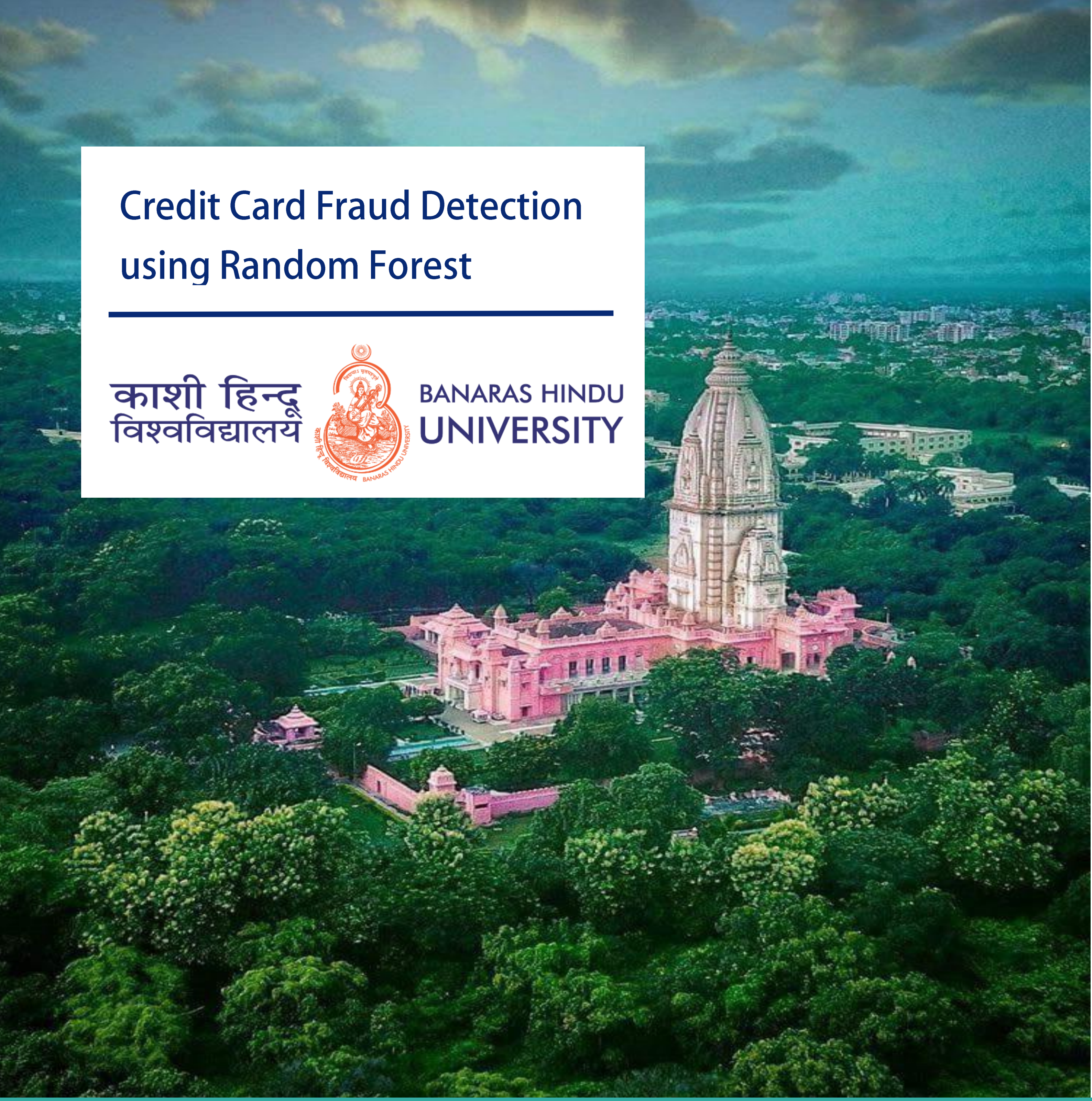
Credit Card Fraud Detection using Random Forest

काशी हिन्दू विश्वविद्यालय BANARAS HINDU UNIVERSITY



SUBMITTED BY

RAHUL GOSWAMI
MSC . Statistics and Computing
Roll No. 18419STC019
Enrollment No . 404655
DST-CIMS
Institute of Science ,BHU

UNDER SUPERVISION OF

Dr. Mahaveer Singh Panwar
Department of Statistics
BHU

ACKNOWLEDGEMENT

I want to thank MAHADEV for everything

It gives me immense pleasure to express my gratitude, gratefulness and indebtedness to reverend teacher and supervisor Dr. Mahaveer Singh Panwar ,Department of Statistics, Banaras Hindu University for his untiring help, constant encouragement, worthy supervision without which it would not have been possible for me to complete this project work. The guidance and valuable criticism that I received from him, during the entire period of the work, has been a great help in the completion of this work.

I acknowledge with my regard and sincere thanks to all the developers of Julia , Sanddance and ggplot .

This was the first time when I got this opportunity to experience to work on Algorithms from scratch of Machine Learning & hence I feel it, as an achievement of my life.

CERTIFICATE

This is to certify that **RAHUL GOSWAMI** student of DST-CIMS, Banaras Hindu University , enrollment number 404655, have completed the project entitled "**Credit Card Fraud Detection Using Random Forest" ,** conducted under the guidance and supervision of **Dr. Mahaveer Singh Parmar,** Department of Statistics, Banaras Hindu University

DATE:

Dr. MAHAVEER SINGH PANWAR

DEPARTMENT OF STATISTICS

Banaras Hindu University

Credit Card Fraud Detection

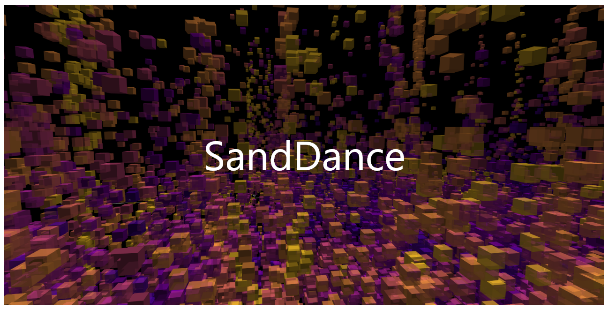
Contents

Introduction

Nowadays, Machine Learning algorithms are used everywhere to predict, those which we cannot see with our bared eye, it is true that humans have extraordinary reasoning skill, but in computing we somehow lag machines. Today data is everywhere, and these data leads to learning process, reasoned by human brains which leads exceptional result that we have not accepted. There are a lot of Machine Learning techniques, but the one we are going to use here is Random Forest, we are going to explore basic random forest and deduct some conclusion for the estimation of hyperparameters.

Technology

We are going to use Julia as a programming language to write the code for random forest from **scratch** and we are using microsoft's sanddance and ggplot2 from R as a visualization tool.



Julia

Julia is a latest programming , language developed at MIT. It mainly focuses on Data Science.We are using this language because it is easy to use in multithreadining which is quite helpful in running such a high level of code which breaks computational barrier of code on simple home Pc or laptops . However Julia is still in progress and most of the packages are not developed that is why we are writing our code from scratch with a little bit help from basic packages such as.

- DataFrames : To handle DataFrames
- CSV : To import CSV file
- Random : To generate Random number

To get using this package we must include following code

```
using Pkg

#Pkg.add("Random");
#Pkg.add("CSV");
```

```
#Pkg.add("DataFrames");

using Random
using CSV
using DataFrames
```

Sanddance

Sandance is webbased opensource application , can be used in Visual Studio Code , Developed by Visualization and Interactive Data Analysis Team (VIDA) in Microsoft Research to ease in Visulization. It have diiferent plotting options , and mainly ised with Graphical Interface , there is not any CLI interface for it. While it is easy to use and have a decent learning Curve.

ggplot2

ggplot2 is one of the most advanced visualizing package in R . It is quite popular in data scientist and it is being used in most of the places from enterprises to academics.ggplot2 is a system for declaratively creating graphics, based on The Grammar of Graphics book written by Leland Wilkinson.from the ggplot2 github it is described as

> ggplot2 is now over 10 years old and is used by hundreds of thousands of people to make millions of plots. That means, by-and-large, ggplot2 itself changes relatively little. When we do make changes, they will be generally to add new functions or arguments rather than changing the behaviour of existing functions, and if we do make changes to existing behaviour we will do them for compelling reasons.

Data

The data we are going to use is a credit card data, fetched from Kaggle, the link for the data is https://www.kaggle.com/mlg-ulb/creditcardfraud/download, It contains transaction made by credit card holders of Europe for two days . It has 492 frauds out of 284,807 transactions. Dataset is **highly unbalanced**. It contains 31 Columns.

- Class: It's a binary flag 1 if transaction is fraudulent and 0 if not
- Time : Time elapsed between transaction
- Amount: Amount of transactions
- V1-V28: PCA output, not explained due to confidentiality

Now let us take a look at our dataset.

```
data = CSV.read("creditcard.csv");
first(data,10)
```

```
## 10×31 DataFrame. Omitted printing of 25 columns
##   Row   Class  Time     V1         V2          V3         V4
##         Int64  Float64  Float64    Float64     Float64    Float64
##
##   1     0      0.0      -1.35981   -0.0727812  2.53635    1.37816
##   2     0      0.0      1.19186    0.266151    0.16648    0.448154
##   3     0      1.0      -1.35835   -1.34016    1.77321    0.37978
##   4     0      1.0      -0.966272  -0.185226   1.79299    -0.863291
##   5     0      2.0      -1.15823   0.877737    1.54872    0.403034
##   6     0      2.0      -0.425966  0.960523    1.14111    -0.168252
##   7     0      4.0      1.22966    0.141004    0.0453708  1.20261
##   8     0      7.0      -0.644269  1.41796     1.07438    -0.492199
##   9     0      7.0      -0.894286  0.286157    -0.113192  -0.271526
##   10    0      9.0      -0.338262  1.11959     1.04437    -0.222187
```

Now to know about our dataset more we will use `describe()` function from the **DataFrames** Package.

```
show(describe(data),allrows = true)
```

```
## 31×8 DataFrame. Omitted printing of 2 columns
##   Row   variable  mean           min       median       max       nunique
##         Symbol    Float64        Real      Float64      Real      Nothing
##
##   1     Class     0.00172749     0         0.0          1
##   2     Time      94813.9        0.0       84692.0      172792.0
##   3     V1        1.7587e-12     -56.4075  0.0181088    2.45493
##   4     V2        -8.2523e-13    -72.7157  0.0654856    22.0577
##   5     V3        -9.63744e-13   -48.3256  0.179846     9.38256
##   6     V4        8.31623e-13    -5.68317  -0.0198465   16.8753
##   7     V5        1.59201e-13    -113.743  -0.0543358   34.8017
##   8     V6        4.24731e-13    -26.1605  -0.274187    73.3016
##   9     V7        -3.05018e-13   -43.5572  0.0401031    120.589
##   10    V8        8.69288e-14    -73.2167  0.022358     20.0072
##   11    V9        -1.17971e-12   -13.4341  -0.0514287   15.595
##   12    V10       7.09492e-13    -24.5883  -0.0929174   23.7451
##   13    V11       1.87502e-12    -4.79747  -0.0327574   12.0189
##   14    V12       1.05351e-12    -18.6837  0.140033     7.84839
##   15    V13       7.13757e-13    -5.79188  -0.0135681   7.12688
##   16    V14       -1.49137e-13   -19.2143  0.0506013    10.5268
##   17    V15       -5.22595e-13   -4.49894  0.0480715    8.87774
##   18    V16       -2.28069e-13   -14.1299  0.0664133    17.3151
##   19    V17       -6.42845e-13   -25.1628  -0.0656758   9.25353
##   20    V18       4.959e-13      -9.49875  -0.00363631  5.04107
##   21    V19       7.06069e-13    -7.21353  0.00373482   5.59197
##   22    V20       1.76604e-12    -54.4977  -0.0624811   39.4209
##   23    V21       -3.40654e-13   -34.8304  -0.0294502   27.2028
##   24   V22         -5.71336e-13   -10.9331   0.00678194    10.5031
##   25   V23         -9.72529e-13   -44.8077   -0.0111929    22.5284
##   26   V24         1.46414e-12    -2.83663   0.0409761     4.58455
##   27   V25         -6.98909e-13   -10.2954   0.0165935     7.51959
##   28   V26         -5.61525e-13   -2.60455   -0.0521391    3.51735
##   29   V27         3.33211e-12    -22.5657   0.00134215    31.6122
##   30   V28         -3.51888e-12   -15.4301   0.0112438     33.8478
##   31   Amount      88.3496        0.0        22.0          25691.2
```

Procedure

We are goona train a random forest for credit card fraud detection , such that the model can predict the a given transaction is fraudulent or not. We are going to use one of the methodology given by Shiyang Xuan and others at https://ieeexplore.ieee.org/document/8361343 Published at 2018 IEEE 15th International Conference on Networking, Sensing and Control (ICNSC). We are going to use Random Forest I from the paper . ** To write bagging , how it helps in lower variance , why not pruning etc. **

Algorithm

- Creating Tree
  - Step 1 : Bootstrap Random Samples of size M from the data
  - Step 2 : Ramdomly select n number of features , where n should be $\surd$(no.offeatures) as recommended by Debian
  - Step 3 : Create a Root node , With this subset of data that we have produced in from Step 1 and Step 2
  - Step 4 : Classify the data in different classes and calculate their average (usually mean) usuall called centers
  - Step 5 : Calculate the Manhattan Distance of data row with centers
  - Step 6 : Create two child node, attach data row to left child if distance is smaller to class 0 center and vie versa
  - Step 7 : Repeat Step 2-6 until nodes contain data from one class
- Creating Forest
  - Create araay of k number of trees
  - Prediction
    - define threshold t
    - if the number of of trees giving decision "fraud" is greater than t , then classify the transaction as fraud
    - otherwise define it as "non - fraud"

Objective

We have a obstacle in our way, we know we take majority vote to decide whether the transaction is fraudulent or not and in the binary case the decision threshold is often 0.5 means we classify a transaction as fraudulent when in a forest more than half of the trees gives decision that the transaction is fraudulent but it is not always true , as in our case the the data is unbalanced and it is most often in credit card fraud detection. This is since most of the transactions are not fraud and the proportion of fraud is very less as it is in our case. So our objective is to decided a threshold for our data , and interpret it why is it so ? . Another objective is to find the optimal number of trees for the decision and the numbers of bootstrap samples. One other objective is to prove that Accuracy is useless in imbalance data.

Performance Measures

We are gonna make a confusion matrix , confusion matrix is 2×2 matrix contains different value , which we will be used to calculate various performance measures such as accuracy , F-measure , Precision and recall. Confusion Matrix is given by

Confusion Matrix	Fraud	Non-Fraud
Predicted Fraud	TP	FP
Predicted Non-Fraud	FN	TN

- TP : True Positive is the number of cases in which we predicted positive and its true
- TN : True Negative is the number of cases in which we predicted negative and its true
- FP : False Positive also known as type I error,in this case we predict positive and its false
- FN : False Negative also known as type II error , in this case we predict negative and its false

Now Various Measure can be calculated by

$F-measure = \frac{2TP}{2TP+FP+FN}$	$Precision = \frac{TP}{TP+FP}$
$Recall = \frac{TP}{TP+FN}$	$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$

Accuracy Paradox

Accuracy Paradox is a glitch in the metric of accuracy, it happens mostly in imbalanced data as in our case , as accuracy is not a good measure for type of data , it shows that the accuracy is good , but actually performance is very bad we are gonna represent this in our conclusion , we also gonna plot this

** We are gonna us F-measure as our primary Performance Metric **

Data Preprocessing

We are gonna use two data processing , that is

- Missing Value Ratio
- Low Variance Filter

Missing Value Ration

In missing value we generally find the ration of missing value in the features , we just sum up the missing values and divide by the length of the column and then multiply it by 100 to get the percentage of missing value in that column

```
for col in eachcol(data)
  a=sum(ismissing.(col))/length(col)
  println("$a missing data  ")
end
```

```
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
## 0.0 missing data
```

So we can see we have not any Missing Data

Why not Heat Map and Scaling

Our data is quite straight forward most of the data is PCA , so there is minmmum correlation in the data , and scaling have not any benifit in random forest , it is gonna time waste .

Code

We are going to divide our code part in three section , **Functions** which contains the functions we are going to create to run the execution smoothly, **Execution** Execution part to train the model and call the functions on the data and **Visualization**

Functions

We are going to create functions , those are

(1) `rediv()`

It is a function to import CSV format data to julia , and further divide the data set in test , train and validation set . It returns validation and train set whereas it set test set as global so that it can be accessed from any part of the code

```
function rediv()
  data= CSV.read("creditcard.csv")  # Importing data
  x=rand(nrow(data)) ;              # Generating Random variable
  y=x.> 0.8;                        # Setting Boolean Random variable
  test_set=data[y,:];               # Extracting Test set
  train_set=data[.!y,:];
  z=rand(nrow(train_set)).>0.7;
  val_set=train_set[z,:];           # Extracting Train Set
  train_set=train_set[.!z,:];       # Extracting Validation Set
  CSV.write("train_set.csv",train_set)
  CSV.write("val_set.csv",val_set)
  global test_set
  return train_set , val_set
end
```

(2) `sampler()`

`sampler()` function takes three inputs data , n and i. Its main function is to divide the data set by rows when i = 1 and by columns when i =0 and n define the number of rows or columns to be extracted.

```
function sampler(x,n,i)
    if i==1              # for sampling row
           x[shuffle(1:nrow(x))[1:n], :]
    elseif i==0         #for sampling columns
            x[:, shuffle(1:ncol(x))[1:n]]
    else
            return print("i should be 1 or 0")
    end
end
```

(3) `classifier()`

It takes two arguments first one is data and second on is a bool argument when true returns subset of the first argument whose class is true means only fraud transaction and when false it returns only non-fraud transaction

```
function classifier(x,t::Bool) #divide the dataframe x according to different class (i
    a=(x[:,1].==t)
    return x[a,:]
end
```

(4) `center()`

Now center function , will find center of the data (in our case it is mean of coloumn) ,of given class , so our function will take data set and classify the data according to a bool input using our `classifier()` that we just created , then calculate mean of the data set returned by our classifier function , and then it will truncate the first column average because we don't need mean of the feature "class" fo further use

```
function center(x,class::Bool)
        classified=classifier(x,class)
        colmean=[sum(col)/length(col) for col = eachcol(classified)]
        return colmean[2:ncol(x)]
 end
```

(5) `distance()`

This distance function will find the distance between the data and center

```
function distance(center,x)
    k=[]
    for i in 1:nrow(x)
        a=convert(Array{Float64,1},x[i,:])
        a=a[2:ncol(x)]
        j=sum(abs.((a-center)))
        k=push!(k,j)
    end
    return k
end
```

(6) `node()`

now defining node() which will create node of the tree but before that we need to define a structure called node using `struct` command that contains names of the coloumns selected randomly , center and the data

```
struct rnode
    symbol::Array{Symbol,1}
    centre
    data
end
```

Now defining the node function

```
function node(x)
    y       =x[:,1]                     #target variable
    l       =x[:,2:ncol(x)]             #feature
    m       = floor(sqrt(ncol(l)))      #the number of fetures to be randomly taken
    m       = convert(Int64,m)          #converting number on features in integer varia
    j       = (sampler(l,m,0))          # Sampling m coloumns
    data    = hcat(y,j)                 #concating target variable with feature variabl
    center0 = center(data,false)        #calculating center for the data having class 0
    center1 = center(data,true)             #calculating center for the data having cla
    dist0   = distance(center0,data)    #calculating distance of center 0 from data
    dist1   = distance(center1,data)    #calculating distance of center 1 from data
    data0   = x[dist0 .<= dist1,:]
    data1   = x[dist0 .> dist1,:]
    k=rnode(names(j),[center0,center1],[data0,data1])
    return k
end
```

(7) `tree()`

Now we will create tree() function but before that we need to define the strucure tree which is just a nested loop

```
struct rtree
    symbol::Array{Symbol,1}
    center
    data
end
```

now defining our tree function

```
function tree(y)
    k=node(y)
    datum1=k.data[2]        #extracting the data which is nearer to class 1
    datum0=k.data[1]
    if sum(datum1[:,1]) == length(datum1[:,1]) && sum(datum0[:,1]) == 0
        return rtree(k.symbol,k.center,[nothing,nothing])
    elseif sum(datum1[:,1]) == length(datum1[:,1]) && sum(datum0[:,1]) != 0
        return rtree(k.symbol,k.center,[tree(datum0),nothing])
    elseif sum(datum0[:,1]) == 0 && sum(datum1[:,1]) != length(datum1[:,1])
        return rtree(k.symbol,k.center,[nothing,tree(datum1)])
    else
        return rtree(k.symbol,k.center,[tree(datum0),tree(datum1)])
    end
end
```

(8) `forest()`

Now our function forest is just an array of tree trained different bootstrapped sample rows from the train set

```
 function forest(x,n,l)
    y=deepcopy(x)
    # x is dataset
    # n is no. of samples for each tree
    # l is no. of tree
    forest=[]
    for i in 1:l
        data  = sampler(y,n,1)
        trained_tree= tree(data)
        push!(forest,trained_tree)
    end
    return forest
end
```

(9) `treepredict()`

Now treepredict function will predict a dataset and will attach a column to the dataset named class , that represent the dataset is fraud or not a fraud , it will take trained model a tree as an input and a dataframe , then it will start dividing dataset unless `nothing` is observed at the tree end then it will return that dataframe at the end with a class coloumn with value equal to 1 if it prunned at right side of the node and if nothing is observed at the left end it will return class 0 .

```
function treepredict(atree,test)
    d0=sum.(eachrow(abs.((convert(Matrix,test[:,atree.symbol])).- (atree.center[1])')))
    d1=sum.(eachrow(abs.((convert(Matrix,test[:,atree.symbol])).- (atree.center[2])')))
    a=(d0.<d1)
    b= (.!a)
    if atree.data[1]==nothing && atree.data[2]!=nothing
        return vcat(select!(insertcols!(test[a,:],1,:Class=>repeat([0],nrow(test[a,:])))
    elseif atree.data[2]==nothing && atree.data[1]!=nothing
        return vcat(select!(insertcols!(test[b,:],1,:Class=>repeat([1],nrow(test[b,:])))
    elseif atree.data[2]==nothing && atree.data[1]==nothing
```

```
        return select!(vcat(insertcols!(test[a,:],1,:Class=>repeat([0],nrow(test[a,:])))
    else

        return  vcat(treepredict(atree.data[1],test[a,:]),treepredict(atree.data[2],test
    end
end
```

(10) `sort_pred()`

: There is a problem with our `treepredict()` function , it returns classified dataset and it returns ransomly so we observe shuffled rows so thats why this function will attach a column named * index * and then it will run treepredict() function and lastly sort the return from the treepredict function according to index coloumn then remove index function and return.

```
function sort_pred(atree,test)
    test1=deepcopy(test)
    insertcols!(test1,1,:INDEX=>1:nrow(test))
    return select!(sort!(treepredict(atree,test1),:INDEX),Not(:INDEX))
end
```

(11) `forest_pred()`

Now `forest_pred()` will return a dataframe, with a specified number of coloumn that is to the number of tree in a forest that it will take as input , where each coloumn is a decision of a tree

```
function forest_pred(forest,test)
    a=DataFrame()
    for i in forest
        a=hcat(a,sort_pred(i,test),makeunique=true)
    end
    return a
end
```

Execution

In this part we will use the functions to **train** the data and different **measure** such as

- Precision
- Recall
- Accuracy
- f-measure

Now let us use `rediv()` to get our data set ready

```
train_set , val_set = rediv()
a = reverse(nrow(train_set):-1000:1000)    # number of samples to train
b = 100                                     # maximum number of tree to train
```

Now let us take a look at our train_set, test_set and val_set

```
print(train_set)
print(test_set)
print(val_set)
```

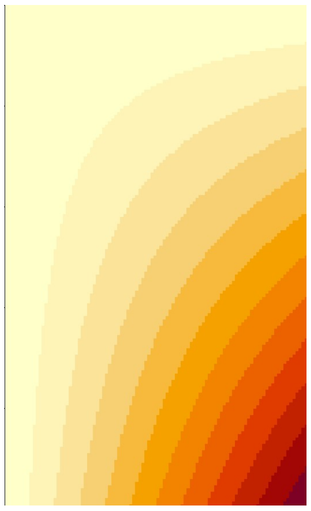
```
## 159322×31 DataFrame. Omitted printing of 25 columns
##   Row     Class  Time      V1        V2         V3         V4
##           Int64  Float64   Float64   Float64    Float64    Float64
##
##   1       0      0.0       -1.35981  -0.0727812 2.53635    1.37816
##   2       0      10.0      1.44904   -1.17634   0.91386    -1.37567
##   3       0      10.0      0.384978  0.616109   -0.8743    -0.0940186
##   4       0      11.0      1.06937   0.287722   0.828613   2.71252
##   5       0      12.0      -2.79185  -0.327771  1.64175    1.76747
##   6       0      12.0      -0.752417 0.345485   2.05732    -1.46864
##   7       0      12.0      1.10322   -0.0402962 1.26733    1.28909
##
##   159315  0      172780.0  1.88485   -0.14354   -0.999943  1.50677
##   159316  0      172782.0  -0.241923 0.712247   0.399806   -0.463406
##   159317  0      172782.0  0.219529  0.881246   -0.635891  0.960928
##   159318  0      172783.0  -1.77513  -0.0042354 1.18979    0.331096
##   159319  0      172784.0  2.03956   -0.175233  -1.19683   0.23458
##   159320  0      172785.0  0.120316  0.931005   -0.546012  -0.745097
##   159321  0      172786.0  -11.8811  10.0718    -9.83478   -2.06666
##   159322  0      172788.0  1.91957   -0.301254  -3.24964   -0.557828

## 56913×32 DataFrame. Omitted printing of 27 columns
##   Row    predicted_class  Given_Class  Time      V1        V2
##          Bool             Int64        Float64   Float64   Float64
##
##   1      0                0            0.0       1.19186   0.266151
##   2      0                0            1.0       -1.35835  -1.34016
##   3      0                0            1.0       -0.966272 -0.185226
##   4      0                0            2.0       -1.15823  0.877737
##   5      0                0            2.0       -0.425966 0.960523
##   6      0                0            7.0       -0.894286 0.286157
##   7      0                0            10.0      1.25      -1.22164
##
##   56906  0                0            172759.0  -0.822731 1.27014
##   56907  0                0            172762.0  1.95555   -0.724606
##   56908  0                0            172764.0  2.07914   -0.0287234
##   56909  0                0            172766.0  1.97518   -0.616244
##   56910  0                0            172768.0  -0.669662 0.923769
##   56911  0                0            172774.0  -0.724123 1.48522
##   56912  0                0            172777.0  -1.26658  -0.400461
##   56913   0                 0            172787.0   -0.732789   -0.0550805

## 68572×31 DataFrame. Omitted printing of 25 columns
##   Row     Class   Time       V1          V2          V3          V4
##           Int64   Float64    Float64     Float64     Float64     Float64
##
##   1       0       4.0        1.22966     0.141004    0.0453708   1.20261
##   2       0       7.0        -0.644269   1.41796     1.07438     -0.492199
##   3       0       9.0        -0.338262   1.11959     1.04437     -0.222187
##   4       0       13.0       -0.436905   0.918966    0.924591    -0.727219
##   5       0       15.0       1.49294     -1.02935    0.454795    -1.43803
##   6       0       22.0       -1.94653    -0.0449005  -0.40557    -1.01306
##   7       0       26.0       -0.529912   0.873892    1.34725     0.145457
##
##   68565   0       172768.0   -2.07617    2.14224     -2.5227     -1.88806
##   68566   0       172769.0   -1.02972    -1.11067    -0.636179   -0.840816
##   68567   0       172770.0   2.00742     -0.280235   -0.208113   0.335261
##   68568   0       172770.0   -0.446951   1.30221     -0.168583   0.981577
##   68569   0       172771.0   -0.515513   0.97195     -1.01458    -0.677037
##   68570   0       172774.0   -0.863506   0.874701    0.420358    -0.530365
##   68571   0       172788.0   -0.24044    0.530483    0.70251     0.689799
##   68572   0       172792.0   -0.533413   -0.189733   0.703337    -0.506271
```

Now let us train the model , we are going to `Threads.@threads` for the for loop so that we can use multiple threads at the same time for faster ,

Let us understand our jungle array it is a 2 dimensional matrix , we can visualize it as a a rectangle which consist very small rectangles, which is our single forest, and as we move from top right the number of tree starts increasing and as we move down number samples starts in increasing so as we move down or right computational barrier starts increasing as in this figure



```
jungle = Array{Any}(undef, length(a))
Threads.@threads for i in shuffle(1:length(a))
                        jungle[i]=forest(train_set,a[i],b)
                 end
```

Now let us store the row number at which our validation set have class 1 that is fraud and 0 that is non-fraud transaction , and further remove the coloumn class from the validation set

```
Class1_index=(1:nrow(val_set))[(val_set[:,1].==1)];
Class0_index=(1:nrow(val_set))[(val_set[:,1].==0)];
select!(val_set,Not(:Class))
```

Now let us predict the transaction in our validation data sets, here `jungle_pred` is a four dimensional array of which first dimension is number of samples in a forest , that is equal to the number of transaction in train set and further repeatedly reducing by 1000 , just for example let us say there are 13200 transaction in train set then first forest is trained with 13200 transaction , then 12200 , then 11200 and go on , while the second dimension represent the number of tree which starts from 100 go to 1 , third argument is threshold after hoe much percent of the tree we assume it is fraud the thresholds are 0.1 ,0.2 ,0.3, 0.4, 0.5, 0.6 , 0.7 , 0.8 , 0.9 ,1.0 and the last argument is prediction that in boolean where 1 represents fraud and 0 represents non-fraudulent transactions.

```
jungle_pred = BitArray(undef, length(a), b,10,nrow(val_set));
nt = Array{Array{Int8}}(undef, length(a),b)
Threads.@threads for i in shuffle(1:length(a))
    pre = forest_pred(jungle[i],val_set)
    for j in b:-1:1
        l=(1:b)[shuffle(1:j)]
        nt[i,j]=l
        suf = pre[:,l]
        for k in 1:10
             jungle_pred[i,j,k,:]=  ((sum(eachcol(suf))/ncol(suf)) .> k*0.1)
        end
    end
end
```

Now our `jungle_pred` array contains prediction of about **159901** forest which consist of 8029500 trees , now sice we have done prediction let us calculate indicators such as

- True Positive (TP) : Means the actual class is fraud and we have predicted class fraud given by `jungle_ind[:,:,:,1]`
- True Negative (TN) : Means the actual class is non-fraud and we have predicted class non-fraud given by `jungle_ind[:,:,:,4]`
- False Positive(FP) : Means prediction is fraud but actually the transaction is not fraud `jungle_ind[:,:,:,3]`
- False Negative(FN) : Means prediction is non fraud but actually it is a fraud `jungle_ind[:,:,:,2]`

The code for calculating indicators are

```
jungle_ind=zeros(Int64 , length(a), b,10,4);

for i in 1:length(a)
  for j in 1:b
    for k in 1:10
      Threads.@threads  for m in Class1_index
                          if jungle_pred[i,j,k,m]==1
                            jungle_ind[i,j,k,1]=jungle_ind[i,j,k,1]+1
                          elseif jungle_pred[i,j,k,m]==0
                            jungle_ind[i,j,k,3]=jungle_ind[i,j,k,3]+1
                          end
                        end
      Threads.@threads for n in Class0_index
                          if jungle_pred[i,j,k,n]==1
                            jungle_ind[i,j,k,2]=jungle_ind[i,j,k,2]+1
                          elseif jungle_pred[i,j,k,n]==0
                            jungle_ind[i,j,k,4]=jungle_ind[i,j,k,4]+1
                          end
                        end
    end
  end
end
```

Now We must Calculate Various Measure , such as precision , accuracy, fmeasure , recall

```
measure=zeros(Float64 , length(a), b,10,4);

Threads.@threads for i in 1:length(a)
for j in 1:b
  for k in 1:10
    if all(jungle_ind[i,j,k,1] .== 0)
      measure[i,j,k,1]=0
      measure[i,j,k,2]=0
    else
      measure[i,j,k,1]=jungle_ind[i,j,k,1]/(jungle_ind[i,j,k,1]+jungle_ind[i,j,k,2])
      measure[i,j,k,2]=jungle_ind[i,j,k,1]/(jungle_ind[i,j,k,1]+jungle_ind[i,j,k,3])
    end
    if all(jungle_ind[i,j,k,1]+jungle_ind[i,j,k,4] .== 0)
    measure[i,j,k,3]=0
    else
  measure[i,j,k,3]=(jungle_ind[i,j,k,1]+jungle_ind[i,j,k,4])/(jungle_ind[i,j,k,1]+jungle
    end
    if measure[i,j,k,1]*measure[i,j,k,2]==0
      measure[i,j,k,4]=0
    else
    measure[i,j,k,4]=(2*measure[i,j,k,1]*measure[i,j,k,2])/(measure[i,j,k,1]+measure[i,j
    end
end
end
end
```

Now storing measure in a dataframe

```
df = DataFrame(Sample = Int64[], Tree = Int64[],Threshold=Float64[], Precission=Float64[

for i in 1:length(a)
    for j in 1:b
        for k in 1:10
            push!(df, [a[i] j k*0.1 measure[i,j,k,1] measure[i,j,k,2] measure[i,j,k,3] m
        end
    end
end
CSV.write("measures.csv",df)
```

Now let us take a look at our measure dataframe

```
print(measures)
```

```
## 159000×7 DataFrame. Omitted printing of 1 columns
##  Row     Sample  Tree   Threshold  Precission  Recall    Accuracy
##          Int64   Int64  Float64    Float64     Float64   Float64
##
##  1       1322    1      0.1        0.860215    0.714286  0.999146
##  2       1322    1      0.2        0.860215    0.714286  0.998863
##  3       1322    1      0.3        0.857143    0.709091  0.998772
##  4       1322    1      0.4        0.857143    0.715596  0.99894
##  5       1322    1      0.5        0.858696    0.711712  0.998767
##  6       1322    1      0.6        0.857143    0.709091  0.998772
##  7       1322    1      0.7        0.850575    0.698113  0.998763
##
##  158993  159322  100    0.3        0.967742    0.810811  0.999441
##  158994  159322  100    0.4        0.967391    0.801802  0.999319
##  158995  159322  100    0.5        0.978261    0.803571  0.999344
##  158996  159322  100    0.6        0.977528    0.783784  0.999282
##  158997  159322  100    0.7        0.974684    0.693694  0.999018
##  158998  159322  100    0.8        1.0         0.616071  0.999076
##  158999  159322  100    0.9        1.0         0.5       0.998595
##  159000  159322  100    1.0        0.0         0.0       0.997307
```

Now we have to find maximum F-measure so we will use `findmax()` function to get where the f-measure is highest

```
findmax(measures)
```

```
## (0.898989898989899, 132713)
```

So 132713 row , gives the maximum f-measure let us take a look at it

```
measures[132713,:]
```

```
## DataFrameRow. Omitted printing of 1 columns
##   Row      Sample   Tree    Threshold   Precission   Recall     Accuracy
##            Int64    Int64   Float64     Float64      Float64    Float64
##
##   132713   133322   72      0.3         0.967391     0.839623   0.999539
```

So we get out metrics as follows

Metrics & Settings	Observed
Sample Size	132719
No. of Tree	72
Threshold	0.3
Precision	0.967391 = 97%
Recall	0.839623 = 84%
Accuracy	0.999539 = 100%
F-Measure	0.898989 = 90%

Now let us predict for the sample size 132719 , with 72 tree and threshold 0.3

```
rename!(test_set,:Class=>:Given_Class)
prediction = forest_pred(jungle[cartesian[1]],test_set)[:,nt[cartesian[1],cartesian[2]]]
prediction_forest = sum(eachcol(absd))/ncol(absd)
res=(absd .> cartesian[3]*0.1)
insertcols!(test_set,1,:predicted_class=>res)
Class1_index=(1:nrow(test_set))[(test_set[:,3].==1)];
Class0_index=(1:nrow(test_set))[(test_set[:,3].==0)];
TP=sum(test_set[Class1_index,2].==1)
TN=sum(test_set[Class0_index,2].==0)
FP=sum(test_set[Class1_index,2].==0)
FN=sum(test_set[Class0_index,2].==1)
println("True Positive = $TP , True Negative = $TN , False Positive = $FP , False Negati
```

```
## True Positive = 83 , True Negative = 56800 , False Positive = 24 , False Negative = 6
```

So Now We can Calculate Confusion Matrix with this result

Confusion Matrix	Fraud	Non-Fraud
Predicted Fraud	83	24

Confusion Matrix	Fraud	Non-Fraud
Predicted Non-Fraud	6	56800

Now let us calculate Performance Metrics

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} = \frac{83+56800}{83+56800+24+6} = \frac{56883}{56913} = 0.99947 \approx 100\%$$

$$Recall = \frac{TP}{TP+FN} = \frac{83}{83+6} = \frac{83}{89} = 0.96629 \approx 97\%$$

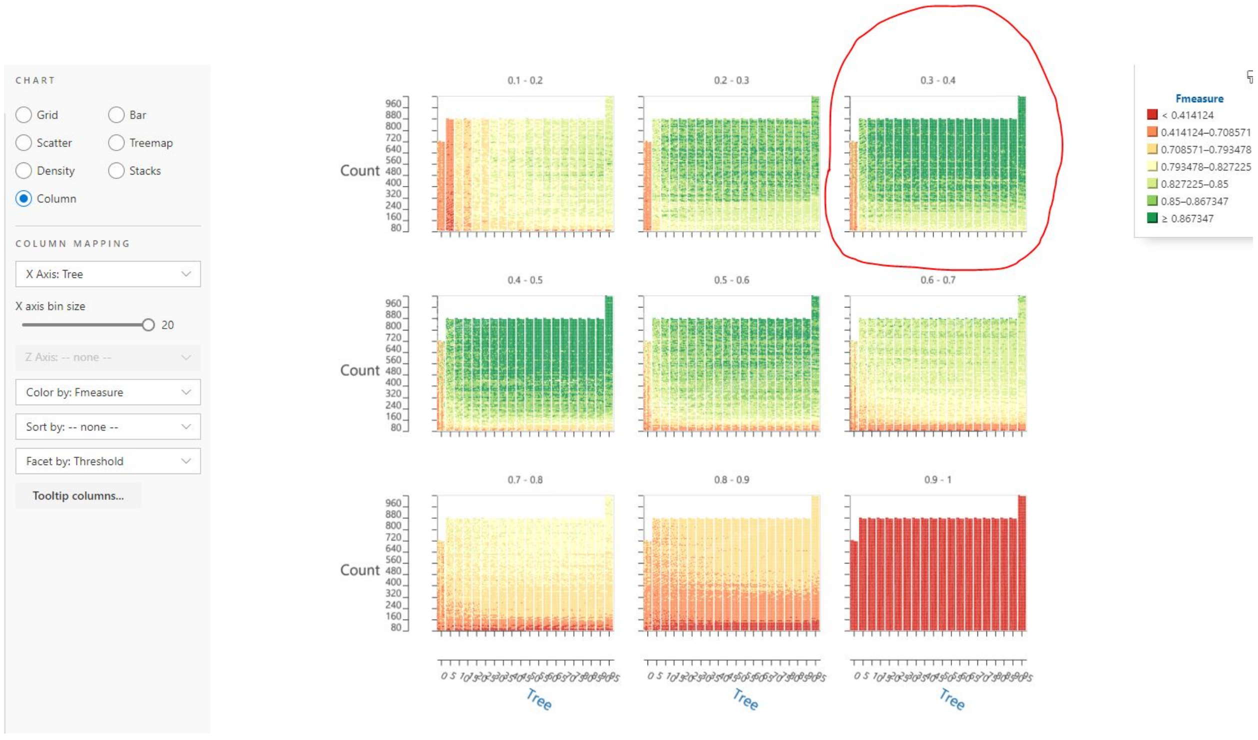
$$Precision = \frac{TP}{TP+FP} = \frac{83}{83+24} = \frac{83}{107} = 0.7757 \approx 78\%$$

$$F-measure = \frac{2TP}{2TP+FP+FN} = \frac{83 \times 2}{83 \times 2+24+6} = \frac{166}{249} = 0.8469 \approx 83\%$$

Visualization

Sanddance

Let us talk of threshold , means the percentage of trees giving decision in favour of fraud , for a forest so that we conclude that it gives us more accurate measures such as f-measure, usually threshold is equal to 0.5 that mean if there are n tree in a forest and if more than n/2 tree gives decision for a transaction to be fraudulent we predict that transaction as fraudulent but in the case of unbalance data it is no more 0.5 so we are gonna analyze that using sanddance



CHART

Grid ( ) Bar ( )
Scatter ( ) Treemap ( )
Density ( ) Stacks ( )
[✓ Column]

COLUMN MAPPING

X Axis: Tree

X axis bin size 20

Z Axis: -- none --

Color by: Fmeasure

Sort by: -- none --

Facet by: Threshold

Tooltip columns...

Fmeasure
🟥 < 0.414124
🟧 0.414124–0.708571
🟨 0.708571–0.793478
🟨 0.793478–0.827225
🟩 0.827225–0.85
🟩 0.85–0.867347
🟩 ≥ 0.867347

We can look at the output of sanddance , in this every facet is a threshold , and x-axis is a number of trees and y-axis represents sample and the color represent f-measure we can see its threshold is more greener at threshold 0.3-0.4 , where as the number of tree is increasing the f-measure is also improving

ggplot2

Now let us use R , first of all import data

```r
measures=read.csv("measures.csv")
```

Now let us add a column called coverage that is multiplication of the columns Sample and tree

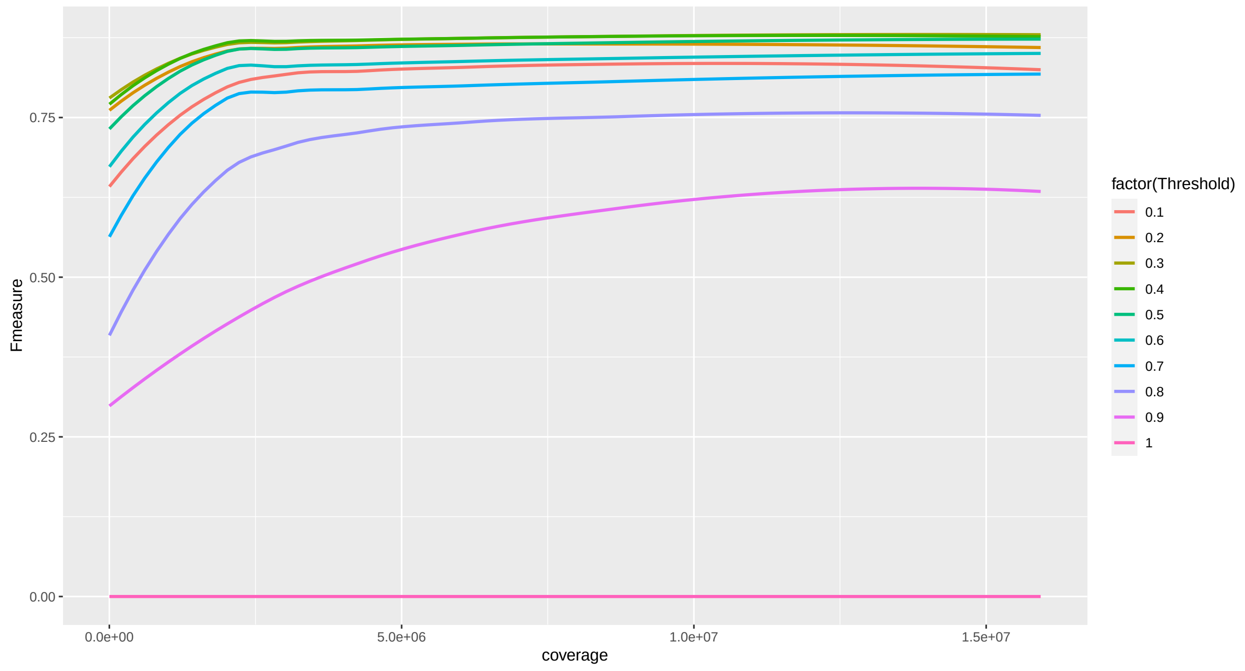
```r
coverage=(measures[,1]*measures[,2]);
measures=add_column(measures,coverage,.after=2);
knitr::kable(head(measures,25),"latex",booktabs=T)
```

Sample	Tree	coverage	Threshold	Precission	Recall	Accuracy	Fmeasure
1322	1	1322	0.1	0.8602151	0.7142857	0.9991462	0.7804878
1322	1	1322	0.2	0.8602151	0.7142857	0.9988626	0.7804878
1322	1	1322	0.3	0.8571429	0.7090909	0.9987720	0.7761194
1322	1	1322	0.4	0.8571429	0.7155963	0.9989402	0.7800000
1322	1	1322	0.5	0.8586957	0.7117117	0.9987673	0.7783251
1322	1	1322	0.6	0.8571429	0.7090909	0.9987719	0.7761194
1322	1	1322	0.7	0.8505747	0.6981132	0.9987625	0.7668394
1322	1	1322	0.8	0.8488372	0.7019231	0.9988799	0.7684211
1322	1	1322	0.9	0.8522727	0.7009346	0.9987623	0.7692308
1322	1	1322	1.0	0.0000000	0.0000000	0.9971050	0.0000000
1322	2	2644	0.1	0.7920792	0.7142857	0.9985537	0.7511737
1322	2	2644	0.2	0.7789474	0.6981132	0.9985418	0.7363184
1322	2	2644	0.3	0.7857143	0.7129630	0.9985852	0.7475728
1322	2	2644	0.4	0.7920792	0.7142857	0.9985598	0.7511737
1322	2	2644	0.5	0.6842105	0.1287129	0.9974760	0.2166667
1322	2	2644	0.6	0.6842105	0.1300000	0.9974512	0.2184874
1322	2	2644	0.7	0.6842105	0.1203704	0.9979060	0.2047244
1322	2	2644	0.8	0.6842105	0.1192661	0.9976007	0.2031250
1322	2	2644	0.9	0.6842105	0.1171171	0.9971412	0.2000000
1322	2	2644	1.0	0.0000000	0.0000000	0.9969011	0.0000000
1322	3	3966	0.1	0.3153846	0.7454545	0.9943343	0.4432432
1322	3	3966	0.2	0.3181818	0.7500000	0.9942794	0.4468085
1322	3	3966	0.3	0.3193916	0.7500000	0.9937681	0.4480000
1322	3	3966	0.4	0.8941176	0.6846847	0.9990453	0.7755102
1322	3	3966	0.5	0.8953488	0.6875000	0.9990017	0.7777778

Now we need to plot coverage vs. Fmeasure , and for smoothing going to use **lowess**, with grouping according to the threshold so we can get curve for every threshold

```r
ggplot(measures,aes(x=coverage,y=Fmeasure,group=Threshold,
                    colour=factor(Threshold)))+geom_smooth(se=0,method="loess")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



We can draw four conclusions from it

1. Threshold for imbalanced dataset is not 0.5, as we can see Fmeasure is highest for 0.3 , whereas for 0.5 performance is not that good
2. At certain coverage there is no much benefit in increasing coverage , that is product of trees and sample size , means after certain level the performance will become constant as we can see for any threshold after certain coverage the slope starts to becoming parallel to x-axis
3. Performance here F-measure is proportional to coverage i.e

$$Performance \propto Coverage$$
$$Performance \propto Number\ of\ trees \times Sample\ Size$$

Intuition We can see in the graph the curve somewhat looks like log curve as

$$Performance = log(Coverage) + f(threshold)$$

Where it looks like $f(threshold)$ is somehow maximum at 0.3 that means

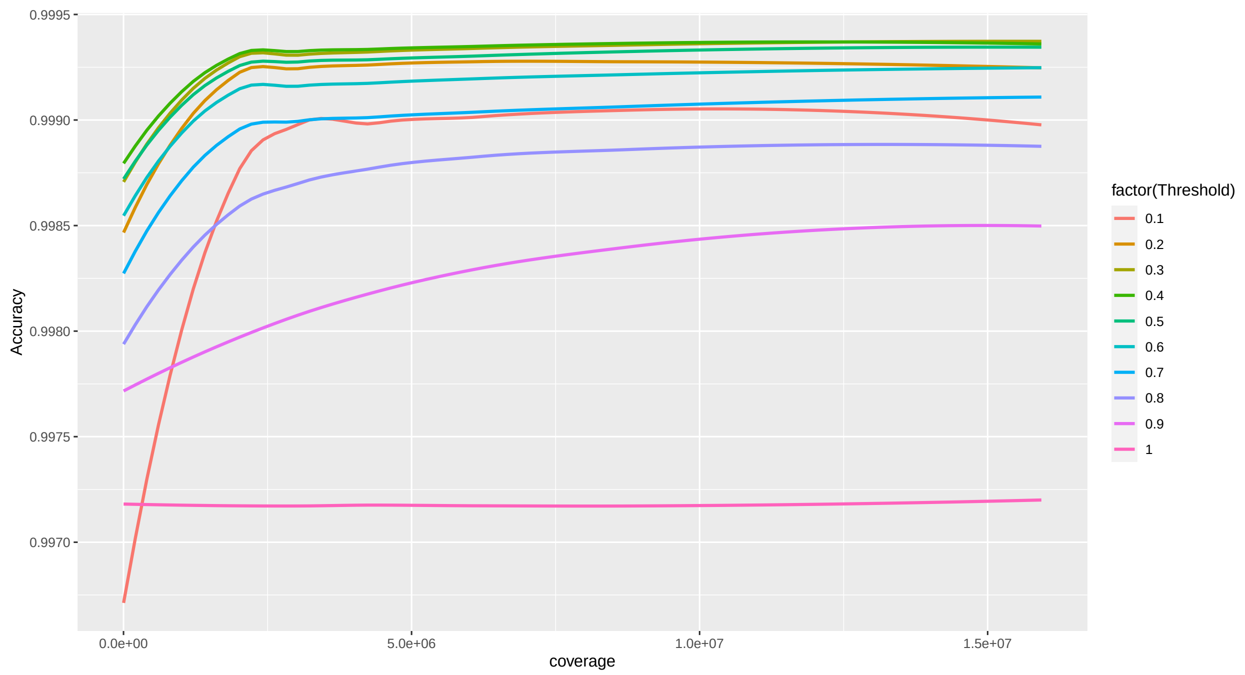
$$\frac{d}{dx}f(threshold)|_{threshold=0.3} \approx 0$$

However it needs further investigation , to conclude something because we have predicted on threshold 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1 only we need high end computing devices to conclude further so calculation can be done more precisely here we have take error of 0.1 which is quite big.

Let us check Accuracy Paradox

```r
ggplot(measures,aes(x=coverage,y=Accuracy,group=Threshold,
                    colour=factor(Threshold)))+geom_smooth(se=0,method="loess")
```

```
## `geom_smooth()` using formula 'y ~ x'
```

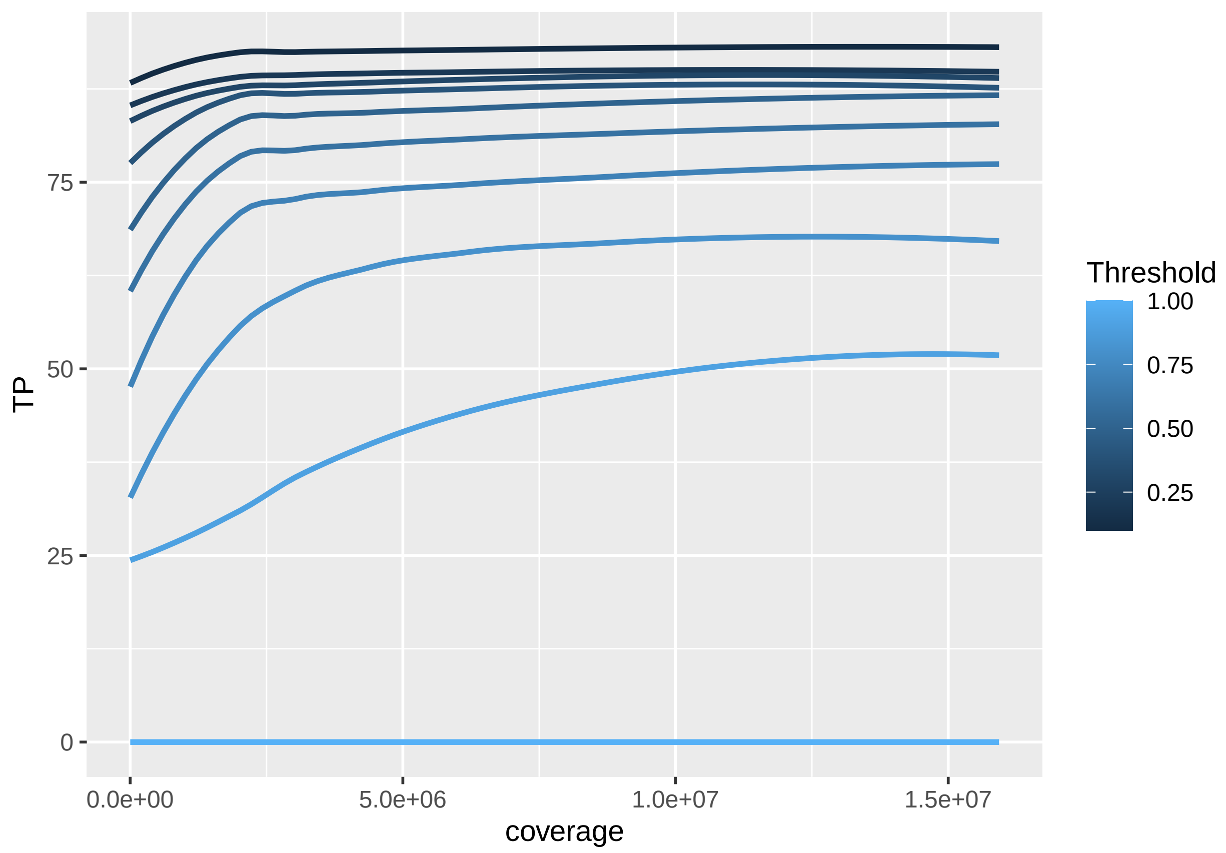


We can conclude from here that accuracy is not a good measure fo imbalance data , as we can see accuracy is higher in most of the cases whether it is not true

Now during fraud detection our main aim is to insure there must not be any transaction which is actually fraud not detected by our model however this is to insure that true positive is equal to the total positive , while reducing false negative , let us take a look of True positive vs Coverage, first of all we will import or `jungle_ind` in R , then use ggplot

```r
ind=read.csv("indicators.csv")
coverage=(ind[,1]*ind[,2]);
ind=add_column(ind,coverage,.after=2);
ggplot(ind,aes(x=coverage,y=TP,group=Threshold,
                    ))+geom_smooth(se=0,method="loess",aes(color=Threshold))
```

```
## `geom_smooth()` using formula 'y ~ x'
```



We can see the True Positive maximum for threshold 0 , means it can successfully classify each and ever fraudulent transaction.

Conclusion & Result

1. Fmeasure is highest if we classify a transaction fraudulent when 30 to 40 percent of the trees , which is quite different for regular random forest model
2. Number of trees , and sample have effect on the performance of forest but after certain limit there is not any benefit in increasing any of them
3. When Performance is constant Number of tree is inversely proportional to Sample size
4. Accuracy is not good metric for imbalance data