This blog post is about my newly released [RGF package](https://github.com/mlampros/RGF) (the blog post consists mainly of the package Vignette). The *RGF* package is a wrapper of the [*Regularized Greedy Forest*](https://github.com/fukatani/rgf_python) *python* package, which also includes a [*Multi-core implementation (FastRGF)*](https://github.com/baidu/fast_rgf). Portability from Python to R was made possible using the [reticulate](https://github.com/rstudio/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*). Detailed installation instructions for all three Operating Systems can be found in the *README.md* file and in the [*rgf\_python*](https://github.com/fukatani/rgf_python) Github repository.

The *Regularized Greedy Forest* algorithm is explained in detail in the paper [*Rie Johnson and Tong Zhang, Learning Nonlinear Functions Using Regularized Greedy Forest*](https://arxiv.org/abs/1109.0887). 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*](https://github.com/imbs-hl/ranger) (random forest algorithm) and [*xgboost*](https://github.com/dmlc/xgboost/tree/master/R-package) (gradient boosting algorithm), in terms of time efficiency and error rate improvement.

**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 (can be downloaded from my [Github repository](https://github.com/mlampros/DataSets)) I’ll use the *FastRGF\_Regressor* rather than the RGF\_Regressor (comparison **without doing hyper-parameter tuning**),

# 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.