Decision Trees and Random Forests

Before starting to build a decision trees, lets first look at what it looks like and how to use it for decision making.

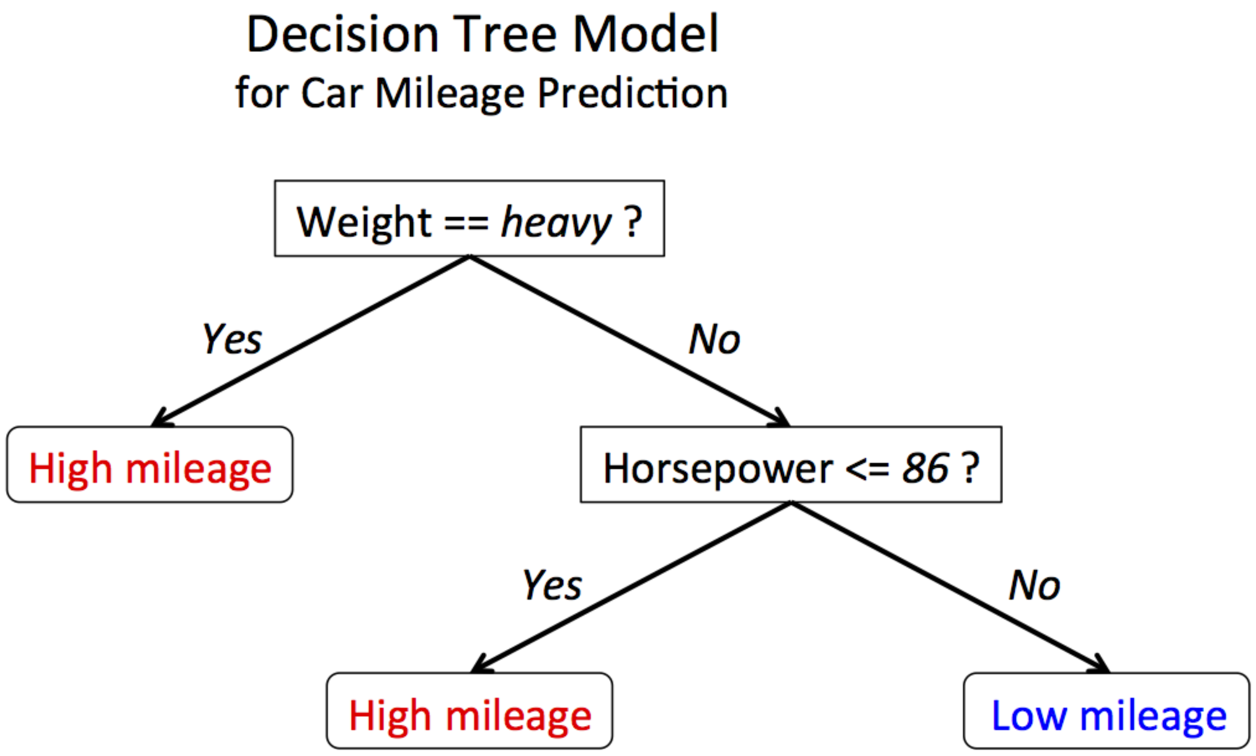


Figure 1: dtree

You can start with a car and sequentially answer those questions as given in the decision tree and you'll arrive at the decision that is whether that car will have high mileage or not. Now you must be wondering how to build this decision tree. We'll reach there eventually, but first consider , what happens if instead of just one car, you start with many and start following the question trail of the decision tree.

When you answer the first question on top, it divides the group into two groups depending on the individual's answers. These groups get broken into sub groups as they go down their subsequent question. Eventually smaller sub groups arrive at decision nodes labelled as "high" or "low". Those entire groups get assigned those decisions.

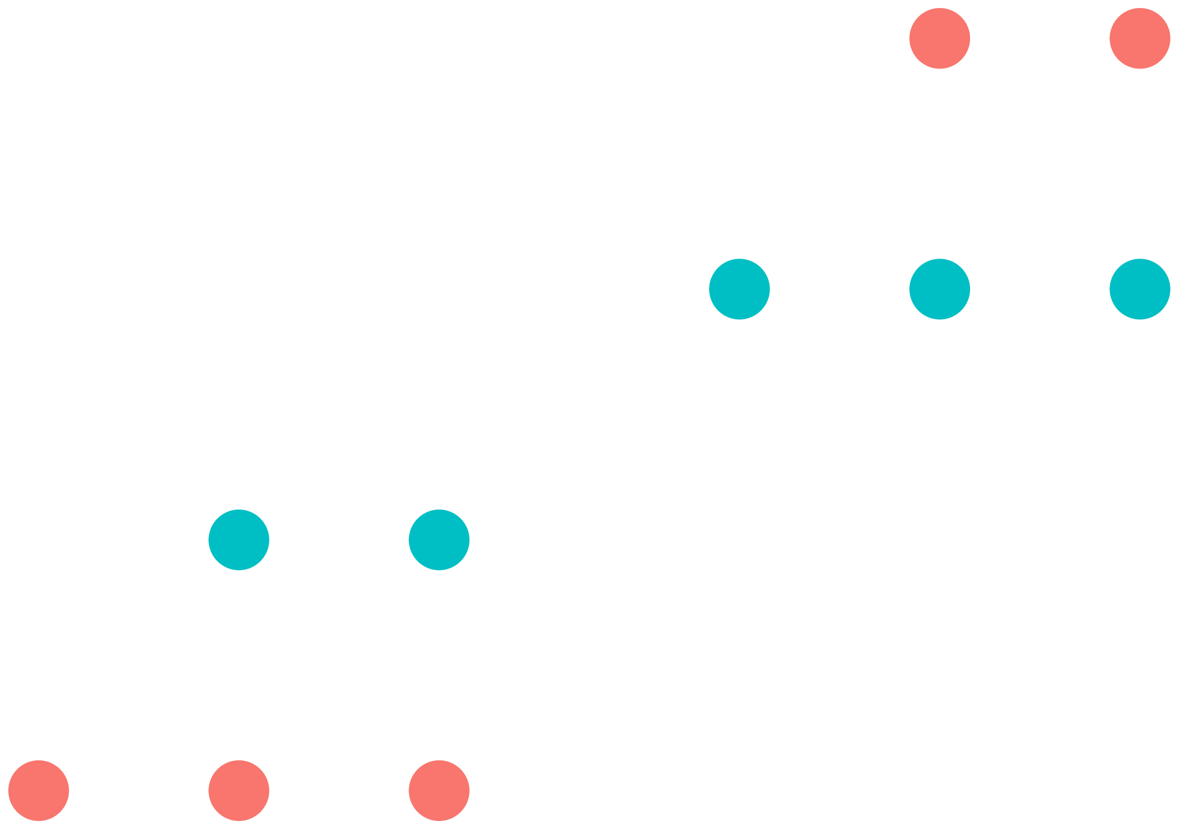
Cleary when the crowd was not divided in these sub groups it was difficult to take an umbrella decision. These question trails given by the decision tree breaks our group into more homogenious sub groups which makes it easier to take umbrella decisions for these smaller [and homogenious] subgroups.

To build a decision tree , we do the same thing. We take a group [entire data] and try to figure how do we split the data so that it makes those subsets of the data more homogenious than before. Now we need to figure out how do we define this homogeneity.

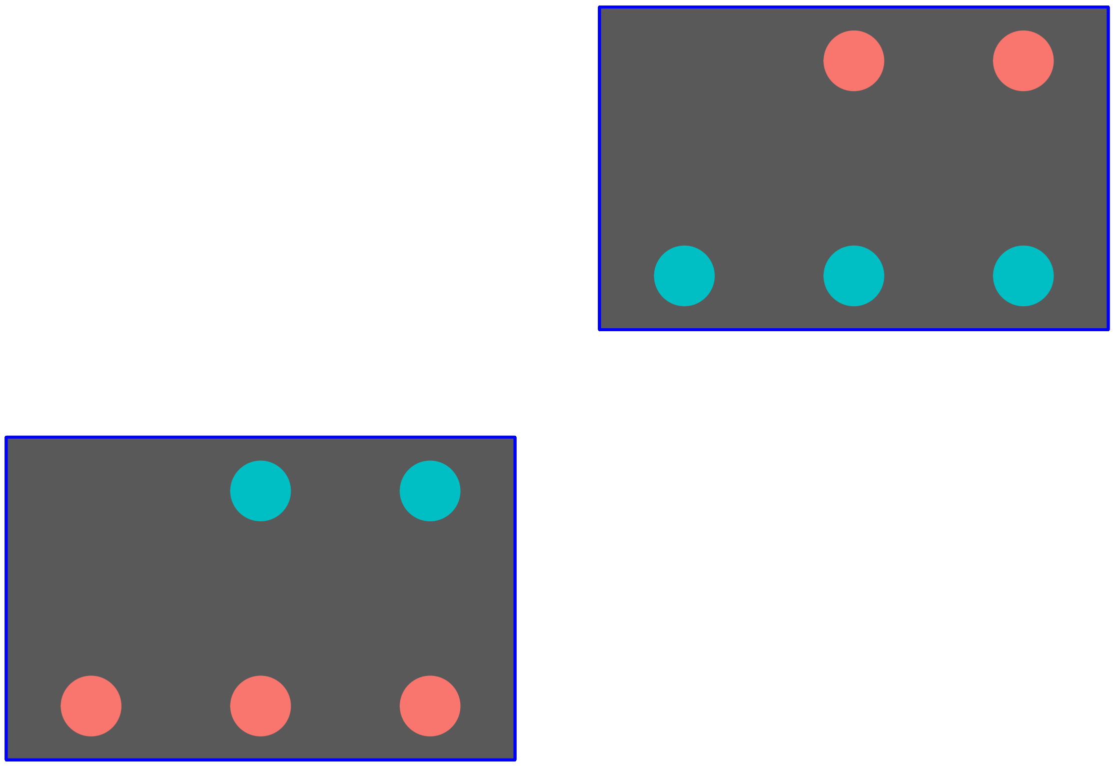
Consider this groups of 10 balls of two different colors in bag. Consider these balls to be observations where color of the ball is final outcome and other characteristics like weight, roughness, density etc are predcitor variables.

If you were asked to take an umbrella call, assign all of them just one color/outcome, you'd be wrong 50% of the time.

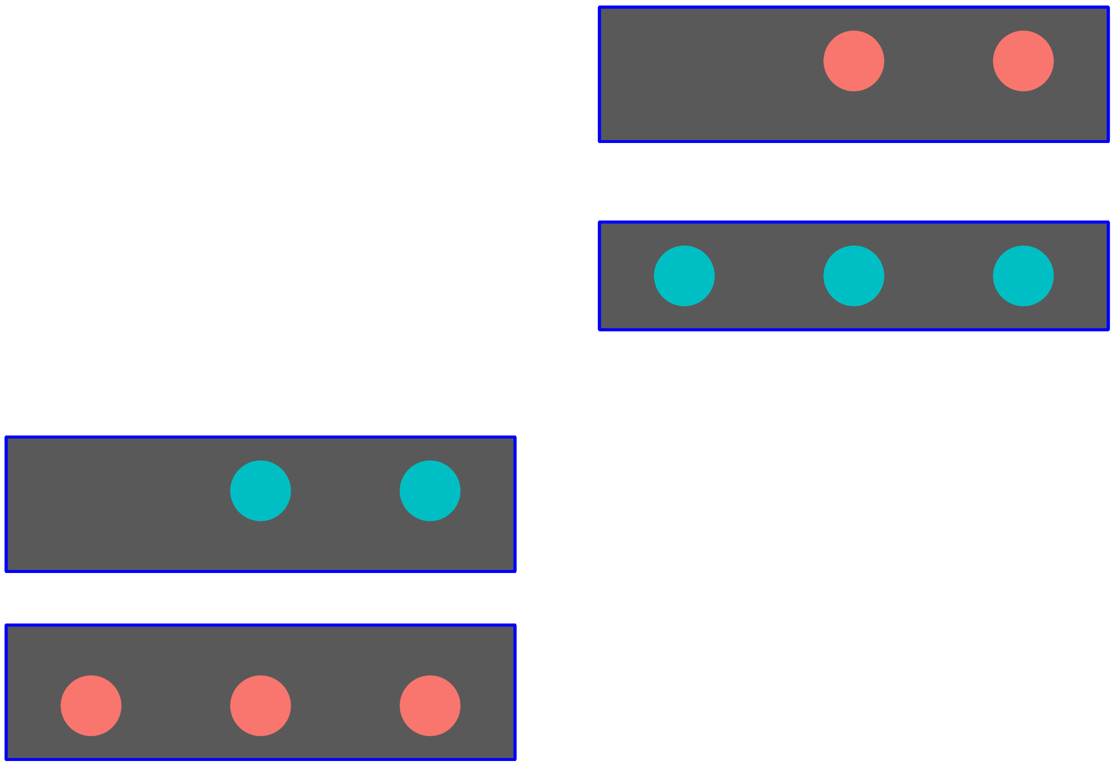
```
## Warning: package 'ggplot2' was built under R version 3.3.2
```



Now lets say i came up with a rule [remember questions in the decision tree?!], If the ball is rough it goes in one bag and if it is smooth it goes in another bag. Basically now i'm dividing this crowd/data into two groups. Now in these two groups if I take the majority color as my umbrella call, I'll be 60% of the time correct. Now thats an improvement.



Now lets say I have been able to separate these smaller groups even further by putting in subsequent questions based on their density and weights . Which further divides the existing groups into smaller , more homogenious groups.



And now when I take the majority as the final decision outcome , I'll be 100% correct!. Lets recap what happend here. We start with a hetrogenous data and split that data to make it more homogeneous based on some cascading rules.

Although very close to how decision trees are built, this still was a hypothetical scenario. We did not discuss how did we come up with those “rules/questions”? How do we measure homegeneity? Next we answer these questions.

Before we take our discussion forward , lets get familiar with a little terminology. The top point where you have all your data undivided is called a parent node. subsequent partitions of your data are called nodes. Node which doesnt have any children node are called terminal nodes or leaves. Terminal nodes are the ones where final decision is taken. Merging a node back with its parent is called pruning a tree. Number of terminal node is called size of a tree.

You can always revisit the paragraph above , in case some terminology doesnt make sense to you.

Now back to our discussion on answer those two questions. Lets say there are k classes in our data. [outcome has k classes which we want to build a prediction model for using predictor variables]. There are three very famous [and most used] measures for homogeneity of a node.

Entropy : $-\sum_{i=1}^{k} p_i * log(p_i)$

Gini index : $1 \ - \ \sum_{i=1}^{k} p_i^2$

Deviance : $-2\sum_{i=1}^{k} n_i log(p_i)$

Here k is number of classes and p_i is the proportion of the i^{th} class in that node.In deviance formula n_i is number of observation belonging to i^{th} class in that node.

To calculate Gini index/Entropy of an entire tree, individual gini indices/entropies can be added by multiplying fractional weightage of their respective nodes. [fractional weightage of a node = (number of obs in the the node)/(total observations)]. Total deviance of a tree can be calculated by simply adding individual deviances of its nodes.

How to make and choose Rules

Making a rule is fairly simple. Rules come from different values of predictor variables and include only one variable at a time. If a categorical variable , lets say take value “a” , “b”, “c”, it gives rise to 3 rules.

R1 : value of the categorical variable is “a” or not

R2 : value of the categorical variable is “b” or not

R3 : value of the categorical variable is “c” or not

If we are considering a numeric variable, we can break it into intervals.Lets say we have a numeric variable which takes values in the range 1-10. We can divide this in 9 break points [intervals] and construct these rules:

R1:value of the variable is >1

R2:value of the variable is >2

.

.

R9:value of the variable is >9

All these rules have binary answers. Depending upon the answer a single observation will go to left or right node and keep on travelling until it reaches to terminal node where it gets assigned to a final class. At each partition we consider all these rules, how do we pick one? You can check whichever results in the most decrease in the gini index [or deviance] of the overall tree is selected at that partition.

Deviance is used as a measure in the base tree package. Gini index is used in the packagae `rpart` and `randomForest`. It doesnt matter much which implementation uses what criterion, at the end with enough data, results do not differ dramatically.

What if our target is not categorical. Meaning what if classification tree is not what we can use. Surprisingly regression trees are not very different from the classification trees. Rules still remain as before, just the method of selecting rules [measuring error] becomes different.

Think about it this way that without any partition in the data your best bet at prediction [for a continuous numeric target] is average. Here the measure of error becomes total sum of squares. As you partition your data this goes down. [Deviance in the case of regression trees is simply error sum of squares]. For an individual node [i^{th} node] it is defined as. And ofcourse the prediction is average of our response variable on the respective terminal node. :

$$SSE \ = \ \sum_{j=1}^{n_i} (\bar{y_i} - y_j)^2$$

SSE for a tree can be measured by summing up SSE for all terminal nodes.

Lets solve our interest rate prediction problem using a decision tree and see the results. Data Prep will not be different for a different algorithm , we'll take the code directly from that module.

```
ld_train=read.csv("~/Dropbox/0.0 Data/loan_data_train.csv",stringsAsFactors = F)

ld_test= read.csv("~/Dropbox/0.0 Data/loan_data_test.csv",stringsAsFactors = F)

ld_test$Interest.Rate=NA

ld_train$data='train'
ld_test$data='test'

ld_all=rbind(ld_train,ld_test)

# drop amount funded by investor

ld_all$Amount.Funded.By.Investors=NULL

library(dplyr)

ld_all=ld_all %>%
  mutate(Interest.Rate=as.numeric(gsub("%","",Interest.Rate)) ,
         Debt.To.Income.Ratio=as.numeric(gsub("%","",Debt.To.Income.Ratio)) ,
         Open.CREDIT.Lines=as.numeric(Open.CREDIT.Lines) ,
         Amount.Requested=as.numeric(Amount.Requested) ,
         Revolving.CREDIT.Balance=as.numeric(Revolving.CREDIT.Balance)
         )

ld_all=ld_all %>%
  mutate(ll_36=as.numeric(Loan.Length=="36 months")) %>%
  select(-Loan.Length)

ld_all=ld_all %>%
  mutate(lp_10=as.numeric(Loan.Purpose=='educational'),
         lp_11=as.numeric(Loan.Purpose %in% c("major_purchase","medical","car")),
         lp_12=as.numeric(Loan.Purpose %in% c("vacation","wedding","home_improvement")),
         lp_13=as.numeric(Loan.Purpose %in% c("other","small_business","credit_card")),
         lp_14=as.numeric(Loan.Purpose %in% c("debt_consolidation","house","moving"))) %>%
  select(-Loan.Purpose)

CreateDummies=function(data,var,freq_cutoff=0){
  t=table(data[,var])
  t=t[t>freq_cutoff]
  t=sort(t)
  categories=names(t)[-1]

  for( cat in categories){
    name=paste(var,cat,sep="_")
    name=gsub(" ","",name)
    name=gsub("-","_",name)
    name=gsub("\\?","Q",name)
    name=gsub("<","LT_",name)
    name=gsub("\\+","",name)
    name=gsub("\\/","_",name)
    name=gsub(">","GT_",name)
    name=gsub("=","EQ_",name)
    name=gsub(",","",name)

    data[,name]=as.numeric(data[,var]==cat)
  }

  data[,var]=NULL
  return(data)
}

ld_all=CreateDummies(ld_all ,"State",100)
ld_all=CreateDummies(ld_all,"Home.Ownership",100)

library(tidyr)

ld_all=ld_all %>%
  separate(FICO.Range,into=c("f1","f2"),sep="-") %>%
  mutate(f1=as.numeric(f1),
         f2=as.numeric(f2),
         fico=0.5*(f1+f2)) %>%
  select(-f1,-f2)

ld_all=CreateDummies(ld_all,"Employment.Length",100)

ld_all=ld_all[!(is.na(ld_all$ID)),]

for(col in names(ld_all)){

  if(sum(is.na(ld_all[,col]))>0 & !(col %in% c("data","Interest.Rate"))){

    ld_all[is.na(ld_all[,col]),col]=mean(ld_all[ld_all$data=='train',col],na.rm=T)
  }
}

## separate train and test

ld_train=ld_all %>% filter(data=='train') %>% select(-data)
ld_test=ld_all %>% filter(data=='test') %>% select(-data,-Interest.Rate)

##

set.seed(2)
s=sample(1:nrow(ld_train),0.7*nrow(ld_train))
ld_train1=ld_train[s,]
ld_train2=ld_train[-s,]
```

To build a decision tree, we can simply use function tree.

```
library(tree)
ld.tree=tree(Interest.Rate~.-ID,data=ld_train1)
```

you can simply look at the result by typing the model object

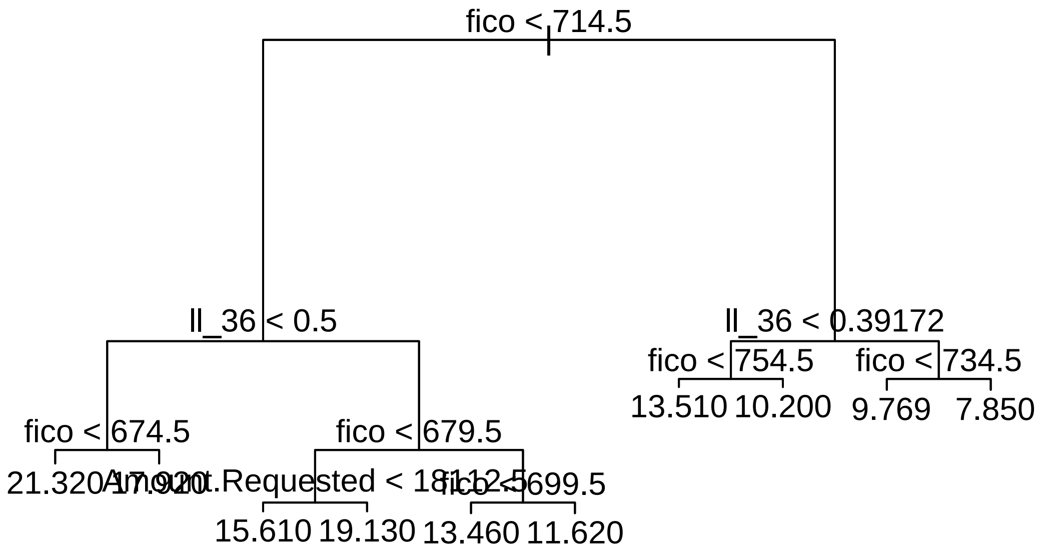
```
ld.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 1539 27260.0 13.020
##    2) fico < 714.5 986 11840.0 15.000
##      4) ll_36 < 0.5 205  1850.0 18.880
##        8) fico < 674.5 58   302.2 21.320 *
##        9) fico > 674.5 147  1065.0 17.920 *
##      5) ll_36 > 0.5 781  6098.0 13.990
##       10) fico < 679.5 302  1800.0 15.940
##         20) Amount.Requested < 18112.5 274  1295.0 15.610 *
##         21) Amount.Requested > 18112.5 28   191.6 19.130 *
##       11) fico > 679.5 479  2420.0 12.750
##         22) fico < 699.5 296  1147.0 13.460 *
##         23) fico > 699.5 183   892.8 11.620 *
##    3) fico > 714.5 553  4651.0  9.490
##      6) ll_36 < 0.39172 121   924.8 12.440
##       12) fico < 754.5 82   468.2 13.510 *
##       13) fico > 754.5 39   168.3 10.200 *
##      7) ll_36 > 0.39172 432  2376.0  8.663
##       14) fico < 734.5 183   899.4  9.769 *
##       15) fico > 734.5 249  1089.0  7.850 *
```

Here each indent represents a splitting node and a line with * at the end represents terminal node. The values at each line are as follows : split rule, number of obs in that node , deviance for the node and average response in that node [because its a regression tree]

Using simple plot function, you can visually also see how the tree is

```
plot(ld.tree)
text(ld.tree)
```



top of the node is rule, observations go to left if the rule is true for them otherwise they go to right. Terminal nodes have average response displayed at the end.

To get prediction , we can again use the good old predict function

```
val.IR=predict(ld.tree,newdata = ld_train2)
```

Performance of tree model :

```
rmse_val=((val.IR)-(ld_train2$Interest.Rate))^2 %>% mean() %>% sqrt()
rmse_val
```

```
## [1] 2.279284
```

So now we know what will be the approximate performance of the decision tree. We can now build the model on entire training data and use that to eventually make prediction for test.

```
ld.tree.final=tree(Interest.Rate~.-ID,data=ld_train)
test.pred=predict(ld.tree.final,newdata=ld_test)
write.csv(test.pred,"mysubmission.csv",row.names = F)
```

Lets see how a decision tree performs on the classfication problem that we handled in the last module . Again the data prep part will not be different.

```
rg_train=read.csv("~/Dropbox/0.0 Data/rg_train.csv",stringsAsFactors = FALSE)
rg_test=read.csv("~/Dropbox/0.0 Data/rg_test.csv",stringsAsFactors = FALSE)

rg_test$Revenue.Grid=NA

rg_train$data='train'
rg_test$data='test'

rg=rbind(rg_train,rg_test)

rg = rg %>%
  mutate(children=ifelse(children=="Zero",0,substr(children,1,1)),
         children=as.numeric(children))

rg=rg %>%
mutate(a1=as.numeric(substr(age_band,1,2)),
       a2=as.numeric(substr(age_band,4,5)),
      age=ifelse(substr(age_band,1,2)=="71",71,ifelse(age_band=="Unknown",NA,0.5*(a1+a2)))
       ) %>%
  select(-a1,-a2,-age_band)
```

```
## Warning in eval(substitute(expr), envir, enclos): NAs introduced by
## coercion

## Warning in eval(substitute(expr), envir, enclos): NAs introduced by
## coercion
```

```
cat_cols=c("status","occupation","occupation_partner","home_status",
           "family_income","self_employed",
           "self_employed_partner","TVarea","gender","region")

for(cat in cat_cols){
  rg=CreateDummies(rg,cat,50)
}

rg=rg %>%
  select(-post_code,-post_area)

rg$Revenue.Grid=as.numeric(rg$Revenue.Grid==1)

for(col in names(rg)){

  if(sum(is.na(rg[,col]))>0 & !(col %in% c("data","Revenue.Grid"))){

    rg[is.na(rg[,col]),col]=mean(rg[rg$data=='train',col],na.rm=T)
  }

}

rg$Revenue.Grid=as.numeric(rg$Revenue.Grid==1)
```

One very important thing that you need to notice and remember here is that , tree function can be used for both regression and classfication problems . The way it differentiates or figures out whether it needs to build a classfication or regression tree is by looking at data type of response. If its numeric it carries out a regression modeling process. And if its factor type it builds a classfication model. So remember to convert your response to factor when you are working with a classification project.

```
rg$Revenue.Grid=as.factor(rg$Revenue.Grid)
```

```
for(col in names(rg)){

  if(sum(is.na(rg[,col]))>0 & !(col %in% c("data","Revenue.Grid"))){

    rg[is.na(rg[,col]),col]=mean(rg[rg$data=='train',col],na.rm=T)
  }

}

rg_train=rg %>% filter(data=='train') %>% select(-data)
rg_test=rg %>% filter(data=='test') %>% select (-data,-Revenue.Grid)

set.seed(2)
s=sample(1:nrow(rg_train),0.8*nrow(rg_train))
```

```
rg_train1=rg_train[s,]
rg_train2=rg_train[-s,]
```

Done with data prep, lets build the model now

```
rg.tree=tree(Revenue.Grid~.-REF_NO-Investment.in.Commudity
             -Investment.in.Derivative-Investment.in.Equity
             -region_SouthEast-TVarea_Central-occupation_Professional
             -family_income_GT_EQ_35000-region_Scotland
             -Portfolio.Balance,data=rg_train1)
```

Notice that dropping variables based on vif is fine (we can be mroe relaxed on the cutoff though). High vif values essentially mean redundancy of information which is useless irrespective of what algorithm you eventually decide to select for final model.

Lets see how it performed on the validation set. Notice the difference how we get probability prediction from a tree model. By default it gives probability for both the classes , we only need one , thats why the square bracket at the end for subsetting.

```
val.score=predict(rg.tree,newdata = rg_train2,type='vector')[,1]
pROC::roc(rg_train2$Revenue.Grid,val.score)$auc
```

```
## Area under the curve: 0.9402
```

Now we know the probable performance , we can go ahead and build the model on entire training data for eventual submission.

```
rg.tree.final=tree(Revenue.Grid~.-REF_NO-Investment.in.Commudity
                   -Investment.in.Derivative-Investment.in.Equity
                   -region_SouthEast-TVarea_Central-occupation_Professional
                   -family_income_GT_EQ_35000-region_Scotland-Portfolio.Balance,
                   data=rg_train)
```

If we needed to submit simple probability scores then we can do this

```
test.score=predict(rg.tree.final,newdata=rg_test,type='vector')[,1]
```

and write this to a csv for submission if needed.

```
write.csv(test.score,"mysubmission.csv",row.names = F)
```

However if we needed to submit hard classes, we'll need to determine cutoff on the probability scores in the usual way that we did in logistic regression module. I'll repeat that just once here in the module, you can follow the same process [finding cutoff on probability score], whenever you need to submit/predict hard classes in place of simple probability score.

```
train.score=predict(rg.tree.final,newdata=rg_train,type='vector')[,1]
real=rg_train$Revenue.Grid

cutoffs=seq(0.001,0.999,0.001)

cutoff_data=data.frame(cutoff=99,Sn=99,Sp=99,KS=99,F5=99,F.1=99,M=99)

for(cutoff in cutoffs){

  predicted=as.numeric(train.score>cutoff)

  TP=sum(real==1 & predicted==1)
  TN=sum(real==0 & predicted==0)
  FP=sum(real==0 & predicted==1)
  FN=sum(real==1 & predicted==0)
  P=TP+FN
  N=TN+FP

  Sn=TP/P
  Sp=TN/N
  precision=TP/(TP+FP)
  recall=Sn

  KS=(TP/P)-(FP/N)
  F5=(26*precision*recall)/((25*precision)+recall)
  F.1=(1.01*precision*recall)/((.01*precision)+recall)

  M=(4*FP+FN)/(5*(P+N))

  cutoff_data=rbind(cutoff_data,c(cutoff,Sn,Sp,KS,F5,F.1,M))
}

cutoff_data=cutoff_data[-1,]

my_cutoff=cutoff_data$cutoff[which.max(cutoff_data$KS)]

my_cutoff
```

```
## [1] 0.001
```

now that you have your cutoff, you can make hard predictions

```
test.predicted=as.numeric(test.score>my_cutoff)
write.csv(test.predicted,"proper_submission_file_name.csv",row.names = F)
```

Random Forest

Decision tree are very good at capturing non linear patterns in the data. Thats a good thing about them and a bad thing about them as well. Good because well, as I said earlier , they can capture non-linear patterns very well. Bad becasue, this capability makes them succeptible to capturing very niche patterns from the training data which might not generalise very well. this is called overfitting or model conforming to noise in the data.

A very simple yet powerful idea of introducing randomness in the process takes care of this problem. Name of the algorithm that we are going to discuss is RandomForest, it works using the fact that noise is a smaller portion of the data. If we randomly subset our data and use that to build our tree instead , it is highly likely that it will not be affected by noise. Lets say hypothetically 85% of the time it will not be affected by noise and 15% of the time it will be. To counter that 15% effect, we can build many such trees, each one being built on a different random subset of the data; and then take the average/majority vote to make prediction.

RandomForest uses two sets of randomness to coutner the effect of noise

1. Each tree is built on a random subset of observations. This averages out effect of noisy observations.
2. For each of these tree, at each splitting node, isntead of all variables in the data being used , only a subset of variables are considered to select the splitting rules from. This averages out effect of noisy variables.

We'll see that R's implementation of RandomForest has 4 parameters.

- **mtry :** Number of variables randomly subsetted at each node to select the splitting rule from. This value should be integer, greater than 1 and less than or equal to number of predictor variables in the data. Default value is p/3 for regression and $\sqrt{p}$ for classfication problem where p is the number of predictor variables in the data.

- **ntree :** This is number of trees in the forest. There is no limit on it as such , a good starting point is 50,100 and you can try out values as large as 1000,5000. Although very high number of trees make sense when the data is huge as well. Default value is 500.
- **nodesize :** This is minimum size of terminal nodes. This essentially stops the hairsplitting of the data, meaning a forced split will not be considered if the node resulting from the split is too small. This generally stems from some niche patterns in the data which do not generalise very well. Again there is no limit on this as such but good range to try can be between 1 to 20. Default value is 1.
- **maxnodes :** This controls size of the tree, it is max number of terminal nodes. Larger you make a tree, more overfitting might happen. Again there is no limit on it as such but a good starting point can be 5 and higher could be anything, although i have rarely seen people going above 100. Default is set to NULL meaning no limits and size is controlled by other factors and data patterns themselves.

You can see that there are many possible values for these paramters. Lets see if we want to try 2 different values for each of these parameters .

- mtry : 5, 10
- ntree : 100, 500
- maxnodes : 15, 20
- nodesize : 2, 5

This leads to $2^4 = 16$ possible combinations of the paramteres.

```
##    mtry ntree maxnodes nodesize
## 1     5   100       15        2
## 2    10   100       15        2
## 3     5   500       15        2
## 4    10   500       15        2
## 5     5   100       20        2
## 6    10   100       20        2
## 7     5   500       20        2
## 8    10   500       20        2
## 9     5   100       15        5
## 10   10   100       15        5
## 11    5   500       15        5
## 12   10   500       15        5
## 13    5   100       20        5
## 14   10   100       20        5
## 15    5   500       20        5
## 16   10   500       20        5
```

We can try using all these combinations and try the resulting models performance on validation and chose the combination which performs best. Instead of doing this manually, we can make use of the package `cvTools` which has functions for paramter selection (also called paramter tuning). It makes use of cross validation, so we dont even need to break our data to get a validation set.

We'll start with the regression problem that we took earlier and set the parameter values that we want to try.

```
param=list(mtry=c(5,10,15,20,25),
           ntree=c(50,100,200,500,700),
           maxnodes=c(5,10,15,20,30,50),
           nodesize=c(1,2,5,10))
```

This leads to 5X4X6X4=480 possible combinations, we should technically CV performance of all these combination to find which one is the best. But that might be an overkill in terms of how much time it might take. If we are looking to do a 10 fold cross validation it will lead to 480X10 = 4800 randomforest models being built. If on an average each randomforest had 100 Trees, we are looking at 480,000 decision trees being built internally. This is beyond resource consuming. At the same time you need to realise that we dont really need to try out all the possible combination. For e.g. [mtry=10,ntree=100,maxnodes=30,nodesize=1] might not be very different from [mtry=10,ntree=100,maxnodes=50,nodesize=2]. Instead of trying out all possible combinations, we can randomly select a much smaller subset to try and get a good enough if not the best combination at a fractional cost in terms of time and resources. Lets write a function which selects a random subset of combination.

```
subset_paras=function(full_list_para,n=10){

  all_comb=expand.grid(full_list_para)

  s=sample(1:nrow(all_comb),n)

  subset_para=all_comb[s,]

  return(subset_para)
}
```

If we pass the list of parameter values (`full_list_para`) that we want to try and specify number of combinations (`n`) we want to try; it randomly selects those many combinations out of all possible parameter combinations and returns them in form of a dataframe.

```
num_trials=50
my_params=subset_paras(param,num_trials)
# Note: A good value for num_trials is around 10-20% of total possible
# combination. It doesnt have to be always 50
```

We'll be using `cvTuning` function from package cvTools to try out all these paramter combinations one by one. We'll compare the cv error measure of all these and pick that as the best combination which results in lowest error [becasue we are working with a regression problem, for classification we'd be looking at highest auc score (for binary classification) or highest accuracy (for more than one class classification)]

we'll start with very high value of error and update in whenever we get a smaller error figure. Every time we find a parameter combiantion which has lowest error so far, it will be printed in the output. Last printed value of paramters will be our final choice of parameters.

```
library(randomForest)
library(cvTools)
```

```
myerror=9999999

for(i in 1:num_trials){
  # print(paste0('starting iteration:',i))
  # uncomment the line above to keep track of progress
  params=my_params[i,]

  k=cvTuning(randomForest,Interest.Rate~.-ID,
             data =ld_train,
             tuning =params,
             folds = cvFolds(nrow(ld_train), K=10, type = "random"),
             seed =2
             )
  score.this=k$cv[,2]

  if(score.this<myerror){
    # print(params)
    # uncomment the line above to keep track of progress
    myerror=score.this
    # print(myerror)
    # uncomment the line above to keep track of progress
    best_params=params
  }

  # print('DONE')
  # uncomment the line above to keep track of progress
}
```

If you want to know the tentative performance measure , that'll be latest value of myerror

```
myerror
```

```
## [1] 1.870957
```

```
best_params
```

```
##   mtry ntree maxnodes nodesize
## 1   20   200       50       10
```

now we have best values of paramteres from cross validation. We'll use these values to build our RandomForest model on entire training data and use that for prediction on test.

```
ld.rf.final=randomForest(Interest.Rate~.-ID,
                         mtry=best_params$mtry,
                         ntree=best_params$ntree,
                         maxnodes=best_params$maxnodes,
                         nodesize=best_params$nodesize,
                         data=ld_train)
```

```
test.pred=predict(ld.rf.model,newdata = ld_test)
write.csv(test.pred,"mysubmission.csv",row.names = F)
```

Before we go and start with our example on classification problem, we need to discuss few things about the randomforest model output which we need to be aware of.

OOB Error

As we know that each individual tree makes use of only subset of observations, randomforest internally makes predictions using that tree for the remaining observations, thus getting out of sample errors for them. These are averaged over all the trees and we end up getting out of sample errors for all the obs . This is called Out Of Bag (OOB) errors and can be used as tentative performance measure as these come from out of sample predictions.

```
ld.rf.final
```

```
##
## Call:
##  randomForest(formula = Interest.Rate ~ . - ID, data = ld_train,      mtry = best_params$mtry, ntree
##                Type of random forest: regression
##                      Number of trees: 200
## No. of variables tried at each split: 20
##
##           Mean of squared residuals: 3.540442
##                     % Var explained: 79.67
```

Here `Mean of squared residuals` is nothing but OOB , MSE (mean squared error), we can take the square root of the same to get rmse which comes out to be roughly same as the cv error.

% Var explained here can be taken as pseudo R^2 in context of regression problems.

Variable Importance

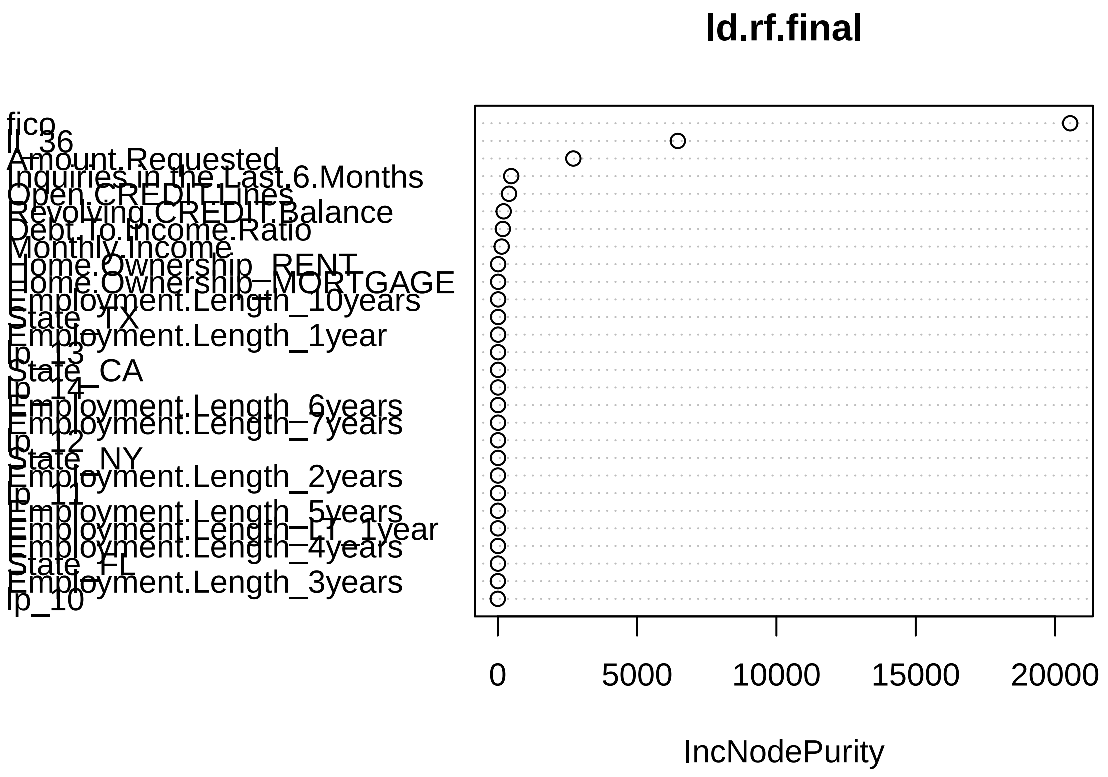
There is one more very useful result that we get out of RandomForest. Imagine that all the predictors that are there in the data, get picked at various nodes across multiple trees in the process of model building. Internally for each variable , when ever its picked (rule from it gets selected for splitting the node), overall decreasing in SSE/Gini is recorded and later averaged over all such instance. This is what is termed as variable importance. Naturally, variables which consistently result in making the model better; end up having higher variable importance.

Even if we dont end up using random forest as one of our final models , we can do a simple run and discard variables which end up having too low variable importance thus reducing data without losing any relevant information for our model.There is no formal cutoff for doing that though. Its subjective.

```
d=importance(ld.rf.final)
d=as.data.frame(d)
d$VariableName=rownames(d)
d %>% arrange(desc(IncNodePurity))
```

```
##    IncNodePurity                   VariableName
## 1   2.054343e+04                           fico
## 2   6.461636e+03                          ll_36
## 3   2.712710e+03               Amount.Requested
## 4   4.836877e+02 Inquiries.in.the.Last.6.Months
## 5   4.044668e+02              Open.CREDIT.Lines
## 6   2.121730e+02        Revolving.CREDIT.Balance
## 7   1.820783e+02           Debt.To.Income.Ratio
## 8   1.410504e+02                 Monthly.Income
## 9   1.509439e+01           Home.Ownership_RENT
## 10  1.455449e+01       Home.Ownership_MORTGAGE
## 11  1.439562e+01      Employment.Length_10years
## 12  1.394004e+01                       State_TX
## 13  1.352320e+01        Employment.Length_1year
## 14  1.219350e+01                          lp_13
## 15  1.180319e+01                       State_CA
## 16  1.165335e+01                          lp_14
## 17  1.127035e+01       Employment.Length_6years
## 18  1.060927e+01       Employment.Length_7years
## 19  9.619512e+00                          lp_12
## 20  9.189549e+00                       State_NY
## 21  7.818041e+00       Employment.Length_2years
## 22  7.803498e+00                          lp_11
## 23  7.650894e+00       Employment.Length_5years
## 24  7.464684e+00     Employment.Length_LT_1year
## 25  7.020047e+00       Employment.Length_4years
## 26  6.827261e+00                       State_FL
## 27  4.425580e+00       Employment.Length_3years
## 28  2.289531e-01                          lp_10
```

```
varImpPlot(ld.rf.final)
```



Partial Dependence Plot

There are no explicit coefficient of variables in tree based algorithms (and any complex algorithm for that matter) which you can use to asses impact and direction of impact of a variable on the response. However we can always visualise how the response changes across different values of given variable. I am showing you how to do it for one variable, you can repeat the process for other variables. Keep in mind that it doesnt make sense in context of flag/binary variables.

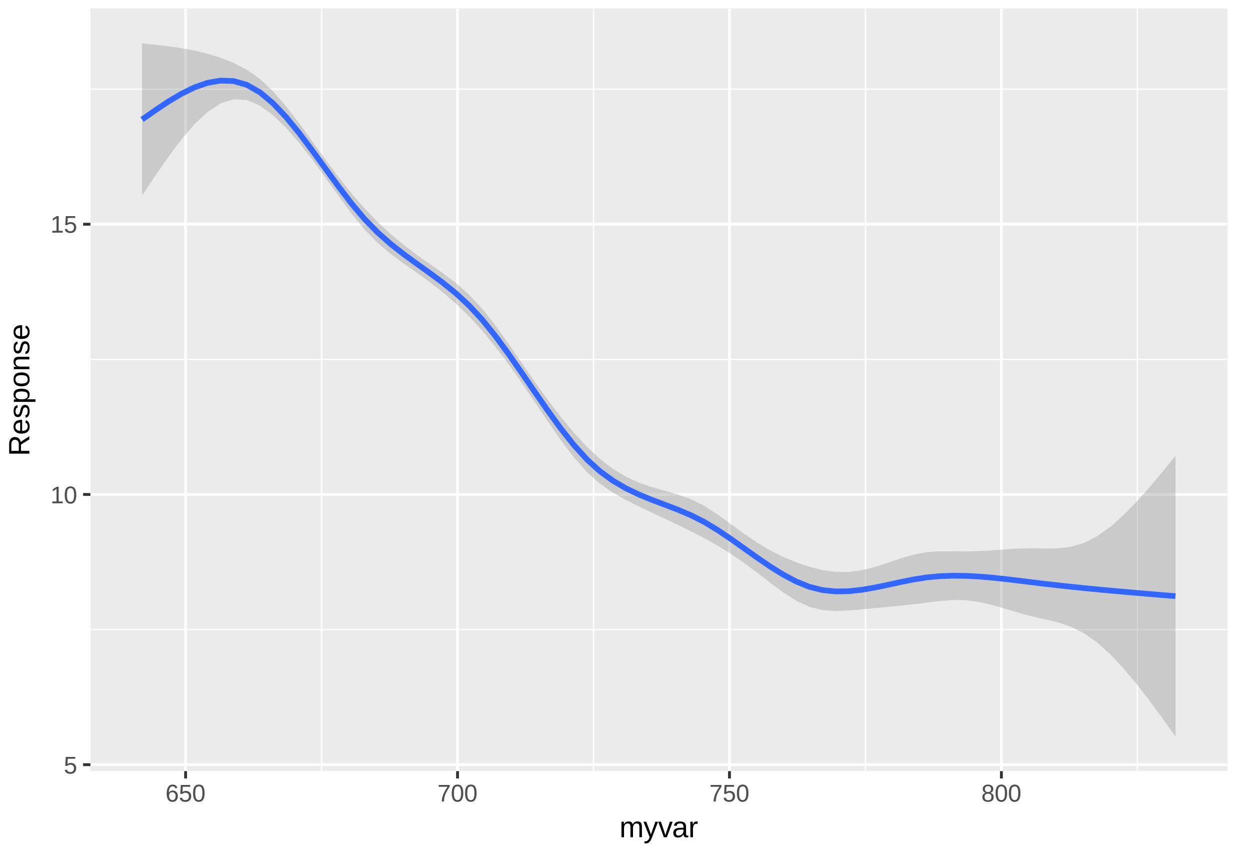
The real values are ofcourse going to be all over the place for any single fixed value of the variable, because other variables are going to take all kind of different values, we'll plot a smoothing curve with variation bands to see how strong the trend is across the variable values.

```
var='fico'

pred.resp = predict(ld.rf.final,newdata=ld_train)
myvar = ld_train[,var]

trend.data=data.frame(Response=pred.resp,myvar=myvar)

trend.data %>% ggplot(aes(y=Response,x=myvar))+
  geom_smooth()
```



where the error band is too wide, pattern is not consistent or number of data points is too low. Otherwise you can see that as fico score increases interest rate goes down, however when fico score is too high , it stops to matter much (trend line is flat). You can similarly asses effect of other variables on the response.

Classification Case with RandomForest

Lets work on a classfication problem with randomForest as well. The data pertains to a census study where along with demographic information for people they were labeled for their income levels as well (>50K or less than 50K annual income) . Our job is to build a model which given people's demographic information can predict their income category. This solution has many application from tax fraud detection to civil policy planning.

we are not working with an explicit test data here, but you know the drill by now to combine and separate later on.

```
ci_train=read.csv("~/Dropbox/0.0 Data/census_income.csv",
                  stringsAsFactors =F)
glimpse(ci_train)
```

```
## Observations: 32,561
## Variables: 15
## $ age            <int> 39, 50, 38, 53, 28, 37, 49, 52, 31, 42, 37, 30,...
## $ workclass      <chr> " State-gov", " Self-emp-not-inc", " Private", ...
## $ fnlwgt         <int> 77516, 83311, 215646, 234721, 338409, 284582, 1...
## $ education      <chr> " Bachelors", " Bachelors", " HS-grad", " 11th"...
## $ education.num  <int> 13, 13, 9, 7, 13, 14, 5, 9, 14, 13, 10, 13, 13,...
## $ marital.status <chr> " Never-married", " Married-civ-spouse", " Divo...
## $ occupation     <chr> " Adm-clerical", " Exec-managerial", " Handlers...
## $ relationship   <chr> " Not-in-family", " Husband", " Not-in-family",...
## $ race           <chr> " White", " White", " White", " Black", " Black...
## $ sex            <chr> " Male", " Male", " Male", " Male", " Female", ...
## $ capital.gain   <int> 2174, 0, 0, 0, 0, 0, 0, 0, 14084, 5178, 0, 0, 0...
## $ capital.loss   <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ hours.per.week <int> 40, 13, 40, 40, 40, 40, 16, 45, 50, 40, 80, 40,...
## $ native.country <chr> " United-States", " United-States", " United-St...
## $ Y              <chr> " <=50K", " <=50K", " <=50K", " <=50K", " <=50K...
```

```
table(ci_train$education,ci_train$education.num)
```

```
##
##                      1     2     3     4     5     6     7     8     9
##    10th              0     0     0     0     0   933     0     0     0
##    11th              0     0     0     0     0     0  1175     0     0
##    12th              0     0     0     0     0     0     0   433     0
##    1st-4th           0   168     0     0     0     0     0     0     0
##    5th-6th           0     0   333     0     0     0     0     0     0
##    7th-8th           0     0     0   646     0     0     0     0     0
##    9th               0     0     0     0   514     0     0     0     0
##    Assoc-acdm        0     0     0     0     0     0     0     0     0
##    Assoc-voc         0     0     0     0     0     0     0     0     0
##    Bachelors         0     0     0     0     0     0     0     0     0
##    Doctorate         0     0     0     0     0     0     0     0     0
##    HS-grad           0     0     0     0     0     0     0     0 10501
##    Masters           0     0     0     0     0     0     0     0     0
##    Preschool        51     0     0     0     0     0     0     0     0
##    Prof-school       0     0     0     0     0     0     0     0     0
##    Some-college      0     0     0     0     0     0     0     0     0
##
##                     10    11    12    13    14    15    16
##    10th              0     0     0     0     0     0     0
##    11th              0     0     0     0     0     0     0
##    12th              0     0     0     0     0     0     0
##    1st-4th           0     0     0     0     0     0     0
##    5th-6th           0     0     0     0     0     0     0
##    7th-8th           0     0     0     0     0     0     0
##    9th               0     0     0     0     0     0     0
##    Assoc-acdm        0     0  1067     0     0     0     0
##    Assoc-voc         0  1382     0     0     0     0     0
##    Bachelors         0     0     0  5355     0     0     0
##    Doctorate         0     0     0     0     0     0   413
##    HS-grad           0     0     0     0     0     0     0
##    Masters           0     0     0     0  1723     0     0
##    Preschool         0     0     0     0     0     0     0
##    Prof-school       0     0     0     0     0   576     0
##    Some-college   7291     0     0     0     0     0     0
```

you can see that these two have perfect correpondence. We'll drop variable education. We'll also convert our response to 0/1 and then factor type for classfication .

```
ci_train=ci_train %>% select(-education)
ci_train$Y=as.numeric(ci_train$Y==" >50K")
ci_train$Y=as.factor(ci_train$Y)
```

And dummy vars for character columns

```
cat_var=names(ci_train)[sapply(ci_train,is.character)]


for(var in cat_var){
  ci_train=CreateDummies(ci_train,var,500)
}
```

For classfication, by default cvTuning uses accuracy as performance measure, which is ok for a multiclass classification (although even there cross entropy would be a better option, but dont worry about that as of now); However for binary class classification auc score is much better and we'll have to write our own custom function for that and pass as an argument to cvTuning. One more change that you'll see in call to cvTuning here is the use of argument `predictArgs`, that is because internally it simply uses hard class prediction by default; if we want to use probabilities we'll need to pass approriate argument values according to the algorithm for which we are doing parameter tuning. For randomForest probability prediction we need to use `type=prob` .

We'll also have new set of parameter values that we want to try out.

```
param=list(mtry=c(5,10,15,20,25,35),
           ntree=c(50,100,200,500,700),
           maxnodes=c(5,10,15,20,30,50,100),
           nodesize=c(1,2,5,10)
           )

mycost_auc=function(y,yhat){
  roccurve=pROC::roc(y,yhat)
  score=pROC::auc(roccurve)
  return(score)
}
```

This is a fixed format that you'll have to use for any custom cost function which you want to use with cvTuning. arguments will be real and predicted values sent by cvTuning.

Also since auc score is higher the better , we'll be modifying our code a little in comparison to regression problem.

```
num_trials=50
my_params=subset_paras(param,num_trials)
my_params

##     mtry ntree maxnodes nodesize
## 212   10    50        5        2
## 256   20   200       10        2
## 266   10   700       10        2
## 101   25   100       20        1
## 218   10   100        5        2
## 554   10   200       30        5
## 162   35   100       50        1
## 268   20   700       10        2
## 198   35   200      100        1
## 803   25   500       50       10
## 182   10    50      100        1
## 23    25   500        5        1
## 588   35   200       50        5
## 167   25   200       50        1
## 334   20    50       30        2
## 575   25    50       50        5
## 368   10   100       50        2
## 534   35   500       20         5
## 5     25    50        5         1
## 337    5   100       30         2
## 144   35   500       30         1
## 215   25    50        5         2
## 409    5   500      100         2
## 574   20    50       50         5
## 86    10   700       15         1
## 698   10   100       15        10
## 664   20    50       10        10
## 381   15   500       50         2
## 233   25   500        5         2
## 542   10    50       30         5
## 766   20   200       30        10
## 117   15   700       20         1
## 626   10   700      100         5
## 714   35   500       15        10
## 483   15    50       15         5
## 235    5   700        5         2
## 689   25   700       10        10
## 157    5   100       50         1
## 681   15   500       10        10
## 492   35   100       15         5
## 839   25   700      100        10
## 490   20   100       15         5
## 793    5   200       50        10
## 73     5   200       15         1
## 241    5    50       10         2
## 495   15   200       15         5
## 697    5   100       15        10
## 262   20   500       10         2
## 659   25   700        5        10
## 711   15   500       15        10
```

```
myauc=0
```

```
for(i in 1:num_trials){
  #print(paste('starting iteration :',i))
  # uncomment the line above to keep track of progress
  params=my_params[i,]

  k=cvTuning(randomForest,Y~.,
             data =ci_train,
             tuning =params,
             folds = cvFolds(nrow(ci_train), K=10, type ="random"),
             cost =mycost_auc, seed =2,
             predictArgs = list(type="prob")
             )
  score.this=k$cv[,2]

  if(score.this>myauc){
    #print(params)
    # uncomment the line above to keep track of progress
    myauc=score.this
    #print(myauc)
    # uncomment the line above to keep track of progress
    best_params=params
  }

  #print('DONE')
  # uncomment the line above to keep track of progress
}
```

```
myauc
```

```
## [1] 0.8945485
```

this is the tentative performance measure. The best paramters are these

```
best_params
```

```
##   mtry ntree maxnodes nodesize
## 1    5   500      100       10
```

Lets use these to build our final model.

```
ci.rf.final=randomForest(Y~.,
                         mtry=best_params$mtry,
                         ntree=best_params$ntree,
                         maxnodes=best_params$maxnodes,
                         nodesize=best_params$nodesize,
                         data=ci_train
                         )
```

you can use this model to predict score for test data (when given) and submit that like this

```
test.score=predict(ci.rf.final,newdata = ci_test,type='prob')[,1]
write.csv(test.score,'mysubmission.csv',row.names = F)
```

As mentioned earlier that if you need to submit/predict hardclasses, predict scores on training data and find cutoff and then use it to convert test.score to hard classes.

Lets look at other results from the model output

```
ci.rf.final
```

```
##
## Call:
##  randomForest(formula = Y ~ ., data = ci_train, mtry = best_params$mtry,      ntree = best_params$nt
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 5
##
##         OOB estimate of  error rate: 14.66%
## Confusion matrix:
##       0    1 class.error
## 0 23648 1072   0.0433657
## 1  3701 4140   0.4720061
```

This shows you OOB estimate of classification error (1- Accuracy). This is generally irrelevant if you were simply more interested in capturing positives.

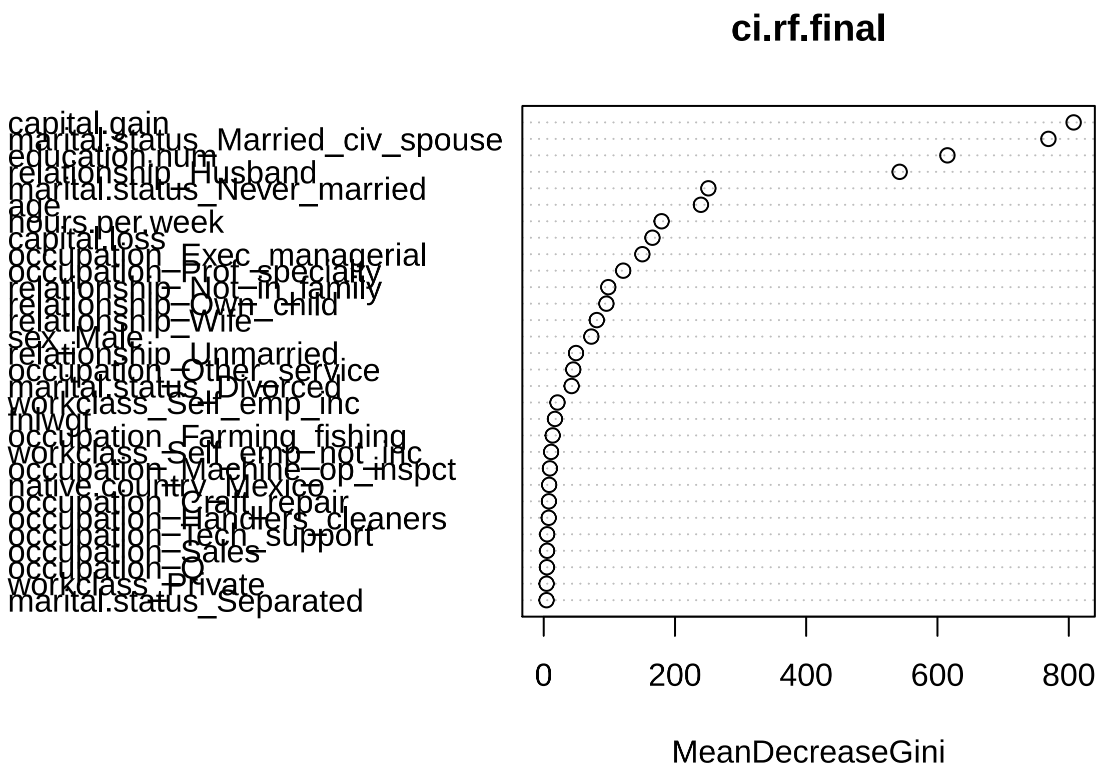
That accruacy is also given on the basis of hard cutoff 0.5 which might not make sense. So dont read too much into this.

Variable importance can be seen following the same process as in regression example.

```
d=importance(ci.rf.final)
d=as.data.frame(d)
d$VariableName=rownames(d)
d %>% arrange(desc(MeanDecreaseGini))
```

```
##    MeanDecreaseGini                      VariableName
## 1        807.409732                      capital.gain
## 2        768.983191 marital.status_Married_civ_spouse
## 3        615.125359                     education.num
## 4        542.355300              relationship_Husband
## 5        251.028740      marital.status_Never_married
## 6        239.572082                               age
## 7        179.708918                    hours.per.week
## 8        165.809264                      capital.loss
## 9        150.462837        occupation_Exec_managerial
## 10       121.302580         occupation_Prof_specialty
## 11        98.582236        relationship_Not_in_family
## 12        95.664829            relationship_Own_child
## 13        80.979367                 relationship_Wife
## 14        72.711119                          sex_Male
## 15        49.392605            relationship_Unmarried
## 16        45.194352          occupation_Other_service
## 17        42.652652           marital.status_Divorced
## 18        21.152889            workclass_Self_emp_inc
## 19        17.244514                            fnlwgt
## 20        13.814342        occupation_Farming_fishing
## 21        11.453542        workclass_Self_emp_not_inc
## 22         9.516203      occupation_Machine_op_inspct
## 23         8.504389             native.country_Mexico
## 24         8.028779           occupation_Craft_repair
## 25         7.733542      occupation_Handlers_cleaners
## 26         5.453047           occupation_Tech_support
## 27         5.355931                  occupation_Sales
## 28         5.008951                      occupation_Q
## 29         4.524239                 workclass_Private
## 30         4.460321          marital.status_Separated
## 31         4.415865       occupation_Transport_moving
## 32         4.344792                       workclass_Q
## 33         4.162388                        race_Black
## 34         4.078792           occupation_Adm_clerical
## 35         3.968206                        race_White
## 36         3.733966      native.country_United_States
## 37         2.724075               workclass_Local_gov
## 38         1.774664               workclass_State_gov
```

```
varImpPlot(ci.rf.final)
```



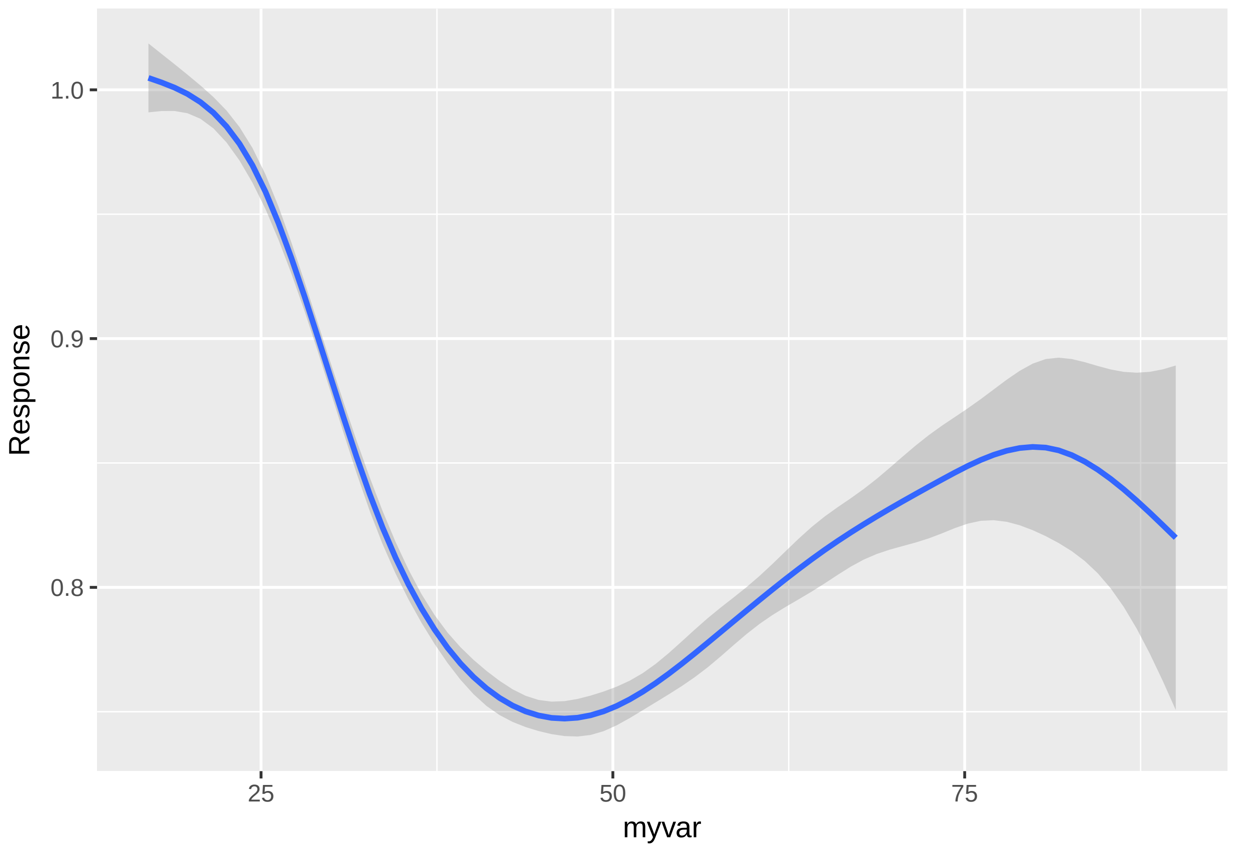
For partial dependence plot , we'll plot probability score vs Variable in question. Process will be very similar.

```
var='age'

pred.resp = predict(ci.rf.final,newdata=ci_train,type='prob')[,1]
myvar = ci_train[,var]

trend.data=data.frame(Response=pred.resp,myvar=myvar)

trend.data %>% ggplot(aes(y=Response,x=myvar))+
  geom_smooth()
```



You can see that until mid forties probability of someone having income level higher than 50K goes down with age, however post that it starts to go up. We dont seem to have much data hence not so reliable trend beyond the age of 75.

You can get similar interpretation for other vars [do keep in mind that it wont make sense for binary/flag vars]

We'll conclude here. In case of any doubts please take to QA Forum on LMS.

Prepared By : Lalit Sachan

Contact: lalit.sachan@edvancer.in