He Said / She Said

Author Classification for Twitter Metadata in #Election2016

Thadryan J. Sweeney - Northeastern University - Data Mining and Machine Learning

Project Goals

This project will classify which presidential candidate composed a given tweet in the 2016 presidential election based on numerical values.

While it may seem counter-intuitive to analyze text without considering the actual words, using numerical values might allow for a computationally lighter alternative to traditional text mining while avoiding some of the complexities and ambiguity that can arise from natural language. This model is a proof of concept for such an approach in binary classification.

The project will follow a slightly modified CRISP-DM structure. Our approach will vary in that model evaluation will be conducted on an ongoing basis to assess candidates for an ensemble model which will be the subject of the evaluation phase of the framework.

We will explore the model using some simple side-by side comparisons to get a general, intuitive sense of where the differences lie before using regression analysis to asses statistical significance. From there, we will create assorted models and evaluate them based on simple accuracy and RMSE. Given that this is binary classifier with comparable entries from each author, simple accuracy will be more useful than it is with large skews in the.

Data Understanding

This dataset is a heavily re-engineered version of the Trump and Hillary Tweets dataset reviewed and presented by Kaggle. The original dataset includes Boolean values such as retweet or quote status and easily factorable information like who the tweet was directed at or who originally wrote it. In addition, I've written a Python function to extract numeric metadata such as number of sentences, average sentence length, average word length, number of URLs, and number of "@" or "#" symbols, etc. ### Loading and Inspecting the Dataset

In this section, we'll load our data and inspect it in a few different ways. After reading the dataset from a CSV file, we will be able to see the full set of available information.

```r
# get the data, check the number of rows
data <- read.csv("engineered_tweet_data.csv", header = TRUE, stringsAsFactors = FALSE)
data$Commas <- NULL # The commas feature gets erased by the red csv function but isn't needed
str(data)
```

```
## 'data.frame':    6112 obs. of  23 variables:
##  $ handle                 : chr  "HillaryClinton" "HillaryClinton" "HillaryClinton" "HillaryClinton"
##  $ text                   : chr  "The question in this election: Who can put the plans into action t
##  $ is_retweet             : chr  "False" "True" "True" "False" ...
##  $ original_author        : chr  "" "timkaine" "POTUS" "" ...
##  $ in_reply_to_screen_name: chr  "" "" "" "" ...
##  $ is_quote_status        : chr  "False" "False" "False" "False" ...
##  $ lang                   : chr  "en" "en" "en" "en" ...
##  $ retweet_count          : int  218 2445 7834 916 859 2181 1303 1833 4132 1087 ...
##  $ favorite_count         : chr  "651" "5308" "27234" "2542" ...
##  $ truncated              : chr  "False" "False" "False" "False" ...
##  $ Words                  : int  18 17 23 15 16 17 16 7 17 17 ...
##  $ Hashtags               : int  0 0 0 0 0 0 1 0 0 0 ...
##  $ At_signs               : int  0 0 2 0 0 0 0 0 0 1 ...
##  $ URLs                   : int  1 1 0 2 1 1 1 2 2 1 ...
##  $ Words_in_caps          : num  0 0 1 0 0 1 1 0 1 1 ...
##  $ Mean_word_length       : num  4.44 4.59 4.96 4.27 5.12 ...
##  $ Periods                : int  0 2 2 1 2 0 2 3 1 1 ...
##  $ Exclamation_points     : int  0 0 0 0 0 1 0 0 1 0 ...
##  $ Question_marks         : int  1 1 0 0 0 0 0 0 0 0 ...
##  $ Colon                  : int  1 0 0 1 0 1 0 1 0 0 ...
##  $ Semi.colons            : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Number_of_sentences    : num  2 3 2 2 2 2 3 1 2 2 ...
##  $ Mean_sentence_length   : num  9 5.67 11.5 7.5 8 ...
```

This shows that our tweaks resulted in usable data but we want only numbers, so we will iterate though and make binary dummy codes or IDs as appropriate.

Data Preparation

```r
# make a copy of the data to manipulate
num.data <- data

# the text is no longer needed
num.data$text <- NULL

# binary dummy codes for Clinton and Trump
num.data$handle[which(num.data$handle == "HillaryClinton") ] <- 1
num.data$handle[which(num.data$handle == "realDonaldTrump") ] <- 0

# simple binary codes for booleans
num.data$is_retweet[which((num.data$is_retweet) == "True")] <- 1
num.data$is_retweet[which((num.data$is_retweet) == "False")] <- 0

num.data$is_quote_status[which((num.data$is_quote_status) == "True")] <- 1
num.data$is_quote_status[which((num.data$is_quote_status) == "False")] <- 0

num.data$truncated[which((num.data$truncated) == "True")] <- 1
num.data$truncated[which((num.data$truncated) == "False")] <- 0

# numeric/factor data associated to create an ID-like structure
# for example this will assign each language a number, each original
# author a number, and each "in reply" unique number
num.data$lang <- as.numeric(factor(num.data$lang))
num.data$original_author <- as.numeric(factor(num.data$original_author))
num.data$in_reply_to_screen_name <- as.numeric(factor(num.data$in_reply_to_screen_name))

# iterate through the dataset and convert to numeric
for (i in 1:ncol(num.data) )
{
  num.data[,i] <- as.numeric(num.data[,i])
}
```

```
## Warning: NAs introduced by coercion
```

```r
# inspec the dataset to verify it worked the way we want it to
str(num.data)
```

```
## 'data.frame':    6112 obs. of  22 variables:
##  $ handle                 : num  1 1 1 1 1 0 1 1 0 1 ...
##  $ is_retweet             : num  0 1 1 0 0 0 0 0 0 1 ...
##  $ original_author        : num  1 253 198 1 1 1 1 1 1 164 ...
##  $ in_reply_to_screen_name: num  1 1 1 1 1 1 1 1 1 1 ...
##  $ is_quote_status        : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ lang                   : num  3 3 3 3 3 3 3 3 3 3 ...
##  $ retweet_count          : num  218 2445 7834 916 859 ...
##  $ favorite_count         : num  651 5308 27234 2542 2882 ...
##  $ truncated              : num  0 0 0 0 0 1 0 0 0 0 ...
##  $ Words                  : num  18 17 23 15 16 17 16 7 17 17 ...
##  $ Hashtags               : num  0 0 0 0 0 0 1 0 0 0 ...
##  $ At_signs               : num  0 0 2 0 0 0 0 0 0 1 ...
##  $ URLs                   : num  1 1 0 2 1 1 1 2 2 1 ...
##  $ Words_in_caps          : num  0 0 1 0 0 1 1 0 1 1 ...
##  $ Mean_word_length       : num  4.44 4.59 4.96 4.27 5.12 ...
##  $ Periods                : num  0 2 2 1 2 0 2 3 1 1 ...
##  $ Exclamation_points     : num  0 0 0 0 0 1 0 0 1 0 ...
##  $ Question_marks         : num  1 1 0 0 0 0 0 0 0 0 ...
##  $ Colon                  : num  1 0 0 1 0 1 0 1 0 0 ...
##  $ Semi.colons            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ Number_of_sentences    : num  2 3 2 2 2 2 3 1 2 2 ...
##  $ Mean_sentence_length   : num  9 5.67 11.5 7.5 8 ...
```

We'll now divide the dataset up by candidate so we can make a few comparisons. We will also see how many records we lose if we accept only complete ones to so if transforms are required (we received a NAs warning). Lastly, we will also make sure we have a fair proportion of each candidate.

```r
# number of rows before we remove all imcomplete records
nrow(num.data)
```

```
## [1] 6112
```

```r
# see how many records we lose is we use complete only
num.data <- num.data[complete.cases(num.data), ]
nrow(num.data)
```

```
## [1] 6105
```

Next we will see how the proportions are; we need a comparable number of tweets from each candidate.

```r
# assure we have a reasonable distribution
prop.table(table(data$handle))
```

```
## 
##  HillaryClinton realDonaldTrump 
##       0.5145615       0.4854385
```

```r
# divide the tweets by candidate for inspection
hc.tweets <- num.data[which(num.data$handle == 1), ]
dt.tweets <- num.data[which(num.data$handle == 0), ]
```

Outliers and Imputation?

This shows we will only lose a few records if we use the complete cases function, so we will take that route as opposed to the risks of imputation.

Given that this is a learning assessment and that imputation is required in a lot of cases I'll describe some strategies that would have been useful if they were needed:

Continuous Variables:

Features in the dataset like number of favorites, mean word length, etc. could likely be imputed using the mean. We have copies of the dataset sorted by candidate which could be used for this. In cases like original author or the name of the person that tweet was directed at, using the negative value would likely be advisable; it's clear that in most cases there were no particular targets.

Booleans

The same to Boolean values like that truncation status. This is an uncommon enough occurrence that we can use the negative:

```
# examine for potential impuations strategies
table(num.data$truncated)
```

```
##
##    0    1
## 6069   36
```

Summative statistics and Comparisons

Given that I re-engineered a lot of this dataset (more than half of the features), I'm going to start by looking at a few simple side-by-side comparisons to make sure the features are distinguishable between the candidates before trying to use them in models. We'll do some light correlation testing to make sure what we're looking at is meaningful. We'll do a more formal significance test via regression later on.

Average Sentence Length

```
# average word length
mean(hc.tweets$Mean_sentence_length)
```

```
## [1] 10.95699
```

```
mean(dt.tweets$Mean_sentence_length)
```

```
## [1] 8.793307
```

```
cor(num.data$Mean_sentence_length, num.data$handle)
```

```
## [1] 0.1980196
```

There seems to be a noticeable difference in the mean sentence length, which is a good sign, but the correlation is light. We'll take a look at a few more.

Average word length

```
# effect size
# average word length
mean(hc.tweets$Mean_word_length)
```

```
## [1] 5.007194
```

```
mean(dt.tweets$Mean_word_length)
```

```
## [1] 4.945543
```

```
cor(num.data$Mean_word_length, num.data$handle)
```

```
## [1] 0.02750857
```

The difference here is likely not going to help us much. Let's keep digging.

Retweets

```
# prop table of retweets
prop.table(table(hc.tweets$is_retweet))
```

```
##
##         0         1
## 0.8241478 0.1758522
```

```
prop.table(table(dt.tweets$is_retweet))
```

```
##
##          0          1
## 0.96190155 0.03809845
```

Correlation for re-tweets

```
# cal cor functions
cor(num.data$is_retweet, num.data$handle)
```

```
## [1] 0.2209909
```

It seems that the candidates retweet at noticeably different rates, but again, the correlation is not especially strong.

Sharing Quotes

```
# prop table of retweets
prop.table(table(hc.tweets$is_quote_status))
```

```
##
##          0          1
## 0.95858554 0.04141446
```

```
prop.table(table(dt.tweets$is_quote_status))
```

```
##
##          0          1
## 0.97538773 0.02461227
```

```
cor(num.data$is_quote_status, num.data$handle)
```

```
## [1] 0.04683813
```

There isn't much to go on here, the differences in quotes retweeted seems subtle at best.

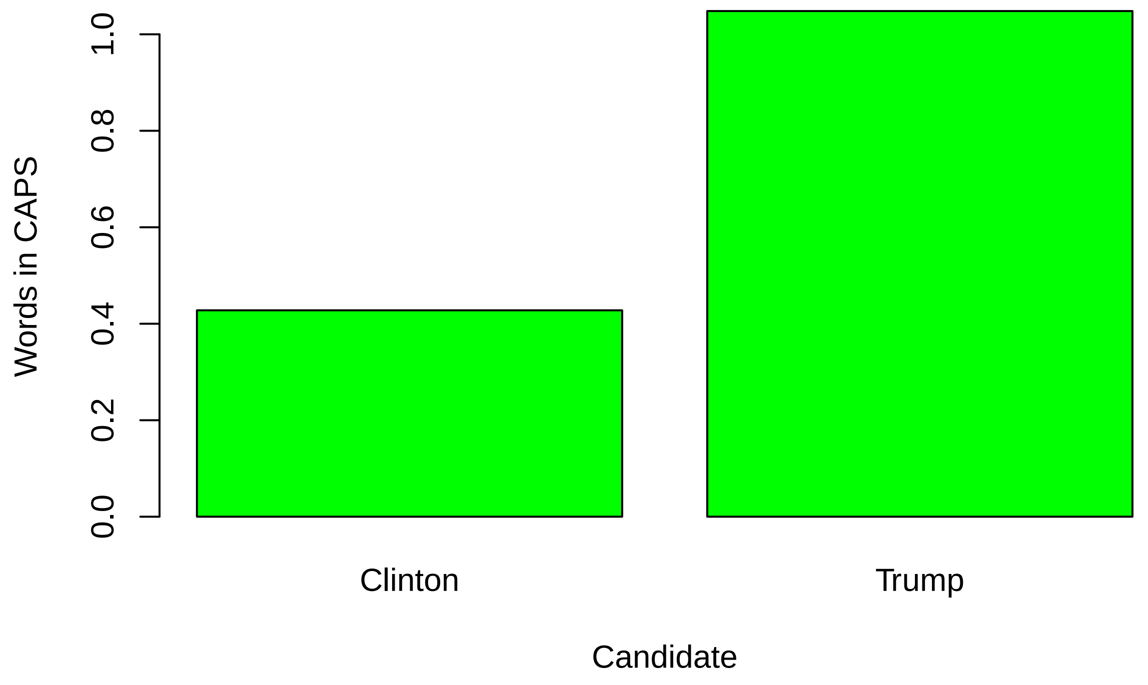
Use of ALLCAPS

```
# prop table of words in caps
hc.mean.caps <- mean(hc.tweets$Words_in_caps)
dt.mean.caps <- mean(dt.tweets$Words_in_caps)
cor(num.data$Words_in_caps, num.data$handle)
```

```
## [1] -0.2285568
```

There is a bit of a skew here. We'll visualize it as it is one of the stronger ones we have.

```
# store and plot caps data
caps.counts <- c(hc.mean.caps, dt.mean.caps)
barplot(caps.counts,
        names = c("Clinton", "Trump"),
        xlab = "Candidate",
        ylab = "Words in CAPS",
        col = "GREEN")
```



Favorite Counts

```
# get data on favorite counts
hc.meanfav <- mean(as.numeric(hc.tweets$favorite_count))
dt.meanfav <- mean(as.numeric(dt.tweets$favorite_count))
cor(num.data$favorite_count, num.data$handle)
```

```
## [1] -0.3265132
```

This our new high water mark, so we will also plot it.

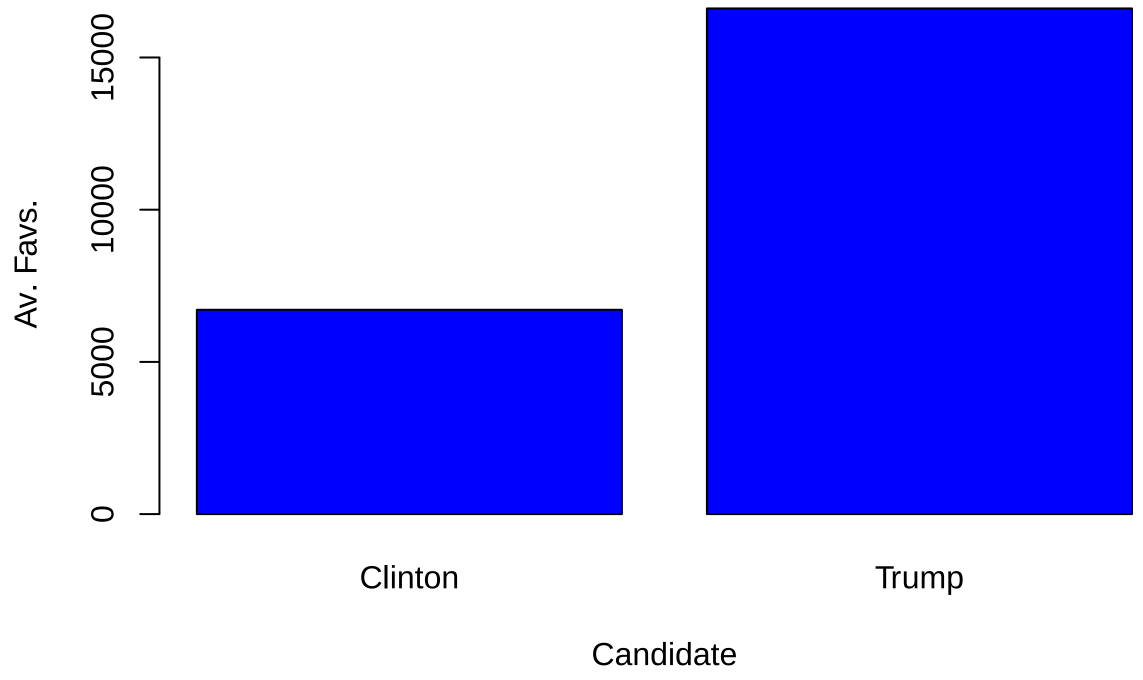
```
# store tweet data in variables
hc.meanfav
```

```
## [1] 6712.199
```

```
dt.meanfav
```

```
## [1] 16611.02
```

```
# store in a vector and plot
fav_counts <- c(hc.meanfav, dt.meanfav)
barplot(fav_counts,
        names = c("Clinton", "Trump"),
        xlab = "Candidate",
        ylab = "Av. Favs.",
        col = "BLUE")
```



The difference here looks significant and the correlation is stronger. In practice, we might need to use an "wide" input of lots of features that are softly correlated to build our model.

After inspecting a handful of differences, I'm confident that the combination of previously existing and engineered features will have enough differences for a model to sort them.

Partition the Datset

Now that we have seen their are patterns in the data for models to identify and converted our numbers to

```
library(caret)
```

```
## Loading required package: lattice

## Loading required package: ggplot2
```

```
# set seed
set.seed(867.5309)

# create index for partitioning
trainIndex <- createDataPartition(num.data$handle, p = 0.5, list = FALSE)

# entries in the index are train, the ones that aren't are test
train <- num.data[trainIndex, ]
train <- train[-3053, ]    # we have an odd numer of samples, some models reject this

# create the inverse partition
test <- num.data[-trainIndex, ]
```

Now that we are equipped with a numeric dataset, we can start applying some preliminary models to see what strategies might get good results for our dataset.

Linear Regression for Prediciton and Exploration.

Firstly, we will try a linear regression model to see if we can get good predictions and hints as to which features might be the most valuable.

```
# we will be using the rmse() function throughout this analysis
library(Metrics)

# establish a linear model
lin <- lm(handle ~., data = train)

# use the model to make predictions
test$lin.pred <- as.numeric(round(predict(lin, data = test)))

# evaluate performace by simple accuracy and rmse
length(which(test$lin.pred == test$handle))/nrow(test) # simple accuracy
```

```
## [1] 0.6081258
```

```
rmse(test$lin.pred, as.numeric(test$handle)) # RMSE
```

```
## [1] 0.6343178
```

These are quite underwhelming; it doesn't bode well for a model if the simple accuracy can be confused for its RMSE. But the reason I included it is because a summary of the model will give us a good idea of which variable are statistically significant. We could, of course, try a few different techniques to improve the regression model, but given the lackluster performance we'll try some other methods first as I suspect some will be more promising out-of-the-box and we can revisit it if needed. In practice, the p-values given in the summary were of as much interest to me as the results.

```
# summary of the LRM
summary(lin)
```

```
##
## Call:
## lm(formula = handle ~ ., data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.21031 -0.23516  0.05339  0.23279  1.73554
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
## (Intercept)              4.440e-01  7.379e-02   6.016 2.00e-09 ***
## is_retweet               1.009e-01  3.904e-02   2.585  0.00978 **
## original_author          3.603e-04  2.318e-04   1.554  0.12024
## in_reply_to_screen_name  6.972e-02  1.119e-02   6.230 5.29e-10 ***
## is_quote_status          8.793e-03  3.491e-02   0.252  0.80117
## lang                    -2.342e-02  9.610e-03  -2.437  0.01488 *
## retweet_count            3.412e-05  3.936e-06   8.668  < 2e-16 ***
## favorite_count          -2.624e-05  1.645e-06 -15.956  < 2e-16 ***
## truncated                1.371e-01  7.711e-02   1.778  0.07553 .
## Words                   -8.611e-03  2.633e-03  -3.270  0.00109 **
## Hashtags                -1.449e-01  1.102e-02 -13.148  < 2e-16 ***
## At_signs                -7.575e-02  8.886e-03  -8.524  < 2e-16 ***
## URLs                     1.485e-01  1.395e-02  10.641  < 2e-16 ***
## Words_in_caps           -4.330e-02  4.949e-03  -8.750  < 2e-16 ***
## Mean_word_length         1.007e-02  6.634e-03   1.518  0.12908
## Periods                  3.866e-02  8.032e-03   4.814 1.56e-06 ***
## Exclamation_points      -2.491e-01  1.269e-02 -19.621  < 2e-16 ***
## Question_marks          -2.535e-02  2.541e-02  -0.997  0.31867
## Colon                   -7.462e-02  1.691e-02  -4.412 1.06e-05 ***
## Semi.colons              6.278e-02  1.972e-01   0.318  0.75019
## Number_of_sentences      9.392e-02  1.904e-02   4.932 8.57e-07 ***
## Mean_sentence_length     2.000e-02  2.926e-03   6.836 9.84e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3406 on 3030 degrees of freedom
## Multiple R-squared:  0.5387, Adjusted R-squared:  0.5355
## F-statistic: 168.5 on 21 and 3030 DF,  p-value: < 2.2e-16
```

```r
test$lin.pred <- NULL
```

This readout provides an excellent readout of features useful in the prediction of the author of the tweet. As noted in the key, the "." and "*" characters dictate the significance to the model. It should be noted that these were considered significant to a model that did not result in useful predictions in practice, but it is useful as a place to start.

Modeling

The k-NN Model

The k-NN model is a simple, venerated approach to numeric data based on euclidean distance, and so makes a good starting point for us. It doesn't technically produce a model; it simply compares new data to periodical observations it has for reference, but can often compete with more sophisticated models and is worth a try for numerical data classification tasks.

```r
# this will require factored outcomes
train$handle <- as.factor(train$handle)
test$handle <- as.factor(test$handle)

# create a caret based knn model
knn <- train(handle ~., data = train, method = "knn")
knn
```

```
## k-Nearest Neighbors
##
## 3052 samples
##   21 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 3052, 3052, 3052, 3052, 3052, 3052, ...
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   5  0.7526533  0.5050136
##   7  0.7640342  0.5280078
##   9  0.7711011  0.5423105
##
```

```
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 9.
```

We can now look at how it will work in context.

```
# apply the model to the testing set
knn.pred <- predict(knn, test)

# see how it did
length(which(knn.pred == test$handle))/nrow(test)
```

```
## [1] 0.7942333
```

```
rmse(as.numeric(knn.pred), as.numeric(test$handle))
```

```
## [1] 0.4536152
```

This is a fairly good start, so we will invest the time to inspect it further and tune it. We will use the trainControl() function to manipulate tuning parameters.

```
# define basic parameters
ctrl <- trainControl(method = "cv", number = 10, selectionFunction = "best")

tuned.knn <- train(handle ~., data = test, method = "knn", trControl = ctrl)
tuned.knn
```

```
## k-Nearest Neighbors
##
## 3052 samples
##   21 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2747, 2746, 2747, 2747, 2747, 2747, ...
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   5  0.7889960  0.5780342
##   7  0.7906354  0.5813194
##   9  0.7912890  0.5827033
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 9.
```

Let’s evaluate the performance of out tuned k-NN model.

```
# apply the model to the testing set
tuned.knn.pred <- predict(tuned.knn, test)

# see how it did
length(which(tuned.knn.pred == test$handle))/nrow(test)
```

```
## [1] 0.8296199
```

```
rmse(as.numeric(tuned.knn.pred), as.numeric(test$handle))
```

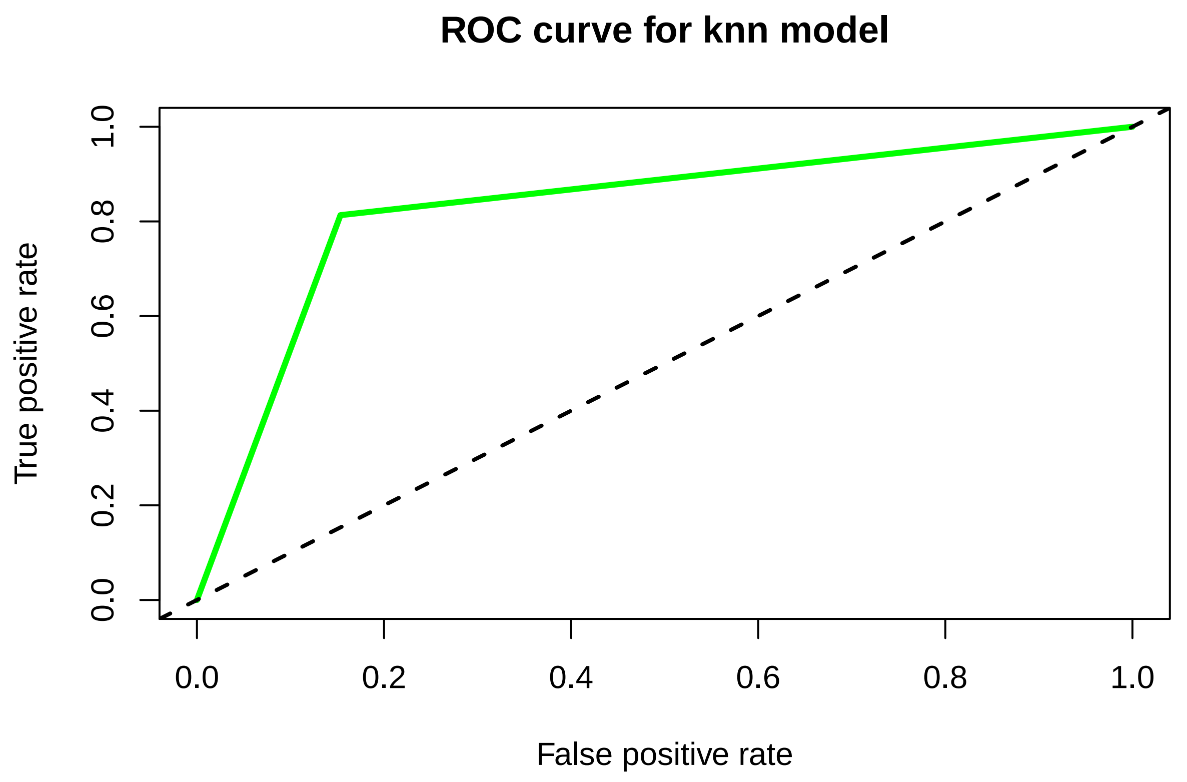
```
## [1] 0.4127712
```

Our tuning pays of with a nudge in the right direction for both metrics. We have some compelling evidence that we are on the right track, so let's take a look at a more detailed evaluation. We will use the ROCR package to measure the area under the curve of our models performance charted by true positives and negative. This entails created and object for the model to get some information from at then plotting it.

```r
library(ROCR)
```

```
## Loading required package: gplots

##
## Attaching package: 'gplots'

## The following object is masked from 'package:stats':
##
##     lowess
```

```r
# create an ROC object to get the ROC
knn.pred.obj <- prediction(predictions = as.numeric(tuned.knn.pred), labels = factor(test$handle))

# evaluate the performace of the object "true positive rates" and "false positive rates"
perf <- performance(knn.pred.obj, measure = "tpr", x.measure = "fpr")

# plot the results
plot(perf, main = "ROC curve for knn model", col = "green", lwd = 3)
abline(a = 0, b = 1, lwd = 2, lty = 2)
```



The further away from the midline our model gets, the better it is. The visual is helpful and suggests we are doing pretty well, but we can quantify for more accurate comparisons. This is the numeric output of the area under the curve.

```r
# quantify results
knn.perf.metrics <- performance(knn.pred.obj, measure = "auc")
unlist(knn.perf.metrics@y.values)
```

```
## [1] 0.8299145
```

The tuned k-NN easily outperformed our mostly-exploratory linear model, and readily benefited from a little tuning, so we will keep it for now.

Nueral Network

Next, we will try a neural network. Neural nets operate as a layer of functions that can activate or remain dormant based on their input, and in turn relay information to another layer of functions. The parameters that we can manipulate are the number of layers and the thresholds for activation.

We can use our network to make predictions.

```r
# make predictions based on our test data
nn.pred <- predict(nnet, test)

# evaluate with out metrics
length(which(nn.pred == test$handle))/nrow(test)
```

```
## [1] 0.8954784
```

```r
rmse(as.numeric(nn.pred), as.numeric(test$handle))
```

```
## [1] 0.323298
```

Not a great start, but let's see if we can coax anything else out of it.

We can now evaluate the new model.

```r
# make predictions
tuned.nn.pred <- predict(tuned.nnet, test)

# evaluate with our metrics
length(which(tuned.nn.pred == test$handle))/nrow(test)
```

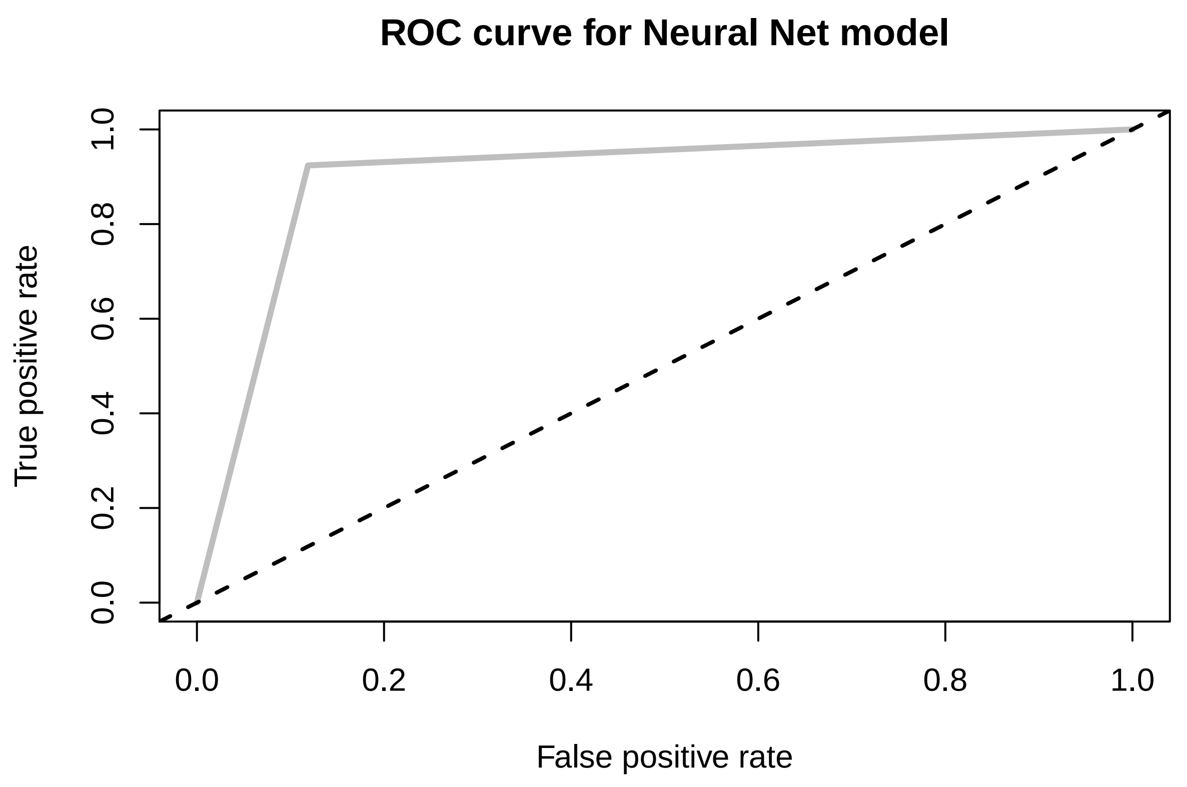
```
## [1] 0.9030144
```

```r
rmse(as.numeric(tuned.nn.pred), as.numeric(test$handle))
```

```
## [1] 0.3114251
```

Again, we see improvements based on a little bit of tuning. To we'll evaluate the predictive power in the same way we did form the k-NN model.

```r
# create an ROC object to get the ROC
tuned.nn.pred.obj <- prediction(predictions = as.numeric(tuned.nn.pred), labels = factor(test$handle))

# evaluate the performace of the object "true positive rates" and "false positive rates"
tuned.nn.perf <- performance(tuned.nn.pred.obj, measure = "tpr", x.measure = "fpr")

# plot the results
plot(tuned.nn.perf, main = "ROC curve for Neural Net model", col = "gray", lwd = 3)
abline(a = 0, b = 1, lwd = 2, lty = 2)
```



Finally, we see the AUC metric for the tuned neural network.

```r
# quantify results
tuned.nn.perf.metrics <- performance(tuned.nn.pred.obj, measure = "auc")
unlist(tuned.nn.perf.metrics@y.values)
```

```
## [1] 0.9026361
```

Rule Learners

Next we will try some rule learners. Rule learners are used for producing human-readable outputs of rules for reaching an outcome, in this case the author The advantage for us her is that we can see very clearly what the model is doing, unlike the "black box" of neural networks. We will use a different library for this, which will make it very easy to output the rules and demonstrate we can operate outside the caret library. In terms of specific implementations, we will use the RIPPER algorithm, which "prunes" branches in the results that aren't helpful to avoid unnecessary computation and improve results.

```r
# RWeka contains the JRip algorithm
library(RWeka)

# use the JRip function, whi
rl <- JRip(handle ~., data = train)

# apply the model to the testing set
rl.pred <- predict(rl, test)

# see how it did
length(which(rl.pred == test$handle))/nrow(test)
```

```
## [1] 0.896789
```

```
rmse(as.numeric(rl.pred), as.numeric(test$handle))
```

```
## [1] 0.3212647
```

Now let's take advantage of the transparency of the model and see the rules.

```
# display rules
rl
```

```
## JRIP rules:
## ===========
##
## (Exclamation_points >= 1) => handle=0 (893.0/59.0)
## (URLs <= 0) and (is_retweet <= 0) and (Mean_sentence_length >= 8.333333) and (favorite_count >= 2173
## (favorite_count >= 8750) and (Periods <= 0) and (Hashtags >= 1) => handle=0 (90.0/5.0)
## (URLs <= 0) and (is_retweet <= 0) and (favorite_count >= 22551) and (At_signs <= 0) => handle=0 (26.
## (At_signs >= 2) and (URLs <= 0) and (is_retweet <= 0) => handle=0 (104.0/1.0)
## (URLs <= 0) and (original_author <= 20) and (Mean_sentence_length >= 10.5) and (favorite_count >= 14
## (favorite_count >= 9000) and (Words_in_caps >= 2) and (Words <= 17) => handle=0 (32.0/4.0)
## (favorite_count >= 7196) and (Periods <= 0) and (Question_marks <= 0) => handle=0 (60.0/11.0)
## (Mean_sentence_length <= 6.333333) and (At_signs >= 2) and (is_retweet <= 0) => handle=0 (15.0/4.0)
## (favorite_count >= 6375) and (Words_in_caps >= 1) and (Mean_word_length >= 5.052632) and (Words <= 1
## (favorite_count >= 4656) and (URLs <= 0) and (Colon >= 1) and (At_signs >= 1) => handle=0 (29.0/5.0)
## (favorite_count >= 4668) and (Hashtags >= 2) => handle=0 (25.0/2.0)
## (Words >= 24) and (favorite_count >= 10061) and (retweet_count <= 4591) and (URLs <= 0) => handle=0
##  => handle=1 (1610.0/140.0)
##
## Number of Rules : 14
```

The Random Forest Model

Random forests are a model that creates a decision tree, much like the rule learners, but generates several and produces a prediction based on a vote to help avoid overfitting the model. Now that we have demonstrated the tuning procedure and seen consistent improvements, we will just start with a tuned model here and change it if needed.

```
# set controls
ctrl <- trainControl(method = "cv", number = 10, selectionFunction = "best")

# create tuned random forest model
tuned.rf <- train(handle ~., data = test, method = "rf", trControl = ctrl)
```

```
## randomForest 4.6-12

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
# apply the model to the testing set
tuned.rf.pred <- predict(tuned.rf, test)
```

```r
# see how it did
length(which(tuned.rf.pred == test$handle))/nrow(test)
```

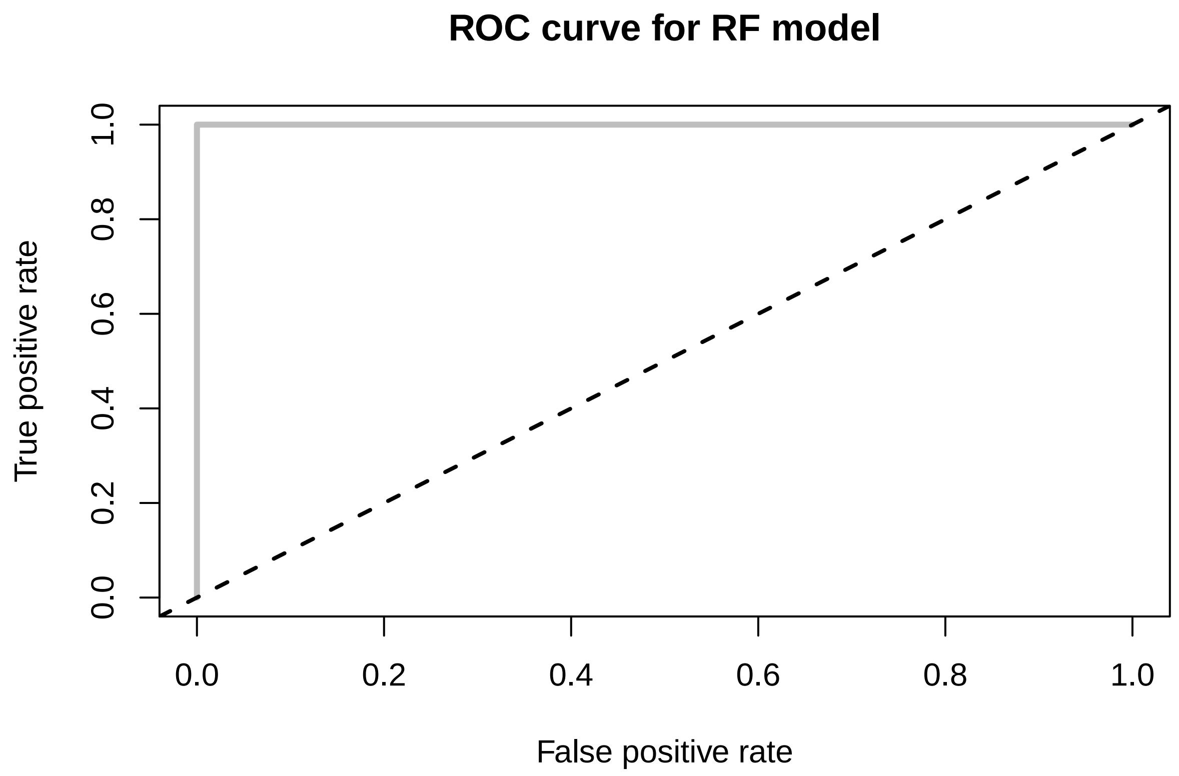
```
## [1] 1
```

```r
rmse(as.numeric(tuned.rf.pred), as.numeric(test$handle))
```

```
## [1] 0
```

Let's see a visual.

```r
# create an ROC object to get the ROC
tuned.rf.pred.obj <- prediction(predictions = as.numeric(tuned.rf.pred), labels = factor(test$handle))

# evaluate the performace of the object "true positive rates" and "false positive rates"
tuned.rf.perf <- performance(tuned.rf.pred.obj, measure = "tpr", x.measure = "fpr")

# plot the results
plot(tuned.rf.perf, main = "ROC curve for RF model", col = "gray", lwd = 3)
abline(a = 0, b = 1, lwd = 2, lty = 2)
```



RF metrics:

```r
# get metrics
tuned.rf.perf.metrics <- performance(tuned.rf.pred.obj, measure = "auc")
unlist(tuned.rf.perf.metrics@y.values)
```

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

Re-evaluating the Dubiously Accurate RF Model

Depending of how the data is partitioned and what I set for a seed value, this model achieves 100% accuracy.

While this is (mostly) a good sign, It could be that we have overfit the model to the point where it will not work for datasets other than this one. Tree-based models are notorious for doing this. That is not a disaster in this context, but can foil a lot of hard work in environments where the models are intended to scale and be applied to new datasets on a regular basis. To see if the model will work on new data, we will create a new, randomized, partition from the dataset and run it again. This will simulate, to some extent, running the model on an unseen dataset.

```r
# we will take another random 50% sample from the dataset and run the RF model on it
trainIndex2 <- createDataPartition(num.data$handle, p = 0.5, list = FALSE)

# create set based on partition index
test2 <- num.data[trainIndex2, ]

# factor the new data set
test2$handle <- as.factor(test2$handle)

# use it for predictions
tuned.rf.pred2 <- predict(tuned.rf, test2)

# evaluate performance
length(which(tuned.rf.pred2 == test2$handle))/nrow(test2)
```

```
## [1] 0.9541435
```

```r
rmse(as.numeric(tuned.rf.pred2), as.numeric(test2$handle))
```

```
## [1] 0.2141414
```

The initial number are good. Let's try the AUC.

```r
# create an ROC object to get the ROC
tuned.rf.pred.obj2 <- prediction(predictions = as.numeric(tuned.rf.pred2), labels = factor(test2$handle

# evaluate the performace of the object "true positive rates" and "false positive rates"
tuned.rf.perf2 <- performance(tuned.rf.pred.obj2, measure = "tpr", x.measure = "fpr")

# plot the results
plot(tuned.rf.perf2, main = "ROC curve for Neural Net model", col = "gray", lwd = 3)
abline(a = 0, b = 1, lwd = 2, lty = 2)
```

[](signatureProject_files/figure-latex/RF "New" Data - AUC-1.pdf)

Let's see some more data on the second pass.

```r
# quantify
tuned.rf.perf.metrics2 <- performance(tuned.rf.pred.obj2, measure = "auc")
unlist(tuned.rf.perf.metrics2@y.values)
```

```
## [1] 0.9538135
```

Given that we don't have an entirely new dataset for me to try this on, using the same model on a newly randomized test set from the original and still performed strongly will have to justify that it has a place in our ensemble.

Avengers Assemble: Creating and Ensemble Model

In practice, most professional data scientists will create ensemble models where different models are given a vote and the final answer is the consensus between them. This can help shore up weaknesses in individual models and prevent overfitting. Sometimes, especially accurate models are given more than one vote, which we will do for our RF model. While it might be tempting to simply use the RF model, given that we have some concerns about overfitting and our other models are also doing well, we'll use the weighted ensemble.

```r
# add a prediction column for each model
test$knn.pred    <- as.numeric(factor(predict(tuned.knn, test)))
test$nnet        <- as.numeric(factor(predict(tuned.nnet, test)))
test$rl          <- as.numeric(factor(predict(rl, test)))

# the highest performer gets two votes, breaks ties
test$rf.pred     <- as.numeric(factor(predict(tuned.rf, test)))
test$rf.pred.v2  <- test$rf.pred

# create a vote column that is empty by default
test$vote <- 0

# create a mode function
vote <- function(x)
{
  # find and tablute max for unique values
  uniq.x <- unique(x)
  uniq.x[which.max(tabulate(match(x, uniq.x)))]
}

# Iterate through the models and cast a vote
for ( i in 1:nrow(test) )
{
  test[i,28] <- vote(c(test[i,23], test[i,24], test[i,25], test[i,26], test[i,27]))
}

# convert to numeric
test$handle <- as.numeric(test$handle)

# accuracy
length(which(test$vote == test$handle))/nrow(test)
```

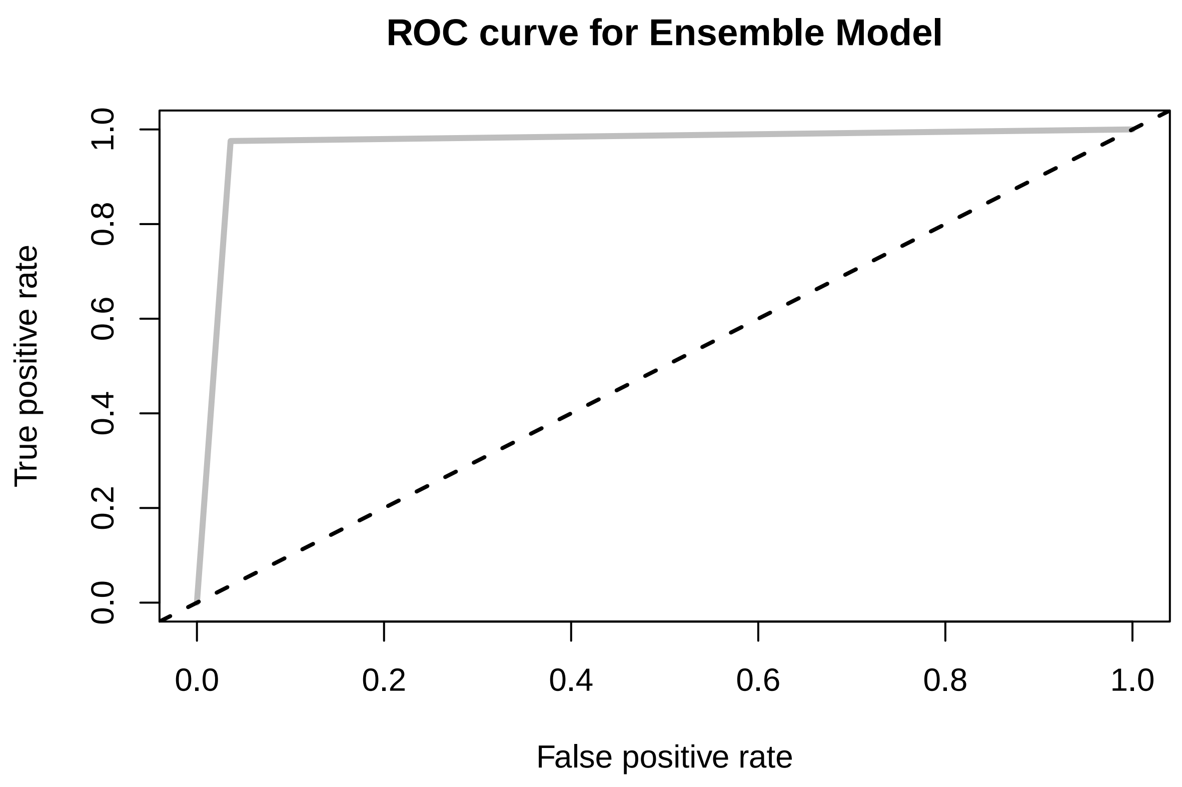
```
## [1] 0.9698558
```

```r
rmse(test$vote, test$handle)
```

```
## [1] 0.1736208
```

Evaluation

```r
# create an ROC object to get the ROC
ensemble.pred.obj <- prediction(predictions = test$vote, labels = factor(test$handle))

# evaluate the performace of the object "true positive rates" and "false positive rates"
ensemble.pred.perf <- performance(ensemble.pred.obj, measure = "tpr", x.measure = "fpr")

# plot the results
plot(ensemble.pred.perf, main = "ROC curve for Ensemble Model", col = "gray", lwd = 3)
abline(a = 0, b = 1, lwd = 2, lty = 2)
```



The AUC metric for the ensemble:

```r
# get AUC mertric
ensemble.perf.metrics <- performance(ensemble.pred.obj, measure = "auc")
unlist(ensemble.perf.metrics@y.values)
```

```
## [1] 0.9697536
```

Deployment

This model seems to be highly effective in our test set. If it were to be implemented, however, the speed with which it could be built would be highly dependent on library availability as it the vote use four different types of models. There would also be infrastructure concerns in terms of harvesting the tweets, but the Python function and script would be viable starts.

APPENDIX A

Support Vector Machine Model

We have plenty of running models that do as well as this or better, so we don't need it for the ensemble, but to demonstrate how transformation and standardization might come into play, we will make an SVM model. This function throws an error if you give it data that are dummy encoded to binary. Using a z score scale will standardize the data in a no binary fashion based on how far they are from the mean. This will allow an SVM to read them. They could be tuned like any of the other caret train() based models implemented above.

```r
# import library
library(kernlab)
```

```
##
## Attaching package: 'kernlab'

## The following object is masked from 'package:ggplot2':
##
##     alpha
```

```r
# make a copy of the data to z score normalize
z.num.data <- num.data

# set seed
set.seed(867.5309)

# create index for partitioning
svm.train.index <- createDataPartition(z.num.data$handle, p = 0.5, list = FALSE)

# entries in the index are train, the ones that aren't are test
svm.train <- z.num.data[svm.train.index, ]
svm.train <- train[-3053, ]   # we have an odd numer of samples, some models reject this

# create the inverse partition
svm.test <- z.num.data[-svm.train.index, ]

# we start at number two because we don't want to normalize the handle
for (i in 2:ncol(z.num.data))
{
  # call the scale funciton on the line
  z.num.data[,i] <- scale(z.num.data[,i])
}

# create SVM predicting handle based on partitions
ctrl <- trainControl(method = "cv", number = 10, selectionFunction = "best")
svm <- ksvm(handle ~., data = svm.train, kernel = "vanilladot", trControl = ctrl)
```

```
##  Setting default kernel parameters
```

```r
# make predictions
svm.test$pred <- predict(svm, test)

# use it to make predictions
svm.test$pred <- as.numeric(factor(svm.test$pred))

svm.test$pred[svm.test$pred == 1 ] <- 0
svm.test$pred[svm.test$pred == 2 ] <- 1

# simple accuracy
length(which(svm.test$pred == svm.test$handle))/nrow(svm.test)
```

```
## [1] 0.8912189
```

```r
# calculate the rmse
RMSE(svm.test$pred, svm.test$handle)
```

```
## [1] 0.3298198
```

str()