Speed up XGBoost optimization using parallel computing in R

Álvaro Orgaz Expósito

OBJECTIVE

The aim of this paper is to reduce computation time in R without using GPU or *Big Data* tools, just using all cores available in the CPU (by default R uses one single core). Basically, we will compare the required time for developing a *data science* task with the default R configuration and with parallel computing configuration.

The task will be to find the optimal combination (higher *AUC* metric) of hyperparameters for the popular *machine learning* algorithm *XGBoost*, which is known as *hyperparameters grid search*. Although this algorithm has more hyperparameters, we will try several combinations of the parameters: *eta* or learning rate, *max_depth* or maximum depth of trees, and *nround* or maximum number of boosting iteration. About the data, we will use the popular dataset *titanic_train* in R library *titanic*. Let's predict the binary variable *Survived* with the explanatory features: *Fare*, *Age*, *Sex*, *Embarked*, *Parch*, *Pclass* and *SibSp*.

CODE

Firstly, we need to load the data and preprocess it. Basically, the code includes:

1. Load the dataset *titanic_train*
2. Preprocess the data to recode properly the levels of the categorical features
3. Create a vector with the target variable y and a matrix (necessary for next steps) x with the preprocessed features

```
# 1
library(titanic)
data <- titanic_train
target <- "Survived"
categorical <- c("Sex","Embarked","Parch","Pclass","SibSp")
numerical <- c("Age","Fare")
data <- data[,c(target,categorical,numerical)]
# 2
valid_levels <- list(
  levels_Survived = c("0","1"),
  levels_Sex = c("female","male"),
  levels_Embarked = c("C","Q","S"),
  levels_Parch =  c("0","1","2","3","4","5","6"),
  levels_Pclass = c("1","2","3"),
  levels_SibSp = c("0","1","2","3","4","5","6","7","8")
)
for(i in categorical){
  level <- paste0("levels_",i)
  data[,i] <- as.numeric(factor(data[,i],levels=valid_levels[[level]],exclude=NULL))
}
# 3
y <- data[,target]
x <- data.matrix(data[,c(categorical,numerical)])
```

Secondly, let's define all the hyperparameter combinations that we want to study in the object *parametrizations* and save it in the file *data.RData.file* together with the data objects x and y.

```
etas <- c(0.01,0.05,0.1,0.15,0.2)
max_depths <- c(5,10,15,20,25)
nrounds <- c(50,100,200,300,400)
parametrizations <- expand.grid(list(eta=etas,max_depth=max_depths,nround=nrounds))
save(x,y,parametrizations,file="data.RData")
```

Thirdly, let's define the function to loop. What does it mean? We will use the function *llply* in *plyr* R package for parallel computing, and its structure is like *llply(data,fun,.parallel=TRUE)* where *fun* is the function to apply for each element in *data*. Then, we will iterate all the defined hyperparameters combinations and apply in each iteration a function that trains a *XGBoost* model and calculate the accuracy metric *AUC* with its predictions.

Note: When using the option *.parallel=TRUE* in the function *llply*, we cannot use objects in the actual session as x, y or *parametrization*. For this reason, we saved them in the file *data.RData* and load it inside the function to loop.

```
auxiliary <- function(i){
  load("data.RData")
  library(xgboost)
  model <- xgboost(data=xgb.DMatrix(data=x,label=y),
                   eta=parametrizations[i,"eta"],
                   max_depth=parametrizations[i,"max_depth"],
                   nround=parametrizations[i,"nround"],
                   objective="binary:logistic",
                   verbose=0)
  library(pROC)
  return(auc(roc(y,predict(model,x))))
}
```

Once the looping function is created, let's activate all available cores in the R session using R packages *parallel* and *doParallel*.

```
max_threads <- parallel::detectCores()
cluster <- parallel::makeCluster(max_threads)
doParallel::registerDoParallel(cluster)
```

Finally, let's compute in parallel the *AUC* value of all *XGBoost* parametrizations using the created function *auxiliar* and the function *llply* from R package *plyr* which use parallel backend provided by the previous activation.

```
start <- Sys.time()
auc_parallel <- plyr::llply(1:nrow(parametrizations),auxiliary,.parallel=TRUE)
Sys.time()-start
```

```
## Time difference of 1.06985 mins
```

Now, we can do the same job but desactivating the parallel option, just using the default configuration in R which only uses one core.

```
start <- Sys.time()
auc <- plyr::llply(1:nrow(parametrizations),auxiliary,.parallel=FALSE)
Sys.time()-start
```

```
## Time difference of 2.822773 mins
```

Note: If you open the task manager in your machine while parallel computing, you will see that the CPU

is working almost at 100% level. But if you open it while default computing, you will see that the CPU is working at a lower level than before.

Let's make a simple check to ensure that all outputs (*AUC* metric for every hyperparameters combination) are equal in parallel and non-parallel procedures.

```
unlist(auc)==unlist(auc_parallel)
```

```
##   [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [15] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [29] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [43] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [57] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [71] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [85] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [99] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [113] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Then, the optimal *hyperparameters* are the same in both procedures.

```
optimal <- which(unlist(auc)==max(unlist(auc)))
parametrizations[optimal,]
```

```
##     eta max_depth nround
## 125 0.2        25    400
```

```
optimal_parallel <- which(unlist(auc_parallel)==max(unlist(auc_parallel)))
parametrizations[optimal_parallel,]
```

```
##     eta max_depth nround
## 125 0.2        25    400
```

Conclusion, we can drastically reduce the required time to find the optimal combination of hyperparameters for a *machine learning* algorithm such as *XGBoost*. However, this can be applied to any computing objective just changing or adapting the *auxiliary* function.