**Motivation**

There are dozens of machine learning algorithms out there. It is impossible to learn all their mechanics; however, many algorithms sprout from the most established algorithms, e.g. ordinary least squares, gradient boosting, support vector machines, tree-based algorithms and neural networks

**Why bother writing from scratch?**

While I like reading machine learning research papers, the maths is sometimes hard to follow. That is why I like implementing the algorithms in R by myself. Of course, this means digging through the maths and the algorithms as well. However, you can challenge your understanding of the algorithm directly.

This blog post is about the random forest, which is probably the most prominent machine learning algorithm. You have probably noticed the asterisk (\*) in the title. These things often suggest, that there has to be something off. Like seeing a price for a cell phone plan in a tv commercial and while reading the fine prints you learn that it only applies if you have successfully climbed Mount Everest and you got three giraffes on your yacht. Also, yes your suspicion is justified; unfortunately, the 100 lines of code only apply if we don’t add the code from the regression tree algorithm, which is essential for a random forest.

**Understanding ML with simple and accessible code**

In this series we try to produce very generic code, i.e. it won’t produce a state-of-the-art performance. It is instead designed to be very generic and easily readable.

Admittedly, there are tons of great articles out there which explain random forests theoretically accompanied with a hands-on example. That is not the objective of this blog post. If you are interested in a hands-on tutorial with all the necessary theory. The objective of this blog post is to establish the theory of the algorithm by writing simple R code. The only thing you need to know, besides the fundamentals of a regression tree, is our objective: We want to estimate our real-valued target (y) with a set of real-valued features (X).

**Reduce dimensionality with Feature Importance**

Before we jump into the random forest code, I would like to touch very briefly on how we can compute feature importance in a regression tree. Surely, there are tons of ways on how you can calculate feature importance, the following approach is, however, quite intuitive and straightforward.

**Evaluating the goodness of a split**

A regression tree splits the data by choosing the feature which minimizes a certain criterion, e.g. the squared error of our prediction. Of course, it is possible, that some features will never be chosen for a split, which makes calculating their importance very easy. However, how can we compute importance with chosen features? A first shot could be to count the number of splits for each feature and relativize it by the total number of all splits. This measure is simple and intuitive, but it cannot quantify how impactful the splits were, and this is something we can accomplish with a very simple but more sophisticated metric. This metric is a weighted goodness of fit. We start by defining our goodness of fit for each node. For instance, the mean squared error, which is defined as:

MSE_{node} = \frac{1}{n_{node}} \sum_{i = 1}^{n_{node}} (y_i - \bar{y}_{node}))^2

This metric describes the average squared error we make when we estimate our target y\_i with our predictor the average value in our current node \bar(y)\_{node}. Now we can measure the improvement by splitting the data with the chosen feature and compare the goodness of fit of the parent node with the performance of its children nodes. Essentially, this is more or less the exact step we performed to evaluate the best splitting feature for the current node.

**Weighting the impact of a split**

Splits at the top of the tree are more impactful as more data reaches the nodes at this stage of the tree. That’s why it makes sense to lay more importance on earlier splits by taking into account the number of observations which reached this node.

w_{node} = \frac{n_{node}}{n}

This weight describes the number of observations in the current node measured by the total number of observations. Combining the results above, we can derive a weighted improvement of a splitting feature p in a single node as:

**Quantifying improvements by splitting the data in a regression tree**

Improvement^{[p]}_{node} = w_{node}MSE_{node} - (w_{left}MSE_{left} + w_{right}MSE_{right})

This weighted improvement is calculated at every node which was split for the respective splitting feature p. To get better interpretability of this improvement, we sum up the improvements for each feature in our tree and normalize it by the overall improvement in our tree.

**Quantifying the importance of a feature in a regression tree**

Importance^{[p]} = \frac{\sum_{k=1}^{p} Improvement^{[k]}}{\sum_{i=1}^{n_{node}} Improvement^{[i]}}

This is the final feature importance measure used within regression tree algorithm. Again, you can follow these steps within the code of the regression tree. I have tagged all variables, functions and column names involved in the feature importance calculation with imp\_\* or IMP\_\* , that should make it a little easier to follow.

Code Of the regression Tree

|  |
| --- |
| # Regression Tree --------------------------------------------------------- |
|  |  |
|  | # This is the splitting criterion we minimize (SSE [Sum Of Squared Errors]): |
|  | # $SSE = \sum\_{i \in S\_1} (y\_i - \bar(y)1)^2 + \sum\_{i \in S\_2} (y\_i - \bar(y)2)^2$ |
|  | sse\_var <- function(x, y) { |
|  | splits <- sort(unique(x)) |
|  | sse <- c() |
|  | for (i in seq\_along(splits)) { |
|  | sp <- splits[i] |
|  | sse[i] <- sum((y[x < sp] - mean(y[x < sp]))^2) + sum((y[x >= sp] - mean(y[x >= sp]))^2) |
|  | } |
|  | split\_at <- splits[which.min(sse)] |
|  | return(c(sse = min(sse), split = split\_at)) |
|  | } |
|  |  |
|  |  |
|  |  |
|  | #' reg\_tree |
|  | #' Fits a simple regression tree with SSE splitting criterion. The estimator function |
|  | #' is the mean. |
|  | #' |
|  | #' @param formula an object of class formula |
|  | #' @param data a data.frame or matrix |
|  | #' @param minsize a numeric value indicating the minimum size of observations |
|  | #' in a leaf |
|  | #' |
|  | #' @return \itemize{ |
|  | #' \item tree - the tree object containing all splitting rules and observations |
|  | #' \item imp - returns the feature importance |
|  | #' \item fit - our fitted values, i.e. X %\*% theta |
|  | #' \item formula - the underlying formula |
|  | #' \item data - the underlying data |
|  | #' } |
|  | #' @export |
|  | #' |
|  | #' @examples # Complete runthrough see: www.github.com/andrebleier/cheapml |
|  | reg\_tree\_imp <- function(formula, data, minsize) { |
|  |  |
|  | # coerce to data.frame |
|  | data <- as.data.frame(data) |
|  |  |
|  | # handle formula |
|  | formula <- terms.formula(formula) |
|  |  |
|  | # get the design matrix |
|  | X <- model.matrix(formula, data) |
|  |  |
|  | # extract target |
|  | y <- data[, as.character(formula)[2]] |
|  |  |
|  | # initialize while loop |
|  | do\_splits <- TRUE |
|  |  |
|  | # create output data.frame with splitting rules and observations |
|  | tree\_info <- data.frame(NODE = 1, NOBS = nrow(data), FILTER = NA, TERMINAL = "SPLIT", |
|  | IMP\_GINI = NA, SPLIT = NA, stringsAsFactors = FALSE) |
|  |  |
|  | # keep splitting until there are only leafs left |
|  | while(do\_splits) { |
|  |  |
|  | # which parents have to be splitted |
|  | to\_calculate <- which(tree\_info$TERMINAL == "SPLIT") |
|  |  |
|  | for (j in to\_calculate) { |
|  |  |
|  | # handle root node |
|  | if (!is.na(tree\_info[j, "FILTER"])) { |
|  | # subset data according to the filter |
|  | this\_data <- subset(data, eval(parse(text = tree\_info[j, "FILTER"]))) |
|  | # get the design matrix |
|  | X <- model.matrix(formula, this\_data) |
|  | } else { |
|  | this\_data <- data |
|  | } |
|  |  |
|  | # estimate splitting criteria |
|  | splitting <- apply(X, MARGIN = 2, FUN = sse\_var, y = this\_data[, all.vars(formula)[1]]) |
|  |  |
|  | # get the min SSE |
|  | tmp\_splitter <- which.min(splitting[1,]) |
|  |  |
|  | # define maxnode |
|  | mn <- max(tree\_info$NODE) |
|  |  |
|  | # paste filter rules |
|  | current\_filter <- c(paste(names(tmp\_splitter), ">=", |
|  | splitting[2,tmp\_splitter]), |
|  | paste(names(tmp\_splitter), "<", |
|  | splitting[2,tmp\_splitter])) |
|  |  |
|  | # Error handling! check if the splitting rule has already been invoked |
|  | split\_here <- !sapply(current\_filter, |
|  | FUN = function(x,y) any(grepl(x, x = y)), |
|  | y = tree\_info$FILTER) |
|  |  |
|  | # append the splitting rules |
|  | if (!is.na(tree\_info[j, "FILTER"])) { |
|  | current\_filter <- paste(tree\_info[j, "FILTER"], |
|  | current\_filter, sep = " & ") |
|  | } |
|  |  |
|  | # calculate metrics within the children |
|  | metr <- lapply(current\_filter, |
|  | FUN = function(i, x, data, formula) { |
|  | df <- subset(x = x, subset = eval(parse(text = i))) |
|  | nobs <- nrow(df) |
|  | w <- nobs/nrow(data) |
|  | y <- df[, all.vars(formula)[1]] |
|  | imp <- mean((y - mean(y, na.rm = TRUE))^2) |
|  | return(c(nobs, w\*imp)) |
|  | }, |
|  | x = this\_data, data = data, formula = formula) |
|  |  |
|  | # extract relevant information |
|  | current\_nobs <- sapply(metr, function(x) x[[1]]) |
|  | imp\_sum\_child <- sum(sapply(metr, function(x) x[[2]])) |
|  | current\_y <- this\_data[, all.vars(formula)[1]] |
|  | imp\_parent <- nrow(this\_data)/nrow(data) \* mean((current\_y-mean(current\_y))^2) |
|  | imp\_gini <- imp\_parent - imp\_sum\_child |
|  |  |
|  | # insufficient minsize for split |
|  | if (any(current\_nobs <= minsize)) { |
|  | split\_here <- rep(FALSE, 2) |
|  | } |
|  |  |
|  | # create children data frame |
|  | children <- data.frame(NODE = c(mn+1, mn+2), |
|  | NOBS = current\_nobs, |
|  | FILTER = current\_filter, |
|  | TERMINAL = rep("SPLIT", 2), |
|  | IMP\_GINI = NA, |
|  | SPLIT = NA, |
|  | row.names = NULL)[split\_here,] |
|  |  |
|  | # overwrite state of current node, add gini importance and split variable |
|  | tree\_info[j, "TERMINAL"] <- ifelse(all(!split\_here), "LEAF", "PARENT") |
|  | tree\_info[j, "IMP\_GINI"] <- imp\_gini |
|  | if (tree\_info[j, "TERMINAL"] == "PARENT") { |
|  | tree\_info[j, "SPLIT"] <- names(tmp\_splitter) |
|  | } |
|  |  |
|  | # bind everything |
|  | tree\_info <- rbind(tree\_info, children) |
|  |  |
|  | # check if there are any open splits left |
|  | do\_splits <- !all(tree\_info$TERMINAL != "SPLIT") |
|  | } # end for |
|  | } # end while |
|  |  |
|  | # calculate fitted values |
|  | leafs <- tree\_info[tree\_info$TERMINAL == "LEAF", ] |
|  | fitted <- c() |
|  | for (i in seq\_len(nrow(leafs))) { |
|  | # extract index |
|  | ind <- as.numeric(rownames(subset(data, eval(parse(text = leafs[i, "FILTER"]))))) |
|  | # estimator is the mean y value of the leaf |
|  | fitted[ind] <- mean(y[ind]) |
|  | } |
|  |  |
|  | # calculate feature importance |
|  | imp <- tree\_info[, c("SPLIT", "IMP\_GINI")] |
|  |  |
|  | if (!all(is.na(imp$SPLIT))) { |
|  | imp <- aggregate(IMP\_GINI ~ SPLIT, FUN = function(x, all) sum(x, na.rm = T)/sum(all, na.rm = T), |
|  | data = imp, all = imp$IMP\_GINI) |
|  | } |
|  |  |
|  | # rename to importance |
|  | names(imp) <- c("FEATURES", "IMPORTANCE") |
|  | imp <- imp[order(imp$IMPORTANCE, decreasing = TRUE),] |
|  |  |
|  | # return everything |
|  | return(list(tree = tree\_info, fit = fitted, formula = formula, |
|  | importance = imp, data = data)) |
|  | } |

**The Random forest Algorithm**

All right, enough with this regression tree and importance – we are interested in the forest in this blog post. The objective of a random forest is to combine many regression or decision trees. Such a combination of single results is referred to as ensemble techniques. The idea of this technique is very simple but yet very powerful.

**Building a regression tree orchestra**

In a symphonic orchestra, different groups of instruments are combined to form an ensemble, which creates more powerful and diverse harmonies. Essentially, it is the same in machine learning, because every regression tree we sprout in random forest has the chance to explore the data from a different angle. Our regression tree orchestra has thus different views on the data, which makes the combination very powerful and diverse opposed to a single regression tree.

**Simplicity of a random forest**

If you are not familiar with the mechanics algorithm, you probably think that the code gets very complicated and hard to follow. Well, to me the amazing part of this algorithm is how simple and yet effective it is. The coding part is not as challenging as you might think. Like in the other blog posts we take a look at the whole code first and then we go through it bit by bit.

**The algorithm at one glimpse**

#' reg\_rf

#' Fits a random forest with a continuous scaled features and target

#' variable (regression)

#'

#' @param formula an object of class formula

#' @param n\_trees an integer specifying the number of trees to sprout

#' @param feature\_frac an numeric value defined between [0,1]

#' specifies the percentage of total features to be used in

#' each regression tree

#' @param data a data.frame or matrix

#'

#' @importFrom plyr raply

#' @return

#' @export

#'

#' @examples # Complete runthrough see: www.github.com/andrebleier/cheapml

reg\_rf <- function(formula, n\_trees, feature\_frac, data) {

# source the regression tree function

source("algorithms/reg\_tree\_imp.R")

# load plyr

require(plyr)

# define function to sprout a single tree

sprout\_tree <- function(formula, feature\_frac, data) {

# extract features

features <- all.vars(formula)[-1]

# extract target

target <- all.vars(formula)[1]

# bag the data

# - randomly sample the data with replacement (duplicate are possible)

train <-

data[sample(1:nrow(data), size = nrow(data), replace = TRUE)]

# randomly sample features

# - only fit the regression tree with feature\_frac \* 100 % of the features

features\_sample <- sample(features,

size = ceiling(length(features) \* feature\_frac),

replace = FALSE)

# create new formula

formula\_new <-

as.formula(paste0(target, " ~ ", paste0(features\_sample,

collapse = " + ")))

# fit the regression tree

tree <- reg\_tree\_imp(formula = formula\_new,

data = train,

minsize = ceiling(nrow(train) \* 0.1))

# save the fit and the importance

return(list(treefit, treeimportance))

}

# apply the rf\_tree function n\_trees times with plyr::raply

# - track the progress with a progress bar

trees <- plyr::raply(

n\_trees,

sprout\_tree(

formula = formula,

feature\_frac = feature\_frac,

data = data

),

.progress = "text"

)

# extract fit

fits <- do.call("cbind", trees[, 1])

# calculate the final fit as a mean of all regression trees

rf\_fit <- apply(fits, MARGIN = 1, mean)

# extract the feature importance

imp\_full <- do.call("rbind", trees[, 2])

# build the mean feature importance between all trees

imp <- aggregate(IMPORTANCE ~ FEATURES, FUN = mean, imp\_full)

# build the ratio for interpretation purposes

impIMPORTANCE / sum(impIMPORTANCE, decreasing = TRUE), ]))

}IMPORTANCE <- impIMPORTANCE)    # export   return(list(fit = rf_fit,               importance = imp[order(imp

As you have probably noticed, our algorithm can be roughly divided into two parts. Firstly, a function sprout\_tree() and afterwards some lines of code which call this function and process its output. Let us now work through all the code chunk by chunk.

# source the regression tree function

source("algorithms/reg\_tree\_imp.R")

# load plyr

require(plyr)

# define function to sprout a single tree

sprout\_tree <- function(formula, feature\_frac, data) {

# extract features

features <- all.vars(formula)[-1]

# extract target

target <- all.vars(formula)[1]

# bag the data

# - randomly sample the data with replacement (duplicate are possible)

train <-

data[sample(1:nrow(data), size = nrow(data), replace = TRUE)]

# randomly sample features

# - only fit the regression tree with feature\_frac \* 100 % of the features

features\_sample <- sample(features,

size = ceiling(length(features) \* feature\_frac),

replace = FALSE)

# create new formula

formula\_new <-

as.formula(paste0(target, " ~ ", paste0(features\_sample,

collapse = " + ")))

# fit the regression tree

tree <- reg\_tree\_imp(formula = formula\_new,

data = train,

minsize = ceiling(nrow(train) \* 0.1))

# save the fit and the importance

return(list(treefit, treeimportance))

}

**Sprouting a single regression tree in a forest**

The first part of the code is the sprout\_tree() function, which is just a wrapper for the regression tree function reg\_tree\_imp(), which we source as the first action of our code. Then we extract our target and the features from the formula object.

**Creating different angles by bagging the data**

Afterwards, we bag the data, which means we are randomly sampling our data with the chance of replacement. Remember when I said every tree will look at our data from a different angle? Well, this is the part where we create the angles. Random sampling with replacement is just a synonym for generating weights on our observations. This means that a specific observation in the data set of a specific tree could be repeated 10 times. The next tree could, however, lose this observation completely. Furthermore, there is another way of creating different angles in our trees: Feature sampling.

**Solving collinearity issues between trees in a forest**

From our complete feature set X we are randomly sampling a feature\_frac \* 100 percent to reduce the dimensionality. With feature sampling we can a) compute faster and b) capture angles on our data from supposedly weaker features as we decrease the value of feature\_frac . Suppose, we have some degree of multicollinearity between our features. It might occur, that the regression tree will select only a specific feature if we use every feature in every tree.

However, supposedly features with less improvement could bare new valuable information for the model, but are not granted the chance. This is something you can achieve by lowering the dimensionality with the argument feature\_frac. If your objective of the analysis is feature selection, e.g. feature importance, you might want to set this parameter to 80%-100% as you will get a more clear cut selection. Well, the rest of the function is fitting the regression tree and exporting the fitted values as well as the importance.

**Sprouting a forest**

In the next code chunk we start applying the sprout\_tree() function n\_trees times with the help of plyr::raply(). This function repeatedly applies a function call with the same arguments and combines the results in a list. Remember, we do not need to change anything in the sprout\_tree() function, since the angles are created randomly every time, we call the function.

# apply the rf\_tree function n\_trees times with plyr::raply

# - track the progress with a progress bar

trees <- plyr::raply(

n\_trees,

sprout\_tree(

formula = formula,

feature\_frac = feature\_frac,

data = data

),

.progress = "text"

)

# extract fit

fits <- do.call("cbind", trees[, 1])

# calculate the final fit as a mean of all regression trees

rf\_fit <- apply(fits, MARGIN = 1, mean)

# extract the feature importance

imp\_full <- do.call("rbind", trees[, 2])

# build the mean feature importance between all trees

imp <- aggregate(IMPORTANCE ~ FEATURES, FUN = mean, imp\_full)

# build the ratio for interpretation purposes

impIMPORTANCE <- impIMPORTANCE / sum(impIMPORTANCE)    # export   return(list(fit = rf_fit,               importance = imp[order(impIMPORTANCE, decreasing = TRUE), ]))

Afterwards, we combine the single regression tree fits a data frame. By calculating the row mean we are taking the average fitted value of every regression tree in our forest. Our last action is to calculate the feature importance of our ensemble. That’s the mean feature importance of a feature in all trees normalized by the overall mean importance of all variables.

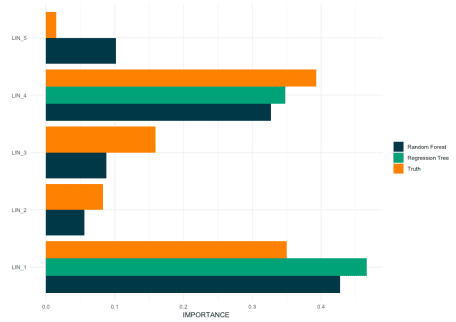
**Applying the algorithm**

Let us apply the function to see, whether the fit is indeed better compared to a single regression tree. Additionally, we can check out the feature importance. First, we simulate data with the Xy package. In this simulation, five linear variables were used to create our target y. To make it a little spicier, we add five irrelevant variables, which were created in the simulation as well. The challenge now, of course, is whether the algorithm will use any irrelevant feature or if the algorithm can perfectly identify the important features. Our formula is:

eq <- y ~ LIN\_1 + LIN\_2 + LIN\_3 + LIN\_4 + LIN\_5 + NOISE\_1 + NOISE\_2 + NOISE\_3 + NOISE\_4 + NOISE\_5

**The power of the forest**

Neither the random forest nor the regression tree has selected any unnecessary features. However, the regression tree was only split by the two most important variables. Whereas, the random forest selected all five relevant features.



This does not mean that the regression tree is not able to find the right answer. It depends on the minimum observation size (minsize) of the tree. Surely the regression tree would eventually find all important features if we lower the minimum size. The random forest, however, found all five essential features with the same minimum size.

Simulate the Data Through Xy Package

**Simulate data**

You can simulate regression and classification data with interactions and a user-specified non-linearity. The usage is in a tidy way. First you create a simulation recipe, which is a combination of the overall task invoked by Xy() . Afterwards effects can be added to the recipe with the add\_\* functions as can be seen in the example below. Finally the simulate() function cooks this recipe.

# load the library

library(Xy)

# simulate regression data

task <- Xy(task = "regression")

# build the recipe

recipe <- task %>%

# adding linear features

add\_linear(p = 5, family = xy\_normal()) %>%

# adding non-linear cubic features

add\_nonlinear(p = 3, nlfun= function(x) x^3, family = xy\_normal()) %>%

# add uninformative effects

add\_uninformative(p = 3, collinearity = TRUE, family = xy\_normal()) %>%

# add dummy variables

add\_discrete(p = 3, levels = 3) %>%

# add normally distributed noise, which correlates with the features

add\_noise(collinearity = TRUE, family = xy\_normal()) %>%

# add an intercept

add\_intercept()

# cook the recipe

sim <- recipe %>%

simulate(n = 100, r\_squared = 0.8)

# fetch the data

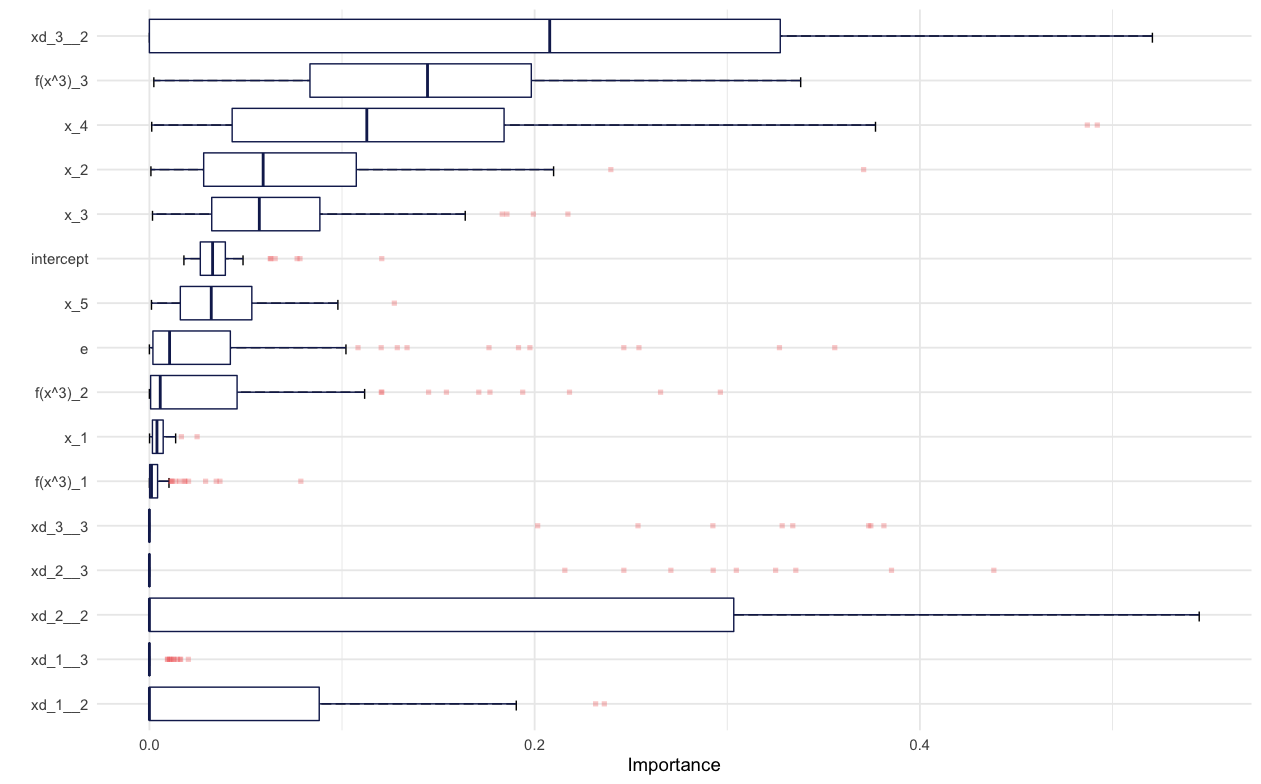
sim %>% pull\_xy()

**Feature Importance**

You can extract a feature importance of your simulation. For instance, to benchmark feature selection algorithms.

# Feature Importance

variable\_importance <- sim %>% importance()

[](https://github.com/andrebleier/Xy/raw/master/man/figures/imp.png)