Model Comparison and Optimization Functions

Kevin McBeth w/ assistance from Chris Lach

Introduction

Modal Comparison is an important activity to be conducted during a data science investigation, due to it's side effect of selecting the highest performing model. One of the most realistic ways to judge models is by considering the nature of the problem, developing a cost function, and then instead of focusing solely on accuracy, consider the cost function to be the most important metric.

In this investigation, we look at five different model types, with four of them being considering using the cost function developed for the problem. These four models are knn, random forest, support vector machines, and finally linear regression. The fifth model is for completion's sake, and uses the keras package to develop a neural network.

In conclusion, the SVM and the random forest had the best performance when considering our defined function of +10 for a true positive and -3 for a false positive. These numbers were selected due to the outputs of the +4, -5 giving results less than 0 always due to the difficulty in identifying positives. We also considered the problem, which has a low cost for a customer saying no compared to a much higher reward for a customer saying yes. The cost is a few minutes of a marketing employee whereas the reward is financial profit due to the customer taking a long term bank account, which the bank could take bigger risks or long term positions with.

To start, we conduct our normal analysis of data cleaning, feature exploration, principal component analysis, and correlation plotting, before moving on to the models of KNN, Random Forest, SVM, logistic regression, and neural networks. We conclude by comparing the models, presented in a single data table.

In each model section, we tune the model, select the best hyperparameters, retrieve the prediction probabilities, calculate the ROC, confusion matrix, and cost function. The cost functions are plotted due to their primary importance, while the other features are added to our comparison table at the end.

Data Preparation

The code below illustrates further necessary steps for our analysis. It includes the data partioning into 80/20 train/test splits, creates a matrix for keras, and most importantly, creating the function of interest, the costFunction. The function takes prediction probabilities and the values for true positive and false positive in our cost function, and then creates an object with the data for each threshold point, the maximum cost, and the median of the thresholds which have a cost equal to the maximum cost.

```
index <- createDataPartition(bank.cleanDataTypes$y, p = 0.8)$Resample1
train <- bank.cleanDataTypes[index]
test <- bank.cleanDataTypes[-index]

#initiallizing vector for keeping model performance characteristics
results <- c()

#bank.matrix is for Keras. Keras only takes matrices, not data tables / data frames.
bank.matrix <- as.matrix(bank.cleanDataTypes[, names(bank.cleanDataTypes) := lapply(.SD, as.numeric)])

#numeric for PCA and corrplot
bank.features <- bank.cleanDataTypes[, names(bank.cleanDataTypes) := lapply(.SD, as.numeric)]

falsePositiveCost = -3
```

```r
truePositiveCost = 10

#function for optimality.
#Arguments: predictions - a vector of true probability predictions
#           TP - cost associated with a true positive
#           FP - cost associated with a false positive
#Returns: object containing data, bestCost, bestThreshold
costFunction <- function(predictions, TP, FP){
  finalCost <- c()
  for (thresholdVar in seq(0,1,.01)){
    predictionsClass <- as.numeric(predictions >= thresholdVar)
    object <- confusionMatrix(predictionsClass, numeric_targets)
    costVar <- object$table[4] * TP + object$table[2] * FP
    costObject <- list(Threshold=thresholdVar, cost=costVar)
    finalCost <- rbind(finalCost, row=costObject)
  }
  costDataTable <- data.table(finalCost)
  outputDataTable <- cbind(threshold=unlist(costDataTable$Threshold), cost=unlist(costDataTable$cost))
  outputDataTable <- data.table(outputDataTable)
  maxCost <- max(outputDataTable$cost)
  maxThreshold <- summary(outputDataTable[which(cost==maxCost),1])[4]
  output <- list(data=costDataTable, bestCost=maxCost, bestThreshold=maxThreshold)

  return(output)
}
```

Feature Exploration

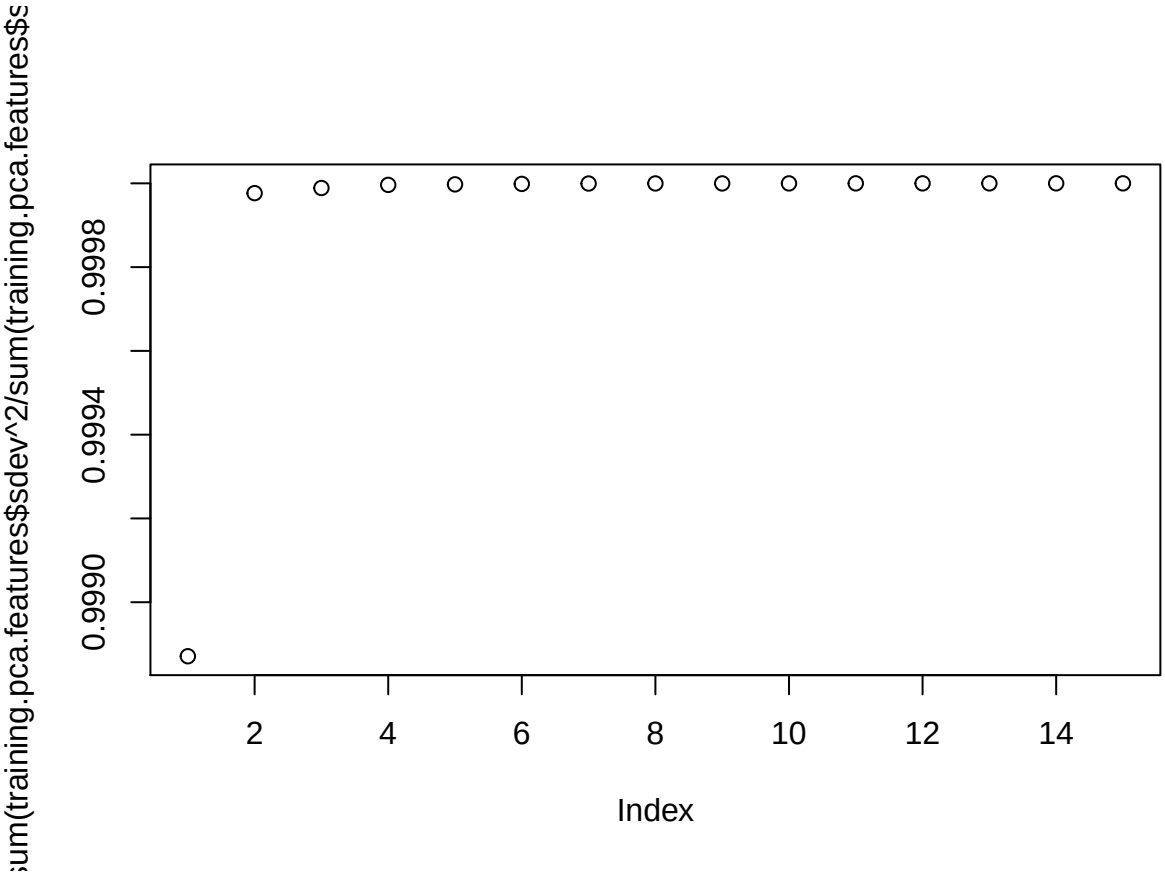
This section is lacking but is included for completeness. My understanding of ensemble learning involves rule based explorations, similar to decision trees. In it, the data would be segmented by rules, such as age. One of our later models, especially decision trees would perform similar segmenting, but in this brief example we're looking at a specifc group (individuals under 30) to see if there is a notable difference between the other statistics for those that say no and those that say yes. In this instance, there is little variance, which is further supported by the majority of our classifiers preferring to classify everything as no, due to it's prevalance being much higher than yes in our data set.

```r
#Example of looking at subsets odata for ensemble, rule based learning
#look for rules such as all individuals < 30 have a strong preference for ...
invisible(summary(train[which(train$age < 30 & train$y == "no")]))
invisible(summary(train[which(train$age < 30 & train$y == "yes")]))
```

Principal Component Analysis

A majority of our variance comes from balance, which makes sense at his by far the highest variance. The next highest variable is pdays, which also has a large difference. This indicates that scaling, which is conducted in most of our models, would significantly help to normalize these high variance points to smaller values. Still, since these variables seem to have little correlation (see next question), it is unlikely that model performance will significantly improve.

```r
training.pca.features <- prcomp(bank.features[,c(1:15)])
plot(cumsum(training.pca.features$sdev^2 / sum(training.pca.features$sdev^2)))
```



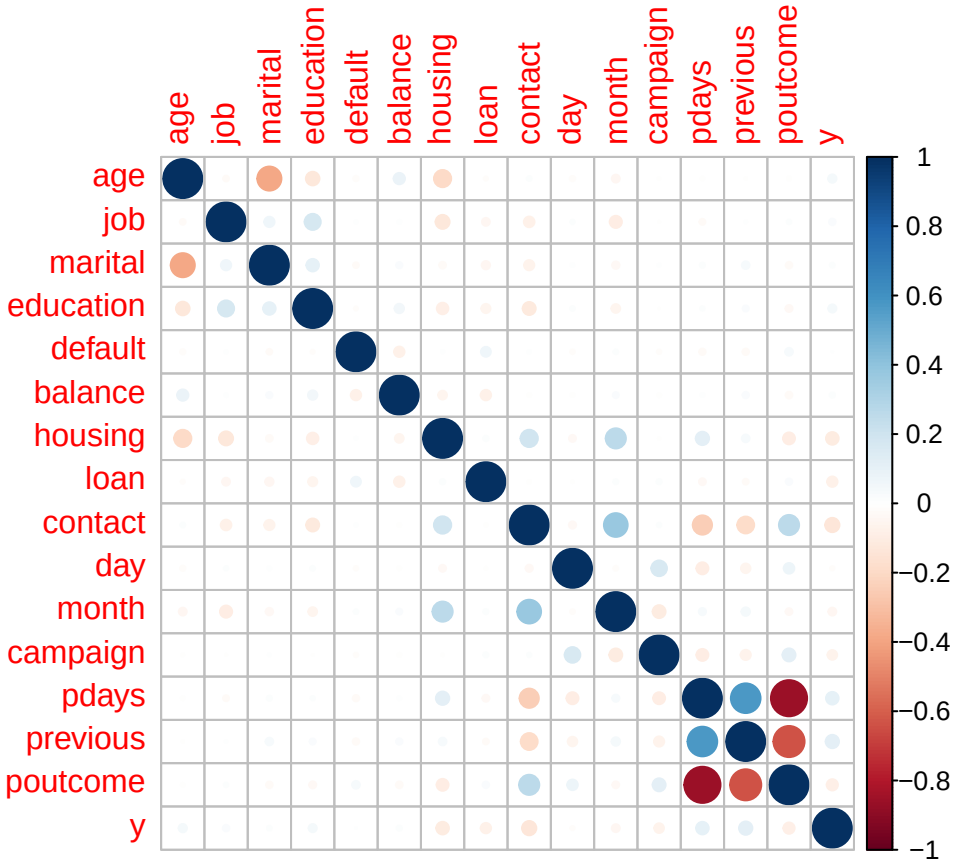

```
training.pca.features$rotation[,c(1,2)]
```

```
##                     PC1           PC2
## age        2.945567e-04 -1.033304e-03
## job        1.059740e-05 -7.439283e-04
## marital    4.975275e-06  1.009346e-04
## education  1.436089e-05  8.626616e-05
## default   -3.028334e-06 -3.292104e-05
## balance    9.999999e-01 -3.143946e-04
## housing   -8.272042e-06  5.812260e-04
## loan      -8.531575e-06 -1.092894e-04
## contact   -2.895764e-06 -2.188934e-03
## day       -2.378168e-05 -7.818879e-03
## month      2.306009e-05  9.934213e-04
## campaign  -1.030923e-05 -2.895874e-03
## pdays      3.142762e-04  9.998774e-01
## previous   1.474443e-05  9.767174e-03
## poutcome  -9.650562e-06 -8.511229e-03
```

Correlation Plot

There is not much correlation to the data for our target variable. Pdays, previous, and poutcome all seem to have strong correlation. As discussed previously, this indicates that we might need to collect more data on individual customers to differentiate them better. This is also supported by an inability to achieve high sensitivity without causing a high false positive rate. This will be supported by our optimization cost functions later in this document.

```
corrplot(cor(bank.features[,]))
```



Models

As introduced, we will now progress through training and finding cost function results for four models, followed by calculating accuracy for a neural networks. The models we will look at are KNN, Random Forest, SVM, and Logistic Regression.

KNN

The first model we will look at is KNN. Initially, we tested the number of clusters from 1 to 20. I chose to use 8 clusters, as performance gains were neglible after that, while the cohen's kappa was indicating random chance for better performance through it's oscillations. Looking at our best accuracy in the confusion matrix, we note a large number of no's and then an almost 50/50 split on our class of interest, positives. Looking at our cost function graph, we find that we can achieve better results at a lower threshold of around 0.2 to 0.25. This means that our model optimizes our function by forcing the model to classify any ambiguity as a positve for exploration. If we had a larger reward, this would push the curve maximum to the left, whereas if we had a larger punishment for false positives, the curve would be pushed to the right.

```
#Defining our Targets
numeric_targets <- as.numeric(test$y) - 1
trControl <- trainControl(method = "cv")
knn.model <- train(y~.,
             method     = "knn",
             tuneGrid   = expand.grid(k=1:20),
             trControl  = trControl,
             preProcess = c("center", "scale"),
             metric     = "Accuracy",
             data       = train)

knn.model
```

```
## k-Nearest Neighbors
##
## 3617 samples
##   15 predictor
##    2 classes: 'no', 'yes'
##
## Pre-processing: centered (41), scaled (41)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 3255, 3255, 3256, 3256, 3255, 3255, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    1  0.8377106  0.1657752
##    2  0.8299751  0.1356373
##    3  0.8659096  0.1404381
##    4  0.8706088  0.1736281
##    5  0.8811129  0.1675532
##    6  0.8825018  0.1857654
##    7  0.8816677  0.1698273
##    8  0.8822187  0.1712283
##    9  0.8819417  0.1587943
##   10  0.8833236  0.1561807
##   11  0.8844309  0.1383011
##   12  0.8827742  0.1247809
```

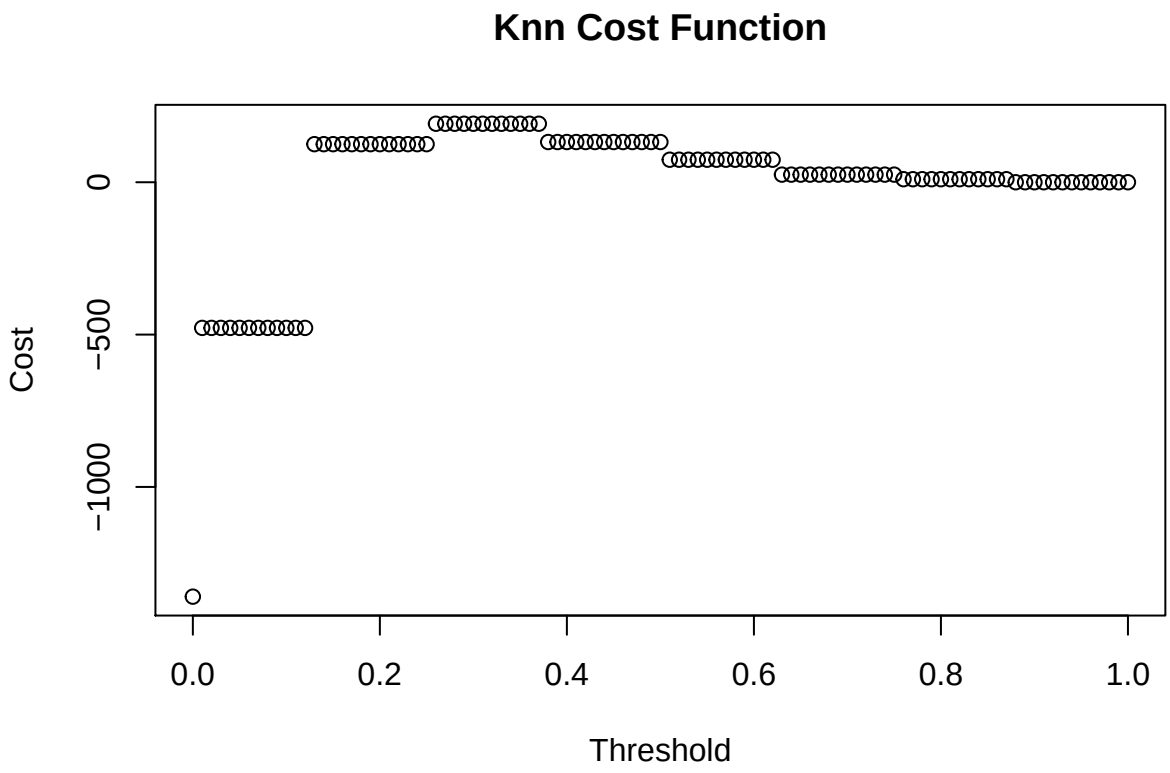
```
##   13  0.8844301  0.1351231
##   14  0.8855382  0.1427335
##   15  0.8871987  0.1560941
##   16  0.8888585  0.1603013
##   17  0.8869232  0.1466697
##   18  0.8888569  0.1535527
##   19  0.8880274  0.1432554
##   20  0.8866454  0.1347976
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 16.
```

```
##Selecting the best parameter without overfitting, k=10
knn.model <- train(y~.,
             method     = "knn",
             tuneGrid   = expand.grid(k=8),
             trControl  = trControl,
             preProcess = c("center", "scale"),
             metric     = "Accuracy",
             data       = train)


knn.predictions <- predict(knn.model, test, type='prob')
knn.predictions.raw <- predict(knn.model, test)
knn.prediction.probs <- knn.predictions[,2]
knn.roc <- roc(numeric_targets, knn.prediction.probs)
knnCost <- costFunction(knn.prediction.probs, truePositiveCost, falsePositiveCost)

print(confusionMatrix(knn.predictions.raw, test$y))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  779  88
##        yes  21  16
##
##                Accuracy : 0.8794
##                  95% CI : (0.8564, 0.8999)
##     No Information Rate : 0.885
##     P-Value [Acc > NIR] : 0.7198
##
##                   Kappa : 0.1773
##  Mcnemar's Test P-Value : 2.588e-10
##
##             Sensitivity : 0.9738
##             Specificity : 0.1538
##          Pos Pred Value : 0.8985
##          Neg Pred Value : 0.4324
##              Prevalence : 0.8850
##          Detection Rate : 0.8617
##    Detection Prevalence : 0.9591
##       Balanced Accuracy : 0.5638
##
##        'Positive' Class : no
```

```
##
```

```
plot(knnCost$data, main= "Knn Cost Function", xlab = "Threshold", ylab = "Cost")
```




```
results <- rbind(results, list(Model="knn", accuracy=confusionMatrix(knn.predictions.raw, test$y)$overal
```

Random Forest

Following the same procedure from before, we tune the model, identifying mtry at 4 to be the best performance. Outside of this code, we manually tuned the parameters ntree and nodesize to prevent overfitting and increasing the capacity of our model for prediction. In terms of accuracy, this model showed improvement over the knn model, and looking at the prediction quality for yes predictions note a much better performance at identifying potential long term balance holders. This is reflected by our cost function, exhibiting a higher peak than the knn model, at a lower threshold.

```
rf.model <- train(y ~.,
                  method = 'rf',
                  data = train,
                  trControl = trainControl(method = 'repeatedcv', number = 2, repeats = 2),
                  tuneGrid = expand.grid(mtry = c(1,2,4, 6, 8, 10, 12)),
                  ntree = 250,
                  nodesize=8)

rf.model
```

```
## Random Forest
##
```

```
## 3617 samples
##   15 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (2 fold, repeated 2 times)
## Summary of sample sizes: 1808, 1809, 1808, 1809
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    1    0.8847112  0.00000000
##    2    0.8851261  0.01541338
##    4    0.8887204  0.12841259
##    6    0.8896877  0.17005634
##    8    0.8884436  0.18623532
##   10    0.8883054  0.19207483
##   12    0.8883051  0.20419790
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 6.
```

```
rf.model <- train(y ~.,
                  method = 'rf',
                  data = train,
                  trControl = trainControl(method = 'repeatedcv', number = 2, repeats = 2),
                  tuneGrid = expand.grid(mtry = 4),
                  ntree = 250,
                  nodesize = 8)

rf.model
```

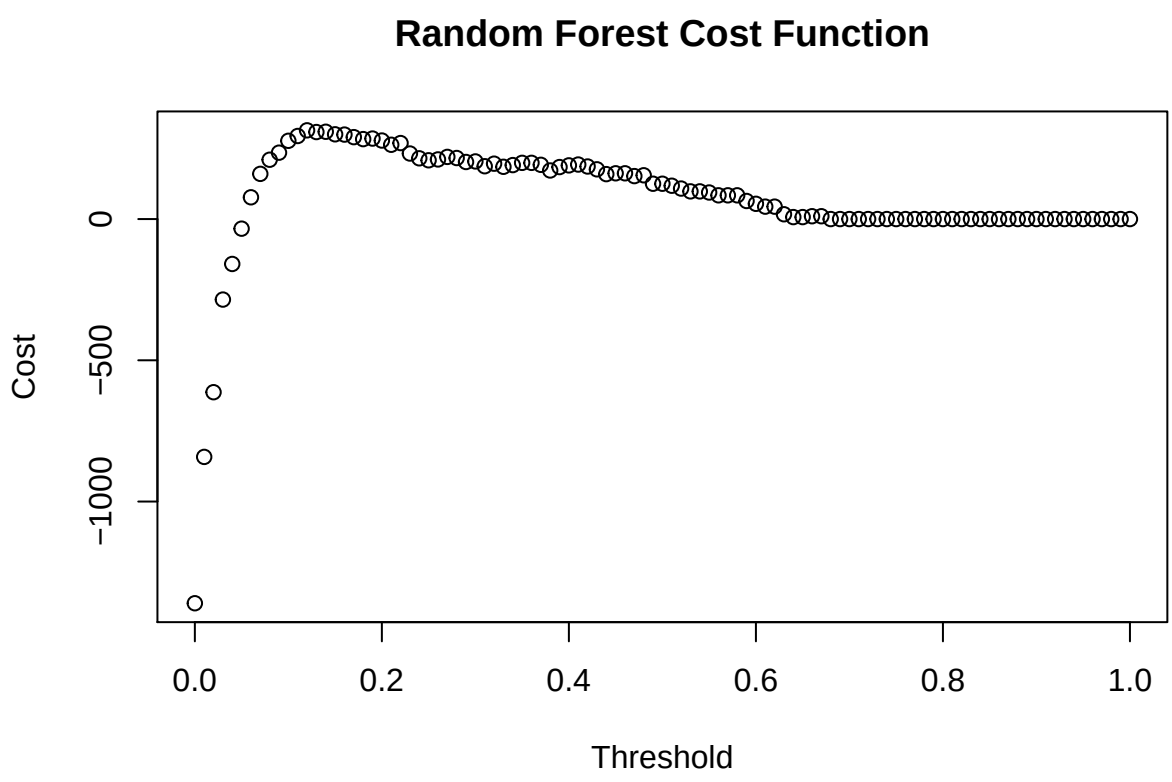
```
## Random Forest
##
## 3617 samples
##   15 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (2 fold, repeated 2 times)
## Summary of sample sizes: 1808, 1809, 1808, 1809
## Resampling results:
##
##   Accuracy   Kappa
##   0.8894111  0.1334278
##
## Tuning parameter 'mtry' was held constant at a value of 4
```

```
rf.predictions <- predict(rf.model, test, type = 'prob')
rf.predictions.raw <- predict(rf.model, test)
rf.prediction.probs <- rf.predictions[, 2]
rf.roc <- roc(numeric_targets, rf.prediction.probs)
rfCost <- costFunction(rf.prediction.probs, truePositiveCost, falsePositiveCost)

print(confusionMatrix(rf.predictions.raw, test$y))
```

```
## Confusion Matrix and Statistics
```

```
##
##          Reference
## Prediction  no yes
##        no  796  90
##        yes   4  14
##
##               Accuracy : 0.896
##                 95% CI : (0.8743, 0.9152)
##    No Information Rate : 0.885
##    P-Value [Acc > NIR] : 0.161
##
##                  Kappa : 0.2024
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.9950
##             Specificity : 0.1346
##          Pos Pred Value : 0.8984
##          Neg Pred Value : 0.7778
##              Prevalence : 0.8850
##          Detection Rate : 0.8805
##    Detection Prevalence : 0.9801
##       Balanced Accuracy : 0.5648
##
##        'Positive' Class : no
##
```

```r
plot(rfCost$data, main= "Random Forest Cost Function", xlab = "Threshold", ylab = "Cost")
```




```
results <- rbind(results, list(Model="rf", confusionMatrix(rf.predictions.raw, test$y)$overall[1], auc=
```

SVM

Following the same procedure, we tune the model, selecting gamma to be 0.01 and C to be 10. The SVM turns out to predict everything as no as it determines that to be the best accuracy measure. Instead, using our cost function, we identify a much lower threshold that can be used to achieve performance much more similar to our Random Forest. In fact, the maximum cost for the two are the same, at a similar threshold.

```
svm.model <- tune(svm,
              y ~ .,
              data = train,
              ranges = list(gamma = c(0.001, 0.01, 0.1),
                            cost = c(0.1, 1, 5, 10)),
              tunecontrol = tune.control(sampling = "cross", cross = 3),
              kernel = 'radial'
)

svm.model
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 3-fold cross validation
##
## - best parameters:
```

```
##  gamma cost
##   0.01    5
##
## - best performance: 0.107821
```

```
svm.model <- svm(y~.,
                 data = train,
                 kernel = 'radial',
                 gamma = 0.01,
                 C = 10,
                 probability = T)

svm.model
```

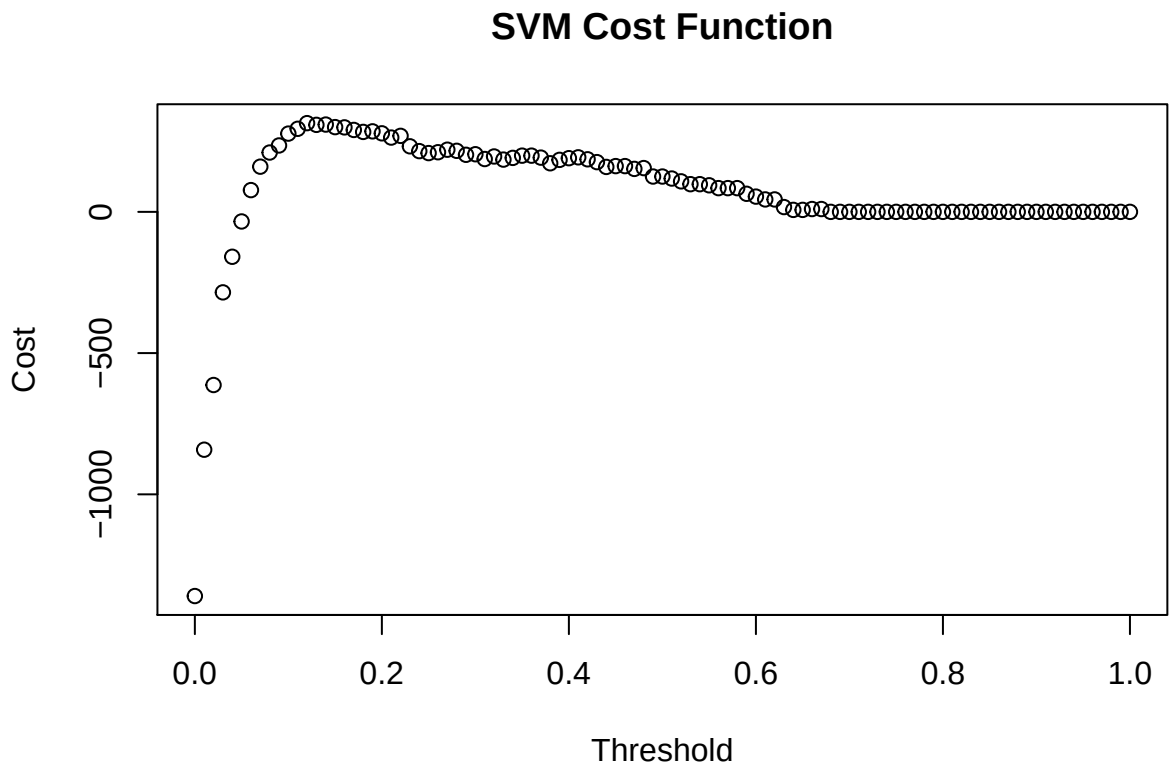
```
##
## Call:
## svm(formula = y ~ ., data = train, kernel = "radial", gamma = 0.01,
##     C = 10, probability = T)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.01
##
## Number of Support Vectors:  1162
```

```
svm.predictions <- predict(svm.model, test, probability = T)
svm.predictions.raw <- predict(svm.model, test)
svm.prediction.probs <- attr(svm.predictions, 'probabilities')[, 2]
svm.roc <- roc(numeric_targets, svm.prediction.probs)
svmCost <- costFunction(rf.prediction.probs, truePositiveCost, falsePositiveCost)

print(confusionMatrix(svm.predictions.raw, test$y))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  800 104
##        yes   0   0
##
##                Accuracy : 0.885
##                  95% CI : (0.8623, 0.905)
##     No Information Rate : 0.885
##     P-Value [Acc > NIR] : 0.5261
##
##                   Kappa : 0
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 1.000
##             Specificity : 0.000
##          Pos Pred Value : 0.885
##          Neg Pred Value :   NaN
##              Prevalence : 0.885
```

```
##          Detection Rate : 0.885
##    Detection Prevalence : 1.000
##       Balanced Accuracy : 0.500
##
##        'Positive' Class : no
##
```

```
plot(svmCost$data, main= "SVM Cost Function", xlab = "Threshold", ylab = "Cost")
```




```
results <- rbind(results, list(Model="svm", confusionMatrix(svm.predictions.raw, test$y)$overall[1], au
```

Logistic Regression

This model does not require any hyperparameter tuning (until cost function optimization). We see similar performance for the accuracy metric with knn, but worse predictive quality for yes results than the SVM and Random Forest models. The cost function has a similar threshold to KNN and the peak is likewise lower than the SVM and Random Forest models.

```
glm.model <- train(y~.,
                   data = train,
                   method = 'glm')
glm.model
```

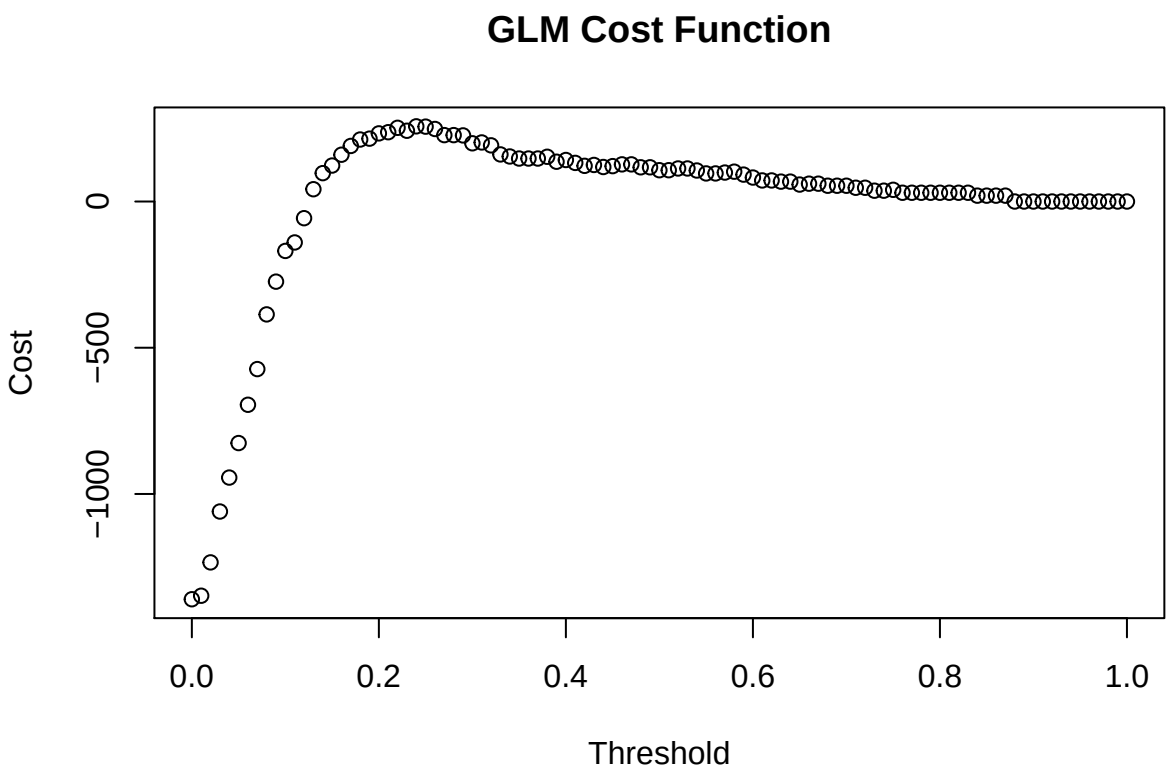
```
## Generalized Linear Model
##
## 3617 samples
##   15 predictor
```

```
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 3617, 3617, 3617, 3617, 3617, 3617, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.8893845  0.1849809
```

```
glm.predictions <- predict(glm.model, test, type = 'prob')
glm.predictions.raw <- predict(glm.model, test)
glm.predictions.train <- predict(glm.model, train)
glm.prediction.probs <- glm.predictions[,2]
glm.roc <- roc(numeric_targets, glm.prediction.probs)
glmCost <- costFunction(glm.prediction.probs, truePositiveCost, falsePositiveCost)

print(confusionMatrix(glm.predictions.raw, test$y))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  789  90
##        yes  11  14
##
##                Accuracy : 0.8883
##                  95% CI : (0.8659, 0.9081)
##     No Information Rate : 0.885
##     P-Value [Acc > NIR] : 0.402
##
##                   Kappa : 0.1805
##  Mcnemar's Test P-Value : 8.407e-15
##
##             Sensitivity : 0.9862
##             Specificity : 0.1346
##          Pos Pred Value : 0.8976
##          Neg Pred Value : 0.5600
##              Prevalence : 0.8850
##          Detection Rate : 0.8728
##    Detection Prevalence : 0.9723
##       Balanced Accuracy : 0.5604
##
##        'Positive' Class : no
##
```

```
plot(glmCost$data, main= "GLM Cost Function", xlab = "Threshold", ylab = "Cost")
```




```
results <- rbind(results, list(Model="glm", confusionMatrix(glm.predictions.raw, test$y)$overall[1], au
```

Neural Network

We include neural networks for completeness. This dataset isn't well suited for neural networks due to less than 5000 points of data. Still the neural net learns to classify everything as negative. In order to convert this in the same manner, the neural net needs to be altered to output values between 0 and 1, using an activation such as sigmoid or tanh. In our case we used sigmoid to output the probability.

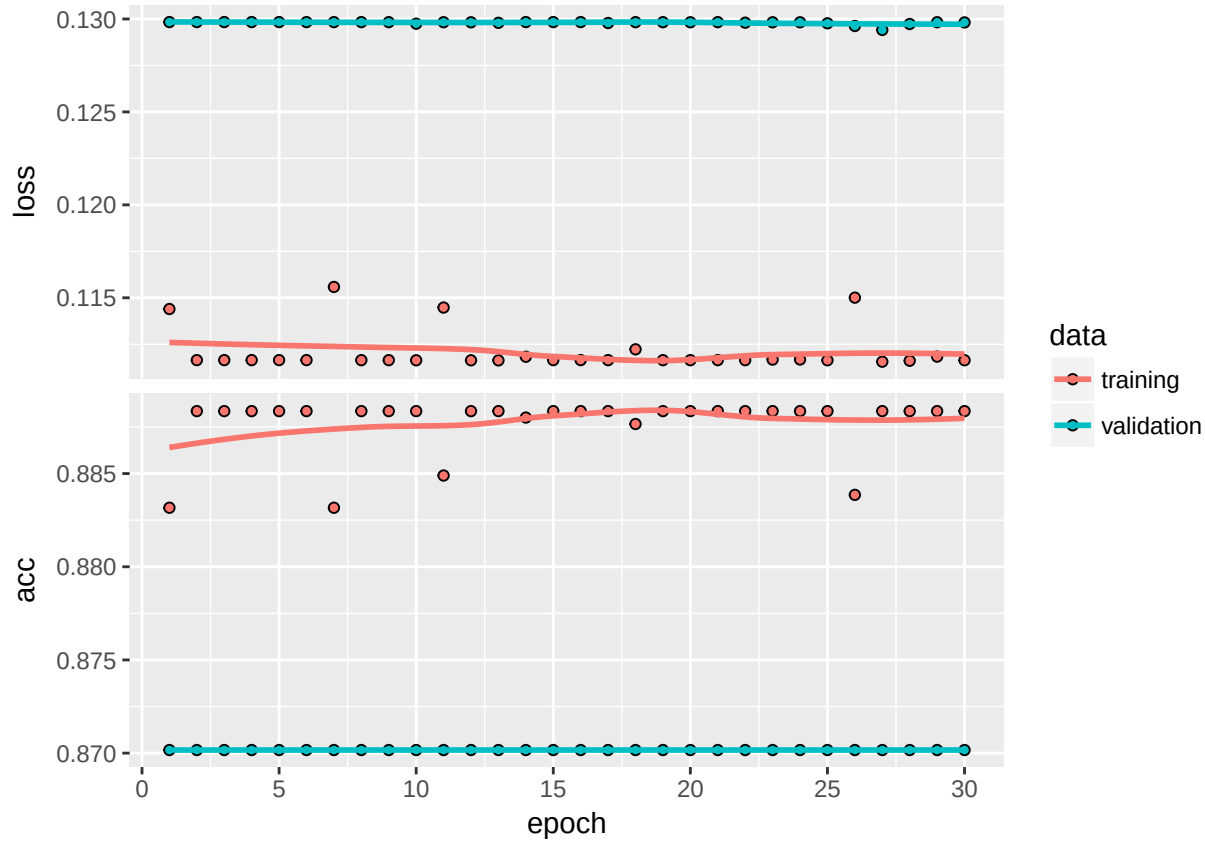
One interesting note, and a future option, as the code would not work with tensorflow backend functions, is developing an alternate optimizer based on the cost function. Effectively, the neural network would be forced to lower it's accuracy, similar to using something such as binary_crossentropy. Still, to do that the result would have to be classification to do so, preventing us from accessing the raw probability predictions.

This is left for another day, as it requires more or less to not use Keras or tensorflow without building a significant API that allows a user to look at the dataset not as flow but as total, such as done in the optimization function above. The data flow prevents looking at complicated statistics such as subsets similar to what one would see in a confusion matrix.

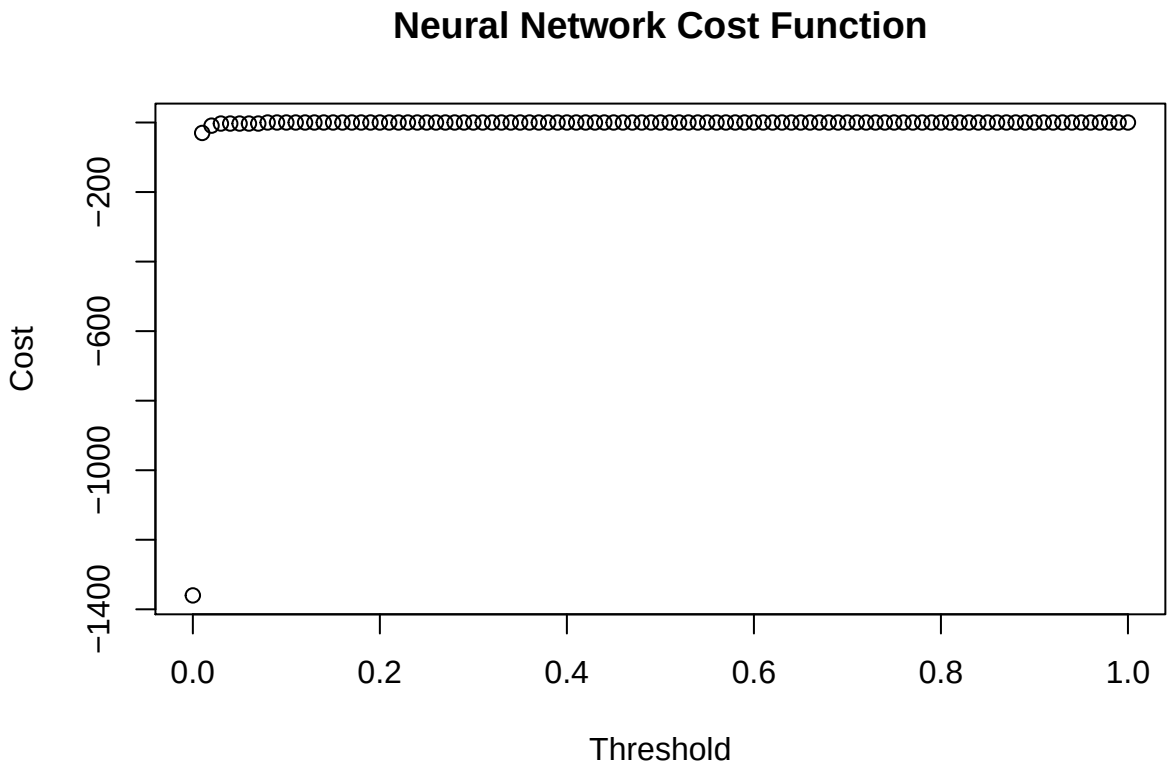
Still, we see that our model has worse predictive quality as currently performed, but would likely perform as well or better than the SVM and the random forest models above.

```
x_train <- bank.matrix[index,c(1:15)]
y_train <- bank.matrix[index,c(16)]

x_test <- bank.matrix[-index,c(1:15)]
y_test <- bank.matrix[-index,c(16)]
```

```
y_train <- y_train - 1
y_test <- y_test - 1


neural.model <- keras_model_sequential()
neural.model %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(15)) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 1, activation = 'sigmoid')

neural.model %>% compile(
  loss = 'mse',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)

history <- neural.model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)

plot(history)
```



```
neural.model %>% evaluate(x_test, y_test)

neuralCost <- costFunction(10*predict(neural.model,x_test), truePositiveCost, falsePositiveCost)
plot(neuralCost$data, main= "Neural Network Cost Function", xlab = "Threshold", ylab = "Cost")
```




```
results <- rbind(results, list(Model="neuralNet", accuracy=evaluate(neural.model, x_test, y_test)$acc, a
```

Conclusion

Looking at our final results, we see that our random forest model and support vector machine gave the best maximum Cost at 314, at a mean threshold of 0.12. This is a function of the cost function itself, as if the rewards were increased the threshold would push left and the peak push up. If the punishments were increased the threshold would push right and the peak would push down. Our best performance lay with our random forest, and as such, is the best predictive model of these five models in their current tuned states.

If the neural network could be optimized under the cost function, I believe it would outperform the other models. Also, if one were to use the random forest and svm models, one might be able to achieve a slightly better performance on the cost optimization function, and thus create a better overall vote.

Overall, we've seen that accuracy is not the best metric for our real world problem of maximizing our cost function. As such, it is extremely important to identify good cost/reward functions for maximum applicability of machine learning to real world problems.

```
print(results)
```

```
##      Model       accuracy  auc       maxCost maxThreshold
## [1,] "knn"       0.8794248 0.6664543 192     "Mean   :0.3150  "
## [2,] "rf"        0.8960177 0.7463401 314     "Mean   :0.12  "
## [3,] "svm"       0.8849558 0.7463401 314     "Mean   :0.12  "
## [4,] "glm"       0.8882743 0.7311058 257     "Mean   :0.24  "
## [5,] "neuralNet" 0.8849558 NA        0       "Mean   :0.54  "
```