RemixAutoML Library Introduction

Adrian Antico

2019-09-07

Contact Info

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

Remix Instute: https://www.remyxcourses.com or https://www.remixinstitute.ai

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. These were actual projects at Fortune 500 companies, Fortune 100 companies, tech startups, and other consulting clients. We are avid R users and feel that the R community could benefit from its release.

Package Design Philosophy

Core packages utilized

There are several core packages RemixAutoML relies on which include. From a data management standpoint, I utilize data.table exclusively for data wrangling of all internal functions due to its ability to handle big data with a minimal memory footprint, along with the speed at which their functions process data. For the machine learning functions, I utilize H2O, Catboost, XGBoost, and forecast due to their quality results and ability to handle big data (some can run on GPU as well). I use these functions routinely for machine learning projects and they continue to outperform every other method I test them against. Many of the other R packages for modeling or data manipulation have slow run times and fail once I get going with bigger data. There are several unique functions to this package that help to optimize the machine learning process, such as feature engineering functions, model evaluation functions, model interpretation functions, and model output optimization functions.

Machine learning methodology

The package is designed to give your models the best chance at being the most accurate for your machine learning tasks. There are functions in here to help you get the most out of your data for the models to take advantage of. There are three types of features you need to manage for machine learning: numeric, categorical, and text features. The functions in this package will help you squeeze as much information out

of your data set as possible. Then you simply supply those data sets into the automated machine learning functions for state of the art output with minimal effort on your part. With the extra time saved you can try out significantly more experiments to ensure you are putting the best models and frameworks possible for your machine learning use-cases.

Handling numerical features

Numerical features can hide relationships in a variety of ways and it's our goal as modeling professionals (or researchers) to capture as many of those relationships as we can (or enough to provide a sufficient return on investment). We have linear and nonlinear relationships, linear and nonlinear interactions, along with threshold effects. Those are what I call column-based model effects. Nowadays, you don't really have to bother with any of those when using tree-based ensembles (boosting and bagging) as those relationships can be captured without the need to manual create features to account for their peculiarities, aside from the target variables in which we still offer automated transformation creation and scoring functions. However, there is still a significant amount of potential information to be extracted from your data if you don't account for time-based effects. In the business world, many applications require modeling data that is collected across time (think transactional data). With time series data (or panel data), where data is collected (or manipulated to be) across equally spaced time periods (hourly, daily, etc.), you tend to look at lags and moving averages of your target variable for predicting the present and future events. Well, why aren't we using the same type of features in our transaction data? That's where we offer our '*GDL*' suite of functions. They are designed to create lags and rolling statistics off of numeric target variables and numeric independent variables, by groups. They can also generate lags and rolling statistics off of the time between events, by groups. This is what I call row-based model effects. Any nonlinearities and interactions will be captured by the tree-based ensembles.

Taking this a step further, there is a concept out there called target encoding. What's done, essentially, is that the mean of the target is used as a replacement for factor levels, thus converting your factor variables into numeric variables. I generally have two problems with this. For one thing, there is inherent forward leak because you are using all values across time to predict values that occur historically. Second, it fails to account for all the other information in your data to be used in the transformation process. If you used the '*GDL*' suite of functions, you can do your target encoding without any forward leak (they are forced to prevent this), but you will also be able to utilize various window sizes (opposed to full history) to capture recent trends or cyclical effects, and do the same thing based on your independent variables that have effects distributed across time.

Handling categorical features

In other modeling frameworks, such as Python, you need to convert your categorical features into dummy variables. This means you need to take care of that coding task along with managing that in a production environment. With this package, categorical features are handled internally within the automated modeling functions. Turning your categorical features into dummy variables is problematic for high cardinality factor variables. There are other approaches and the automated modeling functions will actually test out the other methods to see which ones offer the best performance. So you don't have to deal with the coding and management of factor variables and you get better performance. Of course, if you're working with time-based data, I would recommend using the '*GDL*' suite of functions to convert your factor variables to numeric types to prevent forward leak and to not miss out on all that sweet information to be gained!

Handling text data

We are living in a world now where text data is becoming more readily available and your machine learning models aren't suited to handle them without first managing them. With this package, you can simply run the **AutoWord2VecModeler()** function, which builds skip-gram word2vec models from H2O, for all your text columns, thus replacing your text data with numerical vectors suitable for modeling. The function

will save the model(s) and score new text data on the fly with the **AutoH2OScoring()** function in your production setting. You can choose to build a single model for each text column or a single model for all text columns.

Handling date data

Date columns can hold quite a bit of useful information but aren't typically used inside the ML algorithms. We need to extract information out of them. Currently, the package has a function to extract the time components such as second, minute, hour, weekday, day of month, day of year, week, month, quarter, and year.

Machine learning algorithms

The machine learning algorithms available have been demonstrated to provide optimial performance on a wide-range of business use-cases over the years. They are all intended to remove the coding aspects behind tuning, evaluation, and interpretation, along with consistent output for comparison amongst them.

Interpreting your models

Once you have your models developed, you or a boss may want to see what features are most important and their relationship to the target variable. The functions available are also run internally to build out partial dependence calibration plots. These show you the modeled predicted relationship along with the empirical relationship in the same graph to show the end user how accurate the relationship is what the relationship is (even for categorical variables). Variable importance tables are also available to view along with evaluation calibration plots and boxplots.

Operationalizing models with ease

The scoring process in machine learning is typically straightforward. We have an several all-purpose scoring functions to score all your supervised machine learning models, text models, and clustering models, regardless of type (H2O, XGBoost, and catboost).

How to install H2O

Follow this link to install H2O if it isn't on your machine already. Install H2O

How to install catboost

```
devtools::install_github('catboost/catboost', subdir = 'catboost/R-package')
```

Review catboost

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

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

Automated Supervised Learning Functions

Functions include:

Binary Classification

- `AutoCatBoostClassifier()`
- `AutoXGBoostClassifier()`
- `AutoH2oGBMClassifier()`
- `AutoH2oDRFClassifier()`

MultiClass Classification

- `AutoCatBoostMultiClass()`
- `AutoXGBoostMultiClass()`
- `AutoH2oGBMMultiClass()`
- `AutoH2oDRFMultiClass()`

Regression

- `AutoCatBoostRegression()`
- `AutoH2oGBMRegression()`
- `AutoH2oDRFRegression()`
- `AutoXGBoostRegression()`

Scoring Functions

- `AutoCatBoostScoring()`
- `AutoXGBoostScoring()`
- `AutoH2oMLScoring()`

Hurdle Models

- `AutoCatBoostdHurdleModel()`
- `AutoXGBoostdHurdleModel()`
- `AutoH2oDRFHurdleModel()`
- `AutoH2oGBMdHurdleModel()`

Time Series

- `AutoTS()`
- `AutoCatBoostCARMA()`
- `AutoXGBoostCARMA()`
- `AutoH2oDRFCARMA()`
- `AutoH2oGBMCARMA()`
- `TimeSeriesFill()`

Non Linear Regression

- `AutoNLS()`

Marketing Modleing

- `AutoRecomDataCreate()`
- `AutoRecommender()`
- `AutoRecommenderScoring()`
- `AutoMarketBasketModel()`

General Purpose Modeling and Scoring

- `AutoH2OModeler()`
- `AutoH2OScoring()`

AutoCatBoostRegression()

AutoXGBoostRegression()

AutoH2oGBMRegression()

AutoH2oDRFRegression()

The Auto___Regression() set are automated regression modeling function that runs a variety of steps. First, the functions will run a random grid tune over N number of models and find which model is the best on holdout test data (a default model is always included in that set). Once the model is identified and built, several other outputs are generated: validation data with predictions, evaluation calibration plot, evaluation calibration boxplot, evaluation model metrics, variable importance, partial dependence calibration plots, partial dependence calibration box plots, grid metrics, grid arguments, and column names used in model fitting. You can fit standard expected value regression (all of them) along with quantile regression (catboost and h2o gbm).

AutoCatBoostClassifier()

AutoXGBoostClassifier()

AutoH2oGBMClassifier()

AutoH2oDRFClassifier()

The Auto___Classifier() set are automated binary classification modeling functions that runs a variety of steps. First, the function will run a random grid tune over N number of models and find which model is the best on holdout test data (a default model is always included in that set). Once the model is identified and built, several other outputs are generated: validation data with predictions, ROC plot, evaluation calibration plot, evaluation metrics, variable importance, partial dependence calibration plots, grid metrics, grid arguments, and column names used in model fitting.

AutoCatBoostMultiClass()

AutoXGBoostMultiClass()

AutoH2oGBMMultiClass()

AutoH2oDRFMultiClass()

The Auto___MultiClass() set are automated multiclass modeling functions that runs a variety of steps. First, the function will run a random grid tune over N number of models and find which model is the best on holdout test data (a default model is always included in that set). Once the best model is identified and built, several other outputs are generated: validation data with predictions, evaluation metrics, variable importance, grid metrics, grid arguments, and column names used in model fitting.

AutoCatBoostScoring()

AutoXGBoostScoring()

AutoH2oMLScoring()

The Auto___Scoring() functions will automatically do your data conversion to prepare for scoring and then score your data with rapid speed. For example, the AutoXGBoostScoring() will automatically create your dummy variable columns that you used in model training so that you can generate predictions with ease.

AutoCatBoostHurdleModel()

AutoXGBoostHurdleModel()

AutoH2oDRFHurdleModel()

AutoH2oGBMHurdleModel()

This is a modeling framework for building the necessary models for making predictions for hurdle modeling use-cases. It's generalized so that you can define any number of buckets (zero and greater than zero being the typical hurdle model case). First step is to build either a binary classification model (in the case of a single bucket value, such as zero) or a multiclass model (for the case of multiple bucket values, such as zero and 10). The next step is to subset the data for the cases of: less than the first bucket, in between the first and second, second and third, ..., second to last and last, along with greater than last. For each data subset, a regression model is built for predicting values in the bucket ranges. The final compilation is to multiply the probabilities of being in each bucket times the values supplied by the regression values for each buckets.

AutoH2OModeler()

The supervised learning functions handle multiple tasks internally. The **AutoH2OModeler()** function can build any number of H2O models, automatically compare hyper-parameter tuned versions to baseline versions, selecting a winner, saving the model evaluation and feature interpretation metrics / graphs, along with storing models and their metadata to refer to them later in a production setting. The models available include: Gradient Boosting Machines, LightGBM (Linux only), Distributed Random Forest, XGBoost (Linux only), Deeplearning, and AutoML (for Windows users XGBoost and LightGBM are not available).

AutoH2OScoring()

This function is the complement of the **AutoH2OModeler**, **AutoKMeans**, and **AutoWord2VecModeler** functions. Specify which rows of your model metadata collection file to run and **AutoH2OScoring** will return a list of predicted values, where each element of the list is a set of predicted values from the model it ran. For the **AutoH2OModeler** you will generate a file called grid_tuned_paths.Rdata which contains the path to your models (among other items) that you can pass along to the **AutoH2OScoring** function to automatically score your models. For the **AutoKMeans** you will generate a file called KMeansModelFile.Rdata which contains the paths to the models for scoring your GLRM and KMeans models. For the **AutoWord2VecModeler** you will generate a file called StoreFile.Rdata which contains the paths to your word2vec models for scoring. In total, the **AutoH2OScoring** function can score: Regression models, Quantile regression Models, Binary classification Models, Multinomial classification Models, Multioutcome multinomial classification models, Generalized low rank dimensionality reduction models, KMeans clustering models, and Word2vec models.

AutoTS()

Another automated supervised learning function we have is an automated time series modeling function (AutoTS) that optimally builds out seven types of time series 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 on the target variable and all predictions are back-transformed. It also compares model-based frequency determination versus user-supplied (for the TimeUnit argument) along with the option to have imputation and outlier replacement conducted. The competing models include: DSHW (Double Seasonal Holt Winters), 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).

AutoCatBoostCARMA()

AutoXGBoostCARMA()

AutoH2oDRFCARMA()

AutoH2oGBMCARMA()

This is a unique function that utilizes machine learning models (currently catboost and xgboost) to generate time series forecasts. It replicates the time series models forecasting approach, where it predicts one step ahead and re-generates the model features, and makes the next forecast, etc. If you need to generate forecasts for say 100,000 unique series, you can simply supply the grouping variables that identify which series is which, and generate forecasts for every one.

AutoRecomDataCreate()

This function will automatically turn your transactional data into a binary ratings matrix that you supply to the AutoRecommender() function for model building and the AutoRecommenderScoring() function for scoring.

AutoRecommender()

This function builds out several variations of collaborative filtering models on a binary ratings matrix. To automatically build the binary ratings matrix, see **AutoRecomDataCreate**. The competing models include: RandomItems, PopularItems, UserBasedCF, ItemBasedCF, AssociationRules.

AutoRecommenderScoring()

This function will automatically score your winning model. Simply feed in your data and the winning model returned from the **AutoRecommender** function and this function will generate a table of several recommended products (by rank) for each entity. This process is parallelized for fast scoring.

AutoMarketBasketModel()

AutoMarketBasketModel function runs a market basket analysis automatically. It will convert your data, run the algorithm, and add on additional significance values not orginally contained within.

AutoNLS()

This 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. It returns the interpolated data, the winning model name, the model object, and the evaluation metrics table. The competing models include: Asymptotic, Asymptotic through origin, Asymptotic with offset, Bi-exponential, Four parameter logistic, Three parameter logistic, Gompertz, Michal Menton, Weibull, and Polynomial regression or monotonic regression.

Example of AutoXGBoostRegression()

Find more demos at **https://www.remyxcourses.com/course?courseid=intro-to-remixautoml-in-r**

```
library(RemixAutoML)
library(data.table)

# Create Simulated Data to Demonstrate Modeling
Correl <- 0.85
N <- 1000
data <- data.table::data.table(Adrian = runif(N))
data[, x1 := qnorm(Adrian)]
data[, x2 := runif(N)]
data[, Independent_Variable1 := log(pnorm(Correl * x1 +
                                          sqrt(1-Correl^2) * qnorm(x2)))]
data[, Independent_Variable2 := (pnorm(Correl * x1 +
                                       sqrt(1-Correl^2) * qnorm(x2)))]
data[, Independent_Variable3 := exp(pnorm(Correl * x1 +
                                          sqrt(1-Correl^2) * qnorm(x2)))]
data[, Independent_Variable4 := exp(exp(pnorm(Correl * x1 +
                                              sqrt(1-Correl^2) * qnorm(x2))))]
data[, Independent_Variable5 := sqrt(pnorm(Correl * x1 +
                                           sqrt(1-Correl^2) * qnorm(x2)))]
data[, Independent_Variable6 := (pnorm(Correl * x1 +
                                       sqrt(1-Correl^2) * qnorm(x2)))^0.10]
data[, Independent_Variable7 := (pnorm(Correl * x1 +
                                       sqrt(1-Correl^2) * qnorm(x2)))^0.25]
```

```
data[, Independent_Variable8 := (pnorm(Correl * x1 +
                                       sqrt(1-Correl^2) * qnorm(x2)))^0.75]
data[, Independent_Variable9 := (pnorm(Correl * x1 +
                                       sqrt(1-Correl^2) * qnorm(x2)))^2]
data[, Independent_Variable10 := (pnorm(Correl * x1 +
                                        sqrt(1-Correl^2) * qnorm(x2)))^4]
data[, Independent_Variable11 := as.factor(
  ifelse(Independent_Variable2 < 0.20, "A",
         ifelse(Independent_Variable2 < 0.40, "B",
                ifelse(Independent_Variable2 < 0.6,  "C",
                       ifelse(Independent_Variable2 < 0.8,  "D", "E")))))]
data[, ':=' (x1 = NULL, x2 = NULL)]

# Build Models
TestModel <- RemixAutoML::AutoXGBoostRegression(
  data,
  ValidationData = NULL,
  TestData = NULL,
  TargetColumnName = "Adrian",
  FeatureColNames = 2:12,
  IDcols = NULL,
  TransformNumericColumns = "Adrian",
  eval_metric = "RMSE",
  Trees = 50,
  GridTune = TRUE,
  grid_eval_metric = "mae",
  MaxModelsInGrid = 10,
  NThreads = 8,
  TreeMethod = "hist",
  model_path = NULL,
  metadata_path = NULL,
  ModelID = "FirstModel",
  NumOfParDepPlots = 3,
  Verbose = 0,
  ReturnModelObjects = TRUE,
  SaveModelObjects = FALSE,
  ReturnFactorLevels = TRUE,
  PassInGrid = NULL)
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
#> [1] 6
#> [1] 7
#> [1] 8
#> [1] 9
#> [1] 10
#> [1] 11

# Evaluation metrics
TestModel$EvaluationMetrics
#>    Metric MetricValue
```

```
#>  1:    MAE      0.1344
#>  2:   MAPE      0.0934
#>  3:    MSE      0.0273
#>  4:     CS      0.9578

# Variable Importance
TestModel$VariableImportance
#>                     Feature   Gain  Cover Frequency
#>  1:    Independent_Variable1 0.9649 0.7731    0.8067
#>  2:    Independent_Variable2 0.0303 0.1478    0.1495
#>  3: Independent_Variable11_B 0.0017 0.0233    0.0125
#>  4: Independent_Variable11_C 0.0011 0.0260    0.0102
#>  5: Independent_Variable11_A 0.0010 0.0048    0.0102
#>  6: Independent_Variable11_D 0.0008 0.0209    0.0094
#>  7: Independent_Variable11_E 0.0001 0.0041    0.0016

# Grid List of Arguments Tested
TestModel$GridList
#>      eta max_depth min_child_weight subsample colsample_bytree
#>  1: 0.35         6                2       0.9              1.0
#>  2: 0.30        10                2       1.0              0.8
#>  3: 0.25         6                1       1.0              0.9
#>  4: 0.30         6                3       0.9              0.8
#>  5: 0.30         6                2       0.8              0.9
#>  6: 0.30         8                2       1.0              0.8
#>  7: 0.25        10                2       1.0              0.9
#>  8: 0.25        10                2       1.0              0.8
#>  9: 0.35        10                1       1.0              1.0
#> 10: 0.35         8                3       1.0              1.0
#> 11: 0.30        10                1       1.0              0.8

# Metrics from Grid Tuning
TestModel$GridMetrics
#>     ParamRow EvalStat
#>  1:        1   0.1183
#>  2:        2   0.1182
#>  3:        3   0.1182
#>  4:        4   0.1159
#>  5:        5   0.1188
#>  6:        6   0.1175
#>  7:        7   0.1178
#>  8:        8   0.1178
#>  9:        9   0.1174
#> 10:       10   0.1180
#> 11:       11   0.1182

# Transformation Object
TestModel$TransformationResults
#>    ColumnName MethodName Lambda NormalizedStatistics
#> 1:     Adrian      Asinh     NA                1.2352

# Factor Levels for Scoring
TestModel$FactorLevelsList
```

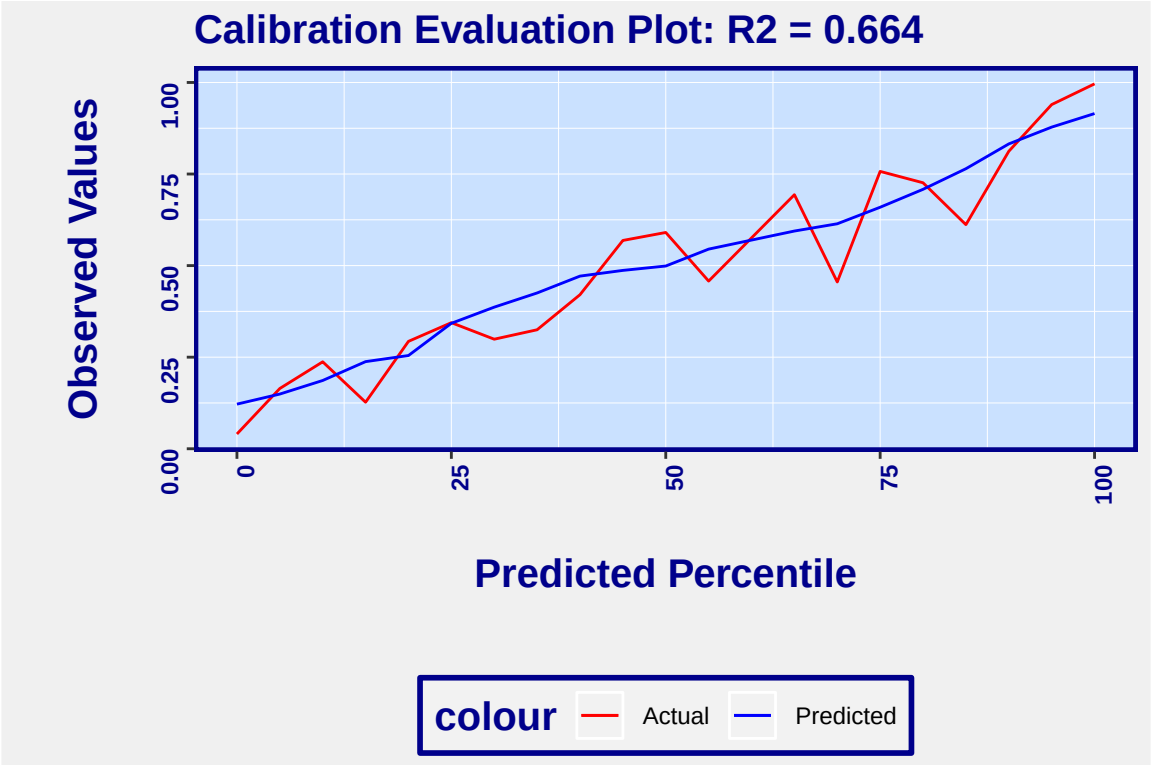
```
#> $Independent_Variable11
#>    Independent_Variable11
#> 1:                      D
#> 2:                      B
#> 3:                      E
#> 4:                      A
#> 5:                      C
```

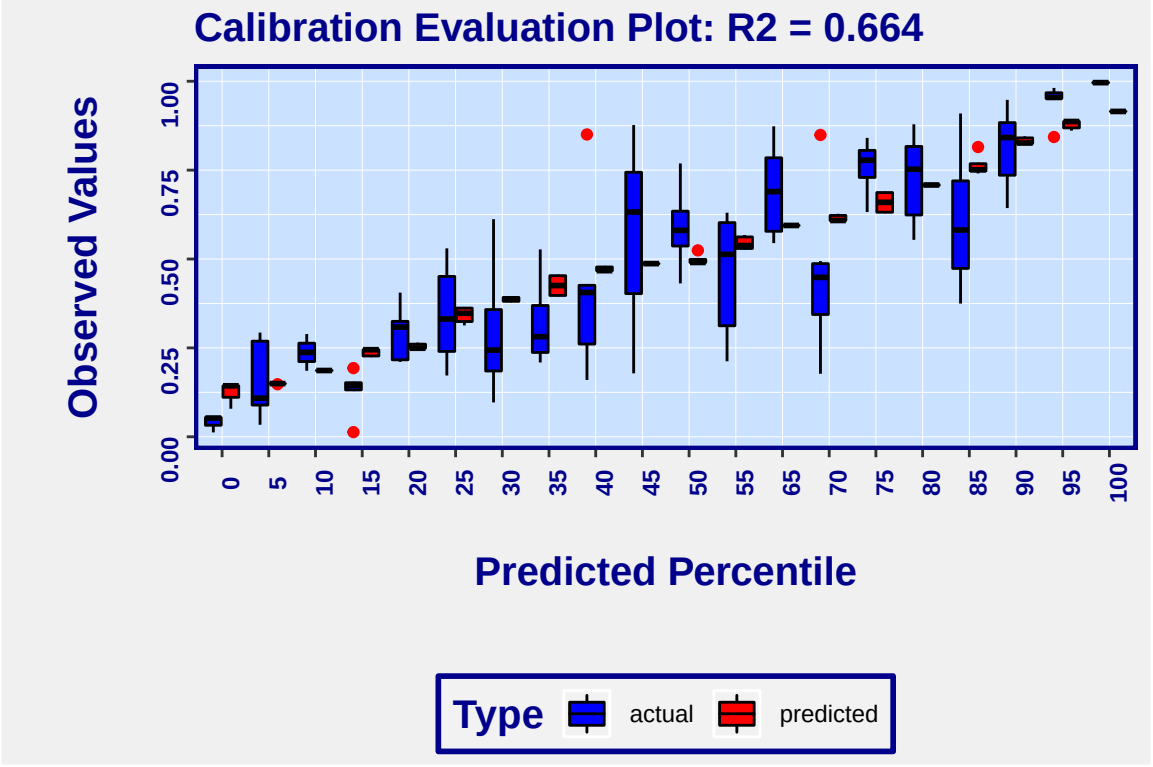
```
# Evaluation Calibration Plot
TestModel$EvaluationPlot
```

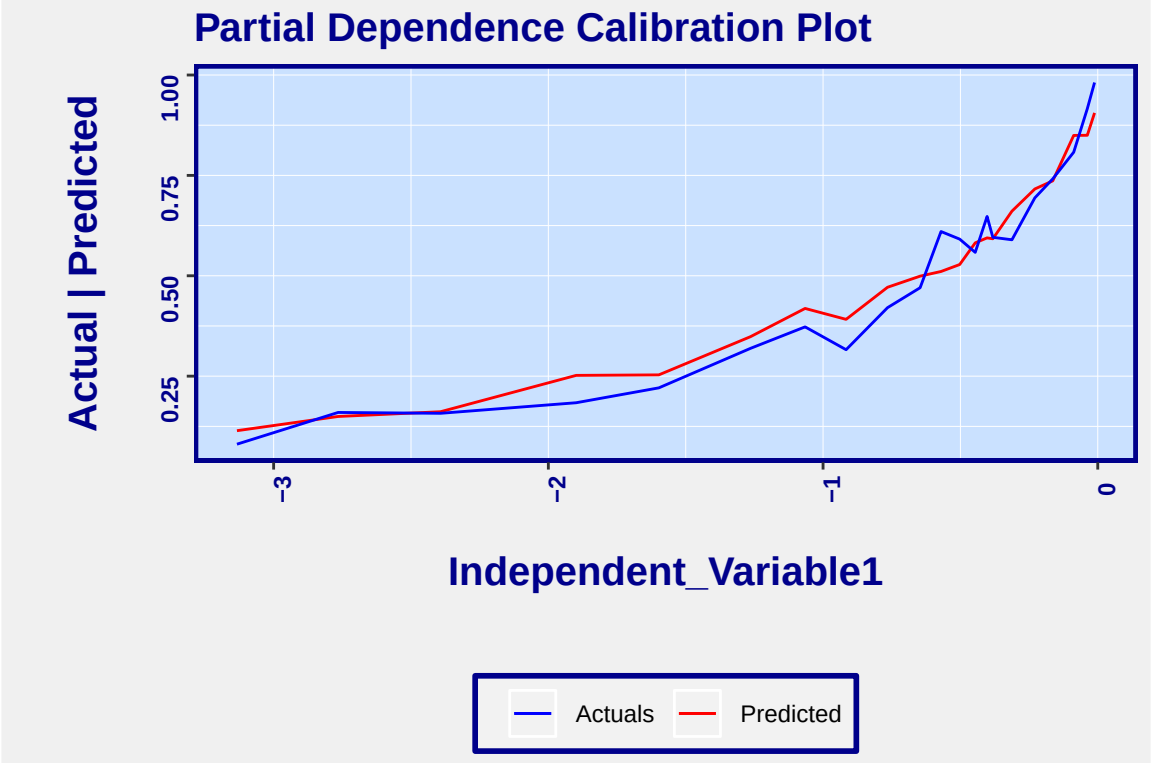
```
# Evaluation Calibration BoxPlot
TestModel$EvaluationBoxPlot
```

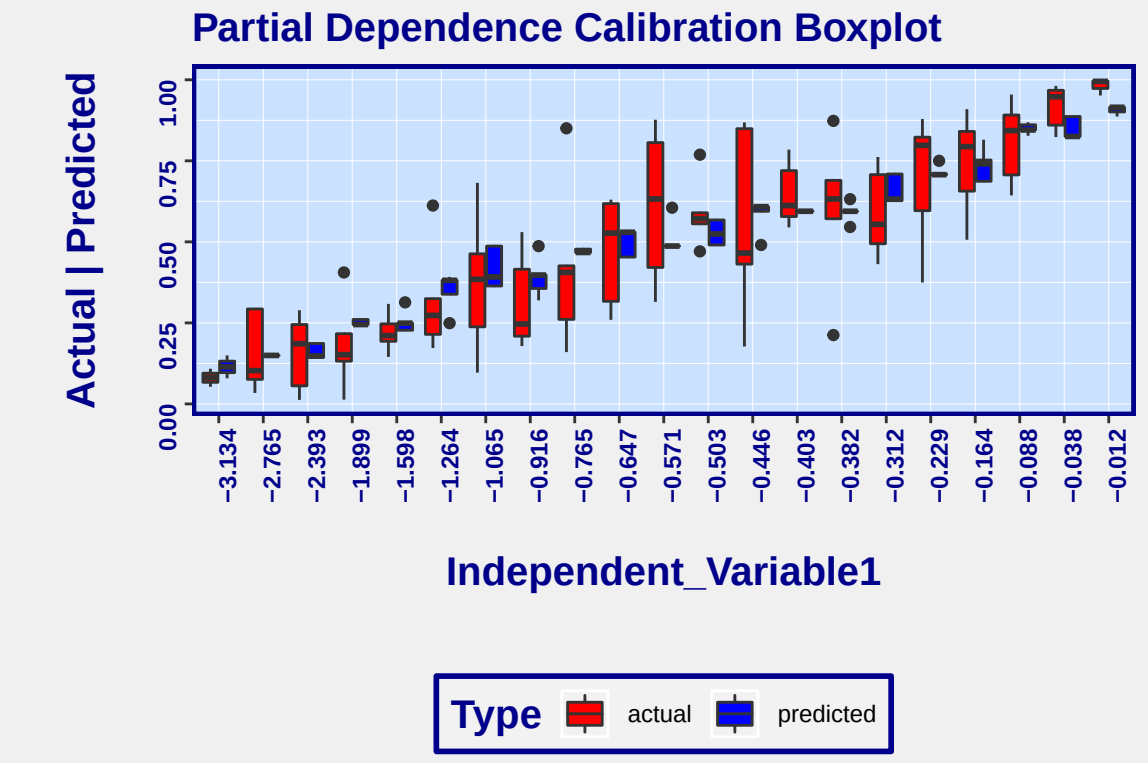
```
# Partial Dependence Calibration Plot
TestModel$PartialDependencePlots$Independent_Variable1
```

```
# Partial Dependence Calibration BoxPlot
TestModel$PartialDependenceBoxPlots$Independent_Variable1
```




Calibration Evaluation Plot: R2 = 0.664
Observed Values
0.00
0.25
0.50
0.75
1.00
0
5
10
15
20
25
30
35
40
45
50
55
65
70
75
80
85
90
95
100
Predicted Percentile
Type
actual
predicted


Partial Dependence Calibration Plot
Actual | Predicted
0.25
0.50
0.75
1.00
−3
−2
−1
0
Independent_Variable1
Actuals
Predicted

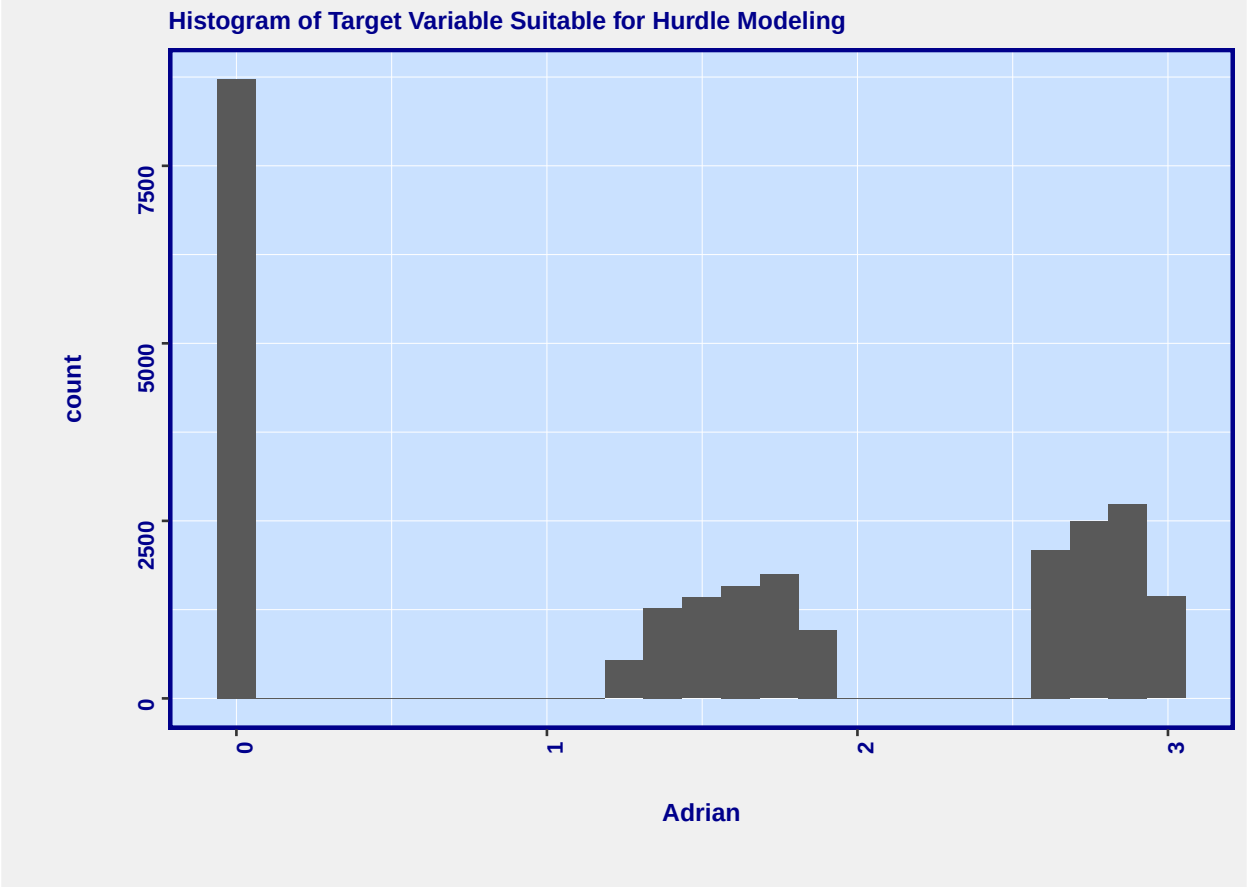


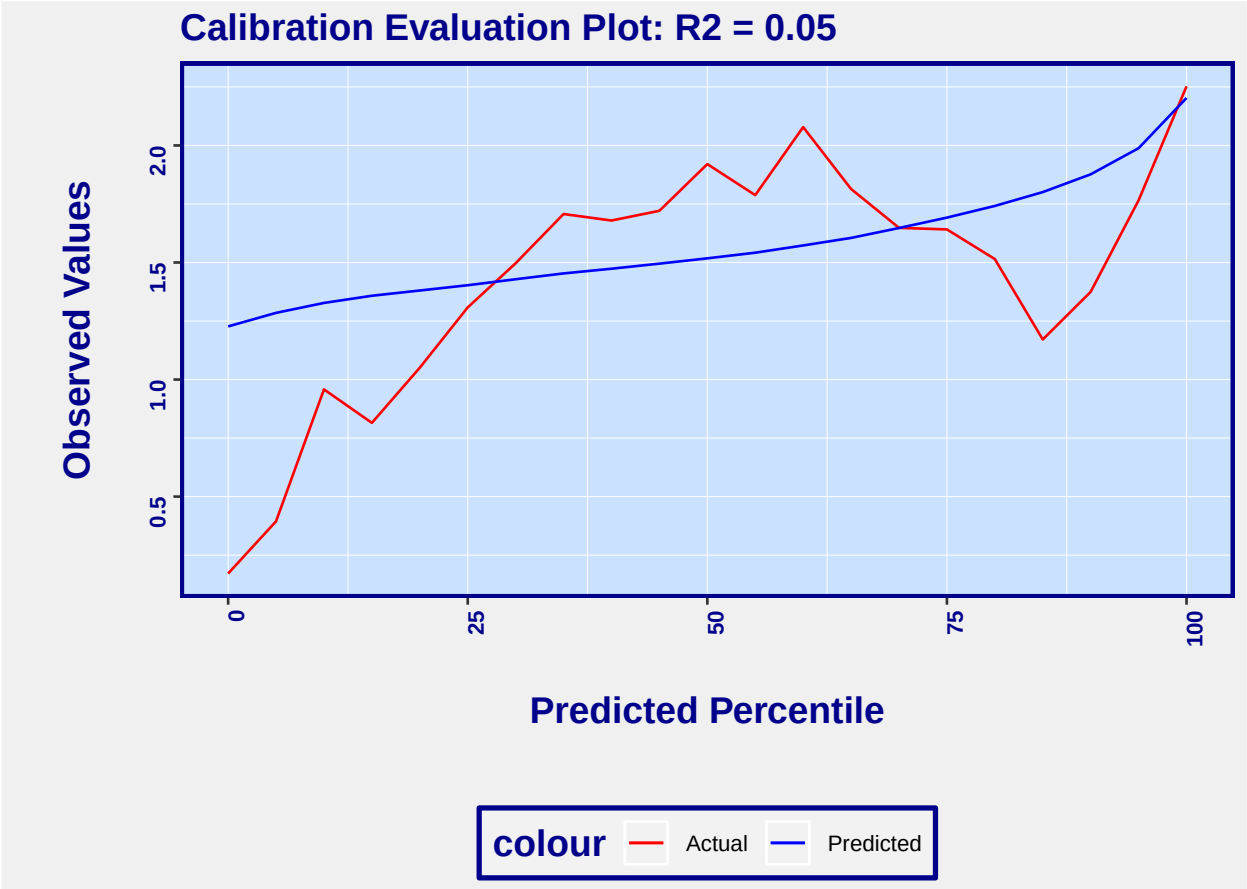
Example of AutoXGBoostHurdleModel()

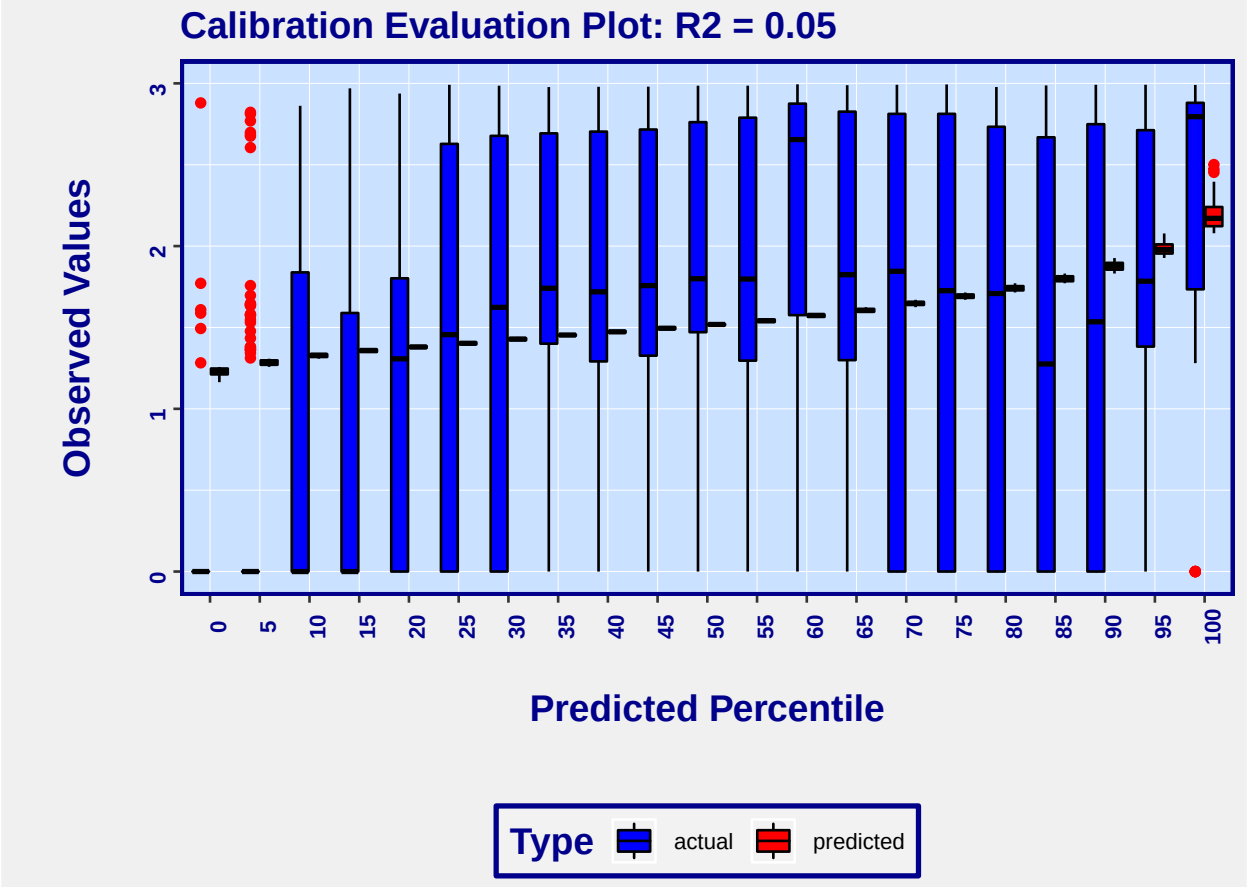
```
library(RemixAutoML)
library(data.table)

# Data Generator Function
dataGen <- function(N = 1000, ID = 1, ZIP = 1, AddDate = FALSE) {
  Correl <- 0.85
  data <- data.table::data.table(Adrian = runif(N))
  data[, x1 := qnorm(Adrian)]
  data[, x2 := runif(N)]
  data[, Independent_Variable1 := log(pnorm(Correl * x1 +
                                              sqrt(1-Correl^2) * qnorm(x2)))]
  data[, Independent_Variable2 := (pnorm(Correl * x1 +
                                           sqrt(1-Correl^2) * qnorm(x2)))]
  data[, Independent_Variable3 := exp(pnorm(Correl * x1 +
                                              sqrt(1-Correl^2) * qnorm(x2)))]
  data[, Independent_Variable4 := exp(exp(pnorm(Correl * x1 +
                                                  sqrt(1-Correl^2) * qnorm(x2))))]
  data[, Independent_Variable5 := sqrt(pnorm(Correl * x1 +
                                               sqrt(1-Correl^2) * qnorm(x2)))]
  data[, Independent_Variable6 := (pnorm(Correl * x1 +
                                           sqrt(1-Correl^2) * qnorm(x2)))^0.10]
  data[, Independent_Variable7 := (pnorm(Correl * x1 +
                                           sqrt(1-Correl^2) * qnorm(x2)))^0.25]
  data[, Independent_Variable8 := (pnorm(Correl * x1 +
                                           sqrt(1-Correl^2) * qnorm(x2)))^0.75]
```

```
  data[, Independent_Variable9 := (pnorm(Correl * x1 +
                                           sqrt(1-Correl^2) * qnorm(x2)))^2]
  data[, Independent_Variable10 := (pnorm(Correl * x1 +
                                            sqrt(1-Correl^2) * qnorm(x2)))^4]
  data[, Independent_Variable11 := as.factor(
    ifelse(Independent_Variable2 < 0.20, "A",
           ifelse(Independent_Variable2 < 0.40, "B",
                  ifelse(Independent_Variable2 < 0.6,  "C",
                         ifelse(Independent_Variable2 < 0.8,  "D", "E")))))]

  # Add date----
  if(AddDate) {
    data <- data[, DateTime := as.Date(Sys.time())]
    data[, temp := 1:.N][, DateTime := DateTime - temp][, temp := NULL]
    data <- data[order(DateTime)]
  }

  # IDcols----
  if(ID == 1) {
    data[, ':=' (x1 = NULL)]
  } else if(ID == 0) {
    data[, ':=' (x1 = NULL, x2 = NULL)]
  }

  # ZIP setup----
  if(ZIP == 1) {
    data[, Adrian := ifelse(Adrian < 0.5, 0, log(Adrian*10))]
  } else if (ZIP == 2) {
    data[, Adrian := ifelse(Adrian < 0.35, 0, ifelse(Adrian < 0.65, log(Adrian*10), log(Adrian*20)))]
  }

  # Return data----
  return(data)
}

# Generate data
data <- dataGen(N = 25000, ID = 2, ZIP = 2, AddDate = FALSE)

# Plot target variable
ggplot2::ggplot(data = data, ggplot2::aes(x = Adrian)) +
  ggplot2::geom_histogram(bins = 25) +
  ggplot2::ggtitle("Histogram of Target Variable Suitable for Hurdle Modeling") +
  RemixAutoML::ChartTheme(Size = 10)

# Build hurdle model
Output <- AutoXGBoostHurdleModel(
  data,
  ValidationData = NULL,
  TestData = NULL,
  Buckets = c(0, 1.6, 2.7),
  TargetColumnName = "Adrian",
  FeatureColNames = 4:ncol(data),
  IDcols = 2:3,
```

```
  TransformNumericColumns = NULL,
  SplitRatios = c(0.7, 0.2, 0.1),
  TreeMethod = "hist",
  NThreads = max(1, parallel::detectCores()-2),
  ModelID = "ModelID",
  Paths = NULL,
  SaveModelObjects = FALSE,
  Trees = 1000,
  GridTune = FALSE,
  MaxModelsInGrid = 1,
  NumOfParDepPlots = 10,
  PassInGrid = NULL)

# Gather output
Output$EvaluationMetrics
#>    Metric MetricValue
#> 1:    MAE      0.9981
#> 2:   MAPE      0.6583
#> 3:    MSE      1.3233
#> 4:   MSLE      1.3233
#> 5:     KL      1.3233
#> 6:     CS      0.7898
#> 7:     R2      0.0390
Output$EvaluationPlot
Output$EvaluationBoxPlot
```


Histogram of Target Variable Suitable for Hurdle Modeling
count
0
2500
5000
7500
0
1
2
3
Adrian


Calibration Evaluation Plot: R2 = 0.05
Observed Values
0.5
1.0
1.5
2.0
0
25
50
75
100
Predicted Percentile
colour
Actual
Predicted



Example of CARMA Suite

Read the blog about them and AutoTS() here: **https://towardsdatascience.com/machine-learning-vs-econometrics-**

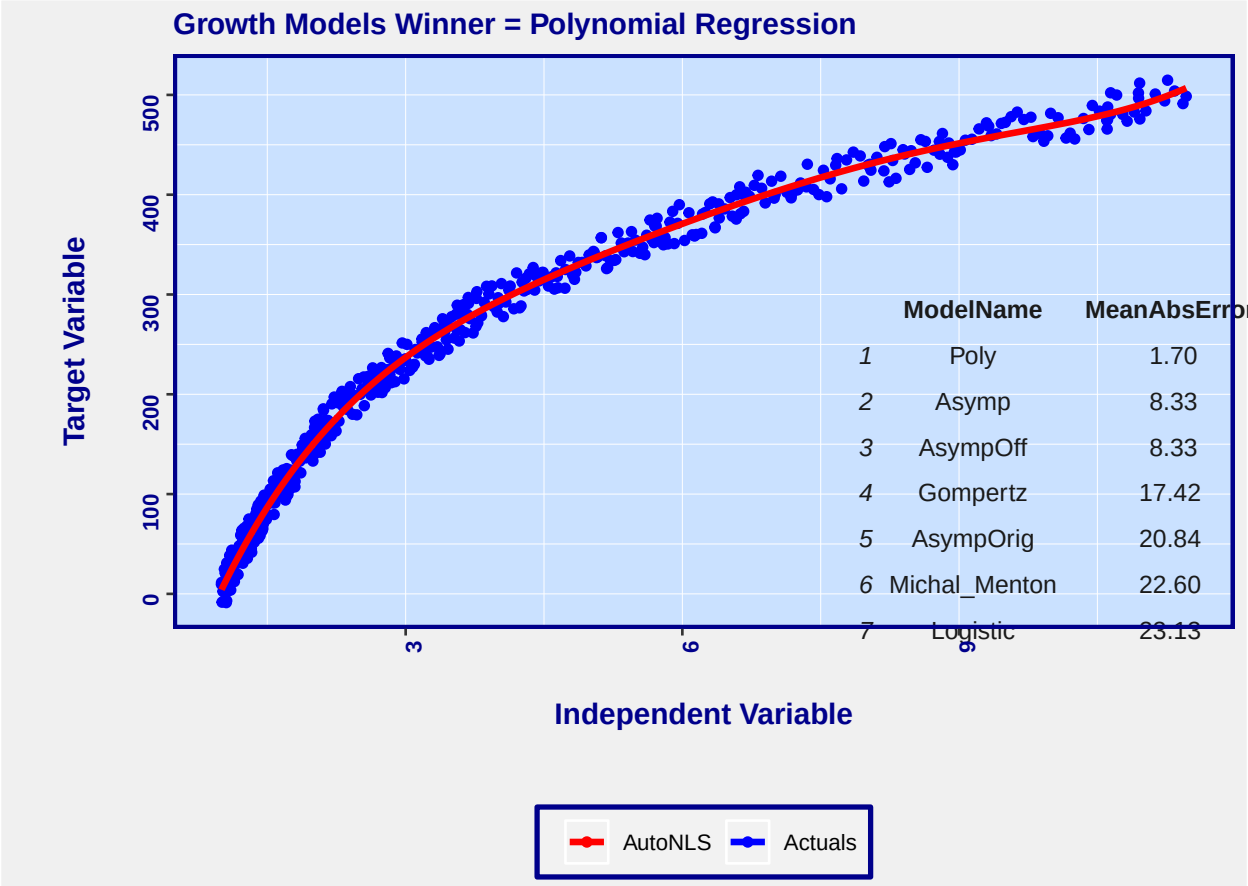
Example of AutoNLS()

Find more demos at **https://www.remyxcourses.com/course?courseid=intro-to-remixautoml-in-r**

```
library(RemixAutoML)
library(data.table)

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

```
    var <- data[i - 1, "Variable"][[1]]
    data.table::set(data,
                    i = i,
                    j = 2L,
                    value = var * (1 + runif(1) / 100))
  }
}

# To keep original values
data1 <- data.table::copy(data)

# Merge and Model data
data11 <- RemixAutoML::AutoNLS(
  data = data,
  y = "Target",
  x = "Variable",
  monotonic = FALSE
)

data2 <- merge(
  data1,
  data11$PredictionData,
  by = "Variable",
  all = FALSE
)

# Model Evaluation Metrics
Metrics <- data11$EvaluationMetrics[, MeanAbsError := round(MeanAbsError, 2)]

# Plot graphs of predicted vs actual
ggplot2::ggplot(data2, ggplot2::aes(x = Variable)) +
  ggplot2::geom_point(
    ggplot2::aes(y = jitter(
      data2[["Target.x"]],
      factor = 100),color = "Actuals")) +
  ggplot2::geom_line(
    ggplot2::aes(y = data2[["Target.y"]],
    color = "AutoNLS"), lwd = 1.25) +
  RemixAutoML::ChartTheme(Size = 12) +
  ggplot2::ggtitle(
    paste0("Growth Models Winner = ",
           data11$ModelName)) +
  ggplot2::ylab("Target Variable") +
  ggplot2::xlab("Independent Variable") +
  ggplot2::annotation_custom(
    gridExtra::tableGrob(
      d = Metrics,
      theme = gridExtra::ttheme_minimal(
        base_size = 10,
        base_colour = "gray10")),
    xmin = 8,
    xmax = 12,
    ymin = -1,
```

```
  ymax = 250) +
ggplot2::scale_colour_manual(
  "",
  breaks = c("AutoNLS", "Actuals"),
  values=c("blue","red"))
```



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 H2O's Generalized Low Rank Model and their KMeans algorithm, with hyper-parameter tuning for both. The function automatically adds the clusters to your data and can save the models for scoring new data with the **AutoH2OScoring** function. 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()
- AutoKMeans()

Demo of ResidualOutliers()

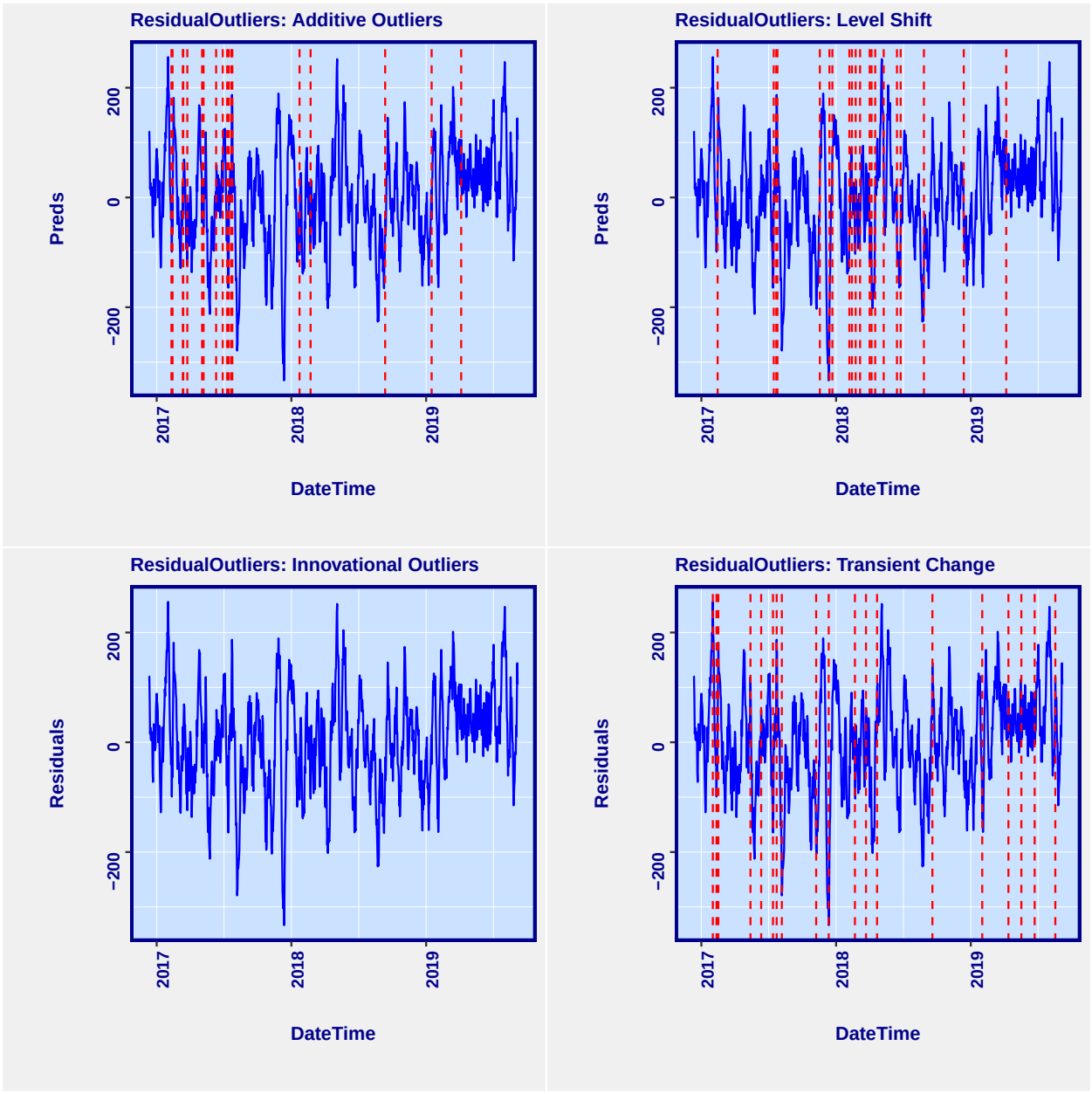
Find more demos at **https://www.remyxcourses.com/course?courseid=intro-to-remixautoml-in-r**

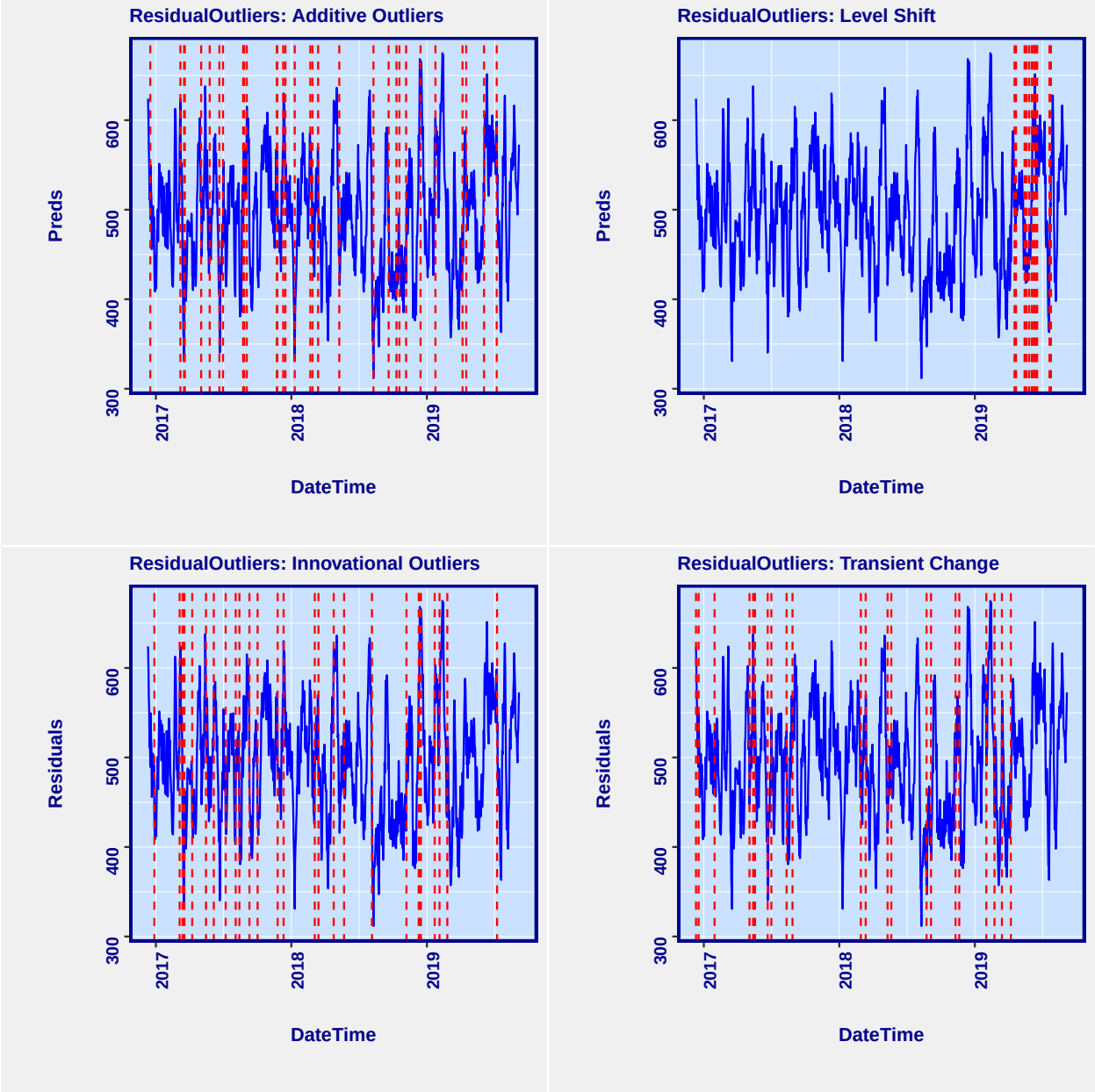
```
# Run on (Target - Predicted)
library(RemixAutoML)
library(data.table)
data <- data.table::data.table(DateTime = as.Date(Sys.time()),
                               Target = as.numeric(stats::filter(rnorm(1000,
                                                                       mean = 50,
                                                                       sd = 20),
                                                                 filter=rep(1,10),
                                                                 circular=TRUE)))
data[, temp := seq(1:1000)][, DateTime := DateTime - temp][, temp := NULL]
data <- data[order(DateTime)]
data[, Predicted := as.numeric(stats::filter(rnorm(1000,
                                                   mean = 50,
                                                   sd = 20),
                                             filter=rep(1,10),
                                             circular=TRUE))]

# Run function and collect results
stuff <- ResidualOutliers(data = data,
                          DateColName = "DateTime",
                          TargetColName = "Target",
                          PredictedColName = "Predicted",
                          TimeUnit = "day",
                          maxN = 5,
                          tstat = 2)
#> Registered S3 method overwritten by 'xts':
#>   method     from
#>   as.zoo.xts zoo
#> Registered S3 method overwritten by 'quantmod':
#>   method            from
#>   as.zoo.data.frame zoo
#> Registered S3 methods overwritten by 'forecast':
#>   method             from
#>   fitted.fracdiff    fracdiff
#>   residuals.fracdiff fracdiff
data     <- stuff$FullData
model    <- stuff$ARIMA_MODEL
outliers <- data[type != "<NA>"]

# Create Plots
p1 <- ggplot2::ggplot(data, ggplot2::aes(x = DateTime)) +
  ggplot2::geom_line(ggplot2::aes(y = Preds),
                     color = "blue") +
  RemixAutoML::ChartTheme(Size = 10) +
  ggplot2::geom_vline(data = outliers[type == "AO", "DateTime"],
                      ggplot2::aes(xintercept = outliers[
```

```
                        type == "AO"][["DateTime"]]),
                      linetype = 8, colour = "red") +
  ggplot2::ggtitle("ResidualOutliers: Additive Outliers")

p2 <- ggplot2::ggplot(data, ggplot2::aes(x = DateTime)) +
  ggplot2::geom_line(ggplot2::aes( y = Residuals),
                     color = "blue") +
 RemixAutoML::ChartTheme(Size = 10) +
  ggplot2::geom_vline(data = outliers[type == "IO", "DateTime"],
                      ggplot2::aes(xintercept = outliers[
                        type == "IO"][["DateTime"]]),
                      linetype = 8, colour = "red") +
  ggplot2::ggtitle("ResidualOutliers: Innovational Outliers")

p3 <- ggplot2::ggplot(data, ggplot2::aes(x = DateTime)) +
  ggplot2::geom_line(ggplot2::aes(y = Preds),
                     color = "blue") +
  RemixAutoML::ChartTheme(Size = 10) +
  ggplot2::geom_vline(data = outliers[type == "LS", "DateTime"],
                      ggplot2::aes(xintercept = outliers[
                        type == "LS"][["DateTime"]]),
                      linetype = 8, colour = "red") +
  ggplot2::ggtitle("ResidualOutliers: Level Shift")

p4 <- ggplot2::ggplot(data, ggplot2::aes(x = DateTime)) +
  ggplot2::geom_line(ggplot2::aes( y = Residuals),
                     color = "blue") +
  RemixAutoML::ChartTheme(Size = 10) +
  ggplot2::geom_vline(data = outliers[type == "TC", "DateTime"],
                      ggplot2::aes(xintercept = outliers[
                        type == "TC"][["DateTime"]]),
                      linetype = 8, colour = "red") +
  ggplot2::ggtitle("ResidualOutliers: Transient Change")

# Print plots
RemixAutoML::multiplot(plotlist = list(p1,p2,p3,p4), cols = 2)

# Run on Target data
data <- data.table::data.table(DateTime = as.Date(Sys.time()),
                               Target = as.numeric(stats::filter(rnorm(1000,
                                                                       mean = 50,
                                                                       sd = 20),
                                                                 filter=rep(1,10),
                                                                 circular=TRUE)))
data[, temp := seq(1:1000)][, DateTime := DateTime - temp][, temp := NULL]
data <- data[order(DateTime)]
data[, Predicted := as.numeric(stats::filter(rnorm(1000,
                                                   mean = 50,
                                                   sd = 20),
                                             filter=rep(1,10),
                                             circular=TRUE))]

# Run function and collect results
```

```
stuff    <- ResidualOutliers(data = data,
                             DateColName = "DateTime",
                             TargetColName = "Target",
                             PredictedColName = NULL,
                             TimeUnit = "day",
                             maxN = 5,
                             tstat = 2)
data     <- stuff$FullData
model    <- stuff$ARIMA_MODEL
outliers <- data[type != "<NA>"]

# Create Plots
p11 <- ggplot2::ggplot(data, ggplot2::aes(x = DateTime)) +
  ggplot2::geom_line(ggplot2::aes(y = Preds),
                     color = "blue") +
  RemixAutoML::ChartTheme(Size = 10) +
  ggplot2::geom_vline(data = outliers[type == "AO", "DateTime"],
                      ggplot2::aes(xintercept = outliers[
                        type == "AO"][["DateTime"]]),
                      linetype = 8, colour = "red") +
  ggplot2::ggtitle("ResidualOutliers: Additive Outliers")

p22 <- ggplot2::ggplot(data, ggplot2::aes(x = DateTime)) +
  ggplot2::geom_line(ggplot2::aes( y = Residuals),
                     color = "blue") +
  RemixAutoML::ChartTheme(Size = 10) +
  ggplot2::geom_vline(data = outliers[type == "IO", "DateTime"],
                      ggplot2::aes(xintercept = outliers[
                        type == "IO"][["DateTime"]]),
                      linetype = 8, colour = "red") +
  ggplot2::ggtitle("ResidualOutliers: Innovational Outliers")

p33 <- ggplot2::ggplot(data, ggplot2::aes(x = DateTime)) +
  ggplot2::geom_line(ggplot2::aes(y = Preds),
                     color = "blue") +
  RemixAutoML::ChartTheme(Size = 10) +
  ggplot2::geom_vline(data = outliers[type == "LS", "DateTime"],
                      ggplot2::aes(xintercept = outliers[
                        type == "LS"][["DateTime"]]),
                      linetype = 8, colour = "red") +
  ggplot2::ggtitle("ResidualOutliers: Level Shift")

p44 <- ggplot2::ggplot(data, ggplot2::aes(x = DateTime)) +
  ggplot2::geom_line(ggplot2::aes( y = Residuals),
                     color = "blue") +
  RemixAutoML::ChartTheme(Size = 10) +
  ggplot2::geom_vline(data = outliers[type == "TC", "DateTime"],
                      ggplot2::aes(xintercept = outliers[
                        type == "TC"][["DateTime"]]),
                      linetype = 8, colour = "red") +
  ggplot2::ggtitle("ResidualOutliers: Transient Change")

# Print plots
```

```
RemixAutoML::multiplot(plotlist = list(p11,p22,p33,p44), cols = 2)
```





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:

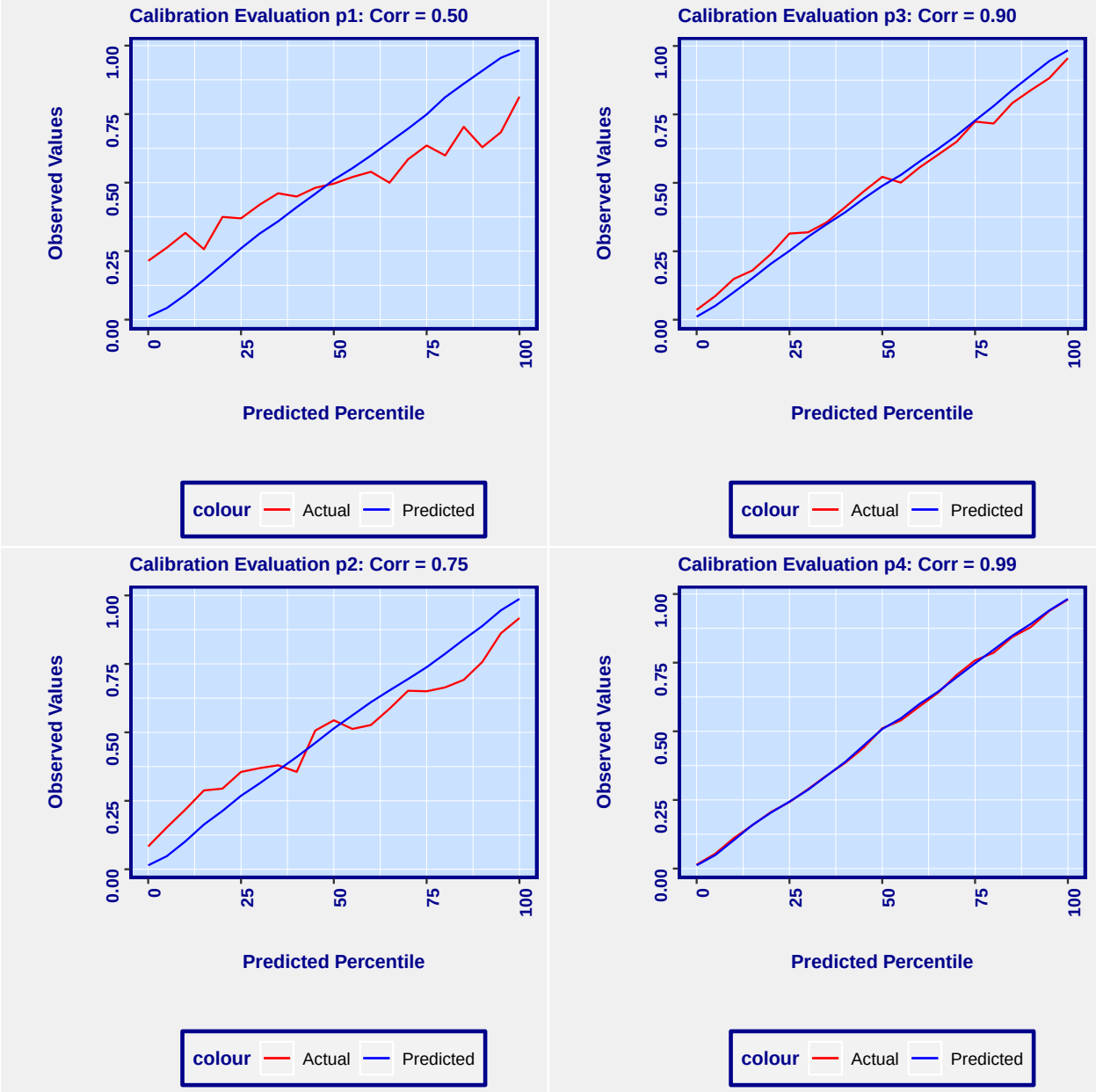
- EvalPlot()
- ParDepCalPlots()
- threshOptim()
- RedYellowGreen()

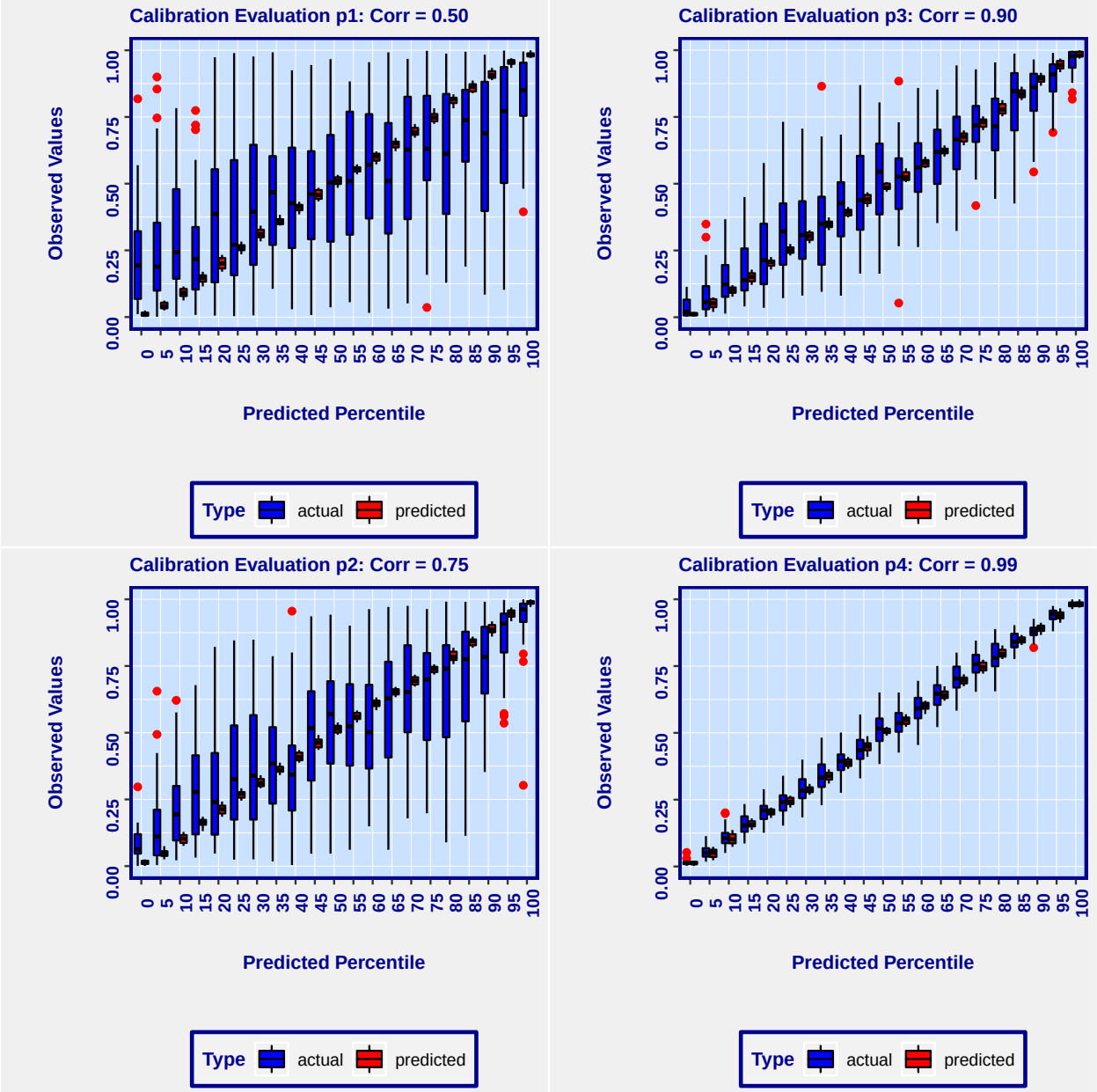
Demo of EvalPlot()

Find more demos at **https://www.remyxcourses.com/course?courseid=intro-to-remixautoml-in-r**

```
library(RemixAutoML)
library(data.table)

# 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 <- RemixAutoML::EvalPlot(data = data1,
                            PredictionColName = "predict",
                            TargetColName = "target",
                            GraphType = "calibration",
                            PercentileBucket = 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 <- RemixAutoML::EvalPlot(data = data2,
                            PredictionColName = "predict",
                            TargetColName = "target",
                            GraphType = "calibration",
                            PercentileBucket = 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 <- RemixAutoML::EvalPlot(data = data3,
                            PredictionColName = "predict",
                            TargetColName = "target",
                            GraphType = "calibration",
```

```
                            PercentileBucket = 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 <- RemixAutoML::EvalPlot(data = data4,
                            PredictionColName = "predict",
                            TargetColName = "target",
                            GraphType = "calibration",
                            PercentileBucket = 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 <- RemixAutoML::EvalPlot(data = data1,
                            PredictionColName = "predict",
                            TargetColName = "target",
                            GraphType = "boxplot",
                            PercentileBucket = 0.05)
p1 <- p1 + ggplot2::ggtitle("Calibration Evaluation p1: Corr = 0.50") +
  RemixAutoML::ChartTheme(Size = 10)

p2 <- RemixAutoML::EvalPlot(data = data2,
                            PredictionColName = "predict",
                            TargetColName = "target",
                            GraphType = "boxplot",
                            PercentileBucket = 0.05)
p2 <- p2 + ggplot2::ggtitle("Calibration Evaluation p2: Corr = 0.75") +
  RemixAutoML::ChartTheme(Size = 10)

p3 <- RemixAutoML::EvalPlot(data = data3,
                            PredictionColName = "predict",
                            TargetColName = "target",
                            GraphType = "boxplot",
                            PercentileBucket = 0.05)
p3 <- p3 + ggplot2::ggtitle("Calibration Evaluation p3: Corr = 0.90") +
  RemixAutoML::ChartTheme(Size = 10)

p4 <- RemixAutoML::EvalPlot(data = data4,
                            PredictionColName = "predict",
                            TargetColName = "target",
                            GraphType = "boxplot",
                            PercentileBucket = 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.

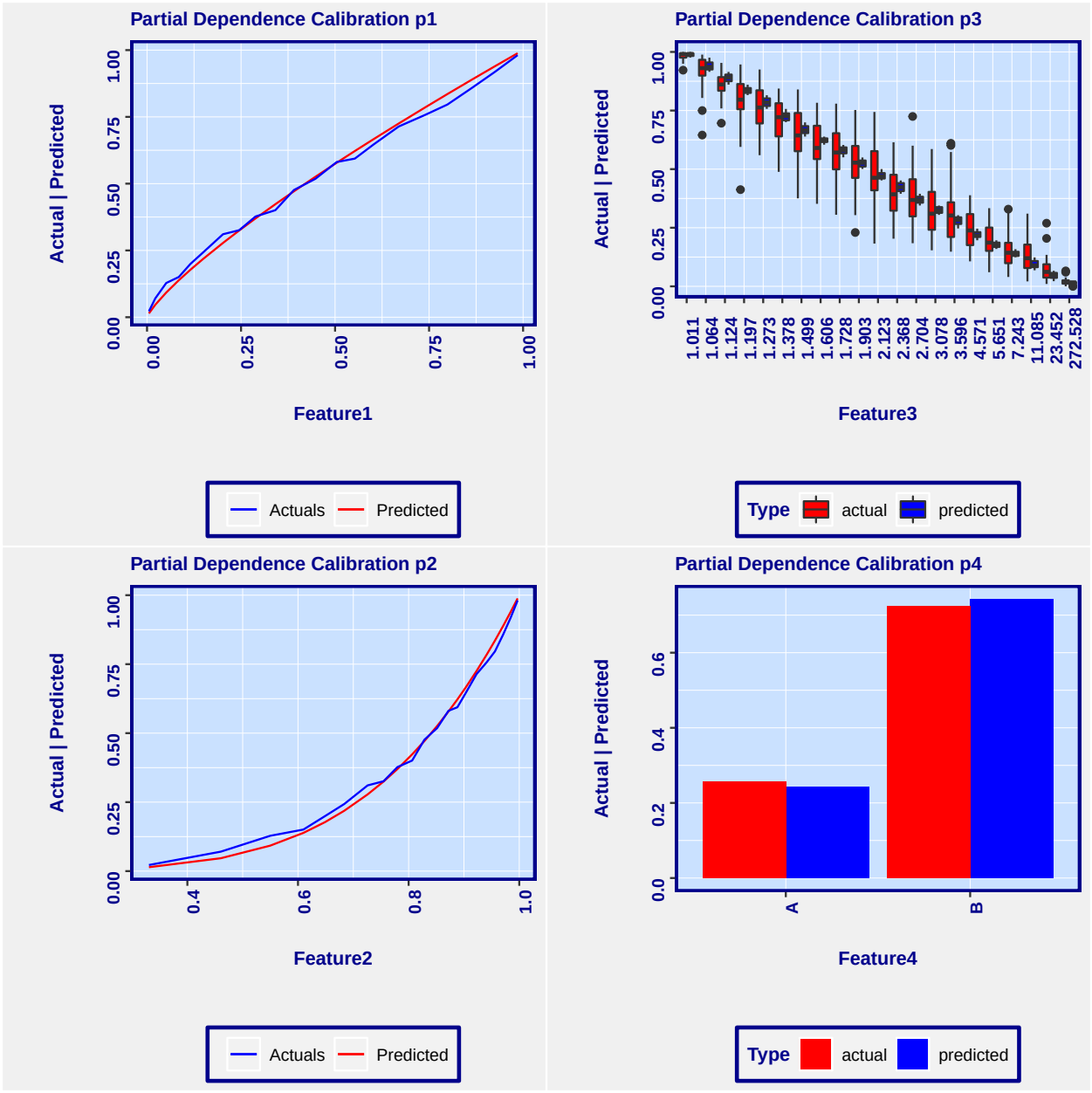
The cost sensitive optimizaition functions provide the user the ability to generate utility-optimized thresholds for classification tasks. There are two of these functions: one for generating a single threshold based on the values supplied to your cost confusion matrix outcomes and the second one provides 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.

Demo of ParDepCalPlots()

Find more demos at **https://www.remyxcourses.com/course?courseid=intro-to-remixautoml-in-r**

```
library(RemixAutoML)
library(data.table)

# 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))]
  Validation[, Feature1 := (pnorm(Correlation * x1 +
                                    sqrt(1 - Correlation ^2) * qnorm(x2)))^1.25]
  Validation[, Feature2 := (pnorm(Correlation * x1 +
                                    sqrt(1 - Correlation ^2) * qnorm(x2)))^0.25]
  Validation[, Feature3 := (pnorm(Correlation * x1 +
                                    sqrt(1 - Correlation ^2) * qnorm(x2)))^(-1)]
  Validation[, Feature4 := pnorm(Correlation * x1 +
                                    sqrt(1 - Correlation ^2) * qnorm(x2))]
  Validation[, Feature4 := ifelse(Feature4 < 0.5, "A",
                                  ifelse(Feature4 < 1, "B",
                                         ifelse(Feature4 < 1.5, "C","D")))]
  return(Validation)
}

# Store data sets
data1 <- dataGen(Correlation = 0.95)

# Generate EvalPlots (calibration)
p1 <- RemixAutoML::ParDepCalPlots(data = data1,
                                  PredictionColName = "predict",
                                  TargetColName = "target",
                                  IndepVar = "Feature1",
                                  GraphType = "calibration",
                                  PercentileBucket = 0.05,
                                  Function = function(x) mean(x,
                                                              na.rm = TRUE),
                                  FactLevels = 10)
p1 <- p1 + ggplot2::ggtitle("Partial Dependence Calibration p1") +
  RemixAutoML::ChartTheme(Size = 10)

p2 <- RemixAutoML::ParDepCalPlots(data = data1,
                                  PredictionColName = "predict",
                                  TargetColName = "target",
                                  IndepVar = "Feature2",
                                  GraphType = "calibration",
                                  PercentileBucket = 0.05,
                                  Function = function(x) mean(x,
                                                              na.rm = TRUE),
                                  FactLevels = 10)
p2 <- p2 + ggplot2::ggtitle("Partial Dependence Calibration p2") +
```

```
  RemixAutoML::ChartTheme(Size = 10)

p3 <- RemixAutoML::ParDepCalPlots(data = data1,
                                  PredictionColName = "predict",
                                  TargetColName = "target",
                                  IndepVar = "Feature3",
                                  GraphType = "boxplot",
                                  PercentileBucket = 0.05,
                                  Function = function(x) mean(x,
                                                              na.rm = TRUE),
                                  FactLevels = 10)
p3 <- p3 + ggplot2::ggtitle("Partial Dependence Calibration p3") +
  RemixAutoML::ChartTheme(Size = 10)

p4 <- RemixAutoML::ParDepCalPlots(data = data1,
                                  PredictionColName = "predict",
                                  TargetColName = "target",
                                  IndepVar = "Feature4",
                                  GraphType = "calibration",
                                  PercentileBucket = 0.05,
                                  Function = function(x) mean(x,
                                                              na.rm = TRUE),
                                  FactLevels = 10)
p4 <- p4 + ggplot2::ggtitle("Partial Dependence Calibration p4") +
  RemixAutoML::ChartTheme(Size = 10)
RemixAutoML::multiplot(plotlist = list(p1,p2,p3,p4), cols = 2)
```



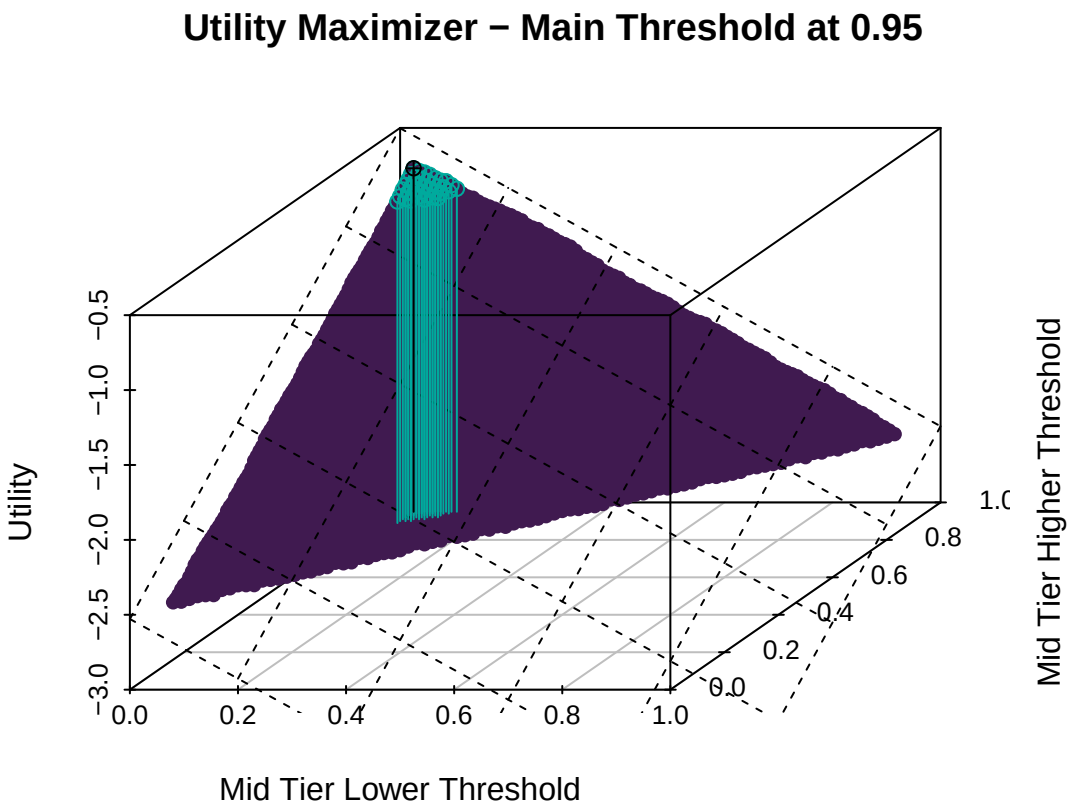
Demo of RedYellowGreen()

Find more demos at **https://www.remyxcourses.com/course?courseid=intro-to-remixautoml-in-r**

```
library(RemixAutoML)
library(data.table)
Correl <- 0.70
  data <- data.table::data.table(target = runif(1000))
  data[, x1 := qnorm(target)]
  data[, x2 := runif(1000)]
  data[, predict := pnorm(Correl * x1 +
                          sqrt(1 - Correl ^2) *
                          qnorm(x2))]
```

```
data[, target := as.numeric(ifelse(target < 0.5, 0, 1))]
data <- RemixAutoML::RedYellowGreen(
  data,
  PredictColNumber  = 4,
  ActualColNumber   = 1,
  TruePositiveCost  = 0,
  TrueNegativeCost  = 0,
  FalsePositiveCost = -3,
  FalseNegativeCost = -2,
  MidTierCost       = -0.5,
  Cores             = 1,
  Boundaries        = c(0.05,0.95)
)

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

TPP	TNP	FPP	FNP	MTDN	MTC	Threshold	MTLT	MTHT	Utility
0	0	-3	-2	TRUE	-0.5	0.95	0.05	0.95	-0.70703
0	0	-3	-2	TRUE	-0.5	0.95	0.06	0.95	-0.72512
0	0	-3	-2	TRUE	-0.5	0.94	0.05	0.94	-0.73517
0	0	-3	-2	TRUE	-0.5	0.95	0.07	0.95	-0.73517
0	0	-3	-2	TRUE	-0.5	0.95	0.08	0.95	-0.75125
0	0	-3	-2	TRUE	-0.5	0.94	0.06	0.94	-0.75326
0	0	-3	-2	TRUE	-0.5	0.93	0.05	0.93	-0.75929
0	0	-3	-2	TRUE	-0.5	0.94	0.07	0.94	-0.76331
0	0	-3	-2	TRUE	-0.5	0.95	0.09	0.95	-0.77336
0	0	-3	-2	TRUE	-0.5	0.93	0.06	0.93	-0.77738



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()`
- `Partial_DT_GDL_Feature_Engineering()`
- `Partial_DT_GDL_Feature_Engineering2()`
- `Scoring_GDL_Feature_Engineering()`
- `AutoWord2VecModeler()`
- `CreateCalendarVariables()`
- `CreateHolidayVariables()`
- `ModelDataPrep()`
- `DummifyDT()`
- `AutoDataPartition()`
- `AutoTransformationCreate()`
- `AutoTransformationScore()`

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 group (**Partial_GDL_** in which there are two versions) is used for cases where you don't need to generate the features across the entire data set but rather for a subset of records. This comes up typically in scoring environments where you only need the features created for one or a few records (not all records). The fourth function (**Scoring_GDL_**) is currently used in the AutoCatBoostCARMA() function for generating the lags and moving average features for one record at a time.

DT_GDL_Feature_Engineering and Scoring_GDL_Feature_Engineering Demo (simulated data)

Find more demos at **https://www.remyxcourses.com/course?courseid=intro-to-remixautoml-in-r**

```
library(RemixAutoML)
library(data.table)

# 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 <- RemixAutoML::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",
```

```
  SimpleImpute   = TRUE)

# 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)]

# Use WindowingLag = 1 to build moving averages off of the lag1 Target Variable to eliminate forward le
ScoringData <- RemixAutoML::Scoring_GDL_Feature_Engineering(
  ScoringData,
  lags           = c(seq(1, 5, 1)),
  periods        = c(3, 5, 10, 15, 20, 25),
  statsNames     = c("MA"),
  targets        = c("Target"),
  groupingVars   = c("GroupVariable"),
  sortDateName   = c("DateTime"),
  timeDiffTarget = c("Time_Gap"),
  timeAgg        = "days",
  WindowingLag   = 1,
  Type           = "Lag",
  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
522.1473	389.8088	519.7938
389.8088	590.9787	562.0782
590.9787	578.5939	542.6682
578.5939	516.6620	513.3481
516.6620	532.7487	529.9783
532.7487	490.6335	522.6302
490.6335	566.5528	530.7392
566.5528	510.7043	478.9083
510.7043	514.9604	479.6384
514.9604	411.0602	489.0341

```
# 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
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

The **AutoWord2VecModeler()** 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 H2O'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 **AutoH2OModeler()** 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. The **AutoDataPartition()** can build out any number of data sets under three schemes: random sampling, timeseries sampling, and time sampling. The random and timeseries allows for stratified sampling while time does not. The **AutoTransformationCreate()** and **AutoTransformationScore()** functions are for automatically applying the best transformation to any of your numeric features or target variable along with saving that data or passing the output file on to the scoring function for automatic inverse transforming for scroring settings.

Miscellaneous Functions

Functions include:

- `AutoWordFreq()`
- `AutoH2OTextPrepScoring()`
- `ProblematicFeatures()`
- `ProblematicRecords()`
- `ChartTheme()`
- `RemixTheme()`
- `multiplot()`
- `PrintObjectsSize()`
- `percRank()`

The **AutoWordFreq** 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 **AutoH2OTextPrepScoring** will automatically prepare your text data for scoring. This function is run internally in the **AutoH2OScoring** function but you can utilize it outside for other purposes. The **ProblematicFeatures** identified problematic columns for machine learning. **ProblematicRecords** finds problematic rows in your data set that you should investigate further. 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. **AutoRecomDataCreate** will turn your transactional data set into a binary ratings matrix fast.

```
library(data.table)

# Create Some Data
data <- data.table::data.table(
DESCR = c("Gru, Gru, Gru, Gru, Gru, Gru, Gru, Gru, Gru, Gru, Gru, Gru, Gru,
           Urkle, Urkle, Urkle, Urkle, Urkle, Urkle, Urkle, Gru, Gru, Gru,
           bears, bears, bears, bears, bears, bears, smug, smug, smug, smug,
           smug, smug, smug, smug, smug, smug, smug, smug, smug, smug, smug,
           eats, eats, eats, eats, eats, eats, beats, beats, beats, beats,
           beats, beats, beats, beats, beats, beats, beats, science, science,
           Dwigt, Dwigt, Dwigt, Dwigt, Dwigt, Dwigt, Dwigt, Dwigt, Dwigt, Dwigt,
           Schrute, Schrute, Schrute, Schrute, Schrute, Schrute, Schrute,
           James, James, James, James, James, James, James, James, James, James,
           Halpert, Halpert, Halpert, Halpert, Halpert, Halpert, Halpert, Halpert"))

# Run function
data <- AutoWordFreq(data,
                     TextColName = "DESCR",
                     GroupColName = NULL,
                     GroupLevel = NULL,
                     RemoveEnglishStopwords = FALSE,
                     Stemming = FALSE,
                     StopWords = c("Bla"))
#>        word freq
#>  1:     gru   16
#>  2:    smug   15
```

```
#>  3:   beats   11
#>  4:   dwigt   10
#>  5:   james   10
#>  6: halpert    8
#>  7: schrute    7
#>  8:   urkle    7
#>  9:   bears    6
#> 10:    eats    6
#> NULL

# View word frequency table
print(data)
#>         word freq
#>  1:      gru   16
#>  2:    smug   15
#>  3:   beats   11
#>  4:   dwigt   10
#>  5:   james   10
#>  6: halpert    8
#>  7: schrute    7
#>  8:   urkle    7
#>  9:   bears    6
#> 10:    eats    6
#> 11: science    2
```