|  |
| --- |
|  |
| DATA ANALYTICS USING R – OPIM 5503 |
| |  |  |  |  | | --- | --- | --- | --- | | Rohit Kapa |  |  | UCONN SCHOOL OF BUSINESS | |

Contents

[H2O Overview 2](#_Toc464030135)

[Project Overview 2](#_Toc464030136)

[Setup and installation 2](#_Toc464030137)

[Installing H2O 2](#_Toc464030138)

[Single machine initialization 3](#_Toc464030139)

[MultiNode initialization 3](#_Toc464030140)

[Importing and exporting data 3](#_Toc464030141)

[Import Data 3](#_Toc464030142)

[Export Data 4](#_Toc464030143)

[H20 to R Utilities 4](#_Toc464030144)

[Data wrangling and manipulation 5](#_Toc464030145)

[Basic data munging 5](#_Toc464030146)

[Outlier and Missing value treatment 6](#_Toc464030147)

[Building clean dataset 6](#_Toc464030148)

[Building train and test datasets 7](#_Toc464030149)

[Model building 8](#_Toc464030150)

[Measuring model accuracy 8](#_Toc464030151)

[GUI features 8](#_Toc464030152)

[R Script – Consumer Credit Risk Evaluation 8](#_Toc464030153)

[References 8](#_Toc464030154)

# H2O Overview

H2O in general is an open source, in-memory, distributed, fast, and scalable machine learning and predictive analytics platform that allows you to build machine learning models on large amounts of data aka Big Data and is used vastly in productionizing these models in real time cases. These algorithms work on distributed map/reduce framework and runs using multi-threading concept. H2O uses parallel computing in a distributed environment (like cluster) while accessing data and stores memory in columnar format. All the core code is written in Java and it uses a distributed Key/Value store to access and reference data, models, objects, etc., across all nodes and machines.

R can be connected to H2O using its REST API which provides an open powerful tool to analyze and process data. Advanced models like Deep Learning, Ensembles and GLRM can be made from R on large datasets and same can be productionized or used using API on real time basis.

H2O supports cross functional platforms and can be run using Python, Java, R interfaces and t also works in single machine environment to distributed environments like Hadoop, Spark etc.

# Project Overview

This paper aims to explore various features present in H2O which can help in solving a data science problem. Various features of H2O which address data pre-processing to model building and testing accuracies are covered in this project although due to the vast abilities of H2O we are restricting our effort focusing on most general concepts.

To make our explanation easier we have used [Lending Club](https://www.lendingclub.com/) openly available loan acceptance data as our primary choice of analysis. The data can be downloaded from this [link](https://www.lendingclub.com/info/download-data.action). We have used loan acceptance data from the year 2014 and 2015(can be found under ‘DOWNLOAD LOAN DATA’). Before using that kindly remove the first row and last 4 rows in the csv file, this is due to irregular values present in these files, else H2O will throw an error while importing all of them together. In case for ease we have created a google share drive where the ready to use data is available, kindly download from this [link](https://drive.google.com/open?id=0B7tnx-9WK8NaZTVoYm9xbU9SRkE). The data dictionary is available too in the lendingclub website and can be downloaded.

# Setup and installation

## Installing H2O

H2O can be installed like any other normal R package from cran repository and can be loaded into our working environment. The following two commands will help to get H2O ready.

install.packages("h2o")

library(h2o)

Although the basic installation is normal, in order to start the H2O cluster, we need to have a JDK (Java Development Kit) running in our system. In case, if there is no JDK available the H2O upon initialization will invoke an error to the console and will provide the path to download JDK.

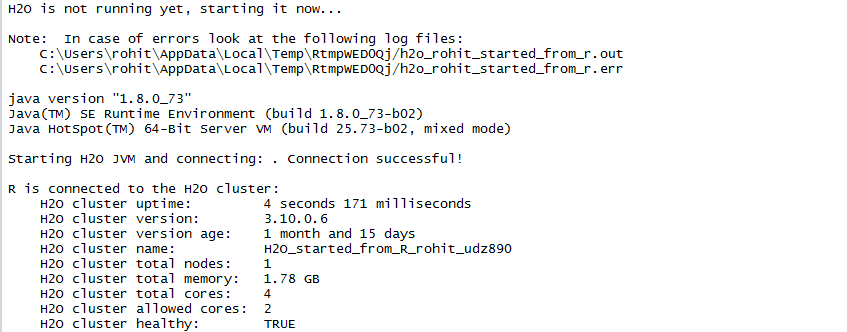
## Single machine initialization

H2O can be initialized in many ways depending on the infrastructure. To initialize/start H2O in a single personal machine aka local mode use the following command

h2o.init()

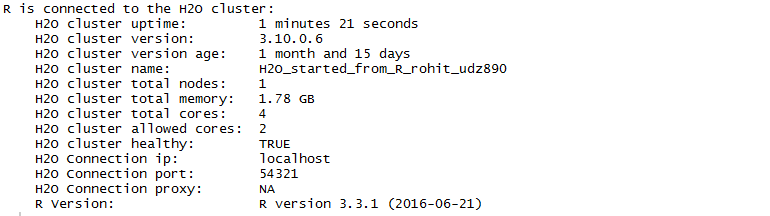
This initiates H2O on default mode where the localhost is the ip and 54321 is the port and one CPU thread for usage (RStudio restricts one core else it will use entire cores) with all the available memory. Although we would recommend going through following command where we are using only 2GB of memory allocation to H2O cluster. Even 2GB of memory is powerful enough to run high dimensional models i.e. high N\*D whereas RStudio generally crashes in solving sometimes which itself indicates the better memory management power of H2O.

h2o.init(ip = 'localhost', port = 54321, max\_mem\_size = '2g')



To obtain the information about the connection we can run the below command.

h2o.clusterInfo()



The h2o virtual machine can be shut down by using the below command, post running this H2O will closes itself and all the objects and memory variables are lost. In general, we can save these objects and use them again offline or restart of cluster.

h2o.shutdown()



## MultiNode initialization

As we know H2O can be connected to distributed system environment, we can connect this to systems like Hadoop, Spark etc using the following command. Here we are connecting to a cluster with 5 nodes and having total memory allocated to H2O as 16 and output directory as hdfsOutputDirName. Typically, the H2O system acts like a job and task tracker and execute in map/reduce framework.

hadoop jar h2odriver.jar -nodes 5 -mapperXmx 16g - output hdfsOutputDirName

# Importing and exporting data

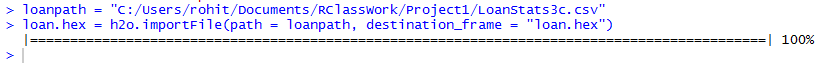
Data can be imported from local machine / Big data cluster into R using H2O. This import does not actually import’s the data, it actually just creates a pointer to this data using key/value mechanism and hence for every object/dataframe/variable we will end up having pointers to their indexes in R environment. It is called import as an environment type of file is created in R which establishes the connection with h2o JVM and actual file present on the disk. H2O supports following files as of now: CSV(delimited) files, ORC, SVMLite, ARFF, XLS, XLSX, Avro, Parquet

## Import Data

To import a single file (loan data file) from H2O's package:

loanpath = "C:/ your path/LoanStats3c.csv"

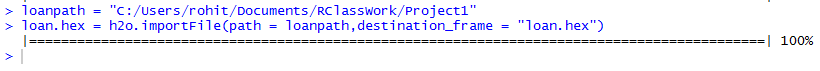
loan.hex = h2o.importFile(path = loanpath, destination\_frame = "loan.hex")



To import an entire folder of files as one data object:

loanpath = "C:/ your path/LoanData"

loan.hex = h2o.importFile(path = loanpath,destination\_frame = "loan.hex")



To import from HDFS and connect to H2O in R using the IP and port of an H2O instance running on your Hadoop cluster:

pathToData = "hdfs://your path in hdfs/datasets/loan.csv"

loan.hex = h2o.importFile(path = pathToData,destination\_frame = "loan.hex")

## Export Data

Similar to an import we can also export any data frame or object present in H2O to outside. To export H2O object data from R

loanpath = "C:/ your path /LoanData"

loan.hex = h2o.uploadFile(path = loanpath, destination\_frame = "loan.hex")

## H20 to R Utilities

In order to convert a H2O based data frame like we created above to R based data frame we can use below command

as.data.frame()

loan.R = as.data.frame(loan.hex)

In order to convert R based data frame to H2O based data frame

as.h2o()

loan.hex = as.h2o(loan.R,destination\_frame = 'loan.hex')

In order to rename the H2O data frame with other name, also ls() gives list of all H2O based objects

loan.hex <- h2o.assign(data = loan.hex, key = "new\_loan.hex")

h2o.ls()

To see the data structure of any object we can check through below code

str()

str(loan.hex)

To convert any data frame to a matrix format

loan.M = as.matrix(loan.hex)

# Data wrangling and manipulation

H2O works similar to R in data wrangling and also provides some more additional features. We will be exploring some of the most common used features in the below section. Please always note that these are applied in H2O and R acts as an interface only, making heavy big data manipulations easy.

## Basic data munging

To check class of our data frame loan.hex:

class(loan.hex)

To check the top 10 rows in our data frame

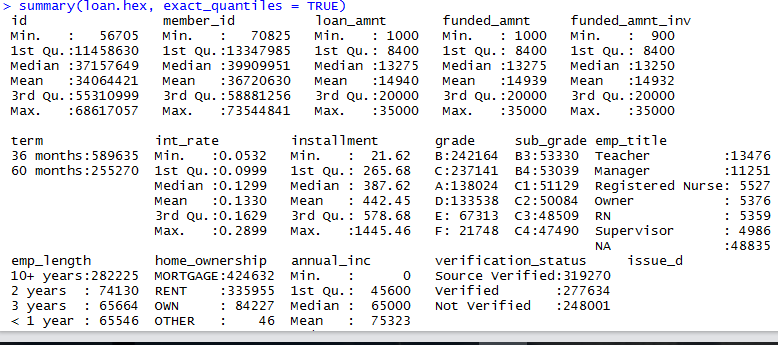
head(loan.hex,10)

To view a data frame in separate window

View(loan.hex)

To see the summary of each column in data frame, here exact\_quantiles is False by default making large data summarization easy.

summary(loan.hex, exact\_quantiles = TRUE)



To view all the columns name present in data frame

colnames(loan.hex)

names(loan.hex)



To check if any column in h2o frame contains continuous data

h2o.anyFactor(loan.hex)

To find the maximum minimum for a variable

min(loan.hex$funded\_amnt)

max(loan.hex$funded\_amnt)

To check the quantile based distribution at every 10 percentile

loan.qs <- quantile(loan.hex$funded\_amnt, probs = (1:10)/10)

loan.qs



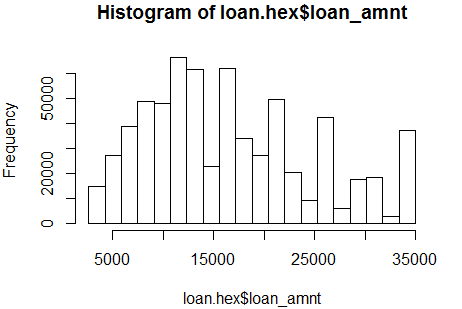
Binding any 2 data sets into one

bind.hex = h2o.cbind(loan.hex, loan.hex)

## Outlier and Missing value treatment

To view the histogram of some of the numeric variables

h2o.hist(loan.hex$loan\_amnt)

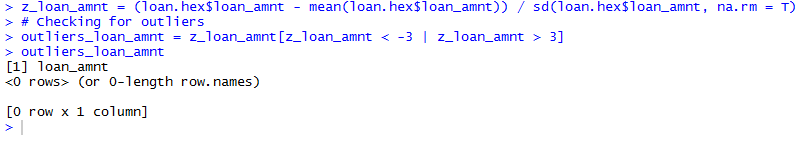


Data in loan\_amt seems to be normal with a few outliers, so now we will check if there are any outliers or not by standardizing the variable first and checking for counts at tails. From the output we can say there are no outliers.

z\_loan\_amnt = (loan.hex$loan\_amnt - mean(loan.hex$loan\_amnt)) / sd(loan.hex$loan\_amnt, na.rm = T)

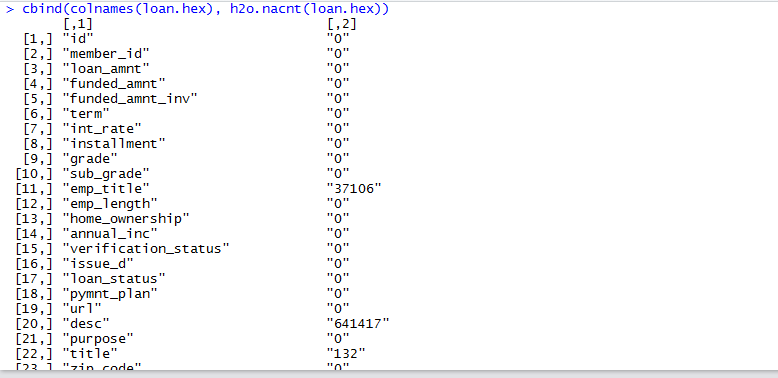
outliers\_loan\_amnt = z\_loan\_amnt[z\_loan\_amnt < -3 | z\_loan\_amnt > 3]

outliers\_loan\_amnt



In order to check the columns which have missing data

cbind(colnames(loan.hex), h2o.nacnt(loan.hex))

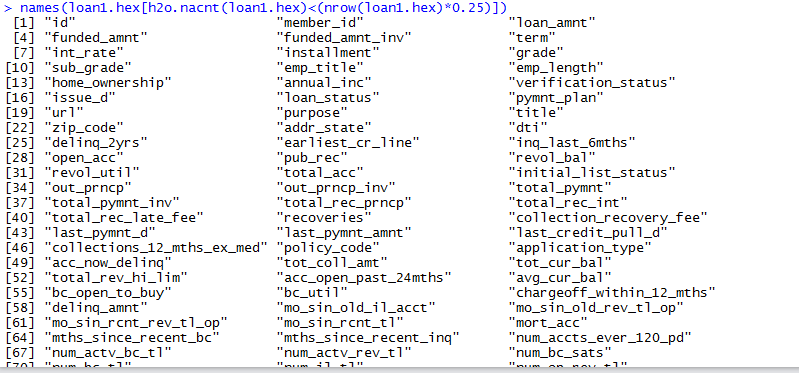


Now we will remove the columns which have a large number of missing values using nacnt(). The last line gives all the columns having greater than 25% of columns as missing.

h2o.nacnt(loan.hex)

names(loan.hex[h2o.nacnt(loan.hex)>(nrow(loan.hex)\*0.25)])

names(loan.hex[h2o.nacnt(loan.hex)<(nrow(loan.hex)\*0.25)])



By viewing the blank counts and also observing these variables it makes sense that these may not be useful in predicting whether the loan will be defaulted or not. So now we will remove the columns which have an excess of missing data.

loan.hex = loan.hex[h2o.nacnt(loan.hex)<(nrow(loan.hex)\*0.25)]

colnames(loan.hex)



## Building clean dataset

names(loan.hex)

After removing the missing data we can see our data has almost 85 variables. By looking at their description in data dictionary now we will remove the variables which might seem as unimportant. As per our analysis we find the following columns as important and hence we are taking all of them into one variable to make tidy clean meaningful dataset.

req\_cols = c(“id”, "addr\_state", "annual\_inc", "application\_type", "total\_pymnt", "avg\_cur\_bal", "collections\_12\_mths\_ex\_med", "delinq\_2yrs", "dti", "emp\_length", "emp\_title", "funded\_amnt", "home\_ownership", "inq\_last\_6mths", "installment", "int\_rate", "loan\_amnt", "mort\_acc", "open\_acc", "pub\_rec", "purpose", "revol\_bal", "revol\_util", "term", "tot\_hi\_cred\_lim", "total\_acc", “total\_bal\_ex\_mort", "verification\_status", "loan\_status")

Creating h2o frame having only the useful columns

loan2.hex = loan.hex[colnames(loan.hex) %in% req\_cols]

Our aim is to see what are the bad loans and what are the good loans and hence now we need to mark them a loan as good and bad. As per our data set these are the current statuses of loans:

summary(loan2.hex$loan\_status)



Now we can see that ‘Charged Off’ and ‘Default’ are the two status where we can define it as a bad loan and “Fully Paid” can be defined as a good loan. Apart from this all other conditions seems to be in running state, hence we will first eliminate these cases/records. After eliminating we will mark bad loan as per our earlier defined criteria as 1 and good loan as 0 making this problem a classification problem.

Our new target variable is bad\_loan in this case.

loan2.hex = loan2.hex[!(loan2.hex$loan\_status %in% c("Current", "In Grace Period", "Late (16-30 days)")), ]

loan2.hex = loan2.hex[!is.na(loan2.hex$id),]

loan2.hex$bad\_loan = loan2.hex$loan\_status %in% c("Charged Off", "Default)

loan2.hex$bad\_loan = as.factor(loan2.hex$bad\_loan)

We can see the number of loans completely paid or defaulted as

nrow(loan2.hex)



## Building train and test datasets

Now with our clean data we are building a train and test data sets. There are two method to achieve this, one either by creating a sample uniform distribution with total length of data frame and divide that as per 75:25 ratio. Now we will then split the data set in training and validation

s <- h2o.runif(loan2.hex)

loan.train <- loan2.hex[s <= 0.75,]

loan.train <- h2o.assign(loan.train, "loan.train")

loan.test <- loan2.hex[s > 0.25,]

loan.test <- h2o.assign(loan.test, "loan.test")

nrow(loan.train) + nrow(loan.test)

nrow(loan2.hex)

Another approach is to split the dataframe using splitFrame() function in H2O. This is quite powerful and works really well when working on huge amount of datasets.

loan.split <- h2o.splitFrame(data = loan2.hex ,ratios = 0.75)

loan.train <- loan.split[[1]]

loan.test <- loan.split[[2]]

# Model building

As we have got our test and train data sets ready for modelling now we will build various models to analyze the loan outcomes. Our target variable is “bad\_loan” and our explanatory variables are 27 around variables. We will take them into new vectors as shown below and will use this in our model building. The reason for taking them into vectors is cause it creates ease in building models later on.

target\_loan = 'bad\_loan'

explanatory\_loan = c("addr\_state", "annual\_inc", "application\_type", "avg\_cur\_bal", “collections\_12\_mths\_ex\_med", "delinq\_2yrs", "dti", "emp\_length", "emp\_title", "funded\_amnt", "home\_ownership", "inq\_last\_6mths", "installment", "int\_rate", "loan\_amnt", "mort\_acc", "open\_acc", "pub\_rec", "purpose", "revol\_bal", "revol\_util", "term", "tot\_hi\_cred\_lim", "total\_acc", "total\_bal\_ex\_mort", "verification\_status")

gbm\_model\_loan <- h2o.gbm(x = explanatory\_loan, y = target\_loan, training\_frame = loan.train, validation\_frame = loan.test, balance\_classes = T,

learn\_rate = 0.05, score\_each\_iteration = T, ntrees = 100, max\_depth = i)

h2o.auc(gbm\_model\_loan)

h2o.auc(gbm\_model\_loan,valid = T)

gbm\_score <- plot\_scoring(model = gbm\_model\_loan)

h2o.varimp(gbm\_model\_loan)

gbm\_model\_loan@model$scoring\_history

gbm\_model\_loan@model$training\_metrics

glm\_model\_loan <- h2o.glm(x = explanatory\_loan, y = target\_loan, training\_frame = loan.train, family = 'binomial',

nfolds=10,alpha = 0.5)

glm\_model\_loan@model$cross\_validation\_metrics

h2o.predict(object = glm\_model\_loan,newdata = loan.test$bad\_loan)

deeplearn\_model\_loan <- h2o.deeplearning(x = explanatory\_loan, y = target\_loan, training\_frame = loan.train,validation\_frame = loan.test)

# Measuring model accuracy

# GUI features

# R Script – Consumer Credit Risk Evaluation

# References

1. <http://docs.h2o.ai/h2o/latest-stable/h2o-docs/booklets/RBooklet.pdf>
2. <http://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html>
3. <https://cran.r-project.org/web/packages/h2o/h2o.pdf>
4. <http://www.h2o.ai>
5. <https://www.lendingclub.com/info/download-data.action>