As advanced machine learning algorithms are gaining acceptance across many organizations and domains, machine learning interpretability is growing in importance to help extract insight and clarity regarding how these algorithms are performing and why one prediction is made over another. There are many methodologies to interpret machine learning results (i.e. variable importance via permutation, partial dependence plots, local interpretable model-agnostic explanations), and many machine learning R packages implement their own versions of one or more methodologies. However, some recent R packages that focus purely on ML interpretability agnostic to any specific ML algorithm are gaining popularity. One such package is DALEX and this post covers what this package does (and does not do) so that you can determine if it should become part of your preferred machine learning toolbox.

We implement machine learning models using H2O, a high performance ML toolkit. Let’s see how DALEX and H2O work together to get the best of both worlds with high performance and feature explainability!

**Learning Trajectory**

We’ll cover the following topics on DALEX in this article:

1. Advantages & disadvantages: a quick breakdown of what DALEX does and does not do.
2. Replication requirements: what you’ll need to reproduce the analysis.
3. DALEX procedures: necessary functions for downstream explainers.
4. Residual diagnostics: understanding and comparing errors.
5. Variable importance: permutation based importance score.
6. Predictor-response relationship: PDP and ALE plots.
7. Local interpretation: explanations for a single prediction.

**DALEX and H2O: Machine Learning Model Interpretability And Feature Explanation**

By Brad Boehmke, Director of Data Science at 84.51°

**1.0 Advantages & disadvantages**

DALEX is an R package with a set of tools that help to provide **D**escriptive m**A**chine **L**earning **EX**planations ranging from global to local interpretability methods. In particular, it makes comparing performance across multiple models convenient. However, as is, there are some problems with this package scaling to wider data sets commonly used by organizations. The following provides a quick list of its pros and cons:

**Advantages**

* ML model and package agnostic: can be used for any supervised regression and binary classification ML model where you can customize the format of the predicted output.
* Provides convenient approaches to compare results across multiple models.
* Residual diagnostics: allows you to compare residual distributions.
* Variable importance: uses a permutation-based approach for variable importance, which is model agnostic, and accepts any loss function to assess importance.
* Partial dependence plots: leverages the pdp package.
* Provides an alternative to PDPs for categorical predictor variables (merging path plots).
* Includes a unique and intuitive approach for local intepretation.

**Disadvantages**

* Some functions do not scale well to wide data (many predictor variables)
* Currently only supports regression and binary classification problems (i.e. no multinomial support).
* Only provides permutation-based variable importance scores (which become slow as number of features increase).
* PDP plots can only be performed one variable at a time (options for two-way interaction PDP plots).
* Does not provide ICE curves.
* Does not provide alternative local interpretation algorithms (i.e. LIME, SHAP values).

**2.0 Replication requirements**

We leverage the following packages:

# load required packages

library(rsample)

library(dplyr)

library(purrr)

library(ggplot2)

library(h2o)

library(DALEX)

# initialize h2o session

h2o.no\_progress()

h2o.init()

##

## H2O is not running yet, starting it now...

##

## Note: In case of errors look at the following log files:

## C:\Users\mdancho\AppData\Local\Temp\Rtmp6f3d5n/h2o\_mdancho\_started\_from\_r.out

## C:\Users\mdancho\AppData\Local\Temp\Rtmp6f3d5n/h2o\_mdancho\_started\_from\_r.err

##

##

## Starting H2O JVM and connecting: . Connection successful!

##

## R is connected to the H2O cluster:

## H2O cluster uptime: 7 seconds 710 milliseconds

## H2O cluster timezone: America/New\_York

## H2O data parsing timezone: UTC

## H2O cluster version: 3.19.0.4208

## H2O cluster version age: 5 months and 3 days !!!

## H2O cluster name: H2O\_started\_from\_R\_mdancho\_dpc744

## H2O cluster total nodes: 1

## H2O cluster total memory: 3.51 GB

## H2O cluster total cores: 4

## H2O cluster allowed cores: 4

## H2O cluster healthy: TRUE

## H2O Connection ip: localhost

## H2O Connection port: 54321

## H2O Connection proxy: NA

## H2O Internal Security: FALSE

## H2O API Extensions: Algos, AutoML, Core V3, Core V4

## R Version: R version 3.4.4 (2018-03-15)

To demonstrate model visualization techniques we’ll use the employee attrition data that has been included in the rsample package. This demonstrates a binary classification problem (“Yes” vs. “No”) but the same process that you’ll observe can be used for a regression problem.

To demonstrate DALEX’s capabilities we’ll use the employee attrition data that has been included in the rsample package. This demonstrates a binary classification problem (“Yes” vs. “No”) but the same process that you’ll observe can be used for a regression problem.  
I perform a few house cleaning tasks on the data prior to converting to an h2o object and splitting.

**NOTE**: To use some of DALEX’s functions, categorical predictor variables need to be converted to factors. Also, I force ordered factors to be unordered as h2o does not support ordered categorical variables.

# classification data

df <- rsample::attrition %>%

mutate\_if(is.ordered, factor, ordered = FALSE) %>%

mutate(Attrition = recode(Attrition, "Yes" = "1", "No" = "0") %>% factor(levels = c("1", "0")))

# convert to h2o object

df.h2o <- as.h2o(df)

# create train, validation, and test splits

set.seed(123)

splits <- h2o.splitFrame(df.h2o, ratios = c(.7, .15), destination\_frames = c("train","valid","test"))

names(splits) <- c("train","valid","test")

# variable names for resonse & features

y <- "Attrition"

x <- setdiff(names(df), y)

We will explore how to visualize a few of the more common machine learning algorithms implemented with h2o. For brevity I train default models and do not emphasize hyperparameter tuning. The following produces a regularized logistic regression, random forest, and gradient boosting machine models; all of which provide AUCs ranging between .75-.79. Although these models have distinct AUC scores, our objective is to understand how these models come to this conclusion in similar or different ways based on underlying logic and data structure.

# elastic net model

glm <- h2o.glm(

x = x,

y = y,

training\_frame = splits$train,

validation\_frame = splits$valid,

family = "binomial",

seed = 123

)

# random forest model

rf <- h2o.randomForest(

x = x,

y = y,

training\_frame = splits$train,

validation\_frame = splits$valid,

ntrees = 1000,

stopping\_metric = "AUC",

stopping\_rounds = 10,

stopping\_tolerance = 0.005,

seed = 123

)

# gradient boosting machine model

gbm <- h2o.gbm(

x = x,

y = y,

training\_frame = splits$train,

validation\_frame = splits$valid,

ntrees = 1000,

stopping\_metric = "AUC",

stopping\_rounds = 10,

stopping\_tolerance = 0.005,

seed = 123

)

# model performance

h2o.auc(glm, valid = TRUE)

## [1] 0.7870935

h2o.auc(rf, valid = TRUE)

## [1] 0.7681021

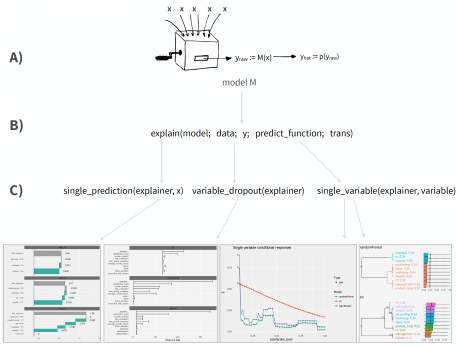
h2o.auc(gbm, valid = TRUE)

## [1] 0.7468242

**3.0 DALEX procedures**

The DALEX architecture can be split into three primary operations:

1. Any supervised regression or binary classification model with defined input (*X*) and output (*Y*) where the output can be customized to a defined format can be used.
2. The machine learning model is converted to an “explainer” object via DALEX::explain(), which is just a list that contains the training data and meta data on the machine learning model.
3. The explainer object can be passed onto multiple functions that explain different components of the given model.

[[](https://pbiecek.github.io/DALEX_docs/2-architecture.html#architecture)](https://pbiecek.github.io/DALEX_docs/2-architecture.html#architecture)

DALEX Application Process and Architecture

Although DALEX does have native support for some ML model objects (i.e. lm, randomForest), it does not have native many of the preferred ML packages produced more recently (i.e. h2o, xgboost, ranger). To make DALEX compatible with these objects, we need three things:

1. x\_valid: Our feature set needs to be in its original form not as an h2o object.
2. y\_valid: Our response variable needs to be a numeric vector. For regression problems this is simple, as it will already be in this format. For binary classification this requires you to convert the responses to 0/1.
3. pred: a custom predict function that returns a **vector** of numeric values. For binary classification problems, this means extracting the probability of the response.

# convert feature data to non-h2o objects

x\_valid <- as.data.frame(splits$valid)[, x]

# make response variable numeric binary vector

y\_valid <- as.vector(as.numeric(as.character(splits$valid$Attrition)))

head(y\_valid)

## [1] 0 0 0 0 0 0

# create custom predict function

pred <- function(model, newdata) {

results <- as.data.frame(h2o.predict(model, as.h2o(newdata)))

return(results[[3L]])

}

pred(rf, x\_valid) %>% head()

## [1] 0.18181818 0.27272727 0.06060606 0.54545455 0.03030303 0.42424242

Once you have these three components, you can now create your explainer objects for each ML model. Considering I used a validation set to compute the AUC, we want to use that same validation set for ML interpretability.

# elastic net explainer

explainer\_glm <- explain(

model = glm,

data = x\_valid,

y = y\_valid,

predict\_function = pred,

label = "h2o glm"

)

# random forest explainer

explainer\_rf <- explain(

model = rf,

data = x\_valid,

y = y\_valid,

predict\_function = pred,

label = "h2o rf"

)

# GBM explainer

explainer\_gbm <- explain(

model = gbm,

data = x\_valid,

y = y\_valid,

predict\_function = pred,

label = "h2o gbm"

)

# example of explainer object

class(explainer\_glm)

## [1] "explainer"

summary(explainer\_glm)

## Length Class Mode

## model 1 H2OBinomialModel S4

## data 30 data.frame list

## y 233 -none- numeric

## predict\_function 1 -none- function

## link 1 -none- function

## class 1 -none- character

## label 1 -none- character

**4.0 Residual diagnostics**

As we saw earlier, the GLM model had the highest AUC followed by the random forest model then GBM. However, a single accuracy metric can be a poor indicator of performance. Assessing residuals of predicted versus actuals can allow you to identify where models deviate in their predictive accuracy. We can use DALEX::model\_performance to compute the predictions and residuals. Printing the output returns residual quantiles and plotting the output allows for easy comparison of absolute residual values across models.

In this example, the residuals are comparing the probability of attrition to the binary attrition value (1-yes, 0-no). Looking at the quantiles you can see that the median residuals are lowest for the GBM model. And looking at the boxplots you can see that the GBM model also had the lowest median ***absolute*** residual value. Thus, although the GBM model had the lowest AUC score, it actually performs best when considering the median absoluate residuals. However, you can also see a higher number of residuals in the tail of the GBM residual distribution (left plot) suggesting that there may be a higher number of large residuals compared to the GLM model. This helps to illustrate how your residuals behave similarly and differently across models.

# compute predictions & residuals

resids\_glm <- model\_performance(explainer\_glm)

resids\_rf <- model\_performance(explainer\_rf)

resids\_gbm <- model\_performance(explainer\_gbm)

# assess quantiles for residuals

resids\_glm

## 0% 10% 20% 30% 40%

## -0.99155845 -0.70432615 0.01281214 0.03402030 0.06143281

## 50% 60% 70% 80% 90%

## 0.08362550 0.10051641 0.12637877 0.17583980 0.22675709

## 100%

## 0.47507569

resids\_rf

## 0% 10% 20% 30% 40%

## -0.96969697 -0.66666667 0.00000000 0.03030303 0.06060606

## 50% 60% 70% 80% 90%

## 0.09090909 0.12121212 0.15151515 0.18181818 0.27272727

## 100%

## 0.66666667

resids\_gbm

## 0% 10% 20% 30% 40%

## -0.96307337 -0.75623698 0.03258538 0.04195091 0.05344621

## 50% 60% 70% 80% 90%

## 0.06382511 0.07845749 0.09643740 0.11312648 0.18169305

## 100%

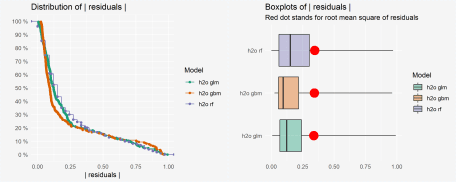
## 0.66208105

# create comparison plot of residuals for each model

p1 <- plot(resids\_glm, resids\_rf, resids\_gbm)

p2 <- plot(resids\_glm, resids\_rf, resids\_gbm, geom = "boxplot")

gridExtra::grid.arrange(p1, p2, nrow = 1)



**5.0 Variable importance**

An important task in ML interpretation is to understand which predictor variables are relatively influential on the predicted outcome. Many ML algorithms have their own unique ways to quantify the importance or relative influence of each feature (i.e. coefficients for linear models, impurity for tree-based models). However, other algorithms like naive Bayes classifiers and support vector machines do not. This makes it difficult to compare variable importance across multiple models.

DALEX uses a model agnostic variable importance measure computed via permutation. This approach follows the following steps:

For any given loss function do

1: compute loss function for full model (denote \_full\_model\_)

2: randomize response variable, apply given ML, and compute loss function (denote \_baseline\_)

3: for variable j

| randomize values

| apply given ML model

| compute & record loss function

end

To compute the permuted variable importance we use DALEX::variable\_importance(). The printed output just provides a data frame with the output and plotting the three variable importance objects allows us to compare the most influential variables for each model. How do we interpret this plot?

1. Left edge of x-axis is the loss function for the \_full\_model\_. The default loss function is squared error but any custom loss function can be supplied.
2. The first item listed in each plot is \_baseline\_. This value represents the loss function when our response values are randomized and should be a good indication of the worst-possible loss function value when there is no predictive signal in the data.
3. The length of the remaining variables represent the variable importance. The larger the line segment, the larger the loss when that variable is randomized.

The results provide some interesting insights. First, the shifted x-axis left edge helps to illustrate the difference in the RMSE loss between the three models (i.e. GLM model has the lowest RMSE suggesting that the greater number of tail residuals in the GBM model is likely penalizing the RMSE score. Second, we can see which variables are consistently influential across all models (i.e. OverTime, EnvironmentSatisfaction, Age), variables that are influential in two but not all three (i.e. BusinessTravel, WorkLifeBalance), and variables which are only influential in one model but not others (i.e. DailyRate, YearsInCurrentRole). This helps you to see if models are picking up unique structure in the data or if they are using common logic.

In this example, all three models appear to be largely influenced by the OverTime, EnvironmentSatisfaction, Age, TotalWorkingYears, and JobLevel variables. This gives us confidences that these features have strong predictive signals.

**TIP**: You can incorporate custom loss functions using the loss\_function argument.

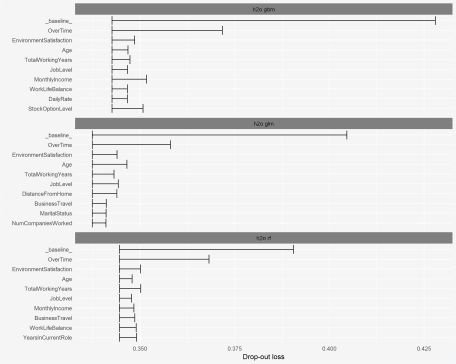
# compute permutation-based variable importance

vip\_glm <- variable\_importance(explainer\_glm, n\_sample = -1, loss\_function = loss\_root\_mean\_square)

vip\_rf <- variable\_importance(explainer\_rf, n\_sample = -1, loss\_function = loss\_root\_mean\_square)

vip\_gbm <- variable\_importance(explainer\_gbm, n\_sample = -1, loss\_function = loss\_root\_mean\_square)

plot(vip\_glm, vip\_rf, vip\_gbm, max\_vars = 10)



One downfall of the permutation-based approach to variable importance is it can become slow. Since the algorithm loops through and applies a model for each predictor variable, the more features in your model the longer it will take. For this example, which includes 30 features, it takes 81 seconds to compute variable importance for all three models. However, when tested on a data set with 100 predictors it took nearly 5 minutes to compute.

**TIP**: variable\_importance includes an n\_sample argument that, by default, will sample only 1000 observations to try increase the speed of computation. Adjusting n\_sample = -1 as I did in the above code chunk just means to use all observations.

**6.0 Predictor-response relationship**

Once we’ve identified influential variables across all three models, next we likely want to understand how the relationship between these influential variables and the predicted response differ between the models. This helps to indicate if each model is responding to the predictor signal similarly or if one or more models respond differently. For example, we saw that the Age variable was one of the most influential variables across all three models. The below partial dependence plot illustrates that the GBM and random forest models are using the Age signal in a similar non-linear manner; however, the GLM model is not able to capture this same non-linear relationship. So although the GLM model may perform better (re: AUC score), it may be using features in biased or misleading ways.

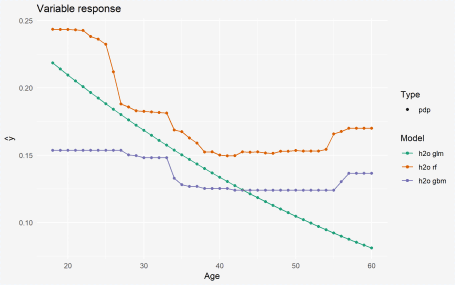
# compute PDP for a given variable --> uses the pdp package

pdp\_glm <- variable\_response(explainer\_glm, variable = "Age", type = "pdp")

pdp\_rf <- variable\_response(explainer\_rf, variable = "Age", type = "pdp")

pdp\_gbm <- variable\_response(explainer\_gbm, variable = "Age", type = "pdp")

plot(pdp\_glm, pdp\_rf, pdp\_gbm)



Although you can use PDPs for categorical predictor variables, DALEX provides *merging path plots* originally provided by the factoMerger package. For example, the EnvironmentSatisfaction variable captures the level of satisfaction regarding the working environment among employees. This variable showed up in all three models’ top 10 most influential variable lists. We can use type = "factor" to create a merging path plot and it shows very similar results for each model. Those employees that have low level of satisfaction have, on average, higher probabilities of attrition. Whereas, employees with medium to very high have about the same likelihood of attriting. The left side of the plot is the merging path plot, which shows the similarity between groups via hierarchical clustering. It illustrates that employees with medium and high satisfaction are most similar, then these employees are next most similar to employees with very high satisfaction. Then finally, the least similar group is the low satisfaction employees.

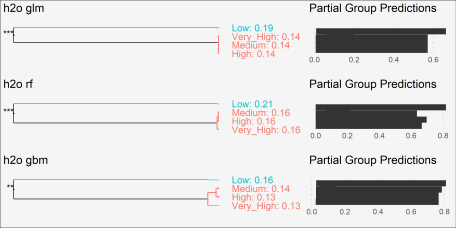
Library(factomerger)

cat\_glm <- variable\_response(explainer\_glm, variable = "EnvironmentSatisfaction", type = "factor")

cat\_rf <- variable\_response(explainer\_rf, variable = "EnvironmentSatisfaction", type = "factor")

cat\_gbm <- variable\_response(explainer\_gbm, variable = "EnvironmentSatisfaction", type = "factor")

plot(cat\_glm, cat\_rf, cat\_gbm)



**7.0 Local interpretation**

The previous plots help us to understand our model from a ***global perspective*** by illustrating errors, identifying the variables with the largest overall impact, and understanding predictor-response relationships across all observations. However, often, we also need to perform ***local interpretation*** which allows us to understand why a particular prediction was made for an observation. Understanding and comparing how a model uses the predictor variables to make a given prediction can provide trust to you (the analyst) and also the stakeholder(s) that will be using the model output for decision making purposes.

Although LIME and SHAP values have recently become popular for local ML interpretation, DALEX uses a process called break down to compute localized variable importance scores.

There are two break down approaches that can be applied. The default is called ***step up*** and the algorithm performs the following steps:

Library(breakdown)

existing\_data = validation data set used in explainer

new\_ob = single observation to perform local interpretation on

p = number of predictors

l = list of predictors

baseline = mean predicted response of existing\_data

for variable i in {1,...,p} do

for variable j in {1,...,l} do

| substitue variable j in existing\_data with variable j value in new\_ob

| predicted\_j = mean predicted response of altered existing\_data

| diff\_j = absolute difference between baseline - predicted

| reset existing\_data

end

| t = variable j with largest diff value

| contribution for variable t = diff value for variable t

| remove variable t from l

end

This is called step up because, essentially, it sweeps through each column, identifies the column with the largest difference score, adds that variable to the list as the most important, sweeps through the remaining columns, identifies the column with the largest score, adds that variable to the list as second most important, etc. until all variables have been assessed.

An alternative approach is called the ***step down*** which follows a similar algorithm but rather than remove the variable with the largest difference score on each sweep, it removes the variable with the smallest difference score. Both approaches are analogous to backward stepwise selection where *step up* removes variables with largest impact and *step down* removes variables with smallest impact.

To perform the break down algorithm on a single observation, use the DALEX::prediction\_breakdown function. The output is a data frame with class “prediction\_breakdown\_explainer” that lists the contribution for each variable.

**TIP**: The default approach is ***step up*** but you can perform ***step down*** by adding the following argument direction = "down".

# create a single observation

new\_cust <- splits$valid[1, ] %>% as.data.frame()

# compute breakdown distances

new\_cust\_glm <- prediction\_breakdown(explainer\_glm, observation = new\_cust)

new\_cust\_rf <- prediction\_breakdown(explainer\_rf, observation = new\_cust)

new\_cust\_gbm <- prediction\_breakdown(explainer\_gbm, observation = new\_cust)

# class of prediction\_breakdown output

class(new\_cust\_gbm)

## [1] "prediction\_breakdown\_explainer" "data.frame"

# check out the top 10 influential variables for this observation

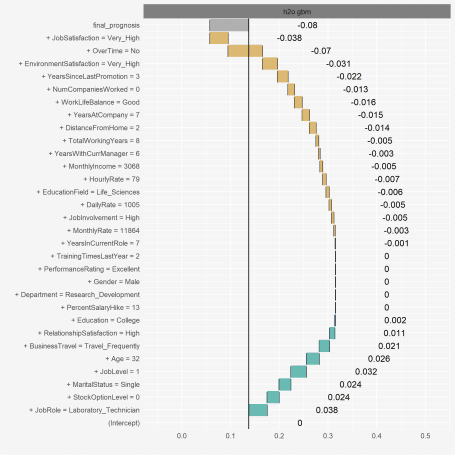
new\_cust\_gbm[1:10, 1:5] %>%

knitr::kable()

|  | **variable** | **contribution** | **variable\_name** | **variable\_value** | **cummulative** |
| --- | --- | --- | --- | --- | --- |
| 1 | (Intercept) | 0.0000000 | Intercept | 1 | 0.0000000 |
| JobRole | + JobRole = Laboratory\_Technician | 0.0377084 | JobRole | Laboratory\_Technician | 0.0377084 |
| StockOptionLevel | + StockOptionLevel = 0 | 0.0243714 | StockOptionLevel | 0 | 0.0620798 |
| MaritalStatus | + MaritalStatus = Single | 0.0242334 | MaritalStatus | Single | 0.0863132 |
| JobLevel | + JobLevel = 1 | 0.0318771 | JobLevel | 1 | 0.1181902 |
| Age | + Age = 32 | 0.0261924 | Age | 32 | 0.1443826 |
| BusinessTravel | + BusinessTravel = Travel\_Frequently | 0.0210466 | BusinessTravel | Travel\_Frequently | 0.1654292 |
| RelationshipSatisfaction | + RelationshipSatisfaction = High | 0.0108112 | RelationshipSatisfaction | High | 0.1762404 |
| Education | + Education = College | 0.0016912 | Education | College | 0.1779315 |
| PercentSalaryHike | + PercentSalaryHike = 13 | 0.0001158 | PercentSalaryHike | 13 | 0.1780473 |

We can plot the entire list of contributions for each variable of a particular model. We can see that several predictors have zero contribution, while others have positive and negative contributions. For the GBM model, the predicted value for this individual observation was positively influenced (increased probability of attrition) by variables such as JobRole, StockOptionLevel, and MaritalStatus. Alternatively, variables such as JobSatisfaction, OverTime, and EnvironmentSatisfaction reduced this observations probability of attriting.

plot(new\_cust\_gbm)



For data sets with a small number of predictors, you can compare across multiple models in a similar way as with earlier plotting (plot(new\_cust\_glm, new\_cust\_rf, new\_cust\_gbm)). However, with wider data sets, this becomes cluttered and difficult to interpret. Alternatively, you can filter for the largest absolute contribution values. This causes the output class to lose its prediction\_breakdown\_explainer class so we can plot the results with ggplot.

Each model has a similar prediction that the new observation has a low probability of predicting:

* GLM: .12
* random forest: 0.18
* GBM: 0.06

However, how each model comes to that conclusion in a slightly different way. However, there are several predictors that we see consistently having a positive or negative impact on this observations’ probability of attriting (i.e. OverTime, EnvironmentSatisfaction, JobSatisfaction are reducing this employees probability of attriting while JobLevel, MaritalStatus, StockOptionLevel, and JobLevel are all increasing the probability of attriting). Consequently, we can have a decent amount of trust that these are strong signals for this observation regardless of model. However, when each model picks up unique signals in variables that the other models do not capture (i.e. DistanceFromHome, NumCompaniesWorked), its important to be careful how we communicate these signals to stakeholders. Since these variables do not provide consistent signals across all models we should use domain experts or other sources to help validate whether or not these predictors are trustworthy. This will help us understand if the model is using proper logic that translates well to business decisions.

library(ggplot2)

# filter for top 10 influential variables for each model and plot

list(new\_cust\_glm, new\_cust\_rf, new\_cust\_gbm) %>%

purrr::map(~ top\_n(., 11, wt = abs(contribution))) %>%

do.call(rbind, .) %>%

mutate(variable = paste0(variable, " (", label, ")")) %>%

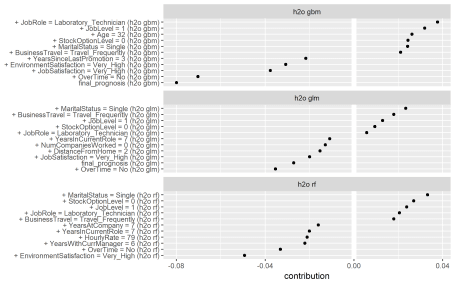
ggplot(aes(contribution, reorder(variable, contribution))) +

geom\_point() +

geom\_vline(xintercept = 0, size = 3, color = "white") +

facet\_wrap(~ label, scales = "free\_y", ncol = 1) +

ylab(NULL)



Unfortunately, a major drawback to DALEX’s implementation of these algorithm’s is that they are not parallelized. Consequently, wide data sets become extremely slow. For example, performing the previous three prediction\_breakdown functions on this attrition data set with 30 predictors takes about 12 minutes. However, this grows exponentially as more predictors are added. When we apply a single instance of prediction\_breakdown to the Ames housing data (80 predictors), it ***took over 3 hours to execute***!

# ames data

ames.h2o <- as.h2o(AmesHousing::make\_ames())

# create local observation

local\_ob <- as.data.frame(ames.h2o[1, ])

# variable names for resonse & features

y <- "Sale\_Price"

x <- setdiff(names(ames.h2o), y)

# random forest model

rf <- h2o.randomForest(

x = x,

y = y,

training\_frame = ames.h2o,

ntrees = 1000

)

# get features for explainer

x\_valid <- as.data.frame(ames.h2o)[, x]

y\_valid <- as.vector(ames.h2o[y])

# create custom predict function

pred <- function(model, newdata) {

results <- as.vector(predict(model, as.h2o(newdata)))

return(results)

}

# create explainer

ames\_rf <- explain(

model = rf,

data = x\_valid,

y = y\_valid,

predict\_function = pred,

label = "ames"

)

# time to compute prediction break down

system.time({

ames\_example <- prediction\_breakdown(ames\_rf, observation = local\_ob)

})

Looking at the underlying code for the prediction\_breakdown function (it simply calls breakDown::broken.default), there are opportunities for integrating parallelization capabilities (i.e. via foreach package). Consequently, prior to adding it to your preferred ML toolkit, you should determine:

1. if you are satisfied with its general alorithmic approach,
2. do you typically use wide data sets, and if so…
3. what is your appetite and bandwidth for integrating parallelization (either in your own version or collaborating with the package authors),
4. and how is performance after parallelization (do you see enough speed improvement to justify use).

Code Chunks - Model Agnostic Approach to Breaking Down of Model Predictions

|  |
| --- |
| #' Model Agnostic Approach to Breaking Down of Model Predictions |
|  | #' |
|  | #' This function implements two greedy strategies for decompositions of model predictions (see the \code{direction} parameter). |
|  | #' Both stategies are model agnostic, they are greedy but in most cases they give very similar results. |
|  | #' Find more information about these strategies in \url{https://arxiv.org/abs/1804.01955}. |
|  | #' |
|  | #' @param model a model, it can be any predictive model, find examples for most popular frameworks in vigniettes |
|  | #' @param new\_observation a new observation with columns that corresponds to variables used in the model |
|  | #' @param data the original data used for model fitting, should have same collumns as the 'new\_observation'. |
|  | #' @param direction either 'up' or 'down' determined the exploration strategy |
|  | #' @param ... other parameters |
|  | #' @param baseline the orgin/baseline for the breakDown plots, where the rectangles start. It may be a number or a character "Intercept". In the latter case the orgin will be set to model intercept. |
|  | #' @param predict.function function that will calculate predictions out of model. It shall return a single numeric value per observation. For classification it may be a probability of the default class. |
|  | #' @param keep\_distributions if TRUE, then the distribution of partial predictions is stored in addition to the average. |
|  | #' |
|  | #' @return an object of the broken class |
|  | #' |
|  | #' @examples |
|  | #' \dontrun{ |
|  | #' library("breakDown") |
|  | #' library("randomForest") |
|  | #' library("ggplot2") |
|  | #' set.seed(1313) |
|  | #' model <- randomForest(factor(left)~., data = HR\_data, family = "binomial", maxnodes = 5) |
|  | #' predict.function <- function(model, new\_observation) |
|  | #' predict(model, new\_observation, type="prob")[,2] |
|  | #' predict.function(model, HR\_data[11,-7]) |
|  | #' explain\_1 <- broken(model, HR\_data[11,-7], data = HR\_data[,-7], |
|  | #' predict.function = predict.function, direction = "down") |
|  | #' explain\_1 |
|  | #' plot(explain\_1) + ggtitle("breakDown plot (direction=down) for randomForest model") |
|  | #' |
|  | #' explain\_2 <- broken(model, HR\_data[11,-7], data = HR\_data[,-7], |
|  | #' predict.function = predict.function, direction = "down", keep\_distributions = TRUE) |
|  | #' plot(explain\_2, plot\_distributions = TRUE) + |
|  | #' ggtitle("breakDown distributions (direction=down) for randomForest model") |
|  | #' |
|  | #' explain\_3 <- broken(model, HR\_data[11,-7], data = HR\_data[,-7], |
|  | #' predict.function = predict.function, direction = "up", keep\_distributions = TRUE) |
|  | #' plot(explain\_3, plot\_distributions = TRUE) + |
|  | #' ggtitle("breakDown distributions (direction=up) for randomForest model") |
|  | #' } |
|  | #' @export |
|  |  |
|  | broken.default <- function(model, new\_observation, data, direction = "up", ..., baseline = 0, |
|  | keep\_distributions = FALSE, predict.function = predict) { |
|  | # just in case only some variables are specified |
|  | # this will work only for data.frames |
|  | if ("data.frame" %in% class(data)) { |
|  | common\_variables <- intersect(colnames(new\_observation), colnames(data)) |
|  | new\_observation <- new\_observation[,common\_variables, drop = FALSE] |
|  | data <- data[,common\_variables, drop = FALSE] |
|  | } |
|  |  |
|  | if (direction == "up") { |
|  | broken\_sorted <- broken\_go\_up(model, new\_observation, data, |
|  | predict.function, ...) |
|  | } else { |
|  | broken\_sorted <- broken\_go\_down(model, new\_observation, data, |
|  | predict.function, ...) |
|  | } |
|  |  |
|  | if (keep\_distributions) { |
|  | # calcuate distribution for partial predictions |
|  | open\_variables <- as.character(broken\_sorted$variable\_name) |
|  | current\_data <- data |
|  | yhats\_distribution <- list(data.frame(variable = "all data", |
|  | label = "all data", |
|  | id = 1:nrow(current\_data), |
|  | prediction = predict.function(model, current\_data, ...))) |
|  | for (i in seq\_along(open\_variables)) { |
|  | tmp\_variable <- open\_variables[i] |
|  | current\_data[,tmp\_variable] <- new\_observation[,tmp\_variable] |
|  | yhats\_distribution[[tmp\_variable]] <- data.frame(variable = tmp\_variable, |
|  | label = as.character(broken\_sorted$variable[i]), |
|  | id = 1:nrow(current\_data), |
|  | prediction = predict.function(model, current\_data, ...) |
|  | ) |
|  | } |
|  | yhats\_df <- do.call(rbind, yhats\_distribution) |
|  | } |
|  |  |
|  |  |
|  | if (tolower(baseline) == "intercept") { |
|  | baseline <- mean(predict.function(model, data, ...)) |
|  | broken\_sorted <- rbind( |
|  | data.frame(variable = "(Intercept)", |
|  | contribution = 0, |
|  | variable\_name = "Intercept", |
|  | variable\_value = 1), |
|  | broken\_sorted) |
|  | } else { |
|  | broken\_sorted <- rbind( |
|  | data.frame(variable = "(Intercept)", |
|  | contribution = mean(predict.function(model, data, ...)) - baseline, |
|  | variable\_name = "Intercept", |
|  | variable\_value = 1), |
|  | broken\_sorted) |
|  | } |
|  |  |
|  | result <- create.broken(broken\_sorted, baseline) |
|  |  |
|  | if (keep\_distributions) { |
|  | attr(result, "yhats\_distribution") <- yhats\_df |
|  | } |
|  |  |
|  | result |
|  | } |
|  |  |
|  | broken\_go\_up <- function(model, new\_observation, data, |
|  | predict.function = predict, ...) { |
|  | # set target distribution |
|  | new\_data <- new\_observation[rep(1L, nrow(data)),] |
|  |  |
|  | # set target |
|  | target\_yhat <- predict.function(model, new\_observation, ...) |
|  | baseline\_yhat <- mean(predict.function(model, data, ...)) |
|  |  |
|  | # set variable indicators |
|  | open\_variables <- 1:ncol(data) |
|  |  |
|  | important\_variables <- c() |
|  | important\_yhats <- list() |
|  |  |
|  | for (i in 1:ncol(data)) { |
|  | yhats <- list() |
|  | yhats\_diff <- rep(-Inf, ncol(data)) |
|  | for (tmp\_variable in open\_variables) { |
|  | current\_data <- data |
|  | current\_data[,tmp\_variable] <- new\_data[,tmp\_variable] |
|  | yhats[[tmp\_variable]] <- predict.function(model, current\_data, ...) |
|  | yhats\_diff[tmp\_variable] <- abs(baseline\_yhat - mean(yhats[[tmp\_variable]])) |
|  | } |
|  | important\_variables[i] <- which.max(yhats\_diff) |
|  | important\_yhats[[i]] <- yhats[[which.max(yhats\_diff)]] |
|  | data[, important\_variables[i]] <- new\_data[, important\_variables[i]] |
|  | open\_variables <- setdiff(open\_variables, which.max(yhats\_diff)) |
|  | } |
|  |  |
|  | varNames <- colnames(data)[important\_variables] |
|  | varValues <- sapply(new\_observation[,important\_variables], as.character) |
|  | contributions <- diff(c(baseline\_yhat, sapply(important\_yhats, mean))) |
|  |  |
|  | broken\_sorted <- data.frame(variable = paste("+", varNames, "=", varValues), |
|  | contribution = contributions, |
|  | variable\_name = varNames, |
|  | variable\_value = varValues) |
|  |  |
|  | broken\_sorted |
|  | } |
|  |  |
|  | broken\_go\_down <- function(model, new\_observation, data, |
|  | predict.function = predict, ...) { |
|  | # set target distribution |
|  | new\_data <- new\_observation[rep(1L, nrow(data)),] |
|  |  |
|  | # set target |
|  | target\_yhat <- predict.function(model, new\_observation, ...) |
|  | baseline\_yhat <- mean(predict.function(model, data, ...)) |
|  |  |
|  | # set variable indicators |
|  | open\_variables <- 1:ncol(data) |
|  |  |
|  | important\_variables <- c() |
|  | important\_yhats <- list() |
|  |  |
|  | for (i in 1:ncol(data)) { |
|  | yhats <- list() |
|  | yhats\_diff <- rep(Inf, ncol(data)) |
|  | for (tmp\_variable in open\_variables) { |
|  | current\_data <- new\_data |
|  | current\_data[,tmp\_variable] <- data[,tmp\_variable] |
|  | yhats[[tmp\_variable]] <- predict.function(model, current\_data, ...) |
|  | yhats\_diff[tmp\_variable] <- abs(target\_yhat - mean(yhats[[tmp\_variable]])) |
|  | } |
|  | important\_variables[i] <- which.min(yhats\_diff) |
|  | important\_yhats[[i]] <- yhats[[which.min(yhats\_diff)]] |
|  | new\_data[, important\_variables[i]] <- data[, important\_variables[i]] |
|  | open\_variables <- setdiff(open\_variables, which.min(yhats\_diff)) |
|  | } |
|  |  |
|  | varNames <- rev(colnames(data)[important\_variables]) |
|  | varValues <- sapply(rev(new\_observation[,important\_variables]), as.character) |
|  | contributions <- diff(c(rev(sapply(important\_yhats, mean)), target\_yhat)) |
|  |  |
|  | broken\_sorted <- data.frame(variable = c(paste("-", varNames, "=", varValues)), |
|  | contribution = contributions, |
|  | variable\_name = varNames, |
|  | variable\_value = varValues) |
|  |  |
|  | broken\_sorted |
|  | } |

**Additional DALEX Resources**

The following provides resources to learn more about the DALEX package:

* DALEX GitHub repo: https://github.com/pbiecek/DALEX
* breakDown package which is called by DALEX: https://github.com/pbiecek/breakDown

What if you could use data science to predict and explain turnover in a way that managers could make better decisions and executives would see results? You will learn the tools to do so in our Virtual Workshop using the R Statistical Programming Language. Here’s an example of a Shiny app you will create.