Disease Prediction with Gene Expression Data

Eugine Kang

Sunday, August 30, 2015

The aggregation of quality health-related data is paramount to the success of all disease detection initiatives. Without correct and current data, diseases are misunderstood, and undetected. Functioning detection systems are necessary for the success of global health. Gene expression data is the key to detecting diseases, but the difficulty comes due to too much data. This analysis will show how to build a quality model with high-dimensional gene expression data. There are many machine algorithms tested for this analysis, but random forest showed the best performance among the models compared. The analysis goes through 4 stages (1. Data Cleaning, 2. Feature Selection, 3. Compare Models, 4. Optimization) until the final model predicts the outcome for the test dataset.

1. Data Cleaning

```
dim(data)
```

```
## [1]   184 12066
```

The data is high-dimensional, meaning the number of variables far exceed the number of samples. Classicial approaches such as least squares linear regression are not appropriate in this setting, and other approaches have the danger of overfitting.

We can reduce the dimension by feature selection and dimension reduction techniques, but let's first see if we can discard variables with little information.

```
var0 <- nearZeroVar(data)
length(var0)
```

```
## [1] 3245
```

3245 variables have zero or near zero variance. We decide to discard these variables, because they won't be useful when classifying disease.

```
dataZero <- data[,-var0]
noNA <- sapply(dataZero, function(x) sum(is.na(x)))
table(noNA)
```

```
## noNA
##    0    1
## 8815    6
```

6 columns have missing values. We only have 184 samples, so removing the samples with missing values will be too valuable of a waste. Imputation fills in the missing values, and will allow us to use every sample to train our model

```
dataImpute <- knnImputation(dataZero, k=10)
```

Multicollinearity, the concept that the variables in a regression might be correlated with each other. In the high-dimensional setting, the multicollinearity problem is extreme. Any variable in the model can be written as a linear combination of all of the other variables in the model. Removing variables with a correlation value above 0.7 is an option we can take.

```
dataScale <- scale(dataImpute, center=TRUE, scale=TRUE)
corMat <- cor(dataScale)
highlyCor <- findCorrelation(corMat, 0.70)
length(highlyCor)
```

```
## [1] 873
```

873 variables are removed due to high correlation with other variables. Variables with low information or high correlation are removed. Let's try to select important variables to our models by feature selection.

```
dataFilter <- as.data.frame(dataScale[,-highlyCor])
dataFilter$Y <- ifelse(label == 1, 1,0)
dataFilter$Y <- as.factor(dataFilter$Y)
```

2. Feature Selection

We will be using Random Forest to find out the importance of each variables for our classification model. 10 fold cross validation is the training scheme for our feature selection method.

```
control <- trainControl(method="repeatedcv", number=10, repeats=1)
```

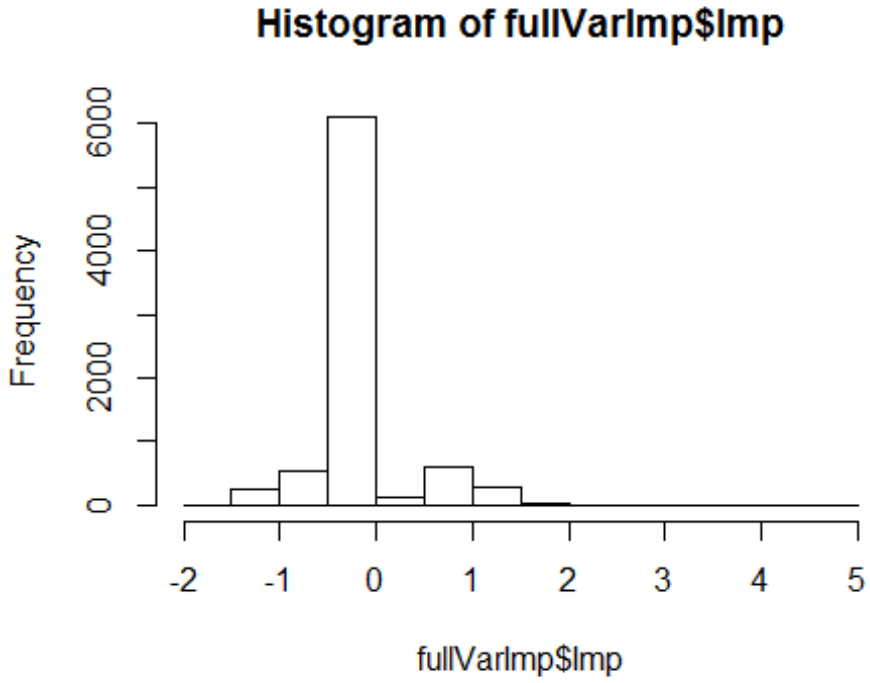
Random Forest is a popular method for feature selection for high-dimensional data. We set the seed for every random iteration so that we can reproduce the results.

```
set.seed(1)
rfModel <- train(Y~., data=datFilter,
                 method="rf",
                 trControl=control,
                 importance=TRUE,
                 do.trace=FALSE
)
# Organize Feature Importance Data
fullImportance <- varImp(rfModel, scale=FALSE)
fullVarImp <- fullImportance$importance
fullVarImp$varName <- row.names(fullVarImp)
```

```
colnames(fullVarImp) <- c("Imp","Imp2","varName")
fullVarImp <- fullVarImp[order(fullVarImp$Imp, decreasing=T),]
```

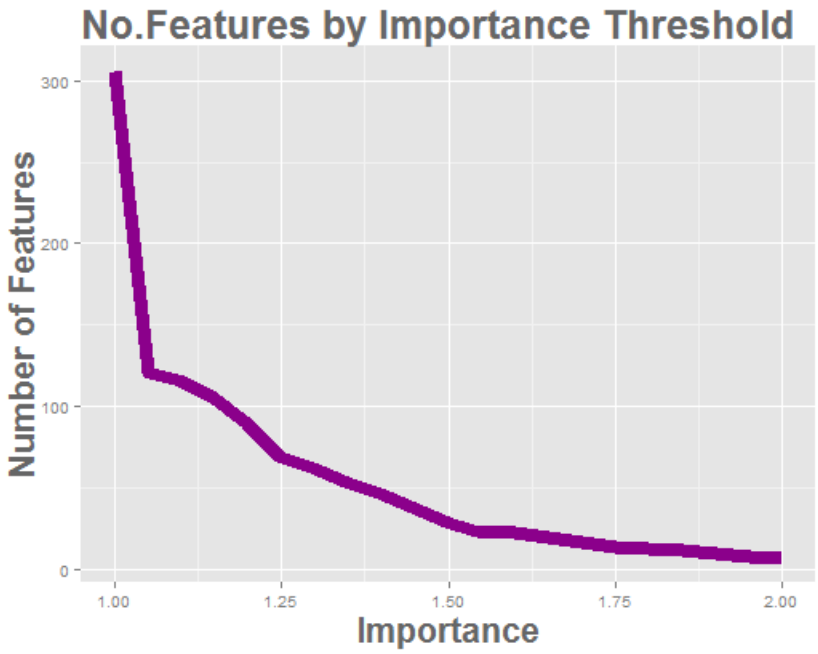
From training a Random Forest model, we get importance levels for each variables that help the classification model. The distribution of importance level is shown by the histogram, and the top variables have an importance higher than 1.

```
hist(fullVarImp$Imp)
```



Our goal is to select the fewest amount of variables, so that we can test more type of models and keep the performance. The number of features selected by importance is shown through the graph.

```
lvl <- seq(1,2,0.05)
noFeatvsImp <- data.frame(Imp=lvl,
                          noFeat=sapply(lvl, function(x) sum(fullVarImp$Imp>x
))
                          )
ggplot(data=noFeatvsImp, aes(x=Imp,y=noFeat)) +
        geom_line(colour="darkmagenta", size=3) +
        ggtitle("No.Features by Importance Threshold") +
        labs(x="Importance",y="Number of Features") +
        theme(plot.title = element_text(color="#666666", face="bold", size=25
, hjust=0)) +
        theme(axis.title = element_text(color="#666666", face="bold", size=22
))
```



For a test run, we will select variables with importance higher than 1.1.

3. Compare Models

We will be testing the performance of 9 models with the selected features with an importance greater than 1.1.

```
featSelect <- function(thres){
        return(fullVarImp[fullVarImp$Imp>thres,"varName"])
        }
featSet <- featSelect(1.1)
# Independent Variables for Model
X <- dataFilter[,featSet]
train <- 1:nrow(X)
# Dependent Variables for Model
Y <- dataFilter$Y
levels(Y)[1] <- make.names(levels(Y))[1]
levels(Y)[2] <- make.names(levels(Y))[2]
```

The models will go through 10-fold cross-validation, and the performance of the models are evaluated through the ROC value. When considering the real life impact of disease detection, we don't want to miss anyone from detection especially when the disease is severe. Sensitivity measures the proportion of positives that are correctly identified, and specificity measures the proportion of negatives that are correctly identified. Metric that

evaluates this performance is ROC, receiver operating characteristic. Our goal is to find a model with a ROC higher than 0.9.

```
folds=10
repeats=1
# ROC will be the metric to evaluate the model "summaryFunction=twoClassSumma
ry"
myControl <- trainControl(method='cv', number=folds, repeats=repeats,
                          returnResamp='none', classProbs=TRUE,
                          returnData=FALSE, savePredictions=TRUE,
                          verboseIter=TRUE, allowParallel=TRUE,
                          summaryFunction=twoClassSummary,
                          index=createMultiFolds(try$Y, k=folds, times=repeat
s))
PP <- c('center', 'scale')
```

The 9 models that we will compare are listed below with the code.

```
set.seed(1)
# Stochastic Gradient Boosting
mdlGBM <- train(X[train,], Y[train],
                method='gbm',
                trControl=myControl,
                tuneLength=10,
                preProcess=PP)
# Boosted Tree
mdlBLACKBOOST <- train(X[train,], Y[train],
                       method='blackboost',
                       trControl=myControl,
                       tuneLength=10,
                       preProcess=PP)
# Random Forest
mdlPARRF <- train(X[train,], Y[train],
                  method='parRF',
                  trControl=myControl,
                  tuneLength=10,
                  preProcess=PP)
# Multi-Layer Perceptron
mdlMLP <- train(X[train,], Y[train],
                method='mlpWeightDecay',
                trControl=myControl,
                trace=FALSE,
                tuneLength=3,
                preProcess=PP)
# k-Nearest Neighbors
mdlKNN <- train(X[train,], Y[train],
                method='knn',
                trControl=myControl,
                tuneLength=10,
```

```
                  preProcess=PP)
# Multivariate Adaptive Regression Spline
mdlEARTH <- train(X[train,], Y[train],
                  method='earth',
                  trControl=myControl,
                  tuneLength=10,
                  preProcess=PP)
# Generalized Linear Model
mdlGLM <- train(X[train,], Y[train],
                method='glm',
                trControl=myControl,
                preProcess=PP)
# Support Vector Machines with Radial Basis Function Kernel
mdlSVM <- train(X[train,], Y[train],
                method='svmRadial',
                trControl=myControl,
                tuneLength=10,
                preProcess=PP)
# Lasso and Elastic-Net Regularized Generalized Linear Models
mdlGLMNET <- train(X[train,], Y[train],
                   method='glmnet',
                   trControl=myControl,
                   tuneLength=10,
                   preProcess=PP)
```

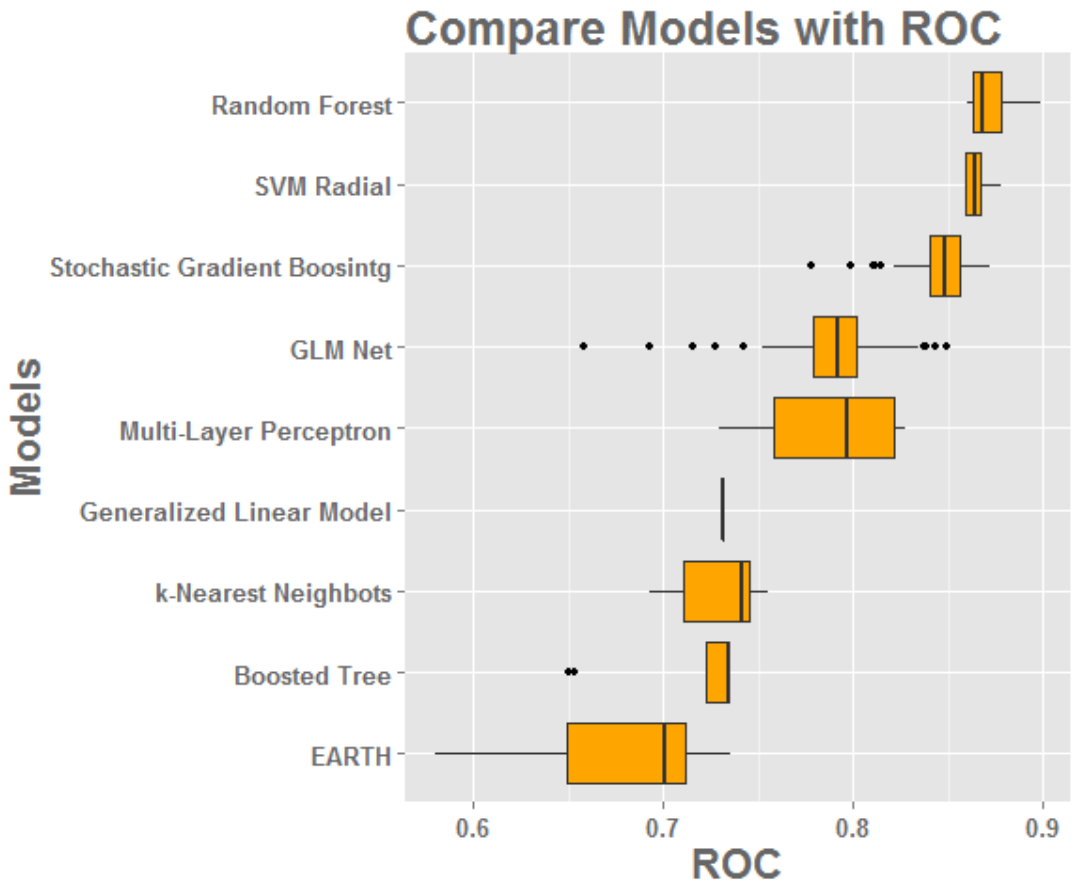
Let's organize the results from each model and compare the performance.

```
models <- c("Stochastic Gradient Boosintg", "Boosted Tree", "Random Forest",
            "Multi-Layer Perceptron", "k-Nearest Neighbots", "EARTH",
            "Generalized Linear Model", "SVM Radial", "GLM Net")
modelList <- list(mdlGBM, mdlBLACKBOOST, mdlPARRF,
                  mdlMLP, mdlKNN, mdlEARTH,
                  mdlGLM, mdlSVM, mdlGLMNET)
modelResult <- data.frame()
for(i in 1:length(models)){
        resultDF <- modelList[[i]]$result[,c("ROC","Sens","Spec")]
        resultDF$Model <- models[i]
        modelResult <- rbind(modelResult,resultDF)
}

# Order models with highest mean ROC value
modelResultDT <- data.table(modelResult)
meanROC <- as.data.frame(modelResultDT[,list(Mean=mean(ROC),Max=max(ROC)),by=
Model])
meanROC <- meanROC[order(meanROC$Mean),]
modelResult$Model <- factor(modelResult$Model,
                        levels = meanROC$Model,ordered = TRUE)
```

The horizontal boxplot showing the ROC distribution for each model shows Random Forest as the best performing model for this model. However, we are not quite at the level we desire. ROC value has not reached 0.9. Through optimization of feature selection and parameter adjustment, we will enhance the performance.

```
ggplot(modelResult, aes(x=Model, y=ROC)) + geom_boxplot(fill="orange") +
        coord_flip() +
        ggtitle("Compare Models with ROC") +
        labs(x="Models",y="ROC") +
        theme(plot.title = element_text(color="#666666", face="bold", size=25
, hjust=0)) +
        theme(axis.title = element_text(color="#666666", face="bold", size=22
)) +
        theme(axis.text = element_text(color="#666666", face="bold", size=13)
)
```

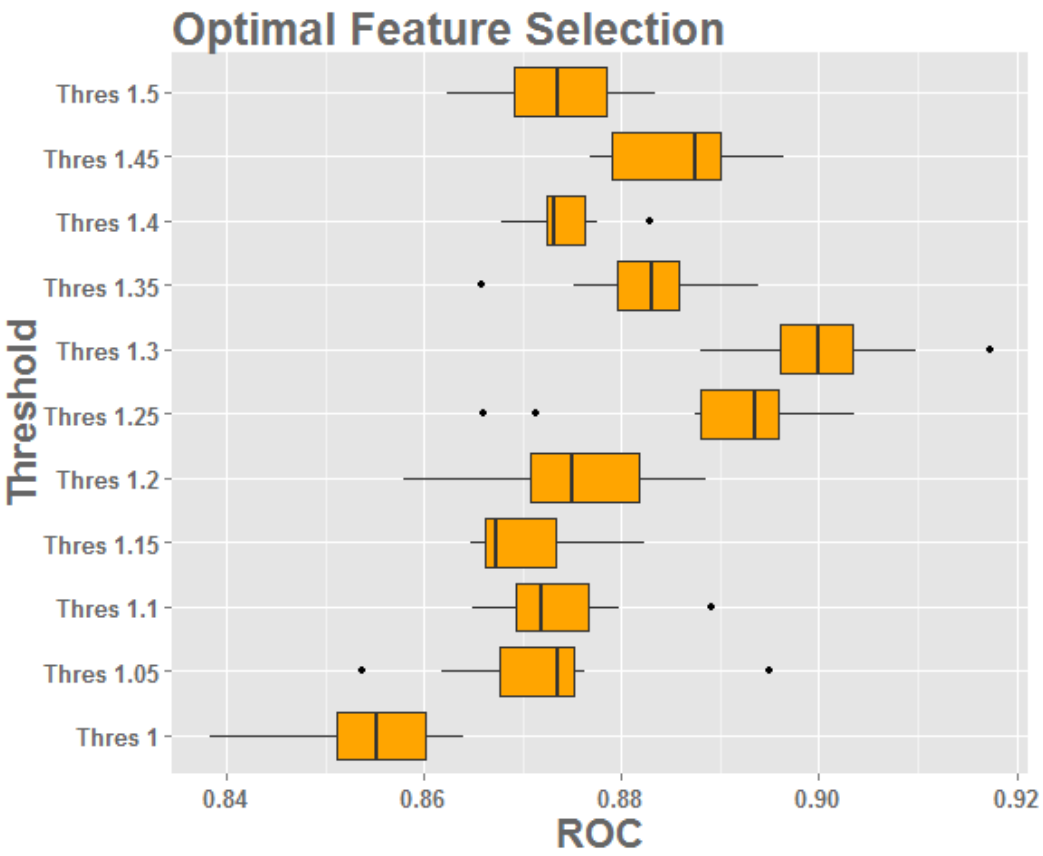


4. Optimization

Let's see if increasing or decreasing the variables selected will enhance the performance of the model. We will go through a range of importance level and see how the performance of the model changes.

```
# Adjust features selected and evaluste with Random Forest
testPerformance <- function(thres){
        # Prepare data.frame for model
        featSet <- featSelect(thres)
        X <- dataFilter[,featSet]
        # Run model
        model <- train(X[train,], Y[train],
                        method='parRF',
                        trControl=myControl,
                        tuneLength=10,
                        preProcess=PP)
        # Return result
        return(model$result)
}
thresLvl <- seq(1,1.5,0.05)
set.seed(1)
testRange <- lapply(thresLvl, function(x) testPerformance(x))
testResult <- data.frame()
for(i in 1:length(thresLvl)){
        df <- testRange[[i]]
        df$Thres <- paste("Thres",thresLvl[i])
        testResult <- rbind(testResult,df)
}

ggplot(testResult, aes(x=Thres, y=ROC)) + geom_boxplot(fill="orange") +
        coord_flip() +
        ggtitle("Optimal Feature Selection") +
        labs(x="Threshold",y="ROC") +
        theme(plot.title = element_text(color="#666666", face="bold", size=25
, hjust=0)) +
        theme(axis.title = element_text(color="#666666", face="bold", size=22
)) +
        theme(axis.text = element_text(color="#666666", face="bold", size=13)
)
```



The importance level that gives us the best performance is 1.3. 62 variables are selected from the original dataset, but we can further improve the performance by adjusting the model's parameter.

```
length(featSelect(1.3))
```

```
## [1] 62
```

```
# What is the best parameter?
testRange[[7]]
```

```
##    mtry       ROC      Sens      Spec      ROCSD     SensSD    SpecSD
## 1     2 0.9098214 0.9583333 0.6000000 0.07668885 0.05892557 0.2554027
## 2     8 0.8905213 0.9075758 0.5833333 0.09445684 0.07290942 0.2870888
## 3    15 0.8958424 0.9234848 0.6309524 0.08654355 0.09349916 0.2228591
## 4    22 0.9012446 0.8984848 0.6309524 0.07260267 0.07817686 0.2515385
## 5    28 0.8985480 0.9068182 0.6261905 0.06572599 0.08451489 0.2213845
## 6    35 0.9037067 0.9068182 0.6476190 0.03674168 0.08451489 0.2563873
## 7    42 0.8970689 0.9075758 0.6285714 0.09345777 0.07290942 0.2729051
## 8    48 0.9027327 0.9151515 0.5952381 0.06219889 0.08955216 0.2629848
## 9    55 0.9172980 0.9068182 0.6285714 0.07017463 0.08451489 0.2202864
## 10   62 0.8880682 0.8977273 0.6571429 0.07580759 0.09879193 0.2492619
```

From the given parameter options, mtry = 55 shows the highest performance. Let's go deeper to determine the exact value for mtry that gives the best performance.

```
repeats = 3
set.seed(1)
X <- dataFilter[,featSelect(1.3)]
optModel <- train(X[train,], Y[train],
                  method='parRF',
                  trControl=myControl,
                  tuneGrid=expand.grid(.mtry=48:60),
                  preProcess=PP)

optModel$result
```

```
##    mtry       ROC      Sens      Spec      ROCSD     SensSD    SpecSD
## 1    48 0.9002345 0.9325758 0.5785714 0.06750470 0.06599080 0.2769944
## 2    49 0.8971320 0.8984848 0.6285714 0.05253412 0.10364001 0.2492619
## 3    50 0.9047258 0.8984848 0.6166667 0.05642427 0.08749189 0.1916137
## 4    51 0.8818994 0.9060606 0.6095238 0.07309963 0.09623830 0.2502707
## 5    52 0.9017226 0.9151515 0.5833333 0.06319356 0.08955216 0.2306372
## 6    53 0.8850469 0.9143939 0.6142857 0.08810999 0.10814746 0.2385710
## 7    54 0.9002345 0.9159091 0.5952381 0.06320964 0.07860394 0.2383596
## 8    55 0.8966089 0.9068182 0.6571429 0.07573409 0.08451489 0.2492619
## 9    56 0.9042749 0.9151515 0.5952381 0.04040431 0.08955216 0.2876916
## 10   57 0.8992424 0.9159091 0.5976190 0.06723584 0.06808354 0.2616521
## 11   58 0.8857864 0.8893939 0.5666667 0.07391299 0.10051647 0.2634156
## 12   59 0.8986742 0.9151515 0.6285714 0.06111200 0.08955216 0.2492619
## 13   60 0.8936688 0.8977273 0.5952381 0.08176576 0.11334137 0.2542161
```

We can see mtry 50 or 56 performs the best. The difference between those two parameters do not seem to be significant. We will decide on mtry value 56 for our final model.

```
set.seed(1)
finalModel <- train(X[train,], Y[train],
                    method='parRF',
                    trControl=myControl,
                    tuneGrid=expand.grid(.mtry=56),
                    preProcess=PP)
```

The final model will predict the disease status from the test dataset. The results from the prediction can be found at
https://github.com/kangeugine/gene_expression/blob/master/testPrediction.txt