

Homework 04: Orange's Customer Relationships: Churn Prediction

Chicago Booth ML Team

Note: In order to illustrate the best practices, this script utilizes the popular [caret](#) package, which wraps around underlying algorithms such as `randomForest` and `GBM` with a consistent interface. We also illustrate the use of **multi-core parallel computation** to speed up computer run-time.

Note: This script takes a **long time** to run completely.

This KDD challenge has 3 predictive tasks, addressing **churn**, **appetency** and **upselling**. We'll tackle **churn** in this script. Doing **appetency** and **upselling** would be very similar.

Load Libraries & Modules; Set Randomizer Seed

```
library(caret)
library(data.table)
library(doParallel)

# load modules from the common HelpR repo
help_repo_raw_url <- 'https://raw.githubusercontent.com/ChicagoBoothML/HelpR/master'
source(file.path(help_repo_raw_url, 'EvaluationMetrics.R'))

# set randomizer's seed
set.seed(99) # Gretzky was #99
```

Parallel Computation Setup

Let's set up a parallel computing infrastructure (thanks to the excellent `doParallel` package by Microsoft subsidiary **Revolution Analytics**) to allow more efficient computation in the rest of this exercise:

```
cl <- makeCluster(detectCores() - 2) # create a compute cluster using all CPU cores but 2
clusterEvalQ(cl, library(foreach))
registerDoParallel(cl) # register this cluster
```

We have set up a compute cluster with **6** worker nodes for computing.

Data Import & Cleaning

```
# download data and read data into data.table format

# *****
# NOTE: the following path is specific to my computer
# You need to change it to a relevant folder on your computer containing the Orange data
```

```

data_folder_path <- '/Cloud/Box Sync/Repos/DATA/DATA__KDDCup2009_OrangeCustomerRelationship'

# *****

# Common NAs:
na_strings <- c(
  '',
  'na', 'n.a', 'n.a.',
  'nan', 'n.a.n', 'n.a.n.',
  'NA', 'N.A', 'N.A.',
  'NaN', 'N.a.N', 'N.a.N.',
  'NAN', 'N.A.N', 'N.A.N.',
  'nil', 'Nil', 'NIL',
  'null', 'Null', 'NULL')

X <- as.data.table(read.table(
  file.path(data_folder_path, 'orange_small_train.data.gz'),
  header=TRUE, sep='\t', stringsAsFactors=TRUE, na.strings=na_strings))

nb_input_features <- ncol(X)
input_feature_names <- names(X)
nb_samples <- nrow(X)

churn <- factor(
  read.table(
    file.path(data_folder_path, 'orange_small_train_churn.labels.txt'),
    header=FALSE, sep='\t')[[1]],
  levels=c(-1, 1),
  labels=c('no', 'yes'))

```

In total, there are **50,000** samples of **230** possible *anonymized* input features that can be used to predict the outcome of interest **churn**.

Let's split the data into a Training set and a Test set:

```

train_proportion <- .4
train_indices <- createDataPartition(
  y=churn,
  p=train_proportion,
  list=FALSE)

X_train <- X[train_indices, ]
X_test <- X[-train_indices, ]
churn_train <- churn[train_indices]
churn_test <- churn[-train_indices]

nb_test_samples <- length(churn_test)

```

Let's also split out a Validation set for the purpose of estimating OOS performance of trained models before testing:

```

valid_proportion <- .25
valid_indices <- createDataPartition(
  y=churn_train,
  p=valid_proportion,
  list=FALSE)

X_valid <- X_train[valid_indices, ]
X_train <- X_train[-valid_indices, ]

```

```

churn_valid <- churn_train[valid_indices]
churn_train <- churn_train[-valid_indices]

nb_train_samples <- length(churn_train)
nb_valid_samples <- length(churn_valid)

```

The numbers of samples in the Training, Validation and Test sets are **15000**, **5001** and **29999** respectively. Just to sanity-check that the data sets have been split representatively by **caret**: the **churn** incidences in the Training, Validation and Test sets are **7.34**, **7.36** and **7.34** respectively.

Getting Rid of Input Features x 's with Too Many Missing Values

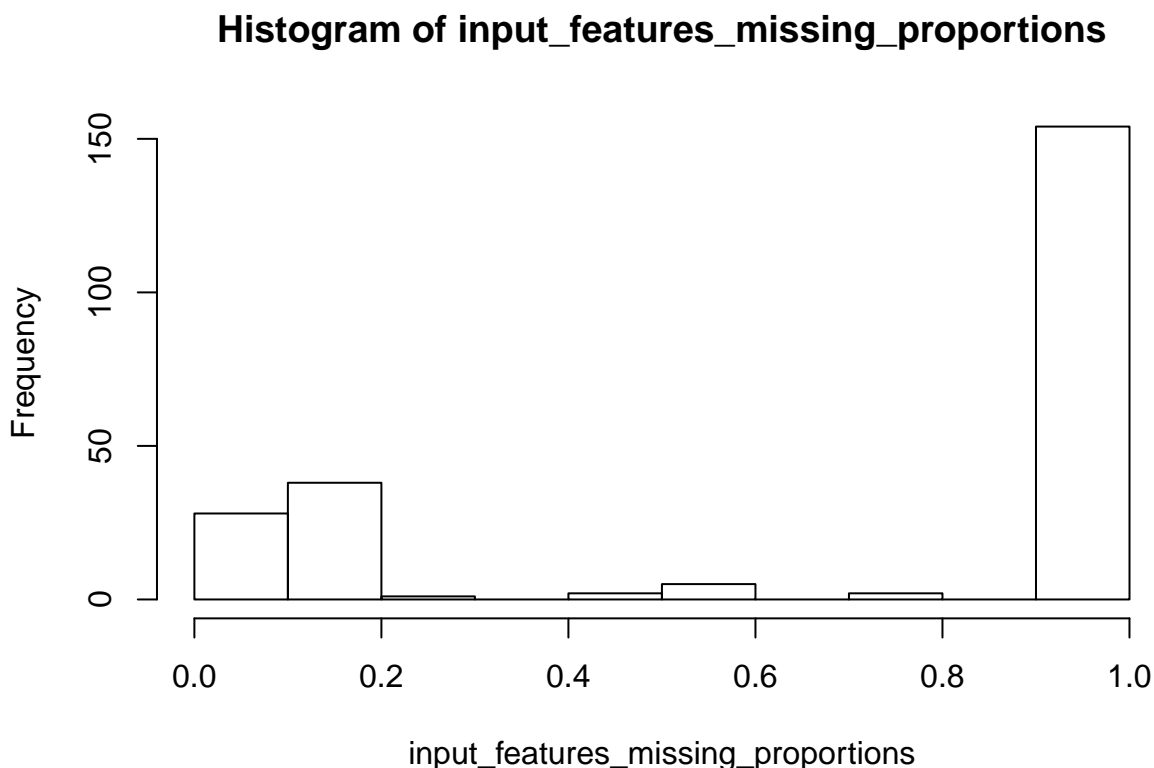
First of all, let's look at the proportions of missing values per input feature column x :

```

input_features_missing_proportions <-
  sapply(X_train, function(col) sum(is.na(col))) / nb_train_samples

hist(input_features_missing_proportions)

```



We can see that there are an awful lot of features with all missing data!! We'll kick them out, for sure. Also, there are a small handful of features that have over 20% missing data; since those are few and we are unlikely to miss out too many signals by removing them, let's not mess around with them either. In sum, we'll remove all features that have over 20% missing value:

```

input_feature_names <-
  input_feature_names[input_features_missing_proportions <= .2]

nb_input_features <- length(input_feature_names)

X_train <- X_train[ , input_feature_names, with=FALSE]

```

We're left with the following **66** input features x 's:

```
input_feature_names
```

```
## [1] "Var6" "Var7" "Var13" "Var21" "Var22" "Var24" "Var25"  
## [8] "Var28" "Var35" "Var38" "Var44" "Var57" "Var65" "Var73"  
## [15] "Var74" "Var76" "Var78" "Var81" "Var83" "Var85" "Var109"  
## [22] "Var112" "Var113" "Var119" "Var123" "Var125" "Var132" "Var133"  
## [29] "Var134" "Var140" "Var143" "Var144" "Var149" "Var153" "Var160"  
## [36] "Var163" "Var173" "Var181" "Var192" "Var193" "Var195" "Var196"  
## [43] "Var197" "Var198" "Var199" "Var202" "Var203" "Var204" "Var205"  
## [50] "Var206" "Var207" "Var208" "Var210" "Var211" "Var212" "Var216"  
## [57] "Var217" "Var218" "Var219" "Var220" "Var221" "Var222" "Var223"  
## [64] "Var226" "Var227" "Var228"
```

The classes of these remaining x 's are:

```
input_feature_classes <- factor(sapply(X_train, class))
```

```
input_feature_classes
```

```
## Var6 Var7 Var13 Var21 Var22 Var24 Var25 Var28 Var35  
## integer integer integer integer integer integer integer numeric integer  
## Var38 Var44 Var57 Var65 Var73 Var74 Var76 Var78 Var81  
## integer integer numeric integer integer integer integer integer numeric  
## Var83 Var85 Var109 Var112 Var113 Var119 Var123 Var125 Var132  
## integer integer integer integer numeric integer integer integer integer  
## Var133 Var134 Var140 Var143 Var144 Var149 Var153 Var160 Var163  
## integer integer integer integer integer integer integer integer integer  
## Var173 Var181 Var192 Var193 Var195 Var196 Var197 Var198 Var199  
## integer integer factor factor factor factor factor factor factor  
## Var202 Var203 Var204 Var205 Var206 Var207 Var208 Var210 Var211  
## factor factor factor factor factor factor factor factor factor  
## Var212 Var216 Var217 Var218 Var219 Var220 Var221 Var222 Var223  
## factor factor factor factor factor factor factor factor factor  
## Var226 Var227 Var228  
## factor factor factor  
## Levels: factor integer numeric
```

Filling Missing Numeric x 's with Means

The following x 's are **integer** or **numeric**:

```
numeric_input_feature_names <-  
  input_feature_names[input_feature_classes != 'factor']
```

```
numeric_input_feature_names
```

```
## [1] "Var6" "Var7" "Var13" "Var21" "Var22" "Var24" "Var25"  
## [8] "Var28" "Var35" "Var38" "Var44" "Var57" "Var65" "Var73"  
## [15] "Var74" "Var76" "Var78" "Var81" "Var83" "Var85" "Var109"  
## [22] "Var112" "Var113" "Var119" "Var123" "Var125" "Var132" "Var133"  
## [29] "Var134" "Var140" "Var143" "Var144" "Var149" "Var153" "Var160"  
## [36] "Var163" "Var173" "Var181"
```

It seems we don't have a problem with numeric columns made up of non-changing values:

```
numeric_input_feature_standard_deviations <-
  sapply(X_train[, numeric_input_feature_names, with=FALSE],
    function(col) sd(col, na.rm=TRUE))

numeric_input_feature_standard_deviations
```

```
##          Var6          Var7          Var13          Var21          Var22
## 2.843250e+03 6.413963e+00 2.699609e+03 5.867414e+02 7.330429e+02
##          Var24          Var25          Var28          Var35          Var38
## 1.096091e+01 2.107095e+02 9.637879e+01 2.860019e+00 3.000131e+06
##          Var44          Var57          Var65          Var73          Var74
## 1.424335e+00 2.028071e+00 1.030915e+01 5.338518e+01 4.038577e+02
##          Var76          Var78          Var81          Var83          Var85
## 1.830733e+06 2.130236e+00 1.055596e+05 1.002977e+02 2.377701e+01
##          Var109         Var112         Var113         Var119         Var123
## 1.634161e+02 1.645895e+02 7.729649e+05 2.291689e+03 2.391889e+02
##          Var125         Var132         Var133         Var134         Var140
## 1.056116e+05 9.773756e+00 2.451729e+06 5.988284e+05 3.036796e+03
##          Var143         Var144         Var149         Var153         Var160
## 6.530104e-01 1.180023e+01 6.573176e+05 4.357551e+06 9.910520e+01
##          Var163         Var173         Var181
## 8.376229e+05 1.252542e-01 2.431791e+00
```

Let's fill up the missing values with the means of the respective columns:

```
numeric_input_feature_means <-
  sapply(X_train[, numeric_input_feature_names, with=FALSE],
    function(col) mean(col, na.rm=TRUE))

for (numeric_col in numeric_input_feature_names) {
  x <- X_train[[numeric_col]]
  missing_value_row_yesno <- is.na(x)
  if (sum(missing_value_row_yesno) > 0) {
    X_train[, numeric_col := as.numeric(x), with=FALSE]
    mu <- numeric_input_feature_means[numeric_col]
    X_train[missing_value_row_yesno, numeric_col := mu, with=FALSE]
  }
}
```

Let's double check to see that the numeric columns have all been filled and that their means stay the same as before the filling:

```
all.equal(
  numeric_input_feature_means,
  sapply(X_train[, numeric_input_feature_names, with=FALSE], mean))
```

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

Cleaning Categorical Variables

Below are categorical features and their number of categories:

```
categorical_input_feature_names <-
  input_feature_names[input_feature_classes == 'factor']

categorical_input_feature_nb_levels <-
```

```
sapply(X_train[, categorical_input_feature_names, with=FALSE],
       function(col) length(levels(col)))
```

```
categorical_input_feature_nb_levels
```

```
## Var192 Var193 Var195 Var196 Var197 Var198 Var199 Var202 Var203 Var204
##      361      51      23      4      225      4291      5073      5713      5      100
## Var205 Var206 Var207 Var208 Var210 Var211 Var212 Var216 Var217 Var218
##       3      21      14      2      6      2      81      2016      13990      2
## Var219 Var220 Var221 Var222 Var223 Var226 Var227 Var228
##       22      4291      7      4291      4      23      7      30
```

Those variables having over 500 categories are likely to be just text / character data. Let's get rid of them:

```
categorical_input_feature_names <-
  categorical_input_feature_names[categorical_input_feature_nb_levels <= 500]

X_train <-
  X_train[, c(numeric_input_feature_names, categorical_input_feature_names), with=FALSE]
```

For the remaining categorical variables, let's:

- Make their missing values another category **zzzMISSING**; and
- Try to consolidate the categories, as having too many categories make modeling less meaningful and numerically more difficult; for each variable, we'll collapse all categories with prevalence of under 5% together into a **zzzOTHER** category;
- Drop categorical variables with only one category (*obviously*); and
- Drop categorical variables with only one non-**zzzMISSING** category.

```
collapsed_categories <- list()

for (cat_col in categorical_input_feature_names) {

  missing_value_row_yesno <- is.na(X_train[[cat_col]])
  if (sum(missing_value_row_yesno) > 0) {
    X_train[missing_value_row_yesno, cat_col := 'zzzMISSING', with=FALSE]
  }

  x <- X_train[[cat_col]]
  for (cat in levels(x)) {
    cat_rows_yesno <- x == cat
    if (sum(cat_rows_yesno) < .05 * nb_train_samples) {
      if (!(cat_col %in% names(collapsed_categories))) {
        collapsed_categories[[cat_col]] <- character()
      }
      collapsed_categories[[cat_col]] <- c(collapsed_categories[[cat_col]], cat)
      X_train[cat_rows_yesno, cat_col := 'zzzOTHER', with=FALSE]
      levels(X_train[[cat_col]])[levels(X_train[[cat_col]]) == cat] <- NA
    }
  }

  cats <- levels(X_train[[cat_col]])
  if ((length(cats) == 1) ||
      (length(cats[(cats != 'zzzMISSING') & (cats != 'zzzOTHER')]) < 2)) {
```

```

    categorical_input_feature_names <- setdiff(categorical_input_feature_names, cat_col)
  }
}

```

Let's double-check by looking at the prevalence of the categories of the remaining categorical variables now:

```

lapply(X_train[ , categorical_input_feature_names, with=FALSE],
       function(col) summary(col) / nb_train_samples)

```

```

## $Var193
## 2Knk1KF      R012 zzzOTHER
## 0.1494      0.7202 0.1304
##
## $Var197
##      OXwj      4871      JlbT      1K27      TyG1      zzzOTHER
## 0.09533333 0.07260000 0.06100000 0.08793333 0.08286667 0.60026667
##
## $Var203
##      9_Y1      HLqf      zzzOTHER
## 0.90200000 0.06466667 0.03333333
##
## $Var205
##      09_Q      sJzTla1      VpdQ      zzzOTHER
## 0.23313333 0.09040000 0.63880000 0.03766667
##
## $Var206
##      hAFG      haYg      IYzP      sYC_      zm5i zzzMISSING
## 0.0548000 0.0588000 0.3397333 0.0822000 0.1302667 0.1117333
##      zzzOTHER
## 0.2224667
##
## $Var207
##      7M47J5GA0pTYIFxg5uy DHn_WUyBhW_whjA88g9bvA64_
##      0.14260000      0.06893333
##      me75fM6ugJ      zzzOTHER
##      0.70006667      0.08840000
##
## $Var208
##      kIsH      sBgB zzzOTHER
## 0.9200 0.0772 0.0028
##
## $Var211
##      L84s      Mtgm
## 0.8079333 0.1920667
##
## $Var212
##      CrNX      NhsEn4L Xfqt03UdzaXh_      zzzOTHER
## 0.0586000 0.5880667 0.1282000 0.2251333
##
## $Var218
##      cJvF      UYBR      zzzOTHER
## 0.50113333 0.48453333 0.01433333
##
## $Var221
##      d0EEeJi      oslk      zCkv      zzzOTHER
## 0.05973333 0.74280000 0.12426667 0.07320000
##
## $Var223

```

```
## jySVZN10Jy LM81689q0p zzzMISSING zzzOTHER
## 0.12140000 0.73073333 0.10540000 0.04246667
##
## $Var226
##      7P5s      Aoh3      FSa2      Qu4f      szEZ      WqMG
## 0.05433333 0.05286667 0.16473333 0.09546667 0.05953333 0.08566667
##      zzzOTHER
## 0.48740000
##
## $Var227
##      6fzt      RAYp      ZI9m      zzzOTHER
## 0.06826667 0.70226667 0.12726667 0.10220000
##
## $Var228
##      55YFVY9 F2FyR07IdsN7I ib5G6X1eUxUn6      zzzOTHER
## 0.08840000 0.65240000 0.05226667 0.20693333
```

Not bad, *eh?*, not bad... It seems we can embark now on the next steps: variable selection.

Selecting Candidate Input Features x 's

```
input_feature_names <-
  c(numeric_input_feature_names, categorical_input_feature_names)

nb_input_features <- length(input_feature_names)

X_train <- X_train[ , input_feature_names, with=FALSE]
```

After data cleaning, we have **53** numeric and categorical input features left:

```
sapply(X_train, class)
```

```
##      Var6      Var7      Var13      Var21      Var22      Var24      Var25
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##      Var28      Var35      Var38      Var44      Var57      Var65      Var73
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "integer"
##      Var74      Var76      Var78      Var81      Var83      Var85      Var109
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##      Var112      Var113      Var119      Var123      Var125      Var132      Var133
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##      Var134      Var140      Var143      Var144      Var149      Var153      Var160
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##      Var163      Var173      Var181      Var193      Var197      Var203      Var205
## "numeric" "numeric" "numeric" "factor" "factor" "factor" "factor"
##      Var206      Var207      Var208      Var211      Var212      Var218      Var221
## "factor" "factor" "factor" "factor" "factor" "factor" "factor"
##      Var223      Var226      Var227      Var228
## "factor" "factor" "factor" "factor"
```

Building models with all of them will still be quite clunky. Let's try to select features containing good amounts of "signals" by:

1. Fitting Random Forests on pairs of features and measuring the OOS performances of such Random Forests
2. Pick pairs of higher OOB performances

3. Pick variables that appear in many well-performing pairs

```
feature_pair_performances <- data.table(
  feature_1=character(),
  feature_2=character(),
  deviance=numeric())

caret_optimized_metric <- 'logLoss' # equivalent to 1 / 2 of Deviance

caret_train_control <- trainControl(
  classProbs=TRUE, # compute class probabilities
  summaryFunction=mnLogLoss, # equivalent to 1 / 2 of Deviance
  method='repeatedcv', # repeated Cross Validation
  number=5, # number of folds
  repeats=1, # number of repeats
  allowParallel=TRUE)

B <- 30

for (i in 1 : (nb_input_features - 1)) {

  feature_1 <- input_feature_names[i]

  for (j in (i + 1) : nb_input_features) {

    cat('pair: ', i, ', ', j, '\n')

    feature_2 <- input_feature_names[j]

    rf_model <- train(
      x=X_train[, c(feature_1, feature_2), with=FALSE],
      y=churn_train,
      method='parRF', # parallel Random Forest
      metric=caret_optimized_metric,
      ntree=B, # number of trees in the Random Forest
      nodesize=300, # minimum node size set small enough to allow for complex trees,
      # but not so small as to require too large B to eliminate high variance
      importance=FALSE, # skip evaluate importance of predictors
      keep.inbag=FALSE, # not relevant as we're using Cross Validation
      trControl=caret_train_control,
      tuneGrid=NULL)

    feature_pair_performances <- rbind(
      feature_pair_performances,
      data.table(
        feature_1=feature_1,
        feature_2=feature_2,
        deviance=2 * rf_model$results$logLoss))
  }
}

feature_pair_performances_top_half <-
  feature_pair_performances[order(deviance), ][1 : round(nrow(feature_pair_performances) / 2), ]

good_feature_appearance_counts <- list()

for (i in 1: nrow(feature_pair_performances_top_half)) {

  feature_1 <- feature_pair_performances_top_half[i, feature_1]
```

```

if (!(feature_1 %in% names(good_feature_appearance_counts))) {
  good_feature_appearance_counts[[feature_1]] <- 1
} else {
  good_feature_appearance_counts[[feature_1]] <-
    good_feature_appearance_counts[[feature_1]] + 1
}

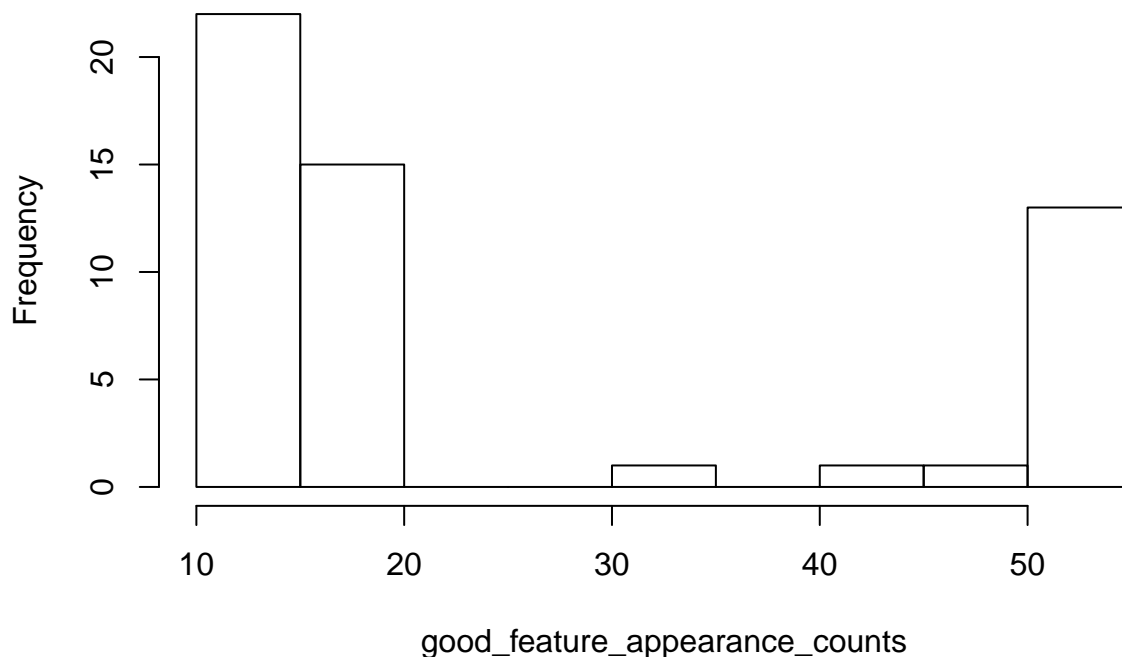
feature_2 <- feature_pair_performances_top_half[i, feature_2]
if (!(feature_2 %in% names(good_feature_appearance_counts))) {
  good_feature_appearance_counts[[feature_2]] <- 1
} else {
  good_feature_appearance_counts[[feature_2]] <-
    good_feature_appearance_counts[[feature_2]] + 1
}
}

good_feature_appearance_counts <-
  unlist(good_feature_appearance_counts)

hist(good_feature_appearance_counts)

```

Histogram of good_feature_appearance_counts



We see that predictive power seems to concentrate in a much smaller number of features. Let's pick the features that appear over 30 times in the "good feature appearances".

```

input_feature_names <-
  names(good_feature_appearance_counts)[good_feature_appearance_counts > 30]

input_feature_names

```

```

## [1] "Var134" "Var153" "Var57" "Var133" "Var38" "Var125" "Var119"
## [8] "Var81" "Var13" "Var113" "Var163" "Var76" "Var149" "Var140"
## [15] "Var28" "Var6"

```

```

nb_input_features <- length(input_feature_names)

X_train <- X_train[ , input_feature_names, with=FALSE]

input_feature_classes <- sapply(X_train, class)

input_feature_classes

##      Var134      Var153      Var57      Var133      Var38      Var125      Var119
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##      Var81      Var13      Var113      Var163      Var76      Var149      Var140
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##      Var28      Var6
## "numeric" "numeric"

```

It turns out that **all of the remaining 16 strong features** are **numeric!** (*meaning the effort we spent on cleaning the categoricals has come to naught... but we couldn't have known that without trying*)

Classification Models

Let's train 2 types of classification models: a Random Forest and a Boosted Trees model:

```

caret_optimized_metric <- 'logLoss'    # equivalent to 1 / 2 of Deviance

caret_train_control <- trainControl(
  classProbs=TRUE,                    # compute class probabilities
  summaryFunction=mnLogLoss,          # equivalent to 1 / 2 of Deviance
  method='repeatedcv',               # repeated Cross Validation
  number=5,                          # 5 folds
  repeats=3,                         # 2 repeats
  allowParallel=TRUE)

```

```

B <- 600

rf_model <- train(
  x=X_train,
  y=churn_train,
  method='parRF',                    # parallel Random Forest
  metric=caret_optimized_metric,
  ntree=B,                          # number of trees in the Random Forest
  nodesize=100,                     # minimum node size set small enough to allow for complex trees,
                                     # but not so small as to require too large B to eliminate high variance
  importance=TRUE,                  # evaluate importance of predictors
  keep.inbag=TRUE,
  trControl=caret_train_control,
  tuneGrid=NULL)

```

```

B <- 1200

boost_model <- train(
  x=X_train,
  y=churn_train,
  method='gbm',                     # Generalized Boosted Models
  metric=caret_optimized_metric,
  verbose=FALSE,

```

```
trControl=caret_train_control,
tuneGrid=expand.grid(
  n.trees=B,           # number of trees
  interaction.depth=10, # max tree depth,
  n.minobsinnode=100,   # minimum node size
  shrinkage=0.01))      # shrinkage parameter, a.k.a. "learning rate"
```

We'll now evaluate the OOS performances of these 2 models on the Validation set to select the better one:

```
low_prob <- 1e-6
high_prob <- 1 - low_prob
log_low_prob <- log(low_prob)
log_high_prob <- log(high_prob)
log_prob_thresholds <- seq(from=log_low_prob, to=log_high_prob, length.out=100)
prob_thresholds <- exp(log_prob_thresholds)

# Prepare Validation Data for evaluation
prepare_oos_input_features <- function(X_OOS) {
  X_OOS <- X_OOS[, input_feature_names, with=FALSE]
  for (numeric_col in input_feature_names) {
    x <- X_OOS[[numeric_col]]
    X_OOS[, numeric_col := as.numeric(x), with=FALSE]
    X_OOS[is.na(x), numeric_col := numeric_input_feature_means[numeric_col], with=FALSE]
  }
  X_OOS
}

X_valid <- prepare_oos_input_features(X_valid)

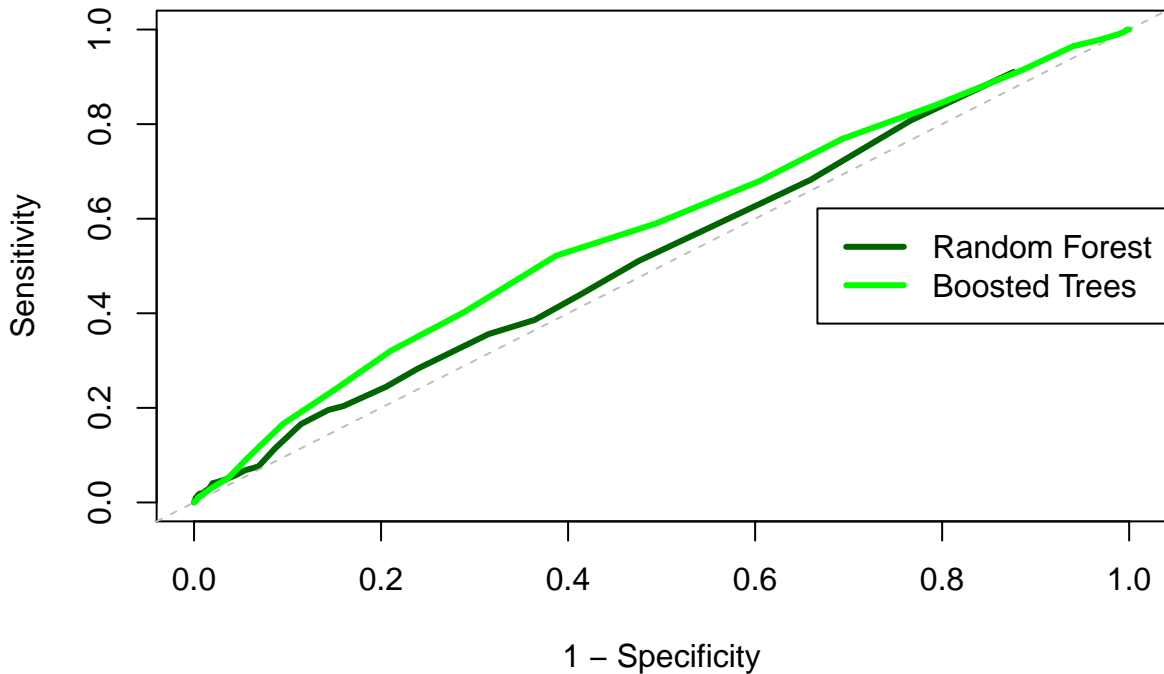
# *** NOTE: **
# the below "bin_classif_eval" function is from the "EvaluationMetrics.R" helper script
# in the "HelpR" GitHub repo

rf_pred_probs <- predict(
  rf_model, newdata=X_valid, type='prob')
rf_oos_performance <- bin_classif_eval(
  rf_pred_probs$yes, churn_valid, thresholds=prob_thresholds)

boost_pred_probs <- predict(
  boost_model, newdata=X_valid, type='prob')
boost_oos_performance <- bin_classif_eval(
  boost_pred_probs$yes, churn_valid, thresholds=prob_thresholds)

plot(x=1 - rf_oos_performance$specificity,
     y=rf_oos_performance$sensitivity,
     type = "l", col='darkgreen', lwd=3,
     xlim = c(0., 1.), ylim = c(0., 1.),
     main = "ROC Curves (Validation Data)",
     xlab = "1 - Specificity", ylab = "Sensitivity")
abline(a=0,b=1,lty=2,col=8)
lines(x=1 - boost_oos_performance$specificity,
     y=boost_oos_performance$sensitivity,
     col='green', lwd=3)
legend('right', c('Random Forest', 'Boosted Trees'),
     lty=1, col=c('darkgreen', 'green'), lwd=3, cex=1.)
```

ROC Curves (Validation Data)



It seems that although neither model seems super impressive – customer churn is probably a hard thing to predict very well – the Boosted Trees model offers a much better classification performance than the Random Forest. We now need to pick a decision threshold for the Boosted Trees model. If we are to be really rigorous, we'll need balance the costs of lost business and the costs of extra incentives to retain customers. Here, to make life simple, we'll pick a subjective threshold that enables us to anticipate **25%** of the churn cases:

```
sensitivity_threshold <- .25
i <- min(which(boost_oos_performance$sensitivity < sensitivity_threshold)) - 1
selected_prob_threshold <- prob_thresholds[i]
```

The selected decision threshold is **0.093** – meaning when we use the Boosted Tree model to predict on new data, we'll predict a customer churn when the predicted probability exceeds that threshold. The expected performance of the model at that threshold is as follows:

```
boost_oos_performance[i, ]

##      threshold accuracy sensitivity specificity precision  f1_score
## 1: 0.09326026 0.755049   0.3206522   0.7895532 0.1079597 0.1615332
##      deviance
## 1: 0.527913
```

Note that the precision of the model at this sensitivity threshold is rather low, meaning that there'll be many false positives. We'll probably need business insights to decide whether to contact certain customers over other, and what incentives to offer them.

Test Performance of Selected Model

Let's then evaluate the performance of the selected Boosted Trees model, with a decision threshold at **0.093**:

```
X_test <- prepare_oos_input_features(X_test)
```

```

boost_test_pred_probs <- predict(
  boost_model, newdata=X_test, type='prob')

boost_test_performance <- bin_classif_eval(
  boost_test_pred_probs$yes, churn_test, thresholds=selected_prob_threshold)

boost_test_performance

```

```

##      accuracy sensitivity specificity   precision    f1_score   deviance
## 0.7543251   0.3458920   0.7866959   0.1138843   0.1713515   0.5169046

```

We can see that the Test performance is similar to what we've estimated from the Validation set.

```

stopCluster(cl)  # shut down the parallel computing cluster

```