**H2O + AWS + purrr (Part III)**

This is the final installment of a three part series that looks at how we can leverage AWS, H2O and purrr in R to build analytical pipelines. In the previous posts I looked at starting up the environment through the EC2 dashboard on AWS’ website. The other aspect we looked at, in Part II, was how we can use purrr to train models using H2O’s awesome api. Here are the links to these first two posts:

* [Part I](https://www.daeconomist.com/post/2018-12011-aws-parti/)
* [Part II](https://www.daeconomist.com/post/2019-01-15-partii/)

For the final blogpost, I will look at how we can use boto3 to start up an EC2 instance programmatically and then connect to it through furrr. For this post I will spin up a spot EC2 instance and show you 2 types of calculations:

* Simple sum function
* H2O training function

**Do Note** I will not show how to set up reticulate (which we need for boto3), but [here](https://rstudio.github.io/reticulate/) is a good place to start if you struggling.

The packages we will use in this post:

* library(tidyverse)
* library(reticulate)
* library(furrr)
* library(glue)
* library(tictoc)[1](https://www.daeconomist.com/post/2019-01-25-partiii/#fn:This-will-just-b)

**Use boto3 to start EC2 instance**

To start up an instance remotely, you need to specify your AWS credentials somewhere. I find the easiest way to do this is to place the credentials in your R environment using usethis::edit\_r\_environ(). You will need to specify 3 separate variables:

* AWS\_ACCESS\_KEY\_ID=XXXXXXXXXXX
* AWS\_SECRET\_ACCESS\_KEY=XXXXXXXXXXXXXX
* AWS\_DEFAULT\_REGION=us-east-2

To generate these, you can log into your AWS dashboard, go to the IAM (Identity and Access Management) dashboard and select the Users tab. On the Users tab, add a user and also the administration rights that you want the user to have. *Remember to restart R once you have filled in the access key information in the .Renviron file for it to take effect.*

At this point, those familiar with cloudyr suite is probably asking – “This is exactly the same as library(aws.ec2), so why use boto3?“. Well, to be honest, I was using aws.ec2 for a while, but I find spot-instances, which the current version of aws.ec2 does not support. In addition I found that boto3 has some other functionalitue – which I prefer. For a full list of boto3 functions to interact with an EC2 instance, have a look at the [reference manual](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html).

To import python’s boto3, we use reticulate::import:

boto <- import("boto3")

# Import EC2 client

ec2 <- boto$client('ec2')

ec2

The ec2 variable you see here is of class . It is what we will use to interact with AWS in a programmatic fashion. Next, lets kick off a new spot-instance. I will use the c4.4xlarge instance for the remainder of the post:

* 16 cores
* 30GB RAM
* EBS Only
* Spot Price: $0.1448/h

Note I also specify the ImageId that I want to use. If you remember from part I and part II, I built an AMI that already has H2O and caret installed: **ami-0157656a8c5b46458**. The function has multiple other options you can read up on in the [reference manual](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html). If you have a prefered SecurityGroupIds, you can also add that into the request as I did here.

response <- ec2$request\_spot\_instances(

InstanceCount = 1L,

LaunchSpecification= list(

ImageId = 'ami-0157656a8c5b46458',

InstanceType = 'c4.4xlarge',

SecurityGroupIds = list("sg-0147497a63ft8ae432"),

Placement = list(AvailabilityZone = "us-east-2a")

),

SpotPrice = '1',

Type = 'one-time'

)

The response variable returns a ton of information. The best way I found to deal with this is just to write a small function which captures the most useful information.

ec2\_info <- function(req\_id, type){

ec2\_all <- ec2$describe\_instances(Filters = list(

list(Name = "spot-instance-request-id",

Values = list(req\_id))))$Reservations[[1]]

if(type == "ip")

return(ec2\_all$Instances[[1]]$PublicIpAddress)

if(type == "instance\_id")

return(ec2\_all$Instances[[1]]$InstanceId)

if(type == "cores")

return(ec2\_all$Instances[[1]]$CpuOptions$CoreCount)

}

This function needs a request\_id as an input and will return either the ip, instance\_id or number of cores. It uses the id as input to a very useful filtering function as part of the EC2 module: describe\_instances. Again, this function can take multiple inputs, I prefer the request\_id.  
To get the request\_id we can use purrr:

xtract\_reqid <- function(response){

tibble(req\_id = response$SpotInstanceRequests %>%

map\_chr(~.x %>% .$SpotInstanceRequestId))

}

req\_id <- response %>% xtract\_reqid()

instance\_information <- req\_id %>%

mutate(

ip = map\_chr(req\_id, ec2\_info, "ip"),

instance\_id = map\_chr(req\_id, ec2\_info, "instance\_id"),

cores = map\_dbl(req\_id, ec2\_info, "cores")\*2

)

instance\_information

# Closing instance

# ec2$terminate\_instances(InstanceIds = list(instance\_information$instance\_id))

Now we have all the pieces, we can conduct some calculations using these machines. We will use furrr to connect. I will show you how to connect to the EC2 instances using a Windows example – the reason being that I found it a bit more tricky than when working on Linux. To see how we can connect when we operating off of a linux machine, see the reference by [Davis Vaughan, Matt Dancho](https://davisvaughan.github.io/furrr/articles/advanced-furrr-remote-connections.html).

Remember to specify where your private key file is located. If you don’t know what a private key is, I refer you to the [first post](https://www.daeconomist.com/post/2018-12011-aws-parti/) if you need some help.

public\_ip <- ec2\_info(req\_id, type = "ip")

ssh\_private\_key\_file <- "C:/Users/User/your\_keypair.ppk"

A very important change that you need to take note of here is the fact that I use [Plink](https://www.ssh.com/ssh/putty/putty-manuals/0.68/Chapter7.html) to connect. The Windows connection I feel is a bit unstable and finicky. So I do recommend you work off of a linux machine if possible.

# Connect!

cl <- future::makeClusterPSOCK(

## Public IP number of EC2 instance

public\_ip,

## User name (always 'ubuntu')

user = "ubuntu",

## Use private SSH key registered with AWS

rshcmd = c("plink", "-ssh", "-i", ssh\_private\_key\_file),

## Set up .libPaths() for the 'ubuntu' user and

## install future/purrr/furrr packages

rscript\_args = c(

"-e", shQuote("local({p <- Sys.getenv('R\_LIBS\_USER'); dir.create(p, recursive = TRUE, showWarnings = FALSE); .libPaths(p)})"),

"-e", shQuote("install.packages(c('future', 'furrr'))")

),

dryrun = F,

verbose = TRUE,

homogeneous = FALSE

)

The makeClusterPSOCK function will become your workhorse function to connect to remote servers. The biggest changes you will make will be in the rscript\_args argument of the function. As you will see later, we can add commands to this argument to initiate the H2O server we want to use for estimation.

To confirm that the processes are running in parallel and that we have an increase in speed, we will first run the sequence sequentially and then parallel. Something that was made clear to my by [Davis Vaughan](https://github.com/DavisVaughan/furrr/issues/58) on why we use nested future\_maps are because when you start spinning up more than 1 instance, you can shard the data across servers and then you must distribute it over cores. So imagine you want to train three models: rf, gbm and deeplearning. You can then spin up three different EC2 instances and distributed the training of the models across mutliple machines. But lets first try and get basic arithmetic going.

f <- function(x){

x <- rnorm(x)

rm(x)

return("success")

}

plan(list(tweak(cluster, workers = cl), tweak(sequential)))

tic()

x <- future\_map(1, ~{

rep(1e6, 500) %>%

future\_map(~f(.x))

})

sec\_time <- toc()

The sequential method takes around 40 seconds to complete. Lets use the plan function to do the next estimation in parallel:

plan(list(tweak(cluster, workers = cl), tweak(multiprocess, workers = 15)))

tic()

x <- future\_map(1, ~{

rep(1e6, 500) %>%

future\_map(~f(.x))

})

par\_time <- toc()

The time difference is staggering, 8 seconds in parallel. We can confirm that the process is running in parallel by SSHing into the remote machine and using htop.



**Using H2O with remote server**

Its all good and well doing random number generation uing a large machine, but in the end we would most likely want to redirect our compuational burden to the remote server, especially for machine learning estimations.

To do this we need to change one small line in the connection function: makeClusterPSOCK. I wrap the makeClusterPSOCK function so that it is easier to use. This new function is called h2o.makeEC2ClusterPsock and it will be our primary function to start and connect to remote server initializing h2o as it connects.

h2o.makeEC2ClusterPsock <- function(public\_ip, ssh\_private\_key\_file, max\_mem\_size = "10g", dryrun = F, h2o = T){

if(h2o){

r\_command <- c(

"-e", shQuote(".libPaths('/home/rstudio/R/x86\_64-pc-linux-gnu-library/3.5')"),

"-e", shQuote(glue("if (!require('furrr')) install.packages('furrr');library(h2oEnsemble);system('pkill java');h2o.init(nthreads = -1, max\_mem\_size = '{max\_mem\_size}');h2o.removeAll();gc()"))

)

} else {

r\_command <- c(

"-e", shQuote(".libPaths('/home/rstudio/R/x86\_64-pc-linux-gnu-library/3.5')"),

"-e", shQuote("if (!require('furrr')) install.packages('furrr')")

)

}

makeClusterPSOCK(

# Public IP number of EC2 instance

workers = public\_ip,

# User name (always 'ubuntu')

user = "ubuntu",

# Use private SSH key registered with AWS

rshcmd = c("plink", "-ssh", "-i", ssh\_private\_key\_file),

# Check available memory

# Set up .libPaths() for the 'ubuntu' user and

# install furrr

rscript\_args = r\_command,

# Switch this to TRUE to see the code that is run on the workers without

# making the connection

dryrun = dryrun,

verbose = T

)

}

We also need to remember to send the data to the remote server so it can be used in the estimation. This is where furrr and futures will be used. I am going to use the Higgs dataset from [PART II](https://www.daeconomist.com/post/2019-01-15-partii/) in the following example. Its important to coerce the data using the as.h2o function using a future\_map as this sends the data to the remote server.

train <- read\_csv("<https://s3.amazonaws.com/erin-data/higgs/higgs_train_5k.csv>") %>%

mutate(response = as.factor(response)) %>%

list

test <- read\_csv("<https://s3.amazonaws.com/erin-data/higgs/higgs_test_5k.csv>") %>%

mutate(response = as.factor(response)) %>%

list

cl <- h2o.makeEC2ClusterPsock(public\_ip = public\_ip, max\_mem\_size = "120g",

ssh\_private\_key\_file = ssh\_private\_key\_file, dryrun = F, h2o = T)

plan(cluster, workers = cl)

training\_df <- tibble(model = "gbm", train, test) %>%

mutate(h2o\_frame\_name = "train.hex",

h2o\_frame\_name\_test = "test.hex") %>%

##### USE FUTURE MAP TO SEND THE TRAINING DF TO REMOTE SERVER

mutate(train = future\_map2(train, h2o\_frame\_name, ~as.h2o(.x, destination\_frame = .y))) %>%

mutate(test = future\_map2(test, h2o\_frame\_name\_test, ~as.h2o(.x, destination\_frame = .y)))

We can confirm that the h2o server is running by using htop again. This time we should see a lot of java processes running. If we inspect the training\_df, you will notice that the class of the training and testing frames have been transformed to H2OFrame. These frames are now sitting on the *remote* server. How cool is that?

training\_df

# A tibble: 1 x 4

# train test h2o\_frame\_name h2o\_frame\_name\_test

#

# train.hex test.hex

The last step is to train this badboy.



I need to use the future\_map function to send the train commands to the remote server. The way to do this is to write a wrapper function for training a specific model.

h2o.model <- function(model, training, validation, Y, X, folds = 2){

if(model == "gbm"){

return(h2o.gbm(y = Y, x = X, training\_frame = training,

validation\_frame = validation,

ntrees = 100,

stopping\_metric = "AUTO",

stopping\_tolerance = 0.01,

max\_depth = 10,

min\_rows = 2,

learn\_rate = 0.1,

stopping\_rounds = 2,

learn\_rate\_annealing = 0.99,

nfolds = folds))

}

}

ml.Res <- training\_df %>%

mutate(trained\_models = future\_pmap(list(model, train, test),

~h2o.model(..1, training = ..2, validation = ..3,

Y = "response", X = setdiff(names(..2), "response"),

folds = 5)))

ml.Res$trained\_models

Your model has now been trained using H2O and AWS on a 16 core 30GB machine, all from your local workstation! This exercise can be obviously be expanded for much more intensive testing and experimentation. But for now, lets leave it there.

**Always always always remember to shut down the instance/s after you are finished**!

ec2$terminate\_instances(InstanceIds = list(instance\_information$instance\_id))

**Conclusion**

If you have gotten this far over the last few weeks, I hope that you have learned as much as I did on how to scale your analytics to almost any size. The power to harness really impressive machines for any estimation problem has changed the way I approach analytics and research in general. The problem is no longer bounded by the size of the machine I have, but rather my experimental design.

I am hoping to talk about this idea of AWS + R at [SatRday Johannesburg, South Africa the 6th of April](https://joburg2019.satrdays.org/). So if you are in the area, remember to buy some tickets and see how one can expand this even further!