ISYE 6501X Week 1 Homework

Clayton Gerstner

Question 2.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a classification model would be appropriate. List some (up to 5) predictors that you might use.

This is an easy one for me. As a systems engineer in IT, basically anything I do with automation requires some kind of classification model, whether or not we call it that. For this purpose I will select a common target for automation in my regular life: classifying users as "normal" vs "hacked or malicious." Some predictors we might use:

- **outbound mail over time:** especially, high positive delta in outbound mail. One of the most common cases in a hacked or malicious user is running a spambot, so outbound mail numbers are good predictors.
- **egress packets per second:** Another very common case is a user packet flooding as part of a botnet or brute forcing passwords. This results in unusually high network traffic, so that's a good predictor.
- **high server load:** Cryptocurrency mining, or other computationally expensive underhandedness, also happens often in hacked users, so observing CPU load can be a good predictor.
- **external reports:** You probably wouldn't automate against it, but when someone is up to no good on your infrastructure on the Internet, people tell you about it. Counting reports of other people seeing malicious activity would be a good predictor; you could even weight it on the quality of the reporting source.

Question 2.2

We'll start with loading our libraries and collecting and reading our data. This is the headerless file because... well, the "features" are meaningless. For convenience and clarity, we'll also factorize the results, column 11, especially for KNN later.

```r
options(scipen = 999)
library(kernlab)  # For SVM
library(kknn)     # For KNN
library(ggplot2)  # For plotting some things
set.seed(1234)    # Reproducibility
datafile <- "cc-data.txt"
if (!file.exists(datafile)) {
  url <- paste0(c("https://prod-edxapp.edx-cdn.org/assets/courseware/v1/39b78ff5c5c28",
                  "981f009b54831d81649/asset-v1:GTx+ISYE6501x+2T2018+type@asset+block/",
                  "2.2credit_card_dataSummer2018.txt", collapse = ''))
  download.file(url, datafile)
}
cc_data <- read.table(datafile)
cc_data[,11] <- as.factor(cc_data[,11])
```

Now to the questions.

Q2.2 Part 1 - SVM with vanilladot

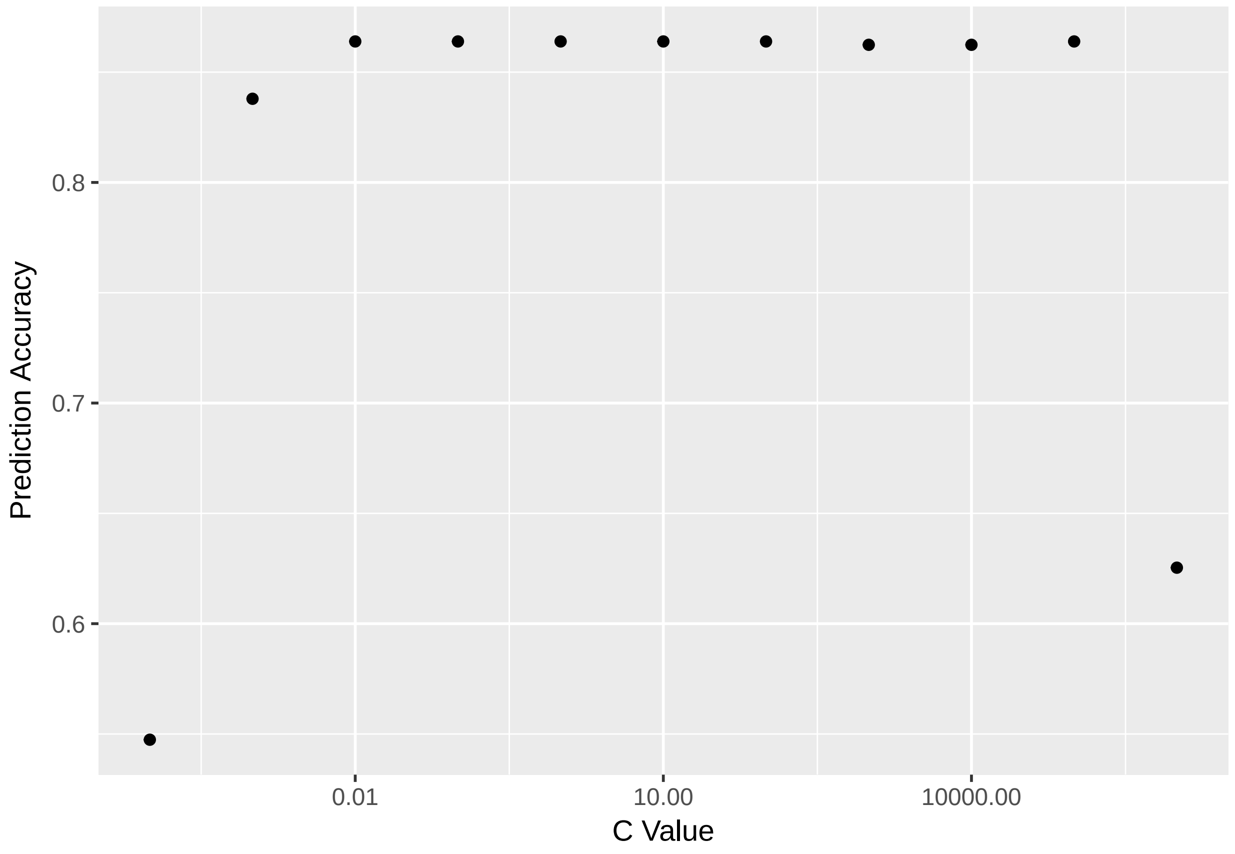
Using the support vector machine function ksvm contained in the R package kernlab, find a good classifier for this data. Show the equation of your classifier, and how well it classifies the data points in the full data set. (Don't worry about test/validation data yet; we'll cover that topic soon.)

Before answering this question directly, I'm personally interested in picking a good C value, so let's try some different orders of magnitude to see what the effect is and what is best.

```r
ooms <- c(10^(-4:6))
svm_accuracies <- double()
for (C_val in ooms) {
  svm_model <- ksvm(as.matrix(cc_data[,1:10]), cc_data[,11],
                                type = 'C-svc', kernel = "vanilladot",
                                C = C_val, scaled = TRUE)
  svm_accuracies[match(C_val, ooms)] <- sum(predict(svm_model, cc_data[,1:10]) ==
                                                    cc_data[,11]) / nrow(cc_data)
}
```

And I feel like plotting it to see if there are any relationships that pop up visually.

```r
  qplot(ooms, svm_accuracies, log = 'x', xlab = "C Value", ylab = "Prediction Accuracy")
```



Well that was pretty boring and nearly a waste of time. Regardless, it doesn't look it matters much for right now, so we'll just stick with 100 in the middle. We have learned that < 10^-4 is worthless, at least!

So we'll get our C = 100 back.

```r
svm_model <- ksvm(as.matrix(cc_data[,1:10]), cc_data[,11],
                  type = 'C-svc', kernel = "vanilladot",
                  C = 100, scaled = TRUE)
```

And we see that our error in the svm_model is pre-calculated to

```
svm_model@error
```

```
## [1] 0.1360856
```

Which lines up with our previously calculated and graphed accuracy, meaning correct + incorrect = 1:

```
svm_accuracy <- sum(predict(svm_model, cc_data[,1:10]) == cc_data[,11]) / nrow(cc_data)
svm_accuracy
```

```
## [1] 0.8639144
```

Now for the equation of this classifier. First we get the coefficients and the beginning of a nice long string for the function.

```
coeffs <- colSums(svm_model@xmatrix[[1]] * svm_model@coef[[1]])
equation <- paste0('(', signif(coeffs, 4), ' * ', names(coeffs), ') + ', collapse = '')
```

Now we finish it off with the intercept and the equality

```
equation <- paste0(equation, signif(-svm_model@b, 4), " = 0", collapse = "")
strwrap(equation, 60)
```

```
## [1] "(-0.001007 * V1) + (-0.001173 * V2) + (-0.001626 * V3) +"
## [2] "(0.003006 * V4) + (1.005 * V5) + (-0.002826 * V6) +"
## [3] "(0.00026 * V7) + (-0.000535 * V8) + (-0.001228 * V9) +"
## [4] "(0.1064 * V10) + 0.08158 = 0"
```

Q2.2 Part 2 - Other Kernels

You are welcome, but not required, to try other (nonlinear) kernels as well; we're not covering them in this course, but they can sometimes be useful and might provide better predictions than vanilladot.

We'll begin by making a manual list of available kernels from the kernlab package for us to iterate over like we did with C values. It would be fun to re-run all of these with different C values, but that would be a lot of processing and I'm just getting carried away. Therefore, we'll leave C=100.

```
kernels <- c('rbfdot', 'polydot', 'tanhdot', 'laplacedot', 'besseldot',
             'anovadot', 'splinedot', 'vanilladot')

out_string <- c("Kernel Accuracy:\n")
for (kern in kernels) {
  svm_model <- ksvm(as.matrix(cc_data[,1:10]), cc_data[,11],
                    type = 'C-svc', kernel = kern,
                    C = 100, scaled = TRUE)
  out_string <- c(out_string, kern, " accuracy: ",
                    as.character(sum(
                      predict(svm_model, cc_data[,1:10]) ==
                              cc_data[,11]) / nrow(cc_data)
                      ),
                    "\n"
                    )
}
```

Let's see what we've got:

```
cat(out_string)
```

```
## Kernel Accuracy:
##  rbfdot  accuracy:  0.957186544342508
##  polydot  accuracy:  0.863914373088685
##  tanhdot  accuracy:  0.7217125382263
##  laplacedot  accuracy:  1
##  besseldot  accuracy:  0.925076452599388
##  anovadot  accuracy:  0.906727828746177
##  splinedot  accuracy:  0.978593272171254
##  vanilladot  accuracy:  0.863914373088685
```

It looks like polydot has the same accuracy as vanilladot, tanhdot didn't do very well at all with only 72.17%, and everything else beat vanilla & poly. Also noticeable, the Laplacian kernel model caught them all.

Q2.2 Part 3 - K-Nearest Neighbors with kknn

Using the k-nearest-neighbors classification function kknn contained in the R kknn package, suggest a good value of k, and show how well it classifies that data points in the full data set. Don't forget to scale the data (scale=TRUE in kknn).

We'll split the data 70/30% for training/testing using random splitting with the sample() function. I chose a split weighted more towards the testing set because while this will give a worse model, it provides more data to actually be classified. Since this is a question about classification, I thought that would be more interesting than a better model classifying less data.

We'll then try some things to determine a good k value. The function default is 11, and we have >600 rows… let's try k = 5 - 95 counting by 10s.

```
set.seed(1234)  # seed reset for reproducibility
train_sample <- sample(nrow(cc_data), floor(nrow(cc_data) * .7))

k_vals <- seq(from = 5, to = 95, by = 10)

out_string <- "K Neighbors Count Accuracy\n"
for (k_val in k_vals) {
  knn_model <- kknn(V11 ~ ., cc_data[train_sample,], cc_data[-train_sample,],
                    k = k_val, scale = TRUE)
  out_string <- c(out_string,
                  "K = ", k_val, ": ",
                  as.character(sum(knn_model$fitted.values ==
                                     cc_data[-train_sample, 11]) /
                                 nrow(cc_data[-train_sample,])),
                  "\n")
}
cat(out_string)
```

```
## K Neighbors Count Accuracy
##  K =  5 :  0.842639593908629
##  K =  15 :  0.868020304568528
##  K =  25 :  0.873096446700508
##  K =  35 :  0.862944162436548
##  K =  45 :  0.857868020304569
##  K =  55 :  0.857868020304569
##  K =  65 :  0.857868020304569
##  K =  75 :  0.857868020304569
##  K =  85 :  0.857868020304569
##  K =  95 :  0.868020304568528
```

There's not a huge difference, but k = 25 eked out a win.

Question 3.1

Using the same data set as in Question 2.2, use the ksvm or kknn function to find a good classifier:

Though train.kknn in part A won't use it I'm going to split the data into three groups for the rest of this question: train == 1, validation == 2, and test == 3. In this fashion, I'll be combining parts A and B somewhat, generating cross-validated models with both KNN and SVM, and then using the third groups

```r
groups <- sample(1:3, nrow(cc_data), TRUE, c(.7, .15, .15))
```

Question 3.1 Part A

a) using cross-validation (do this for the k-nearest-neighbors model; SVM is optional;)

KNN:

We'll be using train.kknn to train based on leave-one-out cross validation. As we've done before we'll try all available kernels and see what kknn prefers.

```r
trained_kknn <- train.kknn(V11 ~ ., cc_data[groups == 1, ], kmax = 30, kernel = c("rectangular",
    "epanechnikov", "biweight", "triweight", "cos", "inv", "gaussian", "optimal"),
    scale = TRUE)
trained_kknn
```

```
##
## Call:
## train.kknn(formula = V11 ~ ., data = cc_data[groups == 1, ],     kmax = 30, kernel = c("rectangular"
##
## Type of response variable: nominal
## Minimal misclassification: 0.125
## Best kernel: rectangular
## Best k: 4
```

Running multiple iterations over different groups, I was interested to discover significant variance in the best kernel. inv, cos, and rectangular made regular appearances in my testing. The minimal misclassification, however, was always very close, indicating accuracy in the 80-90% area.

SVM Cross-validation

Next up, SVM. I'll use the tanhdot specifically because it performed the w orst above. It has the most room for improvement and will be interesting to see if it changes and/or if it performs as badly as expected.

```r
trained_svm <- ksvm(as.matrix(cc_data[,1:10]), cc_data[,11],
                    type = 'C-svc', kernel = "tanhdot",
                    cross = 25, C = 100, scaled = TRUE)
```

```r
trained_svm
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 100
##
## Hyperbolic Tangent kernel function.
##  Hyperparameters : scale =  1  offset =  1
##
## Number of Support Vectors : 184
##
## Objective Function Value : -18741416
## Training error : 0.278287
## Cross validation error : 0.275157
```

Still not great.

Part B

Now let's use the 3-way split groups to select a model with SVM. I'll spare the optional KNN as that would add a lot to an already long assignment.

We'll combine our earlier checking of C values with our earlier checking of kernels. In this fashion we can easily make a quick list of combinations to validate, and then test, against. I'll also use far fewer C values than before, focusing around 10.

```r
c_vals <- c(.1, 1, 10, 100, 1000)
models <- data.frame(Kernel = factor(),
                     C = double(),
                     Error = double())

for (kernel in kernels) {
  for (c_val in c_vals) {
    svm_model <- ksvm(as.matrix(cc_data[groups == 1,1:10]),
                      cc_data[groups == 1,11],
                      type = 'C-svc', kernel = kernel, C = c_val,
                      scaled = TRUE)
    new_row <- data.frame(Kernel = kernel,
                          C = c_val,
                          Error = svm_model@error)
    models <- rbind(models, new_row)
  }
}
```

```r
models[order(models$Error),]
```

```
##        Kernel      C      Error
## 19 laplacedot  100.0 0.00000000
## 20 laplacedot 1000.0 0.00000000
## 5      rbfdot 1000.0 0.00862069
## 33  splinedot   10.0 0.00862069
## 34  splinedot  100.0 0.01077586
## 35  splinedot 1000.0 0.01077586
## 32  splinedot    1.0 0.01293103
## 4      rbfdot  100.0 0.03232759
## 18 laplacedot   10.0 0.03879310
## 31  splinedot    0.1 0.04741379
## 3      rbfdot   10.0 0.06681034
## 25  besseldot 1000.0 0.07327586
## 24  besseldot  100.0 0.07758621
## 30   anovadot 1000.0 0.07758621
## 23  besseldot   10.0 0.08620690
## 29   anovadot  100.0 0.09698276
## 2      rbfdot    1.0 0.10560345
## 22  besseldot    1.0 0.11206897
## 28   anovadot   10.0 0.11206897
## 27   anovadot    1.0 0.11637931
## 16 laplacedot    0.1 0.12068966
## 6     polydot    0.1 0.12284483
## 7     polydot    1.0 0.12284483
## 8     polydot   10.0 0.12284483
## 9     polydot  100.0 0.12284483
## 17 laplacedot    1.0 0.12284483
## 36 vanilladot    0.1 0.12284483
## 37 vanilladot    1.0 0.12284483
## 38 vanilladot   10.0 0.12284483
## 39 vanilladot  100.0 0.12284483
## 1      rbfdot    0.1 0.12500000
## 10    polydot 1000.0 0.12500000
## 26   anovadot    0.1 0.12500000
## 40 vanilladot 1000.0 0.12500000
## 21  besseldot    0.1 0.13362069
## 11    tanhdot    0.1 0.21982759
## 14    tanhdot  100.0 0.25862069
## 15    tanhdot 1000.0 0.25862069
## 12    tanhdot    1.0 0.26508621
## 13    tanhdot   10.0 0.27586207
```

There's some very interesting stuff in there, but for brevity we'll just go for the top 3 kernels. Let's continue with testing on those, laplacedot at C = 100, rbfdot at C = 1000, splinedot at C = 10.

```r
c_vals <- c(100, 1000, 10)
kernels <- c("laplacedot", "rbfdot", "splinedot")
out_string <- "Model - Accuracy\n"

for (i in 1:length(c_vals)) {
  svm_model <- ksvm(as.matrix(cc_data[groups == 1,1:10]),
                    cc_data[groups == 1,11],
                    type = 'C-svc', kernel = kernels[i], C = c_vals[i],
                    scaled = TRUE)
  classified <- predict(svm_model, cc_data[groups == 2,1:10])
  accuracy <- sum(classified == cc_data[groups == 2,11]) / nrow(cc_data[groups == 2,])
  out_string <- c(out_string, kernels[i], " - ", accuracy, "\n")
}
```

```r
cat(out_string)
```

```
## Model - Accuracy
##  laplacedot  -  0.813084112149533
##  rbfdot  -  0.766355140186916
##  splinedot  -  0.719626168224299
```

And lastly, we'll test all 3 with the testing sample, at long last: group == 3.

```r
c_vals <- c(100, 1000, 10)
kernels <- c("laplacedot", "rbfdot", "splinedot")
out_string <- "Model - Accuracy\n"

for (i in 1:length(c_vals)) {
  svm_model <- ksvm(as.matrix(cc_data[groups == 1,1:10]),
                    cc_data[groups == 1,11],
                    type = 'C-svc', kernel = kernels[i], C = c_vals[i],
                    scaled = TRUE)
  classified <- predict(svm_model, cc_data[groups == 3,1:10])
  accuracy <- sum(classified == cc_data[groups == 3,11]) / nrow(cc_data[groups == 3,])
  out_string <- c(out_string, kernels[i], " - ", accuracy, "\n")
}
```

```r
cat(out_string)
```

```
## Model - Accuracy
##  laplacedot  -  0.807228915662651
##  rbfdot  -  0.771084337349398
##  splinedot  -  0.771084337349398
```

As expected, the validation and testing have dropped the actual accuracy considerably from the original predicted accuracy of the model using only the training data. It seems likely that these nonlinear models have suffered over-classification from the start with the high C values.