Chapter 2 Classification Tree

Classification Tree (C-Tree) is a method to generate a set of simple rules that can be applied to classify the observations. This is useful in **Data Mining** and has many potential applications. For example, it can be used to derive simple rules for credit approval, medical diagnosis, etc.

Classification Tree method was first developed by Breiman, Friedman, Olshen, and Stone in 1984. It has been an active research area since then. This method will build a classification tree based on the binary splitting of variables, one at a time. The tree is constructed such that each terminal node is as "pure" as possible. That is, in each terminal node, most of the observations are belongs to the same group. Once the classification tree is built, a set of simple classification can be easily obtained. Since this method will search for all possible binary splitting of the variables, it is computational intensive and may be time consuming.

2.1 The Iris flower data set

To illustrate the C-Tree, we first consider the famous Iris flower example. The data set iris.csv contains the information about Iris flower in 5 columns. The first four columns are the measurement of Sepal length, Sepal width, Petal length and Petal width respectively. The last column indicates three different species of the Iris flower (1= *setosa, 2=versicolor* 3= *virginica*). There are 150 observations and 50 observations from each species. Let us first read in this data set.

```
> d<-read.csv("iris.csv")
> dim(d)
[1] 150   5
> names(d)
[1] "Sepal_len" "Sepal_wid" "Petal_len" "Petal_wid" "Species"
```

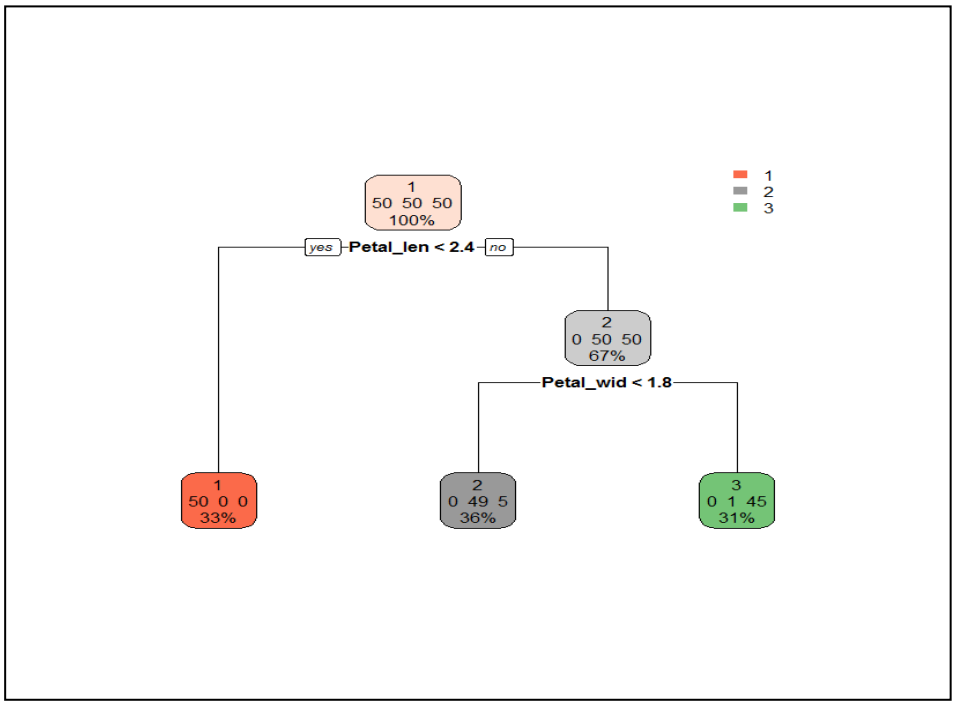
C-Tree has been implemented in R's built-in library *rpart* (stands for **Recursive Partitioning and Regression Trees**). Let us first load the *rpart* library and then built a C-Tree for this Iris data set.

```
library(rpart.plot)           # load rpart.plot library
rpart.plot(ctree,extra=101)   # use rpart.plot, extra=101
```

rpart() is a function in the *rpart* library to build C-Tree. We put the target variable to be classified on the left hand side of the model (specified by the symbol "~") and all the input variables on the right hand side. The result is then saved in the object *ctree*. (use the online

help in R *help(rpart)* for details). There is a function *rpart.plot()* for plotting CTREE output but you have to install *rpart.plot* library. First, mark sure your computer is connected to internet. Go to the top menu choose *package->install package(s)*, choose a mirror site that near your location and choose *rpart.plot* to install. After installation, you should have the *rpart.plot* library in your local hard-disk.

We can plot *ctree* and display the information using *rpart.plot()*. The option *extra=101* will display information of the distribution in each node.



From the output, the classification rules is:

If Petal_len < 2.45 *then* species = 1 (setosa) (50/0/0)
else if Petal_wid < 1.75 *then* species = 2 (versicolor) (0/49/5)
else species = 3 (virginica). (0/1/45)

The numbers in the terminal nodes represent the number of cases. For example, in the node *Petal_len<2.45*, there are 50 setosa, 0 versicolor and 0 virginica; while in the node *Petal_len>=2.45* and *Petal_wid<1.75*, there are 0 setosa, 49 versicolor and 5 virginica. Finally in the node *Petal_len>=2.45* and *Petal_wid>=1.75*, there are 0 setosa, 1 versicolor and 45 virginica.

We can also use print(ctree) to have numerical output instead:

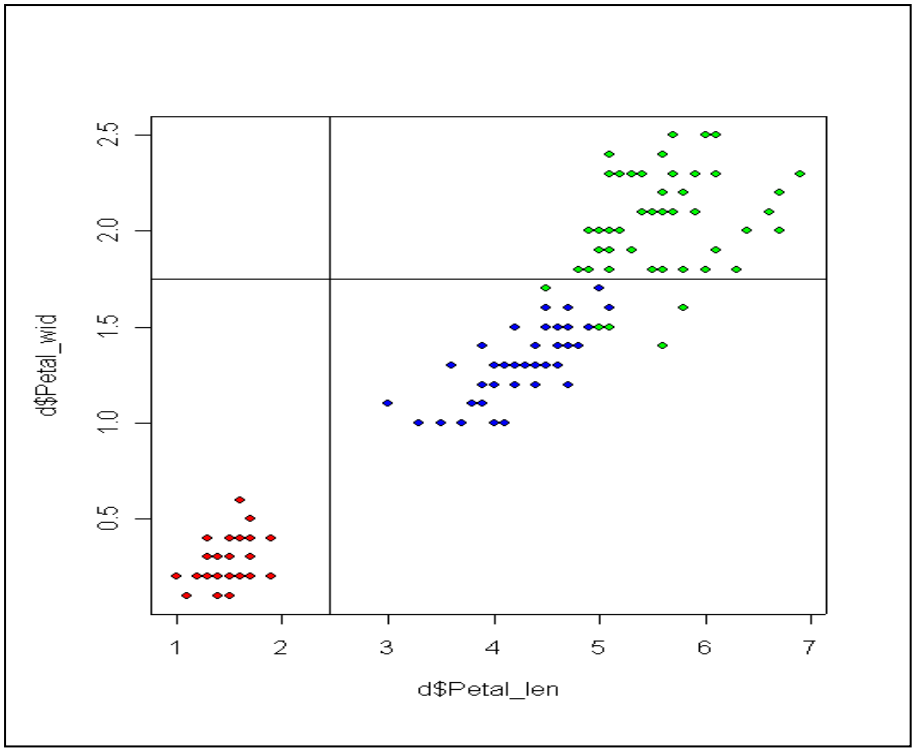
```
> print(ctree)
n= 150
node), split, n, loss, yval, (yprob)
      * denotes terminal node
1) root 150 100 1 (0.33333333 0.33333333 0.33333333)
  2) Petal_len< 2.45 50   0 1 (1.00000000 0.00000000 0.00000000) *
  3) Petal_len>=2.45 100  50 2 (0.00000000 0.50000000 0.50000000)
    6) Petal_wid< 1.75 54   5 2 (0.00000000 0.90740741 0.09259259) *
    7) Petal_wid>=1.75 46   1 3 (0.00000000 0.02173913 0.97826087) *
```

This output shows the details at each node:

1. At node 1 (root), there is totally n=150 cases. Since each group has equal number (50) of cases, the program randomly assign a *yval=1(setosa)* to this node. Then there are 100 cases which are non-setosa, loss=100. The percentage of setosa, versicolor and virginica are given in the parenthesis.
2. At node 2 (left of the split), the split condition is *Petal_len<2.45*. n=50 and all the cases are setosa, therefore the group label is *yval=1* and the loss=0.
3. At node 3 (non-terminal nodel), n=100, therefore 50 versicolor and 50 virginica. Again the program randomly assign a *yval=2(versicolor)* with loss=50.
4. At node 6, n=54. Among them, there are 49 versicolor and 5 virginica. Therefore the *yval=2* and loss=5.
5. At node 7, n=46. There are 1 versicolor and 45 virginica. Therefore the yval=3 and loss=1.

Since this classification tree is relatively simple and the rule also depends on two variables, we can actually plot the observations for illustration.

```
# plot Petal_wid versus Petal_len with different color for each species
> plot(d$Petal_len,d$Petal_wid,pch=21,bg=c("red","blue","green")[d$Species])
> abline(h=1.75)      # add a horizontal line
> abline(v=2.45)      # add a vertical line
```



From this plot, we can clearly see how this classification rule works. The built-in function *predict(ctree)* will produce an *150x3* matrix. Each row represent the probability of that observation belongs to group *1, 2* or *3* respectively. Obviously, we should assign the group label where the prob. is largest. The function *max.col()* will return the column index corresponding to the column maximum which is the group label for the prediction. Finally we produce a classification table (confusion matrix) of this C-Tree.

```
> pr<-predict(ctree)      # pr has 3 columns of prob.
> cl<-max.col(pr)         # find the column index of max in each row of pr
> table(cl,d$Species)     # cross tabulation table

cl   1  2  3
  1 50  0  0
  2  0 49  5
  3  0  1 45
```

From the table, there are five group *3* flowers misclassified into group 2 while there is one group *2* flower misclassified into group *3*. The error rate is *(1+5)/150=4%*.

Although the *rpart()* function is quite easy to use, there are many options hidden in this function. See *help(rpart)* and *help(rpart.control)* for more details. There is one important option need to mention: the *method* option in *rpart* can be one of *"anova", "poisson", "class"* or *"exp"*. If '*y*' is a survival object, then *method="exp"* is assumed, if '*y*' has 2 columns then *method="poisson"* is assumed, if '*y*' is a factor then *method="class"* is assumed, otherwise *method="anova"* is assumed. It is wisest to specify the method directly, especially as more criteria are added to the function. In our example, our '*y*' is a categorical variable; therefore *method="class"* is used.

2.2 Training, validation and testing data set

In data mining, the size of the dataset is huge and we usually partition the data into training, validation and testing data set. The training data is used to build the model. The validation set is used to tune the parameters in the model and the testing data set is used to assess the performance of the model. Let us consider the HMEQ example in chapter 1 again. Firstly we read in the cleaned dataset HMEQ1.CSV:

```
> d<-read.csv("hmeq1.csv")
> (n<-nrow(d))                # get no. of row and display
[1] 3067
> names(d)
 [1] "BAD"     "LOAN"    "MORTDUE" "VALUE"   "REASON"  "JOB"     "YOJ"
 [8] "DEROG"   "DELINQ"  "CLAGE"   "NINQ"    "CLNO"    "DEBTINC"
```

The total sample size is *3067*. Now let us partition *d* into two parts: training dataset *d1* (with sample size=*2045*, roughly *r=2/3* of the total sample size) and testing dataset *d2* (with sample size=*1022*).

```
> set.seed(5104)                              # set the random seed
> r<-2/3                                      # set sampling ratio
> id<-sample(1:n,size=round(r*n),replace=F) # sample r*n random integers from 1 to n
> d1<-d[id,]                                  # training dataset
> d2<-d[-id,]                                 # testing dataset
```

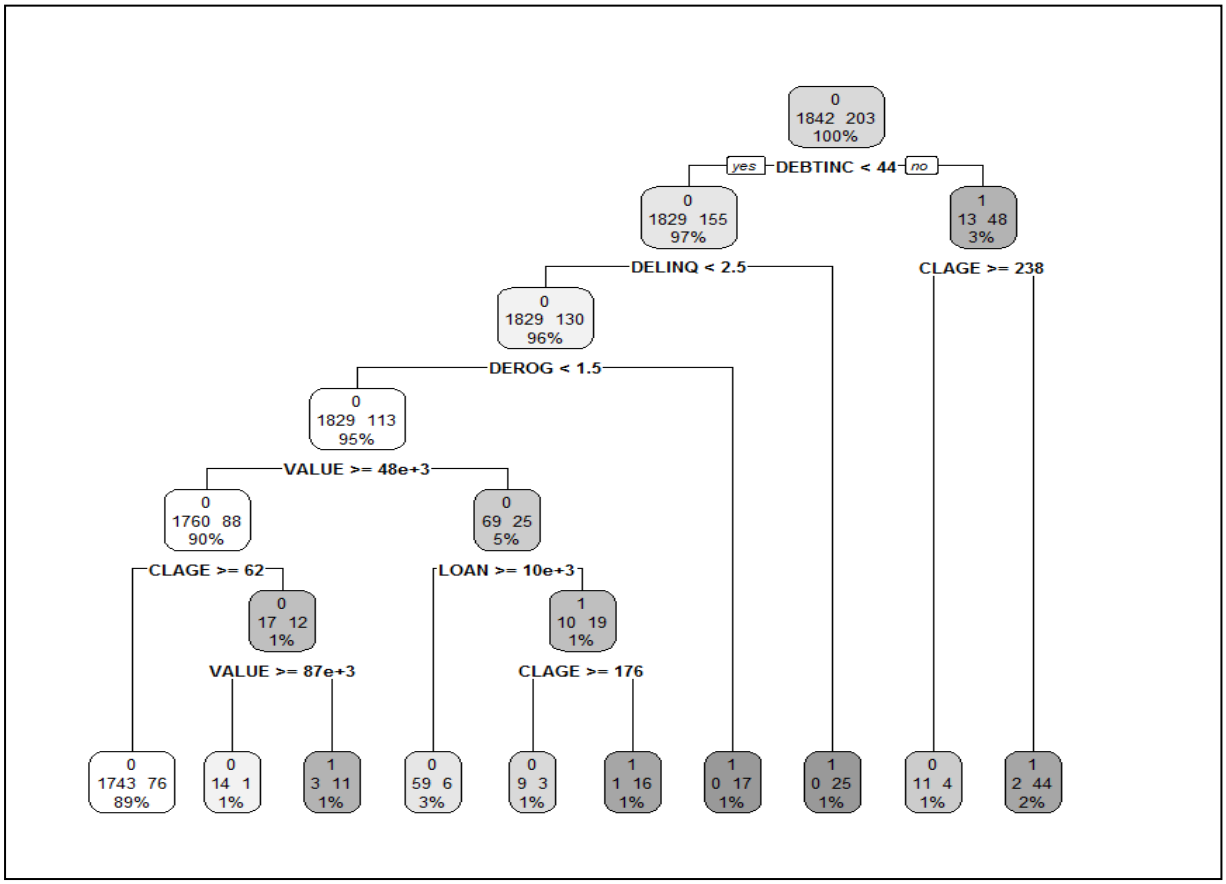
```
> dim(d1)              # display dimension of d1 and d2
[1] 2045   13
> dim(d2)
[1] 1022   13
> names(d)
 [1] "BAD"     "LOAN"    "MORTDUE" "VALUE"   "REASON"  "JOB"     "YOJ"
 [8] "DEROG"   "DELINQ"  "CLAGE"   "NINQ"    "CLNO"    "DEBTINC"
```

The random seed is set so that the training and testing dataset is the same every time when the same seed is used.

2.3 Controlling the Classification Tree

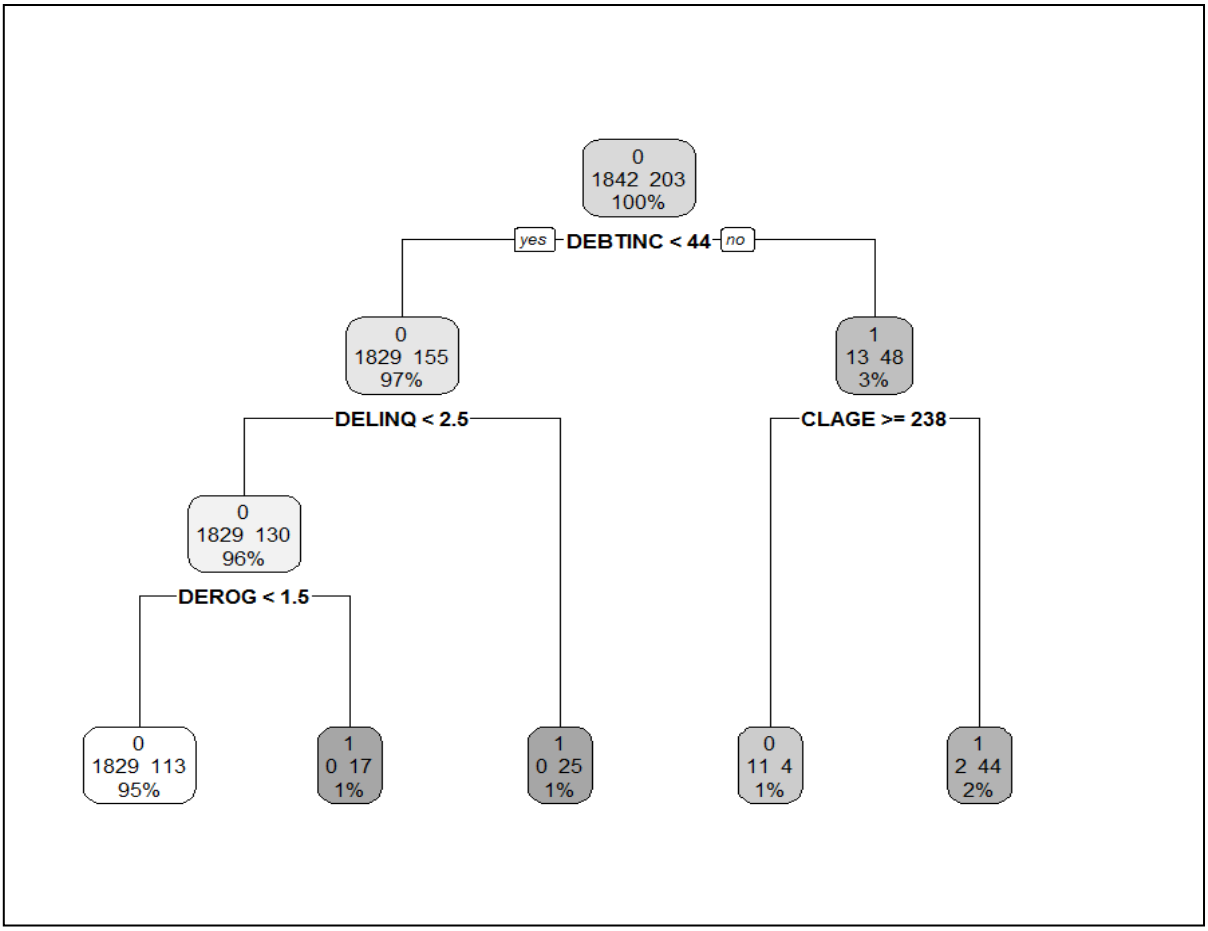
Now we try to build a C-Tree for the training dataset *d1* as follow:

```
> library(rpart)      # load rpart library
> y<-d1$BAD           # create target y
> ctree<-rpart(y~.,data=d1[,2:13],method="class") # alternate way to specify model
> rpart.plot(ctree,extra=101,box.palette=" Grays " ) # plot using grays level
```



The C-Tree built using the default option is far too complicate. Some terminal nodes contain small number of records which the rule may not be very useful. We may have a better control by using the *minsplit* and *maxdepth* options (see help(*rpart.control*) for details). *minsplit* set the minimum number of records in a node and *maxdepth* set the maximum depth of the C-Tree.

```
> ctree<-rpart(y~.,data=d1[,2:13],method="class",minsplit=40,maxdepth=4)
> rpart.plot(ctree,extra=101,box.palette="Grays") # use grays level instead of color
```




```
print(ctree)
n= 2045
node), split, n, loss, yval, (yprob)
      * denotes terminal node
1) root 2045 203 0 (0.90073350 0.09926650)
  2) DEBTINC< 43.68142 1984 155 0 (0.92187500 0.07812500)
    4) DELINQ< 2.5 1959 130 0 (0.93363961 0.06636039)
      8) DEROG< 1.5 1942 113 0 (0.94181256 0.05818744) *
      9) DEROG>=1.5 17   0 1 (0.00000000 1.00000000) *
    5) DELINQ>=2.5 25   0 1 (0.00000000 1.00000000) *
  3) DEBTINC>=43.68142 61  13 1 (0.21311475 0.78688525)
    6) CLAGE>=238.0381 15   4 0 (0.73333333 0.26666667) *
    7) CLAGE< 238.0381 46   2 1 (0.04347826 0.95652174) *
```

By using the options *minsplit=40* and *maxdepth=4*, we obtain a much simpler C-Tree. The classification rule is:

1. If *(DEBTINC<43.68142)* and *(DELINQ<2.5)* and *(DEROG<1.5)* then *BAD*=0. *(1829/113)*
2. If *(DEBTINC<43.68142)* and *(DELINQ<2.5)* and *(DEROG>=1.5)* then *BAD=1. (0/17)*
3. If *(DEBTINC<43.68142)* and *(DELINQ>=2.5)* then *BAD=1. (0/25)*
4. If *(DEBTINC>=43.68142)* and *(CLAGE>=238.0381)* then *BAD=0. (11/4)*
5. If *(DEBTINC>=43.68142)* and *(CLAGE<238.0381)* then *BAD=1. (2/44)*

2.4 Assessing the C-Tree

Continue with the example in section 2.3, we produce a misclassification table for this C-Tree as follow:

```
> pr<-predict(ctree)       # prediction on training dataset d1
> cl<-max.col(pr)          # find col. index of max in each row of pr
> table(cl,d1$BAD)         # classification table
 cl    0    1
  1 1840  117
  2    2   86
```

The error rate is *(2+117)/2045=0.0582*. There are four classification rules. The quality of these classification rules are assessed by **support (or coverage), confidence (or accuracy)** and **capture**:

Support = total number of cases in that rule / total number of cases in the entire dataset,
Confidence = proportion of correctly classified cases in that rule,
Capture = total number of cases of the majority group in that rule / total number of cases of that group in the entire dataset.

For example, there are totally *2045* records in *d1*. Among them, there are *1842* cases of *BAD=0* and *203* cases of *BAD=1*. Therefore,

Rule 1: If *(DEBTINC<43.68142)* and *(DELINQ<2.5)* and *(DEROG<1.5)* then *BAD*=0. *(1829/113)*
Support=*1942/2045=0.9496*, Confidence=*1829/1942=0.9418*, Capture=*1829/1842=0.9929*.

Rule 2: If *(DEBTINC<43.68142)* and *(DELINQ<2.5)* and *(DEROG>=1.5)* then *BAD=1. (0/17)*
Support=*17/2045=0.0083*, Confidence=*17/17=1*, Capture=*17/203=0.0837*.

Rule 3: If *(DEBTINC<43.68142)* and *(DELINQ>=2.5)* then *BAD=1. (0/25)*
Support=*25/2045=0.0122*, Confidence=*25/25=1*, Capture=*25/203=0.1232*.

Rule 4: If *(DEBTINC>=43.68142)* and *(CLAGE>=238.0381)* then *BAD=0. (11/4)*
Support=*15/2045=0.0073*, Confidence=*11/15=0.7333*, Capture=*11/1842=0.00597*.

Rule 5: If *(DEBTINC>=43.68142)* and *(CLAGE<238.0381)* then *BAD=1. (2/44)*
Support=*46/2045=0.0022*, Confidence=*44/46=0.9565*, Capture=*44/203=0.2167*.

In previous section, we have built the C-Tree based on the training dataset *d1* and obtain classification rules for *BAD* and the error rate (training error) is about *5.82%*. However, this error rate is computed using the training dataset *d1* which built the rules. This is the **in-sample** assessment. We can apply these rules to the testing dataset *d2* and produce a classification table for the testing dataset and computing the error rate. R has a built-in function *predict()* help us to apply the *ctree* to *d2* easily as follow:

```
> pr<-predict(ctree,d2)    # or using predict() function on d2
> cl<-max.col(pr)
> table(cl,d2$BAD)         # classification table
cl    0   1
  1 943  48
  2   4  27
```

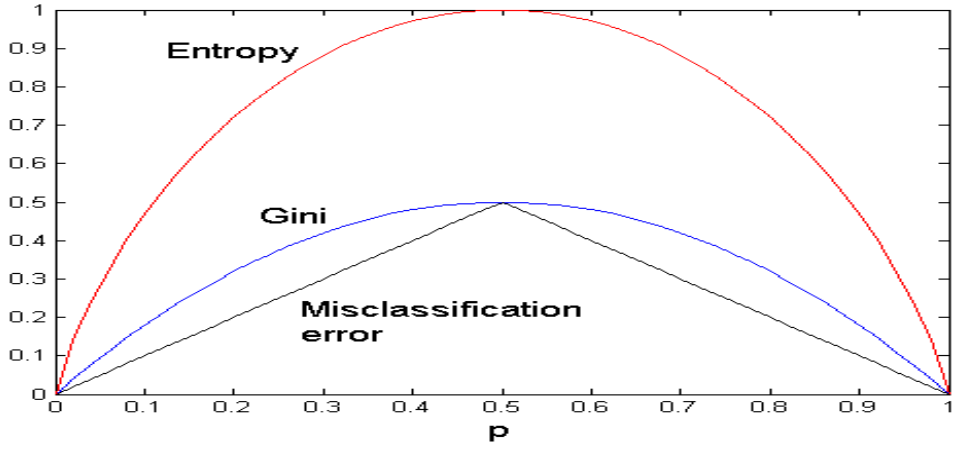
Therefore the error rate (generalization or prediction error) is *(4+48)/1022=0.0509*. Since this testing dataset is independent of the training dataset. This is called the **out-of-sample** assessment and is more accurately reflect the predictive power of these rules.

2.5 Algorithm for C-Tree building

The objective of C-Tree is the find a set of rules such that the distribution of each terminal node is as "pure" as possible. To understand how to build a C-Tree, we need some measure of the impurity of the distribution. Suppose that there are *c* classes and *p(i|t)* is the proportion of records belonging to class *i* at node t, for *i=0,1,..., c-1*. There are several commonly used measures:

1. $Entropy(t) = -\sum_{i=0}^{c-1} p(i \mid t) \log_2 p(i \mid t)$, here we define $0 \log_2 0 = 0$.
2. $Gini(t) = 1 - \sum_{i=0}^{c-1} [p(i \mid t)]^2$
3. $Classification\ error(t) = 1 - \max_i [p(i \mid t)]$

The following is the plot of these functions for *c=2*:



Now we turn to the problem of how to choose the best variable for splitting. We need to compare the impurity of the parent node (before splitting) and the impurity of the child node (after splitting). We define the goodness of a splitting variable v by

$$\Delta(v) = I(parent) - \mathrm{Im}(v) = I(parent) - \sum_{j=1}^{k} N(v_j) I(v_j) / N$$

where *I(.)* is the impurity measure of a given node, *N* is the total number of records in the parent node, *k* is the number of attribute value of *v*, $N(v_j)$ is the number of records associated with the child node v_j. We choose the variable v such that $\Delta(v)$ is largest, or equivalently the impurity measure *Im(v)* is smallest.

Let us first consider a very simple example to illustrate the calculations. Assume that we need to choose between two binary variables *A* and *B* such that

root (6,6)
*node 1:A=0 (4,3) **
*node 2:A=1 (2,3) **

Using variable *A*,
at node 1: *Gini=1-(4/7)^2-(3/7)^2=0.4898*
at node 2: *Gini=1-(2/5)^2-(3/5)^2=0.48*
Δ *=0.5-(7/12)(0.4898)-(5/12)(0.48)=0.5-0.486*

root(6,6)
*node 1:B=0 (1,4) **
*node 2:B=1 (5,2) **

Using variable *B*,
at node 1: *Gini=1-(1/5)^2-(4/5)^2=0.32*
at node 2: *Gini=1-(5/7)^2-(2/7)^2=0.408*
Δ *=0.5-(5/12)(0.32)-(7/12)(0.408)=0.5-0.3715*

Hence variable *B* is prefer to *A* since *Im(A)=0.486* and *Im(B)=0.3715*. The calculation of *Im(v)* can be easily extended to find the best splitting condition for continuous variable where we divide the value of *v* into intervals and obtain the distribution within the interval. For example:

Cheat	No	No	No	Yes	Yes	Yes	No	No	No	No
	Taxable Income									
Sorted Values →	60	70	75	85	90	95	100	120	125	220
Split Positions →	55	65	72	80	87	92	97	110	122	172	230
	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >
Yes	0 3	0 3	0 3	0 3	1 2	2 1	3 0	3 0	3 0	3 0	3 0
No	0 7	1 6	2 5	3 4	3 4	3 4	3 4	4 3	5 2	6 1	7 0
Gini	0.420	0.400	0.375	0.343	0.417	0.400	*0.300*	0.343	0.375	0.400	0.420

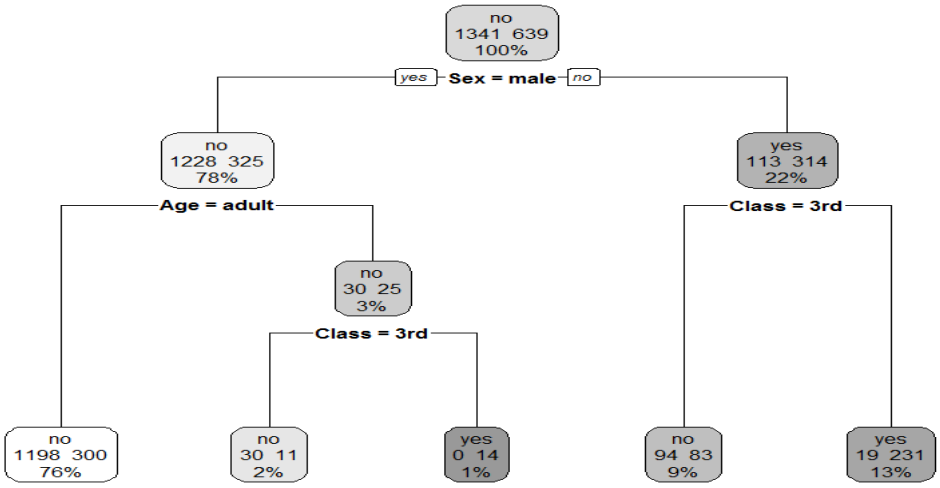
The value of income is sorted and the mid-point between two successive values is used as the split position. The distribution of the target variable is obtained and the impurity measure is computed accordingly. In the above example, the splitting condition *Income<=97* provides the best (smallest) impurity measure.

Child node: *<=97, (Y,N)=(3,3), Gini=1-(3/6)^2-(3/6)^2=1/2*
>97, (Y,N)=(0,4) Gini=1-(0/4)^2-(4/4)^2=0
Impurity measure *Im = (6/10)(1/2)+(4/10)(0) = 3/10 = 0.3.*

2.6 The Titanic dataset

Let use illustrate the C-Tree again by an interesting example. The file "Titanic.csv" contains the information about the *2201* people on board during the first and also the last voyage of Titanic. There are *4* columns: *Class (1^{st}, 2^{nd}, 3^{rd}, crew), Age(adult, child), Sex(Female, Male) and Survive(Yes, No)*. We want to build a classification tree to "predict" who is going to survive in the crash. Instead of using the whole dataset, we randomly sampled *1980* cases (about *90%*) as our training data and the remaining *221* cases as testing data.

```
> d<-read.csv("titanic.csv")                    # read in data
> names(d)                                      # display variables
[1] "Class"    "Age"      "Sex"      "Survive"
> set.seed(12345)                               # set random seed
> (n<-nrow(d))                                  # get and display sample size
[1] 2201
> r<-0.9                                        # set sampling ratio
> id<-sample(1:n,round(r*n))                    # create random sample
> d1<-d[id,]                                    # select training data
> d2<-d[-id,]                                   # select testing data
> dim(d1)
[1] 1981    4
> dim(d2)
[1] 220   4
> ctree<-rpart(Survive~Class+Age+Sex,data=d1,method="class")
> rpart.plot(ctree,extra=101,box.palette="Grays") # use grays level instead of color
```




```
n= 1981
node), split, n, loss, yval, (yprob)
      * denotes terminal node
 1) root 1981 640 no (0.6769308 0.3230692)
   2) Sex=male 1554 326 no (0.7902188 0.2097812)
     4) Age=adult 1499 301 no (0.7991995 0.2008005) *
     5) Age=child 55  25 no (0.5454545 0.4545455)
       10) Class=3rd 41  11 no (0.7317073 0.2682927) *
       11) Class=1st,2nd 14   0 yes (0.0000000 1.0000000) *
   3) Sex=female 427 113 yes (0.2646370 0.7353630)
     6) Class=3rd 177  83 no (0.5310734 0.4689266) *
```

```
> pr<-predict(ctree)       # prediction on training data d1
> cl<-max.col(pr)
> table(cl,d1$Survive)     # classification table for d1
cl    no  yes              # in-sample error rate is (19+395)/1981=20.9%
  1 1322  395
  2   19  245
```

The classification rule from the training data is as follow:

1. If *(Sex=male) and (Age=adult)* then ***NO*** *(1198/300)*
2. If *(Sex=male) and (Age=child)* and *(Class=3rd)* then ***NO*** *(30/11)*
3. If *(Sex=male) and (Age=child)* and *(Class<>3rd)* then ***YES*** *(0/14)*
4. If *(Sex=female) and (Class=3rd)* then ***NO*** *(94/83)*
5. If *(Sex=female) and (Class<>3rd)* then ***YES*** *(19/231)*

The above rules can be simplified into:

1. If *(Class=3rd)* then ***NO***.
2. If *(Class<>3rd)* and *(Sex=female)* then ***YES***.
3. If *(Class<>3rd)* and *(Sex=male)* and *(Age=child)* then ***YES***.

We apply this rule to the testing dataset (*d2*) to assess this rule.

```
> table(d2$Survive,d2$Class,d2$Sex,d2$Age)      # display multi-way table

, ,  = female,  = adult
      1st 2nd 3rd crew          # error=1+5=6
  no    0   1  10    0
  yes  11   8   5    2

, ,  = male,  = adult
      1st 2nd 3rd crew          # error=8+2+5+22=37
  no   14  13  46   58
  yes   8   2   5   22

, ,  = female,  = child         # error=2
      1st 2nd 3rd crew
  no    0   0   2    0
  yes   0   2   2    0

, ,  = male,  = child
      1st 2nd 3rd crew          # error=2
  no    0   0   5    0
  yes   1   1   2    0
```

There are totally *6+37+2+2=47* error cases out of *220* testing cases, so the error rate is *47/220=2136%*. Alternatively, we can simply using *predict()* function on *d2*,

```
> pr<-predict(ctree,d2)         # classification for d2
> cl<-max.col(pr)
> table(cl,d2$Survive)          # out-of-sample error rate = (1+46)/220

cl   no yes
  1 148  46
  2   1  25
```

2.7 Random Forest

Random Forest is an ensemble (also known as classifier combination) method. The idea is simple but computationally intensive. We repeatedly draw many (e.g, *ns=100*) random samples from the original training data and using the model built from each sample to predict the testing data. Each of these *ns* prediction result may not be the same. We use the majority

vote for these *ns* predictions to be our final prediction. In fact, this ensemble method can be applied to any classification methods and not just restricted to classification tree. However, since we built many c-trees using different random samples, we have this fancy name as **Random Forest**. When the random samples drawn from the training data is **bootstrap** sample (i.e., sampling with replacement with same size as the training dataset), we call this as **bootstrap aggregation** method, or simply **Bagging**. Let us formalize the Bagging algorithm of random forest as follow:

1. Suppose we have d1 as training dataset with n1 observations and d2 as testing dataset with n2 observations. Let ns (e.g. 100) as the number of trees built in the random forest.
2. Draw a random sample from d1 **with replacement** with sample size n1 and save it to ds.
3. Built a c-tree using ds and make prediction on the testing dataset d2 using this c-tree.
4. Repeat step 2 to 3 ns times. Save the prediction in step 3 each time.
5. Finally use the majority classification result in these ns predictions as the final prediction.

We will illustrate this random forest using the HMEQ data as follow:

```
# random forest using HMEQ data
d<-read.csv("hmeq1.csv")                    # read in dataset
set.seed(5104)                              # set the random seed
n<-nrow(d)                                  # get sample size
r<-2/3                                      # set sampling ratio
id<-sample(1:n,size=round(r*n),replace=F)  # sample r*n random integers from 1 to n
d1<-d[id,]                                  # training dataset
d2<-d[-id,]                                 # testing dataset
n1<-nrow(d1)                            # get sample size of d1
n2<-nrow(d2)                            # get sample size of d2
ns<-100                                 # set no. of tree in random forest

pred<-matrix(0,nrow=n2,ncol=ns)         # initialize the prediction results to zero
for (i in 1:ns) {                       # loop for ns
  id<-sample(1:n1,replace=T)            # draw n1 random integers with replacement
  ds<-d1[id,]                           # draw random sample
  ctree<-rpart(ds[,1]~.,data=ds[,2:13],method="class",minsplit=40,maxdepth=4)
  pr<-predict(ctree,d2)                 # prediction on d2
  c1<-max.col(pr)-1                     # change predition to 0 or 1
  pred[,i]<-c1                          # save prediction to pred
}

pr<-apply(pred,1,mean)                  # compute row mean of pred
pr1<-(pr>0.5)+0                         # make final prediction

table(pr1,d2$BAD)                       # classfication table for random forest
pr1   0   1
  0 946  47
  1   1  28
```

Note that the error rate of this random forest is *(47+1)/1022=4.7%* and has slight improvement over the ordinary C-TREE on the HMEQ1 dataset!

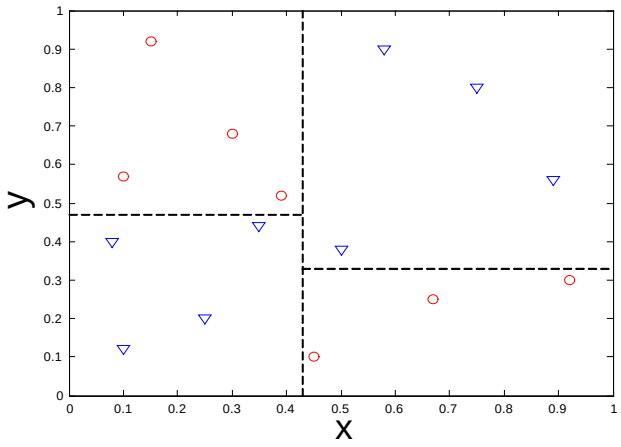
2.8 C-Tree using EXCEL

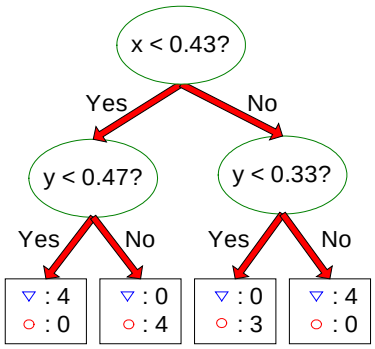
There is a good implementation of Classification tree using EXCEL found in the internet developed by Dr. Saha. The file `ch2-ctree-iris.xls` contains the iris data. Inside the EXCEL file, there are seven separate sheets.

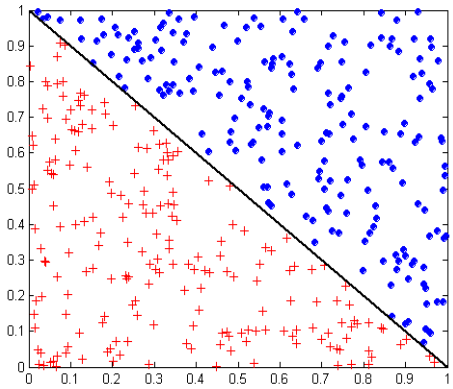
1. ReadMe sheet contains the basic information for using this file. Basically, this file can handle up to *10,000* observations and *50* predictor variables.
2. Data are stored in L24 of the Data sheet. We can specify which variable is class, continuous, categorical or omit. We can copy and paste your own data here to build our own classification tree.
3. In the UserInput sheet, we can specify the parameters to build the tree: minimum node size, maximum purity at each node, and maximum depth of the tree. The data set is divided into two subsets: training data set and testing data set. We can specify the percentage of data points used as testing data. Once the parameters are specified, we can click the Build Tree button to build the tree.
4. Tree sheet contains the classification tree and the Node View contains information in each node. There is a button View node link to the Node view sheet. Information about the **splitting condition, label, confidence** and **support** of each terminal node is given. In node 1, there are *50* records out of total *150* records, therefore the support is *50/150=33%*. The majority class is *setosa* (node *ID=1*). There is no misclassification; therefore the confidence is *100%*. In node *3*, the node *ID=2*. There are *54* records, therefore the support is *54/150=36%*. The misclassification rate is *5/54=9.26%*; therefore the confidence is *100-9.26=90.74%*. Finally, in node *4*, the node *ID=3*. There are *46* records and the support is *46/150=30.67%*. The misclassification rate is *1/46=2.17%* and hence the confidence is *100%-2.33%=97.83%*.
5. The result sheet contains the overall information about this tree as well as the classification (confusion) table of the training data and testing data.
6. Finally a set of classification rules is induced from the tree. The support and confidence of each rule is given as in the Tree sheet. There is an additional **Capture** is also given. The capture is the percentage of correctly classified records captured by the rule. For example, the capture of rule 1 is *100%* since all *50 setosa* species is captured by rule 1. For rule 2, there are *49* out of *50 (=98%) virginica* species are captured. For rule 3, there are *45* out of *50 (=90%) versicolor* species are captured.

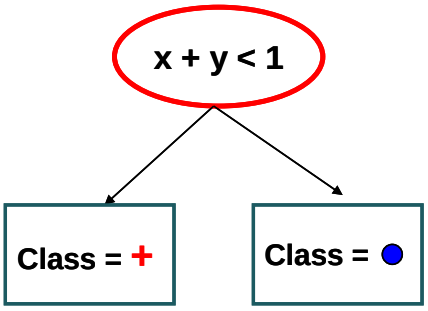
2.9 Final remarks

1. C-Tree is a nonparametric method. It does not require any distributional assumptions.
2. Relatively easy to interpret, especially for small tree.
3. Computationally intensive.
4. The variables are based on binary splitting. It is possible to extend it to multi-way split.
5. The binary splitting is suitable for problem with rectangular decision boundaries. For other type of decision boundaries, the C-Tree may be very complicate.

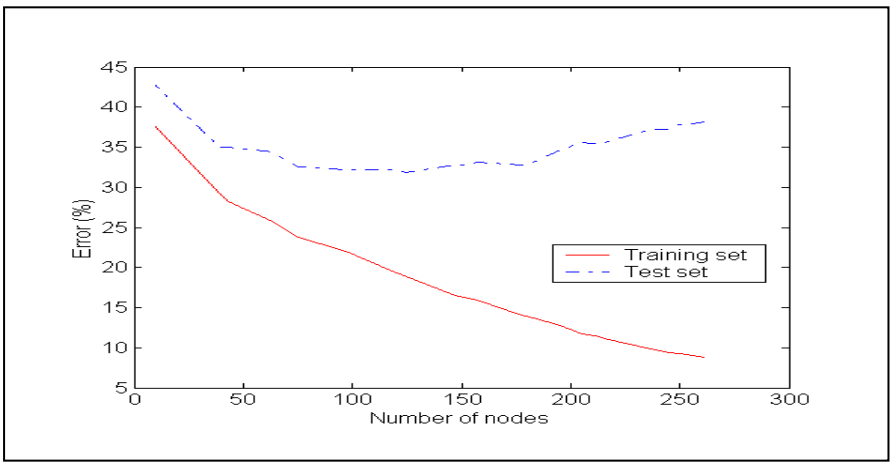








6. Generally speaking, the training error will decrease when the size of the C-Tree increase. However, the generalization error may increase again after certain point. That is, large C-Tree will easily lead to model over-fitting.



Reference:

Chapter 4 of Introduction to Data Mining by Tan, Steinbach and Kumar, Addison Wesley.