Harvardx - Data Science Capstone - Sonar Project

Willian Martins Dalboni

2022-09-19

1) INTRODUCTION

1.1) Objective

A naval mine is a self-contained explosive device placed in water to hinder, damage, or utterly destroy naval ships. A Sonar (Sound Navigation and Ranging) emits sound waves to locate and avoid underwater hazards to navigation, but an experienced sonar operator is necessary to tune the equipment and analyze the submarine structures like rock or debris of similar size and shape. The presented algorithm raises the chance of success of this operator at his job, giving him a data-driven perspective and enhancing the likelihood of a better outcome for the assignment and survivability of the crew.

1.2) Metrics, Terminology, and Usability.

The premise is that the maneuverability and time cost of a stretched trip under enemy waters surpasses the impact of losing the ship or the crew. As the main reason for the algorithm is to enhance the crew's survivability, the metric to pursue is Sensitivity to Mines. Sensitivity refers to the ability of the model to identify a mine correctly. It may seem evident at first glance, but every choice comes with a commitment relationship: It is reasonable for the model wrongly identify some debris as a mine if this ensures more mines are correctly identified. Taking this into consideration, the purpose of the use is when advised by the sonar operator since employing it while cruising through peaceful waters can drag the trip without gain.

For the sake of simplicity, debris on these documents will be called Rocks and identified by R, while Mines will be identified as M.

As with many sensible datasets, the features already passed the security by obscurity procedure, with features with no understandable names and already standardized between 0 and 1 to prevent information leaks. This lack of meaning can create an additional challenge.

2) ANALYSIS

2.1) PREPARING THE DATA

2.1.1) Installing the packages

We will need tools to achieve our goal. They need to be installed and loaded to be called when we need them. We verify if we already have them and install them if they are not present.

```r
#Installing packages
if(!require(dslabs)) install.packages("dslabs")
if(!require(tidyverse)) install.packages("tidyverse")
if(!require(caret)) install.packages("caret")
if(!require(purrr)) install.packages("purrr")
if(!require(MASS)) install.packages("MASS")
if(!require(gam)) install.packages("gam")
if(!require(randomForest)) install.packages("randomForest")
if(!require(dplyr)) install.packages("dplyr")
if(!require(ggthemes)) install.packages
if(!require(knitr)) install.packages("knitr")
if(!require(kableExtra)) install.packages("kableExtra")
if(!require(summarytools)) install.packages("summarytools")
if(!require(ggstatsplot)) install.packages("ggstatsplot")
if(!require(PMCMRplus)) install.packages("PMCMRplus")


#Loading libraries
library(dslabs)
library(tidyverse)
library(caret)
library(purrr)
library(matrixStats)
library(MASS)
library(gam)
library(randomForest)
library(dplyr)
library(ggthemes)
library(knitr)
library(kableExtra)
library(summarytools)
library(ggstatsplot)
library(PMCMRplus)
```

2.1.2) Downloading the dataset

This code downloads the dataset from GitHub. After that, it separates by commas and renames each column to reflect the features(x00) and the outcome(y). An annually updated version of the dataset can be seen in the Reference section of this document.

```r
#Creates a tempfile
dl <- tempfile()

#Download from Github to tempfile [1]
download.file("https://raw.githubusercontent.com/wmdalboni/HarvardX.Capstone.Self/main/sonardataset.csv

#Separate the file by comma, ignoring headers, and save into the dataset variable
dataset <- read.csv(dl, header=FALSE,sep=",")

#view(dataset) #Check!

#Only three digits to an easier visualization.
options(digits=3)
```

```r
# Let us keep the original dataset intact. We may need it and do not want to download it again.
nds <- dataset

#view(nds) #Check!

#Forcing data.frame format.
nds <- as.data.frame(nds)

#Sweep the dataset, changing column names.
for(i in 1:ncol(nds)){
#Rename the dependent variable column to y.
if(i == ncol(nds)){
        colnames(nds)[i] <- "y"
#The y needs to be a factor.
        nds[,i] <-  nds[,i] %>% as.factor()
       } else {
#Rename the independent variables column to x01, x02, x03...
        colnames(nds)[i] <- paste("x",str_pad(i,2,pad ="0"),sep="")
#The x needs to be numeric.
        nds[,i] <-  nds[,i] %>% as.numeric()
  }
}
#glimpse(nds) #Check!
```

2.2) EXPLORATION & INSIGHTS

Observing the datasets is essential to get insights into what we got and what we can create to predict better.

2.2.1) Heads

The "head" command shows us the six rows from the top of each set to find if they have the same names, column order, and data types. We have 60 different features as can be seen.

```r
#Transpose to better visualization
head(nds)[1:10]
```

```
##      x01    x02    x03    x04    x05    x06   x07   x08    x09   x10
## 1 0.0200 0.0371 0.0428 0.0207 0.0954 0.0986 0.154 0.160 0.3109 0.211
## 2 0.0453 0.0523 0.0843 0.0689 0.1183 0.2583 0.216 0.348 0.3337 0.287
## 3 0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.243 0.377 0.5598 0.619
## 4 0.0100 0.0171 0.0623 0.0205 0.0205 0.0368 0.110 0.128 0.0598 0.126
## 5 0.0762 0.0666 0.0481 0.0394 0.0590 0.0649 0.121 0.247 0.3564 0.446
## 6 0.0286 0.0453 0.0277 0.0174 0.0384 0.0990 0.120 0.183 0.2105 0.304
```

```r
head(nds)[11:20]
```

```
##      x11   x12    x13    x14   x15   x16    x17   x18   x19   x20
## 1 0.1609 0.158 0.2238 0.0645 0.066 0.227 0.3100 0.300 0.508 0.480
## 2 0.4918 0.655 0.6919 0.7797 0.746 0.944 1.0000 0.887 0.802 0.782
## 3 0.6333 0.706 0.5544 0.5320 0.648 0.693 0.6759 0.755 0.893 0.862
```

```
## 4 0.0881 0.199 0.0184 0.2261 0.173 0.213 0.0693 0.228 0.406 0.397
## 5 0.4152 0.395 0.4256 0.4135 0.453 0.533 0.7306 0.619 0.203 0.464
## 6 0.2988 0.425 0.6343 0.8198 1.000 0.999 0.9508 0.902 0.723 0.512
```

```
head(nds)[21:30]
```

```
##     x21   x22   x23   x24   x25   x26   x27   x28   x29   x30
## 1 0.578 0.507 0.433 0.555 0.671 0.641 0.710 0.808 0.679 0.386
## 2 0.521 0.405 0.396 0.391 0.325 0.320 0.327 0.277 0.442 0.203
## 3 0.797 0.674 0.429 0.365 0.533 0.241 0.507 0.853 0.604 0.851
## 4 0.274 0.369 0.556 0.485 0.314 0.533 0.526 0.252 0.209 0.356
## 5 0.415 0.429 0.573 0.540 0.316 0.229 0.700 1.000 0.726 0.472
## 6 0.207 0.399 0.589 0.287 0.204 0.578 0.539 0.375 0.341 0.507
```

```
head(nds)[31:40]
```

```
##     x31   x32   x33    x34   x35   x36   x37   x38   x39   x40
## 1 0.131 0.260 0.512 0.7547 0.854 0.851 0.669 0.610 0.494 0.274
## 2 0.379 0.295 0.198 0.2341 0.131 0.418 0.384 0.106 0.184 0.197
## 3 0.851 0.504 0.186 0.2709 0.423 0.304 0.612 0.676 0.537 0.472
## 4 0.626 0.734 0.612 0.3497 0.395 0.301 0.541 0.881 0.986 0.917
## 5 0.510 0.546 0.288 0.0981 0.195 0.418 0.460 0.322 0.283 0.243
## 6 0.558 0.478 0.330 0.2198 0.141 0.286 0.381 0.416 0.405 0.330
```

```
head(nds)[41:50]
```

```
##     x41    x42    x43    x44    x45    x46    x47    x48    x49    x50
## 1 0.051 0.2834 0.2825 0.4256 0.2641 0.1386 0.1051 0.1343 0.0383 0.0324
## 2 0.167 0.0583 0.1401 0.1628 0.0621 0.0203 0.0530 0.0742 0.0409 0.0061
## 3 0.465 0.2587 0.2129 0.2222 0.2111 0.0176 0.1348 0.0744 0.0130 0.0106
## 4 0.612 0.5006 0.3210 0.3202 0.4295 0.3654 0.2655 0.1576 0.0681 0.0294
## 5 0.198 0.2444 0.1847 0.0841 0.0692 0.0528 0.0357 0.0085 0.0230 0.0046
## 6 0.271 0.2650 0.0723 0.1238 0.1192 0.1089 0.0623 0.0494 0.0264 0.0081
```

```
head(nds)[51:60]
```

```
##      x51    x52    x53    x54    x55    x56    x57    x58    x59    x60
## 1 0.0232 0.0027 0.0065 0.0159 0.0072 0.0167 0.0180 0.0084 0.0090 0.0032
## 2 0.0125 0.0084 0.0089 0.0048 0.0094 0.0191 0.0140 0.0049 0.0052 0.0044
## 3 0.0033 0.0232 0.0166 0.0095 0.0180 0.0244 0.0316 0.0164 0.0095 0.0078
## 4 0.0241 0.0121 0.0036 0.0150 0.0085 0.0073 0.0050 0.0044 0.0040 0.0117
## 5 0.0156 0.0031 0.0054 0.0105 0.0110 0.0015 0.0072 0.0048 0.0107 0.0094
## 6 0.0104 0.0045 0.0014 0.0038 0.0013 0.0089 0.0057 0.0027 0.0051 0.0062
```

```
head(nds)[61]
```

```
##   y
## 1 R
## 2 R
## 3 R
## 4 R
## 5 R
## 6 R
```

2.2.2) Glimpse

Glimpse helps us pay attention to the columns' dimensions and variable types. Even though the dataset is relatively small on rows, it has a fair amount of features and probably will be enough for a good prediction.

```
glimpse(nds)
```

```
## Rows: 208
## Columns: 61
## $ x01 <dbl> 0.0200, 0.0453, 0.0262, 0.0100, 0.0762, 0.0286, 0.0317, 0.0519, 0.~
## $ x02 <dbl> 0.0371, 0.0523, 0.0582, 0.0171, 0.0666, 0.0453, 0.0956, 0.0548, 0.~
## $ x03 <dbl> 0.0428, 0.0843, 0.1099, 0.0623, 0.0481, 0.0277, 0.1321, 0.0842, 0.~
## $ x04 <dbl> 0.0207, 0.0689, 0.1083, 0.0205, 0.0394, 0.0174, 0.1408, 0.0319, 0.~
## $ x05 <dbl> 0.0954, 0.1183, 0.0974, 0.0205, 0.0590, 0.0384, 0.1674, 0.1158, 0.~
## $ x06 <dbl> 0.0986, 0.2583, 0.2280, 0.0368, 0.0649, 0.0990, 0.1710, 0.0922, 0.~
## $ x07 <dbl> 0.1539, 0.2156, 0.2431, 0.1098, 0.1209, 0.1201, 0.0731, 0.1027, 0.~
## $ x08 <dbl> 0.1601, 0.3481, 0.3771, 0.1276, 0.2467, 0.1833, 0.1401, 0.0613, 0.~
## $ x09 <dbl> 0.3109, 0.3337, 0.5598, 0.0598, 0.3564, 0.2105, 0.2083, 0.1465, 0.~
## $ x10 <dbl> 0.2111, 0.2872, 0.6194, 0.1264, 0.4459, 0.3039, 0.3513, 0.2838, 0.~
## $ x11 <dbl> 0.1609, 0.4918, 0.6333, 0.0881, 0.4152, 0.2988, 0.1786, 0.2802, 0.~
## $ x12 <dbl> 0.1582, 0.6552, 0.7060, 0.1992, 0.3952, 0.4250, 0.0658, 0.3086, 0.~
## $ x13 <dbl> 0.2238, 0.6919, 0.5544, 0.0184, 0.4256, 0.6343, 0.0513, 0.2657, 0.~
## $ x14 <dbl> 0.0645, 0.7797, 0.5320, 0.2261, 0.4135, 0.8198, 0.3752, 0.3801, 0.~
## $ x15 <dbl> 0.0660, 0.7464, 0.6479, 0.1729, 0.4528, 1.0000, 0.5419, 0.5626, 0.~
## $ x16 <dbl> 0.2273, 0.9444, 0.6931, 0.2131, 0.5326, 0.9988, 0.5440, 0.4376, 0.~
## $ x17 <dbl> 0.3100, 1.0000, 0.6759, 0.0693, 0.7306, 0.9508, 0.5150, 0.2617, 0.~
## $ x18 <dbl> 0.300, 0.887, 0.755, 0.228, 0.619, 0.902, 0.426, 0.120, 0.380, 0.3~
## $ x19 <dbl> 0.508, 0.802, 0.893, 0.406, 0.203, 0.723, 0.202, 0.668, 0.740, 0.3~
## $ x20 <dbl> 0.4797, 0.7818, 0.8619, 0.3973, 0.4636, 0.5122, 0.4233, 0.9402, 0.~
## $ x21 <dbl> 0.578, 0.521, 0.797, 0.274, 0.415, 0.207, 0.772, 0.783, 0.980, 0.6~
## $ x22 <dbl> 0.507, 0.405, 0.674, 0.369, 0.429, 0.399, 0.974, 0.535, 0.889, 0.7~
## $ x23 <dbl> 0.433, 0.396, 0.429, 0.556, 0.573, 0.589, 0.939, 0.681, 0.671, 0.9~
## $ x24 <dbl> 0.555, 0.391, 0.365, 0.485, 0.540, 0.287, 0.556, 0.917, 0.429, 0.9~
## $ x25 <dbl> 0.671, 0.325, 0.533, 0.314, 0.316, 0.204, 0.527, 0.761, 0.337, 0.9~
## $ x26 <dbl> 0.641, 0.320, 0.241, 0.533, 0.229, 0.578, 0.683, 0.822, 0.737, 0.7~
## $ x27 <dbl> 0.7104, 0.3271, 0.5070, 0.5256, 0.6995, 0.5389, 0.5713, 0.8872, 0.~
## $ x28 <dbl> 0.8080, 0.2767, 0.8533, 0.2520, 1.0000, 0.3750, 0.5429, 0.6091, 0.~
## $ x29 <dbl> 0.6791, 0.4423, 0.6036, 0.2090, 0.7262, 0.3411, 0.2177, 0.2967, 0.~
## $ x30 <dbl> 0.3857, 0.2028, 0.8514, 0.3559, 0.4724, 0.5067, 0.2149, 0.1103, 0.~
## $ x31 <dbl> 0.131, 0.379, 0.851, 0.626, 0.510, 0.558, 0.581, 0.132, 0.301, 0.5~
## $ x32 <dbl> 0.2604, 0.2947, 0.5045, 0.7340, 0.5459, 0.4778, 0.6323, 0.0624, 0.~
## $ x33 <dbl> 0.512, 0.198, 0.186, 0.612, 0.288, 0.330, 0.296, 0.099, 0.317, 0.3~
## $ x34 <dbl> 0.7547, 0.2341, 0.2709, 0.3497, 0.0981, 0.2198, 0.1873, 0.4006, 0.~
## $ x35 <dbl> 0.8537, 0.1306, 0.4232, 0.3953, 0.1951, 0.1407, 0.2969, 0.3666, 0.~
## $ x36 <dbl> 0.851, 0.418, 0.304, 0.301, 0.418, 0.286, 0.516, 0.105, 0.219, 0.1~
## $ x37 <dbl> 0.669, 0.384, 0.612, 0.541, 0.460, 0.381, 0.615, 0.192, 0.246, 0.1~
## $ x38 <dbl> 0.6097, 0.1057, 0.6756, 0.8814, 0.3217, 0.4158, 0.4283, 0.3930, 0.~
## $ x39 <dbl> 0.4943, 0.1840, 0.5375, 0.9857, 0.2828, 0.4054, 0.5479, 0.4288, 0.~
## $ x40 <dbl> 0.2744, 0.1970, 0.4719, 0.9167, 0.2430, 0.3296, 0.6133, 0.2546, 0.~
## $ x41 <dbl> 0.0510, 0.1674, 0.4647, 0.6121, 0.1979, 0.2707, 0.5017, 0.1151, 0.~
## $ x42 <dbl> 0.2834, 0.0583, 0.2587, 0.5006, 0.2444, 0.2650, 0.2377, 0.2196, 0.~
## $ x43 <dbl> 0.2825, 0.1401, 0.2129, 0.3210, 0.1847, 0.0723, 0.1957, 0.1879, 0.~
## $ x44 <dbl> 0.4256, 0.1628, 0.2222, 0.3202, 0.0841, 0.1238, 0.1749, 0.1437, 0.~
## $ x45 <dbl> 0.2641, 0.0621, 0.2111, 0.4295, 0.0692, 0.1192, 0.1304, 0.2146, 0.~
```

```
## $ x46 <dbl> 0.1386, 0.0203, 0.0176, 0.3654, 0.0528, 0.1089, 0.0597, 0.2360, 0.~
## $ x47 <dbl> 0.1051, 0.0530, 0.1348, 0.2655, 0.0357, 0.0623, 0.1124, 0.1125, 0.~
## $ x48 <dbl> 0.1343, 0.0742, 0.0744, 0.1576, 0.0085, 0.0494, 0.1047, 0.0254, 0.~
## $ x49 <dbl> 0.0383, 0.0409, 0.0130, 0.0681, 0.0230, 0.0264, 0.0507, 0.0285, 0.~
## $ x50 <dbl> 0.0324, 0.0061, 0.0106, 0.0294, 0.0046, 0.0081, 0.0159, 0.0178, 0.~
## $ x51 <dbl> 0.0232, 0.0125, 0.0033, 0.0241, 0.0156, 0.0104, 0.0195, 0.0052, 0.~
## $ x52 <dbl> 0.0027, 0.0084, 0.0232, 0.0121, 0.0031, 0.0045, 0.0201, 0.0081, 0.~
## $ x53 <dbl> 0.0065, 0.0089, 0.0166, 0.0036, 0.0054, 0.0014, 0.0248, 0.0120, 0.~
## $ x54 <dbl> 0.0159, 0.0048, 0.0095, 0.0150, 0.0105, 0.0038, 0.0131, 0.0045, 0.~
## $ x55 <dbl> 0.0072, 0.0094, 0.0180, 0.0085, 0.0110, 0.0013, 0.0070, 0.0121, 0.~
## $ x56 <dbl> 0.0167, 0.0191, 0.0244, 0.0073, 0.0015, 0.0089, 0.0138, 0.0097, 0.~
## $ x57 <dbl> 0.0180, 0.0140, 0.0316, 0.0050, 0.0072, 0.0057, 0.0092, 0.0085, 0.~
## $ x58 <dbl> 0.0084, 0.0049, 0.0164, 0.0044, 0.0048, 0.0027, 0.0143, 0.0047, 0.~
## $ x59 <dbl> 0.0090, 0.0052, 0.0095, 0.0040, 0.0107, 0.0051, 0.0036, 0.0048, 0.~
## $ x60 <dbl> 0.0032, 0.0044, 0.0078, 0.0117, 0.0094, 0.0062, 0.0103, 0.0053, 0.~
## $ y   <fct> R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, ~
```

2.2.3) Summary

The summary shows the Quartiles, the Minimum, and Maximum values. It is crucial since it can show outliers not apparent using other approaches. This time it becomes evident that the data was previously normalized between 0 and 1. As we can observe, there is no missing data between the features, which will contribute to faster preprocessing.

```r
#Save the transposed summary
sum_nds <- t(summarytools::descr(nds))
t(sum_nds[,1:7]) #Only the columns needed
```

```
##            x01    x02    x03    x04    x05    x06    x07    x08    x09    x10
## Mean    0.0292 0.0384 0.0438 0.0539 0.0752 0.1046 0.1217 0.1348 0.1780 0.2083
## Std.Dev 0.0230 0.0330 0.0384 0.0465 0.0556 0.0591 0.0618 0.0852 0.1184 0.1344
## Min     0.0015 0.0006 0.0015 0.0058 0.0067 0.0102 0.0033 0.0055 0.0075 0.0113
## Q1      0.0133 0.0164 0.0189 0.0244 0.0377 0.0670 0.0806 0.0804 0.0968 0.1111
## Median  0.0228 0.0308 0.0343 0.0440 0.0625 0.0922 0.1069 0.1121 0.1522 0.1824
## Q3      0.0358 0.0481 0.0582 0.0657 0.1011 0.1341 0.1541 0.1698 0.2341 0.2690
## Max     0.1371 0.2339 0.3059 0.4264 0.4010 0.3823 0.3729 0.4590 0.6828 0.7106
##            x11    x12    x13    x14    x15    x16    x17    x18    x19    x20
## Mean    0.2360 0.2502 0.2733 0.2966 0.3202 0.3785 0.4160 0.4523 0.5048 0.5630
## Std.Dev 0.1327 0.1401 0.1410 0.1645 0.2054 0.2326 0.2637 0.2615 0.2580 0.2627
## Min     0.0289 0.0236 0.0184 0.0273 0.0031 0.0162 0.0349 0.0375 0.0494 0.0656
## Q1      0.1282 0.1335 0.1658 0.1744 0.1643 0.1959 0.2055 0.2419 0.2990 0.3505
## Median  0.2248 0.2490 0.2640 0.2811 0.2817 0.3047 0.3084 0.3683 0.4350 0.5425
## Q3      0.3018 0.3316 0.3515 0.3870 0.4530 0.5360 0.6601 0.6791 0.7319 0.8095
## Max     0.7342 0.7060 0.7131 0.9970 1.0000 0.9988 1.0000 1.0000 1.0000 1.0000
##            x21    x22    x23    x24   x25    x26    x27    x28    x29    x30
## Mean    0.6091 0.6243 0.6470 0.6727 0.675 0.6999 0.7022 0.6940 0.6421 0.5809
## Std.Dev 0.2578 0.2559 0.2502 0.2391 0.245 0.2372 0.2457 0.2372 0.2402 0.2207
## Min     0.0512 0.0219 0.0563 0.0239 0.024 0.0921 0.0481 0.0284 0.0144 0.0613
## Q1      0.3975 0.4063 0.4485 0.5405 0.525 0.5435 0.5298 0.5339 0.4613 0.4104
## Median  0.6177 0.6649 0.6997 0.6985 0.721 0.7545 0.7456 0.7319 0.6808 0.6072
## Q3      0.8180 0.8321 0.8522 0.8734 0.875 0.8938 0.9174 0.9019 0.8523 0.7369
## Max     1.0000 1.0000 1.0000 1.0000 1.000 1.0000 1.0000 1.0000 1.0000 1.0000
##            x31    x32    x33    x34   x35    x36    x37    x38    x39    x40
```

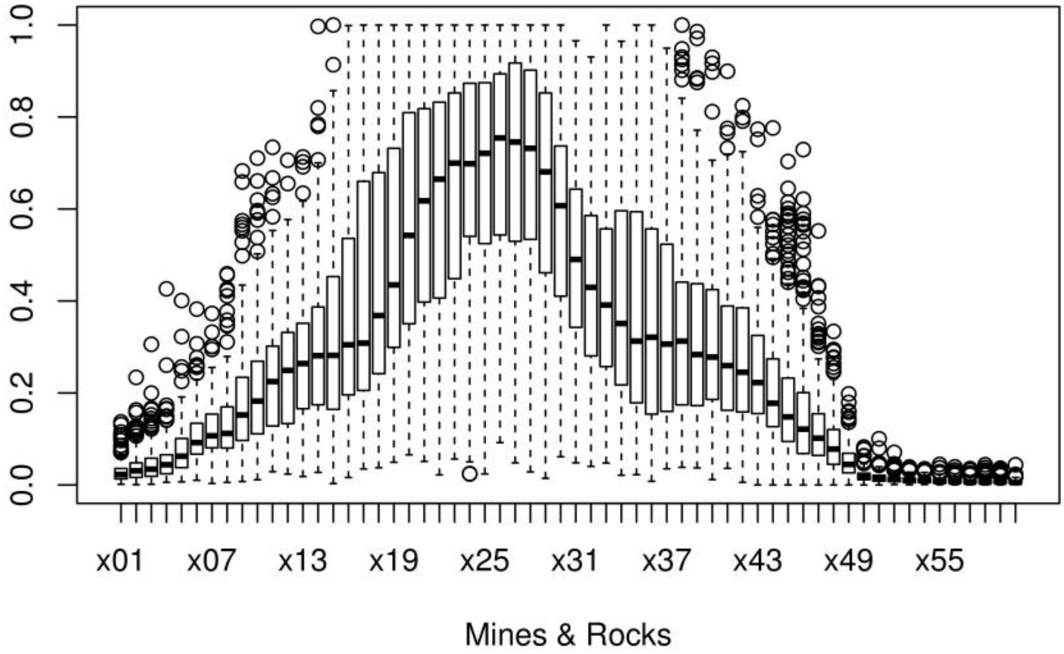
```
## Mean     0.5045 0.4390 0.4172 0.4032 0.3926 0.385 0.3638 0.3397 0.3258 0.3112
## Std.Dev 0.2140 0.2132 0.2065 0.2312 0.2591 0.264 0.2399 0.2130 0.1991 0.1787
## Min      0.0482 0.0404 0.0477 0.0212 0.0223 0.008 0.0351 0.0383 0.0371 0.0117
## Q1       0.3426 0.2806 0.2573 0.2175 0.1785 0.154 0.1600 0.1743 0.1724 0.1859
## Median  0.4903 0.4296 0.3912 0.3510 0.3127 0.321 0.3063 0.3127 0.2835 0.2781
## Q3       0.6432 0.5857 0.5568 0.5961 0.5941 0.557 0.5234 0.4410 0.4375 0.4247
## Max      0.9657 0.9306 1.0000 0.9647 1.0000 1.000 0.9497 1.0000 0.9857 0.9297
##            x41    x42   x43   x44    x45    x46    x47    x48    x49    x50
## Mean     0.289 0.2783 0.247 0.214 0.1972 0.1606 0.1225 0.0914 0.0519 0.0204
## Std.Dev 0.171 0.1687 0.139 0.133 0.1516 0.1339 0.0870 0.0624 0.0360 0.0137
## Min      0.036 0.0056 0.000 0.000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
## Q1       0.162 0.1587 0.155 0.127 0.0945 0.0684 0.0642 0.0450 0.0263 0.0115
## Median  0.260 0.2451 0.223 0.178 0.1480 0.1213 0.1017 0.0781 0.0447 0.0179
## Q3       0.389 0.3851 0.325 0.274 0.2324 0.2006 0.1547 0.1204 0.0689 0.0254
## Max      0.899 0.8246 0.773 0.776 0.7034 0.7292 0.5522 0.3339 0.1981 0.0825
##             x51     x52     x53     x54     x55     x56     x57     x58     x59
## Mean     0.01607 0.01342 0.01071 0.01094 0.00929 0.00822 0.00782 0.00795 0.00794
## Std.Dev 0.01201 0.00963 0.00706 0.00730 0.00709 0.00574 0.00579 0.00647 0.00618
## Min      0.00000 0.00080 0.00050 0.00100 0.00060 0.00040 0.00030 0.00030 0.00010
## Q1       0.00835 0.00725 0.00505 0.00535 0.00410 0.00440 0.00370 0.00360 0.00365
## Median  0.01390 0.01140 0.00955 0.00930 0.00750 0.00685 0.00595 0.00580 0.00640
## Q3       0.02085 0.01675 0.01490 0.01450 0.01210 0.01065 0.01045 0.01040 0.01035
## Max      0.10040 0.07090 0.03900 0.03520 0.04470 0.03940 0.03550 0.04400 0.03640
##             x60
## Mean     0.00651
## Std.Dev 0.00503
## Min      0.00060
## Q1       0.00310
## Median  0.00530
## Q3       0.00855
## Max      0.04390
```

2.3) GRAPHS

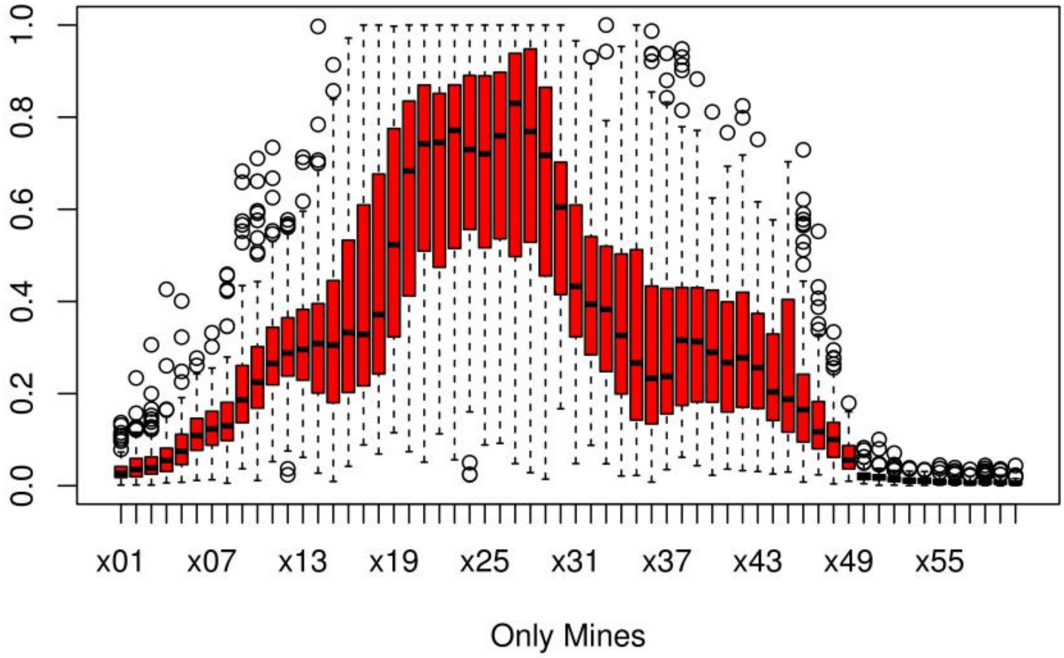
2.3.1) Boxplot: Mines(RED) x Rocks(BLUE)

Sometimes boxplots can help to notice patterns we can explore to create a better prediction or even remove outliers. There is a good amount of variance between each feature, and probably there is no near zero variance feature. While there are features where mines tend to have a higher median, others are lower or very close to rocks.

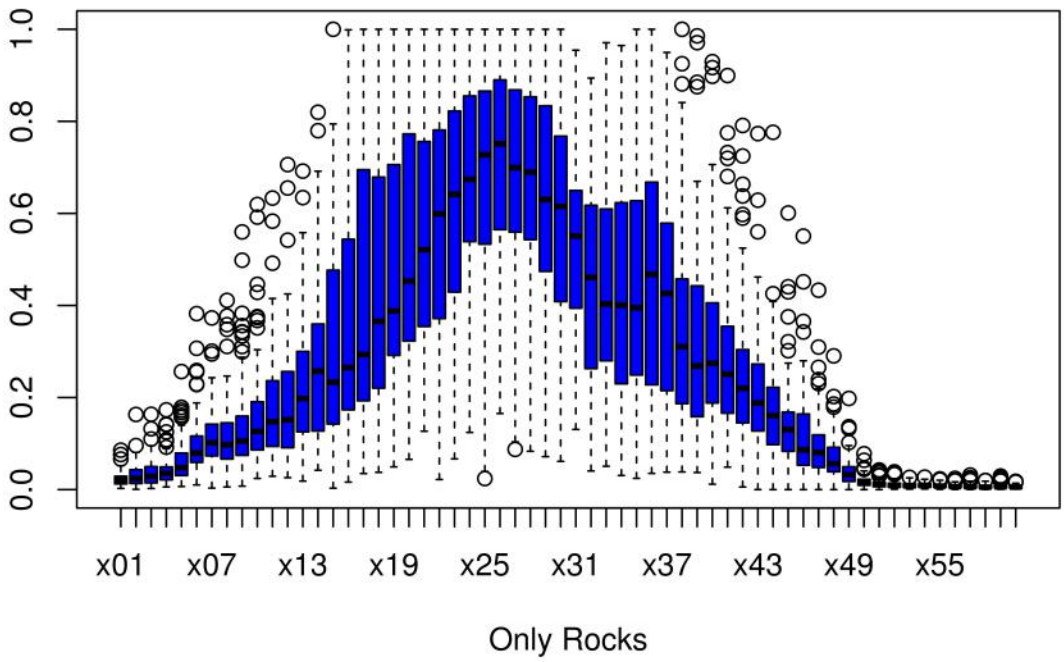
```
#Creates a boxplot with Mine & Rocks

nds %>%
    dplyr::select(-y) %>% #Remove Outcome
    boxplot(col="white", #White Boxes
            xlab="Mines & Rocks") #The x Label
```




```
#Creates a boxplot with Mines only
nds %>%
    filter(y=='M') %>% #Filter by Mines
    dplyr::select(-y) %>% #Remove Outcome
    boxplot(col="red", #Red boxes
            xlab="Only Mines") #The x Label
```




```r
#Creates a boxplot with Rocks only
nds %>%
    filter(y=='R') %>% #Filter by Rocks
    dplyr::select(-y) %>% #Remove Outcome
    boxplot(col="blue",#Blue boxes
            xlab="Only Rocks") #The x Label
```

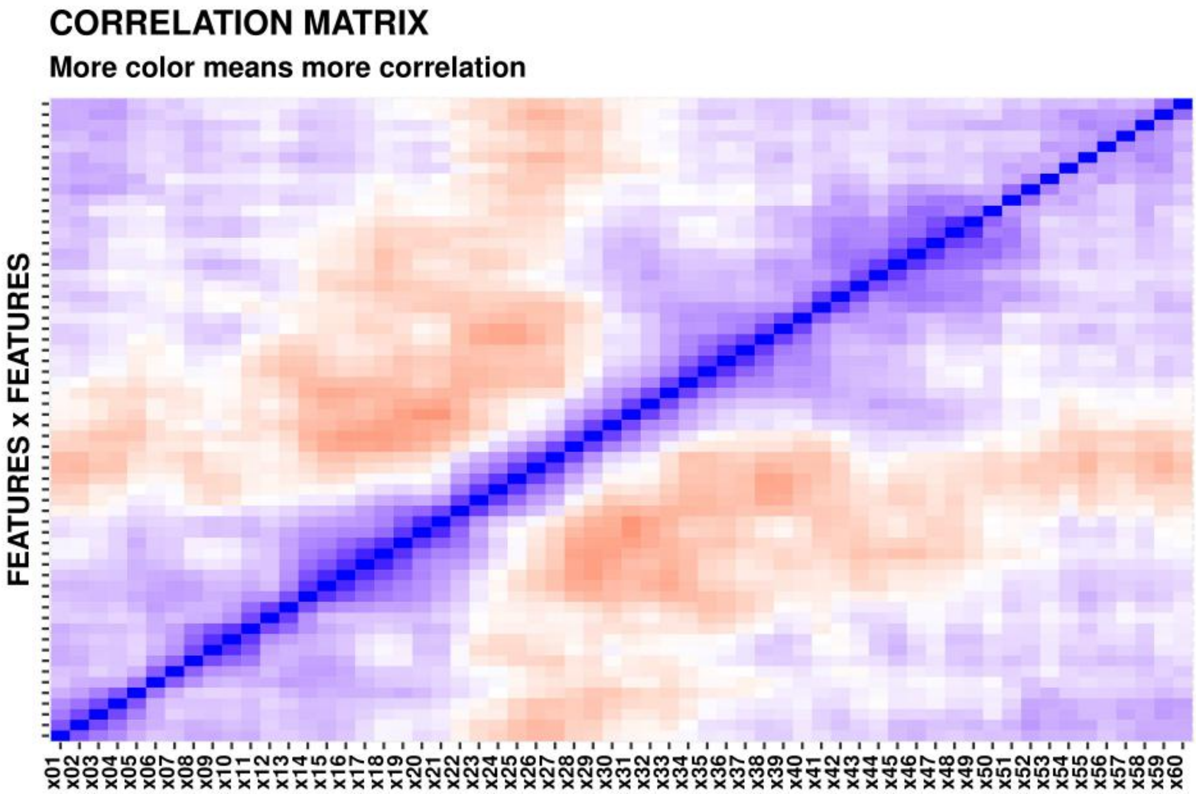


2.3.2) Correlation Matrix: Negative(Blue) x Positive(Red)

More colors mean more correlation. There are features with a strong correlation, which may mean they represent confounded data, or it simply does not add much more information to the final model. We can choose only some of them to use and drop the others to optimize the process.

```
forcormt <- cor(nds[1:60]) %>% #Only the predictors
            as.data.frame() %>% #Save as data frame
            rownames_to_column() %>% #Create a column with the name of the predictors' columns
            pivot_longer(-rowname) %>%# Increase the number of rows and decreasing the number of column
            arrange(desc(value)) #Order highest to lowest to an easier check of the dataset

#view(forcormt) #Check!

forcormt %>%
  ggplot(aes(x=rowname, y=name, fill=value)) +
  geom_tile()+ #Matrix graph
  scale_fill_gradient2(low = "red",#Scores under 0
                       high = "blue", #Scores above 0
                       mid= "white", # No correlation
                       na.value = "grey50", # NA
                       guide = "colourbar", # Color guide
                       aesthetics = "fill") + # Fill the tiles
  labs(
    title = "CORRELATION MATRIX",
```

```
    subtitle = "More color means more correlation",
    x = "", # No label on X to gain more space
    y = "FEATURES x FEATURES",
    size = 8,
    family = "sans") +
  theme( # Details of the theme
    legend.position="none", #  No legend to gain more space
        title= element_text(colour = "black",
                            face="bold",
                            family="sans"),
    axis.text.x = element_text(angle=90, #90 degrees
                               size = 8,
                               face="bold",
                               colour = "black",
                               family="sans",
                               vjust=-0.05), # Adjust x label
    axis.text.y = element_blank()) #No label on y
```



3) PREPROCESSING: DATA DROPPING & FEATURE SELECTION

3.1) Splitting the dataset

There are organizational and functional arguments to divide the dataset at this point. The dataset is separated to make the test set the real-life simulation to be predicted, and having its data leaked into our train set by some calculation could make the model less predictive than it could be. Even though some preprocessing topics under this could be done without separating the dataset, doing it here helps the cadence of reading and organization of headings.

```
#Set seed to be reproducible
set.seed(87, sample.kind = "Rounding")
#Generate dataset index by a percentage
#of the dataset with a similar proportion
#of y on each set
partition_index <- caret::createDataPartition(nds$y,
                                              list=FALSE,
                                              p=0.2) #80x20
#partition_index #Check!

#xy = Features and Predictor
#x = Only features
#y = Only predictors
test_xy <- nds[partition_index,]
train_xy <- nds[-partition_index,]

test_y <- nds[partition_index,] %>% dplyr::select(y)
train_y <- nds[-partition_index,] %>% dplyr::select(y)

test_x <- test_xy %>% dplyr::select(-y)
train_x <- train_xy %>% dplyr::select(-y)

#Check if they have similar proportions
#mean(train_y=="M") #Check!
#mean(test_y=="M") #Check!
```

3.2) Near Zero Variance

It is not uncommon to have columns of independent variables with near zero variance between the values. If not removed, they can negatively impact the result of the final model. We did not observe this possibility by glimpsing the dataset before, and now this code proves it.

```
#zeroVar and nzv TRUE would need to be treated,
#but there is none.
nzv_x <- nearZeroVar(train_x, saveMetrics= TRUE)

nzv_x[1:20,]
```

```
##     freqRatio percentUnique zeroVar   nzv
## x01       1.0          85.5   FALSE FALSE
```

```
## x02        1.0           89.1   FALSE FALSE
## x03        1.0           93.3   FALSE FALSE
## x04        1.0           87.3   FALSE FALSE
## x05        1.0           93.3   FALSE FALSE
## x06        1.0           93.3   FALSE FALSE
## x07        1.0           93.9   FALSE FALSE
## x08        1.0           97.0   FALSE FALSE
## x09        1.0           98.2   FALSE FALSE
## x10        2.0           99.4   FALSE FALSE
## x11        1.0           97.0   FALSE FALSE
## x12        1.0           98.8   FALSE FALSE
## x13        1.5           96.4   FALSE FALSE
## x14        1.0           97.6   FALSE FALSE
## x15        1.0           97.0   FALSE FALSE
## x16        1.0           98.2   FALSE FALSE
## x17        1.5           97.6   FALSE FALSE
## x18        1.5           97.6   FALSE FALSE
## x19        1.0           98.8   FALSE FALSE
## x20        1.0           97.6   FALSE FALSE
```

```
nzv_x[21:40,]
```

```
##     freqRatio percentUnique zeroVar   nzv
## x21       5.0          97.6   FALSE FALSE
## x22       6.0          97.0   FALSE FALSE
## x23       6.0          97.0   FALSE FALSE
## x24       6.0          97.0   FALSE FALSE
## x25       3.0          95.2   FALSE FALSE
## x26       4.0          94.5   FALSE FALSE
## x27       8.0          89.7   FALSE FALSE
## x28       4.5          94.5   FALSE FALSE
## x29       2.5          96.4   FALSE FALSE
## x30       3.0          96.4   FALSE FALSE
## x31       1.0         100.0   FALSE FALSE
## x32       1.0          98.8   FALSE FALSE
## x33       1.0          98.8   FALSE FALSE
## x34       2.0          99.4   FALSE FALSE
## x35       3.0          98.8   FALSE FALSE
## x36       2.0          99.4   FALSE FALSE
## x37       1.0         100.0   FALSE FALSE
## x38       2.0          99.4   FALSE FALSE
## x39       1.0          98.2   FALSE FALSE
## x40       2.0          99.4   FALSE FALSE
```

```
nzv_x[41:60,] #Only the columns needed
```

```
##     freqRatio percentUnique zeroVar   nzv
## x41      1.00          98.2   FALSE FALSE
## x42      1.00         100.0   FALSE FALSE
## x43      2.00          99.4   FALSE FALSE
## x44      1.00          95.8   FALSE FALSE
## x45      1.00          98.8   FALSE FALSE
## x46      1.00          96.4   FALSE FALSE
```

```
## x47        1.00          98.2   FALSE FALSE
## x48        1.00          98.2   FALSE FALSE
## x49        1.00          95.2   FALSE FALSE
## x50        1.25          80.6   FALSE FALSE
## x51        1.00          79.4   FALSE FALSE
## x52        1.00          73.9   FALSE FALSE
## x53        1.33          70.3   FALSE FALSE
## x54        1.25          68.5   FALSE FALSE
## x55        1.25          66.1   FALSE FALSE
## x56        1.00          64.2   FALSE FALSE
## x57        1.40          64.8   FALSE FALSE
## x58        1.25          64.8   FALSE FALSE
## x59        1.25          61.2   FALSE FALSE
## x60        1.20          58.8   FALSE FALSE
```

3.3) High correlated features

This dataset is relatively small but has many columns, and it is possible that some of them are so correlated and confounded that it is possible to use only one of them and achieve the same or even better results. The theory says 0.7 means a high correlation, so we will keep with it to make an arbitrary value of the cut. The correlation strength value can be tuned in the future if the final results are unsatisfactory.

```
#Computes the correlation
cor_trx <- cor(train_x) #Computes the correlation
#glimpse(cor_trx) #Check!

#Finding the indexes of columns with a correlation of more than 0.7
high_cor_trx <- findCorrelation(cor_trx, cutoff =.7)
#view(high_cor_trx) #Check!

#Arbitrary cut
train_x_uncor <- train_x[,-high_cor_trx]
test_x_uncor <- test_x[,-high_cor_trx]

#high_cor_trx #Columns to drop
```

3.4) Linear combinations

Some methods we may want to use can give a lot worst results if they do not have linear independence. The code would create a list with elements for each dependency containing vectors of column numbers as an index to be removed if needed posteriorly.

```
#Find if there are any linear combinations between the features.

#If the main dataset presents no linear combos, the non
#correlated should follow the same rule.
combo <- findLinearCombos(train_x_uncor)

#combo$remove #Check! It is NULL, so there is nothing to remove.
```

As can be seen, there is no such case.

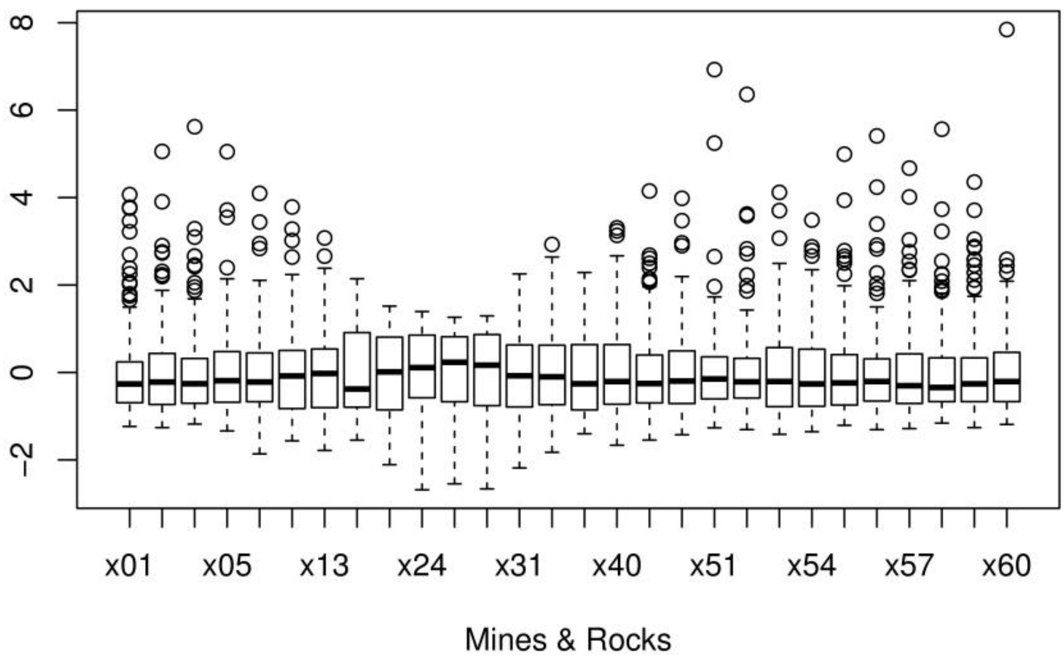
3.5) Center & Scaling

The process of centering and scaling the features enables the code to handle highly varying magnitudes between columns. If not done, the algorithms will fail to weigh the features since the most significant values will be seen as more critical, regardless of the magnitude unit of the values. Each column's mean is calculated and divided by its own standard deviation. After this technique, the values will be calculated as standard deviations from the mean and will be mathematically comparable. The dataset was previously normalized between 0 and 1 and could be used as is, but the centered and scaled perspective can give new insights.

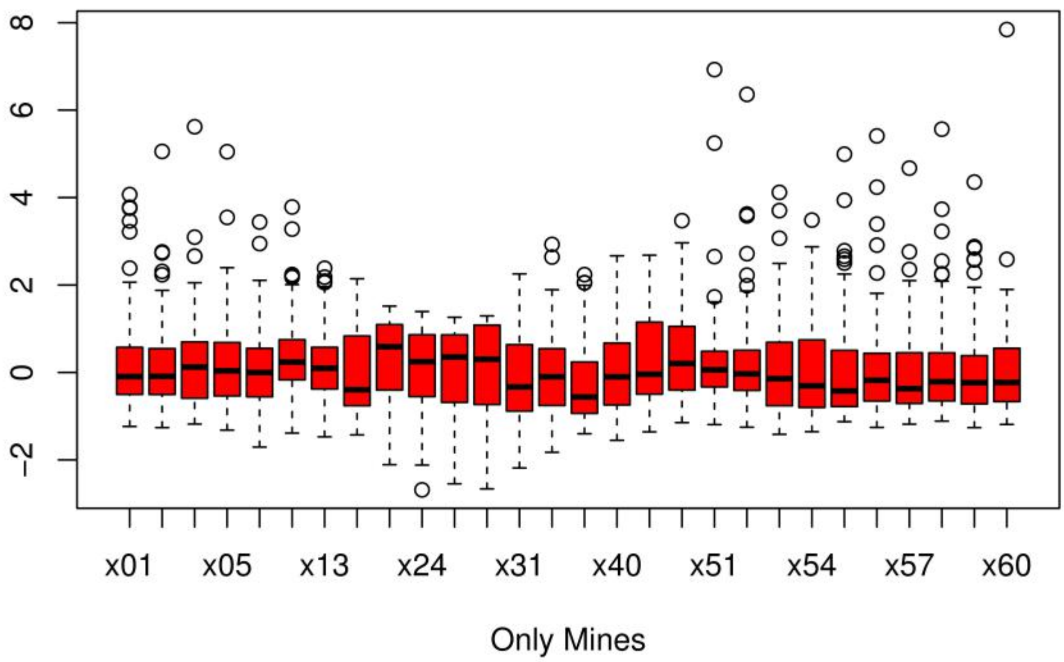
```
#No Scientific notation
options(scipen=999)

#Force the same outcome every time.
set.seed(87, sample.kind = "Rounding")

preProcValues <- preProcess(train_x_uncor,
                            method = c("center", #Center
                                       #average as 0
                                       "scale")) #Scale the
                                       #difference around the
                                       #average.

#Using the preprocessed values around train
#and not the entire dataset may prevent data
#leak because another way the average would not
#be from the train set but the whole dataset.
transf_trx <- predict(preProcValues, train_x_uncor)
transf_tex <- predict(preProcValues, test_x_uncor)

#transf_trx #Check!
#transf_tex #Check!

#Bind the outcome to the features we are
#going to use.
transf_trxy <- cbind(transf_trx,train_y)
transf_texy <- cbind(transf_tex,test_y)
```

The boxplot here has the same purpose as 2.3.1) Boxplot: Mines(RED) x Rocks(BLUE) but uses only train set data. Some people may find it easier to understand, so it is here not only to analyze the train set but as a comparison between normalization and the center and scale method.

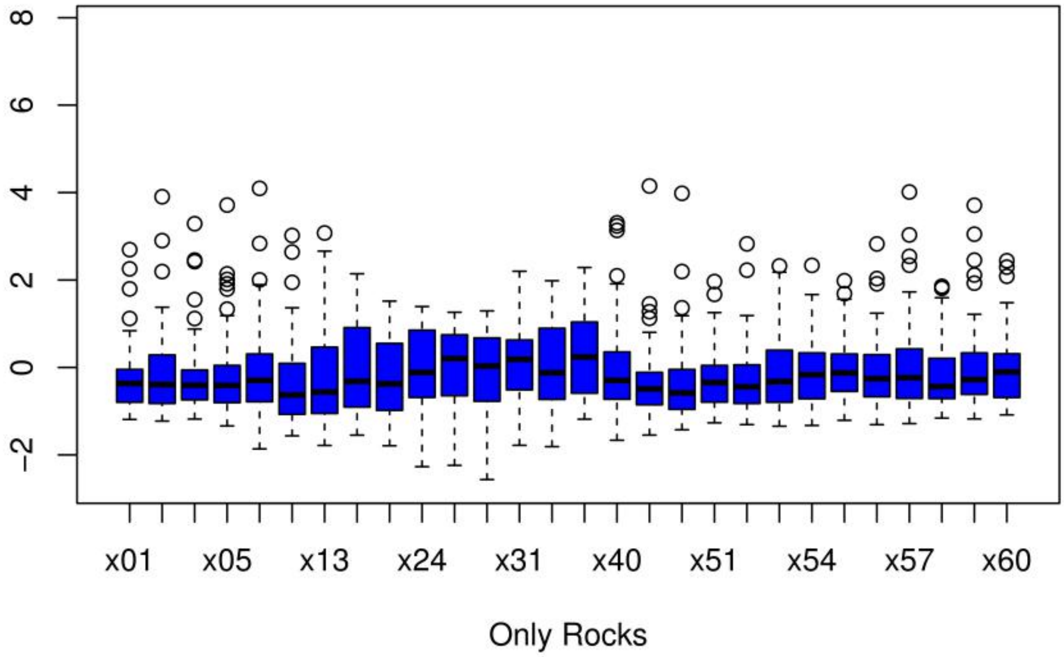
```
#Creates a boxplot with the Train set
transf_trx %>%
    boxplot(col="white", #White Boxes
            xlab="Mines & Rocks",
            ylim=c(min(transf_trx),
                   max(transf_trx))) #Y axis size
```




```r
#Creates a boxplot only with Mines
transf_trxy %>%
    filter(y=='M') %>% #Filter by Mines
    dplyr::select(-y) %>% #Remove Outcome
    boxplot(col="red", #Red Boxes
            xlab="Only Mines",
            ylim=c(min(transf_trx),
                   max(transf_trx))) #Y axis size
```




```
#Creates a boxplot only with Rocks
transf_trxy %>%
    filter(y=='R') %>% #Filter by Rocks
    dplyr::select(-y) %>% #Remove Outcome
    boxplot(col="blue", #Blue Boxes
            xlab="Only Rocks",
            ylim=c(min(transf_trx),
                   max(transf_trx))) #Y axis size
```



3.6) Outliers

As can be seen, some high values could be considered outliers, but intentionally they will be untouched. This comes from the perspective that the data passed through a process of security through obscurity; in other words, the features have been intentionally given no meaningful names and standardized from 0 to 1 to prevent too much knowledge of what the observation means and which equipment was used to collect them. Given this fact, there is a high chance of biasing the data if the outliers' removal is done since the data may already have passed previously by this process, and we lack the knowledge of each feature to make a sound judgment. We will endure and work with what we get the best we can.

4) METHODOLOGY

4.1) Ten-fold Cross Validation

The code instructs that the train_data must be folded into ten parts every time, with nine folds being used to calculate and one to validate the results. Every fold will be used once to validate so that it will be done ten times. There is always a chance of biasing the data by that particularly arbitrary cut every time we split data at random, and more so knowing our dataset is reasonably few on rows, so the process will be repeated five times and take the mean to balance the chance of random unlucky folds biasing the results.

```
fitControl <- trainControl(method = "repeatedcv",
                           number = 10, #10 Folds
                           repeats = 5) #5 Times
```

```r
#Creating a data frame to hold the results
model_frame <- data.frame(matrix(ncol = 3, nrow = 0))
cnames <- c("Method", "Sensitivity", "Accuracy")
colnames(model_frame) <- cnames
```

4.2) Methods

Multiple classification methods will be used, each with their hyperparameters tuned when needed and, in the end, have their Sensitivity to mine and overall Accuracy compared. Even though Sensitivity is the critical value, it is essential to ensure Accuracy is not too low. If the method always says it is a mine, the Sensitivity would be perfect, but the overall Accuracy would be low. Looking at both prevents this mistake.

4.2.1) Logistic Regression (GLM Binomial)

```r
#The seed to be reproducible with the same results
set.seed(87, sample.kind = "Rounding")

#Train the model
train_glm <- train(x=transf_trx, #Features
                   y=transf_trxy$y,#Outcomes
                   method = "glm", #method
                   family= "binomial",#Logit link Binomial
                   trControl=fitControl) #Folds & Repeats
#Saves the prediction
glm_preds <- predict(train_glm, transf_tex)

#Saves the results
glm_conf <- confusionMatrix(data=glm_preds,
                            reference= transf_texy$y)


#Saving the results to further comparison
model_frame <- rbind(model_frame,
                     c("glm",
                     glm_conf$byClass[[1]],
                     glm_conf$byClass[[11]]
                     )
                    )
#Reinforce the column names
colnames(model_frame) <- cnames
```

4.2.3) Linear Discriminant Analysis (LDA)

```r
set.seed(87, sample.kind = "Rounding")

train_lda <- train(x= transf_trx,
                   y= transf_trxy$y,
                   method = "lda",
```

```r
                  trControl=fitControl)

lda_preds <- predict(train_lda,  transf_tex)

lda_conf <- confusionMatrix(data=lda_preds,
                           reference= transf_texy$y)

model_frame <- rbind(model_frame,
                    c("lda",
                    lda_conf$byClass[[1]],
                    lda_conf$byClass[[11]]
                    )
                   )

colnames(model_frame) <- cnames
```

4.2.4) Quadratic Discriminant Analysis (QDA)

```r
set.seed(87, sample.kind = "Rounding")

train_qda <- train(x= transf_trx,
                  y= transf_trxy$y,
                  method = "qda",
                  trControl=fitControl)

qda_preds <- predict(train_qda,  transf_tex)

qda_conf <- confusionMatrix(data=qda_preds,
                           reference= transf_texy$y)

model_frame <- rbind(model_frame,
                    c("qda",
                    qda_conf$byClass[[1]],
                    qda_conf$byClass[[11]]
                    )
                   )
colnames(model_frame) <- cnames
```

4.2.5) Local Polynomial Regression Fitting (gamLoess)

```r
set.seed(87, sample.kind = "Rounding")

train_loess <- train(x=transf_trx,
                    y=transf_trxy$y,
                    method = "gamLoess",
                    trControl = fitControl)

loess_preds <- predict(train_loess, transf_tex)
```

```
loess_conf <- confusionMatrix(data=loess_preds,
                              reference=transf_texy$y)

model_frame <- rbind(model_frame,
                     c("loess",
                     loess_conf$byClass[[1]],
                     loess_conf$byClass[[11]]
                     )
                    )

colnames(model_frame) <- cnames
```

4.2.6) k Nearest Neighbors (kNN)

```
set.seed(87, sample.kind = "Rounding")

#The k nearest neighbors need a single number of
#neighbors to be calculated. It will try multiple
#hyperparameters here to use the best one.
tuning_knn <- data.frame(k = seq(2,30,2))

train_knn <- train(x=transf_trx,
                   y=transf_trxy$y,
                   method = "knn",
                   tuneGrid = tuning_knn, #As above.
                   trControl = fitControl)

knn_preds <- predict(train_knn, transf_tex)

knn_conf <- confusionMatrix(data=knn_preds,
                            reference=transf_texy$y)

model_frame <- rbind(model_frame,
                     c("knn",
                     knn_conf$byClass[[1]],
                     knn_conf$byClass[[11]]
                     )
                    )

colnames(model_frame) <- cnames
```

4.2.7) Random Forest (RF)

```
set.seed(87, sample.kind = "Rounding")

#Number of variables randomly sampled
#as candidates at each split.
tuning_rf <- data.frame(mtry = seq(2,10,2))
```

```r
train_rf <- train(x=transf_trx,
                  y=transf_trxy$y,
                  method = "rf",
                  tuneGrid = tuning_rf, #As above.
                  trControl= fitControl,
                  importance = TRUE)

#train_rf$bestTune #Check

rf_preds <- predict(train_rf, transf_tex)

rf_conf <- confusionMatrix(data=rf_preds,
                           reference=transf_texy$y)

model_frame <- rbind(model_frame,
                     c("rf",
                     rf_conf$byClass[[1]],
                     rf_conf$byClass[[11]]
                     )
                    )
colnames(model_frame) <- cnames
```

5) RESULTS

```r
#order by Sensitivity, our critical value
ord_res_tb <- model_frame %>%
  arrange(desc(Sensitivity))

#Saving to output when needed
out_res <- kable(ord_res_tb) %>%
  kable_paper()

#Forcing position of the table
kable_styling(out_res,
              latex_options = "hold_position")
```

Method	Sensitivity	Accuracy
rf	0.956521739130435	0.803260869565217
qda	0.91304347826087	0.731521739130435
knn	0.91304347826087	0.806521739130435
glm	0.826086956521739	0.81304347826087
lda	0.826086956521739	0.763043478260869
loess	0.826086956521739	0.763043478260869

6) CONCLUSION

The Random Forest model can be used on top of the sonar operator's expertise to enhance the operation's success without an overall accuracy that could hinder the ship's advance too much to be unusable. There is more room for improvement with more time and computational power invested into a more robust tunning. Besides that, the model could benefit from a newer version of the dataset with more observations, and knowing more about each feature could help narrow the outliers even more.

7) REFERENCES

7.1) SONAR MINE DATASET

[1] DALVI, M. (2021, JULY). SONAR MINE DATASET, Version 1. Retrieved August 22, 2022, from https://www.kaggle.com/datasets/mayurdalvi/sonar-mine-dataset