Nathanael Roy

Statistics 208 - Final Project

Classification Methods: Pulsar Stars

Introduction

In this project, I will be using a variety of Statistical Data Mining methods on a data set of pulsar stars. The main question of interest is using different attribute characteristics within the data set to attempt to classify the target attribute which is binary. The main programming language used on this data set is R, although I have also attempted a couple of the methods (tree based) in python as well to learn some of the functionality though this is not discussed here since the results are identical. The main concern of interest is which algorithm does the best job at classifying the data with secondary interests including which algorithms run faster.

Description of the Data

This data is “HTRU2” which I pulled from Kaggle, and it comes from the High Time Resolution Universe Survey (R. J. Lyon). This is labeled data, containing mostly negative examples of 16,259 rows of features associated with candidates deemed not to be pulsar stars. About 9% of the examples, 1,639 are examples of pulsar stars. In this data set there are eight continuous variable features along with the class variable labeling it as a pulsar star or not. The first four variables are statistics about the integrated pulse profile (IP) data associated with a given possible star- referring to the mean, standard deviation, kurtosis, and skewness- while the last four characteristics are from the Dispersion Measure- Signal to Noise Ratio Curve (DM-SNR) which is considered the second best feature for classifying pulsar stars (Lyon).

This data set has been extensively covered by physicists in the interests of attempting to classify candidate pulsar stars ahead of time from the millions of candidate possibilities generated by an ever expanding explosion of data. The methods discussed in this current review of methods would likely fail on modern data streams by being too simplistic, ignoring the details of astrophysics, and returning far too many false positives. However, due to the domain, special attention will be paid to the false positive rate.

The table below shows some descriptive statistics of the eight variables of interest (each row corresponds to one of the variables) along with their mean and standard deviation for the whole data set and the data set split up by mean and standard deviation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Mean | SD | Mean + | SD + | Mean - | SD - |
| Mean IP | 111.1 | 25.65 | 56.69 | 30.01 | 116.6 | 17.48 |
| Standard Deviation IP | 46.55 | 6.843 | 38.71 | 8.034 | 47.34 | 6.183 |
| Excess Kurtosis IP | .4779 | 1.064 | 3.121 | 1.873 | .2104 | .3346 |
| Skewness IP | 1.770 | 6.168 | 15.55 | 14.00 | .3808 | 1.028 |
| Mean DM-SNR | 12.61 | 29.47 | 49.83 | 45.29 | 8.863 | 24.41 |
| Standard Deviation DM-SNR | 26.33 | 19.47 | 56.47 | 19.73 | 23.29 | 16.65 |
| Excess Kurtosis DM-SNR | 8.304 | 4.506 | 2.757 | 3.106 | 8.863 | 4.239 |
| Skewness DM-SNR | 104.9 | 106.5 | 17.93 | 50.90 | 113.6 | 106.7 |

We can see right away that the positive and negative examples may be separable based on these descriptive statistics of the incoming data since the columns for the means of the positive and negative examples look quite a bit different from each other.

Comparing Methods

I performed five methods on this data set. For most of the methods I attempted some amount of tuning as long as time permitted and measured the error rate as well as the amount of time needed to perform the method

The first method attempted was a support vector classifier, using the svm function in R and utilizing a linear classifier. In terms of length of run time this was the most involved method and parameter tuning wasn’t that large a priority because of the amount of time it took to do even one run would have meant that tuning over a grid of parameters to optimize would have taken a great deal of time, although scaling the data seems to have sped up the svm a great deal for some reason. I attempted to fit an svm with a quadratic boundary to classify the data as well but after leaving the code running for a very long time it still would not run.

The second method I attempted was K-Nearest-Neighbors. This method is one of the most intuitive machine learning methods where the use of majority voting is used by the nearest labeled methods. I applied tuning across a number of different values of k and found that between 1 and 10 a value of k=7 performed best.

The third method involved a simple logistic regression. For this we are simply fitting a general linear model which does not have hyperparameters in general. Future considerations may include applying LASSO or Ridge like shrinkage methods to the parameters in logistic regression. Looking at the coefficients all but 2 of the estimates were significant in the logistic regression model.

The fourth method involved using a neural network with one hidden layer. As this is a more complicated model it naturally takes a bit longer than logistic regression but because of the collapse to logistic regression in the case where there are 0 hidden layers I expected to do a little bit better. Initially the differences were not substantial, and just attempting a random size didn’t yield substantially different results. Tuning the size of the hidden layer led to a hidden layer of size 7 which did manage to make the neural network the best model in terms of false positive rates.

The fifth and final method involved fitting a tree-based method. An initial fit of a tree to the data yielded the same very simple two node split with whether Excess Kurtosis was above .66 (or .64 in one of the cross-validation methods, both of these are scaled values). This was a very simple split but yielded a surprisingly competitive model to the other methods explored. An attempt was made to tune the minimum split parameters yielding no new model. However, then the cost function for the amount needed in order to add a split was modified and substantially larger trees appeared.

Here is a table summarizing the results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method: | False Positive | False Negative | Overall Misclassification Rate | Time (seconds) |
| SVM | .0176 | .0050 | .0208 | 3.28 |
| KNN | .0176 | .0061 | .0217 | 1.61 |
| Logistic Regression | .0174 | .0054 | .0174 | .4 |
| Neural Network | .0140 | .0069 | .0191 | 13 |
| Tree Classification (Simple) | .0176 | .0067 | .0222 | .97 |
| Tree Classification (Complex) | .0142 | .0062 | .0187 | 1.81 |

From the table above, the neural network takes by far the most amount of time and the complex tree classification seems to do the best job at predicting. Most of the methods have similar false positive rates of around 1.8% with the neural network and complex tree classifier doing a better job of around 1.4%. The overall error rate is also clustered around 2% for most of the methods with the logistic regression being the best performing method despite not doing as well on the false positive rate.

I would suggest that a tree based method may be the best especially given that in the reference “Why Are Pulsars Hard to Find?” the main method used to classify stars as pulsar stars is a tree based method. Since false positives are the main concern as well trees can easily be modified in order to create a larger cost for false positives than for false negatives to the potentially desired point on an ROC curve.

# Appendix: R Code

#Libraries to use

library(MASS)

library(class)

library(e1071)

library(nnet)

library(rpart)

library(rpart.plot)

library(rattle)

total = 8

# This bit of code sets up the working directory and get the data in

scriptPath <- function() {

getSrcDirectory(scriptPath);

}

wd=scriptPath()

setwd(wd)

#Reading the data into our R envrionment and setting up the data is done here

pulsar = read.csv("pulsar\_stars.csv")

pulsar = pulsar[order(pulsar$target\_class),]

#Drawing some plots to look at the density of the covariates

# par(mfrow=c(1, 2))

# ps = pulsar[which(pulsar$target\_class==1),]

# nps = pulsar[which(pulsar$target\_class==0),]

# colnames <- c("Mean IP","SD IP", "Excess Kurtosis IP", "Skew DM SNR","Mean DM SNR","SD DM SNR", "Excess Kurtosis DM SNR", "Skew DM SNR")

# for (i in 1:8) {

#

# d.pos = density(ps[,i])

# d.neg = density(nps[,i])

# hist(ps[,i], probability =TRUE, main=paste("Positive Density",colnames[i]))

# lines(d, col="red")

# hist(nps[,i],probability = TRUE,main=paste("Negative Density",colnames[i]))

# lines(d, col="red")

# }

#Setting the seed so we can get similar results every time

set.seed(47)

#Use about a fifth of the data for each cross validation set

rows = 1:17898

test\_index1 = sample(1:17898,3580)

rest1 = rows[-test\_index1]

test\_index2 = sample(rest1,3580)

rest2 = rows[-test\_index2]

test\_index3 = sample(rows[-c(test\_index1,test\_index2)],3580)

rest3 = rows[-test\_index3]

test\_index4 = sample(rows[-c(test\_index1,test\_index2,test\_index3)],3579)

rest4 = rows[-test\_index4]

test\_index5 = rows[-c(test\_index1,test\_index2,test\_index3,test\_index4)]

rest5 = rows[-test\_index5]

#Descriptive Statistics about the data and different variables in the data set:

sapply(pulsar,mean)

sapply(pulsar,sd)

pos = which(pulsar$target\_class==1)

sapply(pulsar[pos,],mean)

sapply(pulsar[pos,],sd)

sapply(pulsar[-pos,],mean)

sapply(pulsar[-pos,],sd)

#Useful Functions:

ConfusionMatrix = function(p1,p2,p3,p4,p5){

C\_P = as.data.frame(matrix(nrow=17898,ncol=2))

colnames(C\_P) = c("True","Predicted")

C\_P[,1] = pulsar[,9]

C\_P[test\_index1,2] = p1

C\_P[test\_index2,2] = p2

C\_P[test\_index3,2] = p3

C\_P[test\_index4,2] = p4

C\_P[test\_index5,2] = p5

C.Matrix = table(C\_P[,1],C\_P[,2])

if(dim(C.Matrix)[2]==1){

print(C.Matrix)

return(C.Matrix)

}

else{

colnames(C.Matrix) <- c("Predicted=0","Predicted=1")

rownames(C.Matrix) <- c("True=0","True=1")

print(C.Matrix)

FP = C.Matrix[2,1]

TP = C.Matrix[2,2]

FN = C.Matrix[1,2]

TN = C.Matrix[1,1]

#Calculating the False Positive:

FPR = FP/(FP+TN)

FNR = FN/(FN+TN)

Error.Rate = (FP+FN)/(FP+FN+TP+TN)

print(paste("The False Positive Rate is ",FPR))

print(paste("The False Negative Rate is ",FNR))

print(paste("The Error Rate is ",Error.Rate))

return(C.Matrix)

}

}

#Scaling and centering data:

pulsar[,1:8] = scale(pulsar[,1:8])

#in order to time how long our code takes

time.tests = rep(0,5)

names(time.tests) = c("SVM","KNN","Logistic Regression","Neural Network","Tree")

#

# #First Method: Linear and Quadratic Discriminant Analysis

#Commented out to allow code to run faster

start.time = proc.time()

svmfit1 = svm(as.factor(target\_class)~.,data=pulsar[rest1,],kernel="linear",scale=F)

svmfit2 = svm(as.factor(target\_class)~.,data=pulsar[rest2,],kernel="linear",scale=F)

svmfit3 = svm(as.factor(target\_class)~.,data=pulsar[rest3,],kernel="linear",scale=F)

svmfit4 = svm(as.factor(target\_class)~.,data=pulsar[rest4,],kernel="linear",scale=F)

svmfit5 = svm(as.factor(target\_class)~.,data=pulsar[rest5,],kernel="linear",scale=F)

svmpredict1 = predict(svmfit1,newdata = pulsar[test\_index1,])

svmpredict2 = predict(svmfit2,newdata = pulsar[test\_index2,])

svmpredict3 = predict(svmfit3,newdata = pulsar[test\_index3,])

svmpredict4 = predict(svmfit4,newdata = pulsar[test\_index4,])

svmpredict5 = predict(svmfit5,newdata = pulsar[test\_index5,])

print("SVM Confusion Matrix")

ConfusionMatrix(svmpredict1,svmpredict2,svmpredict3,svmpredict4,svmpredict5)

time.tests[1] = (proc.time() - start.time)[3]

start.time = proc.time()

#Second method: K-Nearest Neighbors

#k\_tuned =tune.knn(x=pulsar[,-9],y=as.factor(pulsar[,9]),k=1:10,tunecontrol=tune.control(sampling="cross",cross=5))

#Tuning gives k of 7

k=7

knn1 <- knn(train = pulsar[rest1,1:8],test = pulsar[test\_index1,1:8],cl=pulsar[rest1,9],k=k)

knn2 <- knn(train = pulsar[rest2,1:8],test = pulsar[test\_index2,1:8],cl=pulsar[rest2,9],k=k)

knn3 <- knn(train = pulsar[rest3,1:8],test = pulsar[test\_index3,1:8],cl=pulsar[rest3,9],k=k)

knn4 <- knn(train = pulsar[rest4,1:8],test = pulsar[test\_index4,1:8],cl=pulsar[rest4,9],k=k)

knn5 <- knn(train = pulsar[rest5,1:8],test = pulsar[test\_index5,1:8],cl=pulsar[rest5,9],k=k)

print("KNN Confusion Matrix with 7 nearest neighbors")

ConfusionMatrix(knn1,knn2,knn3,knn4,knn5)

time.tests[2] = (proc.time() - start.time)[3]

start.time = proc.time()

#Third Method: Logistic Regression Classifiers

logitmodel1 = glm(target\_class~.,data=pulsar[rest1,],family="binomial")

glm1 = round(predict.glm(logitmodel1,type="response",newdata=pulsar[test\_index1,1:8]))

logitmodel2 = glm(target\_class~.,data=pulsar[rest2,],family="binomial")

glm2 = round(predict.glm(logitmodel2,type="response",newdata=pulsar[test\_index2,1:8]))

logitmodel3 = glm(target\_class~.,data=pulsar[rest3,],family="binomial")

glm3 = round(predict.glm(logitmodel3,type="response",newdata=pulsar[test\_index3,1:8]))

logitmodel4 = glm(target\_class~.,data=pulsar[rest4,],family="binomial")

glm4 = round(predict.glm(logitmodel4,type="response",newdata=pulsar[test\_index4,1:8]))

logitmodel5 = glm(target\_class~.,data=pulsar[rest5,],family="binomial")

glm5 = round(predict.glm(logitmodel5,type="response",newdata=pulsar[test\_index5,1:8]))

print("Logistic Regression Confusion Matrix")

ConfusionMatrix(glm1,glm2,glm3,glm4,glm5)

time.tests[3] = (proc.time() - start.time)[3]

start.time = proc.time()

# Fourth Method: Neural Network Classifiers

#tune.nnet(x=pulsar[,1:8],y=pulsar[,9],data=pulsar,size=1:8,trace=FALSE,tunecontrol = tune.control(sampling="cross",cross=5))

#Suggests a size of 7

sz=7

nnetfit1 = nnet(x=pulsar[,1:8],y=class.ind(pulsar$target\_class),data=pulsar,size=sz,subset=rest1,softmax = T,trace=FALSE)

nnetfit2 = nnet(x=pulsar[,1:8],y=class.ind(pulsar$target\_class),data=pulsar,size=sz,subset=rest2,softmax = T,trace=FALSE)

nnetfit3 = nnet(x=pulsar[,1:8],y=class.ind(pulsar$target\_class),data=pulsar,size=sz,subset=rest3,softmax = T,trace=FALSE)

nnetfit4 = nnet(x=pulsar[,1:8],y=class.ind(pulsar$target\_class),data=pulsar,size=sz,subset=rest4,softmax = T,trace=FALSE)

nnetfit5 = nnet(x=pulsar[,1:8],y=class.ind(pulsar$target\_class),data=pulsar,size=sz,subset=rest5,softmax = T,trace=FALSE)

nfit1 = predict(nnetfit1,pulsar[test\_index1,])

nfit2 = predict(nnetfit2,pulsar[test\_index2,])

nfit3 = predict(nnetfit3,pulsar[test\_index3,])

nfit4 = predict(nnetfit4,pulsar[test\_index4,])

nfit5 = predict(nnetfit5,pulsar[test\_index5,])

nfit1 = round(nfit1[,2])

nfit2 = round(nfit2[,2])

nfit3 = round(nfit3[,2])

nfit4 = round(nfit4[,2])

nfit5 = round(nfit5[,2])

print("Neural Network Confusion Matrix")

ConfusionMatrix(nfit1,nfit2,nfit3,nfit4,nfit5)

time.tests[4] = (proc.time() - start.time)[3]

start.time = proc.time()

# Fifth Method: Tree based Classifiers

#tune.rpart(target\_class~.,data=pulsar,minsplit=1:10,tunecontrol = tune.control(sampling="cross",cross=5))

#tune.rpart(target\_class~.,data=pulsar,cp=seq(.0001,.01,.0005),tunecontrol = tune.control(sampling="cross",cross=5))

#The tuning gives a minimum split of 2 and a cp of .0021, removing these gives the simple tree

fittree1 = rpart(target\_class~.,data=pulsar,minsplit=2,cp= .0021,subset=rest1,method="class")

fittree2 = rpart(target\_class~.,data=pulsar,minsplit=2,cp= .0021,subset=rest2,method="class")

fittree3 = rpart(target\_class~.,data=pulsar,minsplit=2,cp= .0021,subset=rest3,method="class")

fittree4 = rpart(target\_class~.,data=pulsar,minsplit=2,cp= .0021,subset=rest4,method="class")

fittree5 = rpart(target\_class~.,data=pulsar,minsplit=2,cp= .0021,subset=rest5,method="class")

fancyRpartPlot(fittree5) #All the trees end up looking pretty identical

tfit1 = predict(fittree1,pulsar[test\_index1,])

tfit2 = predict(fittree1,pulsar[test\_index2,])

tfit3 = predict(fittree1,pulsar[test\_index3,])

tfit4 = predict(fittree1,pulsar[test\_index4,])

tfit5 = predict(fittree1,pulsar[test\_index5,])

tfit1 = round(tfit1[,2])

tfit2 = round(tfit2[,2])

tfit3 = round(tfit3[,2])

tfit4 = round(tfit4[,2])

tfit5 = round(tfit5[,2])

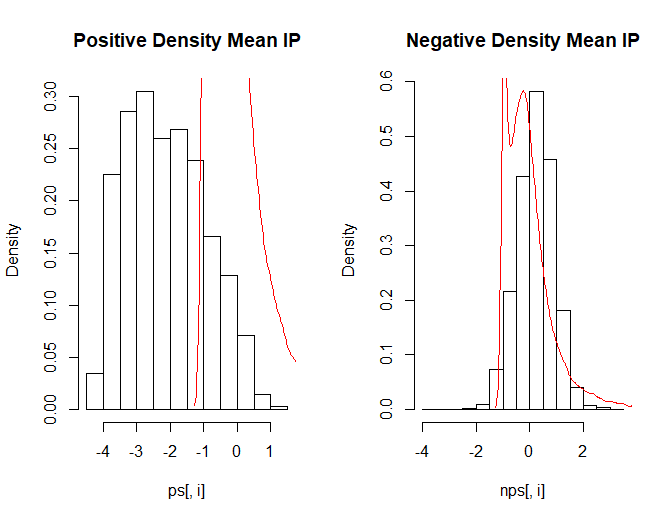
time.tests[5] = (proc.time() - start.time)[3]

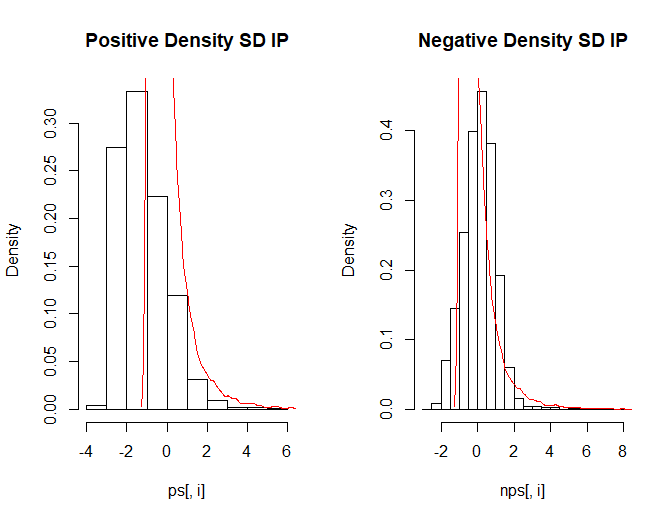
print("Tree Confusion Matrix")

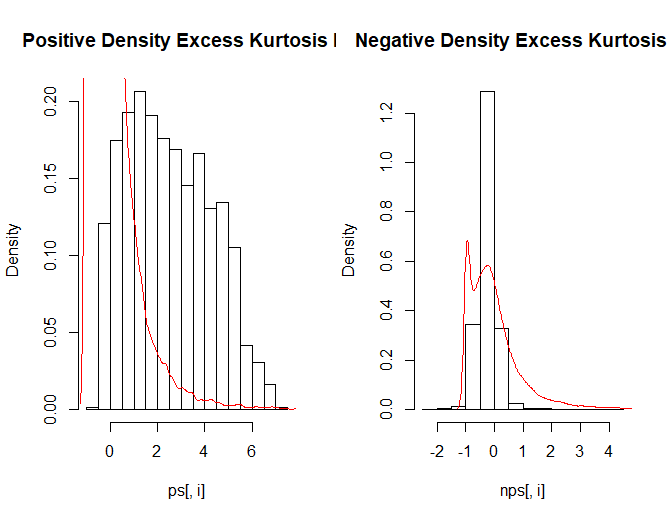
ConfusionMatrix(tfit1,tfit2,tfit3,tfit4,tfit5)

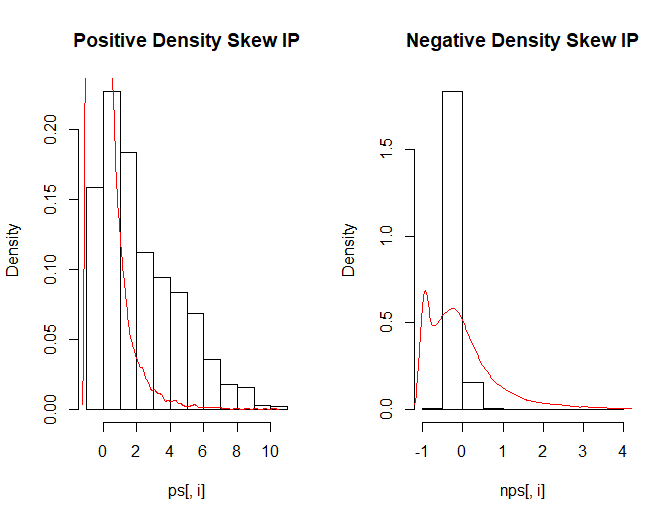
print(time.tests)

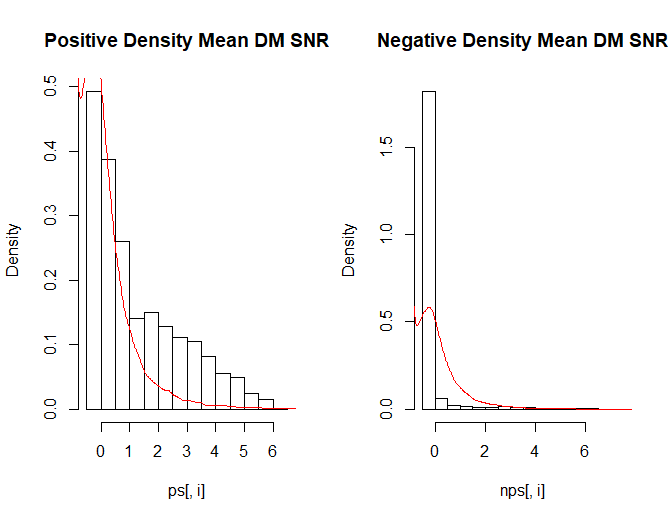
# Appendix: Density Graphs of Covariates

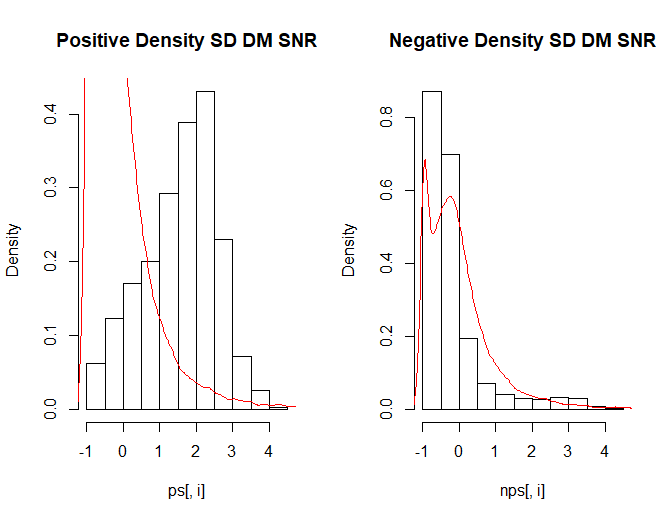


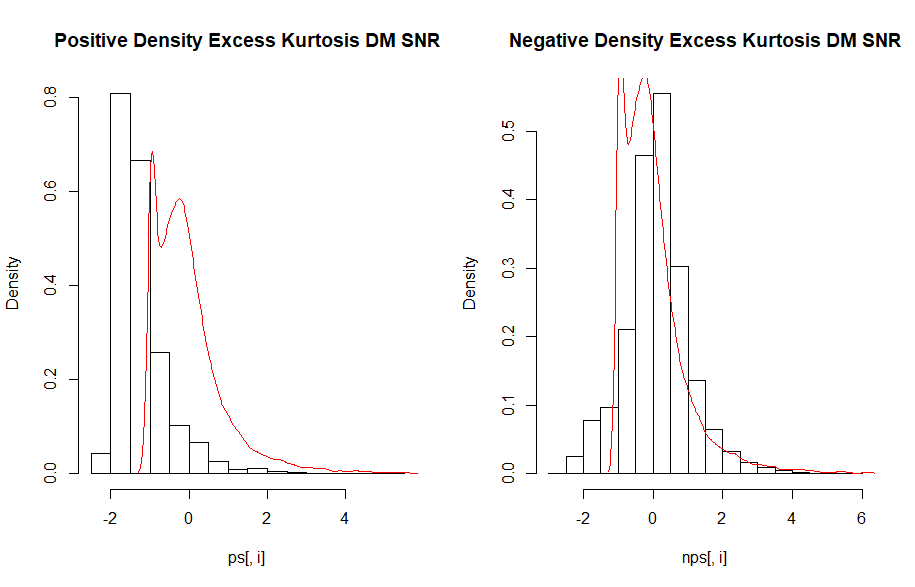


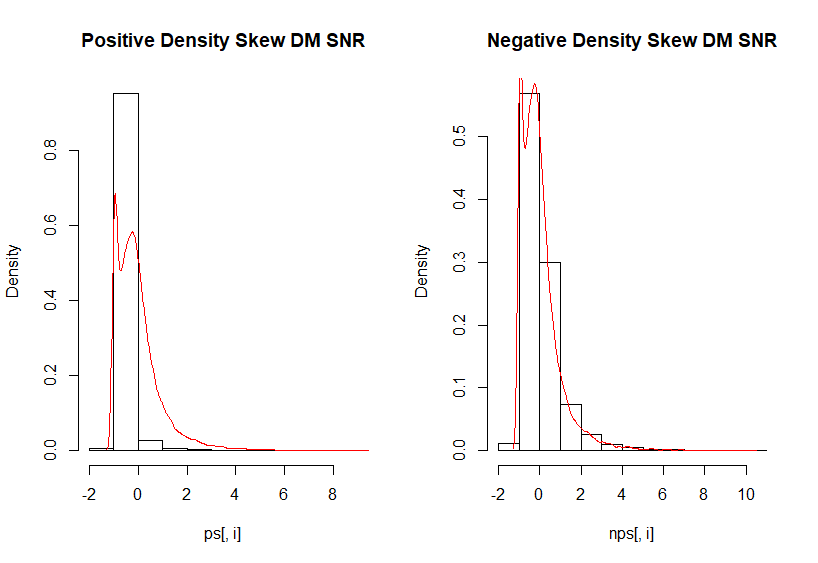












# References

Lyon, R. J. "'Why Are Pulsars Hard To Find?'." *PhD Thesis*. University of Manchester, 2016.

R. J. Lyon, B. W. Stappers, S. Cooper, J. M. Brooke, J. D. Knowles. "Fifty Years of Pulsar Candidate Selection: From simple filters to a new principled real-time classification approach." *Monthly Notices of the Royal Astronomical Society* 459 (1) (2010): 1104-1123.