Week 6 Topics

1. Chapter 9 – Classification and Regression Tree

**Classification Tree**: **Introduction**

Classification tree are machine-learning methods commonly used in data mining (specifically predictions). The goal is to create a model that predicts the value of a target variable based on several input variables. An example is shown in the figure 4.1. Each interior node corresponds to one of the input variables; there are edges (connecting lines)to children for each of the possible values of that input variable. Each leaf (a node with no child) represents a value of the target variable given the values of the input variables represented by the path from the root to the leaf.

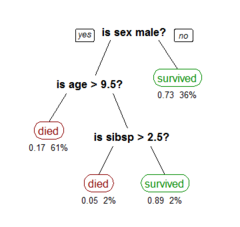


Figure 4.1

A Classification tree is a simple representation for classifying examples. For this section, assume that all of the input features have finite discrete domains, and there is a single target feature called the "classification". Each element of the domain of the classification is called a class. A decision tree or a classification tree is a tree in which each internal (non-leaf) node is labeled with an input feature. The arcs (connecting line) coming from a node labeled with an input feature are labeled with each of the possible values of the target or output feature or the arc leads to a subordinate decision node on a different input feature. Each leaf of the tree is labeled with a class.

In data mining, decision trees can be described also as the combination of mathematical and computational techniques to aid the description, categorization and generalization of a given set of data.

Data comes in records of the form:

The dependent variable, Y, is the target variable that we are trying to understand, classify, or generalize. The vector *x* is composed of the features (attributes), x1, x2, x3 etc., that are used for that task. To clarify, the vector , represents any record in our dataset.

**Classification-type problems.**

Classification-type problems are generally those where we attempt to predict values of a categorical dependent variable (class, group membership, etc.) from one or more continuous and/or categorical predictor variables. For example, we may be interested in predicting who will or will not graduate from college, or who will or will not renew a subscription. These would be examples of simple binary classification problems, where the categorical dependent variable can only assume two distinct and mutually exclusive values. In other cases, we might be interested in predicting which one of multiple different alternative consumer products (e.g., makes of cars) a person decides to purchase, or which type of failure occurs with different types of engines. In those cases there are multiple categories or classes for the categorical dependent variable. There are a number of methods for analyzing classification-type problems and to compute predicted classifications, either from simple continuous predictors (e.g., binomial or multinomial logit regression in *GLZ*), from categorical predictors (e.g., *Log-Linear analysis* of multi-way frequency tables), or both (e.g., via ANCOVA-like designs in *GLZ* or *GDA*). The *CHAID* also analyzes classification-type problems, and produces results that are similar (in nature) to those computed by *CART*. Note that various neural network architectures are also applicable to solve classification-type problems.

**Classification Tree Analytics**

In most cases, we have many attributes with many possible values. Therefore, we need a to use classification tree structure to classifying data such that the classification model helps us to classify a new set of independent variables (i.e. ) into one of the classes as accurately as possible. Keep in mind, we use Classification Trees where the target variable is categorical and the tree is used to identify the "class" within which a target variable would likely fall into.

But we use Regression Tree where the target variable is continuous and tree is used to predict it's value.

There are two key ideas underlying classification trees. The first is the idea of recursive partitioning of the space of the independent variables. The second is of pruning using validation data. In the next few sections we describe recursive partitioning, subsequent sections explain the pruning methodology.

**Recursive Partitioning**

Let us denote the dependent (categorical) variable by y and the independent variables (i.e. predictors) by . Recursive partitioning divides up the p dimensional space of the x variables into non-overlapping rectangles. This division is accomplished recursively. First one of the variables is selected, say and a value of , say is chosen to split the p dimensional space into two parts: one part is the p-dimensional hyper-rectangle that contains all the points with and the other part is the hyper-rectangle with all the points with . Then one of these two parts is divided in a similar manner by choosing a variable again (it could be xi or another variable) and a split

value for the variable. This results in three rectangular regions. (From here onwards we refer to

hyper-rectangles simply as rectangles.) This process is continued so that we get smaller and

smaller rectangles. The idea is to divide the entire x-space up into rectangles such that each

rectangle is as homogenous or ‘pure’ as possible. By ‘pure’ we mean containing points that

belong to just one class. (Of course, this is not always possible, as there may be observations that belong to different classes but have exactly the same values for every one of the independent variables.)

Let us illustrate recursive partitioning with an example.

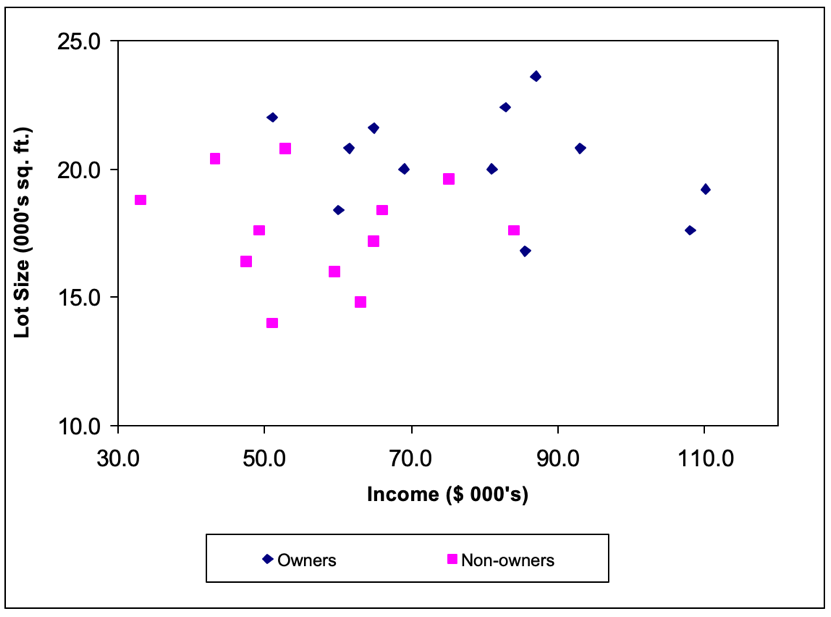
# Classification Tree Example

Example 1 (Johnson and Wichern)

A riding-mower manufacturer would like to find a way of classifying families in a city into those

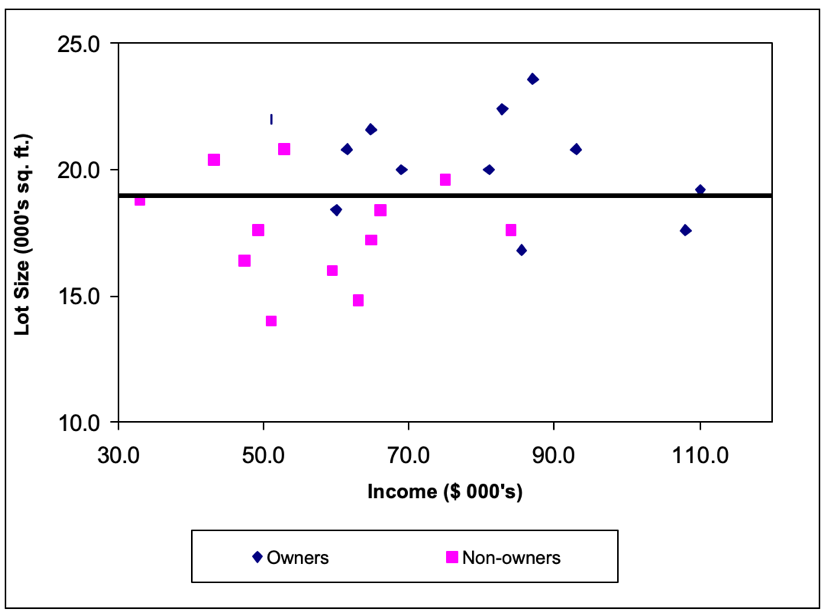
that are likely to purchase a riding mower and those who are not likely to buy one. A pilot

random sample of 12 owners and 12 non-owners in the city is undertaken. The data are shown in Table below and plotted in Figure below. The independent variables here are Income () and Lot\_Size (). The categorical y variable has two classes: owners and non-owners.



|  |  |  |  |
| --- | --- | --- | --- |
| **Observation** | **Income** | **Lot\_Size** | **Ownership** |
| 1 | 85.5 | 16.8 | Owner |
| 2 | 108 | 17.6 | Owner |
| 3 | 60 | 18.4 | Owner |
| 4 | 110.1 | 19.2 | Owner |
| 5 | 69 | 20 | Owner |
| 6 | 81 | 20 | Owner |
| 7 | 61.5 | 20.8 | Owner |
| 8 | 93 | 20.8 | Owner |
| 9 | 64.8 | 21.6 | Owner |
| 10 | 51 | 22 | Owner |
| 11 | 82.8 | 22.4 | Owner |
| 12 | 87 | 23.6 | Owner |
| 13 | 51 | 14 | Nonowner |
| 14 | 63 | 14.8 | Nonowner |
| 15 | 59.4 | 16 | Nonowner |
| 16 | 47.4 | 16.4 | Nonowner |
| 17 | 64.8 | 17.2 | Nonowner |
| 18 | 84 | 17.6 | Nonowner |
| 19 | 49.2 | 17.6 | Nonowner |
| 20 | 66 | 18.4 | Nonowner |
| 21 | 33 | 18.8 | Nonowner |
| 22 | 75 | 19.6 | Nonowner |
| 23 | 43.2 | 20.4 | Nonowner |
| 24 | 52.8 | 20.8 | Nonowner |

If we apply Classification Tree procedure (Classification And Regression Tree, CART) to this data it will choose for the first split with a splitting value of 19. The space is now divided into two rectangles, one with the Lot\_Size variable, and the other with See Figure below.

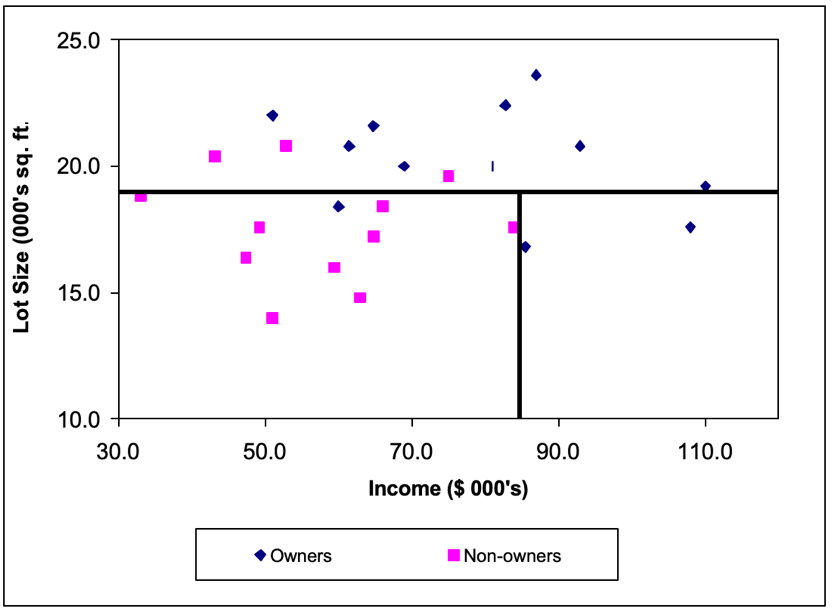


Notice how the split into two rectangles has created two rectangles each of which is much more

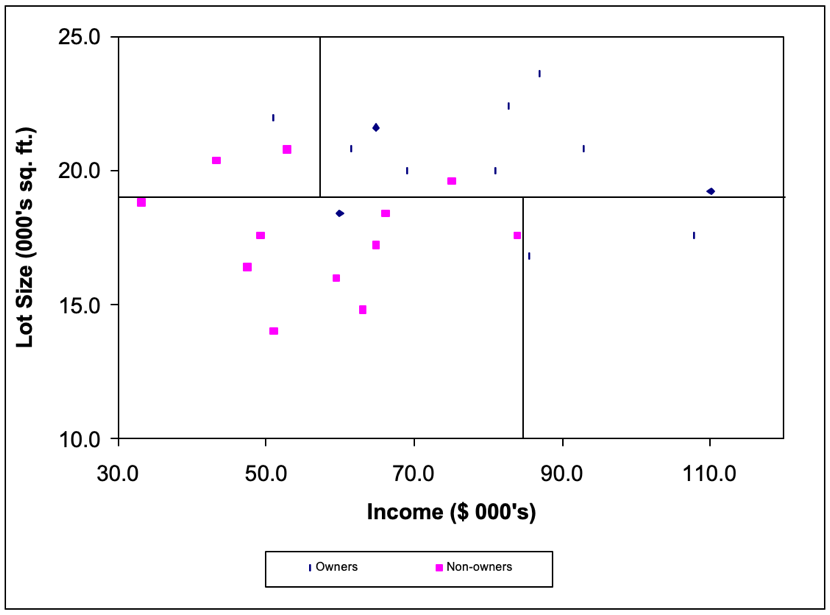
homogenous than the rectangle before the split. The upper rectangle contains points that are

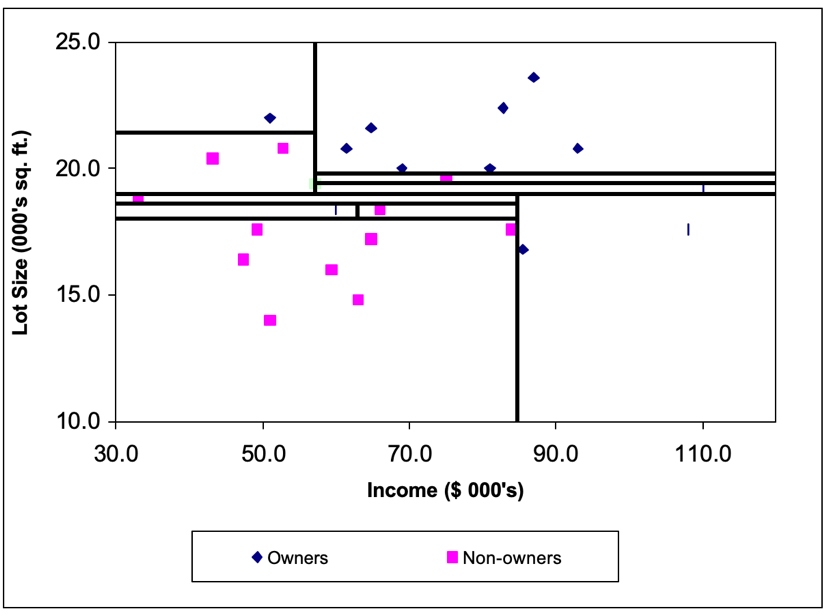
mostly owners (9 owners and 3 non-owners) while the lower rectangle contains mostly nonowners (9 non-owners and 3 owners).

How did CART decide on this particular split? We will learn in the next section how this is done however, basically it examined each variable and all possible split values for each variable to find the best split.

The next split is on the Income variable, at the value 84.75. Figure below shows that once again the CART procedure has wisely chosen to split a rectangle to increase the purity of the resulting rectangles. The left lower rectangle which contains data points with and has all but one points that are non-owners; while the right lower rectangle which contains data points (observations) with and consists exclusively of owners.

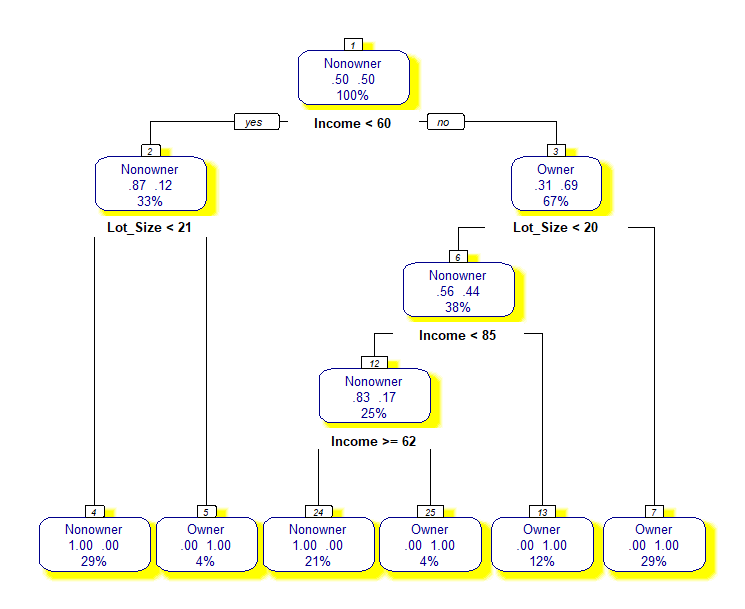
And the next split is shown below. Notice that now each rectangle is pure – it contains data points from just one of the two classes.



We can see how the recursive partitioning is refining the set of constituent rectangles to become purer as the algorithm proceeds. The final stage of the recursive partitioning is shown in Figure below.

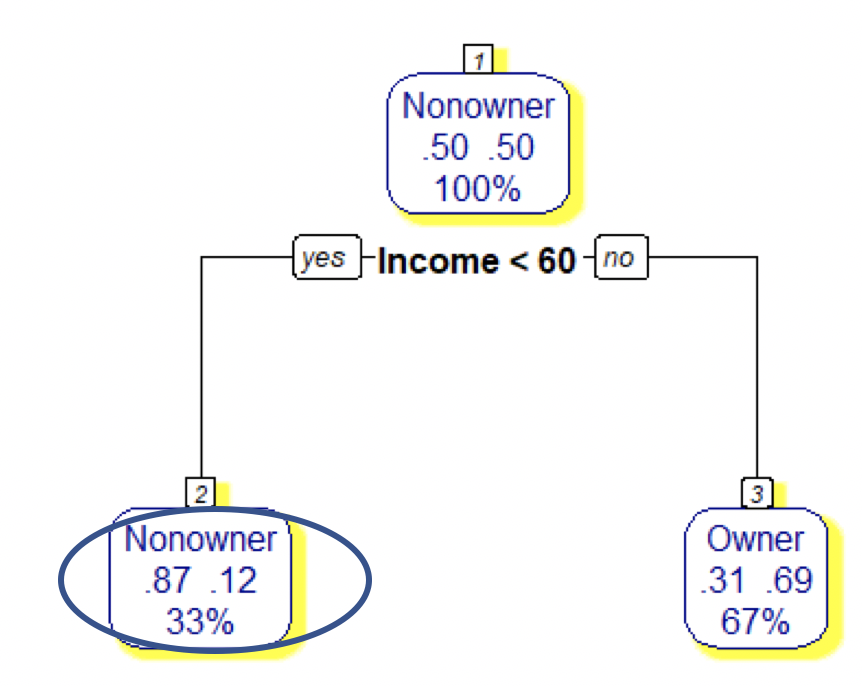
The reason the method is called a classification tree algorithm is that each split can be depicted as a split of a node into two successor nodes. The first split is shown as a branching of the root node of a tree in

The full tree is shown in Figure below. The numbers inside the leaf nodes are showing the

1. percentage of each class out of the total observations reached that leaf node.
2. the class with the highest vote is the class that we would predict for the new observation.
3. the pair numbers in the leaf node show the percentage of the observations of a class from all nodes reached that leaf. the class with the most votes in the rectangle.

In our example we didn’t partition the data to be able to validate and test the model and then use it for prediction. This is because the data is not large enough to give us a valuable information for predicting new cases. However we can try to use the classification tree we should predict

## Node Information

The following table shows the node 2 classification data. Compare the data in the table with node 2 info. That is income greater than 60.

|  |  |  |  |
| --- | --- | --- | --- |
| **Observation** | **Income** | **Lot\_Size** | **Ownership** |
| 1 | 85.5 | 16.8 | Owner |
| 2 | 108 | 17.6 | Owner |
| 3 | 60 | 18.4 | Owner |
| 4 | 110.1 | 19.2 | Owner |
| 5 | 69 | 20 | Owner |
| 6 | 81 | 20 | Owner |
| 7 | 61.5 | 20.8 | Owner |
| 8 | 93 | 20.8 | Owner |
| 9 | 64.8 | 21.6 | Owner |
| 10 | 51 | 22 | Owner |
| 11 | 82.8 | 22.4 | Owner |
| 12 | 87 | 23.6 | Owner |
| 13 | 51 | 14 | Nonowner |
| 14 | 63 | 14.8 | Nonowner |
| 15 | 59.4 | 16 | Nonowner |
| 16 | 47.4 | 16.4 | Nonowner |
| 17 | 64.8 | 17.2 | Nonowner |
| 18 | 84 | 17.6 | Nonowner |
| 19 | 49.2 | 17.6 | Nonowner |
| 20 | 66 | 18.4 | Nonowner |
| 21 | 33 | 18.8 | Nonowner |
| 22 | 75 | 19.6 | Nonowner |
| 23 | 43.2 | 20.4 | Nonowner |
| 24 | 52.8 | 20.8 | Nonowner |

There are 33% of the total observations fall into this node that is equal to . Out of these 8 nodes 87% are Nonowner that is and the rest are owner!

# How Classification Tree is Built

# Node impurity/purity in classification trees

Metrics used or measuring "best" split generally measure the homogeneity of the target variable within the subsets. These metrics are applied to each candidate subset, and the resulting values are combined (e.g., averaged) to provide a measure of the quality of the split. The unit of these measures are called measure of purity.

A node is 100% impure when a node is split evenly 50/50 and 100% pure when all of its data belongs to a single class. In order to optimize our model we need to reach maximum purity and avoid impurity

## Purity of a Node, Split, and Tree

Purity of a node is the number observations of the dominating class divided by the total number of observation in that node.

Purity of a split is the weighted average of the node in that split. Here the weighted is the number of observation in a node in the split.

We use the following example which shows the first split representing the above example. As we see in the small dataset, there are out of 24 observation, there are 12 are owners and 12 nonowners (50%). That is why the Root node purity is 50%.

To calculate the split purity, first we have to calculate the number of majority class in each node:

Fortunately, in this example the purity of each node is given by R. I just show how you calculate it if the number of observation is given instead

Left Node:

Node class: Nonowner

Total observation =

Number of nonowners =

Node Purity =

Right Node:

Node Class: Owner

Total observation =

Number of owners =

Node Purity =

Now to calculate the Purity of split we calculate the weighted average of these nodes purities

General formula of the overall purity of a classification tree is calculated a:

Where the *Ni* is the number of observations in the node *i*  and *ui* is the purity measure of the node *i,* and T is the total number of observations

## Branching Out

We will discuss two method for finding the best branching-out from a classification tree node or splitting a node for two new nodes.

* CART Algorithm – uses Gini index
* ID3 Algorithm – uses Entropy metric

# CART Algorithm and Gini Index

Classification and Regression Trees (CART) algorithm steps are:

1. Compute Gini Index for dataset in the parent node.
2. For every attribute/feature: (branching value)
   1. Calculate Gini index for categorical values (Target Majority Classes)
   2. Take weighted average of the current attribute (branching attribute, purity of split)
   3. Calculate the Gini gain
3. Pick the best Gini gain attribute (branching value)
4. Repeat until we get the tree we desired

The Gini index of a node x is calculated as shown below

Where P is the Proportion each class and k is numbers of classes.

For our example (previous page) if we have the node 2 with 1 owner and 7 Nonowners, then:

k = {1, 2} and P1 = 1/8 and P2 = 7/8

and the Gini Index for that node is,

when we use the Gini method the most impurity is (k-1)/k (k is the number of classes) and 0 when completely pure (all observations belong to the same class). In the root of our example

we have (2-1)/2 that is 0.5 or 50%

# ID3 Algorithm Entropy measure

The entropy of a node x is calculated as shown below

Where P is the Proportion each class and k is numbers of classes.

For our example (previous page) if we have the node 2 with 1 owner and 7 Nonowners, then:

k = {1, 2} and P1 = 1/8 and P2 = 7/8and the Entropy measure is

when we use the entropy method the most impurity is and 0 is when completely pure (all observations belong to the same class)

We can use any of above metrics and compute the level of impurity for a given branching. This is done by calculating the weighted average of a branching (Gini Gain).

Using any one of the above measures of impurity, the method first computes the impurity of each leaf (new nodes from the split) node at any point in time. Later it computes the overall impurity of the stage as the weighted average. Among all possible splits at a given stage, it chooses the split that yields the lowest overall impurity – which is the same as looking for highest purity, if we define purity as (1 – impurity). Considering the first split from the root:

So the overall Gini after the first split is:

We see a reduction in impurity from 0.5 to 0.36 after one split at income < 60.

Important to know that we divide by 24 total observation because the parent node was root but if for example we split the right node (3) then the divisor is 16, the total observations in that node

# Learning Algorithms and Overfitting in Classification Tree

When classification tree is fully grown, it may lose some generalization capability. This phenomenon is known as overfitting. Overfitting is the phenomenon in which the learning system tightly fits the given training data so much that it would be inaccurate in predicting the outcomes of the untrained data. In classification trees, overfitting occurs when the tree is designed so as to perfectly fit all samples in the training data set.

Possible causes of overfitting:

* Due presence of noise in data. For example, in training dataset some observations are misclassified
* Due lack of Complete observation. For example, lack of value for target variable for some observation in training dataset

Simply, overfitting means too much reliance on the training data. A validation set for a data mining model can prevent overfitting.

# Learning Algorithms

Usually a learning algorithm is trained using some set of "training data": exemplary situations for which the desired output is known. The goal is that the algorithm will also perform well on predicting the output when fed "validation data" that was not encountered during its training.

*Ockham's razor’s principle: "simpler solutions are more likely to be correct than complex ones."*

Overfitting is the use of models or procedures that violate *Ockham's razor*, for example by including more adjustable parameters than are ultimately optimal, or by using a more complicated approach than is ultimately optimal. For an example where there are too many adjustable parameters, consider a dataset where training data for *y* can be adequately predicted by a linear function of two dependent variables. Such a function requires only three parameters (the intercept and two slopes). Replacing this simple function with a new, more complex quadratic function, or with a new, more complex linear function on more than two dependent variables, carries a risk: Ockham's razor implies that any given complex function is *a priori* (a belief) less probable than any given simple function. If the new, more complicated function is selected instead of the simple function, and if there was not a large enough gain in training data fit to offset the complexity increase, then the new complex function "overfits" the data, and the complex overfitted function will likely perform worse than the simpler function on validation data outside the training dataset, even though the complex function performed as well, or perhaps even better, on the training dataset.

When comparing different types of models, complexity cannot be measured solely by counting how many parameters exist in each model; the expressivity of each parameter must be considered as well. For example, it is nontrivial to directly compare the complexity of a neural net (which can track curvilinear relationships) with *m* parameters to a regression model with *n* parameters.

Overfitting is especially likely in cases where learning was performed too long or where training examples are rare, causing the learner to adjust to very specific random features of the training data, that have no causal relation to the target function. In this process of overfitting, the performance on the training examples still increases while the performance on unseen data becomes worse.

As you see in this tree, all leaves are 100% pure!

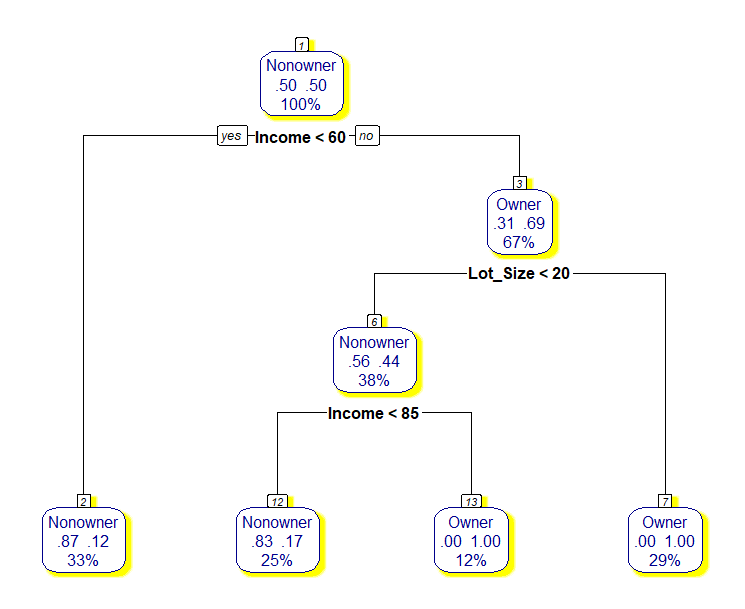
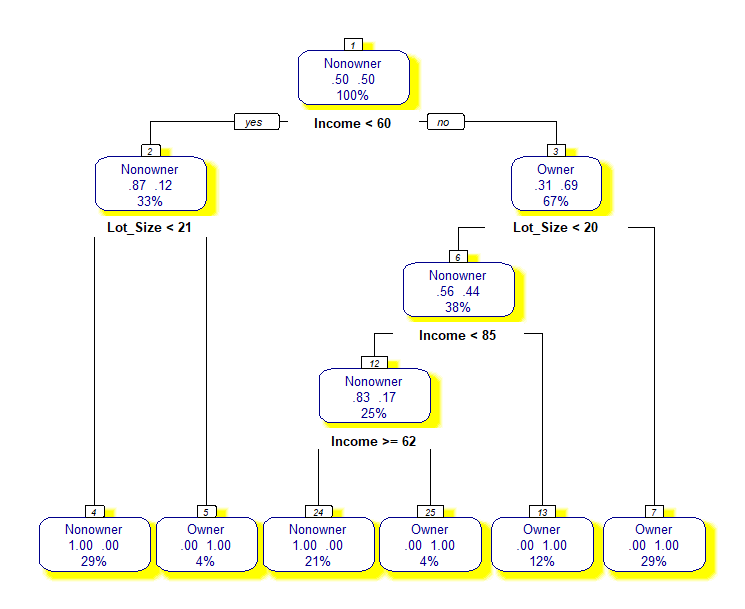
It is useful to note that the type of trees grown by CART (called binary trees) have the property that the number of leaf nodes is exactly one more than the number of classifier nodes.

# Classification Tree pruning

Decision trees that are trained on any training data run the risk of overfitting the training data.

What I mean by this is that eventually each leaf will represent a very specific set of attribute combinations that are seen in the training data, and the tree will consequently not be able to classify attribute value combinations that are not seen in the training data. In order to prevent this from happening, we must prune the decision tree.

By pruning we mean that the lower ends (the leaves) of the tree are “snipped” until the tree is much smaller. The figure below left shows our example of a full tree, and the same tree on the right after it has been pruned to have only 4 leaves.



As you see the left tree leaves are 100% pure (identify only one class). That means our model is an overfit one!

However, pruning is not done when we have an overfitted model, we also prune a tree which is extremely large. On real-life data, the complete tree will almost always be huge. Sure, it will probably have a low error rate – but it will be very cumbersome to use. Furthermore, it will likely overfit the data. On the other hand if complexity factor is very high, then even a single split will increase the cost a lot and so the best tree will be just root node alone and classify every case as simply the class of the majority case in the training data using what we earlier in KNN called Naïve rule!

Several ways we can stop the tree grow but adding the minsplit and minbucket parameters to the rpart() function is R. here is the definition of these parameters:

* minsplit: specifies the minimum number of observation in a node we consider for split.

That is if the node we set minsplit to 10, then if a node has less than 10 won’t be split.

* minbucked: specifies that no leaf node can have more the identified value

That means setting minbucket to 10 ensures that no leaf node has less than 10 observations

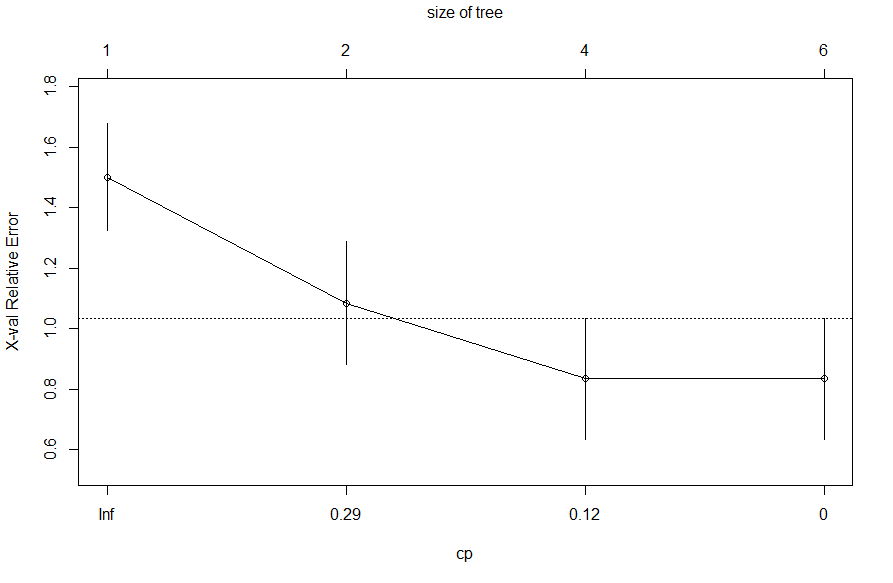
* maxdepth =
* Use the complexity factor.

For the above figure, I used minsplit equal to 5.

## Pruning by Complexity parameters

When rpart() runs, other than generating the tree it goes through an additional process called *cross validation.* In that process, it generates trees of various sizes (number of leaf nodes) and compute the average and standard deviation of the error for trees of each size. In addition it also compute the complexity factor that leads to trees of various sizes. We can use this information to prune the tree. Here are the steps:

1. Generate a plot of cp() and tree size versus the error rate. Actually, since the error rate will vary vastly from situation to situation, we will plot the relative error rate. This is just a scaled version of the actual error rate with the highest error rate (for a tree with just one node) being converted to 1 and all the other error rates proportionally scaled. I create the CP plot using the codes generate the complete tree for the mowers dataset of our previous example



The cp plot shows that the minimum size of tree is 1 node (root node with no split) and the maximum of size is 6 nodes which is the number leaf nodes in our complete tree (left one).

We can also print the cp information by using printcp(model name) function. The following are the output of cp for out tree.

Classification tree:

rpart(formula = Ownership ~ ., data = mower.df, minsplit = 2, cp = 0)

Variables actually used in tree construction:

[1] Income Lot\_Size

Root node error: 12/24 = 0.5

n= 24

CP nsplit rel error xerror xstd

1 0.500000 0 1.00000 1.50000 0.17678

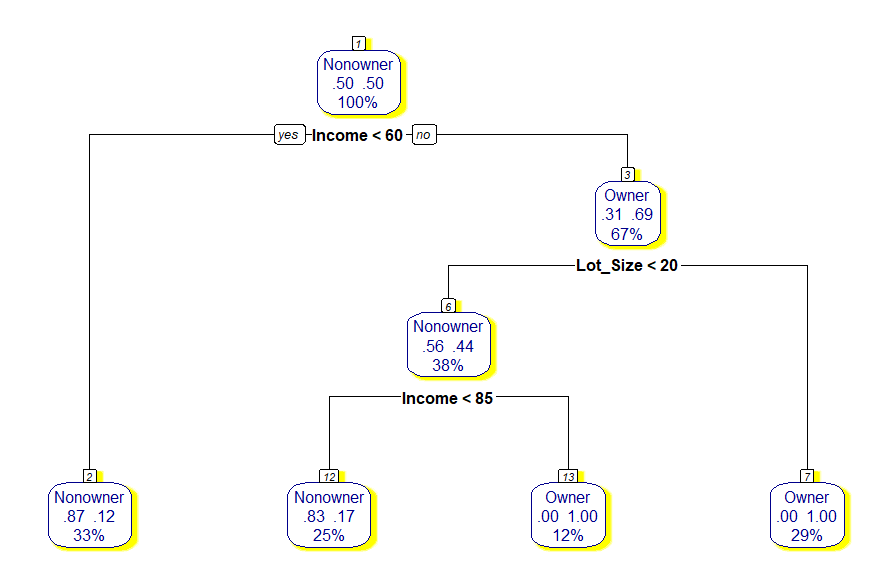
2 0.166667 1 0.50000 1.08333 0.20341

3 0.083333 3 0.16667 0.83333 0.20127

4 0.000000 5 0.00000 0.83333 0.20127

1. We select the tree size with lowest error rate. In practice we allow for some random variation and select the tree that corresponds to the leftmost point on or below the dashed line. This will correspond to a tree of smaller size whose error is within one standard deviation of the error of the smallest tree.
2. Run the R prune() function with tree model and cp equal to error. In my example I chose 0.12.

The following figure shows the prune tree with size 4 and having smallest error. Surprising, this is a tree I created with minsplit = 5!



1. Classification Tree Analytics with R

To classify a categorical target variable in classification tree, you should:

* Install and activate the *rpart* and *rpart.plot* packages
* Loading the data into Rstudio/R environment
* Invoke the classification tree function with the following arguments:
  1. Target and predictor attributes
  2. The source dataset name
  3. The tree size

# Explanation of R Code for Classification tree

As we learned in this paper, recursive partitioning is a fundamental tool in data mining. It helps us explore the structure of a set of data, while developing easy to visualize decision rules for predicting a categorical (classification tree) or continuous (regression tree) outcome. This section briefly describes R **CART** modeling functions, conditional inference trees, and random forests.

## CART Modeling via rpart package

The CART algorithm can be generated through the rpart package. Detailed information on rpart is available in “An Introduction to Recursive Partitioning Using the RPART package routines”. The general steps are provided below followed by my codes for example is used in this paper.

* Grow the Tree

To grow a tree, use:

rpart(formula, data=, method=,control=) where

|  |  |
| --- | --- |
| formula | is in the format  *outcome* ~ *predictor1*+*predictor2*+*predictor3*+ect. |
| data = | specifies the data frame |
| method= | **"class"** for a classification tree  **"anova"** for a regression tree |
| control= | optional parameters for controlling tree growth. For example, control=rpart.control(minsplit=30, minbucket = 10, cp=0.001) requires that the minimum number of observations in a node be 30 before attempting a split, minbucket will ensure that no leaf node has less than 10 observations, and that a split must decrease the overall lack of fit by a factor of 0.001 (cost complexity factor) before being attempted. |

* The following functions help us to examine the results

*Prp() and rpart.plot()* both provide the tree visualization. The *prp()* is more detailed.

##\_\_\_\_\_\_\_\_\_ R Codes for mower ownership example\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

library(rpart)

library(rpart.plot)

library(randomForest)

library(xgboost)

library(Matrix)

library(caret)

library(gains)

mower.df <- read.csv("Mower.csv")

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Build a model, get the CP and visualize\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mower.tr <- rpart(Ownership ~ ., data = mower.df, minsplit = 2, cp = 0, method = “class”)

plotcp(mower.tr)

printcp(mower.tr)

prp(mower.tr, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE, faclen = 4, varlen = 8, col = "darkblue", shadow.col = "yellow")

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ build a model, prune it with specified minsplit and visualize\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mower.tr2 <- rpart(Ownership ~ ., data = mower.df, minsplit = 5, cp = 0, method = “class”)

prp(mower.tr2, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE, faclen = 4, varlen = 8, col = "darkblue", shadow.col = "yellow")

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Prune the first model, prune it with selected CP value and visualize\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mower.tr.pruned <- prune(mower.tr, cp = 0.12)

prp(mower.tr.pruned, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE, faclen = 4, varlen = 8, col = "darkblue", shadow.col = "yellow")

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Prune the first model, prune it with specified minbucket and visualize\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mower.tr3 <- rpart(Ownership ~ ., data = mower.df, minbucket = 6, cp = 0)

prp(mower.tr3, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE, faclen = 4, varlen = 8, col = "darkblue", shadow.col = "yellow")

Note: in the above procedure, I didn’t create data partitions, didn’t test the models, and didn’t predict new observations class.

## Complete R code sequence for classification tree algorithm procedure

**Reading data and exploring:**

bh<-read.csv("boston-housing-classification.csv")

summary(bh$MEDV\_CAT)

**Data Prtitioning:**

Library(caret)

set.seed(2021)

sam<-createDataPartition(bh$MEDV\_CAT, p=0.7, list = FALSE)

train<-bh[sam,]

text<-bh[-sam,]

**Building Tree (on training dataset):**

library(rpart)

bh.tree<-rpart(MEDV\_CAT ~ ., data = train, control = rpart.control(minbucket = 10, cp=0))

**Visualizing Tree:**

library(rpart.plot)

prp(bh.tree, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE, faclen = 4, varlen = 8, shadow.col = "green")

**Predicting Training and Test Dataset and Create CM:**

pred.train<-predict(bh.tree, train, type="class")

table(train$MEDV\_CAT,pred.train, dnn=c("Actual", "Predicted"))

pred.test<-predict(bh.tree, test, type="class")

table(test$MEDV\_CAT, pred.test, dnn=c("Actual", "Predicted"))

**Building a Larger Tree:**

bh.tree.B<-rpart(MEDV\_CAT ~ ., data = train, control = rpart.control(minsplit = 10, cp=0))

prp(bh.tree.B, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE, faclen = 4, varlen = 8, shadow.col = "red")

**Pruning Tree**

printcp(bh.tree.B)

plotcp(bh.tree.B) //Visualize the complexity parameters)

bh.pruned<-prune(bh.tree.B, 0.018)

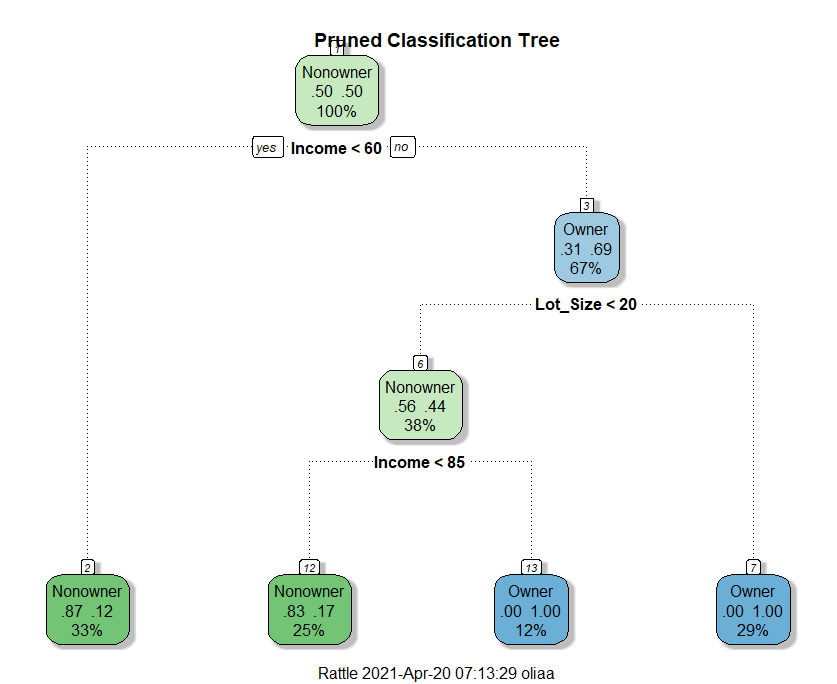
**Visualizing Tree:**

prp(bh.pruned, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE, faclen = 4, varlen = 8, shadow.col = "gray")

## Use rattle package build, prune and plot the tree

To enhance our code and visualization, let us take some help from rattle :

Rattle() is one unique feature of R which is specifically built for data mining in R. It provides its own GUI apart from the R Console which makes it easier to analyze data. It has built-in graphics, which provides us better visualizations as well.

Here are sample codes and visualization. Prune the tree to create an optimal decision tree (All in one!)

library(rattle)

ptree<- prune(tree, cp= tree$cptable[which.min(tree$cptable[,"xerror"]),"CP"])

fancyRpartPlot(ptree, uniform=TRUE, main="Pruned Classification Tree")

**Factors** are used to represent categorical data. **Factors can** be ordered or unordered and are an important **class** for statistical analysis and for plotting. ... While **factors** look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.