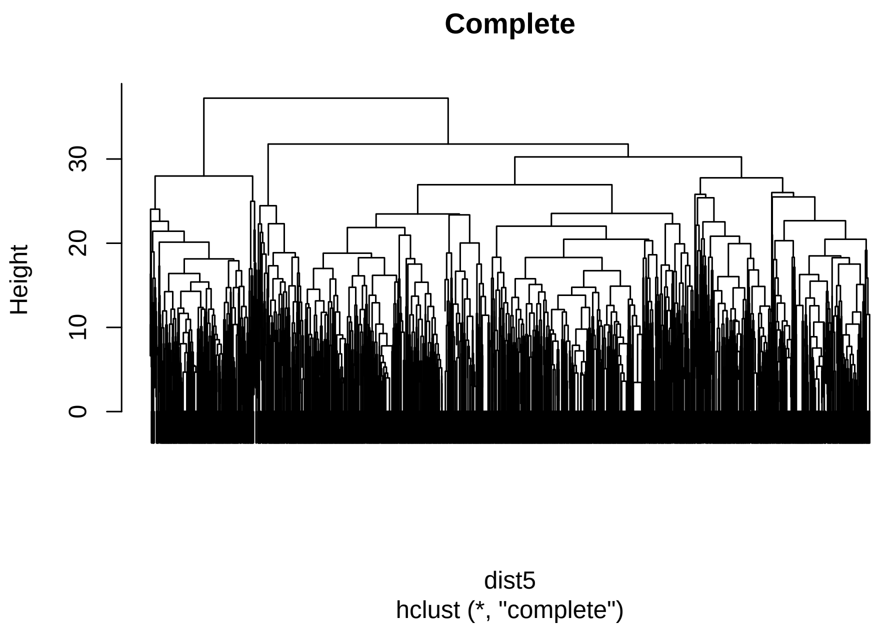
6.2 Hierarchical Clustering

I will just use the number of clusters decided by kmeans for this section, and compare which measure agrees with kmeans more.

```
dist5 = dist(trn5)
plot(hclust(dist5, method = "average"), labels = F,
    main = "Average")
```




```r
plot(hclust(dist5, method = "complete"), labels = F,
    main = "Complete")
```




```
cluster.hc.c = cutree(hclust(dist5, method = "complete"),
    10)
cluster.hc.a = cutree(hclust(dist5, method = "average"),
    10)
table(cluster.hc.c, cluster.hc.a)
```

```
##               cluster.hc.a
## cluster.hc.c     1    2    3    4    5    6    7
##            1  1645    1    0    0    0    0    0
##            2   602    0   12    0    0    0    1
##            3   793    6    0    0    0    0    0
##            4   784    0    0    0    0    0    1
##            5  1508    0    0    0    0    0    0
##            6   376    0    0    1    0    0    0
##            7    39   31    0    0    0    0    0
##            8     6    0    0    0    0    1    0
##            9     0    0    0    0    2    0    0
##            10    0    0    0    0    0    0    7
##               cluster.hc.a
## cluster.hc.c     8    9   10
##            1     0    0    0
```

```
##             2     0     0     0
##             3     0     0     0
##             4     0     0     0
##             5     0     0     1
##             6     4     0     0
##             7     0     1     0
##             8     0     0     0
##             9     0     0     0
##             10    0     0     0
```

```r
table(cluster.hc.c, cluster.km$cluster)
```

```
##
## cluster.hc.c   1   2   3   4   5   6   7   8   9
##            1 230   1   0 119   1 139 561   2  13
##            2  64  14   2   0  27  72  27   0 402
##            3   0  58  97   0 494   3  76   0  71
##            4 272   0   0   0   1 360  55  37  11
##            5   0 578 204 260  20   0 309   0  47
##            6  60   0   0   0   0  16   1 294   8
##            7   0   0  42   0  29   0   0   0   0
##            8   0   0   0   0   3   3   0   1   0
##            9   0   0   0   0   0   1   0   0   1
##            10  0   0   0   0   0   0   0   0   7
##
## cluster.hc.c  10
##            1 580
##            2   7
##            3   0
##            4  49
##            5  91
##            6   2
##            7   0
##            8   0
##            9   0
##            10  0
```

```r
table(cluster.hc.a, cluster.km$cluster)
```

```
##
## cluster.hc.a    1    2    3    4    5    6    7
```

```
##            1   626  650  307  379  568  589 1028
##            2     0    0   37    0    0    0    1
##            3     0    0    0    0    7    0    0
##            4     0    0    0    0    0    0    0
##            5     0    0    0    0    0    1    0
##            6     0    0    0    0    0    0    0
##            7     0    0    0    0    0    0    0
##            8     0    0    0    0    0    4    0
##            9     0    0    1    0    0    0    0
##            10    0    1    0    0    0    0    0
##
## cluster.hc.a    8    9   10
##            1   332  545  729
##            2     0    0    0
##            3     0    5    0
##            4     1    0    0
##            5     0    1    0
##            6     1    0    0
##            7     0    9    0
##            8     0    0    0
##            9     0    0    0
##            10    0    0    0
```

Looks like "complete" agrees with kmeans more.

6.3 MDS

I set $k = 3$ and plot a 3D plot, whose screenshot is shown here. It's hard to see some obvious pattern out of this.

7 Summary

Even though Neural Networks gave the best results in terms of accuracy, I consider myself getting lucky this time. It is very arbitary what values I used for the parameters and I don't know a good way to tune them towards the optimal, if the optimal does exist. But for other methods, we can use approaches like cross validation to do exausted search, even though I could

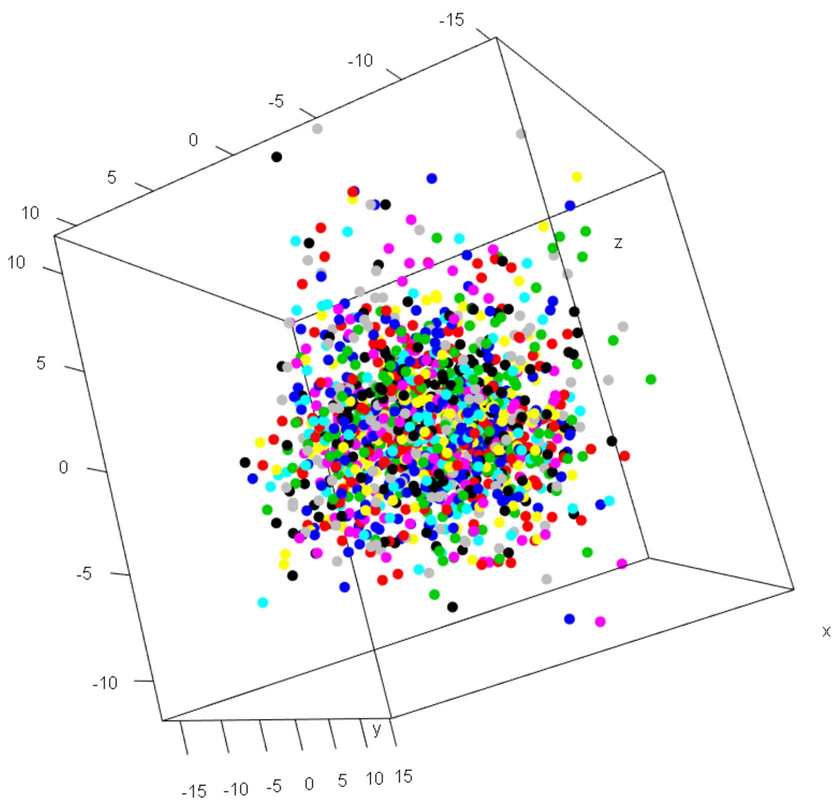


Figure 1: screenshot of 3D plot

take a while. Hence, unless the data is really large, I would still go with methods like boosting first, rather than Neural Network.