RemixAutoML Library Introduction

Adrian Antico

2019-03-18

Contact Info

Email: adrianantico@gmail.com

LinkedIn: https://www.linkedin.com/in/adrian-antico/

Remix Instute: https://www.remyxcourses.com or adrian.antico@remix.institute

Vignette Intent

This vignette is designed to give you the highlights of the set of automated machine learning functions available in the RemixAutoML package. To see the functions in action, visit the Remyx Courses website for the free course at **https://www.remyxcourses.com** and walk through them (and check our the other courses too!).

Package Goals

The **RemixAutoML** package (*Remix Automated Machine Learning*) is designed to automate and optimize the quality of machine learning, the pace of development, along with the handling of big data and the processing time of data management. The library has been a development task at Remix Institute over the course of the past year to consolidate all of our winning methods for successfully completing machine learning and data science consulting projects. We are avid R users and feel that the R community could benefit from its release.

Function Design Philosophy

The two core packages RemixAutoML relies on are H20 and data.table. There are other packages used, for example, the forecast package, but H20 and data.table are used the most. I use data.table for data wrangling of all internal functions due to its ability to handle big data with minimal memory and the speed at which their functions process data. I chose to use H20 and their machine learning algorithms because of their high quality results, flexibility of use, ease of operationalization, and ability to manage big data.

There are five categories of functions (currently) in this library I'll go over:

- Automated Supervised Learning
- Automated Unsupervised Learning
- Automated Model Evaluation, Feature Interpretation, and Cost Sensitive Optimization
- Automated Feature Engineering
- A Few Miscellaneous Functions

Automated Supervised Learning Functions

AutoH20Modeler()

The supervised learning functions handle multiple tasks internally. For example, the **AutoH20Modeler** function can build any number of models, automatically compare hyper-parameter tuned versions to baseline versions, selecting a winner, saveing the model evaluation and feature interpretation metrics / graphs, along with storing models and their metadata to refer to them in a production setting.

The models available include:

- Gradient Boosting Machines (Linux only)
- LightGBM (Linux only)
- Distributed Random Forest (DRF)
- XGBoost (Linux only)
- Deeplearning
- AutoML (for Windows users XGBoost and LightGBM are not tried)

For Windows users (Mac?), XGBoost is not available and therefore neither is LightGBM (XGBoost and LightGBM are not utlized in AutoML model selection with Windows).

AutoTS()

Another automated supervised learning function we have is an automated time series modeling function that optimially builds out eight types of forecasting models, compares them on holdout data, picks a winner, rebuilds the winner on full data, and generates the forecasts for the number of desired periods. The intent is to make these processes fast, easy, and of high quality. Every model makes use of the optimal settings of their paramters to give them the best chance of being the best. Each model uses a box-cox transformation and predicts are back-transformed.

The models tried include:

- ARFIMA (Autoregressive Fractional Integrated Moving Average)
- ARIMA (Autoregressive Integrated Moving Average)
- ETS (Exponential Smoothing and Holt Winters)
- TBATS (Exonential Smoothing State Space Model with Box-Cox Transformation, ARMA Errors, Trend and Seasonal Components)
- TSLM (Time Series Linear Model)
- NN (Autoregressive Neural Network)
- Facebook's Prophet

nlsModelFit()

The other automated supervised learning function builds nonlinear regression models for a more niche set of tasks. It's set up to generate interpolation predictions, such as smoothing cost curves for optimization tasks.

The models competing include:

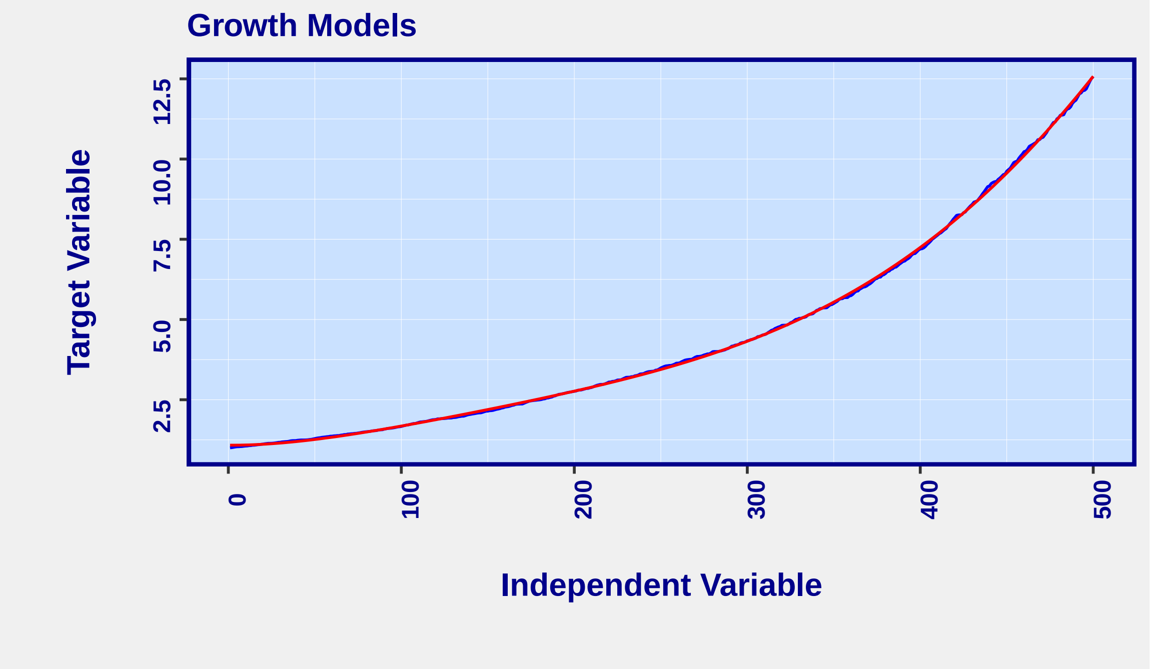
- Asymptotic
- Asymptotic through origin
- Asymptotic with offset

- Bi-exponential
- Four parameter logistic
- Three parameter logistic
- Gompertz
- Michal Menton
- Weibull
- Polynomial regression or monotonic regression

Example of nlsModelFit with Plot (simulated data)

Find more demos at **https://www.remyxcourses.com**

```
library(RemixAutoML)

# Create Growth Data
data <- data.table::data.table(Variable = seq(1,500,1), Target = rep(1, 500))
for (i in as.integer(2:500)) {
  var = data[i-1, "Target"][[1]]
  data.table::set(data, i = i, j = 2L, value = var * (1 + runif(1)/100))
}

# Build Model and Merge Onto Source Data
data1 <- data.table::copy(data)
data2 <- merge(data1,
               nlsModelFit(data = data, y = "Target", x = "Variable", monotonic = FALSE),
               by = "Variable",
               all = FALSE)
data2[, Target.z := lm(Target.x ~ poly(Variable,10))]
#> Warning in `[.data.table`(data2, , `:=`(Target.z, lm(Target.x ~
#> poly(Variable, : Supplied 12 items to be assigned to 500 items of column
#> 'Target.z' (recycled leaving remainder of 8 items).
ggplot2::ggplot(data2, ggplot2::aes(x = Variable)) +
  ggplot2::geom_line(ggplot2::aes(y = data2[["Target.x"]], color = "Target"), color = "blue") +
  ggplot2::geom_line(ggplot2::aes(y = data2[["Target.y"]], color = "Predicted"), color = "red") +
  ChartTheme(Size = 12) + ggplot2::ggtitle("Growth Models") +
  ggplot2::ylab("Target Variable") + ggplot2::xlab("Independent Variable")
```

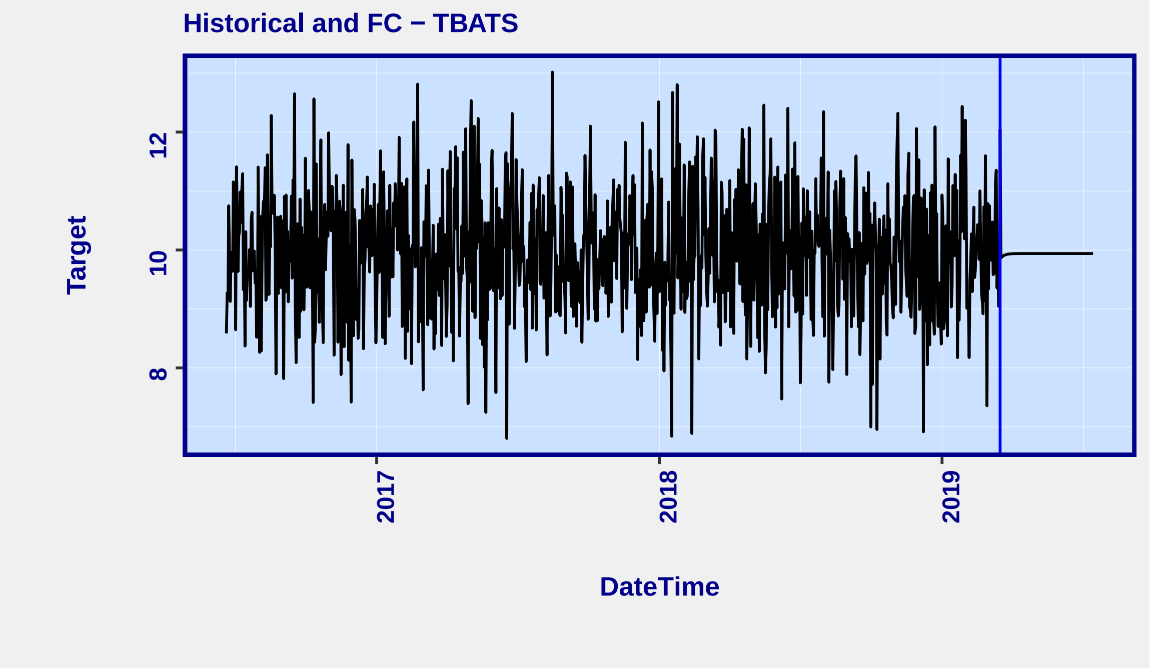


Example of AutoTS with Plot (simulated data)

Find more demos at **https://www.remyxcourses.com**

```
library(RemixAutoML)
data <- data.table::data.table(DateTime = as.Date(Sys.time()),
                               Target = (10 + arima.sim(model = list(2,0,2), n = 1000)))
data[, temp := seq(1:1000)][, DateTime := DateTime - temp][, temp := NULL]
data <- data[order(DateTime)]
output <- AutoTS(data,
                 TargetName     = "Target",
                 DateName       = "DateTime",
                 FCPeriods      = 120,
                 HoldOutPeriods = 30,
                 TimeUnit       = "day",
                 Lags           = 5,
                 SLags          = 1,
                 NumCores       = 4,
                 SkipModels     = NULL,
                 StepWise       = TRUE)
#> [1] "ARFIMA FITTING"
#> [1] "ARIMA FITTING"
#> [1] "ETS FITTING"
#> [1] "TBATS FITTING"
#> [1] "TSLM FITTING"
#> [1] "NNet FITTING"
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
#> [1] "PROPHET FITTING"
#> [1] "FIND WINNER"
#> [1] "GENERATE FORECASTS"
data1 <- output[[1]]
maxDate <- data[, max(DateTime)]
data.table::setnames(data1, names(data1), c("DateTime","Target"))
data2 <- data.table::rbindlist(list(data[, Target := as.numeric(Target)],data1))
ggplot2::ggplot(data2, ggplot2::aes(x = DateTime, y = Target)) +
  ggplot2::geom_line() + ChartTheme(Size = 10) +
  ggplot2::geom_vline(ggplot2::aes(xintercept = maxDate), color = "blue") +
  ggplot2::ggtitle(paste0("Historical and FC - ", output[[2]][1,1][[1]])) +
  ggplot2::theme(legend.position="none")

knitr::kable(output[[2]])
```

ModelName	MeanResid	MeanPercError	MAPE	ID
TBATS	0.0634951	-0.0857255	0.1011000	1
ETS	0.0220069	-0.0891849	0.1032459	2
ARFIMA	0.0218760	-0.0891958	0.1032526	3
ARIMA	0.0218735	-0.0891960	0.1032527	4
NN	-0.1682298	-0.1046958	0.1136217	5



Automated Unsupervised Learning Functions

The suite of functions in this category currently handle optimized row-clustering and anomaly detection. For the row-clustering, we utilize H20's Generalized Low Rank Model and their KMeans algorithm, with hyper-parameter tuning for both. We have a few others currently in development and will release those when they are complete. The anomaly detection functions we have currently are for time series applications. We have a control chart methodology version that lets you build upper and lower confidence bounds by up to two grouping variables along with a time series modeling version. The clustering function and the control chart method function update your data set that you feed in with new columns that store the clusterID or anomaly information. The time series function updates your data, supplies you with the final time series model built, and a data.table that only contains anomalies.

Functions include:

- `GenTSAnomVars()`
- `ResidualOutliers()`
- `GLRM_KMeans_Col()`

Automated Model Evaluation, Feature Interpretation, and Cost Sensitive Optimization Functions

The model evaluation graphs are calibration plots or calibration boxplots. The calibration plots are used for regression (expected value and quantile regession), classification, and multinomial modeling problems. The calibration boxplots are used for regression (expected value and quantile regression). These graphs display both the actual target values and the predicted values, grouped by the number of bins that you specify. The calibration boxplots are useful to understand not only the model bias but also the model variance, across the range of predicted values.

Functions include:

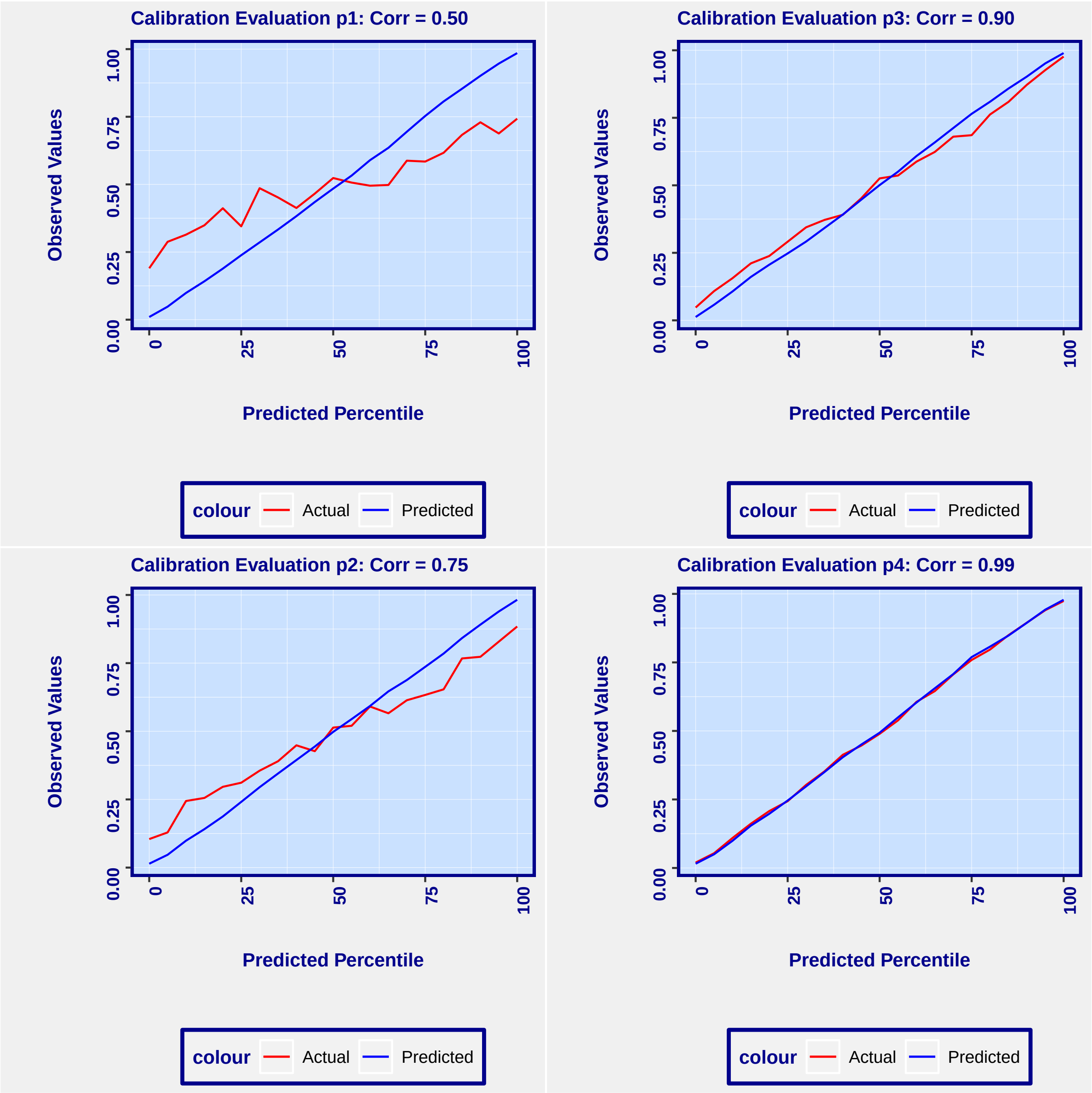
- `EvalPlot()`

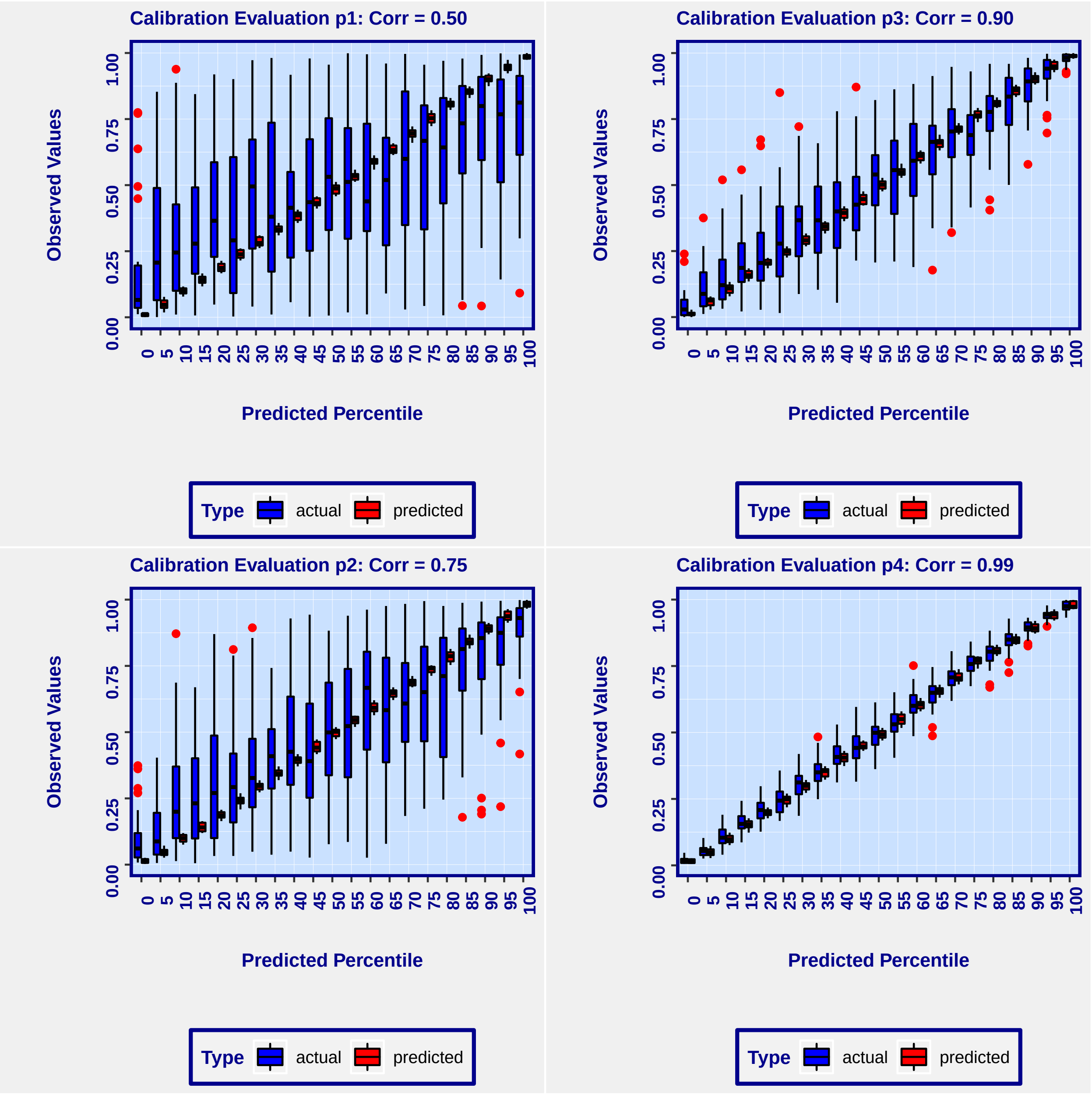
Demo of EvalPlot() for calibration and boxplots

Find more demos at **https://www.remyxcourses.com**

```r
library(RemixAutoML)

# Data generator function
dataGen <- function(Correlation = 0.95) {
  Validation <- data.table::data.table(target = runif(1000))
  Validation[, x1 := qnorm(target)]
  Validation[, x2 := runif(1000)]
  Validation[, predict := pnorm(Correlation * x1 +
                                  sqrt(1 - Correlation ^2) * qnorm(x2))]
  return(Validation)
}

# Store data sets
data1 <- dataGen(Correlation = 0.50)
data2 <- dataGen(Correlation = 0.75)
data3 <- dataGen(Correlation = 0.90)
data4 <- dataGen(Correlation = 0.99)

# Generate EvalPlots (calibration)
p1 <- EvalPlot(data = data1,
               PredColName = "predict",
               ActColName = "target",
               type = "calibration",
               bucket = 0.05,
               aggrfun = function(x) mean(x, na.rm = TRUE))
p1 <- p1 + ggplot2::ggtitle("Calibration Evaluation p1: Corr = 0.50") +
  RemixAutoML::ChartTheme(Size = 10)

p2 <- EvalPlot(data = data2,
               PredColName = "predict",
               ActColName = "target",
               type = "calibration",
               bucket = 0.05,
               aggrfun = function(x) mean(x, na.rm = TRUE))
p2 <- p2 + ggplot2::ggtitle("Calibration Evaluation p2: Corr = 0.75") +
  RemixAutoML::ChartTheme(Size = 10)

p3 <- EvalPlot(data = data3,
               PredColName = "predict",
               ActColName = "target",
               type = "calibration",
               bucket = 0.05,
               aggrfun = function(x) mean(x, na.rm = TRUE))
p3 <- p3 + ggplot2::ggtitle("Calibration Evaluation p3: Corr = 0.90") +
  RemixAutoML::ChartTheme(Size = 10)

p4 <- EvalPlot(data = data4,
               PredColName = "predict",
               ActColName = "target",
               type = "calibration",
               bucket = 0.05,
               aggrfun = function(x) mean(x, na.rm = TRUE))
p4 <- p4 + ggplot2::ggtitle("Calibration Evaluation p4: Corr = 0.99") +
  RemixAutoML::ChartTheme(Size = 10)
RemixAutoML::multiplot(plotlist = list(p1,p2,p3,p4), cols = 2)

# Generate EvalPlots (boxplots)
p1 <- EvalPlot(data = data1,
               PredColName = "predict",
               ActColName = "target",
               type = "boxplot",
               bucket = 0.05)
p1 <- p1 + ggplot2::ggtitle("Calibration Evaluation p1: Corr = 0.50") +
  RemixAutoML::ChartTheme(Size = 10)

p2 <- EvalPlot(data = data2,
               PredColName = "predict",
               ActColName = "target",
               type = "boxplot",
               bucket = 0.05)
p2 <- p2 + ggplot2::ggtitle("Calibration Evaluation p2: Corr = 0.75") +
  RemixAutoML::ChartTheme(Size = 10)
```

```
p3 <- EvalPlot(data = data3,
               PredColName = "predict",
               ActColName = "target",
               type = "boxplot",
               bucket = 0.05)
p3 <- p3 + ggplot2::ggtitle("Calibration Evaluation p3: Corr = 0.90") +
  RemixAutoML::ChartTheme(Size = 10)

p4 <- EvalPlot(data = data4,
               PredColName = "predict",
               ActColName = "target",
               type = "boxplot",
               bucket = 0.05)
p4 <- p4 + ggplot2::ggtitle("Calibration Evaluation p4: Corr = 0.99") +
  RemixAutoML::ChartTheme(Size = 10)
RemixAutoML::multiplot(plotlist = list(p1,p2,p3,p4), cols = 2)
```


Calibration Evaluation p1: Corr = 0.50
Observed Values
0.00
0.25
0.50
0.75
1.00
0
25
50
75
100
Predicted Percentile
colour
Actual
Predicted
Calibration Evaluation p3: Corr = 0.90
Observed Values
0.00
0.25
0.50
0.75
1.00
0
25
50
75
100
Predicted Percentile
colour
Actual
Predicted
Calibration Evaluation p2: Corr = 0.75
Observed Values
0.00
0.25
0.50
0.75
1.00
0
25
50
75
100
Predicted Percentile
colour
Actual
Predicted
Calibration Evaluation p4: Corr = 0.99
Observed Values
0.00
0.25
0.50
0.75
1.00
0
25
50
75
100
Predicted Percentile
colour
Actual
Predicted



The feature interpretation function graphs are very similar in nature to the model evaluation graphs. They display partial dependence calibration line plots, partial dependence calibration boxplots, and partial dependence calibration bar plots (for factor variables with the ability to limit the number of factors shown with the remainder grouped into "other"). The line graph version is for numerical features and have the ability to aggregate by quantile for quantile regression.

Functions include:

- `ParDepCalPlots()`

The cost sensitive optimizaition functions provide the user the ability to generate utility-optimized thresholds for classification tasks. There are two of these functions: one for generating a single threshold based on the values supplied to your cost confusion matrix outcomes and the second one provides two thresholds, where your final predicted classification could be (0|1) and "do something else". With the latter function, you would also need to supply a cost to the "do something else" option.

Functions include:

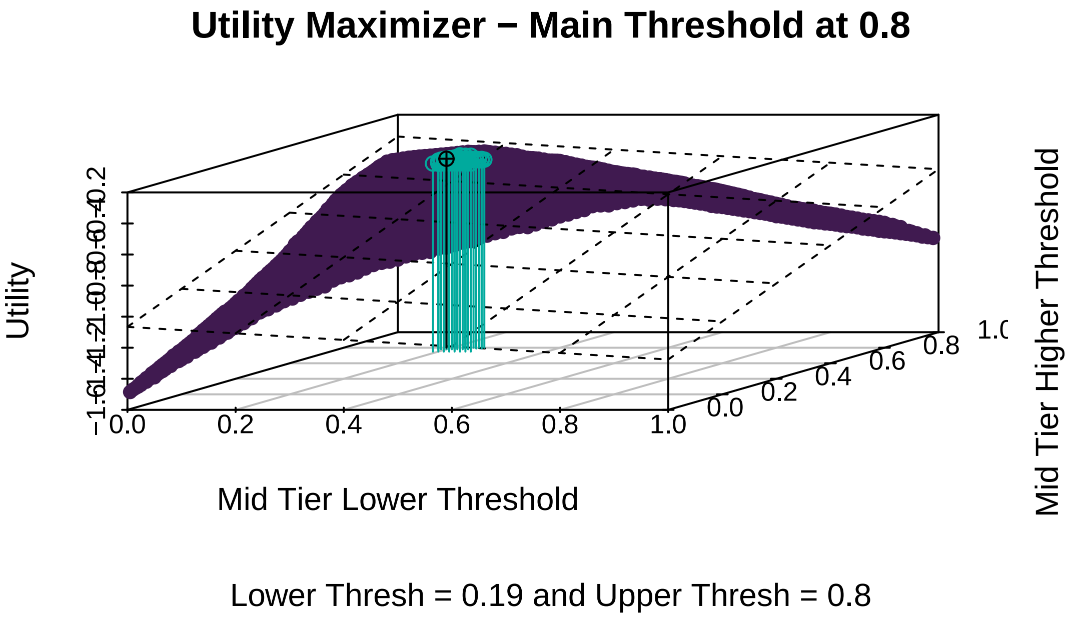
- threshOptim()
- RedYellowGreen()

RedYellowGreen Output (simulated data)

Find more demos at **https://www.remyxcourses.com**

```
library(RemixAutoML)
Correl <- 0.70
  aa <- data.table::data.table(target = runif(1000))
  aa[, x1 := qnorm(target)]
  aa[, x2 := runif(1000)]
  aa[, predict := pnorm(Correl * x1 +
                              sqrt(1 - Correl ^2) *
                              qnorm(x2))]
  aa[, target := as.numeric(ifelse(target < 0.5, 0, 1))]
  data <- RedYellowGreen(
    aa,
    PredictColNumber  = 4,
    ActualColNumber   = 1,
    TruePositiveCost  = 0,
    TrueNegativeCost  = 0,
    FalsePositiveCost = -3,
    FalseNegativeCost = -2,
    MidTierCost       = -0.5,
    Cores = 1
  )

  knitr::kable(data[order(-Utility)][1:10])
```

TPP	TNP	FPP	FNP	MTDN	MTC	Threshold	MTLT	MTHT	Utility
0	0	-3	-2	TRUE	-0.5	0.80	0.19	0.80	-0.3833986
0	0	-3	-2	TRUE	-0.5	0.79	0.19	0.79	-0.3842190
0	0	-3	-2	TRUE	-0.5	0.80	0.21	0.80	-0.3844591
0	0	-3	-2	TRUE	-0.5	0.81	0.19	0.81	-0.3846729
0	0	-3	-2	TRUE	-0.5	0.80	0.22	0.80	-0.3848765
0	0	-3	-2	TRUE	-0.5	0.80	0.23	0.80	-0.3849063
0	0	-3	-2	TRUE	-0.5	0.79	0.21	0.79	-0.3851208
0	0	-3	-2	TRUE	-0.5	0.80	0.20	0.80	-0.3852527
0	0	-3	-2	TRUE	-0.5	0.79	0.23	0.79	-0.3853955
0	0	-3	-2	TRUE	-0.5	0.79	0.22	0.79	-0.3854379



Automated Feature Engineering Functions

This suite of functions are what will take your models to the next level. The core functions are the generalized distributed lag and rolling statistics functions. I have four of them.

Functions include:

- `GDL_Feature_Engineering()`
- `DT_GDL_Feature_Engineering()`
- `FAST_GDL_Feature_Engineering()`
- `Scoring_GDL_Feature_Engineering()`

The first three are used for building out lags and rolling statistics from target variables (numeric type; including classification models (0|1) and multinomial models with a little bit of work) and numeric features over your entire data set (no aggregation is done) with the option for creating the rolling statistics on the main variable or the lag1 version of the main variable. You can also compute time between records (by group) and add their lags and rolling statistics as well (really useful for transactional data). They can be generated using a single grouping variable (for multiple grouping variables you can concatenate them) and you can feed in a list of grouping variables to generate them by. The first function (**GDL__**) has the largest variety of rolling statics options but runs the slowest. The second function (**DT__GDL__**) runs the fastest but only generates moving averages. The third function (**FAST__GDL__**) is used for cases where you don't need to generate the features across the entire data set. Suppose you have a limited number of target variable instance but a rich history of data. You can use the FAST__GDL__ version to create lags and rolling statistics for N number of records previous to each target instance (i.e. not the entire historical data set). The fourth function (**Scoring__GDL__**) is for use in a production setting where you need to generate single instances of the feature set quickly. You basically feed in the same arguments as you used for the other versions and out the other end is the same set of features, identically named.

DT_GDL_Feature_Engineering and Scoring_GDL_Feature_Engineering Demo (simulated data)

Find more demos atFind many more at **https://www.remyxcourses.com**

```
library(RemixAutoML)

# Build data for feature engineering for modeling
N <- 25116
ModelData <-
  data.table::data.table(GroupVariable = sample(
    x = c(letters,LETTERS,paste0(letters, letters),
          paste0(LETTERS, LETTERS),
          paste0(letters, LETTERS),
          paste0(LETTERS, letters))),
    DateTime = base::as.Date(Sys.time()),
    Target = stats::filter(rnorm(N,
                                 mean = 50,
                                 sd = 20),
                           filter = rep(1, 10),
                           circular = TRUE))
ModelData[, temp := seq(1:161), by = "GroupVariable"][
       , DateTime := DateTime - temp][
         , temp := NULL]
ModelData <- ModelData[order(DateTime)]
ModelData <- DT_GDL_Feature_Engineering(
  ModelData,
  lags           = c(seq(1, 5, 1)),
  periods        = c(3, 5, 10, 15, 20, 25),
  statsNames     = c("MA"),
  targets        = c("Target"),
  groupingVars   = "GroupVariable",
  sortDateName   = "DateTime",
  timeDiffTarget = c("Time_Gap"),
  timeAgg        = c("days"),
  WindowingLag   = 1,
  Type           = "Lag",
  Timer          = FALSE,
  SkipCols       = FALSE,
  SimpleImpute   = TRUE)
#> [1] 22

# Build data for feature engineering for scoring
N <- 25116
ScoringData <-
  data.table::data.table(GroupVariable = sample(
    x = c(letters,LETTERS,paste0(letters, letters),
          paste0(LETTERS, LETTERS),
          paste0(letters, LETTERS),
          paste0(LETTERS, letters))),
    DateTime = base::as.Date(Sys.time()),
    Target = stats::filter(rnorm(N,
                                 mean = 50,
                                 sd = 20),
                           filter = rep(1, 10),
                           circular = TRUE))
ScoringData[, temp := seq(1:161),
      by = "GroupVariable"][, DateTime := DateTime - temp]
ScoringData <- ScoringData[order(DateTime)]
ScoringData <- Scoring_GDL_Feature_Engineering(
  ScoringData,
  lags           = c(seq(1, 5, 1)),
  periods        = c(3, 5, 10, 15, 20, 25),
  statsFUNs      = c(function(x) mean(x, na.rm = TRUE)),
  statsNames     = c("MA"),
  targets        = c("Target"),
  groupingVars   = c("GroupVariable"),
  sortDateName   = c("DateTime"),
  timeDiffTarget = c("Time_Gap"),
  timeAgg        = "days",
  WindowingLag   = 1,
  Type           = "Lag",
  Timer          = FALSE,
  SkipCols       = FALSE,
  SimpleImpute   = TRUE,
  AscRowByGroup  = "temp",
  RecordsKeep    = 1
)

# View some of new features
knitr::kable(ModelData[order(GroupVariable,-DateTime)][1:10,c(3,4,14)])
```

Target	GroupVariable_LAG_1_Target	GroupVariableMA_3_GroupVariable_LAG_1_Target
527.7905	546.6630	508.3582
546.6630	490.0472	502.2630
490.0472	488.3643	503.3223
488.3643	528.3775	497.5640
528.3775	493.2250	481.3731
493.2250	471.0895	473.5757
471.0895	479.8049	512.6965
479.8049	469.8328	492.1492
469.8328	588.4518	512.3498
588.4518	418.1630	408.1736

```
# Ensure names equal
knitr::kable(
  data.table::as.data.table(
    cbind(ModelData_Names = sort(names(ModelData)),
          ScoringData_Names = sort(names(ScoringData[, temp := NULL])))))
```

ModelData_Names	ScoringData_Names
DateTime	DateTime
GroupVariable	GroupVariable
GroupVariable_LAG_1_Target	GroupVariable_LAG_1_Target

ModelData_Names	ScoringData_Names
GroupVariable_LAG_2_Target	GroupVariable_LAG_2_Target
GroupVariable_LAG_3_Target	GroupVariable_LAG_3_Target
GroupVariable_LAG_4_Target	GroupVariable_LAG_4_Target
GroupVariable_LAG_5_Target	GroupVariable_LAG_5_Target
GroupVariableMA_10_GroupVariable_LAG_1_Target	GroupVariableMA_10_GroupVariable_LAG_1_Target
GroupVariableMA_10_GroupVariableTime_Gap1	GroupVariableMA_10_GroupVariableTime_Gap1
GroupVariableMA_15_GroupVariable_LAG_1_Target	GroupVariableMA_15_GroupVariable_LAG_1_Target
GroupVariableMA_15_GroupVariableTime_Gap1	GroupVariableMA_15_GroupVariableTime_Gap1
GroupVariableMA_20_GroupVariable_LAG_1_Target	GroupVariableMA_20_GroupVariable_LAG_1_Target
GroupVariableMA_20_GroupVariableTime_Gap1	GroupVariableMA_20_GroupVariableTime_Gap1
GroupVariableMA_25_GroupVariable_LAG_1_Target	GroupVariableMA_25_GroupVariable_LAG_1_Target
GroupVariableMA_25_GroupVariableTime_Gap1	GroupVariableMA_25_GroupVariableTime_Gap1
GroupVariableMA_3_GroupVariable_LAG_1_Target	GroupVariableMA_3_GroupVariable_LAG_1_Target
GroupVariableMA_3_GroupVariableTime_Gap1	GroupVariableMA_3_GroupVariableTime_Gap1
GroupVariableMA_5_GroupVariable_LAG_1_Target	GroupVariableMA_5_GroupVariable_LAG_1_Target
GroupVariableMA_5_GroupVariableTime_Gap1	GroupVariableMA_5_GroupVariableTime_Gap1
GroupVariableTime_Gap1	GroupVariableTime_Gap1
GroupVariableTime_Gap2	GroupVariableTime_Gap2
GroupVariableTime_Gap3	GroupVariableTime_Gap3
GroupVariableTime_Gap4	GroupVariableTime_Gap4
GroupVariableTime_Gap5	GroupVariableTime_Gap5
Target	Target

Functions include:

- `Word2VecModel()`
- `ModelDataPrep()`
- `DummifyDT()`

The **Word2VecModel** function converts your text features into numerical vector representations. You supply the function with your data set and all the text column names you want converted, and out the other end you have a data set with all the features merged on. The models can be saved to file and metadata saves their paths for scoring purposes in a production setting. The models built are based on H20's word2vec algorithm and has done an execellent job at extracting high quality information out of those text columns. The **ModelDataPrep** function is used to prepare your data for modeling with the **AutoH20Modeler** function. It will convert character columns to factors, replace inf values to NA, and impute missing values (both numeric and factor based on supplied values). The **DummifyDT** function will turn your character (or factor) columns into dummy variable columns. You can specify one-hot encoding or not in which you will get N+1 columns for one-hot or N columns otherwise.

Miscellaneous Functions

Functions include:

- `WordFreq()`
- `ChartTheme()`
- `RemixTheme()`
- `multiplot()`
- `PrintObjectsSize()`
- `percRank()`

The **WordFreq** function will go through a process of cleaning your text column, doing some other text operations, and ouput a table with word frequencies and a word cloud plot. The **ChartTheme** and **RemixTheme** functions will turn your ggplots into nicely formatted and colored charts, worthy of presentation. The **multiplot** function are for those who have had a terrible time plotting multiple graphs onto a single image. The **PrintObjectsSize** function is more of a debugging function for inspecting the size of variables in your environment (useful in looping functions). The **percRank** is simply a function to compute the percentile rank of every value in a column of data.