UCONN

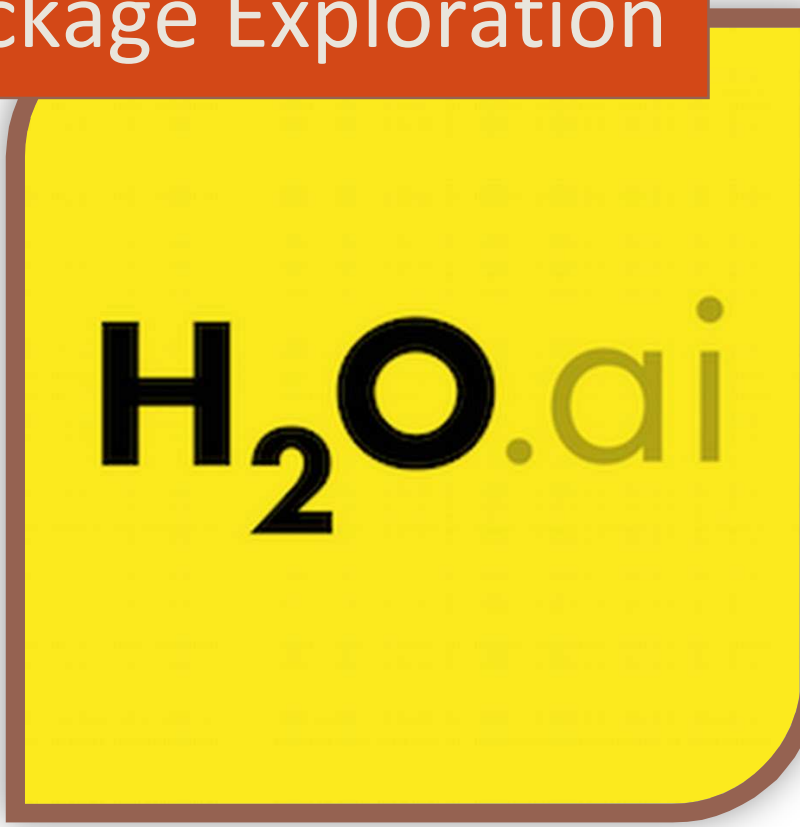
Data Analytics using R

OPIM 5503

Team – smaRt

Project – 2

H2O Package Exploration



11/28/2016

Shaik Shavali

Deepak Sisodiya

Harsha Chandar

Pranav Sachdev

Mir Akram Ali Yadullahi

Contents

H2O Overview

What is H2O?

H2O[i] is:

- open source
- in-memory
- distributed
- fast
- scalable
- machine learning and predictive analytics platform

H2O allows us to build machine learning models on big data in an enterprise / academic environment.

Architecture of H2O

Code of H2O is written in Java. Inside H2O, a Distributed Key/Value store is used to access and reference data, models, objects, etc., across all nodes and machines. The algorithms are implemented on top of H2O's distributed Map/Reduce framework and utilize the Java Fork/Join framework for multi-threading.

The data is read in parallel and is distributed across the cluster and stored in memory in a columnar format in a compressed way. H2O's data parser has built-in intelligence to guess the schema of the incoming dataset and supports data ingest from multiple sources in various formats.

H2O's REST API allows access to all the capabilities of H2O from an external program or script via JSON over HTTP. The Rest API is used by H2O's web interface (Flow UI), R binding (H2O-R), and Python binding (H2O-Python).

Advantages of H2O

With H2O, model-deployment for the various cutting edge Supervised and Unsupervised algorithms like Deep Learning, Tree Ensembles, and GLRM can be performed with ease, which makes H2O a highly sought after API for big data analytics. Computations using H2O has the following advantages:

- Speed
- Quality
- Ease-of-use

Supported File Types

Following file formats are supported with H2O:

- CSV
- OCR
- SMVLite
- ARFF
- XLS
- XLSX
- Avro
- Parquet

Supported Programming Language Environments

H2O can be accessed using:

- Java
- Scala
- Python
- R

Project Overview

This paper aims to explore various features present in H2O which we are trying to demonstrate through 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 demonstrate the features of H2O package, we have used *Black Friday Dataset*[ii], which is available at *Analytics Vidhya website* (www.analyticsvidhya.com/contest/black-friday/).

The problem statement is about a retail company "ABC Private Limited" who wants to understand the customer purchase behavior (purchase amount) against various products of different categories. The data set contains customer demographics (age, gender, marital status, city_type, stay_in_current_city), product details (product_id and product category) and Total purchase_amount from last month.

It's required to build a model to predict the purchase amount of customer against various products which will help them create personalized offer for customers against different products.

Data Dictionary

Variable	Definition
User_ID	User ID
Product_ID	Product ID
Gender	Sex of User

	Age in bins
Occupation	**Occupation (Masked)**
City_Category	**Category of the City (A,B,C)**
Stay_In_Current_City_Years	**Number of years stay in current city**
Marital_Status	**Marital Status**
Product_Category_1	**Product Category (Masked)**
Product_Category_2	**Product may belongs to other category also (Masked)**
Product_Category_3	**Product may belongs to other category also (Masked)**
Purchase	**Purchase Amount (Target Variable)**

H2O Installation and Setup

1. **Download the required package:**

install.packages("h2o")

2. **Initialize the h2o session (in our case, we have initialized on the localhost, on all the CPU cores):**

h2o.init(ip = "localhost", port = 54321, startH2O = TRUE,

nthreads = -1, max_mem_size = NULL, min_mem_size = NULL)

ip - IP address of the server where H2O is running.

port - Port number of the H2O server.

startH2O - (Optional) A logical value indicating whether to try to start H2O from R if no connection with H2O is detected.

nthreads - (Optional) Number of threads in the thread pool, or, number of CPUs used. -2 means use the CRAN default of 2 CPUs. -1 means use all CPUs on the host. A positive integer specifies the number of CPUs directly.

max_mem_size - (Optional) The maximum size, in bytes, of the memory allocation pool to H2O.

min_mem_size - (Optional) The minimum size, in bytes, of the memory allocation pool to H2O.

```
> library(h2o)
> h2o.init(ip = "localhost", port = 54321, startH2O = TRUE,
+ nthreads = -1, max_mem_size = NULL, min_mem_size = NULL)

H2O is not running yet, starting it now...

Note:  In case of errors look at the following log files:
    C:\                                                    _started_from_r.out
    C:\                                                    _started_from_r.err

java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)

Starting H2O JVM and connecting: . Connection successful!

R is connected to the H2O cluster:
    H2O cluster uptime:         3 seconds 185 milliseconds
    H2O cluster version:
    H2O cluster version age:    1 month and 17 days
    H2O cluster name:           H2O_started_from_R_prana_icb728
    H2O cluster total nodes:    1
    H2O cluster total memory:   1.75 GB
    H2O cluster total cores:    4
    H2O cluster allowed cores:  4
    H2O cluster healthy:        TRUE
    H2O Connection ip:          localhost
    H2O Connection port:        54321
    H2O Connection proxy:       NA
    R Version:                  R version 3.3.1 (2016-06-21)
```

3. Getting the cluster info:

h2o.clusterInfo()

```
> h2o.clusterInfo()
R is connected to the H2O cluster:
    H2O cluster uptime:         9 minutes 15 seconds
    H2O cluster version:
    H2O cluster version age:    1 month and 17 days
    H2O cluster name:           H2O_started_from_R_prana_icb728
    H2O cluster total nodes:    1
    H2O cluster total memory:   1.75 GB
    H2O cluster total cores:    4
    H2O cluster allowed cores:  4
    H2O cluster healthy:        TRUE
    H2O Connection ip:          localhost
    H2O Connection port:        54321
    H2O Connection proxy:       NA
    R Version:                  R version 3.3.1 (2016-06-21)
```

Data Preparation

Importing the Data

H2o supports multiple data import options. The command for this is **h2o.importFile.** Previously, there were dedicated command statements depending upon the source such as h2o.importHDFS,h2o.importURL which are now being deprecated. The functionality of this could be handled by the h2o.importFile.

```
#Import data from file
file_path <- "C:/Users/chars/Desktop/R/Project/Project2/Data/train.csv"
file_data<- h2o.importFile(path=file_path)
str(file_data)

#Import folder contents as a single object
folder_path = "C:/Users/chars/Desktop/R/Project/Project2/Folder_Import"
folder_data<- h2o.importFile(path=folder_path)
str(folder_data)

#Import data from url
url<-"https://perso.telecom-paristech.fr/eagan/class/igr204/data/Camera.csv"
url_data<- h2o.importFile(path=url,sep=";",header = T)
str(url_data)
head(url_data)
```

Our data is a local file. We can import this data using the h2o.importFile statement. This statement pulls the information from the location we have specified as a read operation.

The data obtained can be viewed below, using the str(data) command:

```
- attr(*, "data")='data.frame':        10 obs. of  12 variables:
 ..$ User_ID                   : num  1e+06 1e+06 1e+06 1e+06 1e+06 ...
 ..$ Product_ID                : Factor w/ 3631 levels "P00000142","P00000242",..: 673 2377 8
321 3605 2632
 ..$ Gender                    : Factor w/ 2 levels "F","M": 1 1 1 1 2 2 2 2 2 2
 ..$ Age                       : Factor w/ 7 levels "0-17","18-25",..: 1 1 1 1 7 3 5 5 5 3
 ..$ Occupation                : num  10 10 10 10 16 15 7 7 7 20
 ..$ City_Category             : Factor w/ 3 levels "A","B","C": 1 1 1 1 3 1 2 2 2 1
 ..$ Stay_In_Current_City_Years: num  2 2 2 2 NaN 3 2 2 2 1
 ..$ Marital_Status            : num  0 0 0 0 0 0 1 1 1 1
 ..$ Product_Category_1        : num  3 1 12 12 8 1 1 1 1 8
 ..$ Product_Category_2        : num  NaN 6 NaN 14 NaN 2 8 15 16 NaN
 ..$ Product_Category_3        : num  NaN 14 NaN NaN NaN NaN 17 NaN NaN NaN
 ..$ Purchase                  : num  8370 15200 1422 1057 7969 ...
```

Here the data types are determined by h2o by reading a few lines of the data. But, as we can see the data types are incorrectly determined.

For e.g. Stay_in_current_city_years has four values – 1,2,3 and 4+. As this field is determined to be a numerical field, all the values with 4+ are converted into NaNs.

To rectify this, we need to specify the column types in the import statement.

```
#Read train data and provide correct data types
train_data<-h2o.importFile( path = "C:/Users/chars/Desktop/R/Project/Project2/Data/train.csv"
                           ,col.types = c("Factor","Factor","Factor","Factor","Factor","Factor","Factor"
                                          ,"Factor","Factor","Factor","Factor","Int")
                            )

str(train_data)
```

```
  ..$ User_ID                   : Factor w/ 5891 levels "1000001","1000002",..: 1 1 1 1 2 3 4 4 4 5
  ..$ Product_ID                : Factor w/ 3631 levels "P00000142","P00000242",..: 673 2377 853 829 2735 1
321 3605 2632
  ..$ Gender                    : Factor w/ 2 levels "F","M": 1 1 1 1 2 2 2 2 2 2
  ..$ Age                       : Factor w/ 7 levels "0-17","18-25",..: 1 1 1 1 7 3 5 5 5 3
  ..$ Occupation                : Factor w/ 21 levels "0","1","10","11",..: 3 3 3 3 9 8 19 19 19 14
  ..$ City_Category             : Factor w/ 3 levels "A","B","C": 1 1 1 1 3 1 2 2 2 1
  ..$ Stay_In_Current_City_Years: Factor w/ 5 levels "0","1","2","3",..: 3 3 3 3 5 4 3 3 3 2
  ..$ Marital_Status            : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 2 2 2 2
  ..$ Product_Category_1        : Factor w/ 20 levels "1","10","11",..: 14 1 4 4 19 1 1 1 1 19
  ..$ Product_Category_2        : Factor w/ 17 levels "10","11","12",..: NA 14 NA 5 NA 10 16 6 7 NA
  ..$ Product_Category_3        : Factor w/ 15 levels "10","11","12",..: NA 5 NA NA NA NA 8 NA NA NA
  ..$ Purchase                  : num  8370 15200 1422 1057 7969 ...
> h2o.levels(train_data$Stay_In_Current_City_Years)
[1] "0"  "1"  "2"  "3"  "4+"
```

Once our data is loaded, we can perform several data manipulation operations using h2o.

Data Transformation

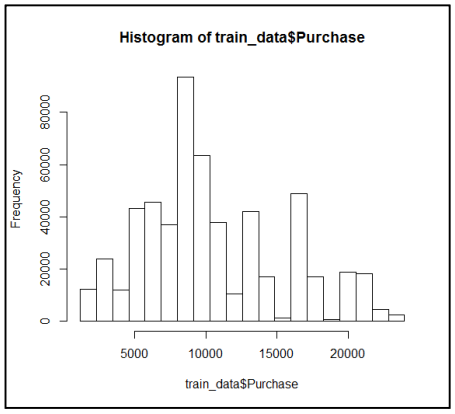
In order to get a sense of the data, we use the following operations

#To check if the data contains any factor columns

h2o.anyFactor(train_data)

#Plotting purchase value

h2o.hist(train_data$Purchase)



We found that the data is almost normally distributed.

1. **To check the range of values, we can use the quantile, min and max functions:**

#Getting quantile values:

quantile(train_data$Purchase)

```
> quantile(train_data$Purchase)
 0.1%    1%   10%   25% 33.3%   50% 66.7%   75%   90%   99% 99.9%
   26   587  3540  5823  6914  8047 10070 12054 16317 20665 23729
>
```

#Getting minimum and maximum values:

max(train_data$Purchase)

23961

min(train_data$Purchase)

12

#To sort the data in ascending order of purchase

h2o.arrange(train_data,Purchase)

2. When using ordinal variables as factors, we might miss out on the information about the order. Therefore, we converted the Age and Stay_in_current_city_years into numerical variables.

```
#Numeric conversion of age variable:

train_data$Age_numeric<-ifelse(train_data$Age=="0-17",0,ifelse(train_data$Age=="18-25",1,ifelse(train_data$Age=="26-35",2,ifelse(train_data$Age=="36-45",3,ifelse(train_data$Age=="46-50",4,ifelse(train_data$Age=="51-55",5,6))))))

#Numeric conversion of Stay in city years variable:

train_data$Stay_numeric<-ifelse(train_data$Stay_In_Current_City_Years=="4+",4,train_data$Stay_In_Current_City_Years)
```

3. Also, to improve the ease of modeling, we converted the Gender variable into a numeric value containing 0s and 1s.

```
#Numeric conversion of Gender variable:

train_data$Gender_numeric<-ifelse(train_data$Gender=="F",0,1)
```

4. We also derived a few variables to aid in determining the purchase amount based on the RFM approach for each categorical variable.

The RFM approach deals with Recency, Frequency and Monetary aspects of purchase. Recency deals with the timing of the purchase (How recent was the customer's last purchase) and is the same for all customers (1 month) so we did not include this variable.

Frequency deals with how often the customer makes the purchase. This was taken as the count of rows of number of times the user had a record.

Monetary deals with the value of money that transaction was worth. This was taken as the mean of purchase value for that record.

We also obtained the frequency and average purchase amount for all categorical variables – This would indicate whether certain groups either appear more in the data or spend more per transaction.

Once these derived values were obtained, they were merged with the training and test data.

This was done using **h20.group_by** and **h2o.merge.**

```
#Get value fields for each categorical variable - mean purchase value and frequency

#Obtaining average purchase amount and frequency for each marital status

ms<-h2o.group_by(train_data,by="Marital_Status",mean("Purchase"),nrow("Purchase"))

#Assigning Column names

colnames(ms)<-c("Marital_Status","Mean_Purchase_MS","Count_MS")
```

```
#Merging this information with the train and test data

train_data<-h2o.merge(train_data,ms,by = "Marital_Status",all.x = T)

test_data<-h2o.merge(test_data,ms,by = "Marital_Status",all.x = T)
```

5. Interactions are important to consider while building models. For example, customers of a particular age group in a particular city might act differently from other customers of that age group. H2o has function to create columns to take the interaction effect into consideration. This can be done through the h20.interaction function.

```
#Creating Interaction variables

train_sample<-train_data[c(3,4,6)]

head(train_sample)

int<-h2o.interaction(train_sample,factors = c("Gender","Age","City_Category")

                     ,pairwise = F,max_factors = 20,min_occurrence = 1)

train_int<-as.data.frame(h2o.cbind(train_sample,int))
```

Note: This can be done only for factor variables.

Max_factors denote the maximum unique values for each column. If a particular interaction column has more than the max_factors, the additional factors are grouped into a column called “other”.

If pairwise is true, the interactions consider two factors at a time for all factors specified. If it is false, higher order interactions are considered.

Also, there are some functions that are not supported by h2o – creating dummy variables are an example. In these cases, it is advisable to use h2o in conjuction with R.

We created one-hot encoded variables to represent categorical variables. One hot encoding is converting the categorical variables into numerical with the help of dummy variables. For example, if a categorical variable has 3 levels, 3 binary columns for each level are created. To do this, we first need the data in R. As this data is big, we are importing the raw data again using data.table's fread function.

Once we have that, we used the dummy library to create the dummy variables.

Also, the impute function available in h2o (**h2o.impute**) doesn't allow user specified values for imputation. So we used a mutate from dplyr to impute the values as well as create a flag denoting the imputation for the Product_Category_2 and Product_Category_3 variables. Once this is done, the data frames are uploaded into h2o using the **as.h2o** function.

```
library(dummies)

train_df<-dummy.data.frame(train_df,names =
c("Gender","Age","Occupation","City_Category","Stay_In_Current_City_Years","Marital_Status"))

test_df<-dummy.data.frame(test_df,names =
c("Gender","Age","Occupation","City_Category","Stay_In_Current_City_Years","Marital_Status"))
```

```
train_df<-mutate(train_df,missingProd2=ifelse(is.na(Product_Category_2),1,0),

            missingProd3=ifelse(is.na(Product_Category_3),1,0),

            numProds=ifelse(missingProd2==1&missingProd3==1,1

                        ,ifelse(missingProd2==0&missingProd3==0,3,2)))%>%

  mutate(Product_Category_2_Imp=ifelse(is.na(Product_Category_2),9999,Product_Category_2),

       Product_Category_3_Imp=ifelse(is.na(Product_Category_3),9999,Product_Category_3)

  )%>%select(-User_ID,-Product_ID,-Product_Category_1,-Product_Category_2,-Product_Category_3,-
Purchase)


test_df<-mutate(test_df,missingProd2=ifelse(is.na(Product_Category_2),1,0),

            missingProd3=ifelse(is.na(Product_Category_3),1,0),

            numProds=ifelse(missingProd2==1&missingProd3==1,1

                        ,ifelse(missingProd2==0&missingProd3==0,3,2)))%>%
```

```
mutate(Product_Category_2_Imp=ifelse(is.na(Product_Category_2),9999,Product_Category_2),
Product_Category_3_Imp=ifelse(is.na(Product_Category_3),9999,Product_Category_3)
)%>%select(-User_ID,-Product_ID,-Product_Category_1,-Product_Category_2,-Product_Category_3)
```

```
#Uploading the data frames to h2o cloud as a h20 frame
train_df<-as.h2o(train_df)
test_df<-as.h2o(test_df)
```

6. **These frames are then merged with the frames available already in h2o using cbind**

```
#Combing the two h2o frames
train_all<-h2o.cbind(train_data,train_df)
test_all<-h2o.cbind(test_data,test_df)
```

7. **This data can then be split into training and validation sets using the h2o.splitFrame function.**

```
#Splitting train data for modeling:
model_split <- h2o.splitFrame(data = train_all, ratios = 0.75)
```

Modeling

Random Forest

Random Forests are one of the most widely used and one of the most popular ensemble learning methods. All ensemble learning methods are based on the premise of combining a group of weak learners (in case of the random forests, these are individual decision trees) into a 'Strong' learner, that has a very low bias (extracts complete information from the features) and a low variance (produces estimates that do not overfit the training data). In the case of Random Forests, the algorithm builds a lot of 'bushy' trees and then averages them in order to reduce the variance. In order to reduce the correlation between the trees, the algorithm picks a subset (m) of the attributes randomly and uses these randomly chosen attributes at each of the decision tree nodes. As such, in the process, reduces a lot of noise that could have been present due to the presence of spurious relationships between the response and the predictors in the training sample data.

Algorithm for Random Forests

Key Notations

N = Number of samples in the training dataset

P = Number of input parameters

m = Number of variables randomly selected by the algorithm to be used at each node of the decision tree

ntrees = Number of decision trees constructed

Algorithm Steps:

1. A Bootstrap sample (sample with replacements) of size N is chosen from the input training dataset
2. A decision tree is constructed on this dataset
3. A subset (m) of the input variables is chosen at random, which are used at every node of the decision trees
4. Steps 1-3 are repeated a number of times (equal to ntrees)
5. An Ensemble model of all the decision trees is considered as the final random forest model

The tuning parameter for random forests is m, which can be changed in order to give the optimal performance

A Random Forest Implementation in H2O package

The speed and optimized performance of the H2O package makes it very simple and intuitive to build customized random forests models in H2O. Apart from the great speed benefits while processing really big datasets, H2O package also provides an exemplary feature of grid search and model selection. H2O provides functions to perform dynamic parameter tuning, estimating the model performance based on the 'out-of-bag' training samples (an inbuilt cross validation method in random forests) and also calculate the model performance on a new test dataset. In this report, we will construct a simple random forest in H2O with 1000 trees and m=5. We will then perform a grid search to tune the hyper-parameter m in order to obtain the best performing model (the model with the smallest 'Root Mean Squared Error' value)

A simple Random Forest Implementation in H2O

Dependent Variable – Purchase Amount

Independent Variables

```
> names(train[x])
 [1] "Age"                          "City_Category"                "Occupation"
 [4] "Gender"                       "Marital_Status"               "Product_Category_1"
 [7] "Product_Category_2"           "Product_Category_3"           "Mean_Purchase_MS"
[10] "Count_MS"                     "Mean_Purchase_Gen"            "Count_Gen"
[13] "Mean_Purchase_Occ"            "Count_Occ"                    "Mean_Purchase_City"
[16] "Count_City"                   "Mean_Purchase_Age"            "Count_Age"
[19] "Mean_Purchase_User"           "Count_User"                   "Mean_Purchase_Product"
[22] "Count_Product"                "Gender_Age"                   "Gender_Marital_Status"
[25] "Gender_Occupation"            "Gender_City_Category"         "Age_Marital_Status"
[28] "Age_Occupation"               "Age_City_Category"            "Marital_Status_Occupation"
[31] "Marital_Status_City_Category" "Occupation_City_Category"
```

Training Sample Size – 412,551

Test Sample Size – 137,517

m = 5

ntrees = 1000

```
#Define a simple random forest algorithm with the following metrics
#mtries = 5; ntrees = 1000
rand.for.1k = h2o.randomForest(x = x,y = y,training_frame = train,
                               model_id = 'rand.for.1k',seed = 14,mtries = 5,ntrees = 1000)
```

Model Performance on the training sample data

```
#Check the model performance on the training data
h2o.performance(model = rand.for.1k)
```

```
> stime_1k
[1] "2016-11-26 21:48:17 EST"
> etime_1k
[1] "2016-11-26 22:57:42 EST"
> h2o.performance(model = rand.for.1k)
H2ORegressionMetrics: drf
** Reported on training data. **
** Metrics reported on Out-Of-Bag training samples **

MSE:  6184579
RMSE:  2486.881
MAE:  1847.302
RMSLE:  0.3398661
Mean Residual Deviance :  6184579
```

The algorithm took 1 hour 10 minutes to run. The RMSE on the out-of-bag training sample was observed to be 2,486

We estimated the model performance by fitting the model on the test dataset as follows

```
#Evaluate the model performance on test/validation data
rand.for.test.perf = h2o.performance(model = rand.for.1k,newdata = test)
rand.for.test.perf
```

```
> rand.for.test.perf = h2o.performance(model = rand.for.1k,newdata = test)
> rand.for.test.perf
H2ORegressionMetrics: drf

MSE:  6243007
RMSE:  2498.601
MAE:  1860.163
RMSLE:  0.3419711
Mean Residual Deviance :  6243007
```

The RMSE on the test sample was noted to be 2,498

H2O Grid Search and Model Selection

One of the most amazing features of H2O is the availability of the h2o.grid function. This function performs an automatic hyper-parameter tuning and selects the best model from a range of executed models. The grid search and model selection is performed in the following steps

1. Define a list of all the hyperparameters to be tuned for the model
2. Using the h2o.grid function, construct the random forest algorithm with the hyperparameter list as one of the inputs
3. Once the function runs, sort the models based on the lowest value of the model selection criteria (in this case, the RMSE)
4. Select the best performing model

These steps are coded in R as below

```
#Grid Search and Model Selection

#1. Defining the tuning parameter - mtries
randfor_pars = list(mtries=seq(2,12,1),ntrees=c(400),max_depth=c(10))
#Capture the start time
time_start_gr = Sys.time()
#2. Using the h2o.grid function, specify the hyper-parameters to be tuned
randfor_grid1 = h2o.grid(algorithm = "randomForest",grid_id = "randfor_grid1",x=x,y=y,training_frame=train,
                         validation_frame=test,seed=84,hyper_params = randfor_pars)
#Capture the end time
time_end_gr = Sys.time()
#3. Sort the resulting models based on the lowest value of RMSE
randfor_perf_grid = h2o.getGrid(grid_id = "randfor_grid1",sort_by = 'RMSE',decreasing = FALSE)
#Print all the models developed
print(randfor_perf_grid)
#4. Select the best model from the various models developed by the grid search
randfor_best_id = randfor_perf_grid@model_ids[[1]]
h2O.randfor_best = h2o.getModel(randfor_best_id)
h2o.performance(h2O.randfor_best)
#Obtain the variable importance plot
h2o.varimp_plot(h2O.randfor_best)
```

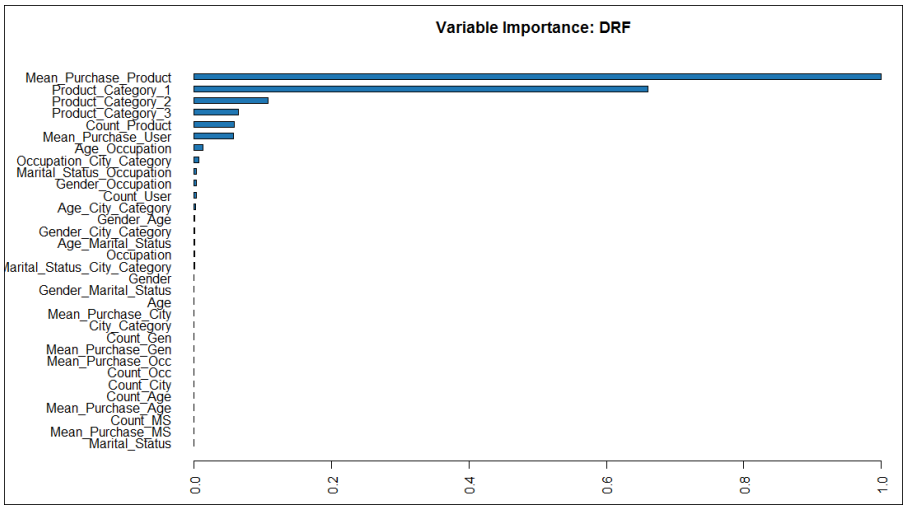
Results of Grid Search and Model Selection

The following were the results of the models developed as a part of the automatic hyper parameter tuning

```
H2O Grid Details
================

Grid ID: randfor_grid1
Used hyper parameters:
  -  max_depth
  -  mtries
  -  ntrees
Number of models: 7
Number of failed models: 0

Hyper-Parameter Search Summary: ordered by increasing RMSE
  max_depth mtries ntrees             model_ids               rmse
1        10      8    400 randfor_grid1_model_6  2542.069910135469
2        10      7    400 randfor_grid1_model_5  2554.445300827827
3        10      6    400 randfor_grid1_model_4  2576.413302908297
4        10      5    400 randfor_grid1_model_3 2612.0497646763365
5        10      4    400 randfor_grid1_model_2  2674.562622115729
6        10      3    400 randfor_grid1_model_1  2792.639193733161
7        10      2    400 randfor_grid1_model_0 3109.9805154640308
> time_start_gr
[1] "2016-11-27 10:38:52 EST"
> time_end_gr
[1] "2016-11-27 11:20:03 EST"
```

From the above developed models, we choose the best performing model with an RMSE of 2,542 which was obtained with only 400 trees and the number of variables as 8. The overall grid selection developed 7 different models and was executed within an expedited time span of 42 minutes for 7 models.

The variable importance plot for the best random forest model is shown below



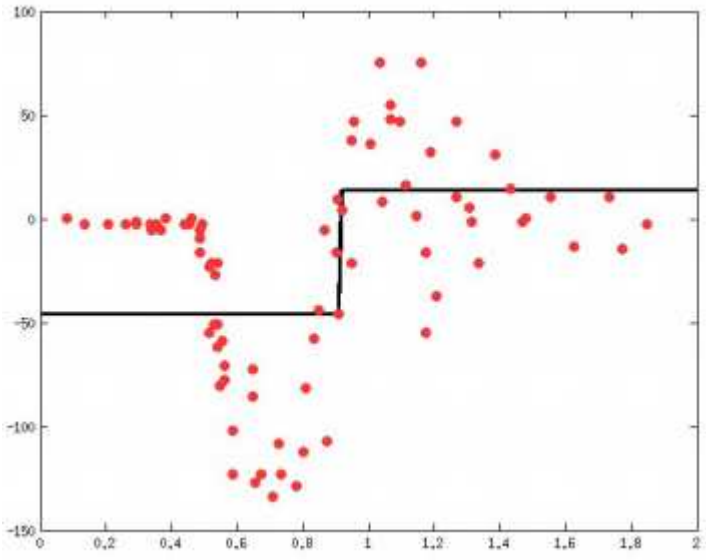
Gradient Boosting Model (GBM)

H2O's Gradient Boosting Machine (GBM) offers a Stochastic GBM. It has the capability of increasing the performance compared to the original GBM implementation. Gradient Boosting is an ensemble learning technique for modeling.

Boosting is designed to reduce bias and variance. In boosting algorithm, weak learners are iteratively learned and added to a final strong predictor. After a weak learner is added, errors are calculated and future weak learner is modeled based on the errors. This process is repeated again and again. Thus with each iteration our model's accuracy increases and also the model becomes complex. The boosting methods are not very robust to noisy data and outliers.

Working of GBM Algorithm

1. A weak learner is trained on the input data (weak learner in this example is a tree with only one split)



For simplicity, let's assume we have a single continuous input variable(x) and one continuous output variable (y). In the above figure, a sample distribution of y values for different values of x is shown.

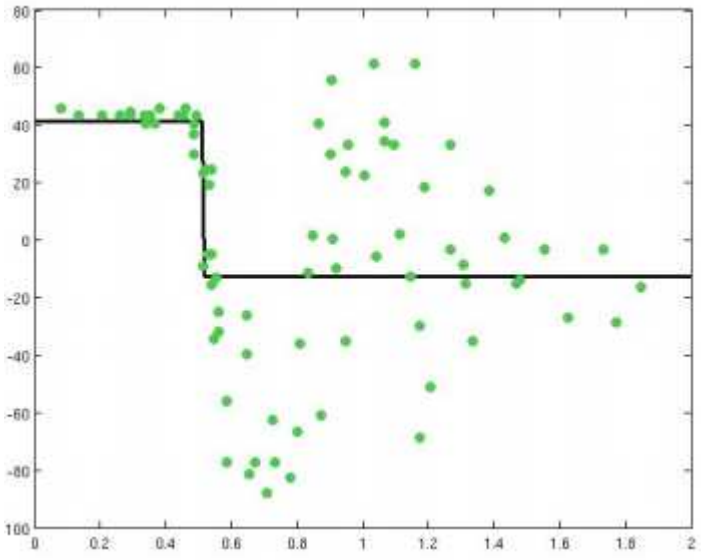
A weak learner (say weak_learner_1) in the form of a decision tree with one node is created.

If x > 0.87 then y = 10
if x < 0.87 then y = -40

2. Compute the residuals based on the predictions made by weak_learner_1.

3. Train a weak learner to predict the residuals. In other words, the variable to be predicted now is the residual (say Residual1) and input is x.
   Residual1 = y – predictions by weak_learner_1
   Thus, again a weak learner (say weak_learner_1) is trained on the residuals of the previous model.

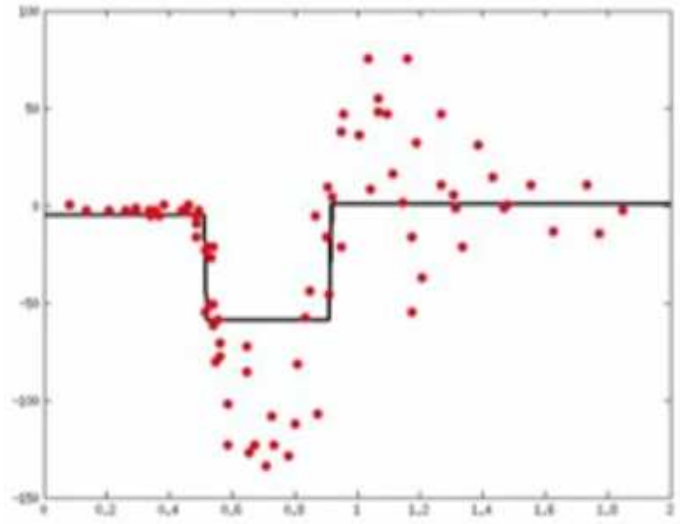
Plot of Residual1 vs x values is shown in the figure below and black line represents a weak learner (named weak_learner_2) in the form of a single node decision tree on this residual data.



If x > 0.5 then Residual1 = -12

if x < 0.5 then Residual1 = 41

4. Combine weak_learner_1 and weak_learner_2 and make predictions on the original data
   For Combined decision tree model (say combined_learner_1), the thresholds will be:
   x = 0.87 and x=0.5
   A new model slightly more complex than the single layer decision tree is created. The combined model with two thresholds is shown in the figure below.

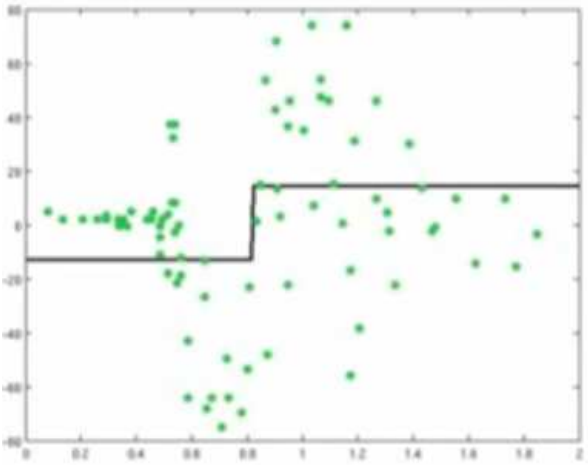


5. Now compute the residuals based on the predictions made by the combined_learner_1. (Similar to step 2 above)
6. Train a weak learner to predict the residuals (say Residual2) of combined_learner_1. In other words, the variable to be predicted now is the Residual2 and input is x.

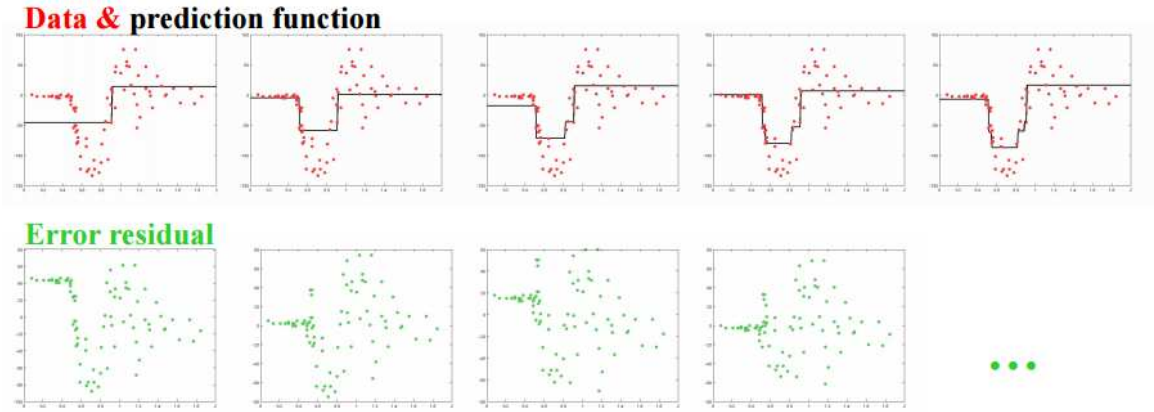
   Residual2 = y – predictions by combined_learner_1

Thus, a weak learner (say weak_learner_3) is trained on the residuals of the previous model combined_learner_1.

Plot of Residual2 vs x values is shown in the figure below and black line represents a weak learner (named weak_learner_3) in the form of a single node decision tree on the residual data.



7. Now combine combined_learner_1 and weak_learner_3 to create a new slightly more complex model.
8. Repeat these steps until you reach the desired accuracy level.



So, we are learning a sequence of models. As we are combining more and more weak learners, the accuracy of model is increasing on training data and also model's complexity is increasing.

GBM implementation in H20 package

After loading the H20 library, h20.gbm() function is required to be used in order to build GBM model. There are quite a lot parameters which we can specify for building and tuning a GBM model.

Case1: Basic GBM model

Y - Dependent variable
x - All independent variable names as a vector
training_frame - training data on which model is to be trained
model_id – specify a unique model id
seed – specify a seed for reproducibility

```
gbm.model.1 <- h2o.gbm(y=dep.var, x=indep.var, training_frame = train,
+                       model_id = "gbm.model.1",
+                       seed = 1000)
```

```
gbm.performance.1 = h2o.performance(model = gbm.model.1,newdata =
valid)
print(gbm.performance.1)
Output :
> gbm.performance.1
H2ORegressionMetrics: gbm

MSE:  7688605
RMSE:  2772.833
MAE:  2072.584
RMSLE:  0.3838856
Mean Residual Deviance :  7688605
```

Optional parameters:

ntrees – Specify number of Trees (default value is 50)

In H2O package GBM algorithm has an early stopping option available, though it is not enabled by default. There are several parameters that should be used to control early stopping. The three that are common are: `stopping_rounds`, `stopping_metric` and `stopping_tolerance`.

The stopping_metric is the metric to measure performance (MSE in our case). This metric would be calculated on the validation data set. The `score_tree_interval` specifies the number of trees after which model should be scored. After how many scoring intervals model training should stop is specified by ` stopping_rounds `. `stopping_tolerance` specifies the expected incremental improvement to continue training.

Example:

```
gbm.model.3 <- h2o.gbm(y=dep.var, x=indep.var, training_frame =
train,
                       model_id = "gbm.model.3",
                       validation_frame = valid,
                       ntrees = 500,
                       score_tree_interval = 3,
                       stopping_rounds = 4,
                       stopping_metric = "MSE",
                       stopping_tolerance = 0.5,
                       seed = 1000)
```

Case 2: GBM with Cartesian Grid Search

Instead of running multiple models manually one by one, we can train a bunch of models in one go by using the Cartesian grid search functionality of h2o package. All models specified in the grid would be checked to identify the best set of parameters which is giving the best performance.

GBM hyperparameters are specified as shown below. For each parameter combination, a model will be trained on training data and evaluated on validation data set. For the below code, a total of 36 models (2*3*2*3) will be trained and evaluated.

```
gbm_params.1 <- list(learn_rate = c(0.01, 0.1),
+                     max_depth = c(3, 6, 9),
+                     sample_rate = c(0.7, 1.0),
+                     col_sample_rate = c(0.3, 0.5, 1.0))
> gbm_grid.1 <- h2o.grid("gbm", x = indep.var, y = dep.var,
+                        grid_id = "gbm_grid.1",
+                        training_frame = train,
+                        validation_frame = valid,
+                        ntrees = 100,
+                        seed = 1000,
+                        hyper_params = gbm_params.1)
```

Output:

```
================

Grid ID: gbm_grid.1
Used hyper parameters:
  -  col_sample_rate
  -  learn_rate
  -  max_depth
  -  sample_rate
Number of models: 36
Number of failed models: 0

Hyper-Parameter Search Summary: ordered by increasing rmse
  col_sample_rate learn_rate max_depth sample_rate          model_ids
             rmse
1             0.5        0.1         9         1.0 gbm_grid.1_model_34
2651.9131575224515
2             1.0        0.1         9         0.7 gbm_grid.1_model_17
 2655.947010657833
3             1.0        0.1         9         1.0 gbm_grid.1_model_35
2660.2528018506137
4             0.5        0.1         9         0.7 gbm_grid.1_model_16
 2661.530909921139
5             0.3        0.1         9         1.0 gbm_grid.1_model_33
2677.8878763480902
```

Case 3: GBM with Random Grid Search

Instead of searching over all possible combinations we can do a random search and specify the time for which we want to do this process of model building. In the code lines below time limit of 180 secs has been set.

```
gbm_params.2 <- list(learn_rate = seq(0.1, 0.3, 0.01),
+                     max_depth = seq(5, 10, 1), #updated
+                     sample_rate = seq(0.9, 1.0, 0.05),
+                     col_sample_rate = seq(0.1, 1.0, 0.1))
>
> search_criteria <- list(strategy = "RandomDiscrete",
+                         max_runtime_secs = 180)
> gbm_grid.2 <- h2o.grid("gbm", x = indep.var, y = dep.var,
+                        grid_id = "gbm_grid.2",
+                        training_frame = train,
+                        validation_frame = valid,
+                        ntrees = 100,
+                        seed = 1000,
+                        hyper_params = gbm_params.2,
+                        search_criteria = search_criteria)
```

Output:

```
Hyper-Parameter Search Summary: ordered by increasing rmse
  col_sample_rate learn_rate max_depth sample_rate          model_ids
             rmse
1             0.6       0.11         9         0.9 gbm_grid.2_model_3
2651.297844499686
2             0.6       0.24         9        0.95 gbm_grid.2_model_8 2
683.3718134920673
3             0.3       0.26         9         1.0 gbm_grid.2_model_4
2687.474578517369
4             0.4       0.14         7         0.9 gbm_grid.2_model_6
2698.513146220439
5             1.0       0.28         5         1.0 gbm_grid.2_model_7
2710.037313719596
6             0.4       0.18         6         0.9 gbm_grid.2_model_1
2715.902246423054
7             0.9       0.14         5         0.9 gbm_grid.2_model_2 2
723.9385093213727
8             0.4       0.15         5         1.0 gbm_grid.2_model_5
2738.512541289683
9             0.1       0.12        10        0.95 gbm_grid.2_model_0 2
788.5655852890422
```

Thus, we found that the best set of parameters which give a minimum RMSE of 2651.3 on validation data are:

col_sample_rate = 0.6
learn_rate = 0.11
max_depth=9
sample_rate=0.9

General Linear Model (GLM)

After the data preparation stage, the train dataset consists of new variables generated by transformations of original variables or dummies. The added variables are listed below:

```
> colnames(train)
 [1] "Product_ID"                  "User_ID"                     "Age"
 [4] "City_Category"               "Occupation"                  "Gender"
 [7] "Marital_Status"              "Stay_In_Current_City_Years"  "Product_Category_1"
[10] "Product_Category_2"          "Product_Category_3"          "Purchase"
[13] "Age_numeric"                 "Gender_numeric"              "Stay_numeric"
[16] "Mean_Purchase_MS"            "Count_MS"                    "Mean_Purchase_Gen"
[19] "Count_Gen"                   "Mean_Purchase_Occ"           "Count_Occ"
[22] "Mean_Purchase_City"          "Count_City"                  "Mean_Purchase_Age"
[25] "Count_Age"                   "Mean_Purchase_User"          "Count_User"
[28] "Mean_Purchase_Product"       "Count_Product"               "User_ID_Gender"
[31] "User_ID_Age"                 "User_ID_Product_ID"          "User_ID_Marital_Status"
[34] "User_ID_Occupation"          "User_ID_City_Category"       "Gender_Age"
[37] "Gender_Product_ID"           "Gender_Marital_Status"       "Gender_Occupation"
[40] "Gender_City_Category"        "Age_Product_ID"              "Age_Marital_Status"
[43] "Age_Occupation"              "Age_City_Category"           "Product_ID_Marital_Status"
[46] "Product_ID_Occupation"       "Product_ID_City_Category"    "Marital_Status_Occupation"
[49] "Marital_Status_City_Category" "Occupation_City_Category"   "GenderF"
[52] "GenderM"                     "Age0-17"                     "Age18-25"
[55] "Age26-35"                    "Age36-45"                    "Age46-50"
[58] "Age51-55"                    "Age55+"                      "Occupation0"
[61] "Occupation1"                 "Occupation2"                 "Occupation3"
[64] "Occupation4"                 "Occupation5"                 "Occupation6"
[67] "Occupation7"                 "Occupation8"                 "Occupation9"
[70] "Occupation10"                "Occupation11"                "Occupation12"
[73] "Occupation13"                "Occupation14"                "Occupation15"
[76] "Occupation16"                "Occupation17"                "Occupation18"
[79] "Occupation19"                "Occupation20"                "City_CategoryA"
[82] "City_CategoryB"              "City_CategoryC"              "Stay_In_Current_City_Years0"
[85] "Stay_In_Current_City_Years1" "Stay_In_Current_City_Years2" "Stay_In_Current_City_Years3"
[88] "Stay_In_Current_City_Years4+" "Marital_Status0"            "Marital_Status1"
[91] "missingProd2"                "missingProd3"                "numProds"
[94] "Product_Category_2_Imp"      "Product_Category_3_Imp"
```

Model-1:

Purchase if the dependent variable, therefore marked as Output:

Input variables in this case consists of dummy variables and new imputed variables:

```
# Initializing input variables
ip <- c(51:90,93:95)
```

```
[49] [redacted]                    [redacted]                    "GenderF"
[52] "GenderM"                     "Age0-17"                     "Age18-25"
[55] "Age26-35"                    "Age36-45"                    "Age46-50"
[58] "Age51-55"                    "Age55+"                      "Occupation0"
[61] "Occupation1"                 "Occupation2"                 "Occupation3"
[64] "Occupation4"                 "Occupation5"                 "Occupation6"
[67] "Occupation7"                 "Occupation8"                 "Occupation9"
[70] "Occupation10"                "Occupation11"                "Occupation12"
[73] "Occupation13"                "Occupation14"                "Occupation15"
[76] "Occupation16"                "Occupation17"                "Occupation18"
[79] "Occupation19"                "Occupation20"                "City_CategoryA"
[82] "City_CategoryB"              "City_CategoryC"              "Stay_In_Current_City_Years0"
[85] "Stay_In_Current_City_Years1" "Stay_In_Current_City_Years2" "Stay_In_Current_City_Years3"
[88] "Stay_In_Current_City_Years4+" "Marital_Status0"            "Marital_Status1"
```

```
[91] [redacted]                    [redacted]                    "numProds"
[94] "Product_Category_2_Imp"      "Product_Category_3_Imp"
```

Fitting the model:

```
system.time(
  linear_model2 <- h2o.glm( y = op, x = ip, training_frame = train, family = "gaussian")
)
```

- **The model took 2.41 seconds to fit**

```
  |=============================================================================================================| 100%
   user  system elapsed
   0.25    0.03    2.41
```

Model performance:

```
> h2o.performance(linear_model)
H2ORegressionMetrics: glm
** Reported on training data. **

MSE:  22795070
RMSE:  4774.418
MAE:  3772.294
RMSLE:  0.7405207
Mean Residual Deviance :  22795070
R^2 :  0.09600331
Null Deviance :1.040026e+13
Null D.o.F. :412448
Residual Deviance :9.401804e+12
Residual D.o.F. :412407
AIC :8158299
```

Model-2:

Purchase if the dependent variable, therefore marked as Output:

Input variables in this case consists of transformed (continuous age variable,) datatypes and new imputed variables:

```
# Initializing input variables
ip <- c(4,5,7,9:11,93:95)
```

```
> colnames(train)
 [1] "[redacted]"                "[redacted]"                "[redacted]"
 [4] "City_Category"             "Occupation"                "[redacted]"
 [7] "Marital_Status"            "[redacted]"                "Product_Category_1"
[10] "Product_Category_2"        "Product_Category_3"        "[redacted]"
```

```
[91] "[redacted]"                "[redacted]"                "numProds"
[94] "Product_Category_2_Imp"    "Product_Category_3_Imp"
```

- **The model took 1.69 seconds to fit**

```
|======================================================================================================| 100%
 user  system elapsed
 0.30    0.00    1.69
```

Model performance:

```
> h2o.performance(linear_model2)
H2ORegressionMetrics: glm
** Reported on training data. **

MSE:  21270586
RMSE:  4612.005
MAE:  3591.521
RMSLE:  0.7290002
Mean Residual Deviance :  21270586
R^2 :  0.1564606
Null Deviance :1.040026e+13
Null D.o.F. :412448
Residual Deviance :8.773032e+12
Residual D.o.F. :412375
AIC :8129814
```

- The model performance of Model-2 is better. Using it to predict the price (GLM output):

```
#making predictions
linear_predict <- as.data.frame(h2o.predict(linear_model2, test))
#create a data frame and writing submission file
a = as.vector(test$User_ID)
b = as.vector(test$Product_ID)
c = as.vector(linear_predict$predict)
predicted_purchase <- data.frame(User_ID = a, Product_ID = b, Purchase = c)
```

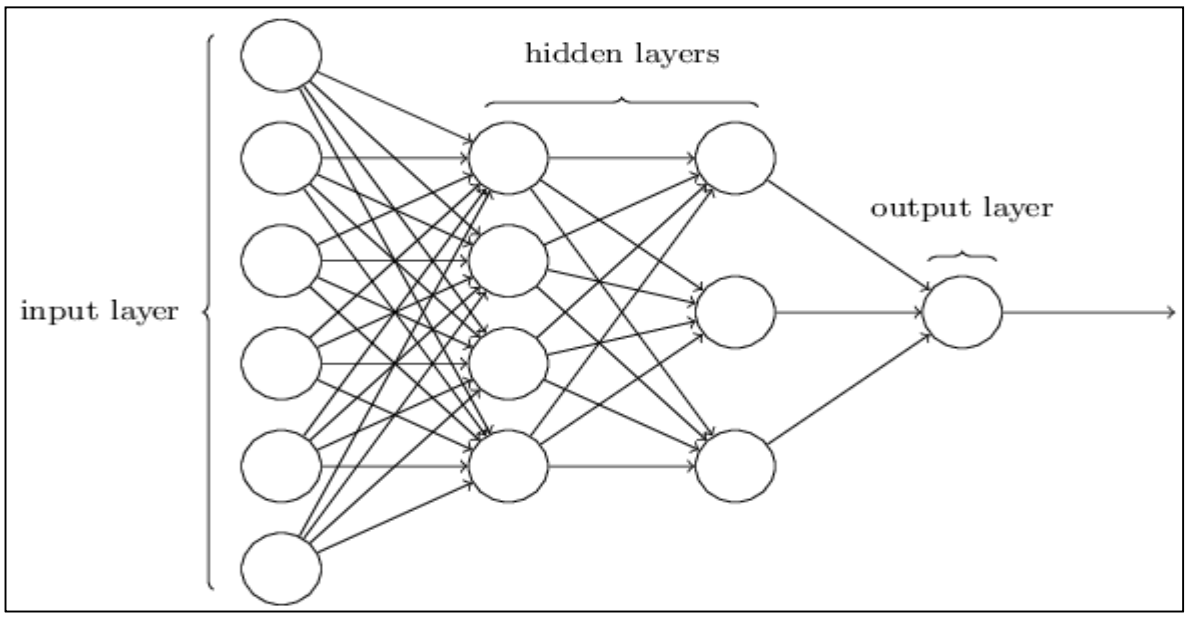
- The predicted purchase value (through GLM) is:

	User_ID	Product_ID	Purchase
1	1000001	P00085442	8973.766
2	1000003	P00193542	9765.988
3	1000004	P00346142	9671.878
4	1000005	P00031342	8033.395
5	1000006	P0096642	10513.548

Deep Learning

H2o uses Multi-Layer perceptron type of neural network. It is also called as Feed forward neural network as the information flows through each layer till it reach the output function. If there is a feedback connection between layers then they are called a recurrent neural networks.

Architecture:



As we can see from above figure, neural network are organized into three layers. Each layers contains interconnected nodes which holds an 'activation function' in them. Raw features are given to input layers. Actual processing is done in hidden layers which try to identify the patterns among the features. Output layers collects all the data and outputs the result.

H2o Deep learning Model Syntax:

```
h2o.deeplearning(x, y, training_frame,
                 validation_frame, nfolds,
                  activation, hidden, stopping_rounds = 5, L1
Regularization,
                 L2 Regularization, Classification Stop, Regression
Stop)
```

Parameters	Definition
X,Y	Column name of dependent and independent variables
training_frame	Data source
validation_frame	Unique dataset with same shape as training dataset, used in model validation
activation	Activation function to be used in neuron ["Tanh", "Rectifier", "Maxout"]

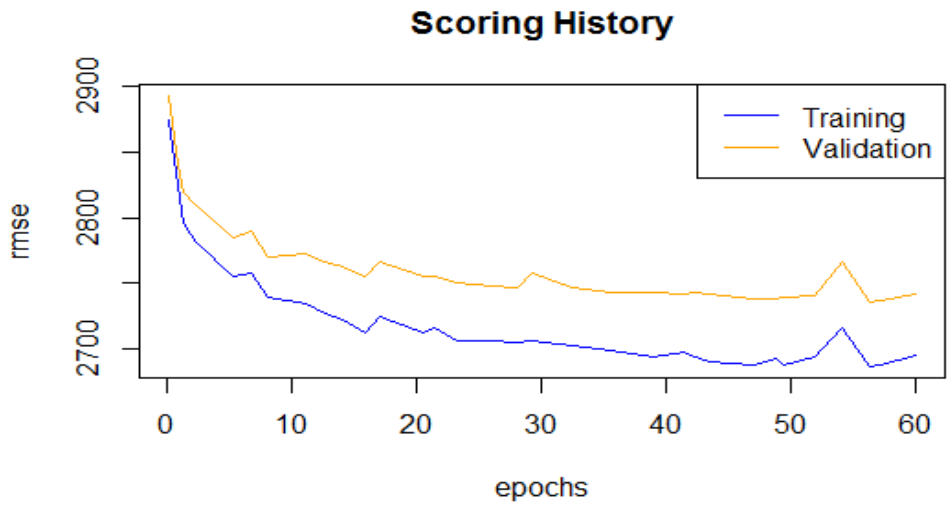
nfolds	Number of folds of cross validation
Hidden	Number and Size of hidden layer Ex: (20,20,20) ; three layers with each layer 20 nodes.
Epochs	Number of passes over the training dataset. Can be tuned for better performance.
L1,L2 Regularization	Regularization that can be used to reduce the complexity of model and avoid overfitting.
Classification, Regression Stop	Stopping criteria in terms of entropy and mean squared error. When error is below this threshold, training stops.

Building Deeplearning model

```
DL <- h2o.deeplearning(x = indep.var, y = dep.var, training_frame = train.h2o,

                       model_id = "dl_fit1", hidden = c(50,50),  seed = 1000)

DL.perf1 = h2o.performance(model = DL,newdata = valid.h2o)

h2o.rmse(DL.perf1)
```

Epochs can be tuned to get the better performance. Following plot show the performance of model on validation and training dataset

```
DL_1 <- h2o.deeplearning(x = indep.var,y = dep.var, training_frame=train.h2o,
validation_frame=valid.h2o,  model_id = "DL_3", epochs = 200,  hidden = c(20,20), nfolds = 3,
score_interval = 1, stopping_rounds = 5,  stopping_metric = "MSE",   stopping_tolerance = 1e-3,
seed = 5555)

plot(DL_1, timestep = "epochs",  metric = "rmse")
```



We can also tune the parameters using “Grid” option in h2o. Let us try to tune hidden, input_dropout_ratio and rate parameters using “RandomDiscrete” strategy.

```
hyper_params <-
list(hidden=list(c(32,32,32),c(64,64,64)),input_dropout_ratio=c(0,0.05),rate=c(0.01,0.02))

search_criteria_DL <- list(strategy = "RandomDiscrete",   max_models = 36)

grid_DL <- h2o.grid(algorithm="deeplearning", grid_id="dl_grid", training_frame=train.h2o,
validation_frame=valid.h2o,  x = indep.var, y = dep.var, epochs=50, stopping_metric="MSE",
stopping_tolerance=1e-2,  stopping_rounds=2, adaptive_rate=F, momentum_start=0.5,
momentum_stable=0.9,  momentum_ramp=1e7,  l1=1e-5,l2=1e-5,  activation = "Rectifier",

 hyper_params=hyper_params,  search_criteria = search_criteria_DL)
```

References:

1. [i] http://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html
2. [ii] https://datahack.analyticsvidhya.com/contest/black-friday/
3. https://www.analyticsvidhya.com/blog/2016/05/h2o-data-table-build-models-large-data-sets/
4. http://docs.h2o.ai/h2o/latest-stable/index.html
5. http://www.h2o.ai/
6. Machine Learning with R and H2O – Aiello, http://www.h2o.ai/resources/
7. H2O - Hands on with R, Python and Flow with Amy Wang
8. https://datahack.analyticsvidhya.com/contest/black-friday-data-hack/
9. https://en.wikipedia.org/wiki/H2O_(software)
10. https://www.analyticsvidhya.com/blog/2015/11/secrets-from-data-hackathon/
11. https://en.wikipedia.org/wiki/Generalized_linear_model
12. https://www.gitbook.com/book/h2o/h2o-world-2015-training/details
13. http://sli.ics.uci.edu/Classes/2015W-273a