1 The doughnut experiment

We simulate data in a 2D doughnut shape. There are 2 quantitive features (x_1, x_2) and a binary outcome. A 2D doughnut has two circles. Observations with (x_1, x_2) falling between the circles have a high probability to be in class 1, and those falling inside the inner circle or outside the outer circle have a high probability to be in class 0. We also generate additional noise features $x_3, \cdots, x_{10}$. We compare the performance of various prediction models on this data.

You could similarly generate a high-dimensional doughnut data, although it would be difficult to visualize.

```
library(class)          ## knn()
library(splines)        ## ns()
library(gam)            ## gam() allowing multiple smoothing splines
library(rpart)          ## rpart()
library(randomForest)   ## randomForest()
library(xgboost)        ## xgboost()
library(e1071)          ## svm()
library(rgl)            ## for 3D plots
```

1.1 Generate data

Set seed and sample size. Define the 2D doughnut region by specifying the inner and outer radii. Assign a high probability to the doughnut region and a low probability outside this region. You may treak these numbers to see how much the performance would change for various methods. To generate the outcome deterministically, set $p_{in} = 1$ and $p_{out} = 0$. To generate the outcome probabilistically, set these probabilities to be between 0 and 1.

```
N = 1000; Ntrain = 800
radius1 = 1; radius2 = 1.5  ## radii that define our doughnut
pin = 0.9; pout = 0.1       ## probabilities of class 1 for inside and outside the doughnut
#N = 5000; Ntrain = 4000
#pin = 1; pout = 0
```

First, generate the features. Then, generate the outcome according to the first two features. Check whether the data are well distributed in their location (inside, on, and outside the doughnut).

```
set.seed(2019)
for(i in 1:10)
  assign(paste("x", i, sep=''), rnorm(N))

r = sqrt(x1^2 + x2^2)                        ## radius based on the first 2 feature
table(cut(r, c(0, radius1, radius2, Inf)))    ## distribution inside/on/outside the doughnut

p = ifelse(r>radius1 & r<radius2, pin, pout)  ## each obs has a probability Pr(in class 1)
y = ifelse(runif(N)<p, 1, 0)                 ## generate y according to the probabilities
jy = jitter(y)                               ## jittering y for some plots

table(y, cut(r, c(0, radius1, radius2, Inf))) ## outcome distribution
```

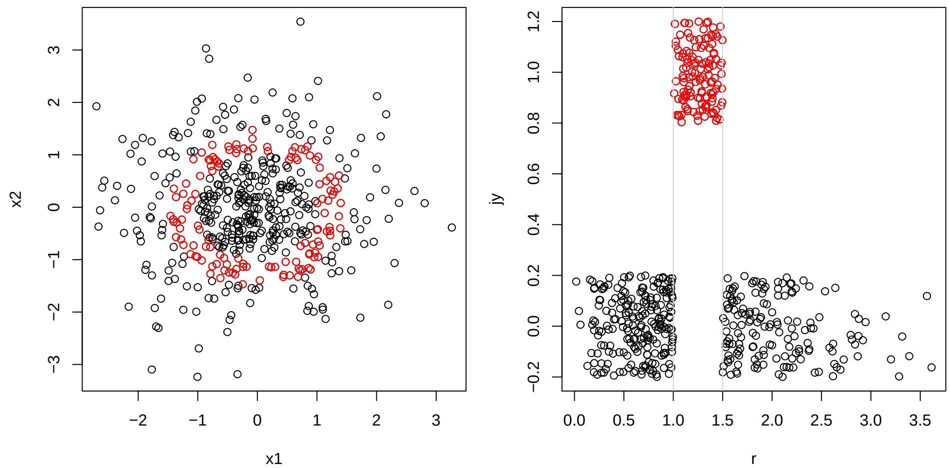
We put all data into a data frame, and then split the data into training and test sets.

```
train = sample(1:N, Ntrain)
mydata = data.frame(x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, y)
mydata.train = mydata[train,]
mydata.test =  mydata[-train,]
```

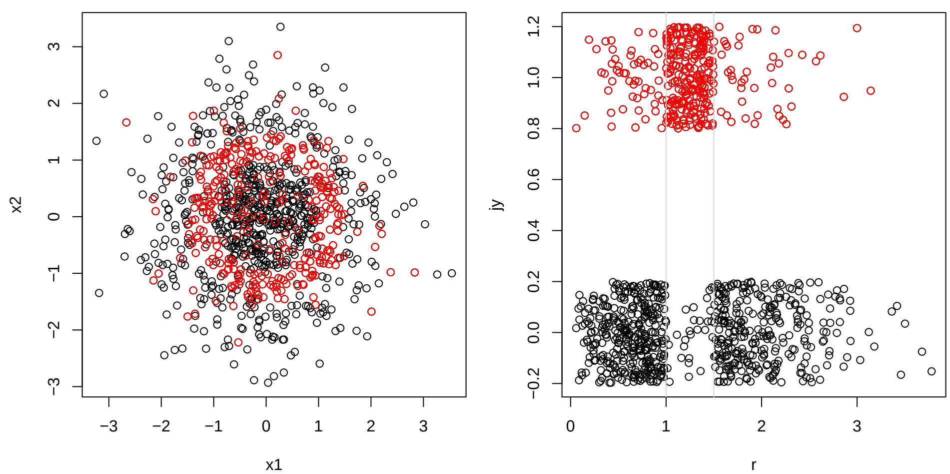
Plot the data.

```
plot(x1, x2); points(x1[y==1], x2[y==1], col=2)
plot(r, jy); points(r[y==1], jy[y==1], col=2); abline(v=c(radius1, radius2), col='lightgrey')
```

This is what I got with $p_{in} = 1$, $p_{out} = 0$, and $N = 500$:



This is what I got with $p_{in} = 0.9$, $p_{out} = 0.1$, and $N = 1000$:



For every prediction model, we will make a plot on the test set, and display the model on a grid. We create some relevant functions here.

```
getpred = function(mod, dataset) {
  if (class(mod)[1] %in% c('glm','Gam'))
    ytestpred = predict(mod, dataset, type="response")
  if (class(mod)[1] %in% c('rpart','randomForest.formula'))
    ytestpred = predict(mod, dataset, type="class")
  if (class(mod)[1] %in% c('svm.formula', 'xgb.Booster'))
    ytestpred = predict(mod, dataset)
  ytestpred
}

testseteval = function(mod) {
  ytestpred = getpred(mod, mydata.test)
  table(y[-train], ytestpred)
}

testsetplot = function(mod) {
  ytestpred = getpred(mod, mydata.test)
  testsetplot2(ytestpred)
```

```
}

testsetplot2 = function(ytestpred) {
  plot(ytestpred, jy[-train], xlab='predicted y', ylab='observed y (jittered)',
       main=paste("cor =", round(cor(as.numeric(ytestpred), y[-train]), 3)))
}

gridlen = 201
gridx1 = seq(-3, 3, length.out=gridlen)
gridx2 = gridx1
g1 = rep(gridx1, each=gridlen)
g2 = rep(gridx2, gridlen)

## grid to evaluate models on
testgrid = data.frame(x1=g1, x2=g2,
                      x3=rnorm(gridlen^2), x4=rnorm(gridlen^2), x5=rnorm(gridlen^2),
                      x6=rnorm(gridlen^2), x7=rnorm(gridlen^2), x8=rnorm(gridlen^2),
                      x9=rnorm(gridlen^2), x10=rnorm(gridlen^2), r=sqrt(g1^2+g2^2))

## 2D and 3D plots on a grid
gridplots = function(mod) {
  gridpred = matrix(getpred(mod, testgrid), gridlen)
  gridplots2(gridpred)
}

gridplots2 = function(gridpred) {
  plot(x1, x2);  points(x1[y==1], x2[y==1], pch=19)
  contour(gridx1, gridx2, gridpred, add=T, col=1:5, lwd=3, levels=c(0.1,0.3,0.5,0.7,0.9))
  rgl::plot3d(g1, g2, gridpred)
}
```

1.2 KNN

KNN using only x1 and x2. You may treak the parameter `k`.

```
table(knn(mydata.train[, 1:2], mydata.test[, 1:2], y[train], k=10), y[-train])

knn1.pred = matrix(knn(mydata.train[, 1:2], testgrid[, 1:2], y[train], k=10), gridlen)
gridplots2(knn1.pred)
```

KNN using all features.

```
table(knn(mydata.train[, 1:10], mydata.test[, 1:10], y[train], k=10), y[-train])

knn2.pred = matrix(knn(mydata.train[, 1:10], testgrid[, 1:10], y[train], k=10), gridlen)
gridplots2(knn2.pred)
```

1.3 Logistic regression

Only linear terms in x1 and x2.

```
mod1 = glm(y ~ x1 + x2, mydata, subset=train, family=binomial)
testsetplot(mod1)
gridplots(mod1)
```

Now add quadratic terms in x1 and x2.

```
mod2 = glm(y ~ x1 + x2 + I(x1^2) + I(x2^2), mydata, subset=train, family=binomial)
testsetplot(mod2)
gridplots(mod2)
```

Logistic regression or GAM with splines on x1 and x2. You may treak the df parameters.

```
#mod3 = glm(y ~ ns(x1, df=5) + ns(x2, df=5), mydata, subset=train, family=binomial)
mod3 = gam(y ~ s(x1, df=5) + s(x2, df=5), mydata, subset=train, family=binomial)
testsetplot(mod3)
gridplots(mod3)
```

Using splines on the radius based on the first two features, $r = \sqrt{x_1^2 + x_2^2}$. You may treak the df parameter.

```
mod4 = glm(y[train] ~ ns(r, df=4), data=data.frame(y,r)[train,], family=binomial)
testsetplot2(predict(mod4, data.frame(r=r[-train])))
gridplots(mod4)
```

```
mod5 = glm(y ~ ., mydata, subset=train, family=binomial)
testsetplot(mod5)
gridplots(mod5)
```

1.4 Classification trees

```
tree1 = rpart(factor(y) ~ x1+x2, data=mydata, subset=train, method='class')
testseteval(tree1)
gridplots(tree1)
```

```
tree1b = rpart(factor(y) ~ x1+x2, data=mydata, subset=train, method='class',
               control=rpart.control(cp=0.001))
cpminxerror = function(cptable) cptable[which.min(cptable[,"xerror"]), "CP"]
tree1bprune = prune(tree1b, cp = cpminxerror(tree1b$cptable))
testseteval(tree1b)
gridplots(tree1b)
```

```
tree2 = rpart(factor(y) ~ ., data=mydata, subset=train, method='class')
testseteval(tree2)
gridplots(tree2)
```

```
tree2b = rpart(factor(y) ~ ., data=mydata, subset=train, method='class',
               control=rpart.control(cp=0.001))
tree2bprune = prune(tree2b, cp = cpminxerror(tree2b$cptable))
testseteval(tree2b)
gridplots(tree2b)
```

1.5 Random forest

RF seems to be worse than a classification tree for a small N but better for a large N.

```
rf1 = randomForest(factor(y) ~ x1+x2, data=mydata, subset=train,
                   ntree=1000, importance=TRUE)
testseteval(rf1)
gridplots(rf1)
```

```
rf2 = randomForest(factor(y) ~ ., data=mydata, subset=train,
                   ntree=1000, importance=TRUE)
```

```
testseteval(rf2)
gridplots(rf2)
```

1.6 XGBoost

```
bst1 = xgboost(data=as.matrix(mydata.train[, 1:2]), label=y[train],
               eta=.3, nround=200,
               objective="binary:logistic")

testsetplot2(predict(bst1, as.matrix(mydata.test[,1:2])))

bst1.pred = matrix(predict(bst1, as.matrix(testgrid[, 1:2])), gridlen)
gridplots2(bst1.pred)
bst1.pred2 = bst1.pred > 0.5
gridplots2(bst1.pred2)
```

```
bst2 = xgboost(data=as.matrix(mydata.train[, 1:10]), label=y[train],
               eta=.3, nround=200,
               objective="binary:logistic")

testsetplot2(predict(bst2, as.matrix(mydata.test[,1:10])))

bst2.pred = matrix(predict(bst2, as.matrix(testgrid[,1:10])), gridlen)
gridplots2(bst2.pred)
bst2.pred2 = bst2.pred > 0.5
gridplots2(bst2.pred2)
```

1.7 SVM

Only x1 and x2 are considered. You may play with the cost parameter.

```
svm1 = svm(factor(y) ~ x1 + x2, data=mydata, subset=train,
           kernel="polynomial", cost=1)
testseteval(svm1)
gridplots(svm1)
```

This model is great when the sample size is large.

```
svm1b = svm(factor(y) ~ x1 + x2, data=mydata, subset=train,
           kernel="radial", cost=1)
testseteval(svm1b)
gridplots(svm1b)
```

```
svm2 = svm(factor(y) ~ ., data=mydata, subset=train,
           kernel="polynomial", cost=1)
testseteval(svm2)
gridplots(svm2)
```

```
svm2b  = svm(factor(y) ~ ., data=mydata, subset=train,
           kernel="radial", cost=1)
testseteval(svm2b)
gridplots(svm2b)
```

Now add quadratic terms.

```r
svm3 = svm(factor(y) ~ x1 + x2 + I(x1^2) +I(x2^2),
           data=mydata, subset=train,
           kernel="polynomial", cost=1, scale=FALSE)
testseteval(svm3)
gridplots(svm3)
```

```r
svm3b = svm(factor(y) ~ x1 + x2 + I(x1^2) +I(x2^2),
           data=mydata, subset=train,
           kernel="radial", cost=1, scale=FALSE)
testseteval(svm3b)
gridplots(svm3b)
```

1.8 ANN?