This blog post is about my newly released RGF package (the blog post consists mainly of the package Vignette). The *RGF* package is a wrapper of the *Regularized Greedy Forest* *python* package, which also includes a *Multi-core implementation (FastRGF)*. Portability from Python to R was made possible using the reticulate package and the installation requires basic knowledge of Python. Except for the Linux Operating System, the installation on Macintosh and Windows might be somehow cumbersome (*on windows the package currently can be used only from within the command prompt*).

A small synopsis would be *“… the resulting method, which we refer to as regularized greedy forest (RGF), integrates two ideas: one is to include tree-structured regularization into the learning formulation; and the other is to employ the fully-corrective regularized greedy algorithm ….”*.

At the time of writing this blog post (14 – 02 – 2018), there isn’t a corresponding implementation of the algorithm in the R language, so I decided to port the Python package in R taking advantage of the reticulate package. In the next lines, I will explain the functionality of the package and I compare RBF with other similar implementations, such as *ranger* (random forest algorithm) and *xgboost* (gradient boosting algorithm), in terms of time efficiency and error rate improvement.

Code Chunks – Fast RGF Classifier

|  |
| --- |
| FastRGF\_Classifier <- R6::R6Class( |
|  | "FastRGF\_Classifier", |
|  | inherit = Internal\_class, |
|  | lock\_objects = FALSE, |
|  | public = list( |
|  |  |
|  | initialize = function(n\_estimators = 500 |
|  | , max\_depth = 6 |
|  | , max\_leaf = 50 |
|  | , tree\_gain\_ratio = 1.0 |
|  | , min\_samples\_leaf = 5 |
|  | , loss = "LS" |
|  | , l1 = 1.0 |
|  | , l2 = 1000.0 |
|  | , opt\_algorithm = "rgf" |
|  | , learning\_rate = 0.001 |
|  | , max\_bin = NULL |
|  | , min\_child\_weight = 5.0 |
|  | , data\_l2 = 2.0 |
|  | , sparse\_max\_features = 80000 |
|  | , sparse\_min\_occurences = 5 |
|  | , calc\_prob = "sigmoid" |
|  | , n\_jobs = 1 |
|  | , verbose = 0 |
|  | ) { |
|  |  |
|  | # exceptions for 'min\_samples\_leaf', 'max\_bin' |
|  | #--------------------------------------------- |
|  |  |
|  | # must be either > as.integer(1.0) or in (0, 0.5] |
|  | if (min\_samples\_leaf >= 1.0) { |
|  | min\_samples\_leaf <- as.integer(min\_samples\_leaf) |
|  | } |
|  |  |
|  | # must be either NULL or an integer |
|  | if (!is.null(max\_bin)) { |
|  | max\_bin <- as.integer(max\_bin) |
|  | } |
|  |  |
|  | # initialize FastRGF\_Classifier |
|  | #------------------------------ |
|  | private$rgf\_init = RGF\_mod$FastRGFClassifier( |
|  | n\_estimators = as.integer(n\_estimators) |
|  | , max\_depth = as.integer(max\_depth) |
|  | , max\_leaf = as.integer(max\_leaf) |
|  | , tree\_gain\_ratio = tree\_gain\_ratio |
|  | , min\_samples\_leaf = min\_samples\_leaf |
|  | , loss = loss |
|  | , l1 = l1 |
|  | , l2 = l2 |
|  | , opt\_algorithm = opt\_algorithm |
|  | , learning\_rate = learning\_rate |
|  | , max\_bin = max\_bin |
|  | , min\_child\_weight = min\_child\_weight |
|  | , data\_l2 = data\_l2 |
|  | , sparse\_max\_features = as.integer(sparse\_max\_features) |
|  | , sparse\_min\_occurences = as.integer(sparse\_min\_occurences) |
|  | , calc\_prob = calc\_prob |
|  | , n\_jobs = as.integer(n\_jobs) |
|  | , verbose = as.integer(verbose) |
|  | ) |
|  | } |
|  | ) |
|  | ) |

Code Chunks – First RGF Regressor

|  |
| --- |
| FastRGF\_Regressor <- R6::R6Class( |
|  | "FastRGF\_Regressor", |
|  | inherit = Internal\_class, |
|  | lock\_objects = FALSE, |
|  | public = list( |
|  |  |
|  | initialize = function(n\_estimators = 500 |
|  | , max\_depth = 6 |
|  | , max\_leaf = 50 |
|  | , tree\_gain\_ratio = 1.0 |
|  | , min\_samples\_leaf = 5 |
|  | , l1 = 1.0 |
|  | , l2 = 1000.0 |
|  | , opt\_algorithm = "rgf" |
|  | , learning\_rate = 0.001 |
|  | , max\_bin = NULL |
|  | , min\_child\_weight = 5.0 |
|  | , data\_l2 = 2.0 |
|  | , sparse\_max\_features = 80000 |
|  | , sparse\_min\_occurences = 5 |
|  | , n\_jobs = 1 |
|  | , verbose = 0 |
|  | ) { |
|  |  |
|  | # exceptions for 'min\_samples\_leaf', 'max\_bin' |
|  | #--------------------------------------------- |
|  |  |
|  | # must be either > as.integer(1.0) or in (0, 0.5] |
|  | if (min\_samples\_leaf >= 1.0) { |
|  | min\_samples\_leaf <- as.integer(min\_samples\_leaf) |
|  | } |
|  |  |
|  | # must be either NULL or an integer |
|  | if (!is.null(max\_bin)) { |
|  | max\_bin <- as.integer(max\_bin) |
|  | } |
|  |  |
|  | # initialize FastRGF\_Regressor |
|  | #----------------------------- |
|  | private$rgf\_init <- RGF\_mod$FastRGFRegressor( |
|  | n\_estimators = as.integer(n\_estimators) |
|  | , max\_depth = as.integer(max\_depth) |
|  | , max\_leaf = as.integer(max\_leaf) |
|  | , tree\_gain\_ratio = tree\_gain\_ratio |
|  | , min\_samples\_leaf = min\_samples\_leaf |
|  | , l1 = l1 |
|  | , l2 = l2 |
|  | , opt\_algorithm = opt\_algorithm |
|  | , learning\_rate = learning\_rate |
|  | , max\_bin = max\_bin |
|  | , min\_child\_weight = min\_child\_weight |
|  | , data\_l2 = data\_l2 |
|  | , sparse\_max\_features = as.integer(sparse\_max\_features) |
|  | , sparse\_min\_occurences = as.integer(sparse\_min\_occurences) |
|  | , n\_jobs = as.integer(n\_jobs) |
|  | , verbose = as.integer(verbose) |
|  | ) |
|  | } |
|  | ) |
|  | ) |

Code Chunks - Internal Class

|  |
| --- |
| Internal\_class <- R6::R6Class( |
|  | "Internal\_class", |
|  | lock\_objects = FALSE, |
|  | public = list( |
|  |  |
|  | # 'fit' function |
|  | #---------------- |
|  | fit = function(x, y, sample\_weight = NULL) { |
|  | private$rgf\_init$fit(x, y, sample\_weight) |
|  | return(invisible(NULL)) |
|  | }, |
|  |  |
|  | # 'predict' function |
|  | #-------------------- |
|  | predict = function(x) { |
|  | return(private$rgf\_init$predict(x)) |
|  | }, |
|  |  |
|  | # 'predict' function [ probabilities ] |
|  | #-------------------- |
|  | predict\_proba = function(x) { |
|  | return(private$rgf\_init$predict\_proba(x)) |
|  | }, |
|  |  |
|  | # 'cleanup' function |
|  | #------------------- |
|  | cleanup = function() { |
|  | private$rgf\_init$cleanup() |
|  | return(invisible(NULL)) |
|  | }, |
|  |  |
|  | # 'get\_params' function |
|  | #---------------------- |
|  | get\_params = function(deep = TRUE) { |
|  | return(private$rgf\_init$get\_params(deep)) |
|  | }, |
|  |  |
|  | # score function |
|  | #--------------- |
|  | score = function(x, y, sample\_weight = NULL) { |
|  | return(private$rgf\_init$score(x, y, sample\_weight)) |
|  | }, |
|  |  |
|  | # feature importance |
|  | #------------------- |
|  | feature\_importances = function() { |
|  | return(private$rgf\_init$feature\_importances\_) |
|  | }, |
|  |  |
|  | # dump-model |
|  | #----------- |
|  | dump\_model = function() { |
|  | return(private$rgf\_init$dump\_model) |
|  | }, |
|  |  |
|  | # save\_model |
|  | #----------- |
|  | save\_model = function(filename) { |
|  | private$rgf\_init$save\_model(filename) |
|  | return(invisible(NULL)) |
|  | } |
|  | ), |
|  |  |
|  | private = list( |
|  | rgf\_init = NULL |
|  | ) |
|  | ) |

Code Chunks – RGF Classifier

|  |
| --- |
| RGF\_Classifier <- R6::R6Class( |
|  | "RGF\_Classifier", |
|  | inherit = Internal\_class, |
|  | lock\_objects = FALSE, |
|  | public = list( |
|  |  |
|  | initialize = function(max\_leaf = 1000 |
|  | , test\_interval = 100 |
|  | , algorithm = "RGF" |
|  | , loss = "Log" |
|  | , reg\_depth = 1.0 |
|  | , l2 = 0.1 |
|  | , sl2 = NULL |
|  | , normalize = FALSE |
|  | , min\_samples\_leaf = 10 |
|  | , n\_iter = NULL |
|  | , n\_tree\_search = 1 |
|  | , opt\_interval = 100 |
|  | , learning\_rate = 0.5 |
|  | , calc\_prob = "sigmoid" |
|  | , n\_jobs = 1 |
|  | , memory\_policy = "generous" |
|  | , verbose = 0 |
|  | , init\_model = NULL |
|  | ) { |
|  |  |
|  | # exceptions for 'min\_samples\_leaf', 'n\_iter' |
|  | #-------------------------------------------- |
|  |  |
|  | # must be either > as.integer(1.0) or in (0, 0.5] |
|  | if (min\_samples\_leaf >= 1.0) { |
|  | min\_samples\_leaf <- as.integer(min\_samples\_leaf) |
|  | } |
|  |  |
|  | # must be either NULL or an integer |
|  | if (!is.null(n\_iter)) { |
|  | n\_iter <- as.integer(n\_iter) |
|  | } |
|  |  |
|  | # initialize RGF\_Classifier |
|  | #-------------------------- |
|  | private$rgf\_init <- RGF\_mod$RGFClassifier( |
|  | max\_leaf = as.integer(max\_leaf) |
|  | , test\_interval = as.integer(test\_interval) |
|  | , algorithm = algorithm |
|  | , loss = loss |
|  | , reg\_depth = reg\_depth |
|  | , l2 = l2 |
|  | , sl2 = sl2 |
|  | , normalize = normalize |
|  | , min\_samples\_leaf = min\_samples\_leaf |
|  | , n\_iter = n\_iter |
|  | , n\_tree\_search = as.integer(n\_tree\_search) |
|  | , opt\_interval = as.integer(opt\_interval) |
|  | , learning\_rate = learning\_rate |
|  | , calc\_prob = calc\_prob |
|  | , n\_jobs = as.integer(n\_jobs) |
|  | , memory\_policy = memory\_policy |
|  | , verbose = as.integer(verbose) |
|  | , init\_model = init\_model |
|  | ) |
|  | } |
|  | ) |
|  | ) |

Code Chunks – RGF Regressor

|  |
| --- |
| RGF\_Regressor <- R6::R6Class( |
|  | "RGF\_Regressor", |
|  | inherit = Internal\_class, |
|  | lock\_objects = FALSE, |
|  | public = list( |
|  |  |
|  | initialize = function(max\_leaf = 500 |
|  | , test\_interval = 100 |
|  | , algorithm = "RGF" |
|  | , loss = "LS" |
|  | , reg\_depth = 1.0 |
|  | , l2 = 0.1 |
|  | , sl2 = NULL |
|  | , normalize = TRUE |
|  | , min\_samples\_leaf = 10 |
|  | , n\_iter = NULL |
|  | , n\_tree\_search = 1 |
|  | , opt\_interval = 100 |
|  | , learning\_rate = 0.5 |
|  | , memory\_policy = "generous" |
|  | , verbose = 0 |
|  | , init\_model = NULL |
|  | ){ |
|  |  |
|  | # exceptions for 'min\_samples\_leaf', 'n\_iter' |
|  | #-------------------------------------------- |
|  |  |
|  | # must be either > as.integer(1.0) or in (0, 0.5] |
|  | if (min\_samples\_leaf >= 1.0) { |
|  | min\_samples\_leaf <- as.integer(min\_samples\_leaf) |
|  | } |
|  |  |
|  | # must be either NULL or an integer |
|  | if (!is.null(n\_iter)) { |
|  | n\_iter <- as.integer(n\_iter) |
|  | } |
|  |  |
|  | # initialize RGF\_Regressor |
|  | #------------------------ |
|  | private$rgf\_init <- RGF\_mod$RGFRegressor( |
|  | max\_leaf = as.integer(max\_leaf) |
|  | , test\_interval = as.integer(test\_interval) |
|  | , algorithm = algorithm |
|  | , loss = loss |
|  | , reg\_depth = reg\_depth |
|  | , l2 = l2 |
|  | , sl2 = sl2 |
|  | , normalize = normalize |
|  | , min\_samples\_leaf = min\_samples\_leaf |
|  | , n\_iter = n\_iter |
|  | , n\_tree\_search = as.integer(n\_tree\_search) |
|  | , opt\_interval = as.integer(opt\_interval) |
|  | , learning\_rate = learning\_rate |
|  | , memory\_policy = memory\_policy |
|  | , verbose = as.integer(verbose) |
|  | , init\_model = init\_model |
|  | ) |
|  | } |
|  | ) |
|  | ) |

RGF Clean Temp Files

|  |
| --- |
| RGF\_cleanup\_temp\_files = function() { |
|  |  |
|  | RGF\_utils$cleanup() |
|  |  |
|  | invisible() |
|  | } |

Code Chunks – SCIPY Parse

|  |
| --- |
| TO\_scipy\_sparse = function(R\_sparse\_matrix) { |
|  |  |
|  | if (inherits(R\_sparse\_matrix, "dgCMatrix")) { |
|  | py\_obj <- SCP$sparse$csc\_matrix( |
|  | reticulate::tuple( |
|  | R\_sparse\_matrix@x |
|  | , R\_sparse\_matrix@i |
|  | , R\_sparse\_matrix@p |
|  | ) |
|  | , shape = reticulate::tuple( |
|  | R\_sparse\_matrix@Dim[1] |
|  | , R\_sparse\_matrix@Dim[2] |
|  | ) |
|  | ) |
|  | } |
|  |  |
|  | else if (inherits(R\_sparse\_matrix, "dgRMatrix")) { |
|  |  |
|  | py\_obj <- SCP$sparse$csr\_matrix( |
|  | reticulate::tuple( |
|  | R\_sparse\_matrix@x |
|  | , R\_sparse\_matrix@j |
|  | , R\_sparse\_matrix@p |
|  | ) |
|  | , shape = reticulate::tuple( |
|  | R\_sparse\_matrix@Dim[1] |
|  | , R\_sparse\_matrix@Dim[2] |
|  | ) |
|  | ) |
|  | } |
|  |  |
|  | else { |
|  | stop("the 'R\_sparse\_matrix' parameter should be either a 'dgCMatrix' or a 'dgRMatrix' sparse matrix", call. = FALSE) |
|  | } |
|  |  |
|  | return(py\_obj) |
|  | } |

Code Chunks – Conversion of an object to Scipy Parse Matrix

|  |
| --- |
| #' conversion of an R matrix to a scipy sparse matrix |
|  | #' |
|  | #' |
|  | #' @param x a data matrix |
|  | #' @param format a character string. Either \emph{"sparse\_row\_matrix"} or \emph{"sparse\_column\_matrix"} |
|  | #' @details |
|  | #' This function allows the user to convert an R matrix to a scipy sparse matrix. This is useful because the Regularized Greedy Forest algorithm accepts only python sparse matrices as input. |
|  | #' @export |
|  | #' @references https://docs.scipy.org/doc/scipy/reference/sparse.html |
|  | #' @examples |
|  | #' |
|  | #' try({ |
|  | #' if (reticulate::py\_available(initialize = FALSE)) { |
|  | #' if (reticulate::py\_module\_available("scipy")) { |
|  | #' |
|  | #' library(RGF) |
|  | #' |
|  | #' set.seed(1) |
|  | #' |
|  | #' x = matrix(runif(1000), nrow = 100, ncol = 10) |
|  | #' |
|  | #' res = mat\_2scipy\_sparse(x) |
|  | #' |
|  | #' print(dim(x)) |
|  | #' |
|  | #' print(res$shape) |
|  | #' } |
|  | #' } |
|  | #' }, silent = TRUE) |
|  |  |
|  | mat\_2scipy\_sparse = function(x, format = 'sparse\_row\_matrix') { |
|  |  |
|  | if (!inherits(x, "matrix")) { |
|  | stop("the 'x' parameter should be of type 'matrix'", call. = FALSE) |
|  | } |
|  |  |
|  | if (format == 'sparse\_column\_matrix') { |
|  |  |
|  | return(SCP$sparse$csc\_matrix(x)) |
|  |  |
|  | } else if (format == 'sparse\_row\_matrix') { |
|  |  |
|  | return(SCP$sparse$csr\_matrix(x)) |
|  |  |
|  | } else { |
|  |  |
|  | stop("the function can take either a 'sparse\_row\_matrix' or a 'sparse\_column\_matrix' for the 'format' parameter as input", call. = FALSE) |
|  | } |
|  | } |

Code Chunks – Load The Python Modules

|  |
| --- |
| RGF\_mod <- NULL; RGF\_utils <- NULL; SCP <- NULL; |
|  |  |
|  |  |
|  | .onLoad <- function(libname, pkgname) { |
|  |  |
|  | try({ |
|  | if (reticulate::py\_available(initialize = TRUE)) { |
|  |  |
|  | try({ |
|  | RGF\_mod <<- reticulate::import("rgf.sklearn", delay\_load = TRUE) |
|  | }, silent = TRUE) |
|  |  |
|  | try({ |
|  | RGF\_utils <<- reticulate::import("rgf.utils", delay\_load = TRUE) |
|  | }, silent = TRUE) |
|  |  |
|  | try({ |
|  | SCP <<- reticulate::import("scipy", delay\_load = TRUE, convert = FALSE) |
|  | }, silent = TRUE) |
|  |  |
|  | } |
|  | }, silent=TRUE) |
|  | } |
|  |  |
|  |  |
|  | .onAttach <- function(libname, pkgname) { |
|  | packageStartupMessage("If the 'RGF' package gives the following error: 'attempt to apply non-function' then make sure to open a new R session and run 'reticulate::py\_config()' before loading the package!") |
|  | } |

**The RGF package**

The *RGF* package includes the following R6-classes / functions,

**classes**

| **RGF\_Regressor** | **RGF\_Classifier** | **FastRGF\_Regressor** | **FastRGF\_Classifier** |
| --- | --- | --- | --- |
| fit() | fit(() | fit() | fit() |
| predict() | predict() | predict() | predict() |
| cleanup() | predict\_proba() | cleanup() | predict\_proba() |
| get\_params() | cleanup() | get\_params() | cleanup() |
| score() | get\_params() | score() | get\_params() |
|  | score() |  | score() |

**functions**

dgCMatrix\_2scipy\_sparse()

RGF\_cleanup\_temp\_files()

mat\_2scipy\_sparse()

The package documentation includes details and examples for all R6-classes and functions. In the following code chunks, I’ll explain how a user can work with sparse matrices as all RGF algorithms (besides a dense matrix) **require a python sparse matrix as input**.

**Sparse matrices as input**

The RGF package includes two functions (**mat\_2scipy\_sparse** and **dgCMatrix\_2scipy\_sparse**) which allow the user to convert from a *matrix* / *dgCMatrix* to a *scipy sparse matrix*,

library(RGF)

# conversion from a matrix object to a scipy sparse matrix

#----------------------------------------------------------

set.seed(1)

x = matrix(runif(1000), nrow = 100, ncol = 10)

x\_sparse = mat\_2scipy\_sparse(x, format = "sparse\_row\_matrix")

print(dim(x))

[1] 100 10

print(x\_sparse$shape)

(100, 10)

# conversion from a dgCMatrix object to a scipy sparse matrix

#-------------------------------------------------------------

data = c(1, 0, 2, 0, 0, 3, 4, 5, 6)

# by default column-oriented format

dgcM = Matrix::Matrix(data = data, nrow = 3,

ncol = 3, byrow = TRUE,

sparse = TRUE)

print(dim(dgcM))

[1] 3 3

x\_sparse = dgCMatrix\_2scipy\_sparse(dgcM)

print(x\_sparse$shape)

(3, 3)

**Comparison of RGF with ranger and xgboost**

First the data, libraries and cross-validation function will be inputted (the *MLmetrics* library is also required),

data(Boston, package = 'KernelKnn')

library(RGF)

library(ranger)

library(xgboost)

# shuffling function for cross-validation folds

#-----------------------------------------------

func\_shuffle = function(vec, times = 10) {

for (i in 1:times) {

out = sample(vec, length(vec))

}

out

}

# cross-validation folds [ regression]

#-------------------------------------

regr\_folds = function(folds, RESP, stratified = FALSE) {

if (is.factor(RESP)) {

stop(simpleError("this function is meant for regression for classification use the 'class\_folds' function"))

}

samp\_vec = rep(1/folds, folds)

sort\_names = paste0('fold\_', 1:folds)

if (stratified == TRUE) {

stratif = cut(RESP, breaks = folds)

clas = lapply(unique(stratif), function(x) which(stratif == x))

len = lapply(clas, function(x) length(x))

prop = lapply(len, function(y) sapply(1:length(samp\_vec), function(x) round(y \* samp\_vec[x])))

repl = unlist(lapply(prop, function(x) sapply(1:length(x), function(y) rep(paste0('fold\_', y), x[y]))))

spl = suppressWarnings(split(1:length(RESP), repl))}

else {

prop = lapply(length(RESP), function(y) sapply(1:length(samp\_vec), function(x) round(y \* samp\_vec[x])))

repl = func\_shuffle(unlist(lapply(prop, function(x) sapply(1:length(x), function(y) rep(paste0('fold\_', y), x[y])))))

spl = suppressWarnings(split(1:length(RESP), repl))

}

spl = spl[sort\_names]

if (length(table(unlist(lapply(spl, function(x) length(x))))) > 1) {

warning('the folds are not equally split')

}

if (length(unlist(spl)) != length(RESP)) {

stop(simpleError("the length of the splits are not equal with the length of the response"))

}

spl

}

**single threaded ( small data set )**

In the next code chunk, I’ll perform 5-fold cross-validation using the Boston dataset and I’ll compare time execution and error rate for all three algorithms (comparison **without doing hyper-parameter tuning**),

NUM\_FOLDS = 5

set.seed(1)

FOLDS = regr\_folds(folds = NUM\_FOLDS, Boston[, 'medv'], stratified = T)

boston\_rgf\_te = boston\_ranger\_te = boston\_xgb\_te = boston\_rgf\_time = boston\_ranger\_time = boston\_xgb\_time = rep(NA, NUM\_FOLDS)

for (i in 1:length(FOLDS)) {

cat("fold : ", i, "\n")

samp = unlist(FOLDS[-i])

samp\_ = unlist(FOLDS[i])

# RGF

#----

rgf\_start = Sys.time()

init\_regr = RGF\_Regressor$new(l2 = 0.1)

init\_regr$fit(x = as.matrix(Boston[samp, -ncol(Boston)]), y = Boston[samp, ncol(Boston)])

pr\_te = init\_regr$predict(as.matrix(Boston[samp\_, -ncol(Boston)]))

rgf\_end = Sys.time()

boston\_rgf\_time[i] = rgf\_end - rgf\_start

boston\_rgf\_te[i] = MLmetrics::RMSE(Boston[samp\_, 'medv'], pr\_te)

# ranger

#-------

ranger\_start = Sys.time()

fit = ranger(dependent.variable.name = "medv", data = Boston[samp, ], write.forest = TRUE,

probability = F, num.threads = 1, num.trees = 500, verbose = T,

classification = F, mtry = NULL, min.node.size = 5, keep.inbag = T)

pred\_te = predict(fit, data = Boston[samp\_, -ncol(Boston)], type = 'se')$predictions

ranger\_end = Sys.time()

boston\_ranger\_time[i] = ranger\_end - ranger\_start

boston\_ranger\_te[i] = MLmetrics::RMSE(Boston[samp\_, 'medv'], pred\_te)

# xgboost

#--------

xgb\_start = Sys.time()

dtrain <- xgb.DMatrix(data = as.matrix(Boston[samp, -ncol(Boston)]), label = Boston[samp, ncol(Boston)])

dtest <- xgb.DMatrix(data = as.matrix(Boston[samp\_, -ncol(Boston)]), label = Boston[samp\_, ncol(Boston)])

watchlist <- list(train = dtrain, test = dtest)

param = list("objective" = "reg:linear", "bst:eta" = 0.05, "max\_depth" = 4,

"subsample" = 0.85, "colsample\_bytree" = 0.85, "booster" = "gbtree",

"nthread" = 1)

fit = xgb.train(param, dtrain, nround = 500, print\_every\_n = 100, watchlist = watchlist, early\_stopping\_rounds = 20,

maximize = FALSE, verbose = 0)

p\_te = xgboost:::predict.xgb.Booster(fit, as.matrix(Boston[samp\_, -ncol(Boston)]), ntreelimit = fit$best\_iteration)

xgb\_end = Sys.time()

boston\_xgb\_time[i] = xgb\_end - xgb\_start

boston\_xgb\_te[i] = MLmetrics::RMSE(Boston[samp\_, 'medv'], p\_te)

}

fold : 1

fold : 2

fold : 3

fold : 4

fold : 5

cat("total time rgf 5 fold cross-validation : ", sum(boston\_rgf\_time), " mean rmse on test data : ", mean(boston\_rgf\_te), "\n")

cat("total time ranger 5 fold cross-validation : ", sum(boston\_ranger\_time), " mean rmse on test data : ", mean(boston\_ranger\_te), "\n")

cat("total time xgb 5 fold cross-validation : ", sum(boston\_xgb\_time), " mean rmse on test data : ", mean(boston\_xgb\_te), "\n")

total time rgf 5 fold cross-validation : 0.7730639 mean rmse on test data : 3.832135

total time ranger 5 fold cross-validation : 3.826846 mean rmse on test data : 4.17419

total time xgb 5 fold cross-validation : 0.4316094 mean rmse on test data : 3.949122

**5 threads ( high dimensional dataset and presence of multicollinearity )**

For the high-dimensional data I’ll use the *FastRGF\_Regressor* rather than the RGF\_Regressor (comparison **without doing hyper-parameter tuning**),

DataSets

To download the .zip files \*\*on a linux OS\*\* from within R use :  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/mnist.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/cifar_10.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/africa_soil_train_data.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/sift_10k.txt>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/BSR_bsds500.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/fastText_data.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/wili-2018.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/declaration_human_rights.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/lid.176.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/grid30.zip>")  
  
system("wget <https://raw.githubusercontent.com/mlampros/DataSets/master/grid90.zip>")

Library(RGF)

# download the data from my Github repository (tested on a Linux OS)

system("wget https://raw.githubusercontent.com/mlampros/DataSets/master/africa\_soil\_train\_data.zip")

# load the data in the R session

train\_dat = read.table(unz("africa\_soil\_train\_data.zip", "train.csv"), nrows = 1157, header = T, quote = "\"", sep = ",")

# c("Ca", "P", "pH", "SOC", "Sand") : response variables

# exclude response-variables and factor variable

x = train\_dat[, -c(1, which(colnames(train\_dat) %in% c("Ca", "P", "pH", "SOC", "Sand", "Depth")))]

# take (randomly) the first of the responses for train

y = train\_dat[, "Ca"]

# dataset for ranger

tmp\_rg\_dat = cbind(Ca = y, x)

# cross-validation folds

set.seed(2)

FOLDS = regr\_folds(folds = NUM\_FOLDS, y, stratified = T)

highdim\_rgf\_te = highdim\_ranger\_te = highdim\_xgb\_te = highdim\_rgf\_time = highdim\_ranger\_time = highdim\_xgb\_time = rep(NA, NUM\_FOLDS)

for (i in 1:length(FOLDS)) {

cat("fold : ", i, "\n")

new\_samp = unlist(FOLDS[-i])

new\_samp\_ = unlist(FOLDS[i])

# RGF

#----

rgf\_start = Sys.time()

init\_regr = FastRGF\_Regressor$new(n\_jobs = 5, l2 = 0.1) # I added 'l2' regularization

init\_regr$fit(x = as.matrix(x[new\_samp, ]), y = y[new\_samp])

pr\_te = init\_regr$predict(as.matrix(x[new\_samp\_, ]))

rgf\_end = Sys.time()

highdim\_rgf\_time[i] = rgf\_end - rgf\_start

highdim\_rgf\_te[i] = MLmetrics::RMSE(y[new\_samp\_], pr\_te)

# ranger

#-------

ranger\_start = Sys.time()

fit = ranger(dependent.variable.name = "Ca", data = tmp\_rg\_dat[new\_samp, ],

write.forest = TRUE, probability = F, num.threads = 5, num.trees = 500,

verbose = T, classification = F, mtry = NULL, min.node.size = 5,

keep.inbag = T)

pred\_te = predict(fit, data = x[new\_samp\_, ], type = 'se')$predictions

ranger\_end = Sys.time()

highdim\_ranger\_time[i] = ranger\_end - ranger\_start

highdim\_ranger\_te[i] = MLmetrics::RMSE(y[new\_samp\_], pred\_te)

# xgboost

#--------

xgb\_start = Sys.time()

dtrain <- xgb.DMatrix(data = as.matrix(x[new\_samp, ]), label = y[new\_samp])

dtest <- xgb.DMatrix(data = as.matrix(x[new\_samp\_, ]), label = y[new\_samp\_])

watchlist <- list(train = dtrain, test = dtest)

param = list("objective" = "reg:linear", "bst:eta" = 0.05, "max\_depth" = 6,

"subsample" = 0.85, "colsample\_bytree" = 0.85, "booster" = "gbtree",

"nthread" = 5) # "lambda" = 0.1 does not improve RMSE

fit = xgb.train(param, dtrain, nround = 500, print\_every\_n = 100, watchlist = watchlist,

early\_stopping\_rounds = 20, maximize = FALSE, verbose = 0)

p\_te = xgboost:::predict.xgb.Booster(fit, as.matrix(x[new\_samp\_, ]), ntreelimit = fit$best\_iteration)

xgb\_end = Sys.time()

highdim\_xgb\_time[i] = xgb\_end - xgb\_start

highdim\_xgb\_te[i] = MLmetrics::RMSE(y[new\_samp\_], p\_te)

}

fold : 1

fold : 2

fold : 3

fold : 4

fold : 5

cat("total time rgf 5 fold cross-validation : ", sum(highdim\_rgf\_time), " mean rmse on test data : ", mean(highdim\_rgf\_te), "\n")

cat("total time ranger 5 fold cross-validation : ", sum(highdim\_ranger\_time), " mean rmse on test data : ", mean(highdim\_ranger\_te), "\n")

cat("total time xgb 5 fold cross-validation : ", sum(highdim\_xgb\_time), " mean rmse on test data : ", mean(highdim\_xgb\_te), "\n")

total time rgf 5 fold cross-validation : 92.31971 mean rmse on test data : 0.5155166

total time ranger 5 fold cross-validation : 27.32866 mean rmse on test data : 0.5394164

total time xgb 5 fold cross-validation : 30.48834 mean rmse on test data : 0.5453544

The *README.md* file of the *RGF* package includes the SystemRequirements and installation instructions.