Week 4 Topics

1. Chapter 7 – Classification

# Introduction

There are two forms of data analysis that can be used for extracting models describing important classes or to predict future data trends. These two forms are as follows:

* Classification
* Prediction

Some academic data analytics use the following forms:

* Regression for continuous numeric data (i.e. similar to the Prediction)
* Logistic Regression for discrete data
* Bayesian Classification for categorical data such as KNN

Classification models predict categorical class labels; and prediction models predict continuous valued functions. For example, we can build a classification model to categorize bank loan applications as either safe or risky, or a prediction model to predict the expenditures in dollars of potential customers on computer equipment given their income and occupation.

# What is classification?

Following are the examples of cases where the data analysis task is Classification:

* A bank loan officer wants to analyze the data in order to know which customer (loan applicant) are risky or which are safe.
* A marketing manager at a company needs to analyze a customer with a given profile, who will buy a new computer.

In both of the above examples, a model or classifier is constructed to predict the categorical labels. These labels are risky or safe for loan application data and yes or no for marketing data.

# What is prediction?

Following are the examples of cases where the data analysis task is Prediction

* Suppose the marketing manager needs to predict how much a given customer will spend during a sale at his company. In this example we are bothered to predict a numeric value. *Therefore the data analysis task is an example of numeric prediction*. In this case, a model or a predictor will be constructed that predicts a continuous-valued-function or ordered value.

Note − Regression analysis is a statistical methodology that is most often used for numeric prediction.

## How Does Classification Works?

With the help of the bank loan application that we have discussed above, let us understand the working of classification. The Data Classification process includes two steps :

* Building the Classifier or Model
* Using Classifier for Classification

# Building the Classifier or Model

* This step is the learning step or the learning phase.
* In this step the classification algorithms build the classifier.
* The classifier is built from the training set made up of database observations (records) and their associated class labels (target column).
* Each observation that constitutes the training set is referred to as a category or class. These tuples can also be referred to as sample, object, observations, or data points.

# Using Classifier for Classification

In this step, the classifier is used for classification. Here the test data is used to estimate the accuracy of classification rules. The classification rules can be applied to the new data tuples if the accuracy is considered acceptable.

# Classification and Prediction Issues

The major issue is preparing the data for Classification and Prediction. Preparing the data involves the following activities:

* Data Cleaning − Data cleaning involves removing the noise and treatment of missing values. The noise is removed by applying smoothing techniques and the problem of missing values is solved by replacing a missing value with most commonly occurring value for that attribute.
* Relevance Analysis − Database may also have the irrelevant attributes. Correlation analysis is used to know whether any two given attributes are related.
* Data Transformation and reduction − The data can be transformed by any of the following methods.
  + Normalization − The data is transformed using normalization. Normalization involves scaling all values for given attribute in order to make them fall within a small specified range. Normalization is used when in the learning step, the neural networks or the methods involving measurements are used.
  + Generalization − The data can also be transformed by generalizing it to the higher concept. For this purpose we can use the concept hierarchies.

Note − Data can also be reduced by some other methods such as wavelet transformation, binning, histogram analysis, and clustering.

## Comparison of Classification and Prediction Methods

Here is the criteria for comparing the methods of Classification and Prediction

* **Accuracy** − Accuracy of classifier refers to the ability of classifier. It predict the class label correctly and the accuracy of the predictor refers to how well a given predictor can guess the value of predicted attribute for a new data.
* **Speed** − This refers to the computational cost in generating and using the classifier or predictor.
* **Robustness** − It refers to the ability of classifier or predictor to make correct predictions from given noisy data.
* **Scalability** − Scalability refers to the ability to construct the classifier or predictor efficiently; given large amount of data.
* **Interpretability** − It refers to what extent the classifier or predictor understands.

1. K-Nearest Neighbor (*K-NN*) for classification

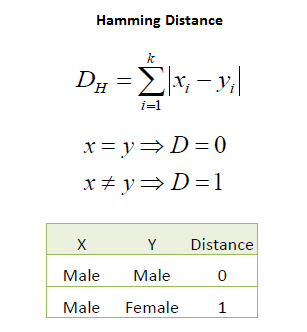
K nearest neighbors is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions). KNN has been used in statistical estimation and pattern recognition already in the beginning of 1970’s as a non-parametric technique

## Algorithm

A case is classified by a majority vote of its neighbors, with the case being assigned to the class most common amongst its K nearest neighbors measured by a distance function. If K = 1, then the case is simply assigned to the class of its nearest neighbor



It should also be noted that all three distance-measures are only valid for continuous variables. In the instance of categorical variables the Hamming distance must be used. It also brings up the issue of standardization of the numerical variables between 0 and 1 when there is a mixture of numerical and categorical variables in the dataset



Choosing the optimal value for K is best done by first inspecting the data. In general, a large K value is more precise as it reduces the overall noise but there is no guarantee. Cross-validation is another way to retrospectively determine a good K value by using an independent dataset to validate the K value. Historically, the optimal K for most datasets has been between 3-10.

## Example:

Consider the following data (Figure 3 visualization and Figure 4 table) concerning credit default. Age and Loan are two numerical variables (predictors) and Default is the target

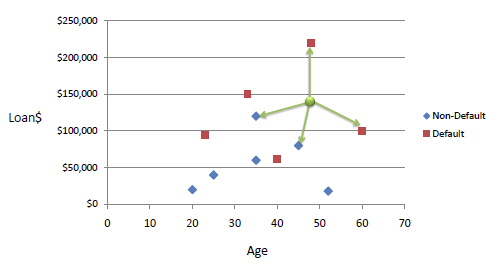


Figure 3: Loa. E fault data visualization

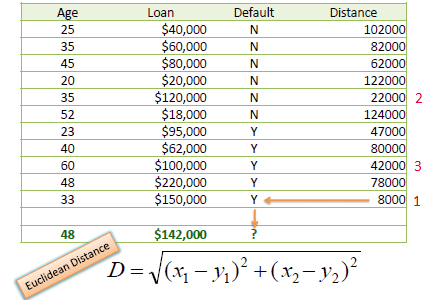


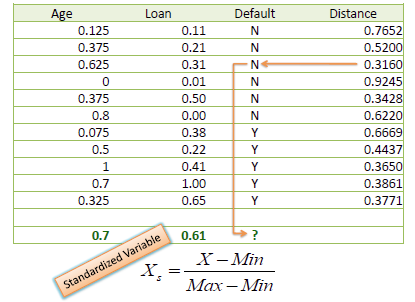
Figure 4: Loan Default data Table

We can now use the training set to classify an unknown case (Age=48 and Loan=$142,000) using **Euclidean distance. If K=1 then the nearest neighbor is the last case in the training set with Default=Y.**

D = Sqrt [(48-33)2 + (142000-150000)2] = 8000.01  >> Default=Y

## Standardized Distance

One major drawback in calculating distance measures directly from the training set is in the case where variables have different measurement scales or there is a mixture of numerical and categorical variables. For example, if one variable is based on annual income in dollars, and the other is based on age in years then income will have a much higher influence on the distance calculated. One solution is to standardize the training set as shown below.



Two methods for adjusting the values to same weigh in computing the distance, Adjusting to (0-1) range and Standardize

* Adjust the range to 0-1:
* Standardize:

Where Xi is a value in the dataset column, Max and Min are maximum and minimum values in that column, and SD is the standard deviation of the values in the column.

In our example, using the standardized distance on the same training set, the unknown case returned a different neighbor, which is not a good sign of robustness.

## Model Evaluation

By Partitioning the data into the training and evolution datasets, we can verify confusion matrix measures. Then calculate the cost of mistakes

## KNN Example

Here an example for KNN algorithm

* Opening the vacation-trip-classification.csv file

vac <- read.csv(“vacation-trip-classification.csv”)

* Explore the data

Create boxplot and histogram for income, family size attributes and boxplot for Income vs Result as well as family\_size vs Result

* Interpret the outcome of the summary exploration
* Standardize the Income and Family\_size.
* Load data partitioning package. Partition the data into 60% and 40% and then partition the 40% partition into two 50% partitions. We should have three new datasets: train.a, train.b, and test
* Load the calssification package.
* Run the KNN algorithm for each target value in train.b using train.a data. Use k = 1and 3. You should. Create confusion matrix for each k. You should have the following confusion matrix:

K = 1 K = 3

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Predicted | |
|  |  | Buyer | Non-buyer |
| Actual | Buyer | 4 | 0 |
| Non-buyer | 0 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Predicted | |
|  |  | Buyer | Non-buyer |
| Actual | Buyer | 3 | 1 |
| Non-buyer | 0 | 4 |

* Interpret the model
* Test the model on dataset which was not used to create and selection of the model. Run the KNN algorithm for each target value in test using train.a data. Use k = 1and 3. You should. Create confusion matrix for each k. You should have the following confusion matrix:

K = 1 K = 3

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Predicted | |
|  |  | Buyer | Non-buyer |
| Actual | Buyer | 2 | 2 |
| Non-buyer | 1 | 2 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Predicted | |
|  |  | Buyer | Non-buyer |
| Actual | Buyer | 1 | 3 |
| Non-buyer | 0 | 3 |

## Attribute Conversion

## Different data types:

* Continuous:
  + Numeric (e.g., salaries, ages, temperatures, rainfall, sales).
* Discrete:
  + Binary (0 or 1; Yes/No; Male/Female)
  + Boolean (True/False)
  + Specific list of allowed values (e.g., zip codes; country names; chemical elements; amino acids; planets)
* Categorical:
  + Non-numeric (character/text data) (e.g., people’s names)
  + Ordinal (ordered) or Nominal (not ordered)

## Changing categorical to numeric

For every possible category we create a new binary attribute. If that category presents in the attribute the new binary attribute value is 1 otherwise is zero. For example in a table we have a field called fuel type. Fuel type can be Petrol, Diesel, and CNG. To convert this categorical field into binary numeric, we create 3 new columns, Diesel, Petrol, and CNG. If a car model (a row) uses diesel, then in the diesel column we put 1 and others zero. Figure 3 shows this transformation in a sample dataset

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | Fuel\_Type | Diesel | Petrol | CNG |
| TOYOTA Corolla 2.0 D4D HATCHB TERRA 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 2.0 D4D HATCHB TERRA 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 2.0 D4D HATCHB TERRA 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 2.0 D4D HATCHB TERRA 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 2.0 D4D HATCHB SOL 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 2.0 D4D HATCHB SOL 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 2.0 D4D 90 3DR TERRA 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 2.0 D4D 90 3DR TERRA 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 1.6 16V VVT I LIFTB TERRA 4/5-Doors | CNG | 0 | 0 | 1 |
| TOYOTA Corolla 1800 T SPORT VVT I 2/3-Doors | Petrol | 0 | 1 | 0 |
| TOYOTA Corolla 1.9 D HATCHB TERRA 2/3-Doors | Diesel | 1 | 0 | 0 |
| TOYOTA Corolla 1.8 VVTL-i T-Sport 3-Drs 2/3-Doors | Petrol | 0 | 1 | 0 |
| TOYOTA Corolla 1.8 16V VVTLI 3DR T SPORT BNS 2/3-Doors | Petrol | 0 | 1 | 0 |
| TOYOTA Corolla 1.8 16V VVTLI 3DR T SPORT 2/3-Doors | Petrol | 0 | 1 | 0 |
| TOYOTA Corolla 1.8 16V VVTLI 3DR T SPORT 2/3-Doors | CNG | 0 | 0 | 1 |
| TOYOTA Corolla 1.8 16V VVTLI 3DR T SPORT 2/3-Doors | Petrol | 0 | 1 | 0 |
| TOYOTA Corolla 1.8 16V VVTLI 3DR T SPORT 2/3-Doors | Petrol | 0 | 1 | 0 |

Figure 5: Car models fuel type category transformed into binary

## Numeric to Categorical

There are several ways to do that. One way is creating a bin for every data range and the name of bin will be the category for that type of data. In the following section I will demonstrate it through an example from our textbook.

## Data Conversion

* Dataset used: conversion.csv
* Code in sequence of use:
  + **Categorical to Numeric**

st<-read.csv("conversion.csv")

library(dummies)

st<-cbind(st, dummy(st$State, sep="-"))

names(st)

[1] "Age" "State" "Height" "Income" "st-NJ" "st-NY" "st-TX" "st-VA"

write.csv(st, file= "catconversion.csv")

* + **Numeric to Categorical**

breaks<-c(-Inf, 10000, 31000, Inf)

names=c("LOW", "Medium", "High")

st$Income\_cat = cut(st$Income, breaks, names)

write.csv(st, file="NumVConversion,csv”

## R Code

## K nearest neighbor (K-NN)

* Dataset used: vacation-trip-classification.csv
* Code in sequence of use:

vac <- read.csv(“vacation-trip-classification.csv”)

boxplot(Income ~ Result, data = vac, ylab = "Income")

boxplot(Familyc\_size ~ Result, data = vac, ylab = "Family\_size")

library(caret)

set.seed(2015)

samp<-createDataPartition(vac$Result, p = 0.6, list = FALSE)

train<-vac[samp,]

rest<-vac[-samp,]

samp<-createDataPartition(rest$Result, p = 0.5, list = FALSE)

eval<-rest[samp,]

test<-rest[-samp,]

library(class)

eval$pred.1<-knn(train[,4:5], eval[,4:5], train[,3], 1)

tab.1<-table(eval$Result,eval$pred.1, dnn = c("Actual", "Predicted"))

tab.1

Predicted

Actual Buyer Non-buyer

Buyer 4 0

Non-buyer 0 4

**eval$pred.3<-knn(train[,4:5], eval[,4:5], train[,3], 3)**

tab.3<-table(eval$Result, eval$pred.3, dnn = c("Actual", "Predicted"))

> tab.3

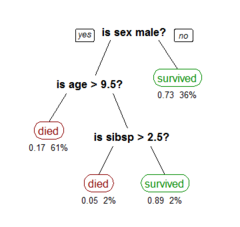
Predicted

Actual Buyer Non-buyer

Buyer 3 1

Non-buyer 0 4

1. Classification Tree

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.

In this figure three attributes values are evaluated:

* Passenger Gender
* Passenger Age
* Passenger number of siblings

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), 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 (hopefully you will all be classify as pass), 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 *GLZ1*), from categorical predictors (e.g., *Log-Linear analysis* of multi-way frequency tables), or both (e.g., via ANCOVA-like designs in *GLZ* or *GDA2*). The *CHAID* also analyzes classification-type problems, and produces results that are similar (in nature) to those computed by *C&RT*. Note that various neural network architectures are also applicable to solve classification-type problems.

1. 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.

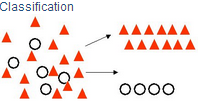


Figure 2: Classification Figure 3: Regression

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

Footnotes:

1. The Generalized Linear/Nonlinear Models (GLZ) module is a comprehensive implementation of the General Linear Model. Both linear and nonlinear effects for any number and type of predictor variables on a discrete or continuous dependent variable can be analyzed.
2. GDA, is a method for data classification commonly used when data can be approximated with a Normal distribution. As first step, you will need a training set, i.e. a bunch of data yet classified
3. Classification Tree Analytics with R

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

* 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

Example: Luxury ownership case

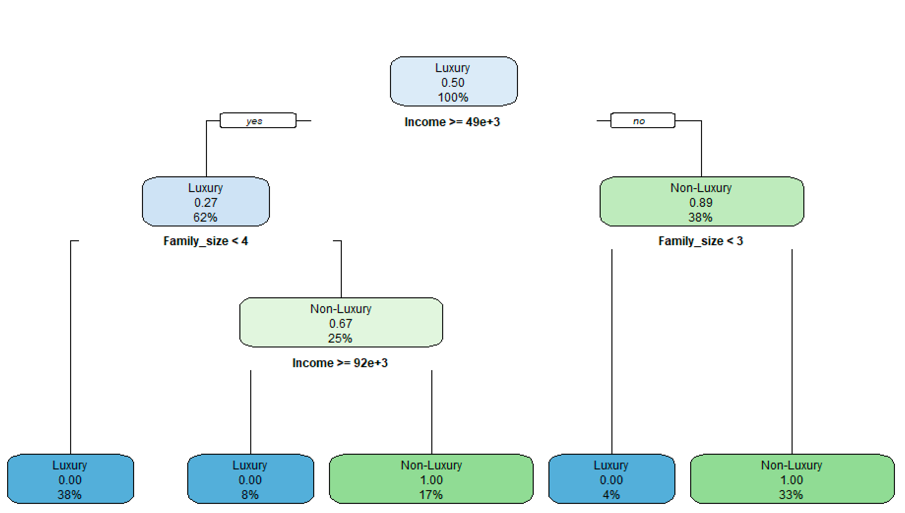
* Load luxurty\_car\_obership.csv file
* Use the code in the Luxury\_Car\_Ownership R-Codes.txt to create decision tree

Figure below is the classification tree for the luxury car case exercise

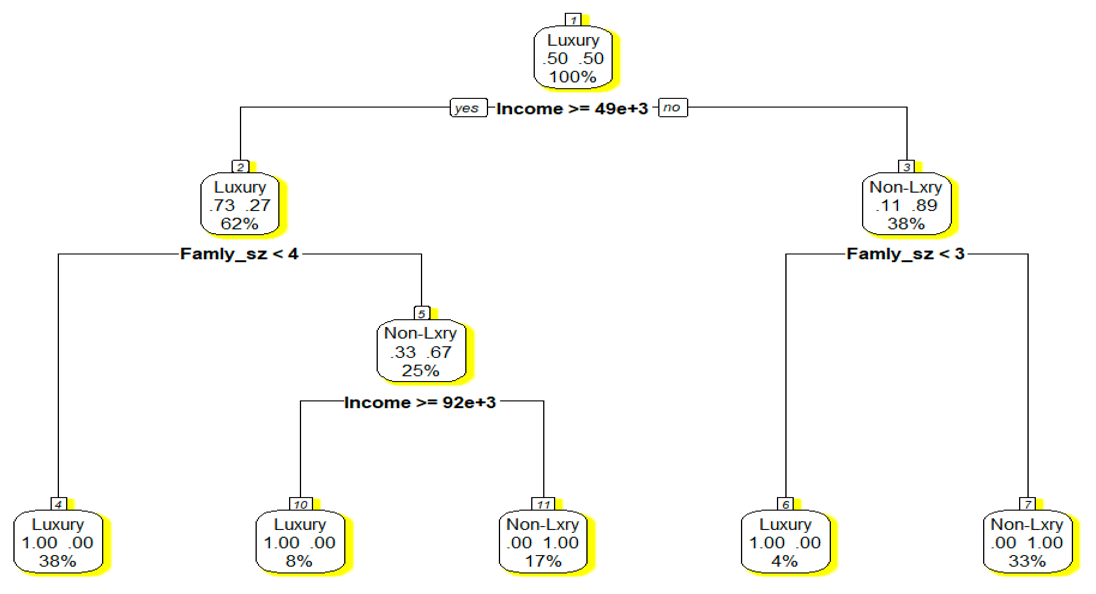
# Classification tree information

R classification tree function nodes provide the following information:

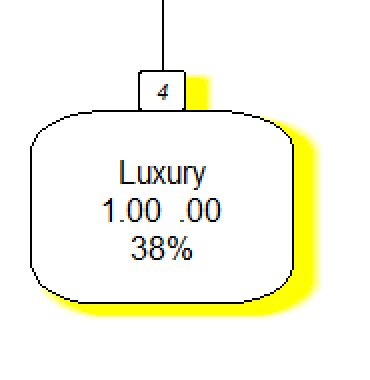
1. Node classification – based on majority class of cases meeting the condition to get to the node from the root node – For example in the left child of the root, the *Luxury* car is the majority class
2. Proportion of cases in the node – For example in the left child of the root, the 0.27 is the proportion of *Non-Luxury* car (the 0.73 is the proportion of the *Luxury* car).

Note: the *rpart.plot()* display the three nodes with one class proportion in all nodes regardless the node class. In the above exercise, all proportions are for *Non-Luxury*. The *prp()* plotting function also plots the tree but nodes show both class proportions.

1. Percentage of total cases meeting the condition for that node – For example in the left child of the root, the 62% of the total observation (i.e. 24) reach this node
2. Condition to branch further from the node. For example in the left child of the root, the condition to branch is Family\_size < 4
3. The R *prp()* function provide a better detailed classification tree info.



The following table shows the node 4 classification data. Compare the data in the table with node 4 info.



|  |  |  |  |
| --- | --- | --- | --- |
| **RID** | **Income** | **Family\_size** | **Car\_type** |
| 1 | 89800 | 1 | Luxury |
| 2 | 47500 | 4 | Non-Luxury |
| 3 | 45000 | 3 | Non-Luxury |
| 4 | 44700 | 2 | Luxury |
| 5 | 59500 | 3 | Luxury |
| 6 | 36100 | 3 | Non-Luxury |
| 7 | 63300 | 4 | Non-Luxury |
| 8 | 52900 | 4 | Non-Luxury |
| 9 | 78200 | 3 | Luxury |
| 10 | 145100 | 4 | Luxury |
| 11 | 88600 | 6 | Non-Luxury |
| 12 | 65600 | 3 | Luxury |
| 13 | 44500 | 3 | Non-Luxury |
| 14 | 94600 | 4 | Luxury |
| 15 | 59400 | 3 | Luxury |
| 16 | 47300 | 3 | Non-Luxury |
| 17 | 72100 | 2 | Luxury |
| 18 | 83000 | 1 | Luxury |
| 19 | 64100 | 4 | Non-Luxury |
| 20 | 42100 | 3 | Non-Luxury |
| 21 | 91500 | 3 | Luxury |
| 22 | 51200 | 3 | Luxury |
| 23 | 13800 | 3 | Non-Luxury |
| 24 | 47500 | 3 | Non-Luxury |

9 row match the condition of the node 4, so 9/24 = %37.5

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, as you see we can try to use the classification tree we should predict.

1. 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 example in the previous page. As we see in the small dataset, there are out of 24 observation, there are 12 are Luxury and 12 Non-Luxury (50%, 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:

Left 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

Total observation =

Number of Luxury =

Node Purity =

Right Node:

Total observation =

Number of Non-Luxury =

Node Purity =

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

The overall purity of a classification tree is calculated as:

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
2. For every attribute/feature: (branching value)
   1. Calculate Gini index for categorical values (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 example if we have a node x with 4 Luxury and 11 Non-luxury, then:

k = {1, 2} and P1 = 4/15 and P2 = 11/15

and the Gini Index,

when we use the Gini method the most impurity is (k-1)/k and 0 when completely pure (all observations belong to the same class)

# 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 example if we have if we have a node x with 4 Luxury and 11 Non-luxury, then:

k = {1, 2} and P1 = 4/15 and P2 = 11/15

and 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 0.3185 after one split at income >= 49e+10.

# Explanation of R Code for Classification tree

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 **CART** modeling, 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 two examples.

* 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. |

* Examine the results

The following functions help us to examine the results

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

1. 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.[[](https://en.wikipedia.org/wiki/Overfitting#cite_note-hawkins-11)

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.

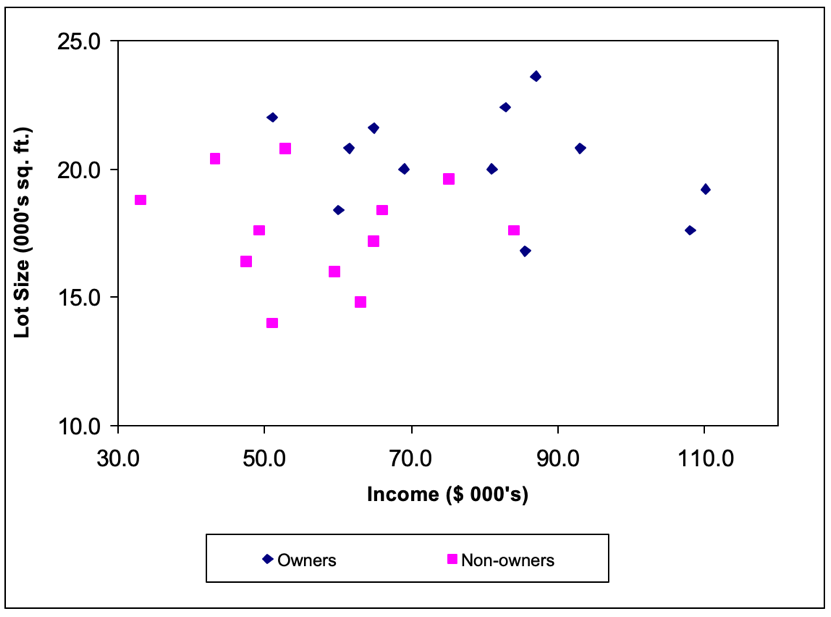
# Classification Tree Example

Example 1 (Johnson and Wichern)

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

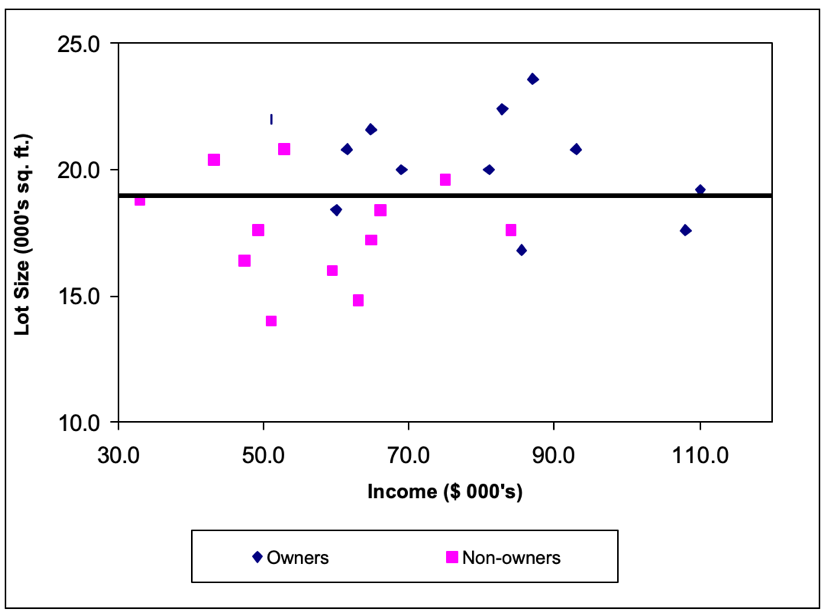
that are likely to purchase a riding mower and those who are not likely to buy one. A pilot

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 (x1) and Lot\_Size (x2). 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 to this data it will choose x2 for the first split with a splitting value of 19. The (x1,x2) space is now divided into two rectangles, one with the Lot\_Size variable, x2 <= 19 and the other with x2 > 19. 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

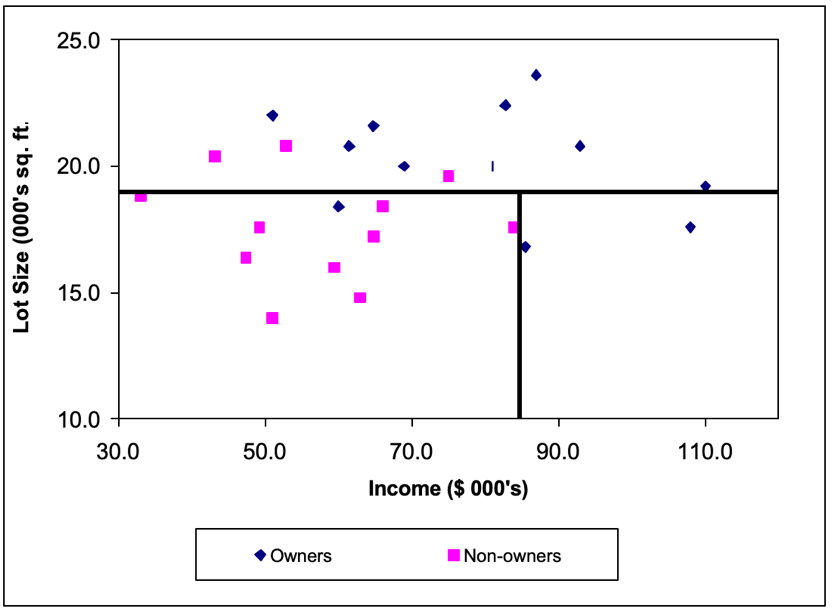
mostly owners (9 owners and 3 non-owners) while the lower rectangle contains mostly nonowners (9 non-owners and 3 owners).

How did CART decide on this particular split? As you remember in the previous sections, it examined each variable and all possible split values for each variable to find the best split.

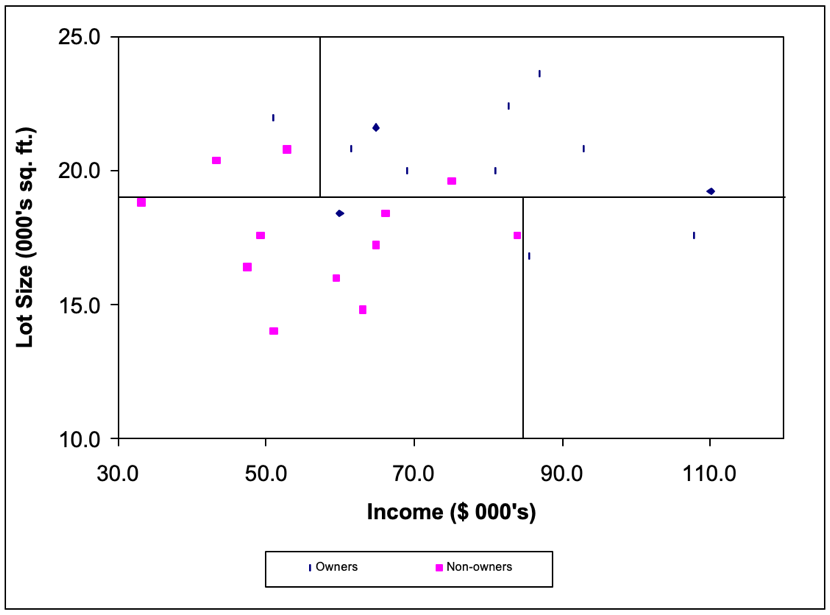
The next split is on the Income variable, x1 at the value 84.75. Figure below shows that once again the CART procedure has wisely chosen to split a rectangle to increase the purity of the resulting rectangles. The left lower rectangle which contains data points with x1<= 84.75 and x2 <= 19 has all

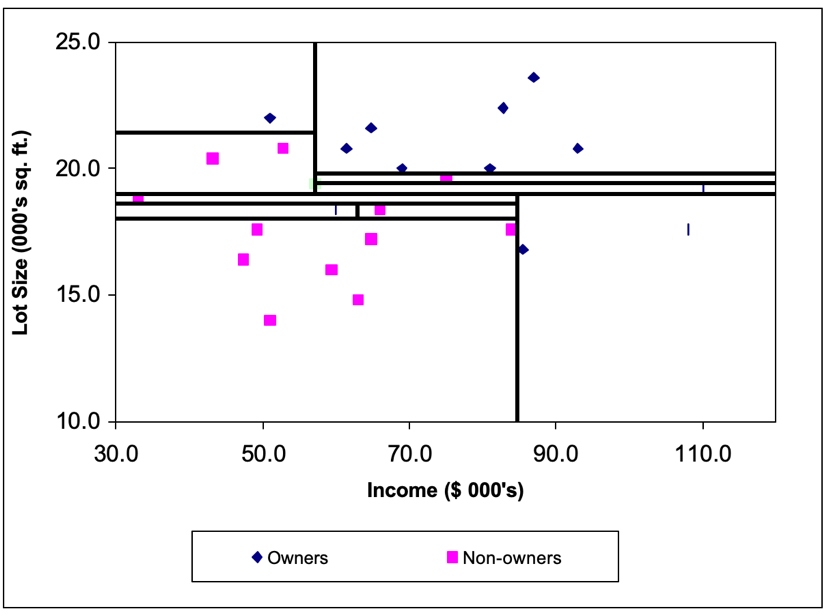
but one points that are non-owners; while the right lower rectangle which contains data points

with x1 > 84.75 and x2 <= 19 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.



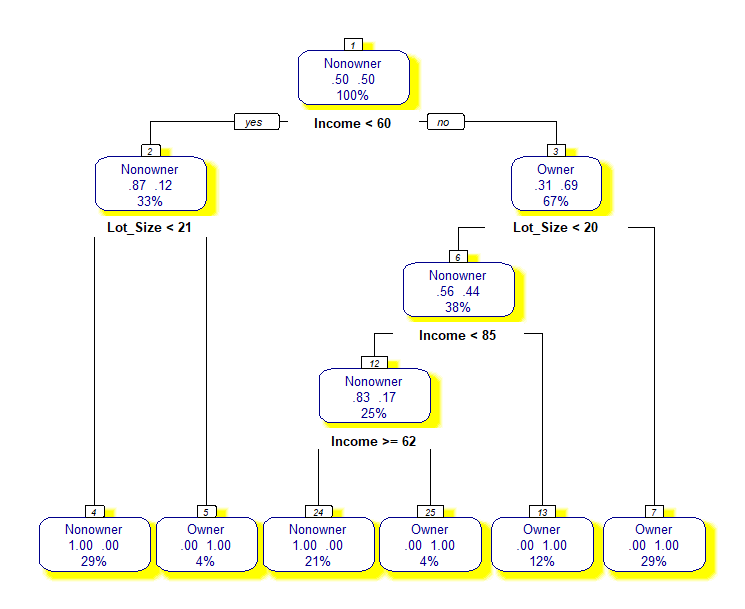
We can see how the recursive partitioning is refining the set of constituent rectangles to become purer as the algorithm proceeds. The final stage of the recursive partitioning is shown in Figure below.

The 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.

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.

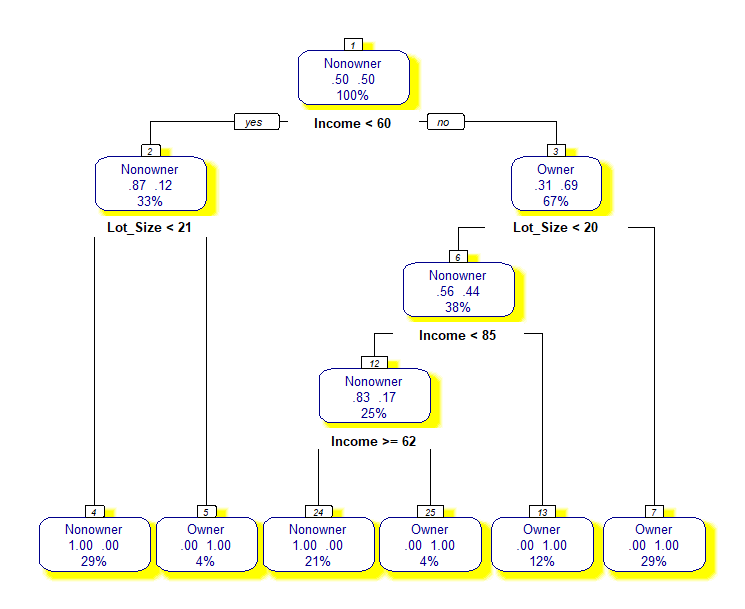


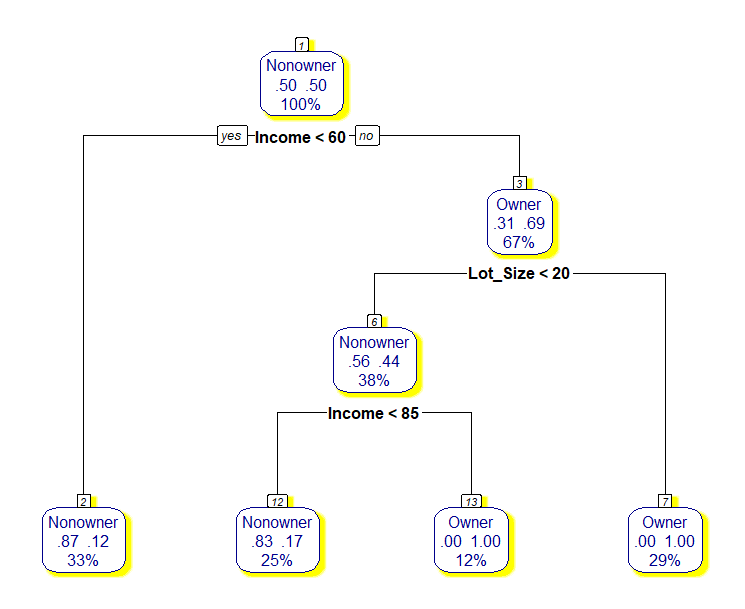
# Classification Tree pruning

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.

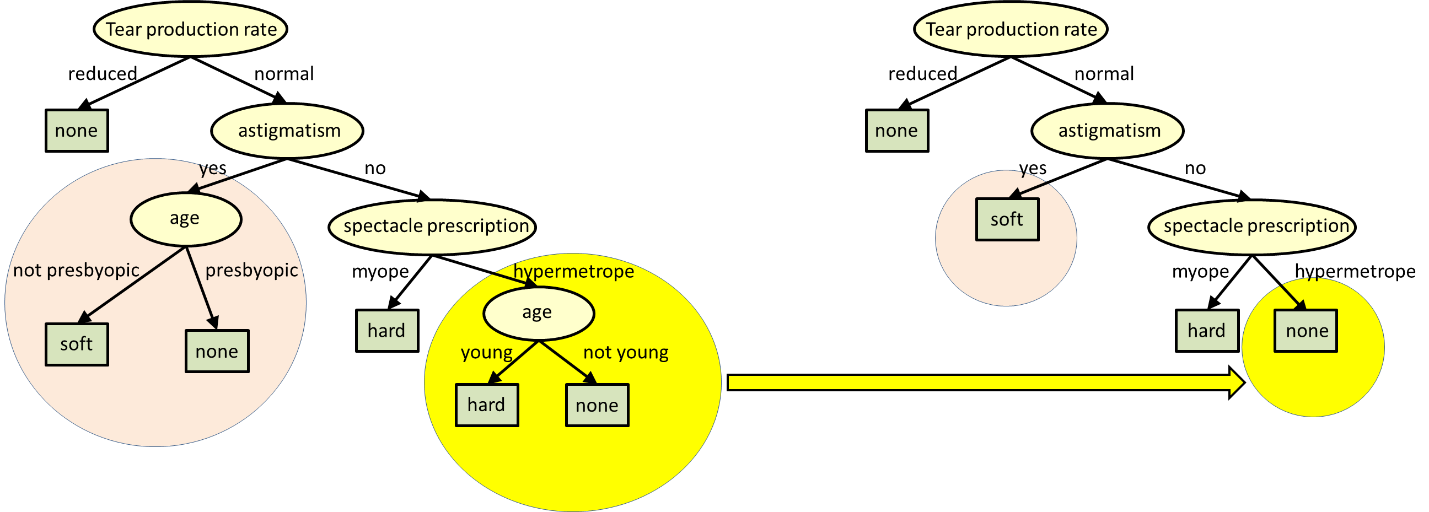




#### Pruning by Information Gain (IG)

The simplest technique is to prune out portions of the tree that result in the least information gain. This procedure does not require any additional data, and only bases the pruning on the information that is already computed when the tree is being built from training data.

The process of IG-based pruning requires us to identify “twigs”, nodes whose children are all leaves. “Pruning” a twig removes all of the leaves which are the children of the twig, and makes the twig a leaf. The figure below illustrates this.



The algorithm for pruning is as follows:

1. Catalog all twigs in the tree
2. Count the total number of leaves in the tree.
3. While the number of leaves in the tree exceeds the desired number:
   1. Find the twig with the least Information Gain
   2. Remove all child nodes of the twig.
   3. Relabel twig as a leaf.
   4. Update the leaf count.

#### Pruning by Classification Performance on Validation Set

An alternate approach is to prune the tree to maximize classification performance on a validation set (as you remember validation dataset is a data set with known labels, which was **not** used to train the tree).

We pass the validation data down the tree. At each node, we record the total number of instances and the number of misclassifications, if that node were actually a leaf. We do this at all nodes and leaves.

Subsequently, we prune all twigs where pruning results in the smallest overall increase in classification error.

The overall algorithm for pruning is as follows:

1. For each instance of validation data:

Recursively pass:

* Catalog all twigs in the tree
* Count the total number of leaves in the tree.
* While the number of leaves in the tree exceeds the desired number:

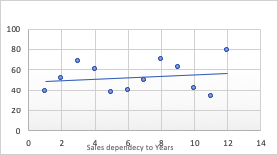
1. Find the twig with the least Information Gain
2. Remove all child nodes of the twig.
3. Relabel twig as a leaf.
4. Update the leaf count.

#### A simple overfitting example:

consider a database of retail purchases that includes the item bought, the purchaser, and the date and time of purchase. It's easy to construct a model that will fit the training set perfectly by using the date and time of purchase to predict the other attributes; but this model will not generalize at all to new data, because those past times will never occur again.

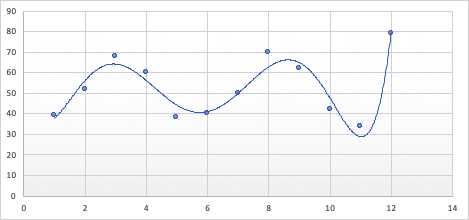
Generally, a learning algorithm is said to overfit relative to a simpler one if it is more accurate in fitting known data (hindsight: Recognition of Realities, Possibilities, or Requirements) but less accurate in predicting new data (foresight). One can intuitively understand overfitting from the fact that information from all past experience can be divided into two groups: ***information that is relevant for the future and irrelevant information ("noise")***. Everything else being equal, the more difficult a criterion is to predict (i.e., the higher its uncertainty), the more noise exists in past information that needs to be ignored. The problem is determining which part to ignore**. A learning algorithm that can reduce the chance of fitting noise is called "robust."**

|  |  |
| --- | --- |
| X (Year) | Y (Sales in Millions) |
| 1 | 39 |
| 2 | 52 |
| 3 | 68 |
| 4 | 60 |
| 5 | 38 |
| 6 | 40 |
| 7 | 50 |
| 8 | 70 |
| 9 | 62 |
| 10 | 42 |
| 11 | 34 |
| 12 | 79 |



Example: We have plotted the following x and y values.

A linear regression solution will provide the y = 0.7413x + 48.015. and the above chart.



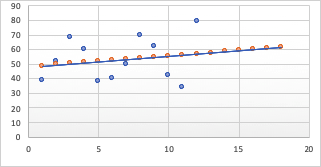
If we try to fit the regression curve to data we will have the following polynomial function

y = 0.0104x6 - 0.3617x5 + 4.7127x4 - 28.186x3 + 75.538x2 - 71.121x + 57.318

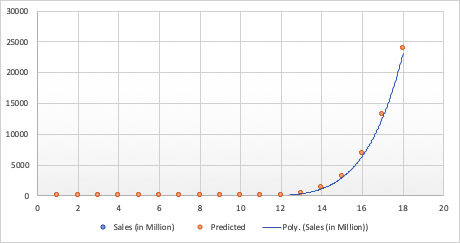
and the following chart which seem fitting better the data.

Now we apply the validation data (in Yellow) and observe the outcomes of both models

|  |  |  |
| --- | --- | --- |
| Year | Sales (I Million) | Predicted |
| 1 | 39 | 48.7563 |
| 2 | 52 | 49.4976 |
| 3 | 68 | 50.2389 |
| 4 | 60 | 50.9802 |
| 5 | 38 | 51.7215 |
| 6 | 40 | 52.4628 |
| 7 | 50 | 53.2041 |
| 8 | 70 | 53.9454 |
| 9 | 62 | 54.6867 |
| 10 | 42 | 55.428 |
| 11 | 34 | 56.1693 |
| 12 | 79 | 56.9106 |
| 13 | 47 | 57.6519 |
| 14 | 51 | 58.3932 |
| 15 | 55 | 59.1345 |
| 16 | 49 | 59.8758 |
| 17 | 53 | 60.6171 |
| 18 | 59 | 61.3584 |



|  |  |  |
| --- | --- | --- |
| Year | Sales (in Million) | Predicted |
| 1 | 39 | 37.9104 |
| 2 | 52 | 56.2344 |
| 3 | 68 | 64.1922 |
| 4 | 60 | 56.2068 |
| 5 | 38 | 44.538 |
| 6 | 40 | 42.0864 |
| 7 | 50 | 52.6854 |
| 8 | 70 | 68.8812 |
| 9 | 62 | 77.2008 |
| 10 | 42 | 70.908 |
| 11 | 34 | 70.2474 |
| 12 | 79 | 150.1764 |
| 13 | 47 | 475.5852 |
| 14 | 51 | 1344.0048 |
| 15 | 55 | 3235.803 |
| 16 | 49 | 6871.8684 |
| 17 | 53 | 13278.7824 |
| 18 | 59 | 23861.4792 |



# rpart.control

Various parameters that control aspects of the rpart fit

Example:

rpart.control(minsplit = 20, minbucket = round(minsplit/3), cp = 0.01,

maxcompete = 4, maxsurrogate = 5, usesurrogate = 2, xval = 10,

surrogatestyle = 0, maxdepth = 30, …)

**minsplit**: the minimum number of observations that must exist in a node in order for a split to be attempted.

**minbucket:** the minimum number of observations in any terminal node.

Note: If only one of minbucket or minsplit is specified, the code either sets minsplit to minbucket\*3 or minbucket to minsplit/3, as appropriate.

The cp complexity parameter. Any split that does not decrease the overall lack of fit by a factor of cp is not attempted. For instance, with Anova (statistics) splitting, this means that the overall R-squared must increase by cp at each step. The main role of this parameter is to save computing time by pruning off splits that are obviously not worthwhile. Essentially, the user informs the program that any split which does not improve the fit by cp will likely be pruned off by cross-validation, and that hence the program need not pursue it.

# 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(2015)

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 to plot the tree

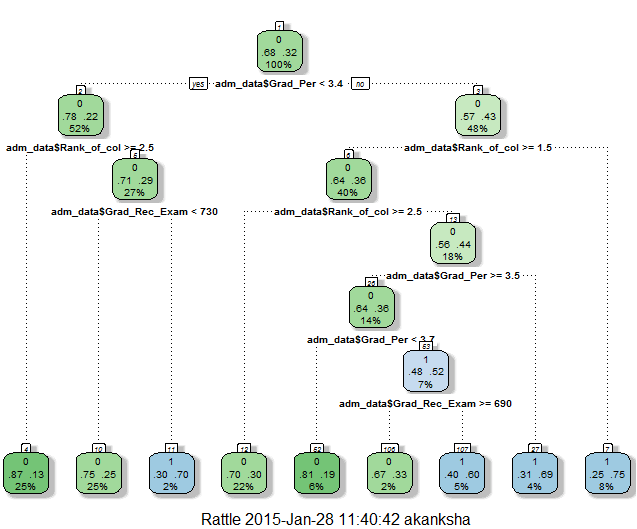
To enhance it, let us take some help from rattle :

library(rattle)

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 we will use just the plotting capabilities of Rattle to achieve a decent decision tree plot.

fancyRpartPlot(tree)



**Prune the tree to create an optimal decision tree (All in one!)**

#intsall package “rattle”

library(rattle)

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

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

