DECISION TREES

A **decision tree** is a type of flowchart that shows a clear pathway to a decision. It starts at a single point (or **root node**) which then branches (or **splits**) in two or more directions. Each branch offers different possible outcomes until a final outcome is achieved. When plotted as a graph, it resembles a tree. The nodes at the end of the decision path are called **leaf nodes** (or **terminal nodes**). Between the root node and the leaf nodes, there are a number of **internal nodes** (or **decision nodes**).

Decision trees work for both categorical and numerical variables. Their objective is to partition the population into homogeneous non-overlapping sets, based on the most significant **input** (or **explanatory** or **predictor**) **variables**. The following two types of trees are commonly used in practice: **regression tree** (for a continuous **target variable**), and **classification tree**, for the categorical target variable.

Classification and Regression Trees

Different rules apply to "grow" regression and classification trees. For regression trees, residual sum of squares (RSS) and chi-squared automated interaction detection (CHAID) splitting criteria are used. For classification trees, entropy, Gini, and CHAID criteria are used.

Sometimes created regression and classification trees are too complex, with redundant splits, and a very small number of cases in leaf nodes. In this case **pruning** of the tree is advisable, which results in a smaller number of leaves with more substantial splitting and easier interpretation. The cost-complexity pruning technique is commonly used.

To compare the performance of fitted regression trees and choose the best-performing tree, it is customary to split randomly the original data set into a training set (containing typically 70% or 80% of the data rows) and a testing set (containing the remaining 30% or 20% of the rows). A decision tree is developed on the training set, and the goodness-of-fit is verified on the testing set. For regression trees (when the response is continuous), the response is predicted on the testing data set, and proportions of predicted values that are within, say, 10%, 15%, and 20% of the true values are computed and compared for different models. The model for which these proportions are the highest is the best predicting model. For classification trees, prediction is in the form of the probability of being in each category. We can assume that the category with the highest probability is the one that is being predicted, and the measure of performance is the proportion of predictions that coincide with the true observations in the testing data set. The model with the highest proportion of correctly predicted categories should be chosen, or, equivalently, the model with the smallest **misclassification rate** should be chosen.

Regression Tree

The RSS Splitting Criterion

For Residual-Sum-of-Squares (RSS) splitting criterion, the predictor space is partitioned into multi-dimensional rectangles $R_1, \ldots, R_J$ that minimize the **residual sum of squares** (RSS) $\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$ where y_i is the observed target variable for individual i, $i = 1, \ldots, n$, and $\hat{y}_{R_j}$ is the estimated average response for all observations in the set R_j. In the tree-building process (also termed **training a decision tree**), a top-down binary splitting is used to obtain two new branches at each decision node. For a predictor variable X, the split $X = s$ is chosen in such a way that it leads to the largest reduction of RSS for the two subregions $\{X < s\}$ and $\{X \geq s\}$.

Remark. It is easy to see that the RSS is minimized at the mean value. Indeed, to minimize $\sum_{i=1}^{n} (y_i - \hat{y})^2$, we take the derivative with respect to $\hat{y}$ and set it equal to zero. We get $-2\sum_{i=1}^{n} (y_i - \hat{y}) = 0$ or $\sum_{i=1}^{n} y_i - n\hat{y} = 0$. From here, $\hat{y} = \frac{1}{n}\sum_{i=1}^{n} y_i = \bar{y}$. $\square$

Remark. The splitting procedure based on minimizing the RSS is an example of the **classification and regression tree (CART)** algorithm. More generally, this algorithm involves splitting based on the minimization of some quantity. Note that in the CART algorithm, only binary splitting is allowed.

Historical Note. The CART algorithm was introduced in the book "Classification and Regression Trees" by Breiman, L., Friedman, J., Olshen, R. and Stone, C, Chapman and Hall, Wadsworth, New York, 1984.

The CHAID Splitting Criterion

The chi-squared automatic interaction detection (CHAID) splitting criterion is based on statistical tests. To grow a tree, the response variable can be either continuous of categorical but the predictor variables must be all categorical only (that is, continuous predictors are automatically segmented).

While training a tree, the next best split is determined by an F-test for the continuous response variable and by a chi-squared test for the categorical response variable. The predictor variable with the smallest (Bonferroni adjusted) p-value, i.e., the predictor variable that will yield the most significant split will be considered for the next split in the tree. If the smallest p-value for any predictor is greater than some pre-specified value of alpha, then no further splits will be performed, and the respective node will become a terminal node.

The Bonferroni Adjusted P-value

The Bonferroni correction helps to protect against inflation of the probability of Type I error when performing multiple simultaneous hypotheses testing (it, however, inflates the probability of Type II error). Suppose we would like to test simultaneously k pairs of hypotheses and maintain the probability of Type I error equal to α. The probability of Type I error is then the probability that at least one test shows significance when no significance exists. That is, at least one test statistic falls in the (adjusted) rejection region. The complement of that is that all test statistics lie in the (adjusted) acceptance region, and this translates into the identity: $1 - \alpha = \mathbb{P}$(all test statistics are in the (adjusted) acceptance region) $= (1 - \alpha_{adjusted})^k$, assuming that tests are independent. From here, the Bonferonni adjusted p-value is a p-value that should be compared to $\alpha_{adjusted} = 1 - (1 - \alpha)^{(1/k)} \approx \alpha/k$. For example, if $k = 5$ and five simultaneous hypothesis testings are carried out, then each p-value should be compared not to $\alpha = 0.05$ but $\alpha/k = 0.05/5 = 0.01$.

Remark. Note that with the CHAID splitting criterion, each internal node can have more than two emanating branches, that is, a tree is not limited to binary splitting.

Historical Note. The CHAID splitting criterion was proposed by G.V Kass in "An Exploratory Technique for Investigating Large Quantities of Categorical Data", Journal of the Royal Statistical Society, Series C (Applied Statistics), Vol. 29, No. 2 (1980), pp. 119-127.

The Cost-complexity Pruning Technique

For a decision tree, the **complexity of the tree** T is defined as $|T|$, the number of terminal nodes (leaves). The **cost** of a decision tree, denoted by $R(T)$, is the RSS for regression trees and the misclassification rate for classification trees. The **cost-complexity (CC) measure** is defined as $CC(T) = R(T) + \alpha \cdot |T|$, a linear combination of the cost and complexity, where the term $\alpha \cdot |T|$ is called the **complexity penalty term**. The parameter α is known as the **cost-complexity parameter (ccp)**.

The cost-complexity pruning algorithm (also known as the **minimal cost-complexity** pruning algorithm) works as follows. The cost-complexity of a single node is $CC(t) = R(t) + \alpha$. The branch T_t is defined to be a tree with t as the root node. In general, the cost of a node is greater than the sum of the costs of its terminal nodes, that is, $R(T_t) < R(t)$. However, the cost-complexity measure of a node t and its branch T_t can be made equal, depending on the value of α. We define the **effective** α of a node to be the value where $CC(T_t) = CC(t)$, or $R(T_t) + \alpha_{eff}\,|T_t| = R(t) + \alpha_{eff}$, giving

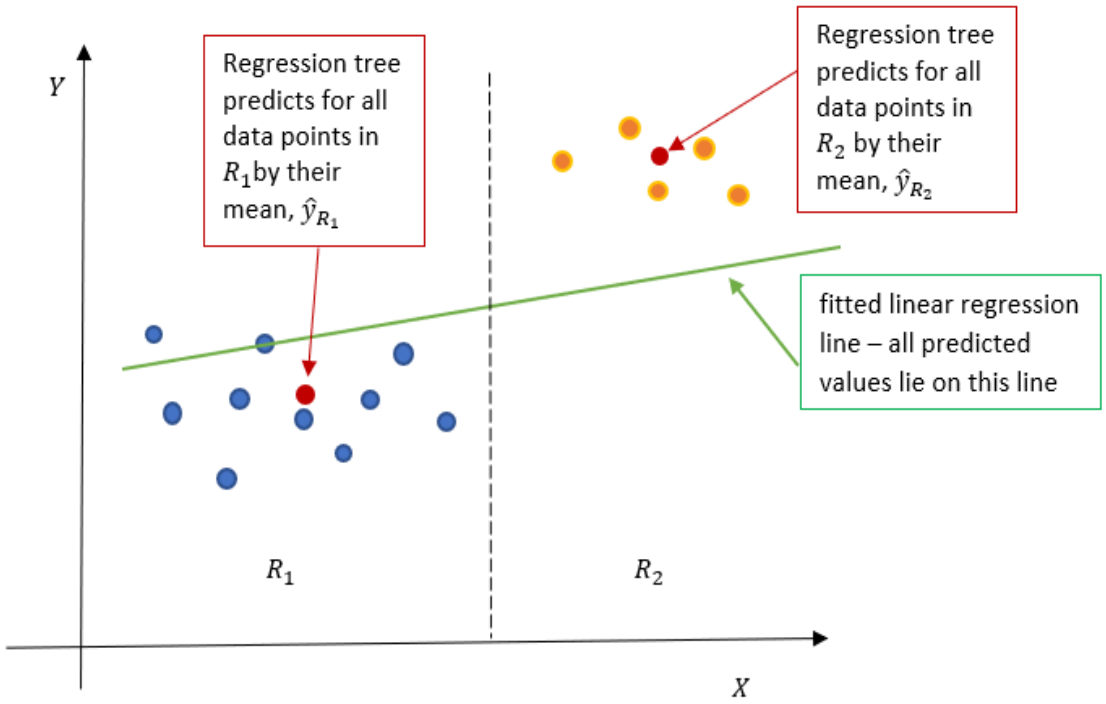
$$\alpha_{eff} = \frac{R(t) - R(T_t)}{|T_t| - 1}.$$

An internal node with the smallest value of α_eff is the **weakest link** and will be pruned. The pruning process continues until there are no more effective α's smaller than a pre-specified value.

More about this pruning algorithm will be explained in the example below.

Historical Note. The cost-complexity pruning algorithm was first described in the book "Classification and Regression Trees" by Breiman, L., Friedman, J., Olshen, R. and Stone, C, Chapman and Hall, Wadsworth, New York, 1984.

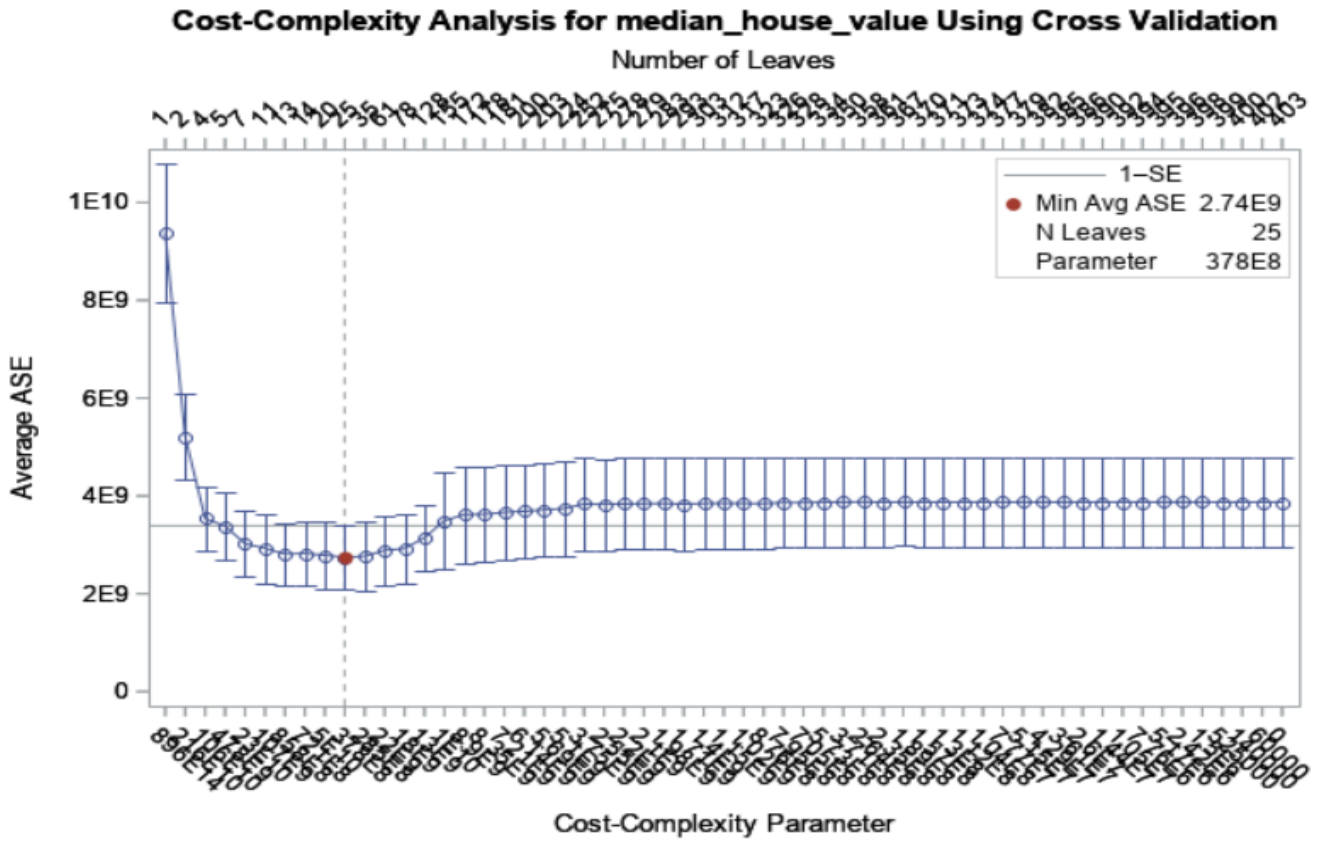
Remark. When data are not distributed in a linear pattern, fitting a regression tree has a clear advantage over a linear regression model as illustrated below.



Example. The data set "housing_data.csv" contains variables aggregated by residential neighborhoods: housing median age, total number of rooms, total number of bedrooms, total population, total number of households, median income, ocean proximity, and median house value. There are 2842 rows in this data set. We model median house value using a regression tree with the RSS splitting and cost-complexity pruning. First, we split the data into 80% training and 20% testing sets. Then we run a full regression tree (without pruning). We use the RSS splitting criterion. Note that since the tree is large, we need to specify a seed. The RSS splitting algorithm is deterministic, but if two splits are equivalent, the order of splits is carried out at random.

In SAS:

```
proc import out=housing datafile="./housing_data.csv" dbms=csv replace;
run;

/*SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS*/
proc surveyselect data=housing rate=0.8 seed=677530
out=housing outall method=srs;
run;

/*RSS SPLITTING CRITERION - FULL TREE*/
proc hpsplit data=housing seed=304576;
class ocean_proximity;
model median_house_value = housing_median_age total_rooms total_bedrooms
population households median_income ocean_proximity;
grow RSS;
partition rolevar=selected(train="1");
run;
```

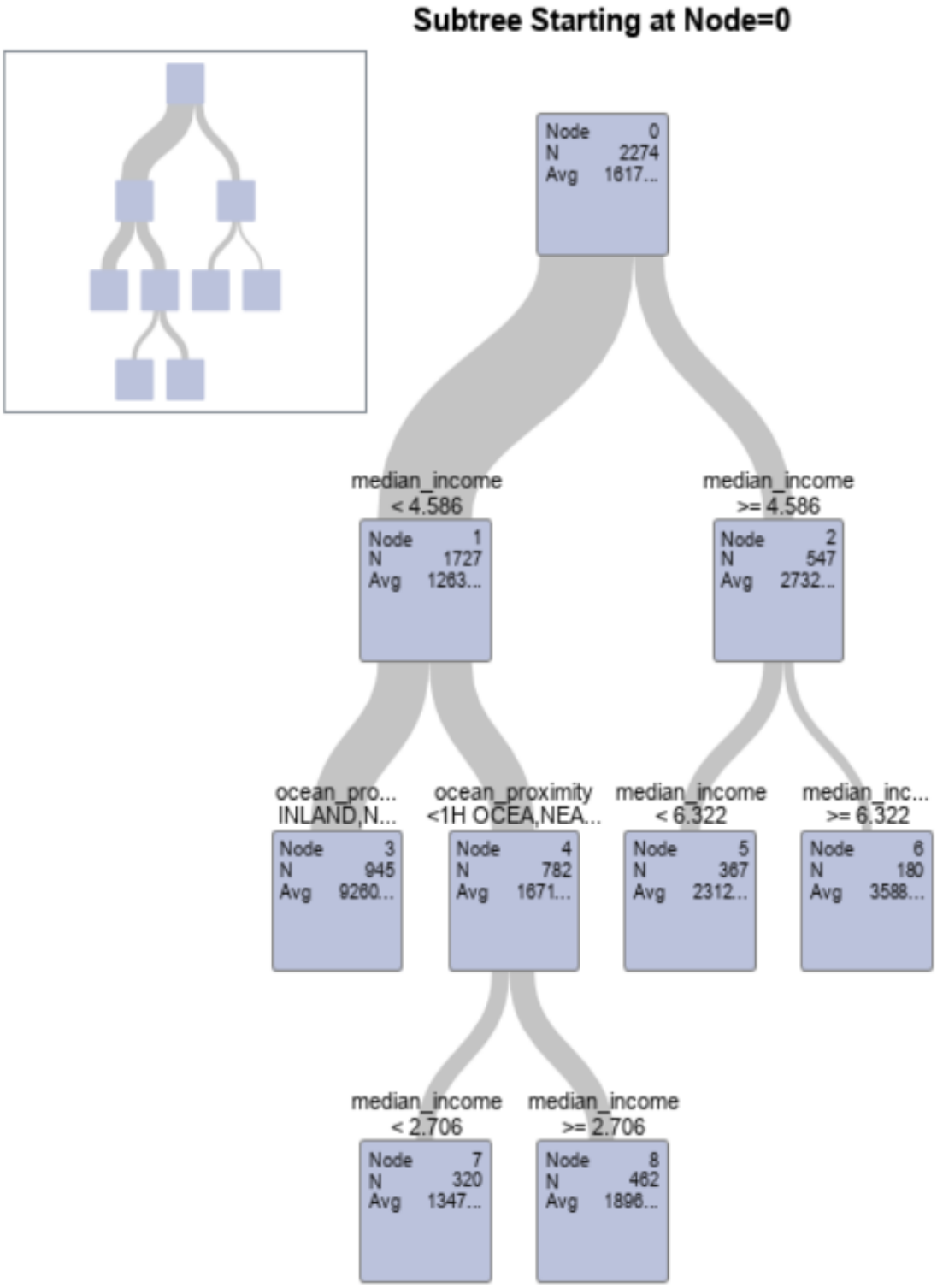


As part of the output, SAS produces a cost-complexity analysis plot. In this plot, the Average ASE (average squared error= RSS/n where n is the sample size of the training set) is plotted against the number of leaves in a tree (top scale on the x-axis). The additional scale on the x-axis (at the bottom) is for the cost-complexity parameter. To compute Average ASE, SAS does (by default) **10-fold cross-validation** by randomly dividing the training set into 10 equal parts and fitting the tree iteratively 10 times, every time holding 1/10th of the data as a **validation set**. ASE is computed for each of the 10 validation sets and then averaged.

As suggested by the plot, the global minimum corresponds to 25 leaf nodes, but this is obviously a very large tree (typically with very poor predictive accuracy). As suggested by Breiman, et. al (1984), it is more reasonable to prune a tree to the number of leaves corresponding to the smallest value of Average ASE below one standard error. The standard error is a standard deviation of the 10 ASE values computed during the cross-validation process. On the plot, the 1-SE line is the horizontal line, and the first value below it corresponds to 5 leaves. Next, we fit a pruned tree with 5 leaf nodes, using the cost-complexity pruning method.

```
/*RSS SPLITTING AND COST-COMPLEXITY PRUNING*/
proc hpsplit data=housing;
class ocean_proximity;
model median_house_value = housing_median_age total_rooms total_bedrooms
population households median_income ocean_proximity;
grow RSS;
prune costcomplexity(leaves=5);
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;
```

Studying the tree on the next page, we can see that all the splits are done on two predictors only. The first split is done with respect to the median income (<4.586, 1727 rows vs. >=4.586, 547), then the latter node is split on the median income again (<6.322, 367 rows vs. >=6.322, 180 rows), and the former node is split on ocean proximity (inland and near ocean, 945 rows vs. others, 782 rows). The latter node is also split on median income (<2.706, 320 rows vs. >=2.706, 462 rows). There are 5 leaf nodes in this pruned tree.


Subtree Starting at Node=0
Node 0
N 2274
Avg 1617...
median_income
< 4.586
median_income
>= 4.586
Node 1
N 1727
Avg 1263...
Node 2
N 547
Avg 2732...
ocean_pro...
INLAND,N...
ocean_proximity
<1H OCEA,NEA...
median_income
< 6.322
median_inc...
>= 6.322
Node 3
N 945
Avg 9260...
Node 4
N 782
Avg 1671...
Node 5
N 367
Avg 2312...
Node 6
N 180
Avg 3588...
median_income
< 2.706
median_income
>= 2.706
Node 7
N 320
Avg 1347...
Node 8
N 462
Avg 1896...

Further, the data set "predicted" contains predicted responses for both training and testing data (identified by the variable "selected"). We limit the data to the testing set and compute proportions of predictions within 10%, 15%, and 20% of the observed values.

```
/*COMPUTING PREDICTION ACCURACY FOR TESTING DATA*/
data test;
set predicted;
if(selected="0");
keep _leaf_ median_house_value P_median_house_value;
run;

data accuracy;
set test;
if(abs(median_house_value-P_median_house_value)<0.10*median_house_value)
then ind10=1; else ind10=0;
if(abs(median_house_value-P_median_house_value)<0.15*median_house_value)
then ind15=1; else ind15=0;
if(abs(median_house_value-P_median_house_value)<0.20*median_house_value)
then ind20=1; else ind20=0;
run;

proc sql;
 select mean(ind10) as accuracy10, mean(ind15) as accuracy15,
 mean(ind20) as accuracy20
 from accuracy;
 quit;
```

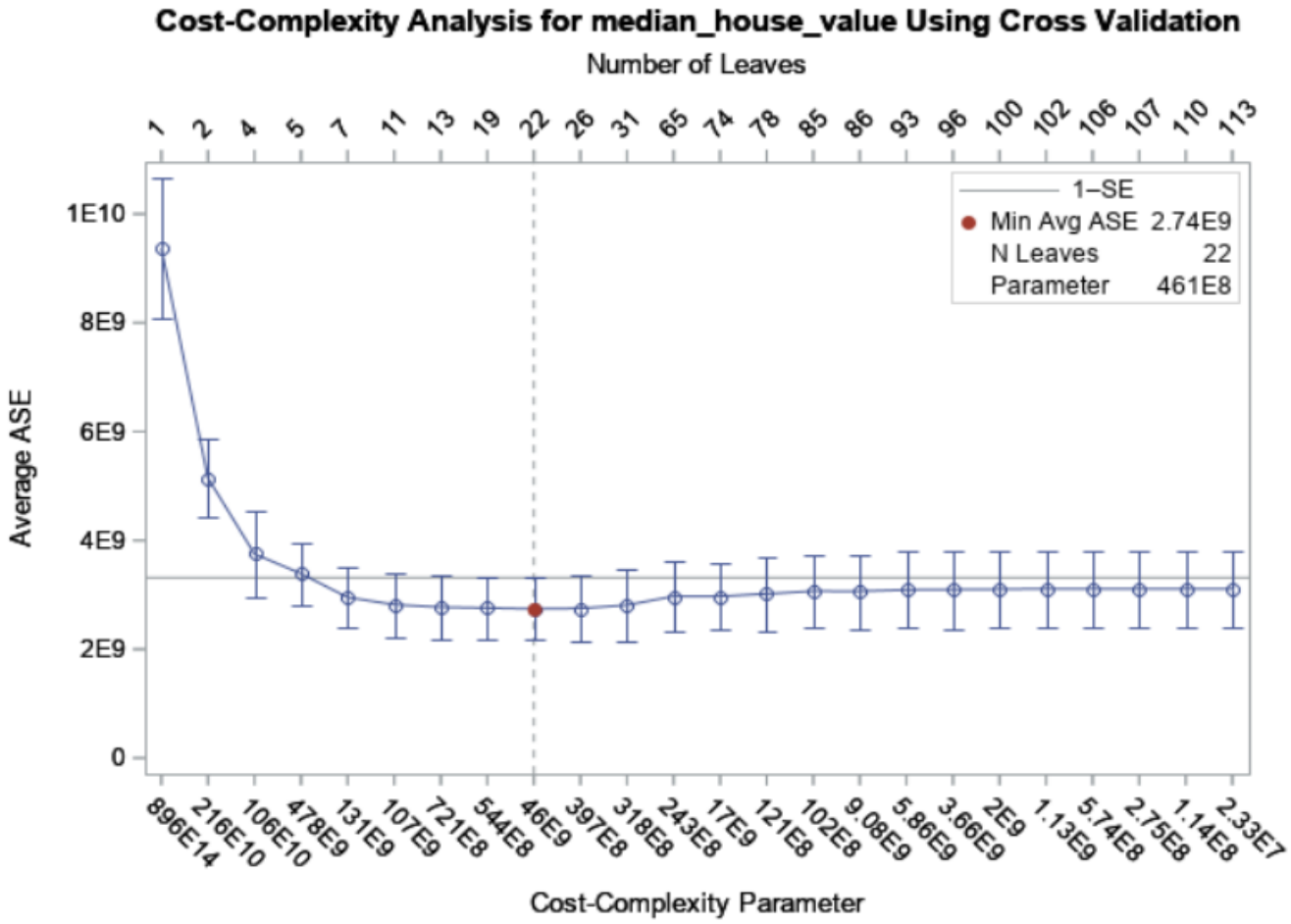
accuracy10	accuracy15	accuracy20
0.257042	0.362676	0.471831

From here, we can see that roughly 27.5% of predictions are within 10%, 36.3% are within 15%, and 47.2% are within 20% of the true observed values in the testing data set.

Now we grow a full regression tree using the CHAID splitting criterion. Note that a seed has to be specified because the algorithm involves random segmentation.

```
/*CHAID SPLITTING CRITERION - FULL TREE*/
```

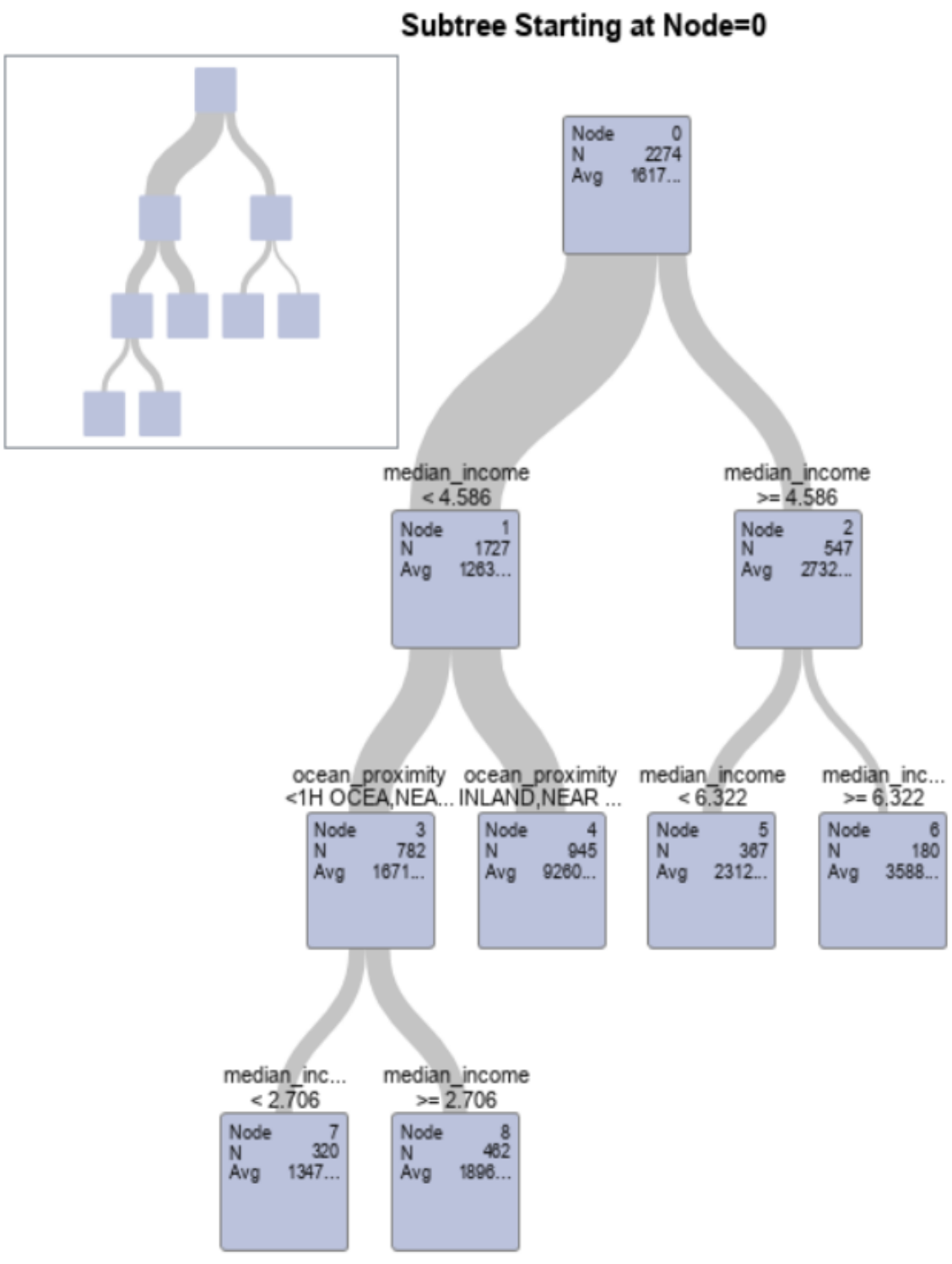
```
proc hpsplit data=housing seed=501231;
class ocean_proximity;
model median_house_value = housing_median_age total_rooms total_bedrooms
population households median_income ocean_proximity;
grow CHAID;
partition rolevar=selected(train="1");
run;
```



As suggested by the graph, the number of leaves in the pruned tree is 5 (on the 1-SE line). We run the following statement and obtain exactly the same regression tree as with the RSS splitting criterion.

```
/*CHAID SPLITTING AND COST-COMPLEXITY PRUNING */
proc hpsplit data=housing seed=501231;
```

```
class ocean_proximity;
model median_house_value = housing_median_age total_rooms total_bedrooms
population households median_income ocean_proximity;
grow CHAID;
prune costcomplexity (leaves=5);
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;
```


Subtree Starting at Node=0
Node 0
N 2274
Avg 1617...
median_income
< 4.586
median_income
>= 4.586
Node 1
N 1727
Avg 1263...
Node 2
N 547
Avg 2732...
ocean_proximity
<1H OCEA,NEA...
ocean_proximity
INLAND,NEAR ...
median_income
< 6.322
median_inc...
>= 6.322
Node 3
N 782
Avg 1671...
Node 4
N 945
Avg 9260...
Node 5
N 367
Avg 2312...
Node 6
N 180
Avg 3588...
median_inc...
< 2.706
median_income
>= 2.706
Node 7
N 320
Avg 1347...
Node 8
N 462
Avg 1896...

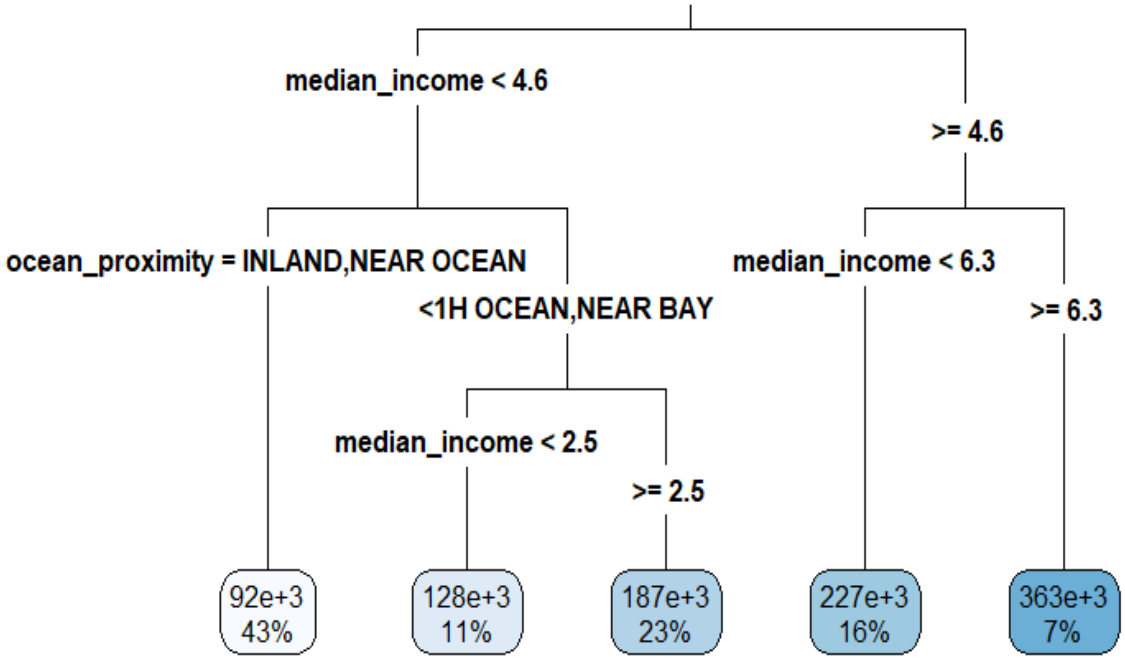
Next, we fit the regression tree with the RSS splitting criteria, using R. We prune the tree to 5 leaves.

In R:

```
housing.data<- read.csv(file="./housing_data.csv", header=TRUE, sep=",")

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
set.seed(105388)
sample <- sample(c(TRUE, FALSE), nrow(housing.data), replace=TRUE, prob=c(0.8,0.2))
train<- housing.data[sample,]
test<- housing.data[!sample,]

#FITTING FULL REGRESSION TREE WITH RSS SPLITTING
install.packages("rpart")
library(rpart) #recursive partitioning and regression
reg.tree.full<- rpart(median_house_value ~ housing_median_age + total_rooms + total_bedrooms
+ population + households + median_income + ocean_proximity, data=train, method="anova",
xval=10, cp=0) #xval=number of cross-validations

printcp(reg.tree.full)
```

```
            CP nsplit rel error  xerror     xstd
1   4.0377e-01      0   1.00000 1.00114 0.038634
2   1.1710e-01      1   0.59623 0.61587 0.024985
3   1.0159e-01      2   0.47913 0.48931 0.022913
4   2.8290e-02      3   0.37754 0.39223 0.019864
5   2.5166e-02      4   0.34925 0.37005 0.020814
6   1.5738e-02      5   0.32408 0.34350 0.020419
7   1.4741e-02      6   0.30834 0.32689 0.020259
8   7.0612e-03      7   0.29360 0.31084 0.019927
9   4.4519e-03      8   0.28654 0.30528 0.019780
10  4.3618e-03      9   0.28209 0.30798 0.020052
```

```
#FITTING REGRESSION TREE WITH RSS SPLITTING AND COST-COMPLEXITY PRUN-
ING
reg.tree.RSS<- rpart(median_house_value ~ housing_median_age + total_rooms + total_bedrooms
+ population + households + median_income + ocean_proximity, data=train, method="anova",
cp=0.026) #pruned tree with 4 splits and 5 leaves

#install.packages("rpart.plot")
library(rpart.plot)
rpart.plot(reg.tree.RSS, type=3)
```

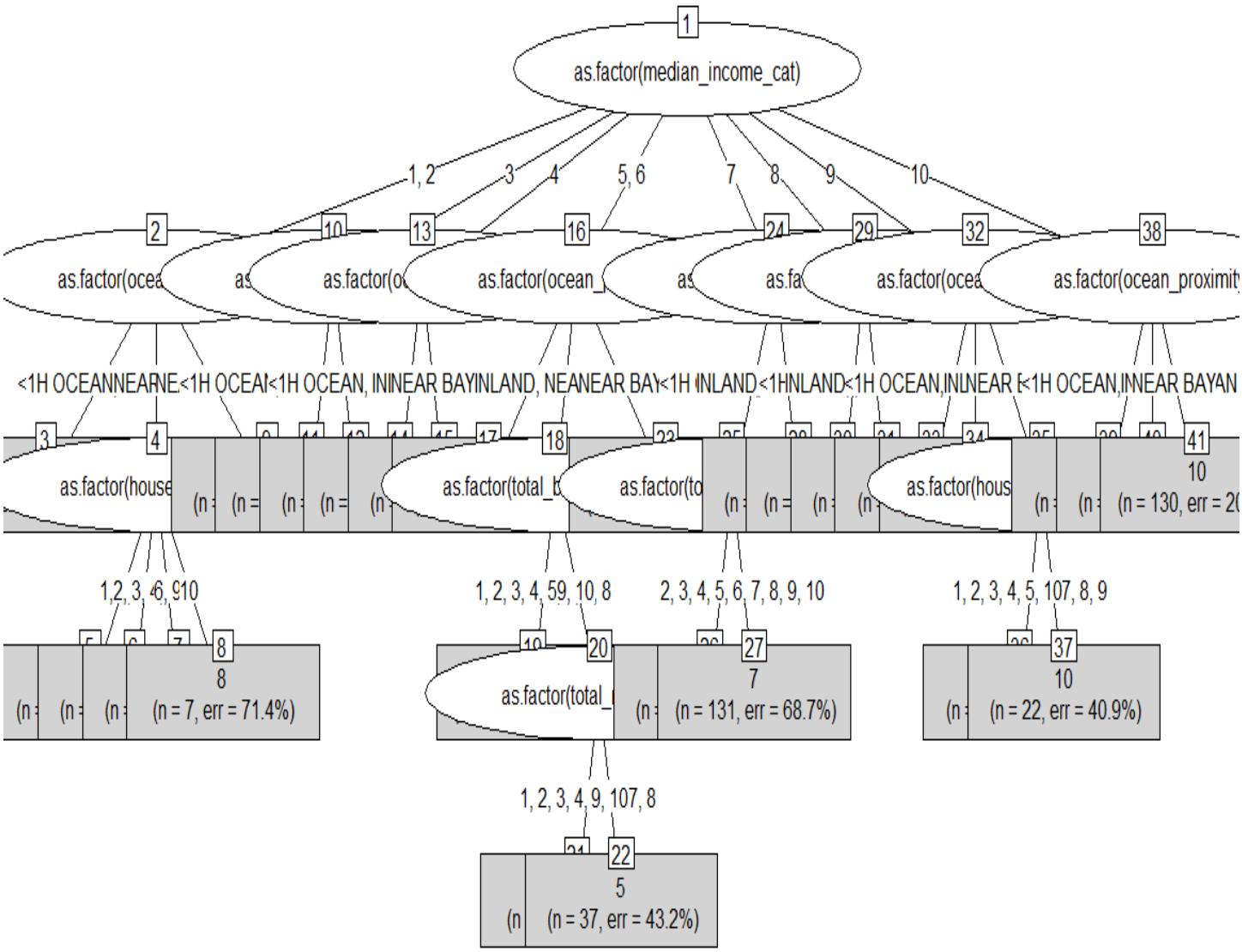



```
#COMPUTING PREDICTION ACCURACY FOR TESTING DATA
P_median_house_value<- predict(reg.tree.RSS, newdata=test)

#accuracy within 10%
accuracy10<-
ifelse(abs(test$median_house_value-P_median_house_value)<0.10*test$median_house_value,1,0)
print(mean(accuracy10))
0.2422018
#accuracy within 15%
accuracy15<-
ifelse(abs(test$median_house_value-P_median_house_value)<0.15*test$median_house_value,1,0)
print(mean(accuracy15))
0.346789
#accuracy within 20%
accuracy20<-
ifelse(abs(test$median_house_value-P_median_house_value)<0.20*test$median_house_value,1,0)
print(mean(accuracy20))
0.440367
```

Note that this tree is similar to the one produced by SAS, and the proportions of accurate predic-

tions are close to the ones in SAS. Now we fit a regression tree with five terminal nodes based on the CHAID splitting criterion.

```
#FITTING REGRESSION TREE WITH CHAID SPLITTING AND COST-COMPLEXITY PRUN-
ING
install.packages("CHAID", repos="http://R-Forge.R-project.org", type="source")

#BINNING CONTINUOUS PREDICTOR VARIABLES
install.packages("dplyr")
library(dplyr)
housing.data<- mutate(housing.data, housing_median_age_cat=ntile(housing_median_age,10),
total_rooms_cat=ntile(total_rooms,10), total_bedrooms_cat=ntile(total_bedrooms,10), popu-
lation_cat=ntile(population,10), households_cat=ntile(households,10),
median_income_cat=ntile(median_income,10), median_house_value_cat=ntile(median_house_value,10))

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
set.seed(105388)
sample <- sample(c(TRUE, FALSE), nrow(housing.data), replace=TRUE, prob=c(0.8,0.2))
train<- housing.data[sample,]
test<- housing.data[!sample,]

library(CHAID)

reg.tree.CHAID<- chaid(as.factor(median_house_value_cat) ~ as.factor(housing_median_age_cat)
+ as.factor(total_rooms_cat) + as.factor(total_bedrooms_cat) + as.factor(population_cat) +
as.factor(households_cat) + as.factor(median_income_cat) + as.factor(ocean_proximity), data=train,
control=chaid_control(maxheight=4))

plot(reg.tree.CHAID, type="simple")
```



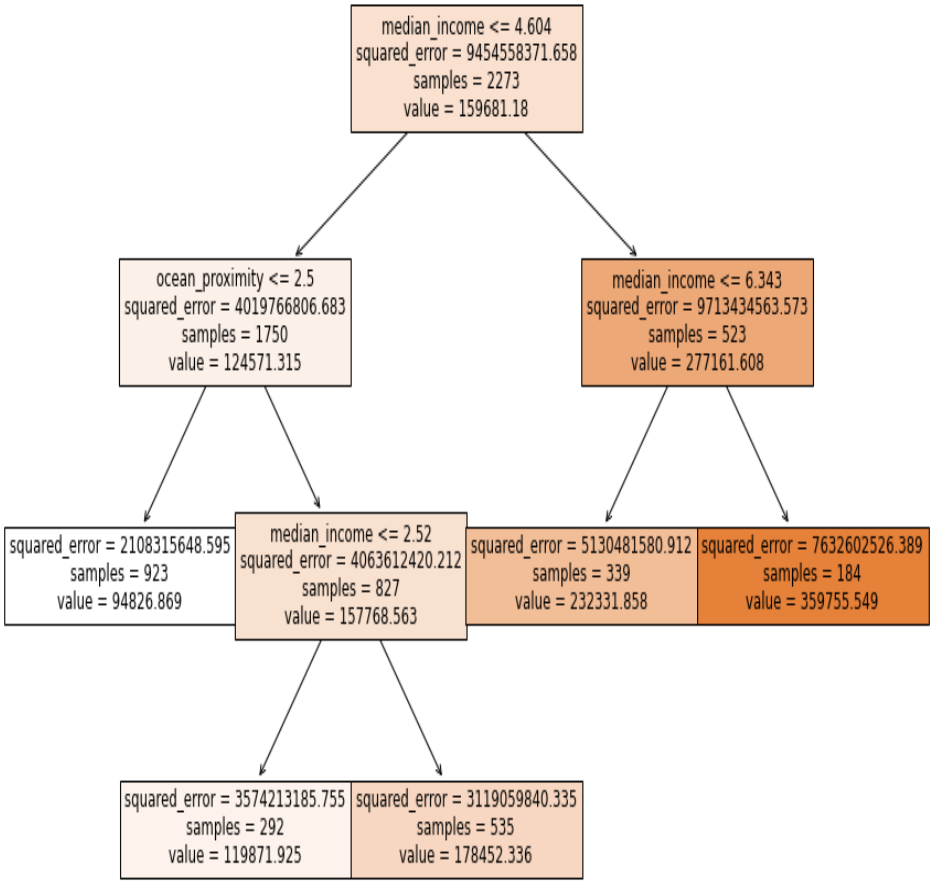

```
#COMPUTING PREDICTION ACCURACY FOR TESTING DATA
predclass<- as.numeric(predict(reg.tree.CHAID, newdata=test))
test<- cbind(test,predclass)

#computing predicted values (mean values per decile in the training set)

aggr.data<- aggregate(train$median_house_value, by=list(train$median_house_value_cat), FUN=mean)
#combining observed and predicted values in the testing set
```

```
aggr.data$predclass<- aggr.data$Group.1
aggr.data$P_median_house_value<- aggr.data$x
test<- left_join(test, aggr.data, by='predclass')

#accuracy within 10%
accuracy10<-
ifelse(abs(test$median_house_value-test$P_median_house_value) <0.10*test$median_house_value,1,0)
print(mean(accuracy10))
0.2880734
#accuracy within 15%
accuracy15<-
ifelse(abs(test$median_house_value-test$P_median_house_value) <0.15*test$median_house_value,1,0)
print(mean(accuracy15))
0.4091743
#accuracy within 20%
accuracy20<-
ifelse(abs(test$median_house_value-test$P_median_house_value) <0.20*test$median_house_value,1,0)
print(mean(accuracy20))
0.5045872
```

Finally, we write a Python code to fit the regression tree using RSS and CHAID splitting criteria.

In Python:

```
import numpy
import pandas
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn import tree
from sklearn.model_selection import train_test_split

housing=pandas.read_csv('C:/Users/000110888/Desktop/housing_data.csv')
coding={'<1H OCEAN': 1, 'INLAND': 2, 'NEAR BAY': 3, 'NEAR OCEAN': 4}
housing['ocean_proximity']=housing['ocean_proximity'].map(coding)
X=housing.iloc[:,0:7].values
y=housing.iloc[:,7].values

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.20,
random_state=348644)

#FITTING REGRESSION TREE WITH RSS SPLITTING CRITERION
rtree = DecisionTreeRegressor(random_state=907420,
criterion="squared_error", max_leaf_nodes=5)
reg_tree_RSS = rtree.fit(X_train, y_train)

#PLOTTING FITTED TREE
fig=plt.figure(figsize=(15,10))
fn=['housing_median_age','total_rooms','total_bedrooms','population',
'households','median_income','ocean_proximity']
tree.plot_tree(reg_tree_RSS, feature_names=fn, filled=True)
```


median_income <= 4.604
squared_error = 9454558371.658
samples = 2273
value = 159681.18
ocean_proximity <= 2.5
squared_error = 4019766806.683
samples = 1750
value = 124571.315
median_income <= 6.343
squared_error = 9713434563.573
samples = 523
value = 277161.608
squared_error = 2108315648.595
samples = 923
value = 94826.869
median_income <= 2.52
squared_error = 4063612420.212
samples = 827
value = 157768.563
squared_error = 5130481580.912
samples = 339
value = 232331.858
squared_error = 7632602526.389
samples = 184
value = 359755.549
squared_error = 3574213185.755
samples = 292
value = 119871.925
squared_error = 3119059840.335
samples = 535
value = 178452.336

```
#COMPUTING PREDICTION ACCURACY FOR TESTING DATA
y_pred=reg_tree_RSS.predict(X_test)

ind10=[]
ind15=[]
ind20=[]

for sub1, sub2 in zip(y_pred, y_test):
    ind10.append(1) if abs(sub1-sub2)<0.10*sub2 else ind10.append(0)
    ind15.append(1) if abs(sub1-sub2)<0.15*sub2 else ind15.append(0)
    ind20.append(1) if abs(sub1-sub2)<0.20*sub2 else ind20.append(0)

#accuracy within 10%
accuracy10=sum(ind10)/len(ind10)
print(accuracy10)

#accuracy within 15%
accuracy15=sum(ind15)/len(ind15)
print(accuracy15)

#accuracy within 20%
accuracy20=sum(ind20)/len(ind20)
print(accuracy20)
```

```
0.2671353251318102
0.36731107205623903
0.46045694200351495
```

For the CHAID splitting criterion, we first split the response variable into deciles and then turn the deciles (0, ..., 9) into nominal classes (0th, ..., 9th). The tree is fitted using the Chefboost package, which fits CHAID trees with nominal response only.

```
#FITTING REGRESSION TREE WITH CHAID SPLITTING CRITERION

#SPLITTING RESPONSE VARIABLE INTO DECILES AND MAKING IT NOMINAL
housing['deciles']=pandas.qcut(housing['median_house_value'], 10, labels=False)
deciles_coding={0:'0th',1:'1st',2:'2nd',3:'3rd',4:'4th',5:'5th',6:'6th',7:'7th',8:'8th',9:'9th'}
housing['deciles']=housing['deciles'].map(deciles_coding)

X=housing.iloc[:,0:7].values
y=housing.iloc[:,7:9].values

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.20, random_state=348644)

X_train=pandas.DataFrame(X_train, columns=['housing_median_age','total_rooms',
'total_bedrooms','population','households','median_income','ocean_proximity'])
y_train=pandas.DataFrame(y_train[:,1], columns=['deciles'])
train_data=pandas.concat([X_train, y_train],axis=1)

#FITTING TREE
from chefboost import Chefboost

config={'algorithm': 'CHAID', 'max_depth': 4}
tree_chaid=Chefboost.fit(train_data, config, target_label='deciles')
```

The fitted tree is not plotted but stored in the `/outputs/rules/rules.py` file. Here are the decision rules for the fitted tree:

```
def findDecision(obj): #obj[0]: housing_median_age, obj[1]: total_rooms, obj[2]: total_bedroo
   # {"feature": "median_income", "instances": 2273, "metric_value": 123.2024, "depth": 1}
   if obj[5]<=3.553185833699956:
      # {"feature": "ocean_proximity", "instances": 1371, "metric_value": 110.1406, "depth":
      if obj[6]<=2.0:
         # {"feature": "housing_median_age", "instances": 764, "metric_value": 109.5716, "dep
         if obj[0]<=26.785340314136125:
            # {"feature": "total_rooms", "instances": 406, "metric_value": 72.5045, "depth":
            if obj[1]<=2437.9162561576354:
               return '0th'
            elif obj[1]>2437.9162561576354:
               return '3rd'
```

```
            else:
               return '0th'
         elif obj[0]>26.785340314136125:
            # {"feature": "population", "instances": 358, "metric_value": 85.0581, "depth": 4
            if obj[3]<=1013.31843575419:
               return '1st'
            elif obj[3]>1013.31843575419:
               return '0th'
            else:
               return '0th'
         else:
            return '0th'
      elif obj[6]>2.0:
         # {"feature": "total_bedrooms", "instances": 607, "metric_value": 54.9472, "depth": 3
         if obj[2]<=757.6878999209841:
            # {"feature": "housing_median_age", "instances": 541, "metric_value": 53.5137, "de
            if obj[0]>39.750462107208875:
               return '3rd'
            elif obj[0]<=39.750462107208875:
               return '5th'
            else:
               return '5th'
         elif obj[2]>757.6878999209841:
            # {"feature": "population", "instances": 66, "metric_value": 25.6699, "depth": 4}
            if obj[3]<=3095.621269773582:
               return '8th'
            elif obj[3]>3095.621269773582:
               return '4th'
            else:
               return '8th'
         else:
            return '5th'
      else:
         return '1st'
   elif obj[5]>3.553185833699956:
      # {"feature": "households", "instances": 902, "metric_value": 97.6671, "depth": 2}
      if obj[4]<=490.00665188470066:
         # {"feature": "ocean_proximity", "instances": 594, "metric_value": 72.1054, "depth":
         if obj[6]>2.0:
            # {"feature": "total_rooms", "instances": 350, "metric_value": 67.4169, "depth": 4
            if obj[1]<=2505.126345836905:
               return '9th'
```

```
            elif obj[1]>2505.126345836905:
               return '9th'
            else:
               return '9th'
         elif obj[6]<=2.0:
            # {"feature": "housing_median_age", "instances": 244, "metric_value": 35.8386, "d
            if obj[0]<=19.64344262295082:
               return '4th'
            elif obj[0]>19.64344262295082:
               return '2nd'
            else:
               return '5th'
         else:
            return '9th'
      elif obj[4]>490.00665188470066:
         # {"feature": "ocean_proximity", "instances": 308, "metric_value": 52.051, "depth": 
         if obj[6]>1.0:
            # {"feature": "housing_median_age", "instances": 261, "metric_value": 43.5737, "d
            if obj[0]<=34.8660205057796:
               return '8th'
            elif obj[0]>34.8660205057796:
               return '8th'
            else:
               return '8th'
         elif obj[6]<=1.0:
            # {"feature": "housing_median_age", "instances": 47, "metric_value": 21.0569, "de
            if obj[0]>16.70212765957447:
               return '8th'
            elif obj[0]<=16.70212765957447:
               return '9th'
            else:
               return '9th'
         else:
            return '9th'
      else:
         return '9th'
   else:
      return '1st'
```

```
#COMPUTING PREDICTION ACCURACY FOR TESTING DATA
y_pred=[]
for i in range(len(y_test)):
    y_pred.append(Chefboost.predict(tree_chaid, X_test[i,:]))

y_test_act=pandas.DataFrame(y_test[:,0], columns=['median_house_value'])
y_pred_class=pandas.DataFrame(y_pred, columns=['deciles'])

pred_data=pandas.concat([y_test_act,y_pred_class],axis=1)

y_train_all=pandas.DataFrame(y_train[:,:], columns=['median_house_value','deciles'])
pred_value=y_train_all.groupby('deciles')['median_house_value'].mean() #computes means for each decile

inner_join = pandas.merge(pred_data, pred_value, on='deciles', how ='inner')

ind10=[]
ind15=[]
ind20=[]
#median_house_value_x=observed value, median_house_value_y=predicted value
for sub1, sub2 in zip(inner_join['median_house_value_x'], inner_join['median_house_value_y']):
    ind10.append(1) if abs(sub1-sub2)<0.10*sub1 else ind10.append(0)
    ind15.append(1) if abs(sub1-sub2)<0.15*sub1 else ind15.append(0)
    ind20.append(1) if abs(sub1-sub2)<0.20*sub1 else ind20.append(0)

#accuracy within 10%
accuracy10=sum(ind10)/len(ind10)
print(accuracy10)

#accuracy within 15%
accuracy15=sum(ind15)/len(ind15)
print(accuracy15)

#accuracy within 20%
accuracy20=sum(ind20)/len(ind20)
print(accuracy20)
```

```
0.17223198594024605
0.2565905096660808
0.3304042179261863
```

□

Classification Tree

A classification tree is a decision tree for a categorical (or even binary) target (response) variable. In classification trees, a natural alternative to the RSS is the **classification error rate**, defined as the fraction of observations in a region that do not belong to the most common class (that is, are misclassified by the tree). Two splitting methods are usually used, one is based on the Gini impurity index and the other is based on entropy.

Gini Impurity Index

The following **Gini impurity index** is commonly used in practice: $G = \sum_{k=1}^{K} p_k(1-p_k)$ where p_k is the proportion of observations in the kth class (that is, the probability of randomly picking a data point in the kth class).

Historical Note. The Gini impurity index is named after an Italian statistician Corrado Gini who proposed the idea in his 1912 paper "Variability and Mutability".

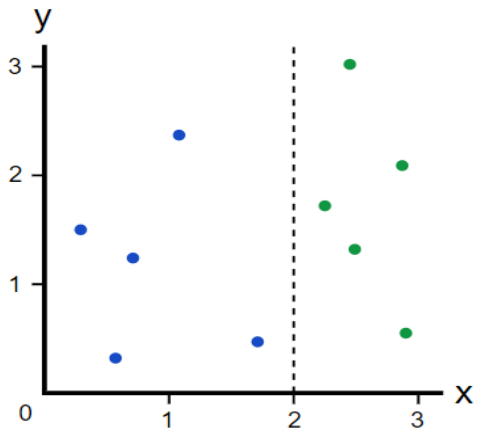
Example. Suppose we observe 10 data points, 5 of which we color blue and the other 5 we color green (see the picture). Suppose we randomly pick a data point and then randomly classify it as blue or green according to the class distribution in the data set. That is, we classify it as blue (or green) with probability $5/10 = 1/2$, since we have 5 data points of each color. What's the probability we classify our data point incorrectly?

$$\mathbb{P}(\text{pick blue, classify blue}) = (1/2)(1/2) = 1/4,$$
$$\mathbb{P}(\text{pick blue, classify green}) = (1/2)(1/2) = 1/4,$$
$$\mathbb{P}(\text{pick green, classify blue}) = (1/2)(1/2) = 1/4,$$
$$\mathbb{P}(\text{pick green, classify green}) = (1/2)(1/2) = 1/4.$$

Therefore, the probability of misclassification is $\mathbb{P}(\text{pick blue, classify green}) + \mathbb{P}(\text{pick green, classify blue}) = 1/4 + 1/4 = 1/2 = 0.5$.

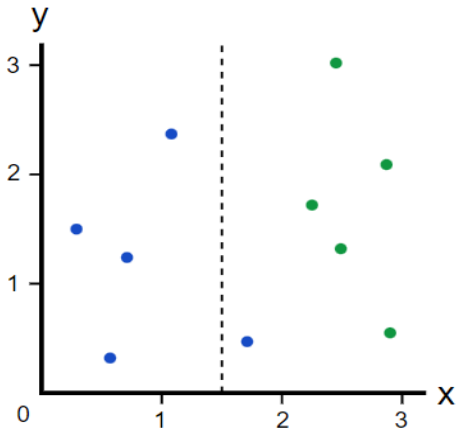
The Gini impurity index is computed as $G = \mathbb{P}(\text{pick blue})\big(1 - \mathbb{P}(\text{pick blue})\big) + \mathbb{P}(\text{pick green})\big(1 - \mathbb{P}(\text{pick green})\big) = (0.5)(1-0.5)+(0.5)(1-0.5) = 0.25+0.25 = 0.5$ and coincides with the probability of misclassification.

If we now make a "perfect" vertical split at $X = 2$ (see the figure below), then 100% of blue dots will be classified correctly as blue and 100% of green dots will be classified correctly as green, so the Gini impurity index for the region on the left is $G_{left} = \mathbb{P}(\text{pick blue})\big(1-\mathbb{P}(\text{pick blue})\big)+\mathbb{P}(\text{pick green})\big(1-\mathbb{P}(\text{pick green})\big) = (1)(1-1)+(0)(1-0) = 0$. The Gini index for the region on the right is $G_{right} = \mathbb{P}(\text{pick blue})\big(1 - \mathbb{P}(\text{pick blue})\big) + \mathbb{P}(\text{pick green})\big(1 - \mathbb{P}(\text{pick green})\big) = (0)(1-0)+(1)(1-1) = 0$. Thus, the "perfect" split turned a data set with a 0.5 impurity into two branches with zero impurity, which is the lowest and the best possible impurity.



A Perfect Split

To illustrate further, consider now an "imperfect" split at $X = 1.5$ where one blue data point falls into the region on the right (as seen in the picture below). The region on the left has only blue data points, so we know that $G_{left} = 0$. The region on the right has 1 blue and 5 green data points, and hence, $G_{right} = (1/6)(1 - 1/6) + (5/6)(1 - 5/6) = 10/36 = 0.2778$.



An Imperfect Split

Finally, the **quality of the split** is determined by weighting the impurity of each region (branch) by how many data points it has. Since the region on the left has 4 data points and the region on the right has 6, we get $(0.4)(0) + (0.6)(0.2778) = 0.1667$. Thus, the amount of impurity that is "removed" by this split is $0.5 - 0.1667 = 0.3333$. This value is called the **Gini gain**. The Gini gain for the "perfect" split is $0.5 - ((0.5)(0) + (0.5)(0)) = 0.5$. When training a decision tree, the best

split is chosen by maximizing the Gini gain. The larger the Gini gain, the better the split. □

Cross-entropy Loss Function

An alternative to the Gini impurity index is the **cross-entropy** (or **cross-entropy loss function** or **Shannon's entropy**) defined by the formula:

$$E = \begin{cases} -\sum_{k=1}^{K} p_k \ln(p_k), & \text{if } 0 < p_k \leq 1, \\ 0, & \text{if } p_k = 0. \end{cases}$$

Historical Note. Claude Shannon (1916-2001) was an American mathematician, electrical engineer, and cryptographer and is known as a "father of information theory".

Example. In our example, the cross-entropy for the entire data set is $E = -(0.5)\ln(0.5) - (0.5)\ln(0.5) = -\ln(0.5) = 0.6931$. For the "perfect" split, the cross-entropy for the left region is $E_{left} = -(1)\ln(1) - 0 = 0$, and so it is for the right region: $E_{right} = 0 - (1)\ln(1) = 0$. For the "imperfect" split, the cross-entropy for the left region is $E_{left} = -(1)\ln(1) - 0 = 0$ and that for the right region is $E_{right} = -(1/6)\ln(1/6) - (5/6)\ln(5/6) = 0.4506$. A split is better if the reduction in the cross-entropy is larger. For the "perfect" split, the cross-entropy is reduced by $0.6931 - [(0.4)(0) + (0.6)(0)] = 0.6931$, whereas for the "imperfect split", it is $0.6931 - [(0.4)(0) + (0.6)(0.4506)] = 0.4227 < 0.6931$, so the "perfect" split wins.

With the cross-entropy loss function, the amount of impurity that is "removed" by a split is termed the **information gain**. The splitting algorithm that uses entropy and information gain as the metric is called the **Iterative Dichotomiser 3 (ID3) algorithm**. A successor of the ID3 algorithm (a recognized improved version of ID3) is **C4.5 algorithm** that uses entropy and gain ratio as the measures. The **gain ratio** is defined by the ratio of information gain and **split information**. How to calculate split information is easier to explain by example.

Example. In the above example, the "imperfect" split can be summarized as follows:

Split	Blue	Green	Total
left	4	0	4
right	1	5	6

Split Information= $-\frac{4}{10}\ln\frac{4}{10} - \frac{6}{10}\ln\frac{6}{10} = 0.673012$, and Gain Ratio=Information Gain/Split Information= $0.4227/0.673012 = 0.628072$. Gain ratios are compared for all candidate variables, and the one with the largest value is selected for the next split. Note that the gain ratio penalizes having too many branches that a split would result in (that is, splitting in a high number of branches results in high information gain, but the split information also increases, so the gain ratio reduces

the bias). □

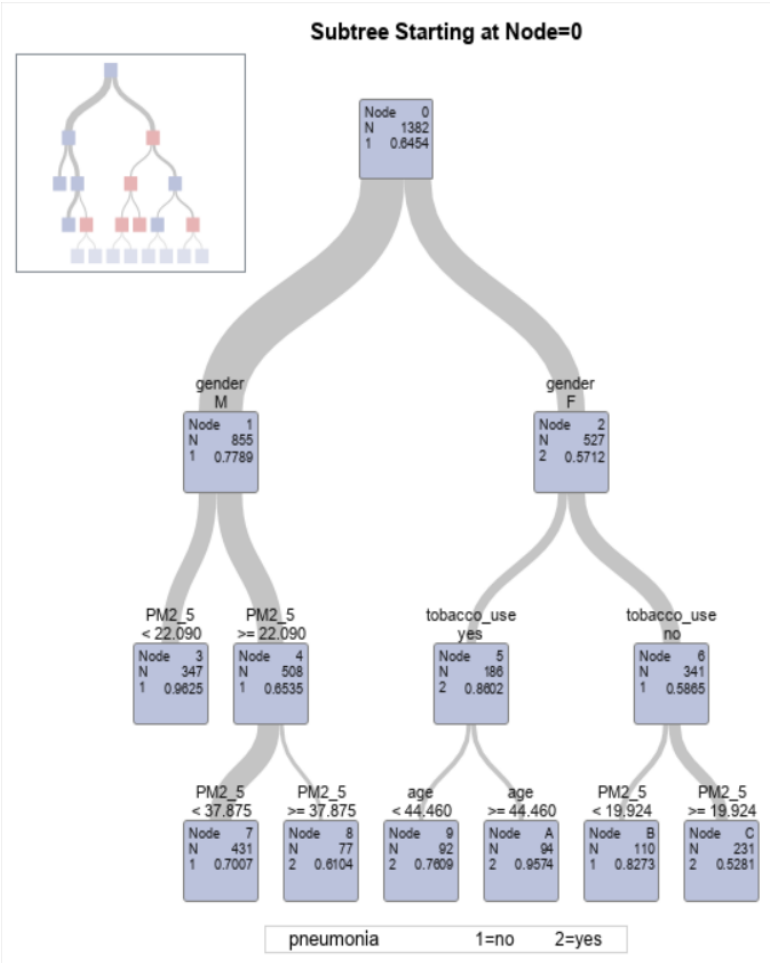
Historical Note. The ID3 algorithm was invented by John Ross Quinlan in 1979, and in 1993 he published "C4.5 Programs for Machine Learning" where he introduced the C4.5 algorithm.

Remark. Note that at any given node, there may be a number of splits on different variables, all of which give almost the same decrease in impurity. Since data are noisy, the choice between competing splits is almost random. However, choosing an alternative split that is almost as good will lead to a different evolution of the tree from that node downward. That's why when a classification tree is developed, a seed should be specified for reproducibility of results.

Example. The data file "pneumonia_data.csv" contains data on individuals' age, gender, an indicator of tobacco use, PM2.5 measurement for the place of residence (atmospheric particulate matter that has a diameter of fewer than 2.5 micrometers, in micro grams per cubic meter), and an indicator of pneumonia. We model pneumonia diagnosis using binary classification trees with the Gini, entropy, and CHAID splitting criteria.

In SAS:

```
proc import out=pneumonia datafile="./pneumonia_data.csv"
dbms=csv replace;

/*SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS*/
proc surveyselect data=pneumonia rate=0.8 seed=6132208
out=pneumonia outall method=srs;
run;

/*GINI SPLITTING AND COST-COMPLEXITY PRUNING */
proc hpsplit data=pneumonia maxdepth=4;
class pneumonia gender tobacco_use;
   model pneumonia(event="yes")= age gender tobacco_use PM2_5;
grow gini;
prune costcomplexity;
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;
```



Further, we want to measure the performance of the tree based on the proportion of observations in the testing set that are predicted correctly. The data set `predicted` contains predicted probabilities of pneumonia for both training and testing sets (indexed by the variable `selected`) (see below).

```
proc print data=predicted (obs=10);
run;
```

Obs	pneumonia	Selected	_Node_	_Leaf_	P_pneumoniano	P_pneumoniayes
1	yes	0	3	0	0.96254	0.03746
2	no	0	7	1	0.70070	0.29930
3	no	1	7	1	0.70070	0.29930
4	no	1	15	5	0.20455	0.79545
5	no	0	19	9	0.51185	0.48815
6	yes	0	19	9	0.51185	0.48815
7	no	1	3	0	0.96254	0.03746
8	no	1	7	1	0.70070	0.29930
9	no	1	7	1	0.70070	0.29930
10	no	1	15	5	0.20455	0.79545

We introduce a cutoff, a value between 0 and 1, such that if a predicted probability of pneumonia is above it, we assume that pneumonia is predicted as present, otherwise absent. We search for an optimal value of cutoff that gives the highest correctly predicted proportion.

```
/*COMPUTING PREDICTION ACCURACY FOR TESTING DATA*/
data test;
set predicted;
if(selected="0");
keep pneumonia P_pneumoniayes;
run;

data cutoffs;
set test;
do i=1 to 99;
tp=(P_pneumoniayes > 0.01*i and pneumonia="yes");
tn=(P_pneumoniayes < 0.01*i and pneumonia="no");
output;
end;
run;

proc sql;
create table rates as
select i, sum(tp+tn)/count(*) as trueclassrate
from cutoffs
group by i;
```

```
select 0.01*i as cutoff, trueclassrate
 from rates
  having trueclassrate=max(trueclassrate);
quit;
```

cutoff	trueclassrate
0.49	0.805797
0.5	0.805797
0.51	0.805797
0.52	0.805797
0.53	0.805797
0.54	0.805797
0.55	0.805797
0.56	0.805797
0.57	0.805797
0.58	0.805797
0.59	0.805797

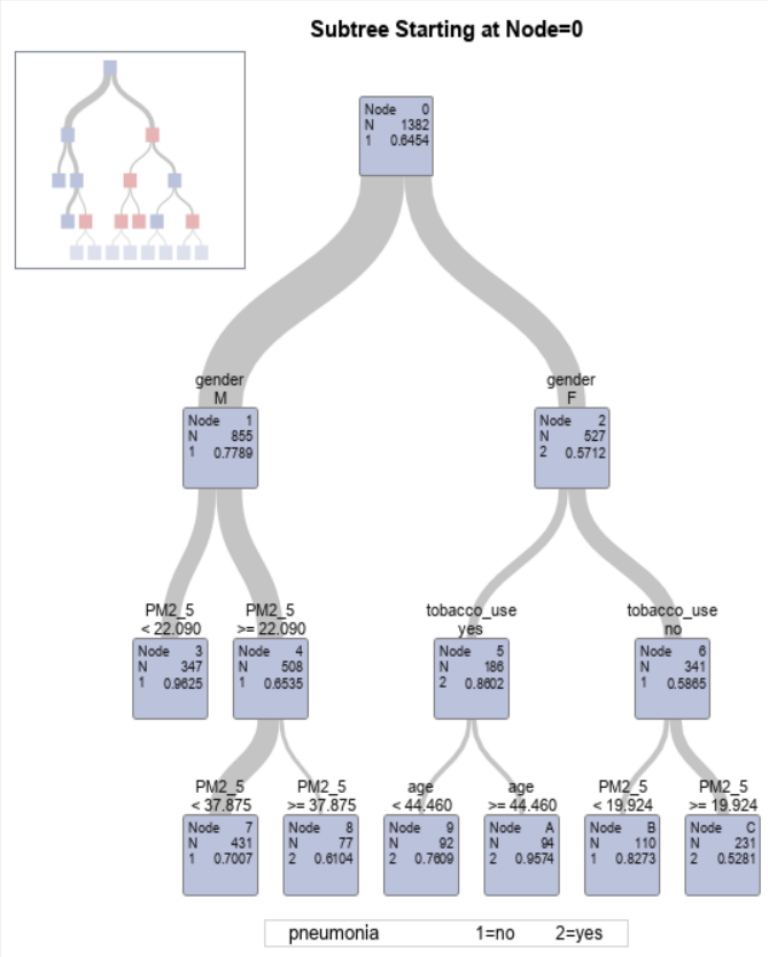
...

0.72	0.805797
0.73	0.805797
0.74	0.805797
0.75	0.805797
0.76	0.805797
0.77	0.805797
0.78	0.805797
0.79	0.805797

From this output, we can see that the largest proportion of correct predictions (80.5797%) is achieved for any cutoff between 0.49 and 0.79.

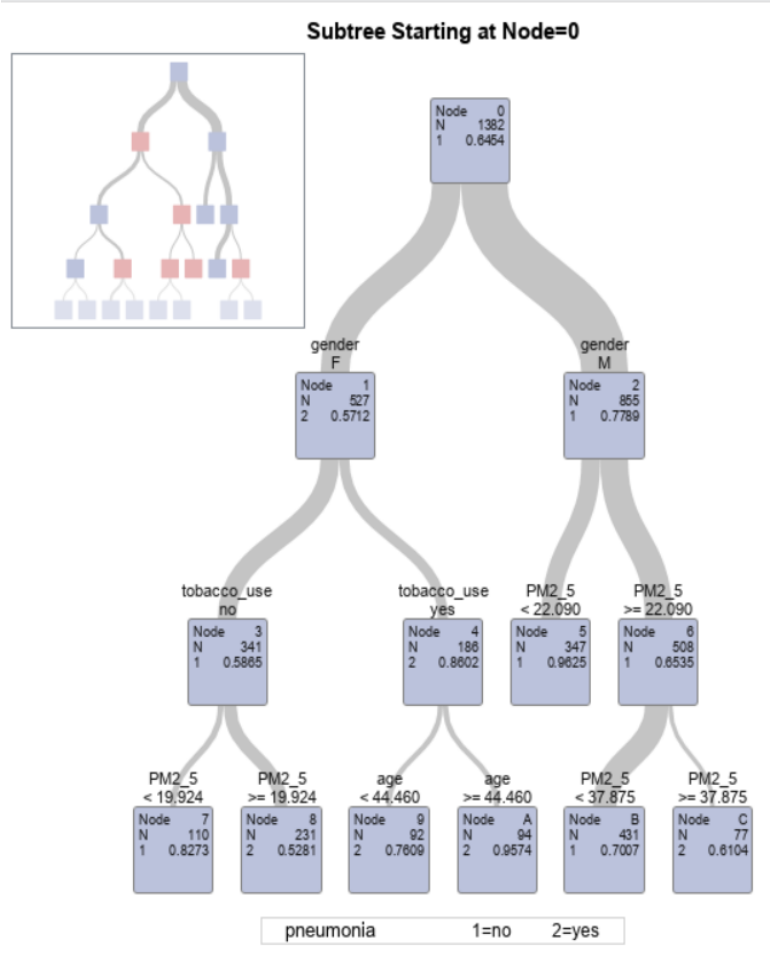
Moving forward, we fit binary classification trees using the entropy and CHAID splitting criteria. We use the cost-complexity pruning algorithm. The trees come out to be the same as the one fitting using the Gini impurity index.

```
/*ENTROPY SPLITTING AND COST-COMPLEXITY PRUNING */
proc hpsplit data=pneumonia maxdepth=4;
class pneumonia gender tobacco_use;
   model pneumonia(event="yes")= age gender tobacco_use PM2_5;
grow entropy;
prune costcomplexity;
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;
```




```
/*CHAID SPLITTING AND COST-COMPLEXITY PRUNING */
proc hpsplit data=pneumonia maxdepth=4;
class pneumonia gender tobacco_use;
   model pneumonia(event="yes")= age gender
```

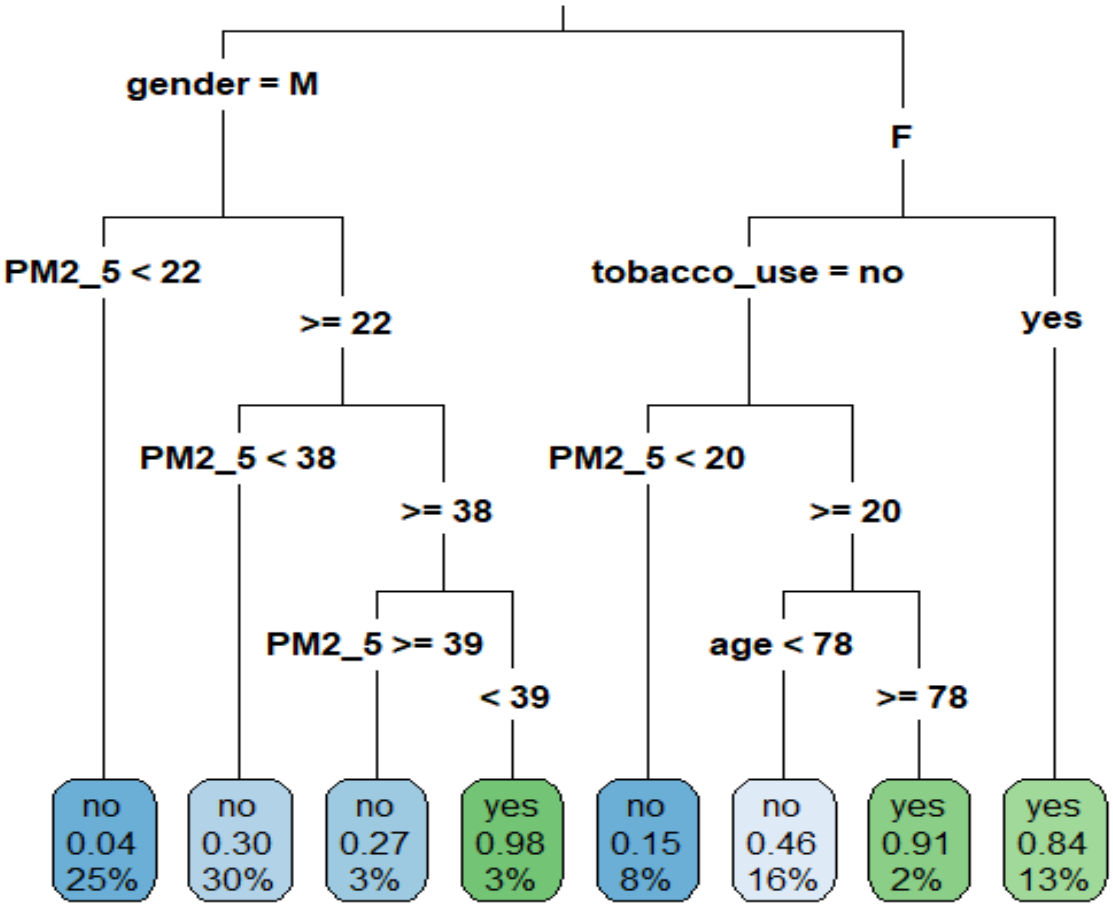
```
tobacco_use PM2_5;
grow CHAID;
prune costcomplexity;
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;
```



In R:

```
pneumonia.data<- read.csv(file="./pneumonia_data.csv", header=TRUE, sep=",")

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
set.seed(283605)
sample <- sample(c(TRUE, FALSE), nrow(pneumonia.data), replace=TRUE, prob=c(0.8,0.2))
```

```
train<- pneumonia.data[sample,]
test<- pneumonia.data[!sample,]

#FITTING PRUNED BINARY TREE WITH GINI SPLITTING
library(rpart)
tree.gini<- rpart(pneumonia ~ age + gender + tobacco_use + PM2_5, data=train, method="class",
parms=list(split="Gini"), maxdepth=4)

library(rpart.plot)
rpart.plot(tree.gini, type=3)
```




```
#COMPUTING PREDICTION ACCURACY FOR TESTING DATA
pred.values<- predict(tree.gini, test)
test<- cbind(test,pred.values)
```

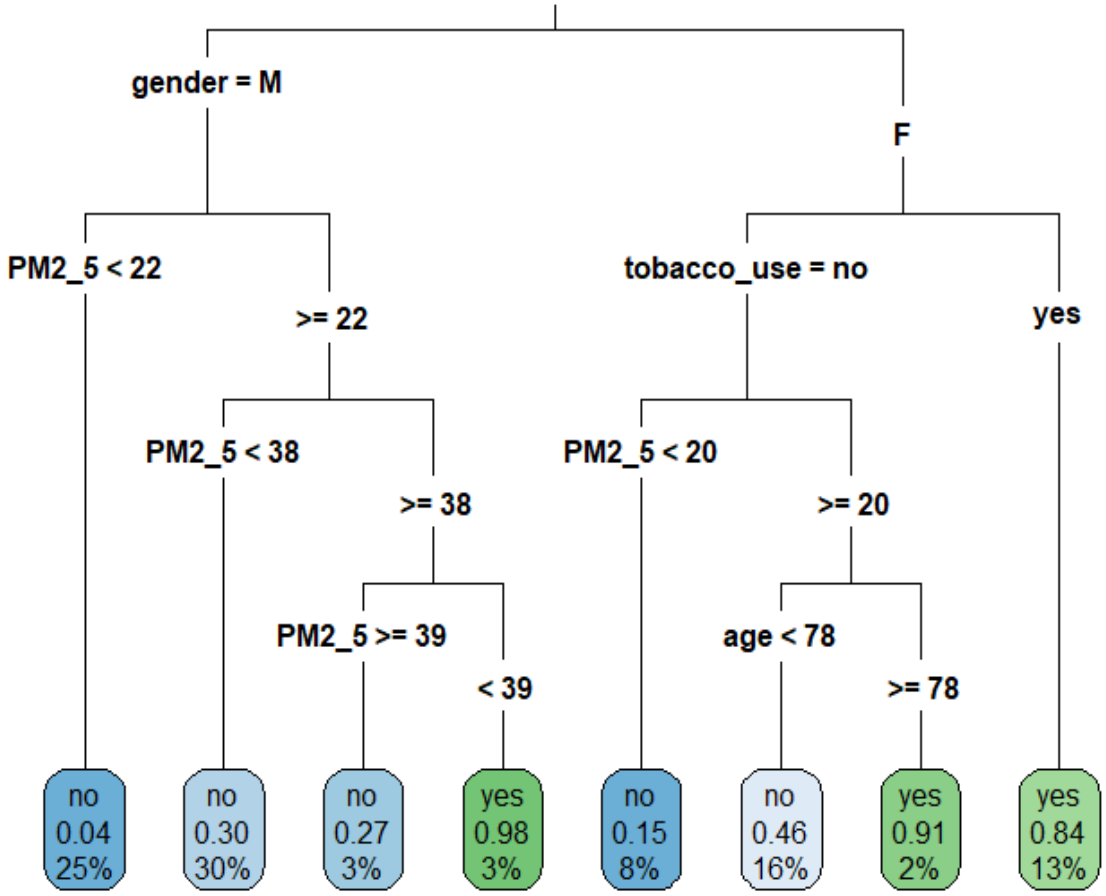
```
tp<- matrix(NA, nrow=nrow(test), ncol=99)
tn<- matrix(NA, nrow=nrow(test), ncol=99)

for (i in 1:99) {
    tp[,i]<- ifelse(test$pneumonia=="yes" & test$yes>0.01*i,1,0)
    tn[,i]<- ifelse(test$pneumonia=="no" & test$yes<=0.01*i,1,0)
}

trueclassrate<- matrix(NA, nrow=99, ncol=2)
for (i in 1:99){
    trueclassrate[i,1]<- 0.01*i
    trueclassrate[i,2]<- sum(tp[,i]+tn[,i])/nrow(test)
}

print(trueclassrate[which(trueclassrate[,2]==max(trueclassrate[,2])),])
```

```
      [,1]      [,2]
 [1,] 0.30 0.7886905
 [2,] 0.31 0.7886905
 [3,] 0.32 0.7886905
 [4,] 0.33 0.7886905
 [5,] 0.34 0.7886905
 [6,] 0.35 0.7886905
 [7,] 0.36 0.7886905
 [8,] 0.37 0.7886905
 [9,] 0.38 0.7886905
[10,] 0.39 0.7886905
[11,] 0.40 0.7886905
[12,] 0.41 0.7886905
[13,] 0.42 0.7886905
[14,] 0.43 0.7886905
[15,] 0.44 0.7886905
[16,] 0.45 0.7886905
```

The prediction accuracy for this tree is 78.86905%, which corresponds to any cutoff between 0.30 and 0.45. Next, we fit a binary tree using the entropy splitting criterion. The tree is identical to the one produced by the Gini criterion.

```
#FITTING PRUNED BINARY TREE WITH ENTROPY SPLITTING
tree.entropy<- rpart(pneumonia ~ age + gender + tobacco_use + PM2_5, data=train, method="class",
parms=list(split="entropy"), maxdepth=4)
```

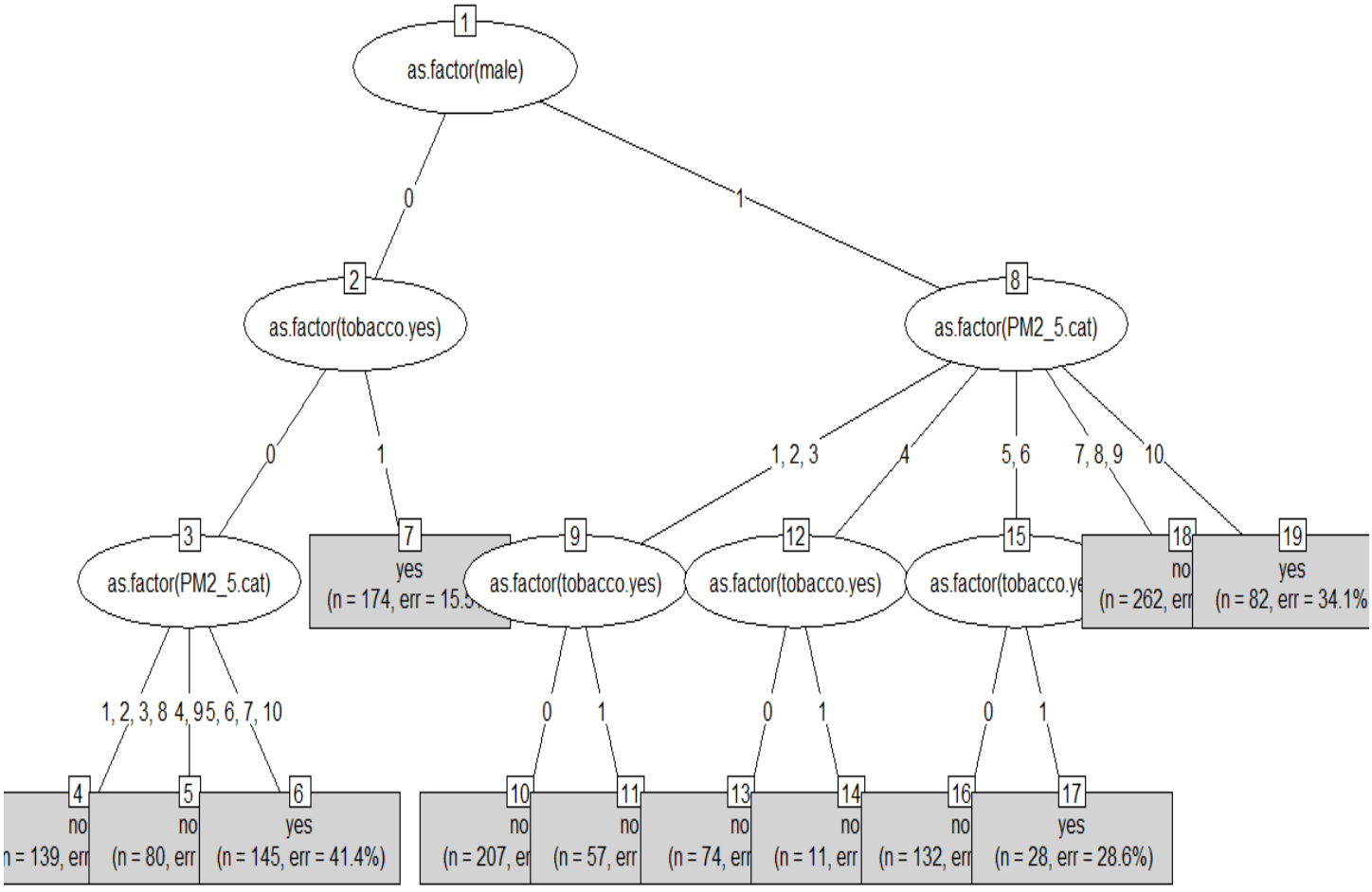
```
rpart.plot(tree.entropy, type=3)
```



Finally, we produce a binary classification tree based on the CHAID splitting criterion.

```
#FITTING PRUNED BINARY TREE WITH CHAID SPLITTING
#BINNING CONTINUOUS PREDICTOR VARIABLES
library(dplyr)

pneumonia.data<- mutate(pneumonia.data, age.cat=ntile(age,10), PM2_5.cat=ntile(PM2_5,10))

#CREATING INDICATORS FOR CATEGORICAL VARIABLES
pneumonia.data$male<- ifelse(pneumonia.data$gender=="M",1,0)
pneumonia.data$tobacco.yes<- ifelse(pneumonia.data$tobacco_use=="yes",1,0)
```

```
#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
set.seed(283605)
sample <- sample(c(TRUE, FALSE), nrow(pneumonia.data), replace=TRUE, prob=c(0.8,0.2))
train<- pneumonia.data[sample,]
test<- pneumonia.data[!sample,]

#FITTING BINARY CLASSIFICATION TREE
library(CHAID)
tree.CHAID<- chaid(as.factor(pneumonia) ~ as.factor(age.cat) + as.factor(male) + as.factor(tobacco.yes)
+ as.factor(PM2_5.cat), data=train, control=chaid_control(maxheight=3))

#PLOTTING FITTED TREE
plot(tree.CHAID, type="simple")
```




```
#COMPUTING PREDICTION ACCURACY FOR TESTING DATA
```

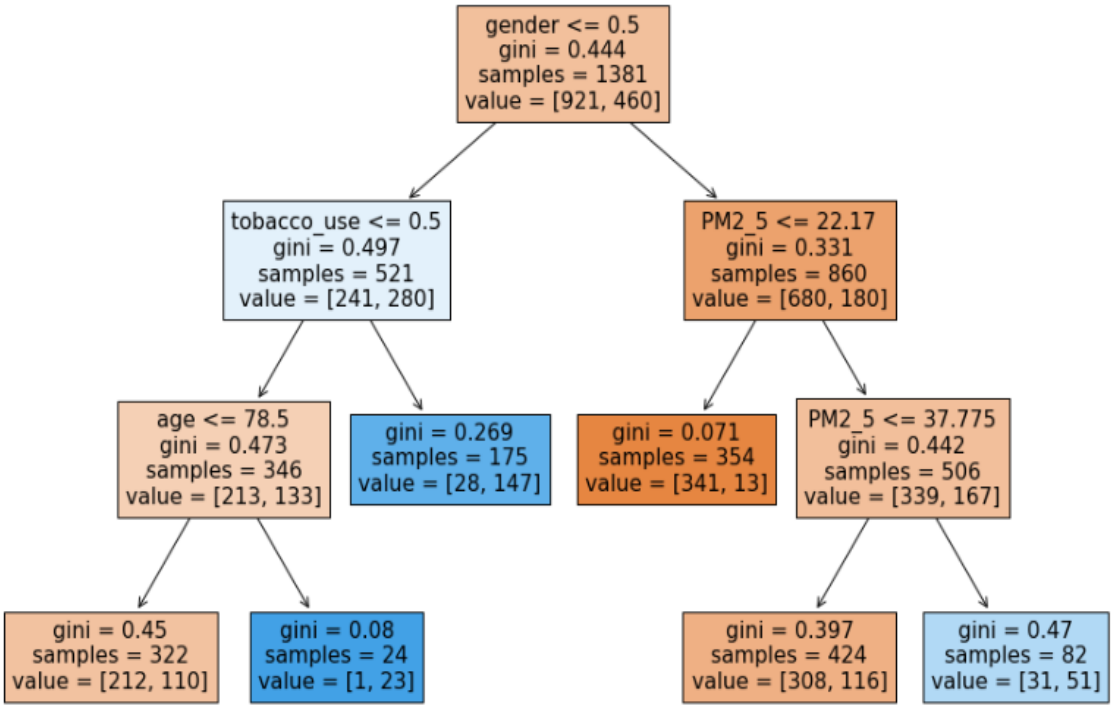
```
pred.pneumonia<- predict(tree.CHAID, newdata=test)
test<- cbind(test,pred.pneumonia)

truepred<- c()
n<- nrow(test)
for (i in 1:n)
    truepred[i]<- ifelse(test$pneumonia[i]==test$pred.pneumonia[i],1,0)

print(truepredrate<- sum(truepred)/length(truepred))
```
0.8035714

Note that the predicted values are yes/no, not probabilities. This tree gives 80.35714% correct predictions.

Next, we employ Python to build binary classification trees with the Gini and entropy slitting criteria.

In Python:

```
import numpy
import pandas
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.model_selection import train_test_split

pneumonia_data=pandas.read_csv('C:/Users/000110888/Desktop/pneumonia_data.csv')
code_gender={'M':1,'F':0}
code_tobacco_use={'yes':1,'no':0}
code_pneumonia={'yes':1,'no':0}

pneumonia_data['gender']=pneumonia_data['gender'].map(code_gender)
pneumonia_data['tobacco_use']=pneumonia_data['tobacco_use'].map(code_tobacco_use)
pneumonia_data['pneumonia']=pneumonia_data['pneumonia'].map(code_pneumonia)

X=pneumonia_data.iloc[:,0:4].values
y=pneumonia_data.iloc[:,4].values

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.20,
random_state=786756)

#FITTING BINARY TREE WITH GINI SPLITTING CRITERION
gini_tree=DecisionTreeClassifier(max_leaf_nodes=6, criterion='gini', random_state=199233)
gini_tree.fit=gini_tree.fit(X_train,y_train)

#PLOTTING FITTED TREE
fig = plt.figure(figsize=(15,10))
tree.plot_tree(gini_tree.fit, feature_names=['gender','age','tobacco_use','PM2_5'], filled=True)
```


gender <= 0.5
gini = 0.444
samples = 1381
value = [921, 460]
tobacco_use <= 0.5
gini = 0.497
samples = 521
value = [241, 280]
PM2_5 <= 22.17
gini = 0.331
samples = 860
value = [680, 180]
age <= 78.5
gini = 0.473
samples = 346
value = [213, 133]
gini = 0.269
samples = 175
value = [28, 147]
gini = 0.071
samples = 354
value = [341, 13]
PM2_5 <= 37.775
gini = 0.442
samples = 506
value = [339, 167]
gini = 0.45
samples = 322
value = [212, 110]
gini = 0.08
samples = 24
value = [1, 23]
gini = 0.397
samples = 424
value = [308, 116]
gini = 0.47
samples = 82
value = [31, 51]

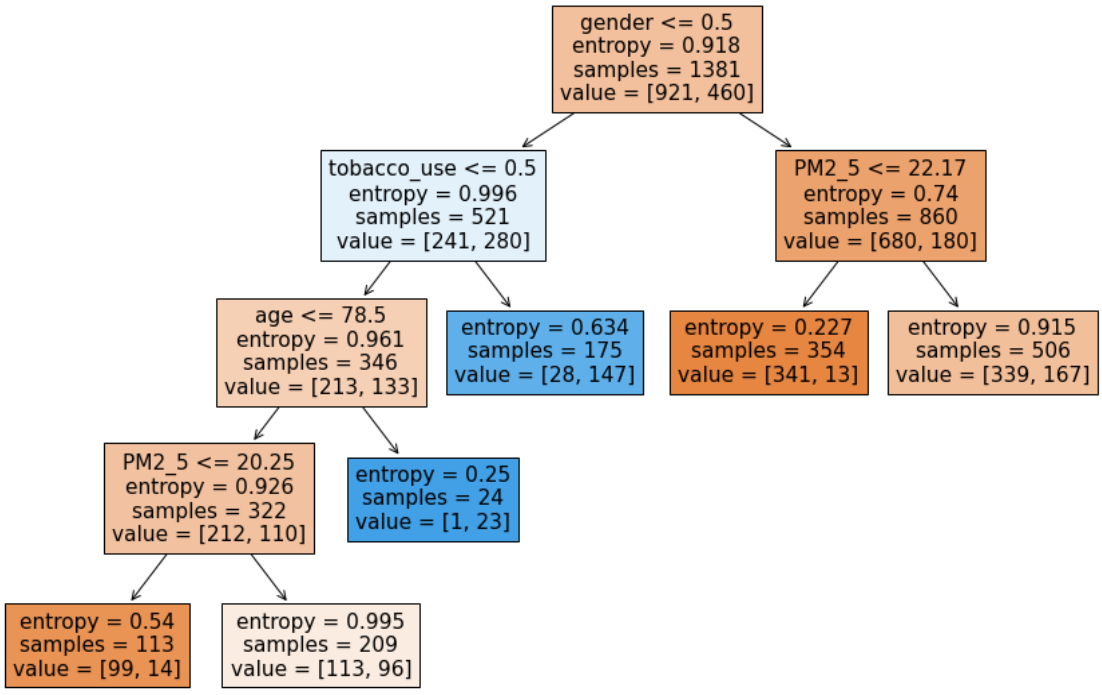
```
#COMPUTING PREDICTION ACCURACY FOR TESTING DATA

def accuracy():
    y_pred=gini_tree.predict_proba(X_test)

#y_pred[::,1] are predicted probabilities of "yes"

    total=len(y_pred)
    trueclassrate=[]
    cutoff=[]

    for i in range(99):
        tp=0
        tn=0
        cutoff.append(0.01*(i+1))
        for sub1, sub2 in zip(y_pred[::,1], y_test):
            tp_ind=1 if (sub1>0.01*(i+1) and sub2==1) else 0
            tn_ind=1 if (sub1<0.01*(i+1) and sub2==0) else 0
            tp+=tp_ind
            tn+=tn_ind
        rate=(tp+tn)/total
        trueclassrate.append(rate)

    df=pandas.DataFrame({'trueclassrate': trueclassrate,'cutoff': cutoff})
    max_rate=max(trueclassrate)
    optimal=df[df['trueclassrate']==max_rate]
    print(optimal)

accuracy()
```

```
    trueclassrate  cutoff
34       0.728324    0.35
35       0.728324    0.36
36       0.728324    0.37
37       0.728324    0.38
38       0.728324    0.39
39       0.728324    0.40
40       0.728324    0.41
41       0.728324    0.42
42       0.728324    0.43
43       0.728324    0.44
44       0.728324    0.45
45       0.728324    0.46
46       0.728324    0.47
47       0.728324    0.48
48       0.728324    0.49
49       0.728324    0.50
50       0.728324    0.51
51       0.728324    0.52
52       0.728324    0.53
```

```
53        0.728324     0.54
54        0.728324     0.55
55        0.728324     0.56
56        0.728324     0.57
57        0.728324     0.58
58        0.728324     0.59
59        0.728324     0.60
60        0.728324     0.61
61        0.728324     0.62
```

The true classification rate is 72.8324%, which corresponds to any cutoff between 0.35 and 0.62.

```
#FITTING BINARY TREE WITH ENTROPY SPLITTING CRITERION
gini_tree=DecisionTreeClassifier(max_leaf_nodes=6, criterion='entropy', random_state=199233)
gini_tree.fit=gini_tree.fit(X_train,y_train)

#PLOTTING FITTED TREE
fig = plt.figure(figsize=(15,10))
tree.plot_tree(gini_tree.fit, feature_names=['gender','age','tobacco_use','PM2_5'], filled=True)
```



```
#COMPUTING PREDICTION ACCURACY FOR TESTING DATA
accuracy()
```

```
    trueclassrate  cutoff
33       0.728324    0.34
34       0.728324    0.35
35       0.728324    0.36
36       0.728324    0.37
37       0.728324    0.38
38       0.728324    0.39
39       0.728324    0.40
40       0.728324    0.41
41       0.728324    0.42
42       0.728324    0.43
43       0.728324    0.44
44       0.728324    0.45
```

The true classification rate this three is also 72.8324%, which corresponds to any cutoff between 0.34 and 0.45.

Finally, we use `Chefboost` decision tree framework in Python to fit a binary classification tree with the CHAID splitting criterion.

```
import pandas
from sklearn.model_selection import train_test_split
from chefboost import Chefboost

pneumonia_data=pandas.read_csv('C:/Users/000110888/Desktop/pneumonia_data.csv')
code_gender={'M':1,'F':0}
code_tobacco_use={'yes':1,'no':0}

pneumonia_data['gender']=pneumonia_data['gender'].map(code_gender)
pneumonia_data['tobacco_use']=pneumonia_data['tobacco_use'].map(code_tobacco_use)

X=pneumonia_data.iloc[:,0:4].values
y=pneumonia_data.iloc[:,4].values

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.20,
random_state=786756)

X_train=pandas.DataFrame(X_train, columns=['gender','age','tobacco_use','PM2_5'])
y_train=pandas.DataFrame(y_train, columns=['pneumonia'])
train_data=pandas.concat([X_train, y_train], axis=1) #one-to-one concatenation

#FITTING BINARY TREE WITH CHAID SPLITTING ALGORITHM
config={'algorithm': 'CHAID', "max_depth": 4}
tree_chaid=Chefboost.fit(train_data, config, target_label='pneumonia')
```

The fitted tree is not plotted but stored in the `/outputs/rules/rules.py` file. Here are the decision rules for the fitted tree:

```
def findDecision(obj): #obj[0]: gender, obj[1]: age, obj[2]: tobacco_use, obj[3]: PM2_5
# {"feature": "tobacco_use", "instances": 1381, "metric_value": 28.9752, "depth": 1}
if obj[2]<=0.0:
# {"feature": "PM2_5", "instances": 1052, "metric_value": 35.0779, "depth": 2}
if obj[3]>16.381331614375146:
# {"feature": "age", "instances": 838, "metric_value": 24.3317, "depth": 3}
if obj[1]>31.85100857903719:
# {"feature": "gender", "instances": 674, "metric_value": 16.9648, "depth":
4}
if obj[0]>0.0:
return 'no'
elif obj[0]<=0.0:
return 'no'
else: return 'no'
```

```
elif obj[1]<=31.85100857903719:
# {"feature": "gender", "instances": 164, "metric_value": 14.0331, "depth":
4}
if obj[0]>0.0:
return 'no'
elif obj[0]<=0.0:
return 'no'
else: return 'no'
else: return 'no'
elif obj[3]<=16.381331614375146:
# {"feature": "age", "instances": 214, "metric_value": 25.7088, "depth": 3}
if obj[1]<=50.52336448598131:
# {"feature": "gender", "instances": 109, "metric_value": 17.447, "depth":
4}
if obj[0]>0.0:
return 'no'
elif obj[0]<=0.0:
return 'no'
else: return 'no'
elif obj[1]>50.52336448598131:
# {"feature": "gender", "instances": 105, "metric_value": 17.3185, "depth":
4}
if obj[0]>0.0:
return 'no'
elif obj[0]<=0.0:
return 'no'
else: return 'no'
else: return 'no'
else: return 'no'
elif obj[2]>0.0:
# {"feature": "gender", "instances": 329, "metric_value": 17.9638, "depth": 2}
if obj[0]<=0.0:
# {"feature": "PM2_5", "instances": 175, "metric_value": 17.984, "depth": 3}
if obj[3]>26.570514285714314:
# {"feature": "age", "instances": 89, "metric_value": 13.2291, "depth": 4}
if obj[1]>41.59550561797753:
return 'yes'
elif obj[1]<=41.59550561797753:
return 'yes'
else: return 'yes'
elif obj[3]<=26.570514285714314:
# {"feature": "age", "instances": 86, "metric_value": 11.8925, "depth": 4}
```

```
if obj[1]>43.31395348837209:
return 'yes'
elif obj[1]<=43.31395348837209:
return 'yes'
else: return 'yes'
else: return 'yes'
elif obj[0]>0.0:
# {"feature": "PM2_5", "instances": 154, "metric_value": 9.3417, "depth": 3}
if obj[3]>16.351659726119824:
# {"feature": "age", "instances": 120, "metric_value": 4.2591, "depth": 4}
if obj[1]<=62.55213873641577:
return 'no'
elif obj[1]>62.55213873641577:
return 'yes'
else: return 'yes'
elif obj[3]<=16.351659726119824:
# {"feature": "age", "instances": 34, "metric_value": 10.2831, "depth": 4}
if obj[1]>46.470588235294116:
return 'no'
elif obj[1]<=46.470588235294116:
return 'no'
else: return 'no'
else: return 'no'
else: return 'no'
else: return 'yes'
```

We use the built tree for prediction on the testing set.

```
#COMPUTING PREDICTION ACCURACY
X_test=pandas.DataFrame(X_test, columns=['gender','age','tobacco_use','PM2_5'])

y_pred=[]
for i in range(len(y_test)):
    y_pred.append(Chefboost.predict(tree_chaid, X_test.iloc[i,:]))

y_test=pandas.DataFrame(y_test,columns=['pneumonia'])
y_pred=pandas.DataFrame(y_pred,columns=['predicted'])
df=pandas.concat([y_test,y_pred],axis=1)

match=[]
for i in range(len(df)):
    if df['pneumonia'][i] == df['predicted'][i]:
        match.append(1)
    else:
        match.append(0)

trueclassrate=sum(match)/len(match)

print(trueclassrate)
```

0.7052023121387283

About 70.52% of observations in the testing set are predicted correctly by this classification tree. □

CONFUSION MATRIX

For a binary classification tree, we used the correct classification rate as a measure of model performance. Traditionally, other quantities are also used. We define them below. Suppose, hypothetically speaking, of 100 observations in a testing set, 50 yes's are predicted correctly, 10 yes's are predicted incorrectly, 35 no's are predicted correctly, and 5 no's are predicted incorrectly. We summarize this information in the following table (called **confusion matrix** or **classification matrix**):

	True "Yes"	True "No"	Total
Predicted "Yes"	50	5	55
Predicted "No"	10	35	45
Total	60	40	100

For simplicity of notation, correctly predicted yes's are called "true positive" (TP), incorrectly predicted yes's are called "false negative" (FN), correctly predicted no's are called "true negative" (TN), and incorrectly predicted no's are called "false positive" (FP).

Example. In our example, $TP = 50, FN = 10, TN = 35$, and $FP = 5$. □

The numbers of cases in these categories are used to calculate various measures of model fit:

- **Accuracy (or Correct Classification Rate):** Overall, how often is the classifier correct?

  $(TP + TN)/Total = (TP + TN)/(TP + TN + FP + FN) = (50 + 35)/100 = 0.85$

- **Misclassification Rate:** Overall, how often is it wrong?

  $(FP+FN)/Total = (FP+FN)/(TP+TN+FP+FN) = (5+10)/100 = 0.15 = 1-Accuracy$

- **True Positive Rate (or Sensitivity or Recall):** When it's actually yes, how often does it predict yes?

  $TP/True\ yes = TP/(TP + FN) = 50/60 = 0.8333$

- **False Negative Rate (FNR):** When it's actually yes, how often does it predict no?

  $FN/True\ yes = FN/(TP + FN) = 10/60 = 0.1667 = 1 - Sensitivity$

- **True Negative Rate (or Specificity):** When it's actually no, how often does it predict no?

  $TN/True\ no = TN/(FP + TN) = 35/40 = 0.875$

- **False Positive Rate (FPR):** When it's actually no, how often does it predict yes?

$$FP/True\ no = FP/(FP + TN) = 5/40 = 0.125 = 1 - Specificity$$

• **Positive Predictive Value (PPV, or Precision):** When it predicts yes, how often is it correct?

$$TP/Predicted\ yes = TP/(TP + FP) = 50/55 = 0.9091$$

• **Negative Predictive Value (NPV):** When it predicts no, how often is it correct?

$$TN/Predicted\ no = TN/(FN + TN) = 35/45 = 0.7778$$

These definitions can be presented in a theoretical confusion matrix:

	True "Yes"	True "No"	
Predicted "Yes"	True Positive (TP)	False Positive (FP)	Precision $= \frac{TP}{TP + FP}$
Predicted "No"	False Negative (FN)	True Negative (TN)	Negative Predictive Value $= \frac{TN}{FN + TN}$
	Sensitivity $= \frac{TP}{TP + FN}$, False Negative Rate = 1 − Sensitivity	Specificity $= \frac{TN}{FP + TN}$, False Positive Rate = 1 − Specificity	Accuracy $= \frac{TP + TN}{TP + TN + FP + FN}$, Misclassification Rate = 1 − Accuracy

Another performance measure that is often utilized is the F1-score. It combines recall and precision into a single measure.

• **F1-score:** It is a harmonic mean of sensitivity and precision and can be calculated as follows:

$$\text{F1-score} = \frac{2}{\frac{1}{sensitivity} + \frac{1}{precision}} = \frac{2}{\frac{TP+FN}{TP} + \frac{TP+FP}{TP}}$$
$$= \frac{2TP}{2TP + FN + FP} = \frac{(2)(50)}{(2)(50) + 10 + 5} = \frac{100}{115} = 0.869565.$$

Example. Going back to the pneumonia example, we take the binary classification tree with Gini splitting and cost-complexity pruning and compute the confusion matrix and the performance measures for the test data, using 0.5 as the cutoff.

In SAS:

```
proc import out=pneumonia datafile="./pneumonia_data.csv"
dbms=csv replace;

/*SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS*/
proc surveyselect data=pneumonia rate=0.8 seed=6132208
out=pneumonia outall method=srs;
run;

/*GINI SPLITTING AND COST-COMPLEXITY PRUNING */
proc hpsplit data=pneumonia maxdepth=4;
class pneumonia gender tobacco_use;
   model pneumonia(event="yes")= age gender tobacco_use PM2_5;
grow gini;
prune costcomplexity;
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;

/*COMPUTING CONFUSION MATRIX AND PERFORMANCE MEASURES FOR TESTING SET*/
data test;
set predicted;
if(selected="0");
tp=(P_pneumoniayes > 0.5 and pneumonia="yes");
fp=(P_pneumoniayes > 0.5 and pneumonia="no");
tn=(P_pneumoniano > 0.5 and pneumonia="no");
fn=(P_pneumoniano > 0.5 and pneumonia="yes");
run;
```

```
proc sql;
create table confusion as
select sum(tp) as tp, sum(fp) as fp, sum(tn) as tn,
sum(fn) as fn, count(*) as total
from test;
select * from confusion;
quit;
```

tp	fp	tn	fn	total
55	9	223	58	345

```
proc sql;
select (tp+tn)/total as accuracy, (fp+fn)/total as
misclassrate, tp/(tp+fn) as sensitivity,
fn/(tp+fn) as FNR, tn/(fp+tn) as specificity,
fp/(fp+tn) as FPR, tp/(tp+fp) as precision,
tn/(fn+tn) as NPV, 2*tp/(2*tp+fn+fp) as F1score
from confusion;
quit;
```

accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
0.805797	0.194203	0.486726	0.513274	0.961207	0.038793	0.859375	0.793594	0.621469

In R:

```
pneumonia.data<- read.csv(file="./pneumonia_data.csv", header=TRUE, sep=",")

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
set.seed(283605)
sample <- sample(c(TRUE, FALSE), nrow(pneumonia.data), replace=TRUE, prob=c(0.8,0.2))
train<- pneumonia.data[sample,]
test<- pneumonia.data[!sample,]
```

```
#FITTING PRUNED BINARY TREE WITH GINI SPLITTING
library(rpart)
tree.gini<- rpart(pneumonia ~ age + gender + tobacco_use + PM2_5, data=train, method="class",
parms=list(split="Gini"), maxdepth=4)

#COMPUTING PREDICTION ACCURACY FOR TESTING DATA
pred.values<- predict(tree.gini, test)
test<- cbind(test,pred.values)

tp<- c()
fp<- c()
tn<- c()
fn<- c()

total<- nrow(test)
for (i in 1:total){
    tp[i]<- ifelse(test$yes[i]>0.5 & test$pneumonia[i]=="yes",1,0)
    fp[i]<- ifelse(test$yes[i]>0.5 & test$pneumonia[i]=="no",1,0)
    tn[i]<- ifelse(test$no[i]>0.5 & test$pneumonia[i]=="no",1,0)
    fn[i]<- ifelse(test$no[i]>0.5 & test$pneumonia[i]=="yes",1,0)
}

print(tp<- sum(tp))
```

56

```
print(fp<- sum(fp))
```

8

```
print(tn<- sum(tn))
```

206

```
print(fn<- sum(fn))
```

66

```
print(total)
```

336

```
print(accuracy<- (tp+tn)/total)
```

0.7797619

```
print(misclassrate<- (fp+fn)/total)
```

0.2202381

```
print(sensitivity<- tp/(tp+fn))
```

0.4590164

```
print(FNR<- fn/(tp+fn))
```

0.5409836

```
print(specificity<- tn/(fp+tn))
```

0.9626168

```
print(FPR<- fp/(fp+tn))
```

0.03738318

```
print(precision<- tp/(tp+fp))
```

0.875

```
print(NPV<- tn/(fn+tn))
```

0.7573529

```
print(F1score<- 2*tp/(2*tp+fn+fp))
```

0.6021505

In Python:

```
import numpy
import pandas
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.model_selection import train_test_split

pneumonia_data=pandas.read_csv('C:/Users/000110888/Desktop/pneumonia_data.csv')
code_gender={'M':1,'F':0}
code_tobacco_use={'yes':1,'no':0}
code_pneumonia={'yes':1,'no':0}

pneumonia_data['gender']=pneumonia_data['gender'].map(code_gender)
pneumonia_data['tobacco_use']=pneumonia_data['tobacco_use'].map(code_tobacco_use)
pneumonia_data['pneumonia']=pneumonia_data['pneumonia'].map(code_pneumonia)

X=pneumonia_data.iloc[:,0:4].values
y=pneumonia_data.iloc[:,4].values

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.20,
random_state=786756)

#FITTING BINARY TREE WITH GINI SPLITTING CRITERION
gini_tree=DecisionTreeClassifier(max_leaf_nodes=6, criterion='gini', random_state=199233)
gini_tree.fit=gini_tree.fit(X_train,y_train)
```

```
#COMPUTING CONFUSION MATRIX AND PERFORMANCE MEASURES FOR TESTING SET
y_pred=gini_tree.predict_proba(X_test)

total=len(y_pred)

tpos=[]
fpos=[]
tneg=[]
fneg=[]

for sub1, sub2 in zip(y_pred[::,1], y_test):
    tpos.append(1) if (sub1>0.5 and sub2==1) else tpos.append(0)
    fpos.append(1) if (sub1>0.5 and sub2==0) else fpos.append(0)
    tneg.append(1) if (sub1<0.5 and sub2==0) else tneg.append(0)
    fneg.append(1) if (sub1<0.5 and sub2==1) else fneg.append(0)
    tp=sum(tpos)
    fp=sum(fpos)
    tn=sum(tneg)
    fn=sum(fneg)

print('tp:', tp)
print('fp:', fp)
print('tn:', tn)
print('fn:', fn)
print('total:', total)

accuracy=(tp+tn)/total
misclassrate=(fp+fn)/total
sensitivity=tp/(tp+fn)
FNR=fn/(tp+fn)
specificity=tn/(fp+tn)
FPR=fp/(fp+tn)
precision=tp/(tp+fp)
NPV=tn/(fn+tn)
F1score=2*tp/(2*tp+fn+fp)

print('accuracy:', accuracy)
print('misclassrate:', misclassrate)
print('sensitivity:', sensitivity)
print('FNR:', FNR)
print('specificity:', specificity)
print('FPR:', FPR)
print('precision:', precision)
print('NPV:', NPV)
print('F1score:', F1score)
```

```
tp: 63
fp: 14
tn: 189
fn: 80
total: 346
```

```
accuracy: 0.7283236994219653
misclassrate: 0.27167630057803466
sensitivity: 0.4405594405594406
FNR: 0.5594405594405595
specificity: 0.9310344827586207
FPR: 0.06896551724137931
precision: 0.8181818181818182
NPV: 0.7026022304832714
F1score: 0.5727272727272728
```

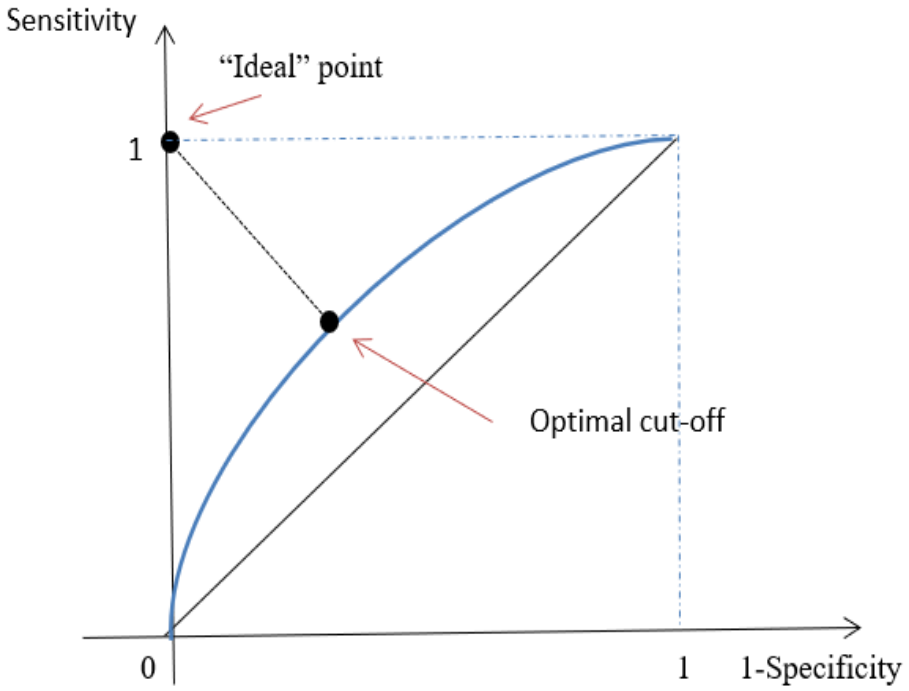
□

RECEIVER OPERATING CHARACTERISTIC (ROC) CURVE

Consider the situation of a binary classification tree. We are given actual observations (yes/no) and predicted probabilities of yes/no. Before we introduced a cut-off, a number between 0 and 1, such that if the predicted probability of "yes" is above the cut-off, then we assume that the predicted value is "yes". We used cut-offs ranging between 0.01 and 0.99 with a step of 0.01 to choose the optimal cutoff that maximizes the true (correct) classification rate (equivalently, minimizes, the misclassification rate). In the previous section, we computed the performance measures assuming the cut-off is 0.5.

A more sophisticated approach relies on the Receiver Operating Characteristic (ROC) curve, schematically presented in the figure below. A ROC curve is a plot of sensitivity (true positive rate) against 1-specificity (false positive rate) for different cut-off points. The cut-off points are often termed *classification thresholds*. A ROC curve connects the origin $(0, 0)$ and the point $(1, 1)$ as a curve that lies above the bisector. Note that the bisector represents a segment on which both sensitivity and specificity are equal to 0.5, corresponding to a random guess.

An ROC curve shows a trade-off between sensitivity and specificity, that is, an increase in sensitivity is accompanied by a decrease in specificity. These two quantities are known to work reciprocally.

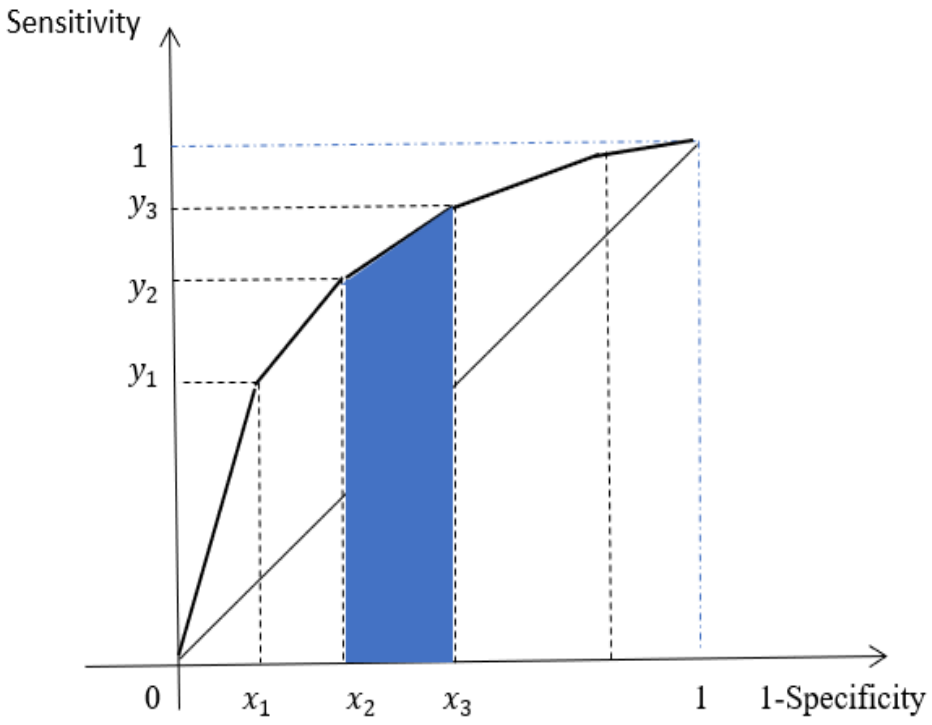


Define the point (0,1) as the "ideal" point. Indeed, at this point sensitivity (true positive rate) and specificity (true negative rate) would both be equal to one, which is clearly a hypothetical situation. Nonetheless, the point on the ROC curve closest to this"ideal" point gives the optimal cut-off for the binary classification.

Area Under the ROC Curve

Often in practice another model performance measure is computed. It is the **area under the curve (AUC)** (or **area under the ROC curve (AUROC)**). The larger the area under the ROC curve, the further the curve is from the bisector, and thus a higher value of AUC indicates a better overall fit of the model (for any classification threshold).

In theory, the ROC curve is smooth, and computing AUC would involve calculus. In practice, however, the cut-offs are chosen on a discrete scale, so the fitted ROC curve is piece-wise linear, and thus, the area can be computed as the sum of areas of the trapezoids (see the figure below). The height of each trapezoid is the distance between two distinct consecutive values of $1 - Specificity$, and the top and bottom sides are distinct consecutive values of $Sensitivity$. Hence, the formula for the area of one trapezoid is $\Big((1 - Specificity)_2 - (1 - Specificity)_1\Big)\Big(Sensivity_1 + Sensitivity_2\Big)/2$.

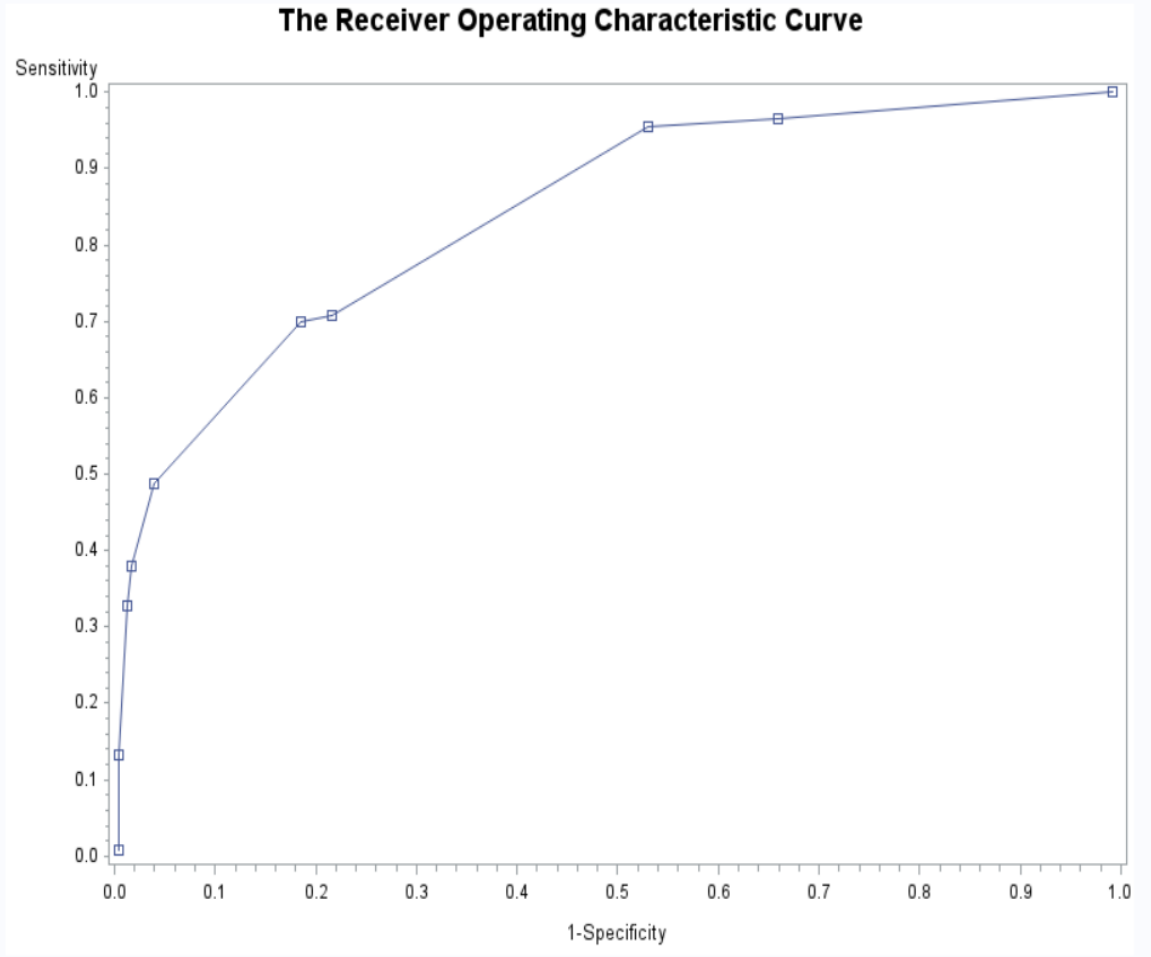

Sensitivity
1
y_3
y_2
y_1
0
x_1
x_2
x_3
1
1-Specificity

Example. Consider the binary classification tree with the Gini splitting algorithm and cost-complexity pruning. We plot the ROC curve and find the optimal cut-off that corresponds to the minimal distance to the "ideal" point.

In SAS:

```
proc import out=pneumonia datafile="./pneumonia_data.csv"
dbms=csv replace;

/*SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS*/
proc surveyselect data=pneumonia rate=0.8 seed=6132208
out=pneumonia outall method=srs;
run;

/*GINI SPLITTING AND COST-COMPLEXITY PRUNING */
proc hpsplit data=pneumonia maxdepth=4;
class pneumonia gender tobacco_use;
   model pneumonia(event="yes")= age gender tobacco_use PM2_5;
grow gini;
prune costcomplexity;
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;

/*COMPUTING CONFUSION MATRICES AND PERFORMANCE MEASURES
FOR TESTING SET FOR A RANGE OF CUTOFFS*/
data test;
set predicted;
if(selected="0");
run;

data cutoffs;
set test;
do i=0 to 101;
tp=(P_pneumoniayes >= 0.01*i and pneumonia="yes");
fp=(P_pneumoniayes >= 0.01*i and pneumonia="no");
tn=(P_pneumoniayes <  0.01*i and pneumonia="no");
fn=(P_pneumoniayes <  0.01*i and pneumonia="yes");
output;
end;
run;
```

```
proc sql;
create table confusion as
select i, sum(tp) as tp, sum(fp) as fp, sum(tn) as tn,
sum(fn) as fn, count(*) as total
from cutoffs
group by i;
quit;

proc sql;
create table measures as
select i, (tp+tn)/total as accuracy, (fp+fn)/total as
misclassrate, tp/(tp+fn) as sensitivity, tn/(fp+tn) as specificity,
fp/(fp+tn) as oneminusspec
from confusion
group by i;
quit;

/*PLOTTING ROC CURVE*/
title 'The Receiver Operating Characteristic Curve';
proc gplot data=measures;
symbol v=square interpol=join;
plot sensitivity*oneminusspec/ vaxis=0 to 1 by 0.1 haxis=0 to 1 by 0.1;
label sensitivity="Sensitivity" oneminusspec="1-Specificity";
run;
```




```
/*REPORTING MEASURES FOR THE POINT ON ROC CURVE CLOSEST TO THE IDEAL POINT (0,1)*/
proc sql;
select accuracy, misclassrate, sensitivity, specificity,
sqrt(oneminusspec**2+(1-sensitivity)**2) as distance, i*0.01 as cutoff
from measures
having distance=min(distance);
quit;
```

accuracy	misclassrate	sensitivity	specificity	distance	cutoff
0.776812	0.223188	0.699115	0.814655	0.35339	0.31
0.776812	0.223188	0.699115	0.814655	0.35339	0.32
0.776812	0.223188	0.699115	0.814655	0.35339	0.33
0.776812	0.223188	0.699115	0.814655	0.35339	0.34
0.776812	0.223188	0.699115	0.814655	0.35339	0.35
0.776812	0.223188	0.699115	0.814655	0.35339	0.36
0.776812	0.223188	0.699115	0.814655	0.35339	0.37
0.776812	0.223188	0.699115	0.814655	0.35339	0.38
0.776812	0.223188	0.699115	0.814655	0.35339	0.39
0.776812	0.223188	0.699115	0.814655	0.35339	0.4
0.776812	0.223188	0.699115	0.814655	0.35339	0.41
0.776812	0.223188	0.699115	0.814655	0.35339	0.42
0.776812	0.223188	0.699115	0.814655	0.35339	0.43
0.776812	0.223188	0.699115	0.814655	0.35339	0.44
0.776812	0.223188	0.699115	0.814655	0.35339	0.45
0.776812	0.223188	0.699115	0.814655	0.35339	0.46
0.776812	0.223188	0.699115	0.814655	0.35339	0.47
0.776812	0.223188	0.699115	0.814655	0.35339	0.48

From this output, we see that any cut-off between 0.31 and 0.48 gives the minimal distance between the ROC curve and the "ideal" point.

```
/*COMPUTING AREA UNDER THE ROC CURVE*/
proc sort data=measures;
by oneminusspec;
run;

data AUC;
set measures;
lagx=lag(oneminusspec);
lagy=lag(sensitivity);
if lagx=. then lagx=0;
if lagy=. then lagy=0;
trapezoid=(oneminusspec-lagx)*(sensitivity+lagy)/2;
AUC+trapezoid;
run;
```

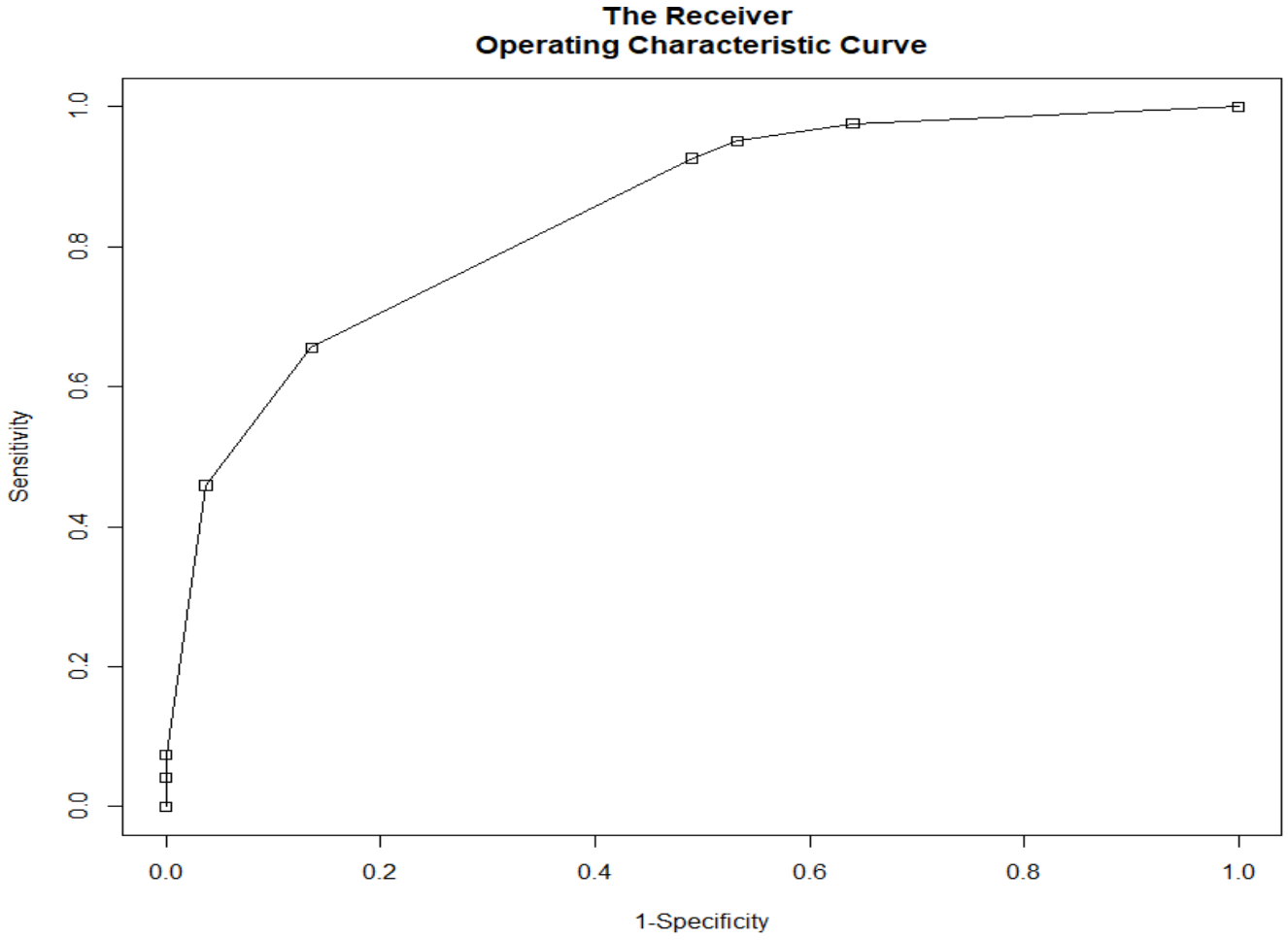
```
proc print data=AUC (firstobs=102) noobs;
var AUC;
run;
```

AUC
0.84082

In R:

```
pneumonia.data<- read.csv(file="./pneumonia_data.csv", header=TRUE, sep=",")

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
set.seed(283605)
sample <- sample(c(TRUE, FALSE), nrow(pneumonia.data), replace=TRUE, prob=c(0.8,0.2))
train<- pneumonia.data[sample,]
test<- pneumonia.data[!sample,]

#FITTING PRUNED BINARY TREE WITH GINI SPLITTING
library(rpart)
tree.gini<- rpart(pneumonia ~ age + gender + tobacco_use + PM2_5, data=train, method="class",
parms=list(split="Gini"), maxdepth=4)

#COMPUTING CONFUSION MATRICES AND PERFORMANCE MEASURES FOR TEST-
ING DATA
#for a range of cut-offs
pred.values<- predict(tree.gini, test)
test<- cbind(test,pred.values)

tpos<- matrix(NA, nrow=nrow(test), ncol=102)
fpos<- matrix(NA, nrow=nrow(test), ncol=102)
tneg<- matrix(NA, nrow=nrow(test), ncol=102)
fneg<- matrix(NA, nrow=nrow(test), ncol=102)

for (i in 0:101) {
    tpos[,i+1]<- ifelse(test$pneumonia=="yes" & test$yes>=0.01*i,1,0)
    fpos[,i+1]<- ifelse(test$pneumonia=="no" & test$yes>=0.01*i, 1,0)
    tneg[,i+1]<- ifelse(test$pneumonia=="no" & test$yes<0.01*i,1,0)
    fneg[,i+1]<- ifelse(test$pneumonia=="yes" & test$yes<0.01*i,1,0)
```

```
}

tp<- c()
fp<- c()
tn<- c()
fn<- c()
accuracy<- c()
misclassrate<- c()
sensitivity<- c()
specificity<- c()
oneminusspec<- c()
cutoff<- c()

for (i in 1:102) {
tp[i]<- sum(tpos[,i])
fp[i]<- sum(fpos[,i])
tn[i]<- sum(tneg[,i])
fn[i]<- sum(fneg[,i])
total<- nrow(test)
accuracy[i]<- (tp[i]+tn[i])/total
misclassrate[i]<- (fp[i]+fn[i])/total
sensitivity[i]<- tp[i]/(tp[i]+fn[i])
specificity[i]<- tn[i]/(fp[i]+tn[i])
oneminusspec[i]<- fp[i]/(fp[i]+tn[i])
cutoff[i]<- 0.01*(i-1)
}

#PLOTTING ROC CURVE
plot(oneminusspec, sensitivity, type="l", lty=1, main="The Receiver Operating Characteristic
Curve", xlab="1-Specificity", ylab="Sensitivity")
points(oneminusspec, sensitivity, pch=0) #pch=plot character, 0=square
```




```
#REPORTING MEASURES FOR THE POINT ON ROC CURVE CLOSEST TO THE IDEAL
POINT (0,1)
distance<- c()
for (i in 1:102)
    distance[i]<- sqrt(oneminusspec[i]^2+(1-sensitivity[i])^2)

measures<- cbind(accuracy, misclassrate, sensitivity, specificity, distance, cutoff)
min.dist<- min(distance)
print(measures[which(measures[,5]==min.dist),])
```

```
 accuracy misclassrate sensitivity specificity  distance cutoff
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.30
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.31
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.32
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.33
```

```
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.34
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.35
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.36
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.37
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.38
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.39
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.40
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.41
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.42
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.43
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.44
0.7886905    0.2113095   0.6557377    0.864486 0.3699738   0.45
```

We can see that for the binary tree built in R, any cut-off between 0.30 and 0.45 is an optimal one.

```
#COMPUTING AREA UNDER THE ROC CURVE
sensitivity<- sensitivity[order(sensitivity)]
oneminusspec<- oneminusspec[order(oneminusspec)]

library(Hmisc) #Harrell Miscellaneous packages
lagx<- Lag(oneminusspec,shift=1)
lagy<- Lag(sensitivity, shift=1)
lagx[is.na(lagx)]<- 0
lagy[is.na(lagy)]<- 0
trapezoid<- (oneminusspec-lagx)*(sensitivity+lagy)/2
print(AUC<- sum(trapezoid))
```

0.8439367

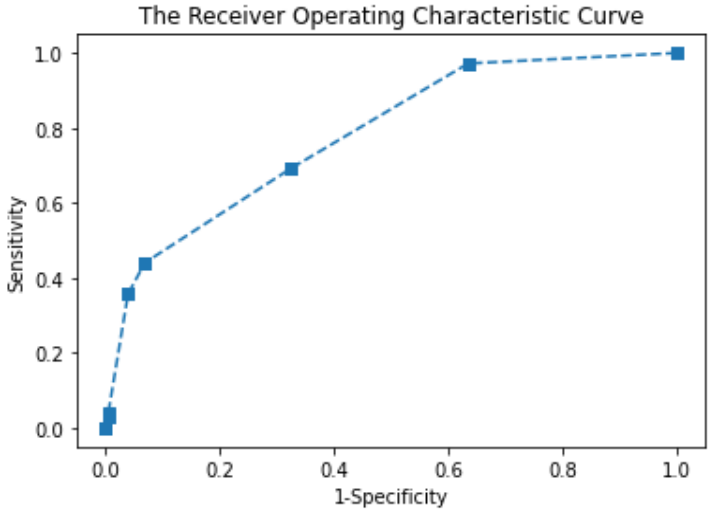
In Python:

```
import numpy
import pandas
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.model_selection import train_test_split

pneumonia_data=pandas.read_csv('C:/Users/000110888/Desktop/pneumonia_data.csv')
code_gender={'M':1,'F':0}
code_tobacco_use={'yes':1,'no':0}
code_pneumonia={'yes':1,'no':0}

pneumonia_data['gender']=pneumonia_data['gender'].map(code_gender)
pneumonia_data['tobacco_use']=pneumonia_data['tobacco_use'].map(code_tobacco_use)
pneumonia_data['pneumonia']=pneumonia_data['pneumonia'].map(code_pneumonia)

X=pneumonia_data.iloc[:,0:4].values
y=pneumonia_data.iloc[:,4].values

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.20, random_state=786756)

#FITTING BINARY TREE WITH GINI SPLITTING CRITERION
gini_tree=DecisionTreeClassifier(max_leaf_nodes=6, criterion='gini', random_state=199233)
gini_tree.fit=gini_tree.fit(X_train,y_train)
```

```
#COMPUTING CONFUSION MATRICES AND PERFORMANCE MEASURES FOR TESTING SET FOR A RANGE OF CUTOFFS
y_pred=gini_tree.predict_proba(X_test)

total=len(y_pred)

cutoff=[]
accuracy=[]
misclassrate=[]
sensitivity=[]
specificity=[]
oneminusspec=[]
distance=[]

for i in range(99):
    tp=0
    fp=0
    tn=0
    fn=0
    cutoff.append(0.01*(i+1))
    for sub1, sub2 in zip(y_pred[::,1], y_test):
        tp_ind=1 if (sub1>0.01*(i+1) and sub2==1) else 0
        fp_ind=1 if (sub1>0.01*(i+1) and sub2==0) else 0
        tn_ind=1 if (sub1<0.01*(i+1) and sub2==0) else 0
        fn_ind=1 if (sub1<0.01*(i+1) and sub2==1) else 0
        tp+=tp_ind
        fp+=fp_ind
        tn+=tn_ind
        fn+=fn_ind
```

```
    accuracy_i=(tp+tn)/total
    misclassrate_i=(fp+fn)/total
    sensitivity_i=tp/(tp+fn)
    specificity_i=tn/(fp+tn)
    oneminusspec_i=fp/(fp+tn)
    distance_i=numpy.sqrt(pow(oneminusspec_i,2)+pow(1-sensitivity_i,2))

    accuracy.append(accuracy_i)
    misclassrate.append(misclassrate_i)
    sensitivity.append(sensitivity_i)
    specificity.append(specificity_i)
    oneminusspec.append(oneminusspec_i)
    distance.append(distance_i)

#PLOTTING ROC CURVE
import matplotlib.pyplot as plot
plt.plot(oneminusspec, sensitivity, linestyle='--', marker='s')
plt.title('The Receiver Operating Characteristic Curve')
plt.xlabel('1-Specificity')
plt.ylabel('Sensitivity')

#REPORTING MEASURES FOR THE POINT ON ROC CURVE CLOSEST TO THE IDEAL POINT (0,1)
df=pandas.DataFrame({'accuracy': accuracy,'misclassrate': misclassrate,'sensitivity':
sensitivity,'specificity': specificity, 'oneminusspec': oneminusspec,'distance': distance,'cut-off': cutoff})
min_distance=min(distance)
optimal=df[df['distance']==min_distance]
print(optimal)

#COMPUTING AREA UNDER THE ROC CURVE
df=df.sort_values('oneminusspec', ascending=True)
df['lagx']=df['oneminusspec'].shift(1)
df['lagy']=df['sensitivity'].shift(1)
df['lagx']=numpy.nan_to_num(df['lagx'],nan=0)
df['lagy']=numpy.nan_to_num(df['lagy'],nan=0)
df['trapezoid']=(df['oneminusspec']-df['lagx'])*(df['sensitivity']+df['lagy'])/2;
AUC=sum(df['trapezoid'])
print(AUC)
```

```
    accuracy  misclassrate  sensitivity  specificity  oneminusspec  distance
27  0.682081      0.317919     0.692308     0.674877      0.325123  0.447638
28  0.682081      0.317919     0.692308     0.674877      0.325123  0.447638
29  0.682081      0.317919     0.692308     0.674877      0.325123  0.447638
30  0.682081      0.317919     0.692308     0.674877      0.325123  0.447638
31  0.682081      0.317919     0.692308     0.674877      0.325123  0.447638
32  0.682081      0.317919     0.692308     0.674877      0.325123  0.447638
33  0.682081      0.317919     0.692308     0.674877      0.325123  0.447638

    cut-off
27     0.28
28     0.29
29     0.30
30     0.31
31     0.32
32     0.33
33     0.34
0.7814776561697745
```



From this output, we see that any cut-off between 0.28 and 0.34 gives the minimal distance between the ROC curve and the "ideal" point. The area under the ROC curve is about 0.78. □

MULTINOMIAL CLASSIFICATION TREE

A **multi-class** (or **multinomial**) **classification tree** classifies observations into one of three or more classes. The same algorithms as for the binary classification tree apply to this case as well.

The output for prediction contains predicted probabilities of each of the multiple classes. We assume that the class with the highest predicted probability is the class actually predicted by the fitted model.

Performance Measures for Individual Classes

The performance measures used for binary classification can be extended to a multi-class classification. Unlike binary classification, there are no positive or negative classes but we can compute TP, TN, FP, and FN for each individual class.

For example, in a data set, there are 50 greens, 40 blues, and 10 reds. The predicted colors are summarized in the following confusion matrix:

True color	Predicted Color		
	green	blue	red
green	35	5	10
blue	5	30	5
red	3	2	5

We consider each color separately and compute all the performance measures. For the green color, the 2-by-2 confusion matrix has the form:

True color	Predicted Color	
	green	not green
green	35	15
not green	8	42

Now we compute the performance measures for this confusion matrix:

$$TP = 35, \;\; TN = 42, \;\; FP = 8, \;\; FN = 15,$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = (35 + 42)/100 = 0.77,$$

$$\text{Misclassification Rate} = \frac{FP + FN}{TP + TN + FP + FN} = 1 - \text{Accuracy} = 0.23,$$

$$\text{Sensitivity} = \frac{TP}{TP + FN} = 35/(35 + 15) = 0.70,$$

$$\text{FNR} = \frac{FN}{TP + FN} = 1 - \text{Sensitivity} = 0.30,$$

$$\text{Specificity} = \frac{TN}{FP + TN} = 42/(8 + 42) = 0.84,$$

$$\text{FPR} = \frac{FP}{FP + TN} = 1 - \text{Specificity} = 0.16,$$

$$\text{Precision} = \frac{TP}{TP+FP} = 35/(35+8) = 0.813953,$$
$$\text{NPV} = \frac{TN}{FN+TN} = 42/(15+42) = 0.736842,$$
$$\text{F1-score} = \frac{2TP}{2TP+FN+FP} = \frac{(2)(35)}{(2)(35)+15+8} = 0.752688.$$

For the blue color, the 2-by-2 confusion matrix is

True color	Predicted Color	
	blue	not blue
blue	30	10
not blue	7	53

The performance measures for this binary classification are

$$TP = 30, \;\; TN = 53, \;\; FP = 7, \;\; FN = 10,$$
$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} = (30+53)/100 = 0.83,$$
$$\text{Misclassification Rate} = \frac{FP+FN}{TP+TN+FP+FN} = 1 - \text{Accuracy} = 0.17,$$
$$\text{Sensitivity} = \frac{TP}{TP+FN} = 30/(30+10) = 0.75,$$
$$\text{FNR} = \frac{FN}{TP+FN} = 1 - \text{Sensitivity} = 0.25,$$
$$\text{Specificity} = \frac{TN}{FP+TN} = 53/(7+53) = 0.95,$$
$$\text{FPR} = \frac{FP}{FP+TN} = 1 - \text{Specificity} = 0.05,$$
$$\text{Precision} = \frac{TP}{TP+FP} = 30/(30+7) = 0.810811,$$
$$\text{NPV} = \frac{TN}{FN+TN} = 53/(10+53) = 0.84127,$$
$$\text{F1-score} = \frac{2TP}{2TP+FN+FP} = \frac{(2)(30)}{(2)(30)+10+7} = 0.779221.$$

Finally, for the red color, the 2-by-2 confusion matrix is

True color	Predicted Color	
	red	not red
red	5	5
not red	15	75

Now we compute the performance measures for this confusion matrix

$$TP = 5, \;\; TN = 75, \;\; FP = 15, \;\; FN = 5,$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = (5 + 75)/100 = 0.80,$$

$$\text{Misclassification Rate} = \frac{FP + FN}{TP + TN + FP + FN} = 1 - \text{Accuracy} = 0.20,$$

$$\text{Sensitivity} = \frac{TP}{TP + FN} = 5/(5 + 5) = 0.50,$$

$$\text{FNR} = \frac{FN}{TP + FN} = 1 - \text{Sensitivity} = 0.50,$$

$$\text{Specificity} = \frac{TN}{FP + TN} = 75/(15 + 75) = 0.8333,$$

$$\text{FPR} = \frac{FP}{FP + TN} = 1 - \text{Specificity} = 0.1667,$$

$$\text{Precision} = \frac{TP}{TP + FP} = 5/(5 + 15) = 0.25,$$

$$\text{NPV} = TN/(FN + TN) = 75/(5 + 75) = 0.9375,$$

$$\text{F1-score} = \frac{2TP}{2TP + FN + FP} = \frac{(2)(5)}{(2)(5) + 5 + 15} = 0.3333.$$

Micro Measures

Turning now to the multinomial model, we compute performance measures based on total $TP = 35 + 30 + 5 = 70$, total $TN = 42 + 53 + 75 = 170$, total $FP = 8 + 7 + 15 = 30$, and total $FN = 15 + 10 + 5 = 30$. These are global measures for the whole model, called **micro-averaged** measures. The micro-averaged measures are:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = (70 + 170)/300 = 0.80,$$

$$\text{Misclassification Rate} = 1 - \text{Accuracy} = 0.20,$$

$$\text{Sensitivity} = \frac{TP}{TP + FN} = 70/(70 + 30) = 0.70,$$

$$\text{FNR} = 1 - \text{Sensitivity} = 0.30,$$

$$\text{Specificity} = \frac{TN}{FP + TN} = 170/(30 + 170) = 0.85,$$

$$\text{FPR} = 1 - \text{Specificity} = 0.15,$$

$$\text{Precision} = \frac{TP}{TP + FP} = 70/(70 + 30) = 0.70,$$
$$\text{NPV} = \frac{TN}{FN + TN} = 170/(30 + 170) = 0.85,$$
$$\text{F1-score} = \frac{2TP}{2TP + FN + FP} = \frac{(2)(70)}{(2)(70) + 30 + 30} = 0.70.$$

Macro Measures

Macro measures are computed as an arithmetic average of individual measures for each class. The macro measures are:

$$\text{Accuracy} = (0.77 + 0.83 + 0.80)/3 = 0.80,$$
$$\text{Misclassification Rate} = 1 - \text{Accuracy} = 0.20,$$
$$\text{Sensitivity} = (0.70 + 0.75 + 0.50)/3 = 0.65,$$
$$\text{FNR} = 1 - \text{Sensitivity} = 0.35,$$
$$\text{Specificity} = (0.84 + 0.95 + 0.8333)/3 = 0.87,$$
$$\text{FPR} = 1 - \text{Specificity} = 0.13,$$
$$\text{Precision} = (0.813953 + 0.810811 + 0.25)/3 = 0.624921,$$
$$\text{NPV} = (0.736842 + 0.84127 + 0.9375)/3 = 0.838537,$$
$$\text{F1-score} = (0.752688 + 0.779221 + 0.3333)/3 = 0.621736.$$

Weighted Macro Measures

The **weighted macro** measures are weighted means of the measures for individual classes, where weights are proportional to the total number of samples in the class. There are 50 greens, 40 blues, and 10 reds, therefore, the weights are 50/(50+40+10)=0.5 for greens, 40/100=0.4 for blues, and 10/100=0.1 for reds. The weighted macro measures are computed as:

$$\text{Accuracy} = (0.77)(0.5) + (0.83)(0.4) + (0.80)(0.1) = 0.797,$$
$$\text{Misclassification Rate} = 1 - \text{Accuracy} = 0.203,$$
$$\text{Sensitivity} = (0.70)(0.5) + (0.75)(0.4) + (0.50)(0.1) = 0.70,$$
$$\text{FNR} = 1 - \text{Sensitivity} = 0.30,$$

$$\text{Specificity} = (0.84)(0.5) + (0.95)(0.4) + (0.8333)(0.1) = 0.88333,$$

$$\text{FPR} = 1 - \text{Specificity} = 0.11667,$$

$$\text{Precision} = (0.813953)(0.5) + (0.810811)(0.4) + (0.25)(0.1) = 0.756301,$$

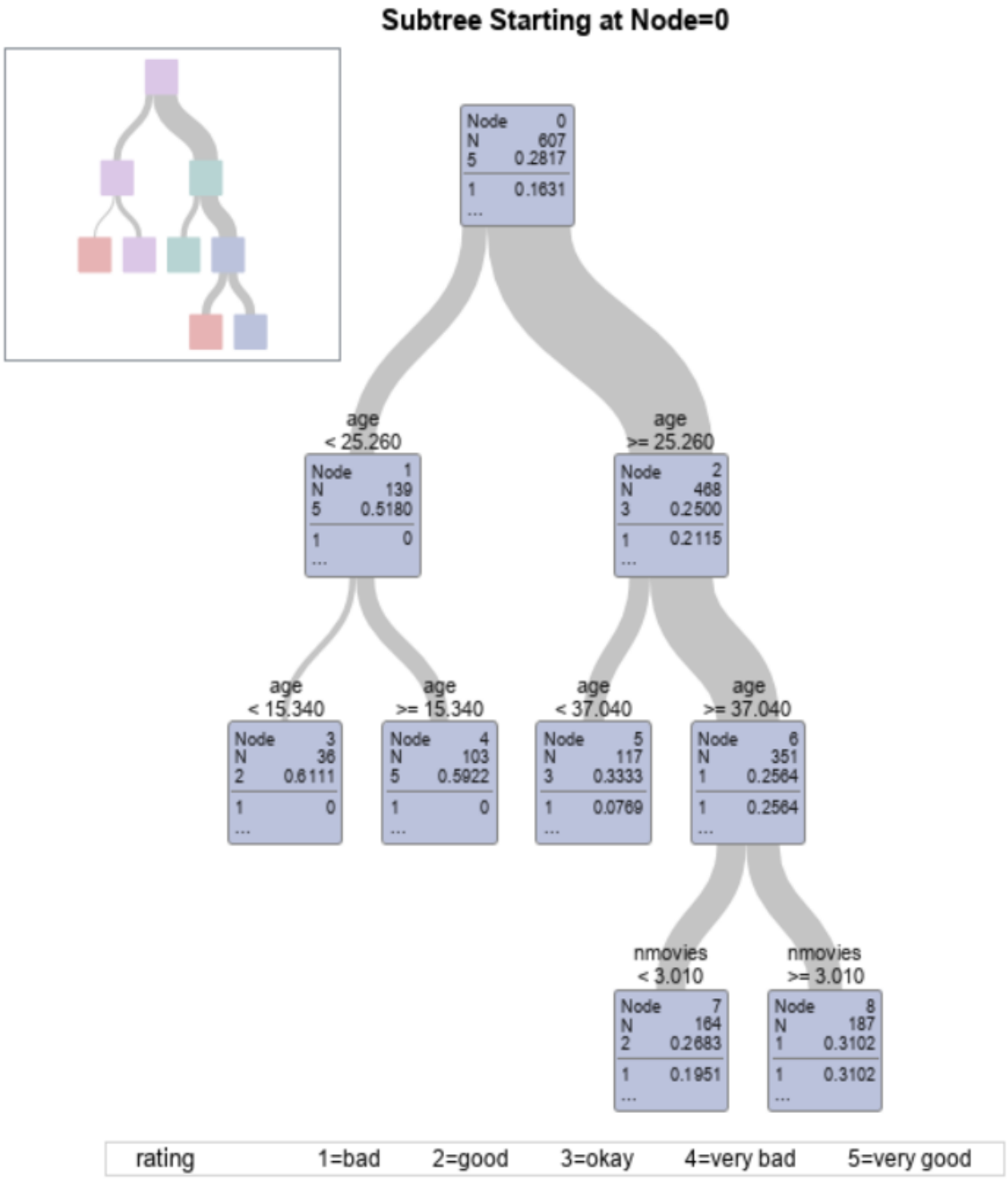
$$\text{NPV} = (0.736842)(0.5) + (0.84127)(0.4) + (0.9375)(0.1) = 0.798679,$$

$$\text{F1-score} = (0.752688)(0.5) + (0.779221)(0.4) + (0.3333)(0.1) = 0.721362.$$

Example. The data set "movie_data.csv" contains data on movie-goers (age, gender, membership, and the number of movies watched in the previous 4 weeks), and their rating of a new movie (very bad/bad/okay/good/very good). There are 758 rows in this data set. We split the data into 80% training and 20% testing sets, and fit a multinomial classification tree using Gini, entropy, and CHAID splitting criteria and the cost-complexity pruning algorithm. We then compute performance measures for individual classes, micro measures, macro measures, and weighted macro measures, and compare the three models based on these measures.

In SAS:

```
proc import out=movie
datafile="./movie_data.csv" dbms=csv replace;
run;

/*SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS*/
proc surveyselect data=movie rate=0.8 seed=118607
out=movie outall method=srs;
run;

/*GINI SPLITTING AND COST-COMPLEXITY PRUNING */
proc hpsplit data=movie;
class rating gender member;
   model rating=age gender member nmovies;
grow gini;
prune costcomplexity;
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;
```




```
/*MACRO FOR COMPUTING PERFORMANCE MEASURES*/
%macro perf_measures(dataset);

/*computing confusion matrix*/
data test;
```

```
set predicted;
if(selected="0");
maxprob=max(P_ratingbad, P_ratinggood, P_ratingokay,
P_ratingvery_bad, P_ratingvery_good);
if maxprob=P_ratingvery_good then predclass='very good';
if maxprob=P_ratinggood then predclass='good';
if maxprob=P_ratingokay then predclass='okay';
if maxprob=P_ratingbad then predclass='bad';
if maxprob=P_ratingvery_bad then predclass='very bad';
run;

/*computing total number of rows in test set*/
proc sql;
create table totalrows as
select count(*) as nrows
from test;
quit;

data _null_;
set totalrows;
 call symput('totalrows', nrows);
run;

/*computing performance measures for individual classes*/
%macro class_metrics(class);
data indiv_class;
set test;
tp=(predclass=&class and rating=&class);
fp=(predclass=&class and rating ne &class);
tn=(predclass ne &class and rating ne &class);
fn=(predclass ne &class and rating=&class);
run;

proc sql;
create table confusion as
select sum(tp) as tp, sum(fp) as fp, sum(tn) as tn,
sum(fn) as fn, count(*) as total
from indiv_class;
quit;

proc sql;
create table measures as
```

```
select &class as class, tp, fp, tn, fn,
(tp+tn)/total as accuracy, (fp+fn)/total as
misclassrate, tp/(tp+fn) as sensitivity,
fn/(tp+fn) as FNR, tn/(fp+tn) as specificity,
fp/(fp+tn) as FPR, tp/(tp+fp) as precision,
tn/(fn+tn) as NPV, 2*tp/(2*tp+fn+fp) as F1score
from confusion;
select * from measures;
quit;

proc append base=&dataset data=measures;
run;

%mend;

%class_metrics('very bad')
%class_metrics('bad')
%class_metrics('okay')
%class_metrics('good')
%class_metrics('very good')

/*computing micro measures*/
proc sql;
create table totals as
select sum(tp) as tp, sum(fp) as fp, sum(tn) as tn,
sum(fn) as fn
from &dataset;
quit;

proc sql;
select 'micro measures',(tp+tn)/(tp+fp+tn+fn)
as accuracy, (fp+fn)/(tp+fp+tn+fn)
as misclassrate, tp/(tp+fn) as sensitivity,
fn/(tp+fn) as FNR, tn/(fp+tn) as specificity,
fp/(fp+tn) as FPR, tp/(tp+fp) as precision,
tn/(fn+tn) as NPV, 2*tp/(2*tp+fn+fp) as F1score
from totals;
quit;

/*computing macro measures*/
proc sql;
select 'macro measures', mean(accuracy) as accuracy,
```

```
mean(misclassrate) as misclassrate, mean(sensitivity) as sensitivity,
mean(FNR) as FNR, mean(specificity) as specificity,
mean(FPR) as FPR, mean(precision) as precision,
mean(NPV) as NPV, mean(F1score) as F1score
from &dataset;
quit;

/*computing weighted macro measures*/
data &dataset;
set &dataset;
weight=(tp+fn)/&totalrows;
w_accuracy=accuracy*weight;
w_misclassrate=misclassrate*weight;
w_sensitivity=sensitivity*weight;
w_FNR=FNR*weight;
w_specificity=specificity*weight;
w_FPR=FPR*weight;
w_precision=precision*weight;
w_FPR=FPR*weight;
w_precision=precision*weight;
w_NPV=NPV*weight;
w_F1score=F1score*weight;
run;

proc sql;
select 'weighted macro measures',sum(w_accuracy)
as accuracy, sum(w_misclassrate) as misclassrate,
sum(w_sensitivity) as sensitivity,
sum(w_FNR) as FNR, sum(w_specificity) as specificity,
sum(w_FPR) as FPR, sum(w_precision) as precision,
sum(w_NPV) as NPV, sum(w_F1score) as F1score
from &dataset;
quit;

%mend;

/*COMPUTING PERFORMANCE MEASURES FOR FITTED GINI TREE*/
%perf_measures(ginitree)
```

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
very bad	0	0	143	8	0.94702	0.05298	0	1	1	0	.	0.94702	0

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
bad	14	36	96	5	0.728477	0.271523	0.736842	0.263158	0.727273	0.272727	0.28	0.950495	0.405797

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
okay	9	52	79	11	0.582781	0.417219	0.45	0.55	0.603053	0.396947	0.147541	0.877778	0.222222

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
good	6	8	98	39	0.688742	0.311258	0.133333	0.866667	0.924528	0.075472	0.428571	0.715328	0.20339

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
very good	16	10	82	43	0.649007	0.350993	0.271186	0.728814	0.891304	0.108696	0.615385	0.656	0.376471

The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
micro measures	0.736755	0.263245	0.315217	0.684783	0.8125	0.1875	0.232	0.868476	0.267281

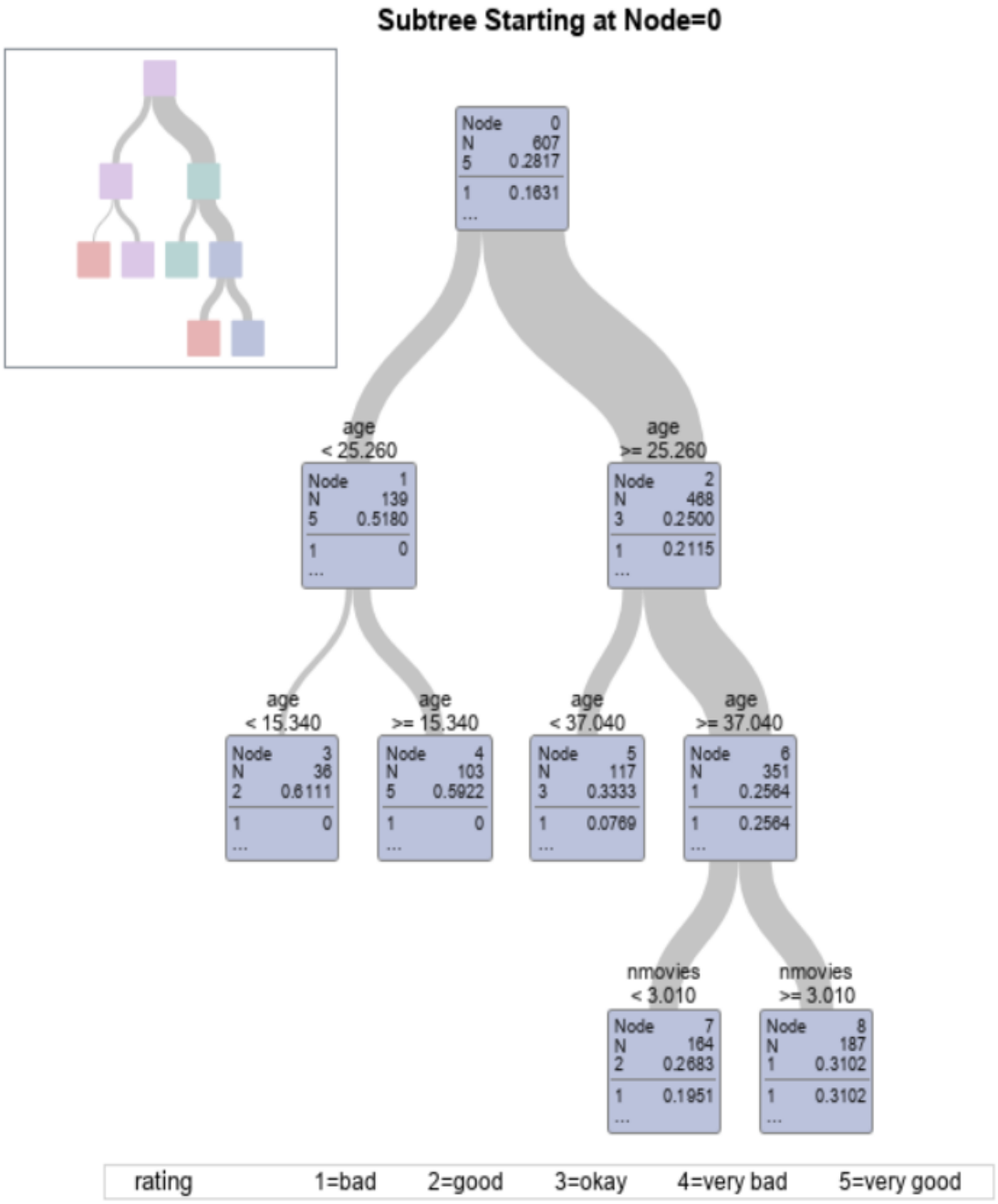
The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
macro measures	0.736755	0.263245	0.330044	0.669956	0.813714	0.186286	0.285371	0.872655	0.207852

The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
weighted macro measures	0.42428	0.184992	0.192053	0.417219	0.499888	0.109384	0.182494	0.499211	0.141107

```
/*ENTROPY SPLITTING AND COST-COMPLEXITY PRUNING*/
proc hpsplit data=movie;
class rating gender member;
   model rating=age gender member nmovies;
grow entropy;
prune costcomplexity;
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;
```




```
/*COMPUTING PERFORMANCE MEASURES FOR FITTED ENTROPY TREE*/
%perf_measures(entropytree)
```

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
very bad	0	0	143	8	0.94702	0.05298	0	1	1	0	.	0.94702	0

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
bad	14	36	96	5	0.728477	0.271523	0.736842	0.263158	0.727273	0.272727	0.28	0.950495	0.405797

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
okay	9	52	79	11	0.582781	0.417219	0.45	0.55	0.603053	0.396947	0.147541	0.877778	0.222222

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
good	6	8	98	39	0.688742	0.311258	0.133333	0.866667	0.924528	0.075472	0.428571	0.715328	0.20339

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
very good	16	10	82	43	0.649007	0.350993	0.271186	0.728814	0.891304	0.108696	0.615385	0.656	0.376471

The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
micro measures	0.736755	0.263245	0.315217	0.684783	0.8125	0.1875	0.232	0.868476	0.267281

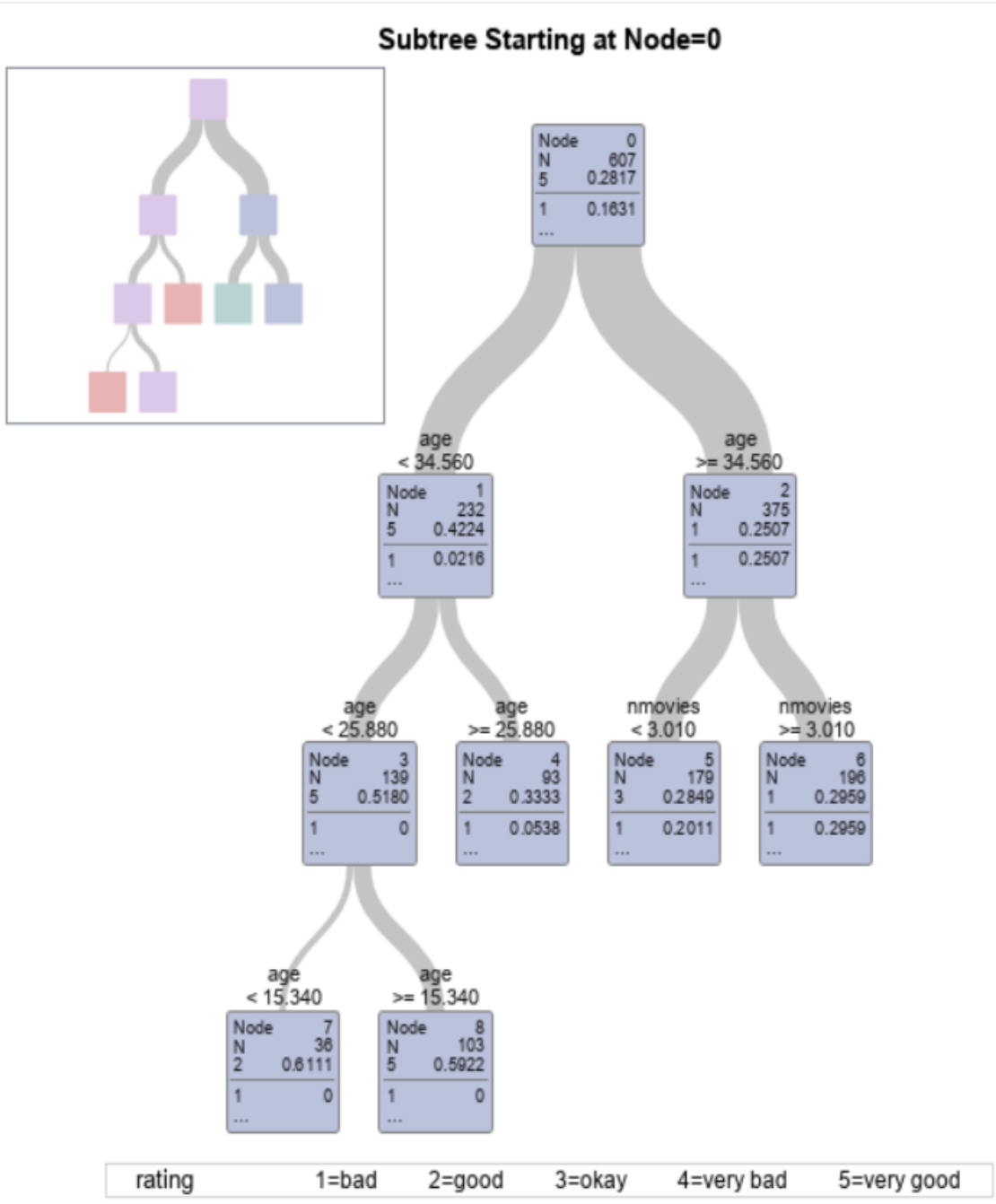
The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
macro measures	0.736755	0.263245	0.330044	0.669956	0.813714	0.186286	0.285371	0.872655	0.207852

The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
weighted macro measures	0.42428	0.184992	0.192053	0.417219	0.499888	0.109384	0.182494	0.499211	0.141107

```
/*CHAID SPLITTING AND COST-COMPLEXITY PRUNING*/
proc hpsplit data=movie;
class rating gender member;
   model rating=age gender member nmovies;
grow CHAID;
prune costcomplexity;
partition rolevar=selected(train="1");
output out=predicted;
ID selected;
run;
```




```
/*COMPUTING PERFORMANCE MEASURES FITTED CHAID TREE*/
%perf_measures(CHAIDtree)
```

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
very bad	0	0	143	8	0.94702	0.05298	0	1	1	0	.	0.94702	0

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
bad	14	39	93	5	0.708609	0.291391	0.736842	0.263158	0.704545	0.295455	0.264151	0.94898	0.388889

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
okay	9	49	82	11	0.602649	0.397351	0.45	0.55	0.625954	0.374046	0.155172	0.88172	0.230769

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
good	6	8	98	39	0.688742	0.311258	0.133333	0.866667	0.924528	0.075472	0.428571	0.715328	0.20339

The SAS System

class	tp	fp	tn	fn	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
very good	16	10	82	43	0.649007	0.350993	0.271186	0.728814	0.891304	0.108696	0.615385	0.656	0.376471

The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
micro measures	0.736755	0.263245	0.315217	0.684783	0.8125	0.1875	0.232	0.868476	0.267281

The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
macro measures	0.736755	0.263245	0.330044	0.669956	0.813757	0.186243	0.282632	0.873262	0.205762

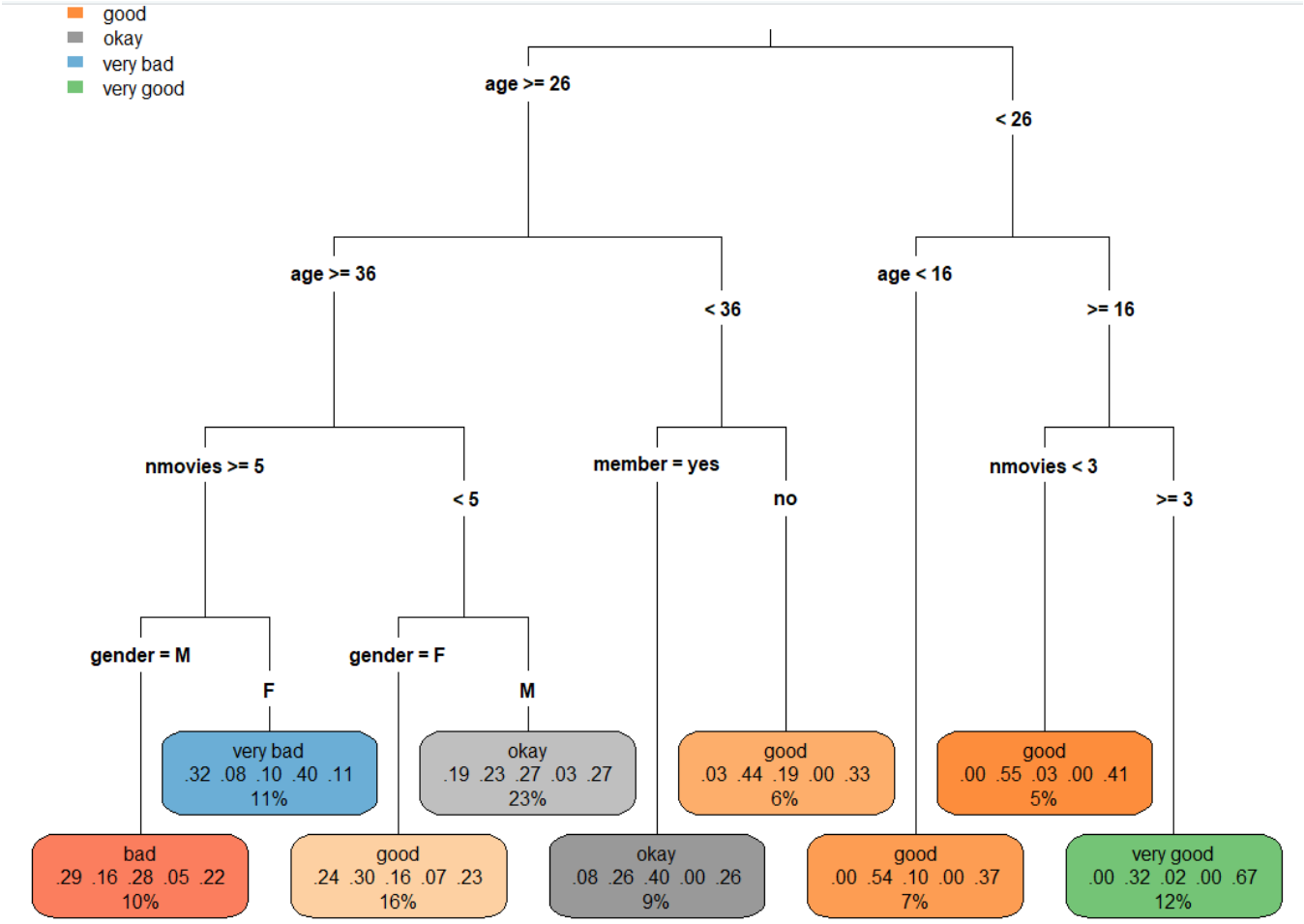
The SAS System

	accuracy	misclassrate	sensitivity	FNR	specificity	FPR	precision	NPV	F1score
weighted macro measures	0.424411	0.18486	0.192053	0.417219	0.500061	0.10921	0.18151	0.499543	0.140111

In R:

```
movie.data<- read.csv(file="./movie_data.csv", header=TRUE, sep=",")

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
set.seed(566222)
sample <- sample(c(TRUE, FALSE), nrow(movie.data),replace=TRUE, prob=c(0.8,0.2))
train<- movie.data[sample,]
test<- movie.data[!sample,]

#FITTING PRUNED MULTINOMIAL CLASSIFICATION TREE WITH GINI SPLITTING
library(rpart)
tree.gini<- rpart(rating ~ age + gender + member + nmovies, data=train, method="class",
```

```
parms=list(split="Gini"), maxdepth=4)

#PLOTTING FITTED TREE
library(rpart.plot)
rpart.plot(tree.gini, type=3)
```




```
#COMPUTING PREDICTED VALUES FOR TESTING DATA
pred.values<- predict(tree.gini, test)

#DETERMINING PREDICTED CLASSES
test<- cbind(test, pred.values)
test$maxprob<- pmax(test$'very bad',test$'bad',test$'okay', test$'good',test$'very good')

test$predclass<- ifelse(test$maxprob==test$'very bad', 'very bad', ifelse(test$maxprob==test$'bad','bad',
ifelse(test$maxprob==test$'okay','okay', ifelse(test$maxprob==test$'good','good','very good'))))
```

```
#DEFINING FUNCTION FOR COMPUTING PERFORMANCE MEASURES
perf.measures<- function() {

#COMPUTING PERFORMANCE MEASURES FOR INDIVIDUAL CLASSES
tp<- c()
fp<- c()
tn<- c()
fn<- c()
accuracy<- c()
misclassrate<- c()
sensitivity<- c()
FNR<- c()
specificity<- c()
FPR<- c()
precision<- c()
NPV<- c()
F1score<- c()

class.metrics<- function(class) {

    tp.class<- ifelse(test$predclass==class & test$liked==class,1,0)
    fp.class<- ifelse(test$predclass==class & test$liked!=class,1,0)
    tn.class<- ifelse(test$predclass!=class & test$liked!=class,1,0)
    fn.class<- ifelse(test$predclass!=class & test$liked==class,1,0)

    message('CLASS MEASURES:')
    message('class:', class)
    print(paste('tp:', tp[class]«- sum(tp.class)))
    #«- is global assignment, works outside the function
    print(paste('fp:', fp[class]«- sum(fp.class)))
    print(paste('tn:', tn[class]«- sum(tn.class)))
    print(paste('fn:', fn[class]«- sum(fn.class)))
    total«- nrow(test)

    print(paste('accuracy:', accuracy[class]«- (tp[class]+tn[class])/total))
    print(paste('misclassrate:', misclassrate[class]«- (fp[class]+fn[class])/total))
    print(paste('sensitivity:', sensitivity[class]«- tp[class]/(tp[class]+fn[class])))
    print(paste('FNR:', FNR[class]«- fn[class]/(tp[class]+fn[class])))
    print(paste('specificity:', specificity[class]«- tn[class]/(fp[class]+tn[class])))
    print(paste('FPR:', FPR[class]«- fp[class]/(fp[class]+tn[class])))
    print(paste('precision:', precision[class]«- tp[class]/(tp[class]+fp[class])))
```

```
    print(paste('NPV:', NPV[class]<- tn[class]/(fn[class]+tn[class])))
    print(paste('F1score:', F1score[class]<- 2*tp[class]/(2*tp[class]+fn[class]+fp[class])))
}

class.metrics(class='very bad')
class.metrics(class='bad')
class.metrics(class='okay')
class.metrics(class='good')
class.metrics(class='very good')

#COMPUTING MICRO MEASURES
tp.sum<- sum(tp)
fp.sum<- sum(fp)
tn.sum<- sum(tn)
fn.sum<- sum(fn)

message('MICRO MEASURES:')
print(paste('accuracy:', accuracy.micro<- (tp.sum+tn.sum)/(tp.sum+fp.sum+tn.sum+fn.sum)))
print(paste('misclassrate:', misclassrate.micro<- (fp.sum+fn.sum)/(tp.sum+fp.sum+tn.sum+fn.sum)))
print(paste('sensitivity:', sensitivity.micro<- tp.sum/(tp.sum+fn.sum)))
print(paste('FNR:', FNR.micro<- fn.sum/(tp.sum+fn.sum)))
print(paste('specificity:', specificity.micro<- tn.sum/(fp.sum+tn.sum)))
print(paste('FPR:', FPR.micro<- fp.sum/(fp.sum+tn.sum)))
print(paste('precision:', precision.micro<- tp.sum/(tp.sum+fp.sum)))
print(paste('NPV:', NPV.micro<- tn.sum/(fn.sum+tn.sum)))
print(paste('F1-score:', F1score.micro<- 2*tp.sum/(2*tp.sum+fn.sum+fp.sum)))

#COMPUTING MACRO MEASURES
message('MACRO MEASURES:')
print(paste('accuracy:', accuracy.macro<- mean(accuracy)))
print(paste('misclassrate:', misclassrate.macro<- mean(misclassrate)))
print(paste('sensitivity:', sensitivity.macro<- mean(sensitivity)))
print(paste('FNR:', FNR.macro<- mean(FNR)))
print(paste('specificity:', specificity.macro<- mean(specificity)))
print(paste('FPR:', FPR.macro<- mean(FPR)))
print(paste('precision:', precision.macro<- mean(precision, na.rm=TRUE)))
print(paste('NPV:', NPV.macro<- mean(NPV)))
print(paste('F1-score:', F1score.macro<- mean(F1score)))

#COMPUTING WEIGHTED MACRO MEASURES
weight<- c()
```

```
for (class in 1:5)
weight[class]<- (tp[class]+fn[class])/total

message('WEIGHTED MACRO MEASURES:')
print(paste('accuracy:', accuracy.wmacro<- weight%*%accuracy))
print(paste('misclassrate:', misclassrate.wmacro<- weight%*%misclassrate))
print(paste('sensitivity:', sensitivity.wmacro<- weight%*%sensitivity))
print(paste('FNR:', FNR.wmacro<- weight%*%FNR))
print(paste('specificity:', specificity.wmacro<- weight%*%specificity))
print(paste('FPR:', FPR.wmacro<- weight%*%FPR))
precision[is.na(precision)]<- 0
print(paste('precision:', precision.wmacro<- weight%*%precision))
print(paste('NPV:', NPV.wmacro<- weight%*%NPV))
print(paste('F1-score:', F1score.wmacro<- weight%*%F1score))

}

#COMPUTING PERFORMANCE MEASURES FOR FITTED GINI TREE
perf.measures()


CLASS MEASURES:
class:very bad
[1] "tp: 3"
[1] "fp: 10"
[1] "tn: 167"
[1] "fn: 13"
[1] "accuracy: 0.880829015544041"
[1] "misclassrate: 0.119170984455959"
[1] "sensitivity: 0.1875"
[1] "FNR: 0.8125"
[1] "specificity: 0.943502824858757"
[1] "FPR: 0.0564971751412429"
[1] "precision: 0.230769230769231"
[1] "NPV: 0.927777777777778"
[1] "F1score: 0.206896551724138"
CLASS MEASURES:
class:bad
[1] "tp: 10"
[1] "fp: 24"
[1] "tn: 139"
[1] "fn: 20"
[1] "accuracy: 0.772020725388601"
```

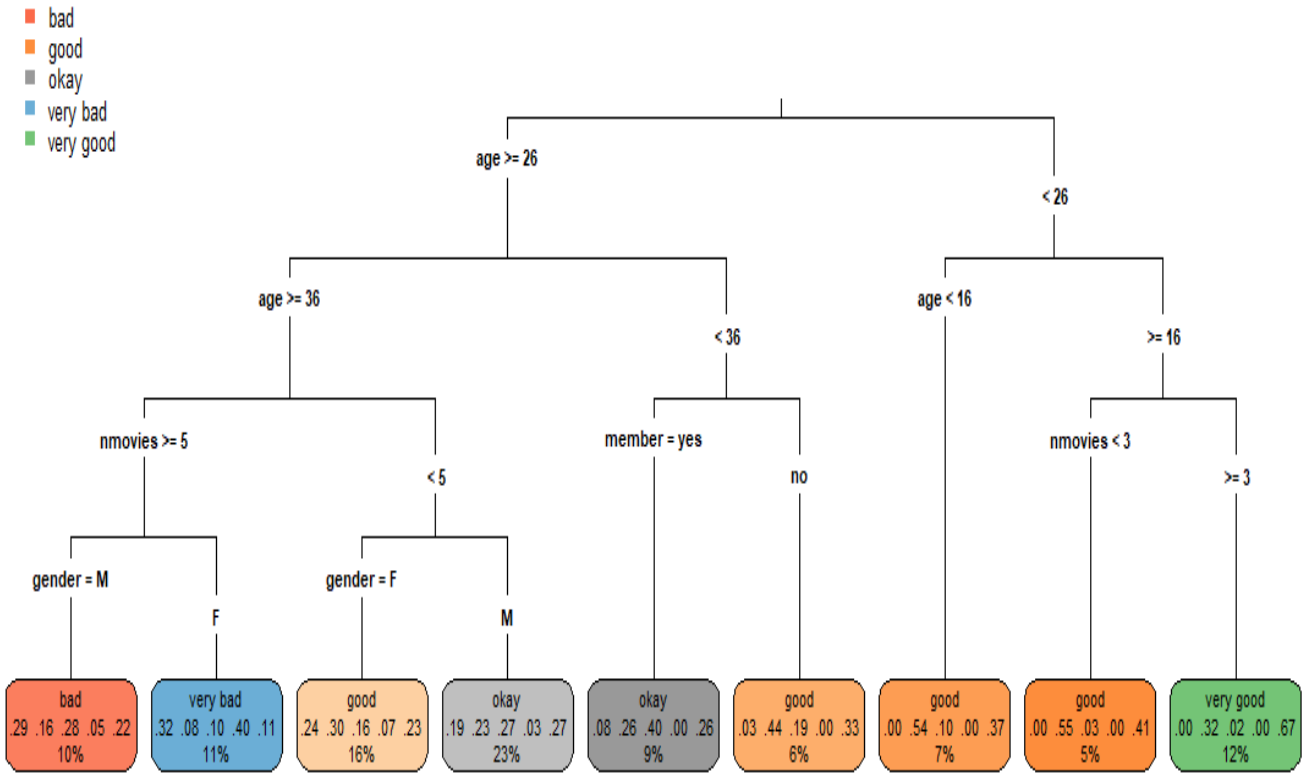
```
[1] "misclassrate: 0.227979274611399"
[1] "sensitivity: 0.333333333333333"
[1] "FNR: 0.666666666666667"
[1] "specificity: 0.852760736196319"
[1] "FPR: 0.147239263803681"
[1] "precision: 0.294117647058824"
[1] "NPV: 0.874213836477987"
[1] "F1score: 0.3125"
CLASS MEASURES:
class:okay
[1] "tp: 12"
[1] "fp: 49"
[1] "tn: 108"
[1] "fn: 24"
[1] "accuracy: 0.621761658031088"
[1] "misclassrate: 0.378238341968912"
[1] "sensitivity: 0.333333333333333"
[1] "FNR: 0.666666666666667"
[1] "specificity: 0.687898089171974"
[1] "FPR: 0.312101910828025"
[1] "precision: 0.19672131147541"
[1] "NPV: 0.818181818181818"
[1] "F1score: 0.247422680412371"
CLASS MEASURES:
class:good
[1] "tp: 20"
[1] "fp: 44"
[1] "tn: 95"
[1] "fn: 34"
[1] "accuracy: 0.595854922279793"
[1] "misclassrate: 0.404145077720207"
[1] "sensitivity: 0.37037037037037"
[1] "FNR: 0.62962962962963"
[1] "specificity: 0.683453237410072"
[1] "FPR: 0.316546762589928"
[1] "precision: 0.3125"
[1] "NPV: 0.736434108527132"
[1] "F1score: 0.338983050847458"
CLASS MEASURES:
class:very good
[1] "tp: 14"
[1] "fp: 7"
```

```
[1] "tn: 129"
[1] "fn: 43"
[1] "accuracy: 0.740932642487047"
[1] "misclassrate: 0.259067357512953"
[1] "sensitivity: 0.245614035087719"
[1] "FNR: 0.754385964912281"
[1] "specificity: 0.948529411764706"
[1] "FPR: 0.0514705882352941"
[1] "precision: 0.666666666666667"
[1] "NPV: 0.75"
[1] "F1score: 0.358974358974359"
MICRO MEASURES:
[1] "accuracy: 0.722279792746114"
[1] "misclassrate: 0.277720207253886"
[1] "sensitivity: 0.305699481865285"
[1] "FNR: 0.694300518134715"
[1] "specificity: 0.826424870466321"
[1] "FPR: 0.173575129533679"
[1] "precision: 0.305699481865285"
[1] "NPV: 0.826424870466321"
[1] "F1-score: 0.305699481865285"
MACRO MEASURES:
[1] "accuracy: 0.722279792746114"
[1] "misclassrate: 0.277720207253886"
[1] "sensitivity: 0.294030214424951"
[1] "FNR: 0.705969785575049"
[1] "specificity: 0.823228859880366"
[1] "FPR: 0.176771140119634"
[1] "precision: 0.340154971194026"
[1] "NPV: 0.821321508192943"
[1] "F1-score: 0.292955328391665"
WEIGHTED MACRO MEASURES:
[1] "accuracy: 0.694542135359339"
[1] "misclassrate: 0.305457864640661"
[1] "sensitivity: 0.305699481865285"
[1] "FNR: 0.694300518134715"
[1] "specificity: 0.810444817536544"
[1] "FPR: 0.189555182463457"
[1] "precision: 0.385869452420659"
[1] "NPV: 0.792968118413444"
[1] "F1-score: 0.312741888755092"
```

#FITTING PRUNED MULTINOMIAL CLASSIFICATION TREE WITH ENTROPY SPLIT-

TING
library(rpart)
tree.entropy<- rpart(rating ~ age + gender + member + nmovies, data=train, method="class", parms=list(split="entropy"))

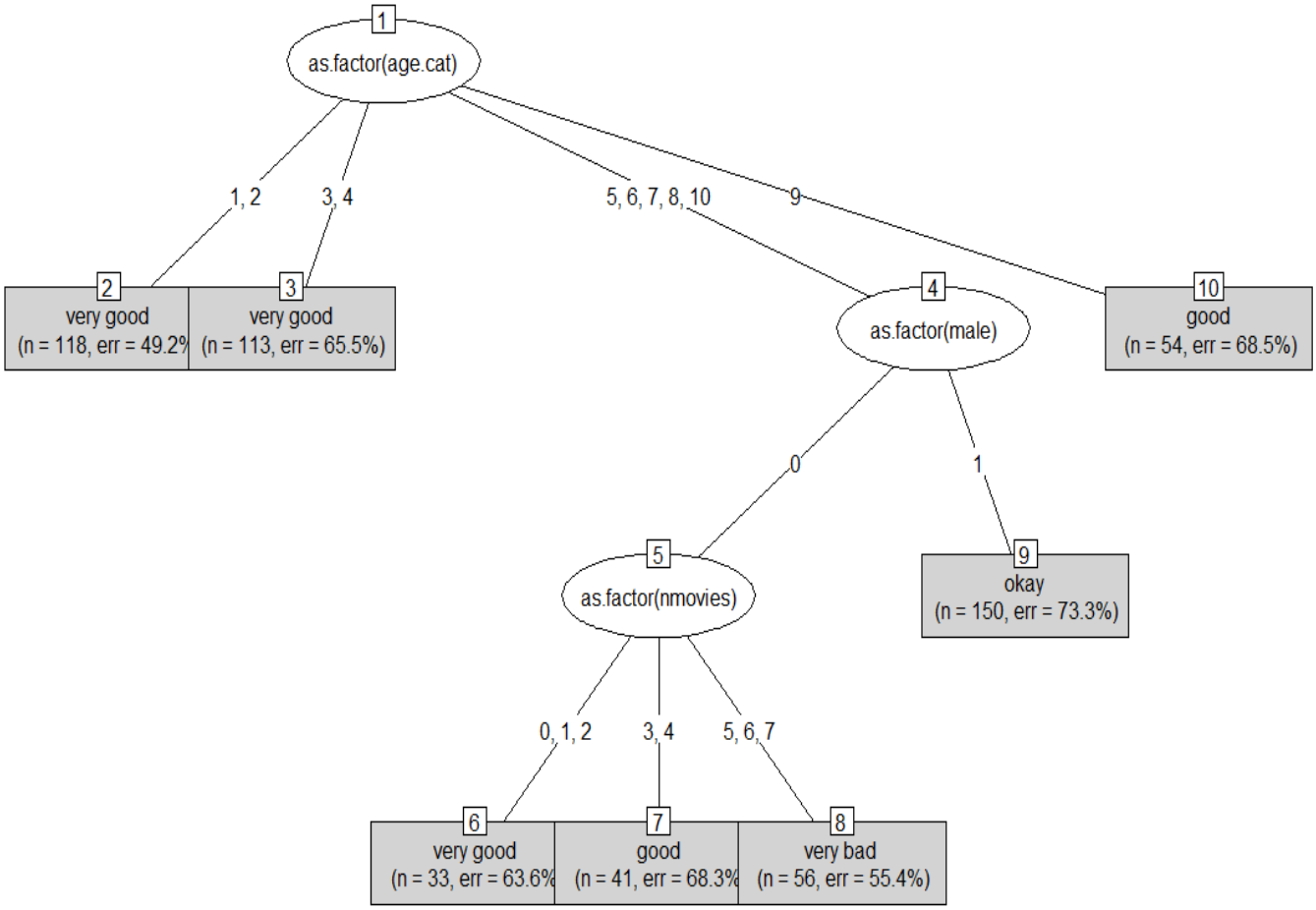
#PLOTTING FITTED TREE
rpart.plot(tree.entropy, type=3)
#Note: same as tree.gini



#FITTING PRUNED MULTINOMIAL CLASSIFICATION TREE WITH CHAID SPLITTING
#BINNING CONTINUOUS PREDICTOR VARIABLES
library(dplyr)
movie.data<- mutate(movie.data, age.cat=ntile(age,10))

#CREATING INDICATORS FOR CATEGORICAL VARIABLES
movie.data$male<- ifelse(movie.data$gender=="M",1,0)
movie.data$member.yes<- ifelse(movie.data$member=="yes",1,0)

```
#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
set.seed(566222)
sample <- sample(c(TRUE, FALSE), nrow(movie.data), replace=TRUE, prob=c(0.8,0.2))
train<- movie.data[sample,]
test<- movie.data[!sample,]

#FITTING BINARY CLASSIFICATION TREE
library(CHAID)
tree.CHAID<- chaid(as.factor(rating) ~ as.factor(age.cat) + as.factor(male) + as.factor(member.yes)
+ as.factor(nmovies), data=train)

#PLOTTING FITTED TREE
plot(tree.CHAID, type="simple")
```



```
#COMPUTING PREDICTED VALUES FOR TESTING DATA
pred.pneumonia<- predict(tree.CHAID, newdata=test)

#COMPUTING PERFORMANCE MEASURES FOR FITTED CHAID TREE
test$predclass<- pred.values
perf.measures()


CLASS MEASURES:
class:very bad
[1] "tp: 3"
[1] "fp: 8"
[1] "tn: 169"
[1] "fn: 13"
[1] "accuracy: 0.89119170984456"
[1] "misclassrate: 0.10880829015544"
[1] "sensitivity: 0.1875"
[1] "FNR: 0.8125"
[1] "specificity: 0.954802259887006"
[1] "FPR: 0.0451977401129944"
[1] "precision: 0.272727272727273"
[1] "NPV: 0.928571428571429"
[1] "F1score: 0.222222222222222"
CLASS MEASURES:
class:bad
[1] "tp: 0"
[1] "fp: 0"
[1] "tn: 163"
[1] "fn: 30"
[1] "accuracy: 0.844559585492228"
[1] "misclassrate: 0.155440414507772"
[1] "sensitivity: 0"
[1] "FNR: 1"
[1] "specificity: 1"
[1] "FPR: 0"
[1] "precision: NaN"
[1] "NPV: 0.844559585492228"
[1] "F1score: 0"
CLASS MEASURES:
class:okay
[1] "tp: 12"
[1] "fp: 47"
[1] "tn: 110"
```

```
[1] "fn: 24"
[1] "accuracy: 0.632124352331606"
[1] "misclassrate: 0.367875647668394"
[1] "sensitivity: 0.333333333333333"
[1] "FNR: 0.666666666666667"
[1] "specificity: 0.700636942675159"
[1] "FPR: 0.299363057324841"
[1] "precision: 0.203389830508475"
[1] "NPV: 0.82089552238806"
[1] "F1score: 0.252631578947368"
CLASS MEASURES:
class:good
[1] "tp: 9"
[1] "fp: 31"
[1] "tn: 108"
[1] "fn: 45"
[1] "accuracy: 0.606217616580311"
[1] "misclassrate: 0.393782383419689"
[1] "sensitivity: 0.166666666666667"
[1] "FNR: 0.833333333333333"
[1] "specificity: 0.776978417266187"
[1] "FPR: 0.223021582733813"
[1] "precision: 0.225"
[1] "NPV: 0.705882352941177"
[1] "F1score: 0.191489361702128"
CLASS MEASURES:
class:very good
[1] "tp: 36"
[1] "fp: 47"
[1] "tn: 89"
[1] "fn: 21"
[1] "accuracy: 0.647668393782383"
[1] "misclassrate: 0.352331606217617"
[1] "sensitivity: 0.631578947368421"
[1] "FNR: 0.368421052631579"
[1] "specificity: 0.654411764705882"
[1] "FPR: 0.345588235294118"
[1] "precision: 0.433734939759036"
[1] "NPV: 0.809090909090909"
[1] "F1score: 0.514285714285714"
MICRO MEASURES:
[1] "accuracy: 0.724352331606218"
```

```
[1] "misclassrate: 0.275647668393782"
[1] "sensitivity: 0.310880829015544"
[1] "FNR: 0.689119170984456"
[1] "specificity: 0.827720207253886"
[1] "FPR: 0.172279792746114"
[1] "precision: 0.310880829015544"
[1] "NPV: 0.827720207253886"
[1] "F1-score: 0.310880829015544"
MACRO MEASURES:
[1] "accuracy: 0.724352331606218"
[1] "misclassrate: 0.275647668393782"
[1] "sensitivity: 0.263815789473684"
[1] "FNR: 0.736184210526316"
[1] "specificity: 0.817365876906847"
[1] "FPR: 0.182634123093153"
[1] "precision: 0.283713010748696"
[1] "NPV: 0.82179995969676"
[1] "F1-score: 0.236125775431486"
WEIGHTED MACRO MEASURES:
[1] "accuracy: 0.683964670192488"
[1] "misclassrate: 0.316035329807512"
[1] "sensitivity: 0.310880829015544"
[1] "FNR: 0.689119170984456"
[1] "specificity: 0.77594855551869"
[1] "FPR: 0.22405144448131"
[1] "precision: 0.251598765949257"
[1] "NPV: 0.797834187071944"
[1] "F1-score: 0.271010381574411"
```

In Python:

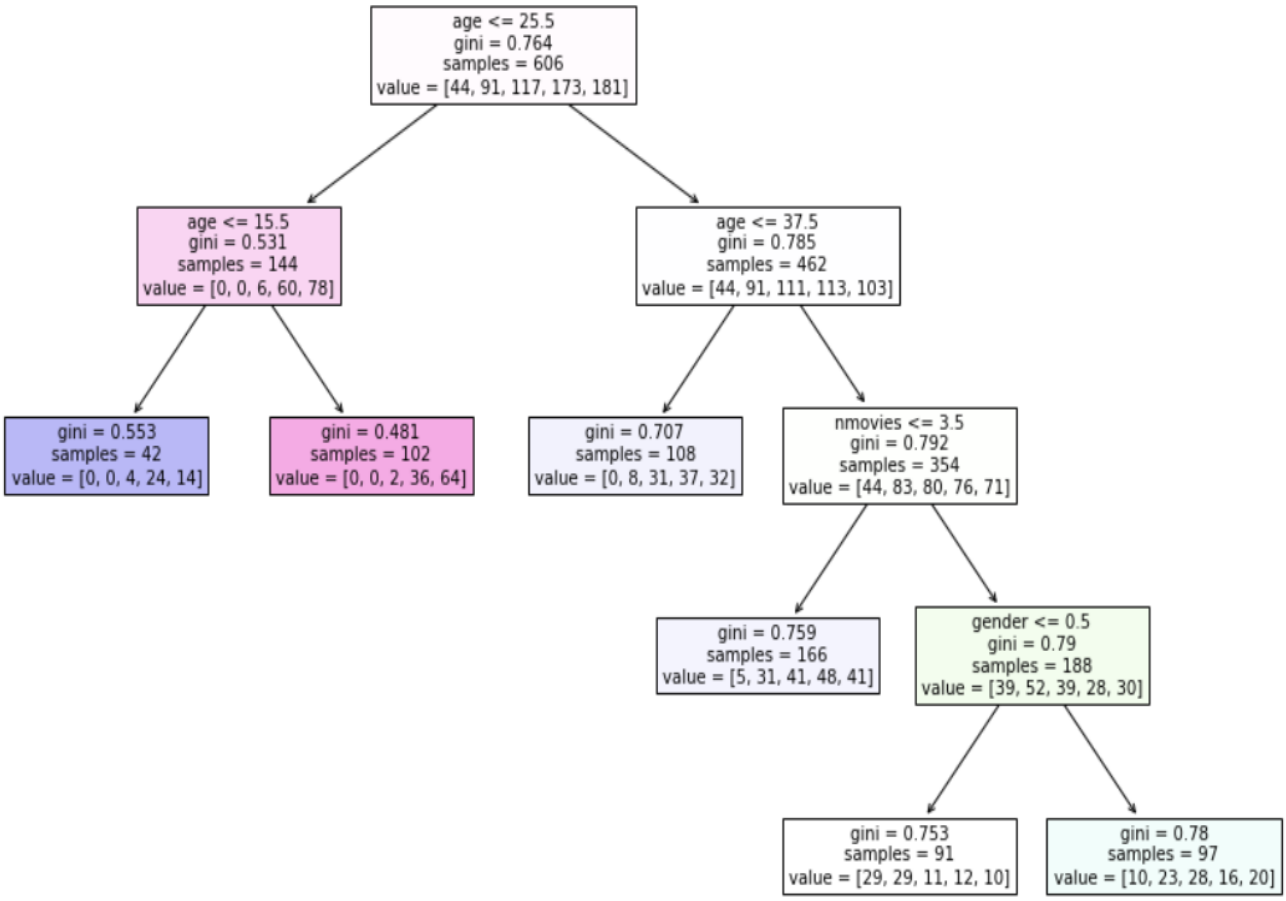
```
import numpy
import pandas
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.model_selection import train_test_split

movie_data=pandas.read_csv('C:/Users/000110888/Desktop/movie_data.csv')
code_gender={'M':1,'F':0}
code_member={'yes':1,'no':0}
code_rating={'very bad':1,'bad':2,'okay':3,'good':4,'very good':5}

movie_data['gender']=movie_data['gender'].map(code_gender)
movie_data['member']=movie_data['member'].map(code_member)
movie_data['rating']=movie_data['rating'].map(code_rating)

X=movie_data.iloc[:,0:4].values
y=movie_data.iloc[:,4].values

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.20,
random_state=687088)

#########################################################################

#DEFINING FUNCTION FOR COMPUTING PERFORMANCE MEASURES
def perf_measures():

    #COMPUTING PERFORMANCE MEASURES FOR INDIVIDUAL CLASSES
    tp=[]
    fp=[]
    tn=[]
    fn=[]

    for cls in range(5):
        tp_sum=0
        fp_sum=0
        tn_sum=0
        fn_sum=0
        for sub1, sub2 in zip(predclass, y_test):
```

```
        if sub1==cls+1 and sub2==cls+1:
            tp_sum+=1

        if sub1==cls+1 and sub2!=cls+1:
            fp_sum+=1

        if sub1!=cls+1 and sub2!=cls+1:
            tn_sum+=1

        if sub1!=cls+1 and sub2==cls+1:
            fn_sum+=1

    tp.append(tp_sum)
    fp.append(fp_sum)
    tn.append(tn_sum)
    fn.append(fn_sum)


accuracy=[]
misclassrate=[]
sensitivity=[]
FNR=[]
specificity=[]
FPR=[]
precision=[]
NPV=[]
F1score=[]

print('CLASS MEASURES:')
for cls in range(5):
    accuracy_cls=(tp[cls]+tn[cls])/(tp[cls]+fp[cls]+tn[cls]+fn[cls])
    misclassrate_cls=(fp[cls]+fn[cls])/(tp[cls]+fp[cls]+tn[cls]+fn[cls])
    sensitivity_cls=tp[cls]/(tp[cls]+fn[cls])
```

```
        FNR.append(FNR_cls)
        specificity.append(specificity_cls)
        FPR.append(FPR_cls)
        precision.append(precision_cls)
        NPV.append(NPV_cls)
        F1score.append(F1score_cls)

        print()
        print('CLASS:', cls+1)
        print('tp:', tp[cls])
        print('fp:', fp[cls])
        print('tn:', tn[cls])
        print('fn:', fn[cls])
        print('accuracy:', accuracy[cls])
        print('misclassrate:', misclassrate[cls])
        print('sensitivity:', sensitivity[cls])
        print('FNR:', FNR[cls])
        print('specificity:', specificity[cls])
        print('FPR:', FPR[cls])
        print('precision:', precision[cls])
        print('NPV:', NPV[cls])
        print('F1score:', F1score[cls])

    #COMPUTING MICRO MEASURES
    tp_sum=numpy.sum(tp)
    fp_sum=numpy.sum(fp)
    tn_sum=numpy.sum(tn)
    fn_sum=numpy.sum(fn)

    print()
    print('MICRO MEASURES:')
    accuracy_micro=(tp_sum+tn_sum)/(tp_sum+fp_sum+tn_sum+fn_sum)
    misclassrate_micro=(fp_sum+fn_sum)/(tp_sum+fp_sum+tn_sum+fn_sum)
    sensitivity_micro=tp_sum/(tp_sum+fn_sum)
    FNR_micro=fn_sum/(tp_sum+fn_sum)
    specificity_micro=tn_sum/(fp_sum+tn_sum)
    FPR_micro=fp_sum/(fp_sum+tn_sum)
    precision_micro=tp_sum/(tp_sum+fp_sum)
    NPV_micro=tn_sum/(fn_sum+tn_sum)
    F1score_micro=2*tp_sum/(2*tp_sum+fn_sum+fp_sum)

    print('accuracy:', accuracy_micro)
    print('misclassrate:', misclassrate_micro)
    print('sensitivity:', sensitivity_micro)
    print('FNR:', FNR_micro)
    print('specificity:', specificity_micro)
    print('FPR:', FPR_micro)
    print('precision:', precision_micro)
    print('NPV:', NPV_micro)
    print('F1-score:', F1score_micro)
```

```
#COMPUTING MACRO MEASURES
accuracy_macro=numpy.mean(accuracy)
misclassrate_macro=numpy.mean(misclassrate)
sensitivity_macro=numpy.mean(sensitivity)
FNR_macro=numpy.mean(FNR)
specificity_macro=numpy.mean(specificity)
FPR_macro=numpy.mean(FPR)
precision_macro=numpy.mean(precision)
NPV_macro=numpy.mean(NPV)
F1score_macro=numpy.mean(F1score)

print()
print('MACRO MEASURES:')
print('accuracy:', accuracy_macro)
print('misclassrate:', misclassrate_macro)
print('sensitivity:', sensitivity_macro)
print('FNR:', FNR_macro)
print('specificity:', specificity_macro)
print('FPR:', FPR_macro)
print('precision:', precision_macro)
print('NPV:', NPV_macro)
print('F1-score:', F1score_macro)

#COMPUTING WEIGHTED MACRO MEASURES
weight=[]

for cls in range(5):
    weight_cls=(tp[cls]+fn[cls])/(tp[cls]+fp[cls]+tn[cls]+fn[cls])
    weight.append(weight_cls)

accuracy_wmacro=numpy.dot(weight,accuracy)
misclassrate_wmacro=numpy.dot(weight,misclassrate)
sensitivity_wmacro=numpy.dot(weight,sensitivity)
FNR_wmacro=numpy.dot(weight,FNR)
specificity_wmacro=numpy.dot(weight,specificity)
FPR_wmacro=numpy.dot(weight,FPR)
precision_wmacro=numpy.dot(weight,precision)
NPV_wmacro=numpy.dot(weight,NPV)
F1score_wmacro=numpy.dot(weight,F1score)

print()
print('WEIGHTED MACRO MEASURES:')
print('accuracy:', accuracy_wmacro)
print('misclassrate:', misclassrate_wmacro)
print('sensitivity:', sensitivity_wmacro)
print('FNR:', FNR_wmacro)
print('specificity:', specificity_wmacro)
print('FPR:', FPR_wmacro)
print('precision:', precision_wmacro)
print('NPV:', NPV_wmacro)
print('F1-score:', F1score_wmacro)
```

```
#FITTING MULTINOMIAL CLASSIFICATION TREE WITH GINI SPLITTING CRITERION
gini_tree=DecisionTreeClassifier(max_leaf_nodes=6, criterion='gini', random_state=707720)
gini_tree.fit=gini_tree.fit(X_train,y_train)

#PLOTTING FITTED TREE
fig=plt.figure(figsize=(15,10))
tree.plot_tree(gini_tree.fit, feature_names=['age','gender','member','nmovies'], filled=True)
```



```
#COMPUTING PREDICTED VALUES FOR TESTING DATA
y_pred=gini_tree.predict_proba(X_test)

#DETERMINING PREDICTED CLASSES
predclass=[]
for i  in range(0, len(y_pred)):
    list=[y_pred[i,0],y_pred[i,1], y_pred[i,2], y_pred[i,3], y_pred[i,4]]
    predclass.append(list.index(max(list))+1)
predclass=numpy.asarray(predclass)

#COMPUTING PERFORMANCE MEASURES FOR FITTED GINI TREE
print()
print('PERFORMANCE MEASURES FOR FITTED GINI TREE')
perf_measures()
```

```
PERFORMANCE MEASURES FOR FITTED GINI TREE
CLASS MEASURES:

CLASS: 1
tp: 8
fp: 15
tn: 127
fn: 2
accuracy: 0.8881578947368421
misclassrate: 0.1118421052631579
sensitivity: 0.8
FNR: 0.2
specificity: 0.8943661971830986
FPR: 0.1056338028169014
precision: 0.34782608695652173
NPV: 0.9844961240310077
F1score: 0.48484848484848486

CLASS: 2
tp: 0
fp: 0
tn: 125
fn: 27
accuracy: 0.8223684210526315
misclassrate: 0.17763157894736842
```

sensitivity: 0.0
FNR: 1.0
specificity: 1.0
FPR: 0.0
precision: 0
NPV: 0.8223684210526315
F1score: 0.0

CLASS: 3
tp: 5
fp: 21
tn: 106
fn: 20
accuracy: 0.7302631578947368
misclassrate: 0.26973684210526316
sensitivity: 0.2
FNR: 0.8
specificity: 0.8346456692913385
FPR: 0.16535433070866143
precision: 0.19230769230769232
NPV: 0.8412698412698413
F1score: 0.19607843137254902

CLASS: 4
tp: 23
fp: 53
tn: 58
fn: 18
accuracy: 0.5328947368421053
misclassrate: 0.46710526315789475
sensitivity: 0.5609756097560976
FNR: 0.43902439024390244
specificity: 0.5225225225225225
FPR: 0.4774774774774775
precision: 0.3026315789473684
NPV: 0.7631578947368421
F1score: 0.39316239316239315

CLASS: 5
tp: 13
fp: 14
tn: 89

fn: 36
accuracy: 0.6710526315789473
misclassrate: 0.32894736842105265
sensitivity: 0.2653061224489796
FNR: 0.7346938775510204
specificity: 0.8640776699029126
FPR: 0.13592233009708737
precision: 0.48148148148148145
NPV: 0.712
F1score: 0.34210526315789475

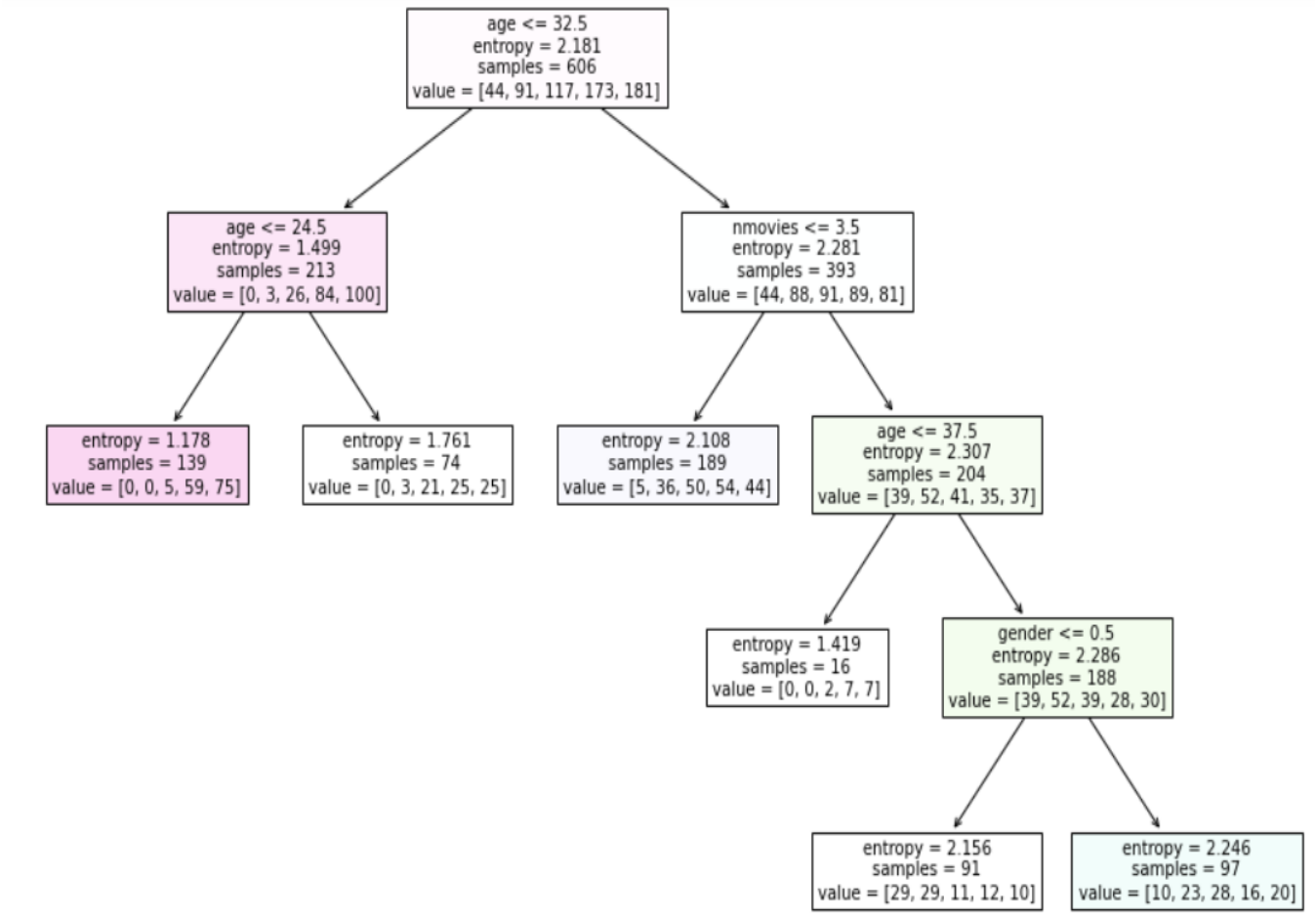
MICRO MEASURES:
accuracy: 0.7289473684210527
misclassrate: 0.2710526315789474
sensitivity: 0.3223684210526316
FNR: 0.6776315789473685
specificity: 0.8305921052631579
FPR: 0.16940789473684212
precision: 0.3223684210526316
NPV: 0.8305921052631579
F1-score: 0.3223684210526316

MACRO MEASURES:
accuracy: 0.7289473684210527
misclassrate: 0.2710526315789474
sensitivity: 0.36525634644101546
FNR: 0.6347436535589845
specificity: 0.8231224117799745
FPR: 0.17687758822002553
precision: 0.26484936793861275
NPV: 0.8246584562180646
F1-score: 0.28323891450826433

WEIGHTED MACRO MEASURES:
accuracy: 0.6846866343490305
misclassrate: 0.31531336565096957
sensitivity: 0.3223684210526316
FNR: 0.6776315789473684
specificity: 0.7932436378472407
FPR: 0.20675636215275928
precision: 0.2913581612282383
NPV: 0.7845929495045243

```
F1-score: 0.2804819845210101
```

```
#FITTING MULTINOMIAL CLASSIFICATION TREE WITH ENTROPY SPLITTING CRITERION
entropy_tree=DecisionTreeClassifier(max_leaf_nodes=6, criterion='entropy', random_state=707720)
entropy_tree.fit=entropy_tree.fit(X_train,y_train)

#PLOTTING FITTED TREE
fig=plt.figure(figsize=(15,10))
tree.plot_tree(entropy_tree.fit, feature_names=['age','gender','member','nmovies'], filled=True)
```



```
#COMPUTING PREDICTED VALUES FOR TESTING DATA
y_pred=entropy_tree.predict_proba(X_test)

#DETERMINING PREDICTED CLASSES
predclass=[]
for i  in range(0, len(y_pred)):
    list=[y_pred[i,0],y_pred[i,1], y_pred[i,2], y_pred[i,3], y_pred[i,4]]
    predclass.append(list.index(max(list))+1)
predclass=numpy.asarray(predclass)

#COMPUTING PERFORMANCE MEASURES FOR FITTED ENTROPY TREE
print()
print('PERFORMANCE MEASURES FOR FITTED ENTROPY TREE')
perf_measures()
```

```
PERFORMANCE MEASURES FOR FITTED ENTROPY TREE
CLASS MEASURES:

CLASS: 1
tp: 8
fp: 15
tn: 127
fn: 2
accuracy: 0.8881578947368421
misclassrate: 0.1118421052631579
sensitivity: 0.8
FNR: 0.2
specificity: 0.8943661971830986
FPR: 0.1056338028169014
precision: 0.34782608695652173
NPV: 0.9844961240310077
F1score: 0.48484848484848486

CLASS: 2
tp: 0
fp: 0
tn: 125
fn: 27
accuracy: 0.8223684210526315
misclassrate: 0.17763157894736842
```

```
sensitivity: 0.0
FNR: 1.0
specificity: 1.0
FPR: 0.0
precision: 0
NPV: 0.8223684210526315
F1score: 0.0

CLASS: 3
tp: 5
fp: 21
tn: 106
fn: 20
accuracy: 0.7302631578947368
misclassrate: 0.26973684210526316
sensitivity: 0.2
FNR: 0.8
specificity: 0.8346456692913385
FPR: 0.16535433070866143
precision: 0.19230769230769232
NPV: 0.8412698412698413
F1score: 0.19607843137254902

CLASS: 4
tp: 21
fp: 51
tn: 60
fn: 20
accuracy: 0.5328947368421053
misclassrate: 0.46710526315789475
sensitivity: 0.5121951219512195
FNR: 0.4878048780487805
specificity: 0.5405405405405406
FPR: 0.4594594594594595
precision: 0.2916666666666667
NPV: 0.75
F1score: 0.37168141592920356

CLASS: 5
tp: 15
fp: 16
tn: 87
```

fn: 34
accuracy: 0.6710526315789473
misclassrate: 0.32894736842105265
sensitivity: 0.30612244897959184
FNR: 0.6938775510204082
specificity: 0.8446601941747572
FPR: 0.1553398058252427
precision: 0.4838709677419355
NPV: 0.71900826446281
F1score: 0.375

MICRO MEASURES:
accuracy: 0.7289473684210527
misclassrate: 0.2710526315789474
sensitivity: 0.3223684210526316
FNR: 0.6776315789473685
specificity: 0.8305921052631579
FPR: 0.16940789473684212
precision: 0.3223684210526316
NPV: 0.8305921052631579
F1-score: 0.3223684210526316

MACRO MEASURES:
accuracy: 0.7289473684210527
misclassrate: 0.2710526315789474
sensitivity: 0.3636635141861623
FNR: 0.6363364858138377
specificity: 0.8228425202379469
FPR: 0.17715747976205298
precision: 0.26313428273456324
NPV: 0.8234285301632582
F1-score: 0.28552166643004745

WEIGHTED MACRO MEASURES:
accuracy: 0.6846866343490305
misclassrate: 0.31531336565096957
sensitivity: 0.32236842105263164
FNR: 0.6776315789473685
specificity: 0.7918441801371034
FPR: 0.20815581986289658
precision: 0.2891708153285901
NPV: 0.7833030236786503

```
F1-score: 0.2852919979335258
```

```
#FITTING MULTINOMIAL CLASSIFICATION TREE WITH CHAID SPLITTING CRITERION (ORDINAL CHAID TREE)
from chefboost import Chefboost

movie_data=pandas.read_csv('./movie_data.csv')
code_gender={'M':1,'F':0}
code_member={'yes':1,'no':0}
#Note: rating is used as nominal

movie_data['gender']=movie_data['gender'].map(code_gender)
movie_data['member']=movie_data['member'].map(code_member)

X=movie_data.iloc[:,0:4].values
y=movie_data.iloc[:,4].values

#SPLITTING DATA INTO 80% TRAINING AND 20% TESTING SETS
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.20,
random_state=687088)

X_train=pandas.DataFrame(X_train, columns=['age','gender','member','nmovies'])
y_train=pandas.DataFrame(y_train, columns=['rating'])
train_data=pandas.concat([X_train, y_train], axis=1) #one-to-one concatenation

#FITTING BINARY TREE WITH CHAID SPLITTING ALGORITHM
config={'algorithm': 'CHAID', 'max_depth': 3}
tree_chaid=Chefboost.fit(train_data, config, target_label='rating')
```

```
def findDecision(obj): #obj[0]: age, obj[1]: gender, obj[2]: member, obj[3]: nmovies
# {"feature": "age", "instances": 606, "metric_value": 40.4586, "depth": 1}
if obj[0]>24.271368369871805:
# {"feature": "nmovies", "instances": 467, "metric_value": 20.0029, "depth": 2}
if obj[3]<=6:
# {"feature": "member", "instances": 458, "metric_value": 17.7061, "depth": 3}
if obj[2]>0:
# {"feature": "gender", "instances": 290, "metric_value": 15.2447, "depth":
4}
if obj[1]>0:
return 'good'
elif obj[1]<=0:
return 'bad'
else: return 'bad'
elif obj[2]<=0:
# {"feature": "gender", "instances": 168, "metric_value": 12.3326, "depth":
4}
if obj[1]>0:
return 'very good'
```

```
elif obj[1]<=0:
return 'okay'
else: return 'okay'
else: return 'very good'
elif obj[3]>6:
# {"feature": "member", "instances": 9, "metric_value": 2.5345, "depth": 3}
if obj[2]>0:
# {"feature": "gender", "instances": 7, "metric_value": 0.8165, "depth": 4}
if obj[1]<=0:
return 'very bad'
elif obj[1]>0:
return 'bad'
else: return 'bad'
elif obj[2]<=0:
return 'very bad'
else: return 'very bad'
else: return 'very bad'
elif obj[0]<=24.271368369871805:
# {"feature": "nmovies", "instances": 139, "metric_value": 17.1734, "depth": 2}
if obj[3]>3:
# {"feature": "gender", "instances": 70, "metric_value": 12.6878, "depth": 3}
if obj[1]>0:
# {"feature": "member", "instances": 36, "metric_value": 8.7214, "depth": 4}
if obj[2]>0:
return 'very good'
elif obj[2]<=0:
return 'very good'
else: return 'very good'
elif obj[1]<=0:
# {"feature": "member", "instances": 34, "metric_value": 9.7217, "depth": 4}
if obj[2]>0:
return 'very good'
elif obj[2]<=0:
return 'good'
else: return 'good'
else: return 'very good'
elif obj[3]<=3:
# {"feature": "gender", "instances": 69, "metric_value": 11.7781, "depth": 3}
if obj[1]>0:
# {"feature": "member", "instances": 36, "metric_value": 9.6067, "depth": 4}
if obj[2]>0:
return 'very good'
```

```
elif obj[2]<=0:
return 'very good'
else: return 'very good'
elif obj[1]<=0:
# {"feature": "member", "instances": 33, "metric_value": 7.5356, "depth": 4}
if obj[2]>0:
return 'good'
elif obj[2]<=0:
return 'good'
else: return 'good'
else: return 'good'
else: return 'very good'
else: return 'very good'
```

```
#COMPUTING PREDICTION ACCURACY
X_test=pandas.DataFrame(X_test, columns=['age','gender','member','nmovies'])

y_pred=[]
for i in range(len(y_test)):
    y_pred.append(Chefboost.predict(tree_chaid, X_test.iloc[i,:]))
predclass=numpy.asarray(y_pred)

#COMPUTING PERFORMANCE MEASURES FOR FITTED CHAID TREE
#turning rating into ordinal to use perf_measures function
code_rating={'very bad':1,'bad':2,'okay':3,'good':4,'very good':5}
df=pandas.DataFrame({'y_test': y_test,'predclass': predclass})
df['y_test']=df['y_test'].map(code_rating)
df['predclass']=df['predclass'].map(code_rating)
y_test=df['y_test']
predclass=df['predclass']

print()
print('PERFORMANCE MEASURES FOR FITTED CHAID TREE')
perf_measures()
```

```
PERFORMANCE MEASURES FOR FITTED CHAID TREE
CLASS MEASURES:
```

CLASS: 1
tp: 1
fp: 1
tn: 141
fn: 9
accuracy: 0.9342105263157895
misclassrate: 0.06578947368421052
sensitivity: 0.1
FNR: 0.9
specificity: 0.9929577464788732
FPR: 0.007042253521126761
precision: 0.5
NPV: 0.94
F1score: 0.16666666666666666

CLASS: 2
tp: 10
fp: 24
tn: 101
fn: 17
accuracy: 0.7302631578947368
misclassrate: 0.26973684210526316
sensitivity: 0.37037037037037035
FNR: 0.6296296296296297
specificity: 0.808
FPR: 0.192
precision: 0.29411764705882354
NPV: 0.8559322033898306
F1score: 0.32786885245901637

CLASS: 3
tp: 0
fp: 12
tn: 115
fn: 25
accuracy: 0.756578947368421
misclassrate: 0.24342105263157895
sensitivity: 0.0
FNR: 1.0
specificity: 0.905511811023622
FPR: 0.09448818897637795

precision: 0.0
NPV: 0.8214285714285714
F1score: 0.0

CLASS: 4
tp: 17
fp: 46
tn: 65
fn: 24
accuracy: 0.5394736842105263
misclassrate: 0.4605263157894737
sensitivity: 0.4146341463414634
FNR: 0.5853658536585366
specificity: 0.5855855855855856
FPR: 0.4144144144144144
precision: 0.2698412698412698
NPV: 0.7303370786516854
F1score: 0.3269230769230769

CLASS: 5
tp: 18
fp: 23
tn: 80
fn: 31
accuracy: 0.6447368421052632
misclassrate: 0.35526315789473684
sensitivity: 0.3673469387755102
FNR: 0.6326530612244898
specificity: 0.7766990291262136
FPR: 0.22330097087378642
precision: 0.43902439024390244
NPV: 0.7207207207207207
F1score: 0.4

MICRO MEASURES:
accuracy: 0.7210526315789474
misclassrate: 0.2789473684210526
sensitivity: 0.3026315789473684
FNR: 0.6973684210526315
specificity: 0.8256578947368421
FPR: 0.17434210526315788
precision: 0.3026315789473684

```
NPV: 0.8256578947368421
F1-score: 0.3026315789473684

MACRO MEASURES:
accuracy: 0.7210526315789474
misclassrate: 0.27894736842105267
sensitivity: 0.25047029109746877
FNR: 0.7495297089025312
specificity: 0.8137508344428588
FPR: 0.1862491655571411
precision: 0.30059666142879915
NPV: 0.8136837148381616
F1-score: 0.24429171920975196

WEIGHTED MACRO MEASURES:
accuracy: 0.6689750692520776
misclassrate: 0.3310249307479224
sensitivity: 0.3026315789473685
FNR: 0.6973684210526316
specificity: 0.766122593266926
FPR: 0.23387740673307394
precision: 0.29945305036862846
NPV: 0.7783224955083824
F1-score: 0.28633534103227803
```

□