ETM 538, Winter '19, Project

Predicting Blood Donations

Jordan Hilton, Andey Nunes, Mengyu Li, Peter Boss

March 13, 2019

Introduction

We are analyzing a data set on blood donations, obtained from a recent Driven Data competition: https://www.drivendata.org/competitions/2/warm-up-predict-blood-donations/. The goal of the competition is to use data on past donations to predict whether a donation was made by a specific patient in the month of March 2007.

If our model successfully predicts donations, we can meet the business goal of creating accurate forecasting for blood donations. While the data in this specific instance comes from a blood donation truck in Taiwan, having a reliable predictive model would help blood donations services, hospitals, and the healthcare system at large—they could plan for storage and transportation of blood products, determine appropriate staffing levels, have more reliable schedules for procedures that require blood, etc. Having a more robust understanding of blood donation practices between countries could also help governments determine and employ best practices for encouraging regular and predictable blood donation from more people.

Our analysis uses 4 models:

1. Naive linear regression
2. K Nearest Neighbors
3. Decision Trees
4. Cross-validation with random forest, which includes another linear regression

Using lowest error rate in the predictions as the criterion, we determined that the strongest model is

Note: # # # ADD THE SPECIFIC ERROR INFO HERE # # #

KNN Error rate: 10.5% at k=3

Initial Data Exploration

```r
training <- read_csv("projectdata.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   X1 = col_double(),
##   `Months since Last Donation` = col_double(),
##   `Number of Donations` = col_double(),
##   `Total Volume Donated (c.c.)` = col_double(),
##   `Months since First Donation` = col_double(),
##   `Made Donation in March 2007` = col_double()
## )
```

```r
glimpse(training)
```

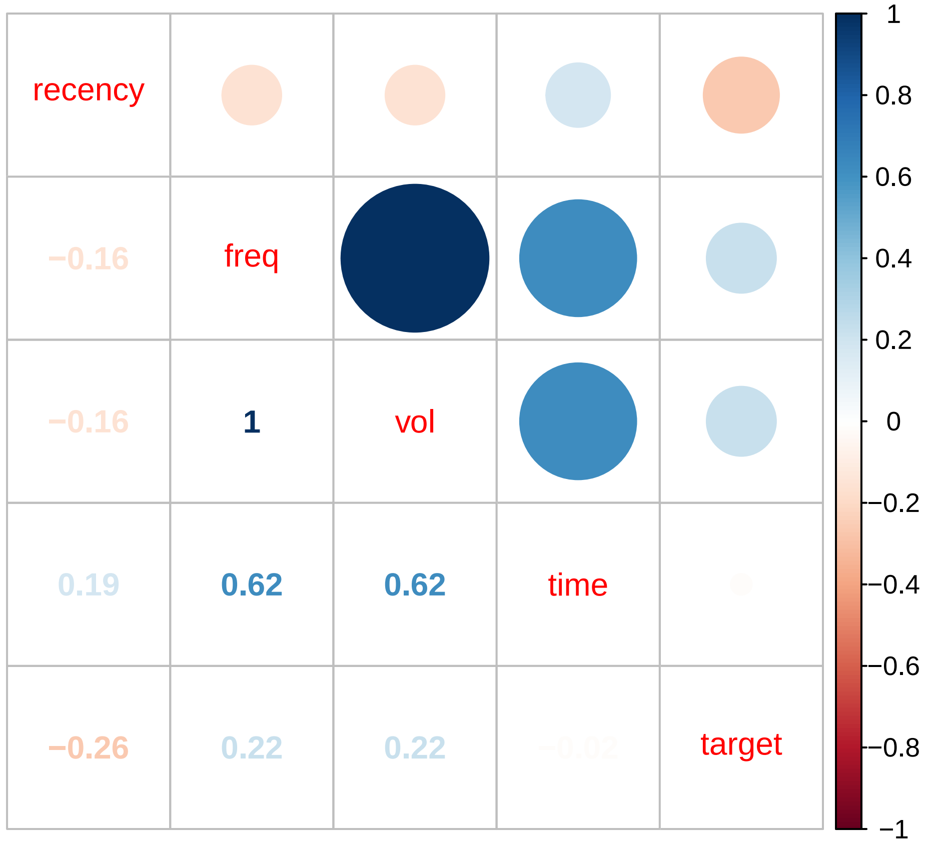
```
## Observations: 576
## Variables: 6
## $ X1                            <dbl> 619, 664, 441, 160, 358, 335, 47...
## $ `Months since Last Donation`  <dbl> 2, 0, 1, 2, 1, 4, 2, 1, 5, 0, 2,...
## $ `Number of Donations`         <dbl> 50, 13, 16, 20, 24, 4, 7, 12, 46...
## $ `Total Volume Donated (c.c.)` <dbl> 12500, 3250, 4000, 5000, 6000, 1...
## $ `Months since First Donation` <dbl> 98, 28, 35, 45, 77, 4, 14, 35, 9...
## $ `Made Donation in March 2007` <dbl> 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1,...
```

The data set comprises 576 observations on each of the variables below. To facilitate easier analysis, we have renamed the variables.

- A count variable, functioning as a unique ID for each person who donated blood. This was not a meaningful component of our analysis, so we ignored it in our models.
- `Months since Last Donation` indicates how many months since the most recent donation event. Renamed as `recency`.
- `Months since First Donation` indicates the total time span in months since the first donation event. Renamed as `time`.
- `Number of Donations`. Renamed as `freq`, for "frequency."
- `Total Volume Donated (c.c.)`. The original data set makes note that this field indicates the monetary measure being used for the business case. Renamed as `vol`, for "volume."
- `Made Donation in March 2007`. The binary variable that we are trying to classify and predict. Renamed as `target`.

```r
# remove ID variable
training <- training[,-1]

# create and apply a vector of names
names(training) <- c("recency", "freq", "vol", "time", "target")
training_mat <- data.matrix(training, rownames.force = T)
```

We examine a plot of the correlations between the variables:



We see there is a perfect correlation between `freq` and `vol`. We note that the `vol` values are exactly 250 times the `freq` values, implying that people are donating exactly 250 CCs at a time. Since `vol` is an exact linear multiple of another variable, we do not include it in our analyses.

Model 1: Naive Linear Regression

Our first model is ordinary linear regression. We chose this model because it's a typical starting point for data analysis, and it's easy to run and interpret. As discussed below, the results did not prove to be useful, but we still consider it a valuable starting point.

We create and inspect the linear model.

```
linearmodel <- lm(target ~ recency + freq + time, data=data)
summary(linearmodel)
```

```
##
## Call:
## lm(formula = target ~ recency + freq + time, data = data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.9731 -0.2838 -0.1654  0.0144  0.9722
##
```

```
## Coefficients:
##              Estimate Std. Error t value            Pr(>|t|)
## (Intercept)  0.311590   0.033564    9.28 < 0.0000000000000002 ***
## recency     -0.009486   0.002239   -4.24         0.000026519 ***
## freq         0.022200   0.004002    5.55         0.000000045 ***
## time        -0.003023   0.000953   -3.17              0.0016 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.403 on 572 degrees of freedom
## Multiple R-squared:  0.117,  Adjusted R-squared:  0.112
## F-statistic: 25.2 on 3 and 572 DF,  p-value: 0.00000000000000261
```

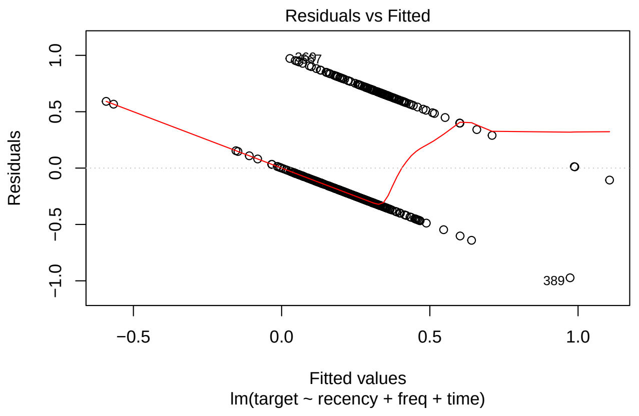
While the model as a whole is statistically significant with a p-value of $2.6 \cdot 10^{-15}$, the low R^2 indicates that our 3 independent variables do a poor job predicting blood donation in the linear model. Each variable is significant in the full linear model, but we formally check that it's appropriate to use each variable.

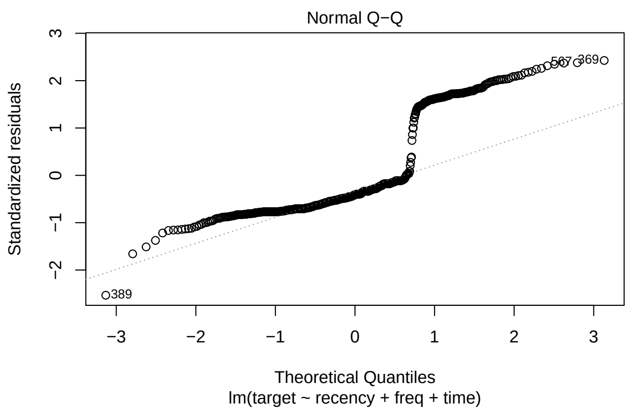
```
step(linearmodel)
```

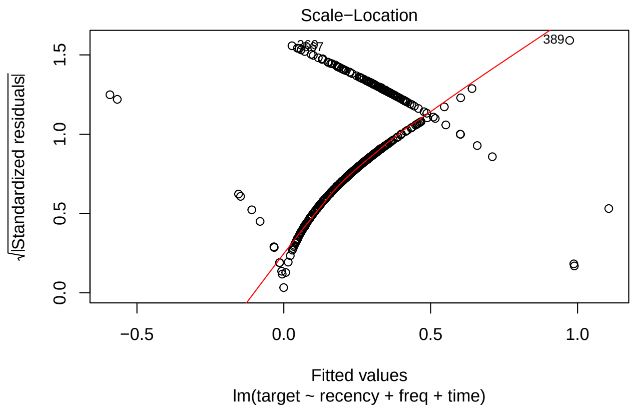
```
## Start:  AIC=-1044
## target ~ recency + freq + time
##
##           Df Sum of Sq  RSS   AIC
## <none>                 92.7 -1044
## - time     1      1.63 94.3 -1036
## - recency  1      2.91 95.6 -1028
## - freq     1      4.99 97.7 -1016

##
## Call:
## lm(formula = target ~ recency + freq + time, data = data)
##
## Coefficients:
## (Intercept)      recency         freq         time
##     0.31159     -0.00949      0.02220     -0.00302
```

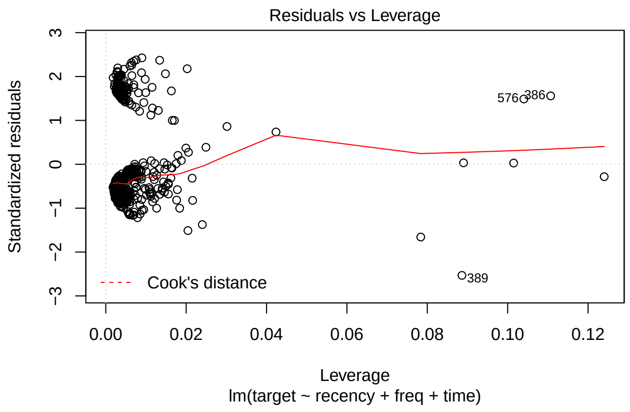
Each variable does contribute sufficiently to a reduction in the sum of the squares of error, and we can't reduce our AIC by eliminating a variable. We examine the standard residual plots to verify the errors are normally distributed.

```
plot(linearmodel)
```









The residual plots all look awful. Most notably, each plot splits the errors into two distinct groups. The most basic checks for normality are violated in each graph, and there are high leverage points. We conclude that the data does not follow a normal distribution, so linear regression is inappropriate for this data set. Much of the difficulty with this analysis probably comes from the fact that the response variable is binary. If we were continuing to apply regression, logistic regression would probably be appropriate here. Instead, we chose types of analysis that were more in line with the material learned in this class.

Note: This model does not include interactions between the input variables, but we also tried a linear regression model including those interactions. The R^2 value was slightly higher but still far too low to be useful (0.16 rather than 0.11), and the errors also exhibited the same problem of lack of normality. We conclude that including variable interactions did not help make linear regression appropriate for this data. The analysis was very similar to what has already been presented, so it is omitted here.

Model 2: K Nearest Neighbors

We begin our Nearest Neighbor analysis by scaling each column by the maximum value. Every entry in the columns is now between 0 and 1. Next we assign weights to each variable.

```
weights <- c(1/74,1/50,1/12500,1/98) ## make a vector of independent variable weights, scaling each by
names(weights) <- c("monthssincelast", "numberdonations", "monthssincefirst")  ##name them so we don't
scaledinstances <- cbind(instances[,1]*weights[1],instances[,2]*weights[2],instances[,3]*weights[3])
## make a scaled version of our instances so that the distances are equivalent
```

We divide our working data into two buckets: one for training, and one for calculating nearest neighbors to assign weights.

```
validationbucket <- scaledinstances[501:576,]
databucket <- scaledinstances[1:500,]
```

We write a function to calculate distances between two different points. We choose the Euclidean distance for two reasons. One, some of our data is in numeric form instead of factors, so Euclidean distance is more appropriate for those values. Two, we will compare our results against a prebuilt nearest-neighbors package, and that package is keyed to Euclidean distance.

```
distance <- function(x,y){
  result <- dist(rbind(x,y), method="euclidean") ## find the manhattan distance between two instances
  return(as.numeric(result)) #the result is weirdly vectorized so we just grab the value out of it
}
```

Here is an example of the distance calculation, working on one of our validation points and one of our data points.

```
databucket[1,]
```

```
## [1] 0.02703 1.00000 0.00784
```

```
validationbucket[1,]
```

```
## [1] 0.18919 0.08000 0.00208
```

```
distance(databucket[1,], validationbucket[1,])
```

```
## [1] 0.9342
```

We create a table to calculate the distances between each of our validation points and each of the training points. Each column, iterated on j, is a validation point. Each row, iterated on i, is a data point.

```
distancesbyrow <- data.frame() #length(data) rows iterated by i for data, length(validation) columns it

for(i in 1:length(databucket[,1])){
  for(j in 1:length(validationbucket[,1])){
    distancesbyrow[i,j]<-distance(databucket[i,],validationbucket[j,])  # calculate the distance betwee
  }
}

head(distancesbyrow[,1:6])
```

```
##       V1      V2     V3     V4      V5     V6
## 1 0.9342 0.96040 0.9539 0.7058 0.96040 0.8555
## 2 0.2611 0.22654 0.2753 0.2244 0.22654 0.2140
## 3 0.2974 0.28292 0.3138 0.2027 0.28292 0.2376
## 4 0.3587 0.36102 0.3767 0.2054 0.36101 0.2896
## 5 0.4369 0.44188 0.4553 0.2582 0.44188 0.3651
## 6 0.1351 0.04005 0.1366 0.2897 0.04006 0.1571
```

We create a table for the results: which point in the data is nearest the ith validation point, and what is the value of that point for our `target` variable (whether or not a donation was made). We were concerned about possible cases where multiple points were nearest but that recommended different `target` outcomes. But no such "ambiguous" values actually appeared in the analysis.

```
distanceresults <- data.frame() #length(validation) rows, one for each test case. first column is minim

for(i in 1:length(validationbucket[,1])){ #traversing across our validation data
  distanceresults[i,1]<-min(distancesbyrow[,i]) #the minimum distance for each column of distancesbyrow
  distanceresults[i,2]<-sum(distanceresults[i,1]==distancesbyrow[,i]) #the number of occurences of this
  distanceresults[i,3]<-data[which.min(distancesbyrow[,i]),6] ## pulls the "donation" value for the firs
  for(j in 1:length(validationbucket[,1])){ ## traversing across the distances for one possible case
    if(distancesbyrow[j,i]==distanceresults[i,1] & data[j,6]!=distanceresults[i,3]){
     distanceresults[i,3]<-"amb"
    } ## if the distance for this case is minimum AND the playvalue is not equal to the first playvalue
  }
}

colnames(distanceresults)<-c("minimumdistance","occurrences", "donation")
head(distanceresults)
```

```
##   minimumdistance occurrences donation
## 1         0.00000           1        0
## 2         0.00008           1        0
## 3         0.00016           1        1
## 4         0.02703            1        0
## 5         0.00016            1        0
## 6         0.00000            1        1
```

```
sum(distanceresults$occurrences==1) # the number of possible classes that have a unique closest neighbo
```

```
## [1] 54
```

```
sum(distanceresults$donation=="amb") # the number of unambiguous results
```

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

We compare the results of our training against the real donation values, and we calculate the error rate.

```
predictedresults<-distanceresults[,3]
originaldata <- data[501:576,6]
correctanswers <- sum(predictedresults==originaldata)
errorrate_knn <- 1 - correctanswers/length(originaldata)
errorrate_knn
```

```
## [1] 0.1974
```

The error rate is 19.7%.

Ordinarily, our next step would be to adjust the weight vector and re-run the analysis, with a goal of minimizing the error rate. In this instance, though, there are robust existing R packages that we can use to compare results. We employ the `class` package to see another approach to the problem.

```
train<-databucket
test<-validationbucket
cl<-data[1:500,6]
libraryresults<-knn(train, test, cl, k=4, prob=TRUE)
libraryresults
```

```
##  [1] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [36] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [71] 0 0 0 0 0 0
## attr(,"prob")
##  [1] 0.5000 1.0000 0.7500 1.0000 1.0000 0.7500 0.7500 1.0000 0.7500 1.0000
## [11] 1.0000 0.7500 0.8333 0.8333 0.8333 1.0000 1.0000 1.0000 1.0000 1.0000
## [21] 1.0000 1.0000 0.7500 1.0000 1.0000 1.0000 0.7500 1.0000 0.7500 1.0000
## [31] 0.5000 0.7500 1.0000 1.0000 1.0000 1.0000 0.7500 0.7500 0.7500 0.7500
## [41] 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
## [51] 0.7778 1.0000 0.7500 0.7500 0.7500 1.0000 0.7500 0.7500 0.9091 0.9091
## [61] 0.7500 0.9091 0.9091 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
## [71] 1.0000 1.0000 1.0000 1.0000 0.8889 0.9000
## Levels: 0 1
```

We've chose $k = 3$ nearest neighbors, and the library returns (in the first result) the winning classification for each test row, and also the probability of that classification based on the nearest neighbors. We calculate the error rate:

```
predictedresults<-as.numeric(libraryresults)-1
originaldata<-data[501:576,6]
correctanswers<-sum(predictedresults==originaldata)
errorrate<-1-correctanswers/length(originaldata)
errorrate
```

```
## [1] 0.09211
```

Using the same code, we checked the error rate considering the 1, 2, 3, 4 nearest neighbors:

```r
k<-c(1,2,3,4)
error<-c("18.4%", "13.1%", "10.5%", "9.2%" )
pander(cbind(k,error))
```

k	error
1	18.4%
2	13.1%
3	10.5%
4	9.2%

We want to avoid overfitting, so we choose $k = 3$ nearest neighbors for the analysis. We note that the built-in package has slightly smaller error than our manual work even at $k = 1$, and that at $k = 3$ the error rate is half of what it was in our original work. Note also that the built-in package decides points with tied votes in their nearest neighbors at random, so you might get slightly different results when rerunning this code.

The data contest (from Data Driven) includes a specific set of test data. Here are our model's predictions against those data points.

```r
testdata<-read.csv("project test data.csv")
testdata<-testdata[-c(1,4,6)]
testdata<-cbind(testdata[,1]*weights[1],testdata[,2]*weights[2],testdata[,3]*weights[3])
train<-scaledinstances
test<-testdata
cl<-data[,6]
libraryfullresults<-knn(train, test, cl, k=, prob=TRUE)
head(libraryfullresults)
```

```
## [1] 1 0 0 0 0 1
## Levels: 0 1
```

Model 3: Decision Tree

In this section, we first check the strcution of data frame which consist of 576 observation of 6 variables.

```r
dfRawData <- read.csv('projectdata.csv')
str(dfRawData)
```

```
## 'data.frame':    576 obs. of  6 variables:
##  $ X                          : int  619 664 441 160 358 335 47 164 736 436 ...
##  $ Months.since.Last.Donation : int  2 0 1 2 1 4 2 1 5 0 ...
##  $ Number.of.Donations        : int  50 13 16 20 24 4 7 12 46 3 ...
##  $ Total.Volume.Donated..c.c..: int  12500 3250 4000 5000 6000 1000 1750 3000 11500 750 ...
##  $ Months.since.First.Donation: int  98 28 35 45 77 4 14 35 98 4 ...
##  $ Made.Donation.in.March.2007: int  1 1 1 1 0 0 1 0 1 0 ...
```

Pre-process data

In this section, data is cleared and prepared ready for processing.

```
dfModelData <- dfRawData[-1] # Exclude the first column
dfModelData <- na.omit(dfModelData) # remove any NAs
colnames(dfModelData) <- c('MonthsSinceLastDonation', 'NumberOfDonations', 'TotalVolumeDonatedCC', 'Mon
dfModelData$MadeDonationInMarch2007 <- as.factor(dfModelData$MadeDonationInMarch2007)
```

Split data into train and test data

In this section, we create the train data and test date. By using p=0.8, it means the data split should be done in 80:20 ratio. And before training decision tree classifier, set.seed().

```
# Split the data into training and test set
set.seed(123)
# 80% of data are selected as train data
trnSamples <- dfModelData$MadeDonationInMarch2007 %>%
  createDataPartition(p = 0.8, list = FALSE)
trnData <- dfModelData[trnSamples, ]
testData <- dfModelData[-trnSamples, ]
```

Trained decision tree classifier result.

We check the result of our train()method by a print dtree_fit variable. It's showing us the accuracy metrics or different values of cp as the following. We also used cross validation to divide it into 10 folds.

```
control <- trainControl(method = 'repeatedcv', number = 10, repeats = 5)
# Train the model
model_DT <- train(MadeDonationInMarch2007 ~., data = trnData, method = 'rpart',  parms = list(split = "
# Estimate variable importantce
model_DT
```

```
## CART
##
## 462 samples
##   4 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 462, 462, 462, 462, 462, 462, ...
## Resampling results across tuning parameters:
##
##   cp        Accuracy  Kappa
##   0.000000  0.7656    0.3270
##   0.008675  0.7859    0.3631
##   0.017351  0.7922    0.3758
##   0.026026  0.7937    0.3615
##   0.034701  0.7943    0.3587
##   0.043377  0.7967    0.3702
##   0.052052  0.7914    0.3286
##   0.060727  0.7829    0.2673
##   0.069403  0.7722    0.2020
##   0.078078  0.7668    0.1335
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.04338.
```

From above, the cp= 0.04337671. We are ready to predict classes for our test set, and use predict()method.

Confusion Matrix

The following result shows that the classifier with thre criterion as information gain is giving 0.7193 of accuracy for the test set. The prediction accuracy of 0.7193, which means that the percentage that the actual/prefernce result is 0, and the prediction result is 0; the actual is 1, and that prediction result is 1 is 0.7193. The error rate = 1- accuracy = 0.2807, kappa =0.1817, which means the kappa coefficient is slight.

```
test_pred <- predict(model_DT, newdata = testData)
confusionMatrix(test_pred, testData$MadeDonationInMarch2007 )  #check accuracy
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 73 18
##          1 14  9
##
##                Accuracy : 0.719
##                  95% CI : (0.627, 0.799)
##     No Information Rate : 0.763
##     P-Value [Acc > NIR] : 0.886
##
##                   Kappa : 0.182
##  Mcnemar's Test P-Value : 0.596
##
##             Sensitivity : 0.839
##             Specificity : 0.333
##          Pos Pred Value : 0.802
##          Neg Pred Value : 0.391
##              Prevalence : 0.763
##          Detection Rate : 0.640
##    Detection Prevalence : 0.798
##       Balanced Accuracy : 0.586
##
##        'Positive' Class : 0
##
```

Model 4: Cross Validation Folds & Splits

The workflow and code used here is adapted from the *Machine Learning in the Tidyverse* course by Dmitriy Gorenshteyn (DataCamp 2019). The approach involves using list-columns to store list objects (such as model output) in a tibble, which is a special R data frame. These lists can then be iterated over with specialized `map` functions from the `purrr` package to calculate and extract information.

Using the training data, we will create a 5-fold cross-validation split tibble. This will set us up to itertively generate multiple models on our new train/validation subsets.

```
# using the rsample library
cv_split <- vfold_cv(training, v = 5)
print(cv_split)# uncomment the front of this line if you want to preview
```

```
## #  5-fold cross-validation
## # A tibble: 5 x 2
##   splits            id
##   <list>            <chr>
## 1 <split [460/116]> Fold1
## 2 <split [461/115]> Fold2
## 3 <split [461/115]> Fold3
## 4 <split [461/115]> Fold4
## 5 <split [461/115]> Fold5
```

The `cv_split` object has five rows and two columns. The first column, `splits`, is a list column containing the training and validation data. The second column is a character vector containing the fold id generated by the `vfold_cv()` function. We can iterate over the `splits` column and extract the train and validation columns

```
cv_data <- cv_split %>%
   mutate(train = map(splits, ~training(.x)),
          validate = map(splits, ~testing(.x)))
glimpse(cv_data)
```

```
## Observations: 5
## Variables: 4
## $ splits   <list> [<rsplit[460 x 116 x 576 x 5]>, <rsplit[461 x 115 x ...
## $ id       <chr> "Fold1", "Fold2", "Fold3", "Fold4", "Fold5"
## $ train    <list> [<tbl_df[460 x 5]>, <tbl_df[461 x 5]>, <tbl_df[461 x...
## $ validate <list> [<tbl_df[116 x 5]>, <tbl_df[115 x 5]>, <tbl_df[115 x...
```

We've just created two new list columns containing the data that we can now train and validate models over. In the next two sections I will prepare and evaluate a logistic regression model and a random forests model on these cross-validation sets.

Model preparation

logistic regression model

```
cv_models_glm <- cv_data %>%
   mutate(glm_model = map(train, ~glm(formula = target~., data = .x, family = "binomial"))
          )
```

random forest

```
# set forest parameter
n_trees <- 500
# Build a random forest model for each fold
cv_models_rf <- cv_data %>%
  mutate(rf_model = map(train, ~ranger(formula = target~., data = .x,
                                       num.trees = n_trees,))
```

```
        )
```

Now we have two tibbles, one containing a set of logistic models and the cross-validation data, and the other containing the set of random forest models. We can now iterate over these tibbles to extract the actual values and generate prediction comparisons.

Model evaluation

The *cross-validation model evaluation process* follows the same general set of steps iterated over each fold: 1. extract the actual target values from the validation set 2. use the models to make target predictions that will be compared to the actual target 3. for each fold, calculate the following:

- Accuracy where
- Mean Absolute Error (`MAE`) where

$$MAE = \frac{\sum_{i=1}^{n} |Actual_i - Predicted_i|}{n}$$

4. Take the average over all MAE values to determine which model performs best on these sets of training & validation splits

In the following subsections, the eval code chunk follows a similar pattern:

- the *actual* and *predicted* values are extracted from the validation sets into a `cv_prep_`

Lets see how random forests compared.

logistic regression model

```
# extract actual values
cv_prep_glm <- cv_models_glm %>%
   mutate(validate_actual = map(validate, ~.x$target),
          validate_predicted = map2(.x = glm_model, .y = validate,
                                    ~predict(.x, .y, type = "response") > 0.5))
```

```
# the function mae() is from library(Metrics)
# Calculate the mean absolute error for each validate fold
cv_eval_glm <- cv_prep_glm %>%
  mutate(validate_mae = map2_dbl(.x = validate_actual,
                                 .y = validate_predicted,
                                 ~mae(actual = .x, predicted = .y)),
         glm_accuracy = map2_dbl(.x = validate_actual,
                                 .y = validate_predicted,
                                 ~accuracy(actual = .x, predicted = .y)),
         glm_precision = map2_dbl(.x = validate_actual,
                                 .y = validate_predicted,
                                 ~precision(actual = .x, predicted = .y)),
         glm_recall = map2_dbl(.x = validate_actual,
                                 .y = validate_predicted,
                                 ~recall(actual = .x, predicted = .y)))

# Print the validate_mae column
glm_results <- tibble(foldID = 1:5,
                      MAE = cv_eval_glm$validate_mae,
                      accuracy = cv_eval_glm$glm_accuracy,
                      precision = cv_eval_glm$glm_precision,
                      recall = cv_eval_glm$glm_recall
                      )

kable(glm_results)
```

foldID	MAE	accuracy	precision	recall
1	0.2500	0.7500	0.5000	0.0690
2	0.2696	0.7304	0.0000	0.0000
3	0.2696	0.7304	0.8000	0.1176
4	0.1304	0.8696	0.7143	0.2778
5	0.2348	0.7652	0.5000	0.1852

The average mean absolute error across all logistic regression models was 0.2309. Looks like logistic regression has a similar error rate as the linear regression model. Let's see how the random forest model did.

random forest

```
# Generate predictions using the random forest model
cv_prep_rf <- cv_models_rf %>%
  mutate(validate_actual = map(validate, ~.x$target),
         validate_predicted = map2(.x = rf_model, .y = validate, ~predict(.x, .y, type = "response")$pre
```

```
# Calculate validate MAE for each fold
cv_eval_rf <- cv_prep_rf %>%
  mutate(validate_mae = map2_dbl(validate_actual, validate_predicted, ~mae(actual = .x, predicted = .y))

# Print the validate_mae column
cv_eval_rf$validate_mae
```

```
##      1      2      3      4      5
## 0.2155 0.2348 0.2609 0.1739 0.2609
```

```
# Calculate the mean of validate_mae column
```

The mean absolute error rate was 0.2292. We can tune some model parameters to see if this can be improved.

Tune hyper-parameters

The `mtry` parameter selects a random subset of variables to model. Since we only have 5 variables to begin with, we are somewhat limited in the range of this parameter, but lets try it anyway. We start by creating another tibble containing a set of the 5-fold cross-validation data for each `mtry` value. Here the values will range between 1 and 3 so we will get 15 sets of training|validation splits respective of the parameter setting. We will iterate over this tibble to create 15 random forest models and continue to utilize the list-column workflow to predict on validation splits and evaluate all of these models.

```
# Prepare for tuning cross validation folds by varying mtry
cv_tune <- cv_data %>%
   crossing(mtry = 1:3)

# Build a model for each fold & mtry combination
cv_model_tunerf <- cv_tune %>%
  mutate(rf_model =
            map2(.x = train, .y = mtry,
                 ~ranger(formula = target~., data = .x,
                         mtry = .y, num.trees = n_trees)))

glimpse(cv_model_tunerf)
```

```
## Observations: 15
## Variables: 6
## $ splits   <list> [<rsplit[460 x 116 x 576 x 5]>, <rsplit[460 x 116 x ...
## $ id       <chr> "Fold1", "Fold1", "Fold1", "Fold2", "Fold2", "Fold2",...
## $ train    <list> [<tbl_df[460 x 5]>, <tbl_df[460 x 5]>, <tbl_df[460 x...
## $ validate <list> [<tbl_df[116 x 5]>, <tbl_df[116 x 5]>, <tbl_df[116 x...
## $ mtry     <int> 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3
## $ rf_model <list> [<0.50154, 0.27705, 0.52802, 0.83531, 0.43755, 0.646...
```

Now lets evaluate the models using the mean absolute error, accuracy, precision, and recall metrics that we used to evaluate the logistic regression model.

```
# Generate validate predictions for each model
cv_prep_tunerf <- cv_model_tunerf %>%
  mutate(rf_validate_actual = map(validate, ~.x$target),
     rf_validate_predicted = map2(.x = rf_model, .y = validate, ~predict(.x, .y, type = "response")$pred

# Calculate validate MAE for each fold and mtry combination
cv_eval_tunerf <- cv_prep_tunerf %>%
  mutate(rf_validate_mae = map2_dbl(.x = rf_validate_actual, .y = rf_validate_predicted, ~mae(actual =
         rf_accuracy = map2_dbl(.x = rf_validate_actual,
                                .y = rf_validate_predicted,
                                ~accuracy(actual = .x, predicted = .y)),
         rf_precision = map2_dbl(.x = rf_validate_actual,
                                .y = rf_validate_predicted,
                                ~precision(actual = .x, predicted = .y)),
         rf_recall = map2_dbl(.x = rf_validate_actual,
                                .y = rf_validate_predicted,
                                ~recall(actual = .x, predicted = .y)))
```

```
# Calculate the mean validate_mae for each mtry used
rf_eval_summary <- cv_eval_tunerf %>%
   group_by(mtry) %>%
   summarise(rf_mean_mae = mean(rf_validate_mae),
            rf_mean_accuracy = mean(rf_accuracy),
            rf_mean_precision = mean(rf_precision),
            rf_mean_recall = mean(rf_recall))
kable(rf_eval_summary)
```

mtry	rf_mean_mae	rf_mean_accuracy	rf_mean_precision	rf_mean_recall
1	0.2257	0.7743	0.5641	0.3319
2	0.2309	0.7691	0.5467	0.3780
3	0.2275	0.7725	0.5502	0.3992

It looks like we get lowest error rate using mtry 2, but best recall and accuracy on the mtry 3. Still the mean error rate for any model is much higher than other methods previously, therefore, it will not be used to generate predictions for the DrivenData contest. If we did want to generate the vector of predictions for the test data, here is how we would go about doing that in four steps (though here they are not evaluated because we are using the knn predictors):

1. Select random forest model with the lowest average mae (or other metric).

```
best <- filter(cv_eval_tunerf, rf_validate_mae == min(cv_eval_tunerf$rf_validate_mae))
kable(tibble(id = best$id, mtry = best$mtry))
```

id	mtry
Fold4	1

2. Use these parameters to train our best rf model.

```
# Build the model using all training data and the best performing parameter
best_model <- ranger(formula = target~., data = training,
                     mtry = best$mtry, num.trees = n_trees)
```

3. Load test data
4. Generate prediction vector

Conclusion

The knn model had the best error rate.

References

Data is courtesy of Yeh, I-Cheng via the UCI Machine Learning repository (https://archive.ics.uci.edu/ml/datasets/Blood+Transfusion+Service+Center)

https://archive.ics.uci.edu/ml/machine-learning-databases/blood-transfusion/transfusion.names

Code examples were borrowed from DataCamp course material presented by Dmitriy Gorenshteyn, "Machine Learning in the Tidyverse" https://www.datacamp.com/courses/machine-learning-in-the-tidyverse

Appendix