STP 598 Machine Learning
Final Project: Binary Classification for Drug Discovery

Arizona State University Applied Math, May 4, 2018
Marion Badal
(**Not in a group**)

Introduction

Drug invention and discovery is a highly active field and modern computer hardware and methods from computational statics and machine learning are routinely employed in attempts to predict performance and attributes of new compounds. The dataset used in this project is the same one described in the paper by R. E. McCulloch et al. in Annals of Applied Statistics, Vol 4 No. 1, 2010. The purpose is to predict if a certain drug compound, which is comprised of a specific combination of molecular descriptors, successfully treats a given disorder.

The Dataset

The data consists of p = 266 molecular descriptors (variables) and one binary outcome which implies either success or failure of a given variable combination (compound) in treating the disorder. There are n = 29,374 compounds of which 542 are defined as active (where Y = 1 implies success in treatment and Y = 0 otherwise). No specific description of the large list of variables is provided, as they are highly specialized molecular forms familiar only to experts in the relevant field. However, such a description is not necessary in successfully applying machine learning methods to the dataset.

Data Pre-Processing

The pre-processing of the data was by far the most time consuming aspect of this work. Initially the dataset was randomly (and naively we might add) split 50/50 into training and test sets, with each set containing half of the active and half of the inactive compounds. First attempts to apply our well-known machine learning methods directly to this raw data were completely unsuccessful, yielding very poor predictive accuracy and even non-convergence. After further study and learning about the best ways to approach the problem, it became clear that the underlying issue is that this is a highly unbalanced dataset, meaning one of the classes (in this case Y = 0) far outweighs the number of the other class (Y = 1). In the present case, only about 1.8 percent of the entire dataset was of the active type while the rest were inactive compounds. This presents serious problems for any machine learning method and can very easily bias the model being trained toward the Y = 0 class. After all, even a ridiculous model, one that always predicts an outcome of Y = 0, would be right 98.2 percent of the time. Of course it would be wrong 100 percent of the time for the Y = 1 cases which is not useful.

This is a serious issue for binary classification but fortunately it is also a well-known one which is common to multiple fields including drug discovery, and several approaches exist to address it. We chose the method of up-sampling (also called oversampling) the minority class (Y = 1). We performed repeated random draws (with replacement) of the training minority class (271 cases) to populate our training set until there was an equal number of the two classes in the training set. Thus our training set contains 14416 instances of Y = 0 and another 14416 where Y = 1. The test set consists of 14416 instances of Y = 0 and 271 with Y = 1. This method is similar in principle to that employed by McCulloch et al. but differs in the details of the actual train test split. We preferred for the test set to have the same proportion of the two classes as the original raw dataset, and the above recipe achieves that.

Machine Learning Methods

We applied four different methods to the train/test data and chose them based on their well-known strong performance for large and difficult datasets like the one we are using. The methods employed are Logit (as a baseline method), Deep Neural Nets, Boosting, and Random Forests. We use AUC and plot lift and ROC curves for each method as a means to judge predictive performance. Due to the large size of the dataset, a modest amount of parameter tuning was carried out for each method since computations were easily taking hours in some cases, even for a small number of options. Thus exploring larger parameter spaces would have been prohibitive with the amount of time and computing power available. Subsequently cross validation was not used to tune each parameter for optimal performance as the data set and variables were so numerous. The machine employed for the computations is a desktop computer with 16GB of RAM and a quad-core processor. While RAM was not an issue, it was clear that more cores would have been advantageous if one desired to run deep neural nets with several hidden layers and hundreds of neurons as well as any of the tree-based methods using thousands of trees. Such extreme settings were beyond the capability of our machine. Nevertheless, using a more measured and conservative choice of parameter settings, we were still able to achieve good results, matching and in some cases very slightly exceeding the predictive performance obtained by McCulloch et al.

1. Logistic Regression (Logit)

Using the package glm, we performed logit on the training data to construct the model as a baseline method to which the more sophisticated tools could be compared. Further, the package pROC and function mlfuns.R were called in order to obtain the ROC (Receiver Operating Characteristic) curve as well as find the corresponding Area Under the Curve (AUC). The related Lift curve was obtained as well. The resulting out-of-sample AUC of **0.7301** was reasonably competitive with the other methods but was nevertheless outperformed by a nontrivial margin as will be seen.

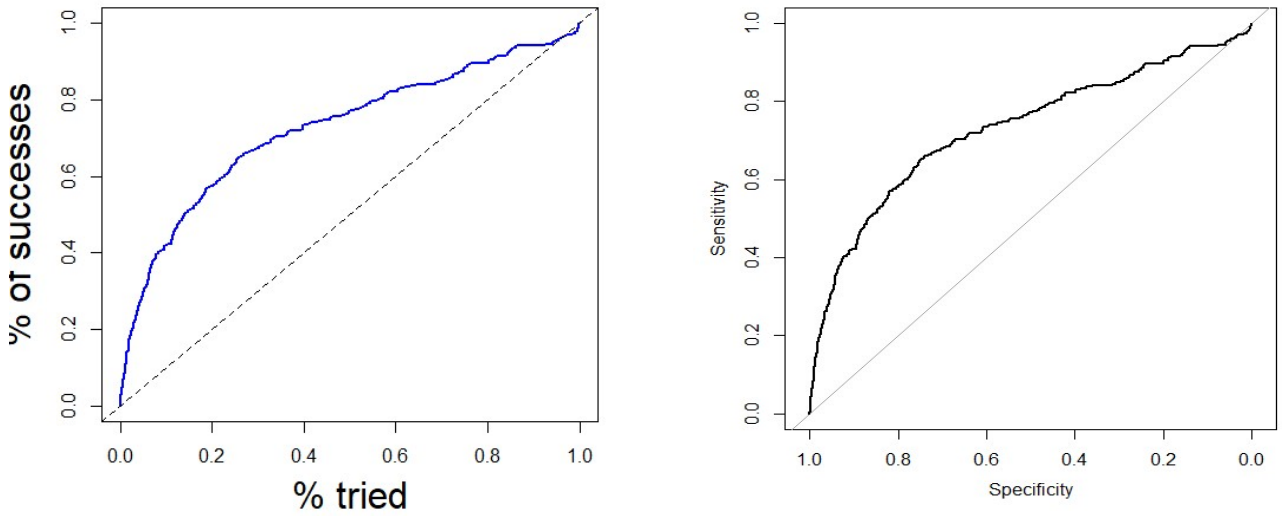

Figure 1. Lift and ROC curves for out-of-sample performance of Logit. The ROC has an AUC of 0.7301 indicating reasonable predictive performance.

2. Deep Neural Nets

Using the package h2o, we trained both single-layer neural nets as a warm up exercise in order to gauge expected performance levels and computational demands as well as deep neural nets in order to more thoroughly train the model. After tuning number of layers, number of neurons, epochs, and the activation function, we settled on 2 hidden layers with 20 neurons each using 50 epochs and the Tanh activation. We found that doubling neurons and epochs did not produce noticeably better results and adding hidden layers was too computationally expensive. The settings we have chosen yielded a strong out-of-sample AUC of **0.7559** and a very reasonable lift curve. The variable selection in h2o performed by the neural net identified the top five most important variables in the dataset as: x.MAXDN, x.IVDE, x.D.Dr12, x.D.Dr11, x.T.S..Cl.

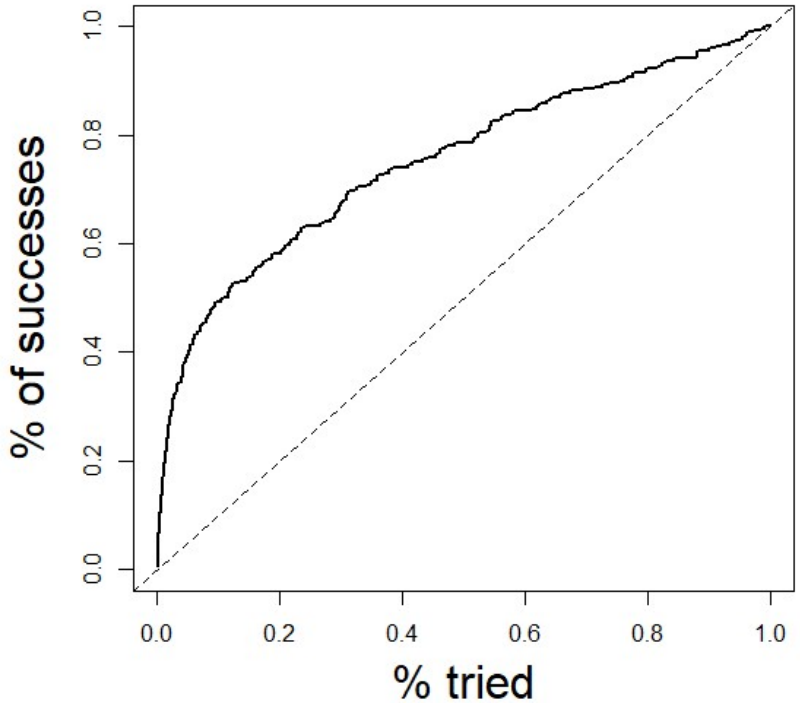

Figure 2. The out-of-sample lift curve for the deep neural net exhibits the expected behavior from a well-trained model.

3. Boosting

We found the tree-based methods to be the most computationally expensive, even for modest sizes and numbers of trees. However, they were also the best performers in predictive accuracy, yielding the best AUC values. We used the gbm package and looped over a handful of values for three important boosting parameters, namely: tree-depth (2,4), number of trees (500,1000), and shrinkage factor (0.1, 0.01). This gives $2^3 = 8$ distinct settings, of which the best out-of-sample AUC was found to be an impressive **0.7804**. Seen below, the lift and ROC curves are good, as expected.

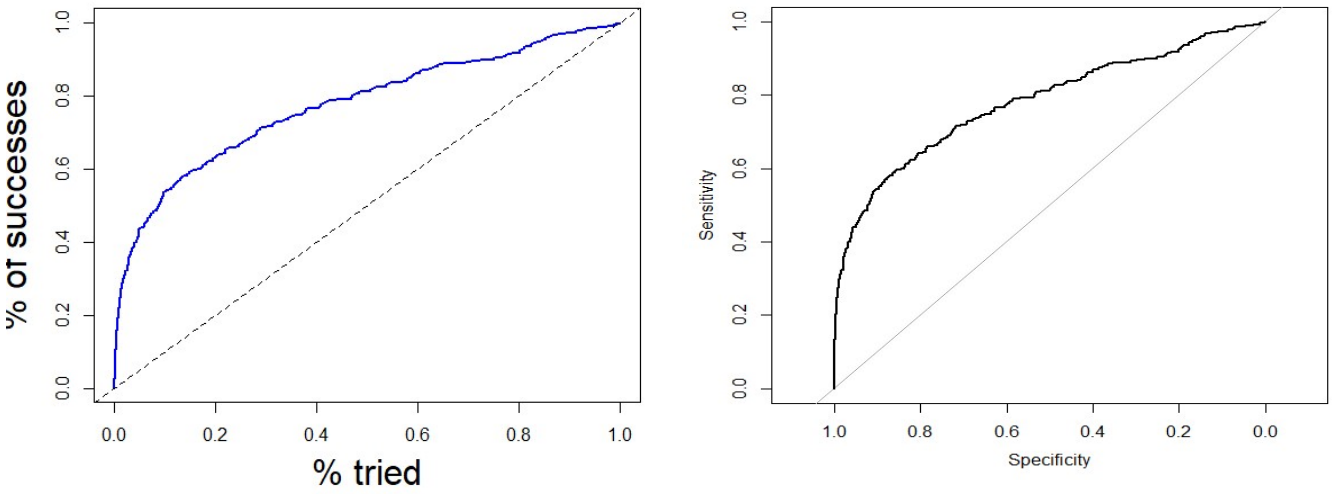

Figure 3. The out-of-sample lift and ROC curves for boosting.

4. Random Forest

We used the randomForest package in R and looped over the parameter ntree (500,1000). Due to computational cost, we chose to use the default value of mtry as sqrt(p) = 16, where p is the number of variables. After much reading, we found that changing mtry does not often make a significant difference and used this observation to our advantage. Here, mtry is the number of trees randomly drawn from the forest. A strong out-of-sample AUC of **0.7910** was obtained along with good lift and ROC curves seen below.

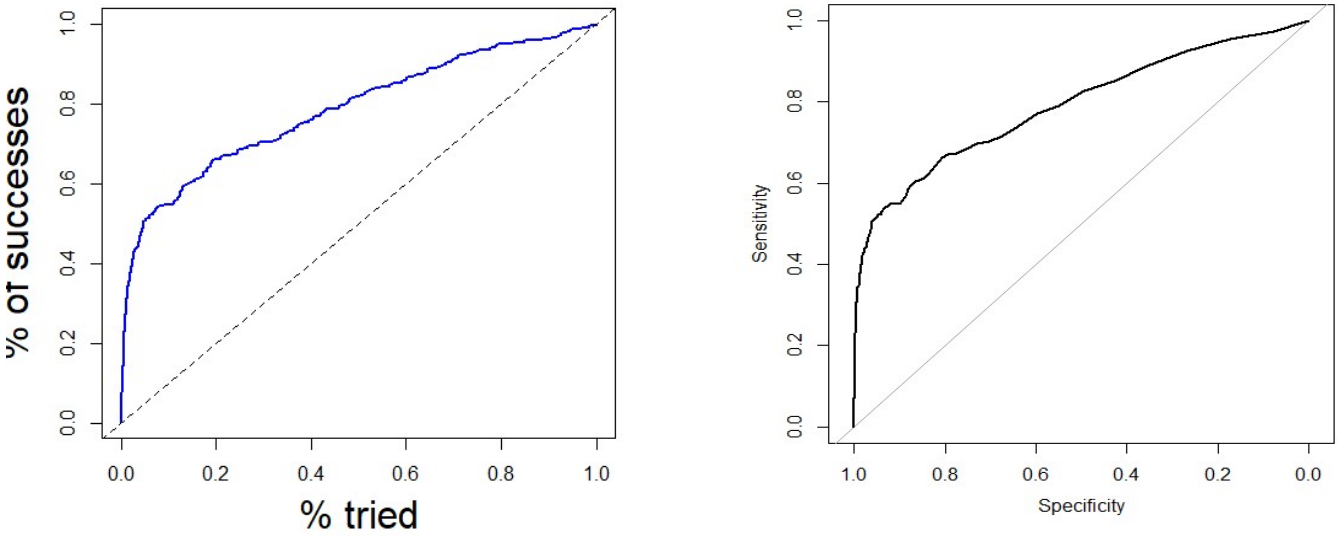

Figure 4. The out-of-sample lift and ROC curves for random forests.

Variable Importance

Here we display graphs of the variable importance obtained for neural nets, boosting, and random forests. These graphs list the top most influential variable in the whole data set. Level of influence is determined in different ways for each method and package. For example, randomForest uses mean reduction in GINI to measure importance while boosting uses relative importance. The plot for boosting is crowded since it shows all variables on the histogram but only labels a few of them. Consequently we explicitly list the most important variables for boosting below. We notice that all the methods give different lists for most important variables. Indeed, even within a single method (e.g. neural nets), different runs of the same method yielded slightly different results for the variable importance list. This is not surprising since these methods make random subset draws from the data when training the model. We note that the variable importance graph for boosting shows a very strong dependence on the first dozen variables so that after that relative influence has dropped dramatically. For boosting, the top seven variables are: x.MDDD, x.T.S..Cl., x.SEigZ, x.BIC2, x.IVDE, x.PW2, x.Xindex

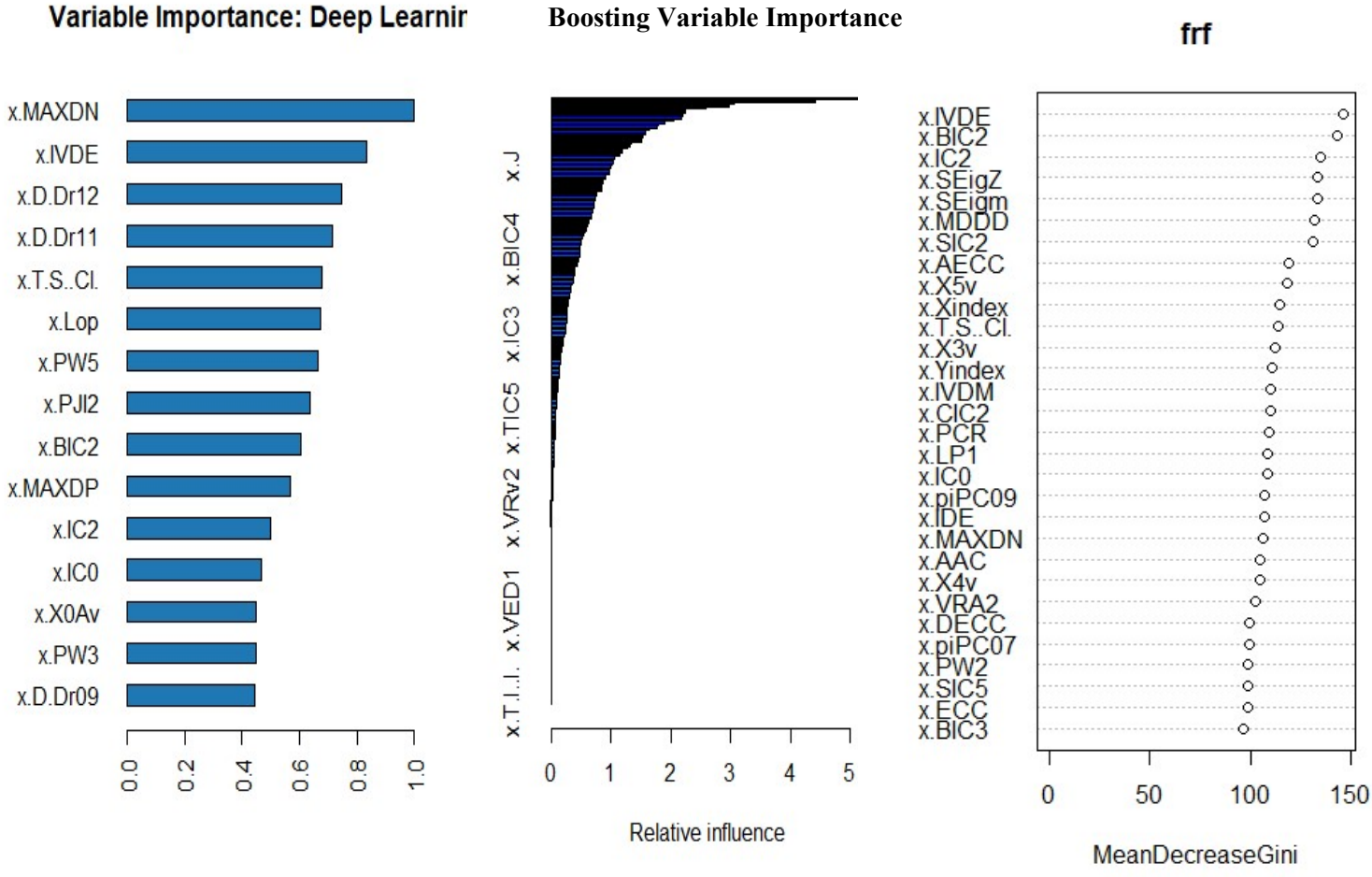

Figure 5. Variable importance plots for deep neural net, boosting, and random forest respectively. Results for each method differ with respect to what variables were chosen as most important. However, there is some overlap as well.

Summary of AUC Performance

Here we present a table of the results comparing out-of-sample AUC of all the methods employed in this work. As noted earlier, all of the methods yielded reasonable results, meaning their AUC values are competitive and lift and ROC curves are good, but the tree-based methods were the strongest performers, with random forests leading the way. This gain in performance comes at the price of long computational times. We note that our results are very similar to those of McCulloch et al.

Method	AUC
Random Forest	0.7910
Boosting	0.7804
Neural Net	0.7559
Logit	0.7301

Future Directions

In this work, we employed most of the major machine learning methods which excel at classification to study the drug discovery problem. While we would not necessarily reach for yet another method, being able to run larger random forests, deeper and broader neural nets across a larger parameter space would be an interesting direction. The need for a more powerful computing platform would first have to be addressed. With more time and resources, one could probe the dataset with greater depth, seeking optimal tuning parameter values to see how and if the values obtained would hold up to such an investigation.

Conclusion

The purpose of this study was to identify appropriate methods which can predict with reasonable accuracy new active compounds in the drug discovery dataset. Employing several well-known machine learning methods, namely logit, deep neural nets, boosting, and random forests, we have been able to identify several sets of influential molecular structures which appear to have a greatly increased chance of producing an active compound when combined. The AUC and variable importance data are critical in making such a determination and our results are promising. An important lesson learned during this process is the influence that proper and thorough data pre-processing has on the success of the project. This was an essential aspect of the work, without which progress would not have been possible.

Please see following pages for the R script used in this work.
(NOTE: We have not included the additional functions provided by Dr. McCulloch)

R Script for Project

```
# Import data
temp = read.csv("drug-discovery.csv")
thedata = temp
numobs = dim(thedata)[1]  # get number of observations

# Combine y variable "somewhat active = 1" and "highly active = 2" into "active = 1"
for(i in 1:numobs){
  if(thedata[i,1] == 2){
    thedata[i,1] = 1
  }
}

# To create train/test split first extract all rows with y = 0 and place them
# in new data frame and do the same with all rows with y = 1. Then split these two
# data frames in half and recombine half the y = 0 rows and half the y = 1 rows to
# get two new data frames which will be the train and test splits. The recombination
# can also be done randomly to obtain multiple train/test splits as needed.

#-----------------------------------------------------
# Create data frame containing all rows with y = 0
yzeros = thedata[FALSE,]  # Create empty data frame with all the correct columns
for(i in 1:numobs){  # Fill the empty data frame only with rows containing y = 0
  if(thedata[i,1] == 0){
    yzeros[i,] = thedata[i,]
  }
}

yzeros_clean = yzeros[complete.cases(yzeros),]  # Remove NA rows
rownames(yzeros_clean) <- NULL  # reset row names to numerical order

# Output data files to disk
# write.csv(yzeros, "yzeros.csv")
# write.csv(yzeros_clean, "yzeros_clean.csv")


#-----------------------------------------------------
# Create data frame containing all rows with y = 1
yones = thedata[FALSE,]  # Create empty data frame with all the correct columns
for(i in 1:numobs){  # Fill the empty data frame only with rows containing y = 1
  if(thedata[i,1] == 1){
    yones[i,] = thedata[i,]
  }
}

yones_clean = yones[complete.cases(yones),]  # Remove NA rows
rownames(yones_clean) <- NULL  # reset row names to numerical order

# Output data files to disk
#write.csv(yones, "yones.csv")
#write.csv(yones_clean, "yones_clean.csv")


#-----------------------------------------------------


# train = sample(1:nrow(testitxm),floor(nrow(testitxm)/2)) #indices of train set
# Create randomly chosen numbers (without replacement) to randomly draw
# from both the y=1 and y=0 sets to create train and test splits
set.seed(14)
```

```
onesindeces = sample(1:nrow(yones_clean),nrow(yones_clean))
zerosindeces = sample(1:nrow(yzeros_clean),nrow(yzeros_clean))
# Take about 14K random samples of the first 271 of the 542 y=1 indeces created above
trainonesindeces  =  sample(onesindeces[1:nrow(yones_clean)/2],nrow(yzeros_clean)/2,
replace=TRUE)

# Now can just use half of the index values each for train and test
# pulling from both ones and zeros lists

# set.seed(24)
# testindeces = sample(1:nrow(yones_clean),10)

# rownames(train_disc) <- NULL  # reset the row names of train_disc set
# rownames(test_disc) <- NULL  # reset the row names of test_disc set

#-----------------------------------------------------
# Create data frame containing the training data randomly chosen from the data
emptyhalfframe = thedata[-c((nrow(thedata)/2+1):nrow(thedata)),]  # Create data frame
with all the correct columns
trainset = emptyhalfframe[FALSE,]  # Create empty data frame for train, ready to fill
testset = emptyhalfframe[FALSE,]  # Create empty data frame for test, ready to fill

############################  IMPORTANT:  ########################################
#########################################################################################
# Since our original data set is highly imbalanced (many more y = 0 classifiers than
# y = 1), then to avoid severe bias in our model for y = 0, we shall use upsampling
# to create a training set which contains an equal proportion of y=0 and y=1 classes.
# This will be done by randomly sampling (with replacement) half the set of y=1
#classes until an equal number of y=1 and y=0 are obtained. The other half of the y=1
#class will be reserved for the test set. The test set will maintain the imbalanced
#proportion of y=0 and y=1 classes.

# Fill the train set with same number of y = 1 rows equal to y = 0 rows.
for(i in 1:nrow(yzeros_clean)/2){
  trainset[i,] = yones_clean[trainonesindeces[i],]
}


# Fill the test set with half of randomly drawn rows with y = 1
for(i in 1:nrow(yones_clean)/2){
  testset[i,] = yones_clean[onesindeces[i+nrow(yones_clean)/2],]
}


# Fill the train set with randomly drawn rows with y = 0
for(i in 1:14416){
  trainset[i+nrow(yzeros_clean)/2,] = yzeros_clean[zerosindeces[i],]
}


# Fill the test set with randomly drawn rows with y = 0
for(i in 1:nrow(yzeros_clean)/2){
  testset[i+nrow(yones_clean)/2,]                                                    =
yzeros_clean[zerosindeces[i+nrow(yzeros_clean)/2],]
}


#########################################################
####################  IMPORTANT:  #######################
#########################################################
# It might be a good idea to randomize the rows after
# combining (concatenating) the y=1 and y=0 rows into one
```

```
# test set or one training set. This way, there are not
# large clusterings of y=1 and y=0 in the data. This
# might be an important step in case the code (e.g. h2o)
# is taking random draws from the train set to create
# the model. The way to do this is simply with the short
# R code below:
# temp3 = temp2[sample(nrow(temp2)),]
############################################################
############################################################

# Randomize twice

testset2 = testset[sample(nrow(testset)),]
trainset2 = trainset[sample(nrow(trainset)),]

testset2 = testset2[sample(nrow(testset2)),]
trainset2 = trainset2[sample(nrow(trainset2)),]

# Read number of 1 and 0 entries

table(trainset2$y)
table(testset2$y)

############################################################
############################################################

# yones_clean = yones[complete.cases(yones),]  # Remove NA rows from yones and yzeros
rownames(trainset2) <- NULL  # reset row names to numerical order
rownames(testset2) <- NULL  # reset row names to numerical order


# ============================================================================
# ============================================================================

#-------------------------------------------------------------
#----------------  Neural Nets for Discovery Data ------------
#-------------------------------------------------------------


source("lift.R")
library(h2o)

# trainDf = read.csv("Tabloid_train.csv")
# testDf = read.csv("Tabloid_test.csv")

trainDf = trainset2
testDf = testset2

print(names(trainDf))

# p=ncol(trainDf)-1
# p = 3
# par(mfrow=c(p,2))
# for(i in 1:p) {
#   plot(trainDf[[i]])
#   plot(log(trainDf[[i]]+1))
# }

trainDf$y = as.factor(trainDf$y)
testDf$y = as.factor(testDf$y)

### setup storage for results
```

```
phatL = list() #store the test phat for the different methods here

## get setup to run nn in h2o
h2oServer <- h2o.init(ip="localhost", port=54321, nthreads=4)
train_h2o = as.h2o(trainDf, destination_frame = "project_train")
test_h2o = as.h2o(testDf, destination_frame = "project_test")

### Fit neural net with one hidden layer
##if(file.exists(file.path("./", "model1"))) {
##  model1 = h2o.loadModel(path = file.path("./", "model1"))
##} else {
# model1 = h2o.deeplearning(
#   x=2:262, y=1,
#   training_frame=train_h2o,
#   hidden=5,
#   epochs=50,
#   export_weights_and_biases=T,
#   l1 = 1e-2,
#   model_id = "model1"
# )
# h2o.saveModel(model1, path="./")
##}

## Fit deep neural net with two hidden layers

deep.model = h2o.deeplearning(
  x=2:262, y=1,
  training_frame=train_h2o,
  hidden=c(20,20),
  epochs=50,
  # activation="RectifierWithDropout",
  activation="Tanh",
  l1=1e-3,
  export_weights_and_biases=TRUE,
  model_id = "deep.model"
)
h2o.saveModel(deep.model, path="./")


## Get phat
# phat = predict(model1, test_h2o)
phat = predict(deep.model, test_h2o)
phatL$h1n10 = as.matrix( phat[,3] )

#plot, compare to logit
# par(mfrow=c(1,2))
# plot(phatL$h1n10)
# abline(0,1,col="blue")
lift.many.plot(phatL, testDf$y)
# legend("topleft",legend=names(phatL),col=1:2,lty=rep(1,2),bty="n")
plot(h2o.performance(deep.model)) # Plot ROC curve

# Run deep.model on the test set
testperf = h2o.performance(deep.model,test_h2o)

# Plot ROC curve of deep.model
plot(testperf)

# Obtain confusion matrix from test results
h2o.confusionMatrix(testperf)

# Obtain AUC of ROC curve
```

```
h2o.auc(testperf)

# plot variable importance
h2o.varimp_plot(deep.model, num_of_features = 15)


#-------------------------------------------------------------
#-----------------  Logit for Discovery Data ----------------
#-------------------------------------------------------------

library(caret)


phatL2 = list() #store the test phat for the different methods here

# Removing bad or constant columns identified before by h2o

testDfclean = testDf
trainDfclean = trainDf
trainDfclean$x.T.P..I. <- NULL
trainDfclean$x.T.Br..I. <- NULL


###fit logit
lgfit = glm(y~.,trainDfclean,family=binomial)
print(summary(lgfit))
#predict using logit
phat2 = predict(lgfit,testDf,type="response")
phatL2$logit = matrix(phat2,ncol=1) #logit phat


source("mlfuns.R")

library(pROC)
par(mfrow=c(1,1))
temp=liftf(testDf$y,phatL2$logit[,1])
rocCurve = roc(response = testDf$y,predictor=phatL2$logit[,1])
plot(rocCurve)
cat("auc, logit: ", auc(rocCurve),"\n")

# NOTE: Caret package seems not able to generate confusionMatrix due to mixture of
# numeric and other variables in dataset


#-------------------------------------------------------------
#---------------  Boosting for Discovery Data ---------------
#-------------------------------------------------------------

phatL3 = list()

##settings for boosting
idv = c(2,4); ntv = c(500,1000); shv = c(.1,.01)
setboost = expand.grid(idv,ntv,shv)
colnames(setboost) = c("tdepth","ntree","shrink")
phatL3$boost = matrix(0.0,nrow(testDf),nrow(setboost))
trainDfB = trainDfclean; trainDfB$y = as.numeric(trainDfB$y)-1
testDfB = testDf; testDfB$y = as.numeric(testDfB$y)-1

##fit boosting

library(doParallel)
cl <- makeCluster(4)
```

```
registerDoParallel(cl)
#how many workers?
cat("number of workers is: ",getDoParWorkers(),"\n")

library(gbm)
tm1 = system.time({ #get the time, will use this later
  for(i in 1:nrow(setboost)) {
    cat("on boosting fit ",i,"\n")
    print(setboost[i,])
    ##fit and predict
    fboost = gbm(y~.,data=trainDfB,distribution="bernoulli",
                 n.trees=setboost[i,2],interaction.depth=setboost[i,1],
                 shrinkage=setboost[i,3])
    phat3 = predict(fboost,newdata=testDfB,n.trees=setboost[i,2],type="response")
    phatL3$boost[,i] = phat3
  }
})

stopCluster(cl)


#have to store all the phats in a list.
boostL = list()
for(i in 1:ncol(phatL3$boost)) boostL[[i]]= phatL3$boost[,i]
#get the lift
par(mfrow=c(1,1))
#plot(1:8,1:8)
#for(i in 1:8) abline(v=i,col=i,lwd=2)
temp = liftf(testDf$y,phatL3$boost[,1])

rocCurve = roc(response = testDf$y,predictor=phatL3$boost[,1])
plot(rocCurve)
cat("auc, boost: ", auc(rocCurve),"\n")

# NOTE: Caret package seems not able to generate confusionMatrix due to size

# plot variable importance in boosting
varImpPlot(fboost)



#--------------------------------------------------------------------------
#------------------  Random Forests for Discovery Data  --------------------
#--------------------------------------------------------------------------

phatL4 = list()

##settings for randomForest
p=ncol(trainDf)-1
#mtryv = c(p,sqrt(p))  # for discovery dataset, p = 261 is too large for mtry
mtryv = sqrt(p)  # this is default value for mtry
ntreev = c(500,1000)
setrf = expand.grid(mtryv,ntreev)
colnames(setrf)=c("mtry","ntree")
phatL4$rf = matrix(0.0,nrow(testDf),nrow(setrf))

###fit rf
library(randomForest)
for(i in 1:nrow(setrf)) {
  cat("on randomForest fit ",i,"\n")
  print(setrf[i,])
  #fit and predict
```

```
  frf = randomForest(y~.,data=trainDf,mtry=setrf[i,1],ntree=setrf[i,2])
  phat4 = predict(frf,newdata=testDf,type="prob")[,2]
  phatL4$rf[,i]=phat4
}


# plot lift and ROC Curves
library(pROC)
par(mfrow=c(1,1))
temp=liftf(testDf$y,phatL4$rf[,1])
rocCurve = roc(response = testDf$y,predictor=phatL4$rf[,1])
plot(rocCurve)

# get AUC of ROC curve
cat("auc, RandomForest: ", auc(rocCurve),"\n")

# plot variable importance in randomForest
varImpPlot(frf)
```