1. **Linear Models**
- Simple Linear Regression
- Multiple Regression
- Logistic Regression
- Poisson Regression

2. **Nonlinear Models**
- Nonlinear least squares
- Generalized additive models
- Decision trees
- Random forests

3. **Time Series:**
- Autoregressive Moving Average
- VAR
- GARCH

4. **Clustering**
- K –Means
- PAM
- Hierarchical Clustering

Regression Analysis

- Regression analysis is very widely used statistical tool to establish a relationship model between two variables.
- One of these variable is called predictor variable whose value is gathered through experiments.
- The other variable is called response variable whose value is derived from the predictor variable.

1. Linear Regression

- In linear regression these two variables are related through an equation, where exponent (power) of both these variables is 1.
- Mathematically a linear relationship represents a straight line when plotted as a graph.
- A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.
- The general mathematical equation for a linear regression is −

$$y = ax + b$$

Following is the description of the parameters used :

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are constants which are called the coefficients.

Steps to Establish a Regression

- A simple example of regression is predicting weight of a person when his height is known. To do this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

- Carry out the experiment of gathering a sample of observed values of height and corresponding weight.
- Create a relationship model using the **lm()** functions in R.
- Find the coefficients from the model created and create the mathematical equation using these
- Get a summary of the relationship model to know the average error in prediction. Also called **residuals**.
- To predict the weight of new persons, use the **predict()**function in R.

Input Data

Below is the sample data representing the observations −

- # Values of height 151, 174, 138, 186, 128, 136, 179, 163, 152, 131
- # Values of weight. 63, 81, 56, 91, 47, 57, 76, 72, 62, 48

lm() Function

- This function creates the relationship model between the predictor and the response variable.

Syntax

- The basic syntax for **lm()** function in linear regression is −
- lm(formula,data) Following is the description of the parameters used −
- **formula** is a symbol presenting the relation between x and y.
- **data** is the vector on which the formula will be applied.

Create relationship model and get the coefficients

```
>x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
>y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
>relation <- lm(y~x)
>print(relation)
```

```
Call:
lm(formula = y ~ x)
Coefficients:
(Intercept)        x
-38.4551          0.6746
```

Get the summary of the relationship

```
>print(summary(relation))
Call:
lm(formula = y ~ x)
Residuals:
    Min      1Q  Median      3Q
-6.3002 -1.6629  0.0412  1.8944
    Max
 3.9775
Coefficients:
             Estimate Std. Error
(Intercept) -38.45509    8.04901
x             0.67461    0.05191
            t value Pr(>|t|)
(Intercept)  -4.778  0.00139 **
x            12.997 1.16e-06 ***
---
Signif. codes:
  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05
  ‘.’ 0.1 ‘ ’ 1
Residual standard error: 3.253 on 8 degrees of freedom
Multiple R-squared:  0.9548,  Adjusted R-squared:  0.9491
F-statistic: 168.9 on 1 and 8 DF,  p-value: 1.164e-06
```

predict()

Syntax

predict(object, newdata)

Following is the description of the parameters used:

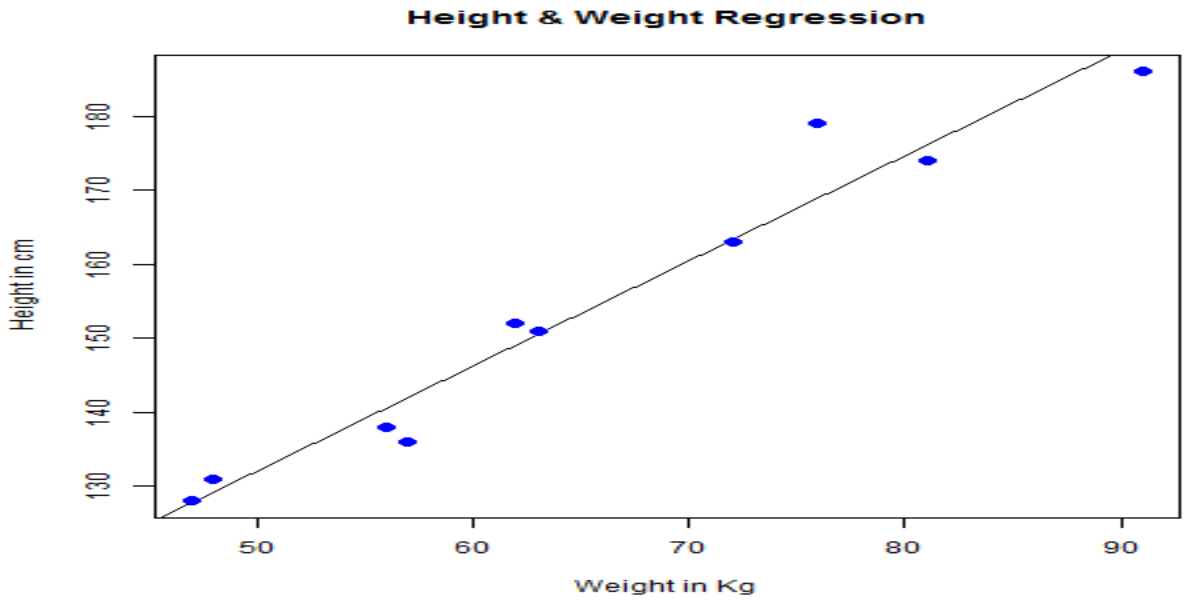
- object is the formula which is already created using the lm() function.
- newdata is the vector containing the new value for predictor variable.

Predict the weight of new persons

```
# The predictor vector.
>x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
# The resposne vector.
>y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
# Apply the lm() function.
>relation <- lm(y~x)
# Find weight of a person with height 170.
>a <- data.frame(x = 170)
>result <-  predict(relation,a)
>print(result)
 1
76.22869
```

Visualize the Regression Graphically

```
# Create the predictor and response variable.
>x<- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
>y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
>relation <- lm(y~x)
# Give the chart file a name.
>png(file = "linearregression.png")
# Plot the chart.
>plot(y,x,col = "blue",main = "Height & Weight Regression",
abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")
# Save the file.
>dev.off()
```



2. Multiple Regression

- Multiple regression is an extension of linear regression into relationship between more than two variables. In simple linear relation we have one predictor and one response variable, but in multiple regression we have more than one predictor variable and one response variable.

The general mathematical equation for multiple regression is

$$y = a + b1x1 + b2x2 + ...bnxn$$

Following is the description of the parameters used

- y is the response variable.
- a, b1, b2...bn are the coefficients.
- x1, x2, ...xn are the predictor variables.

- We create the regression model using the lm() function in R. The model determines the value of the coefficients using the input data. Next we can predict the value of the response variable for a given set of predictor variables using these coefficients.

lm()

- This function creates the relationship model between the predictor and the response variable.

Syntax

lm(y ~ x1+x2+x3...,data)

Following is the description of the parameters used

- **formula** is a symbol presenting the relation between the response variable and predictor variables.
- **data** is the vector on which the formula will be applied.

Example

Input Data

- Consider the data set "mtcars" available in the R environment. It gives a comparison between different car models in terms of mileage per gallon (mpg), cylinder displacement("disp"), horse power("hp"), weight of the car("wt") and some more parameters.
- The goal of the model is to establish the relationship between "mpg" as a response variable with "disp","hp" and "wt" as predictor variables. We create a subset of these variables from the mtcars data set for this purpose.

•

```
>input <- mtcars[,c("mpg","disp","hp","wt")]
>print(head(input))
```

	mpg	disp	hp	wt
Mazda RX4	21.0	160	110	2.620
Mazda RX4 Wag	21.0	160	110	2.875
Datsun 710	22.8	108	93	2.320
Hornet 4 Drive	21.4	258	110	3.215
Hornet Sportabout	18.7	360	175	3.440
Valiant	18.1	225	105	3.460

Create Relationship Model & get the Coeffients

```
input <- mtcars[,c("mpg","disp","hp","wt")]
# Create the relationship model.
model <- lm(mpg~disp+hp+wt, data = input)
# Show the model.
print(model)
```

```
# Get the Intercept and coefficients as vector elements.
cat("# # # # The Coefficient Values # # # ","\n")
a <- coef(model)[1]
print(a)
Xdisp <- coef(model)[2]
Xhp <- coef(model)[3]
Xwt <- coef(model)[4]
print(Xdisp)
print(Xhp)
print(Xwt)
```

Result

Call:

lm(formula = mpg ~ disp + hp + wt, data = input)

Coefficients:

(Intercept)	disp	hp	wt
37.105505	-0.000937	-0.031157	-3.800891

 # # # # The Coefficient Values # # #

(Intercept)

37.10551

disp

-0.0009370091

hp

-0.03115655

wt

-3.800891

3. Logistic Regressions

- The Logistic Regression is a regression model in which the response variable (dependent variable) has categorical values such as True/False or 0/1. It actually measures the probability of a binary response as the value of response variable based on the mathematical equation relating it with the predictor variables.

The general mathematical equation for logistic regression is

$$y = 1/(1+e^{-(a+b1x1+b2x2+b3x3+...)})$$

Following is the description of the parameters used

- y is the response variable.
- x is the predictor variable.
- a and b are the coefficients which are numeric constants.

The function used to create the regression model is the glm() function.

Syntax

glm(formula,data,family)

Following is the description of the parameters used

- formula is the symbol presenting the relationship between the variables.
- data is the data set giving the values of these variables.
- family is R object to specify the details of the model. It's value is binomial for logistic regression.

Example

- The in-built data set "mtcars" describes different models of a car with their various engine specifications. In "mtcars" data set, the transmission mode (automatic or manual) is described by the column am which is a binary value (0 or 1). We can create a logistic regression model between the columns "am" and 3 other columns - hp, wt and cyl.

Select some columns form mtcars.

```
>input <- mtcars[,c("am","cyl","hp","wt")]
>print(head(input))
```

Result

	am	cyl	hp	wt
Mazda RX4	1	6	110	2.620
Mazda RX4 Wag	1	6	110	2.875
Datsun 710	1	4	93	2.320
Hornet 4 Drive	0	6	110	3.215
Hornet Sportabout	0	8	175	3.440
Valiant	0	6	105	3.460

Create Regression Model

- We use the **glm()** function to create the regression model and get its summary for analysis.

```
>input <- mtcars[,c("am","cyl","hp","wt")]
>am.data = glm(formula = am ~ cyl + hp + wt, data = input, family = binomial)
>print(summary(am.data))
```

Result

```
Call:
glm(formula = am ~ cyl + hp + wt, family = binomial, data = input)
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.17272  -0.14907  -0.01464   0.14116   1.27641
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 19.70288    8.11637   2.428   0.0152 *
cyl          0.48760    1.07162   0.455   0.6491
hp           0.03259    0.01886   1.728   0.0840 .
wt          -9.14947    4.15332  -2.203   0.0276 *
```

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 43.2297 on 31 degrees of freedom

Residual deviance: 9.8415 on 28 degrees of freedom

AIC: 17.841

Number of Fisher Scoring iterations: 8

Conclusion

- In the summary as the p-value in the last column is more than 0.05 for the variables "cyl" and "hp", we consider them to be insignificant in contributing to the value of the variable "am". Only weight (wt) impacts the "am" value in this regression model.

4. Poisson Regression

- Poisson Regression involves regression models in which the response variable is in the form of counts and not fractional numbers. For example, the count of number of births or number of wins in a football match series. Also the values of the response variables follow a Poisson distribution.

The general mathematical equation for Poisson regression is −

$$\log(y) = a + b1x1 + b2x2 + bnxn.....$$

Following is the description of the parameters used −

- y is the response variable.
- a and b are the numeric coefficients.
- x is the predictor variable.

The function used to create the Poisson regression model is the glm() function.

Syntax:

```
glm(formula,data,family)
```

- **formula** is the symbol presenting the relationship between the variables.
- **data** is the data set giving the values of these variables.
- **family** is R object to specify the details of the model. It's value is 'Poisson' for Logistic Regression.

Example

- We have the in-built data set "warpbreaks" which describes the effect of wool type (A or B) and tension (low, medium or high) on the number of warp breaks per loom. Let's consider "breaks" as the response variable which is a count of number of breaks. The wool "type" and "tension" are taken as predictor variables.

Input Data

```
>input<-warpbreaks
>print(head(input))
```

Result:

	breaks	wool	tension
1	26	A	L
2	30	A	L
3	54	A	L
4	25	A	L
5	70	A	L
6	52	A	L

Create Regression Model

```
>output <-glm(formula = breaks ~ wool+tension, data = warpbreaks, family =
    poisson)
>print(summary(output))
```

Result:

```
Call:
glm(formula = breaks ~ wool + tension, family = poisson, data = warpbreaks)

Deviance Residuals:
    Min      1Q  Median      3Q     Max
-3.6871 -1.6503 -0.4269  1.1902  4.2616

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.69196    0.04541  81.302  < 2e-16 ***
woolB       -0.20599    0.05157  -3.994 6.49e-05 ***
tensionM    -0.32132    0.06027  -5.332 9.73e-08 ***
tensionH    -0.51849    0.06396  -8.107 5.21e-16 ***
```

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 297.37 on 53 degrees of freedom
Residual deviance: 210.39 on 50 degrees of freedom
AIC: 493.06

Number of Fisher Scoring iterations: 4

Non-linear Models

- When modeling real world data for regression analysis, we observe that it is rarely the case that the equation of the model is a linear equation giving a linear graph.
- Most of the time, the equation of the model of real world data involves mathematical functions of higher degree like an exponent of 3 or a sin function.
- In such a scenario, the plot of the model gives a curve rather than a line. The goal of both linear and non-linear regression is to adjust the values of the model's parameters to find the line or curve that comes closest to your data.
- On finding these values we will be able to estimate the response variable with good accuracy.

1. Non- Linear Least Squares

- In Least Square regression, we establish a regression model in which the sum of the squares of the vertical distances of different points from the regression curve is minimized.
- We generally start with a defined model and assume some values for the coefficients. We then apply the nls() function of R to get the more accurate values along with the confidence intervals.

Syntax

nls(formula, data, start)

Where

- formula is a nonlinear model formula including variables and parameters.
- data is a data frame used to evaluate the variables in the formula.
- start is a named list or named numeric vector of starting estimates.

Example

- We will consider a nonlinear model with assumption of initial values of its coefficients. Next we will see what is the confidence intervals of these assumed values so that we can judge how well these values fit into the model.
- Equation is a = b1*x^2+b2
- Lets assume the initial coefficients to be 1 and 3 and fit these values into nls().

Code:

```
>xvalues <- c(1.6,2.1,2,2.23,3.71,3.25,3.4,3.86,1.19,2.21)
>xvalues
[1] 1.60 2.10 2.00 2.23 3.71 3.25 3.40 3.86 1.19 2.21
```

```
>yvalues <- c(5.19,7.43,6.94,8.11,18.75,14.88,16.06,19.12,3.21,7.58)
>yvalues
[1]  5.19  7.43  6.94  8.11 18.75 14.88 16.06 19.12
[9]  3.21  7.58
>png(file = "nls.png")
>plot(xvalues,yvalues)
# Take the assumed values and fit into the model.
>model <- nls(yvalues ~ b1*xvalues^2+b2,start = list(b1 = 1,b2 = 3))
>model
Nonlinear regression model
  model: yvalues ~ b1 * xvalues^2 + b2
   data: parent.frame()
   b1    b2
1.195 1.997
 residual sum-of-squares: 1.082
Number of iterations to convergence: 1
Achieved convergence tolerance: 1.082e-08
```

Plot the chart with new data by fitting it to a prediction from 100 data points.

```
>new.data <- data.frame(xvalues = seq(min(xvalues),max(xvalues),len = 100))
lines(new.data$xvalues,predict(model,newdata = new.data))
>new.data
 xvalues
1   1.190000
2   1.216970
3   1.243939
4   1.270909
5   1.297879
6   1.324848
7   1.351818
8   1.378788
9   1.405758
10   1.432727
11   And so on 100
```

Save the file.

>dev.off()

null device

1

Get the sum of the squared residuals.

>print(sum(resid(model)^2))

[1] 1.081935

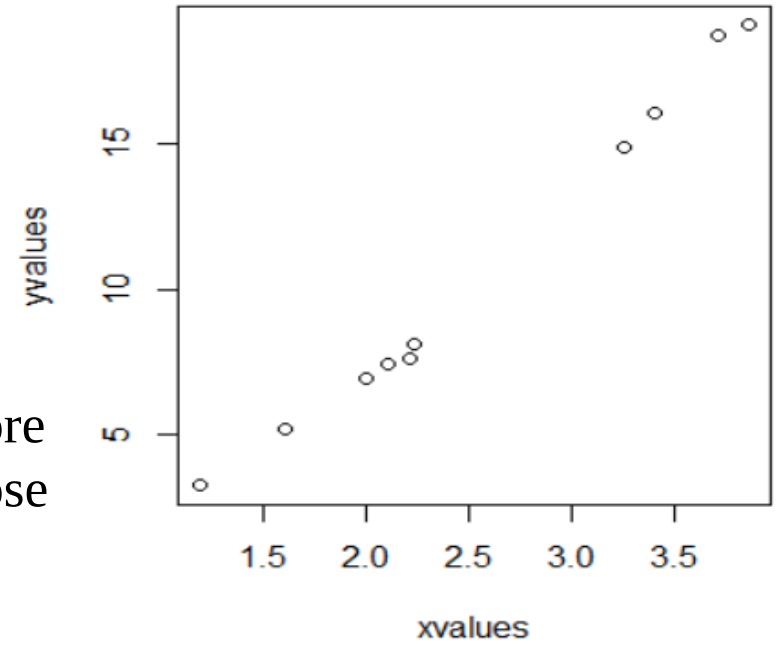
Get the confidence intervals on the chosen values of the coefficients.

>print(confint(model))

Waiting for profiling to be done...

	2.5%	97.5%
b1	1.137708	1.253135
b2	1.497364	2.496484

We can conclude that the value of b1 is more close to 1 while the value of b2 is more close to 2 and not 3.



Spline regression

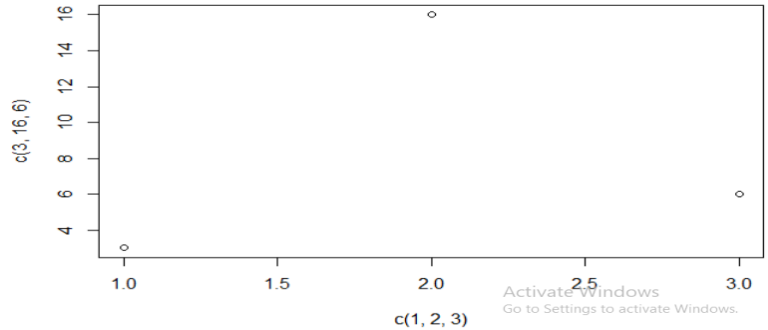
- Polynomial regression only captures a certain amount of curvature in a nonlinear relationship. An alternative, and often superior, approach to modeling nonlinear relationships is to use splines (P. Bruce and Bruce 2017).
- Splines provide a way to smoothly interpolate between fixed points, called knots. Polynomial regression is computed between knots. In other words, splines are series of polynomial segments strung together, joining at knots (P. Bruce and Bruce 2017).
- The R package splines includes the function bs for creating a b-spline term in a regression model.
- You need to specify two parameters: the degree of the polynomial and the location of the knots. In our example, we'll place the knots at the lower quartile, the median quartile, and the upper quartile:

We'll create a model using a cubic spline (degree = 3):

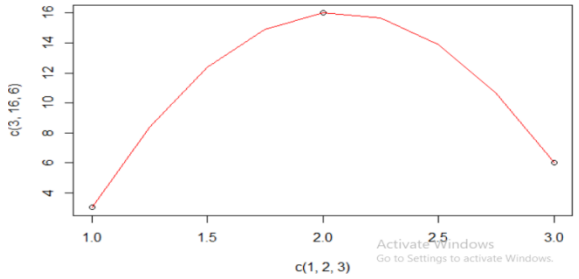
Code:

```
>spline(x=c(1,2,3),y=c(3,16,6),n=9)
$x
[1] 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00
```

$y
[1] 3.00000 8.40625 12.37500 14.90625 16.00000
[6] 15.65625 13.87500 10.65625 6.00000
>splineout<-spline(x=c(1,2,3),y=c(3,16,6),n=9)
>splineout$x
[1] 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00
>splineout$y
[1] 3.00000 8.40625 12.37500 14.90625 16.00000
[6] 15.65625 13.87500 10.65625 6.00000
>plot(c(1,2,3),c(3,16,6))



>points(splineout$x,splineout$y,type='l',col="red")

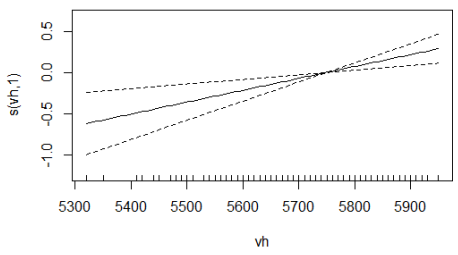


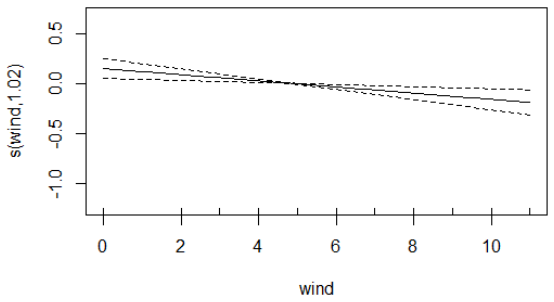
2. Generalized additive models

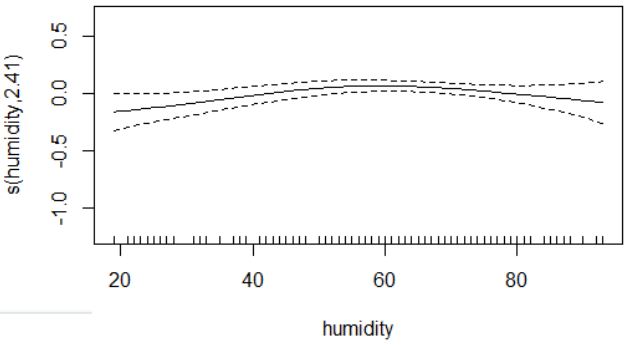
- Once you have detected a non-linear relationship in your data, the polynomial terms may not be flexible enough to capture the relationship, and spline terms require specifying the knots.
- Generalized additive models, or GAM, are a technique to automatically fit a spline regression. This can be done using the mgcv R package:

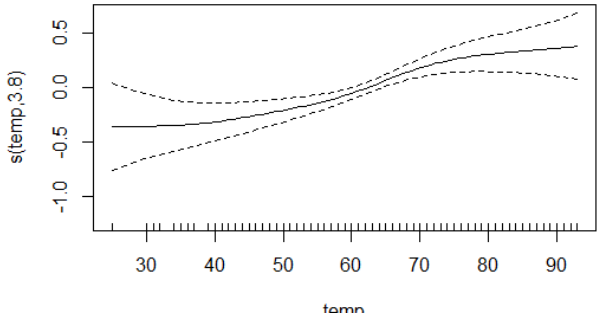
Code:

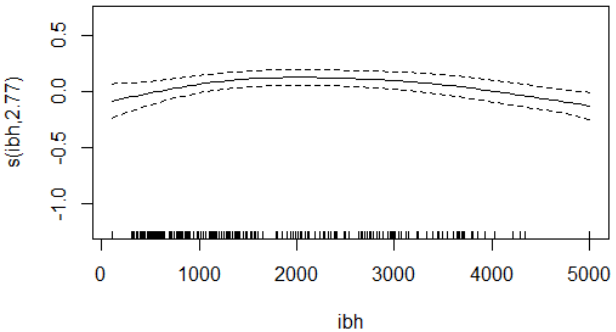
```
install.packages("faraway")
library(faraway)
data("ozone")
install.packages("mgcv")
library(mgcv)
gamobj<-gam(log(O3)~s(vh)+s(wind)+s(humidity)+s(temp)+s(ibh)+
            s(dpg)+s(ibt)+s(vis)+s(doy),
          family=gaussian(link=identity),data=ozone)
summary(gamobj)
pdf("GAMozone.pdf")
par(mfrow=c(3,3))
plot(gamobj)
dev.off()
plot(gamobj)
```

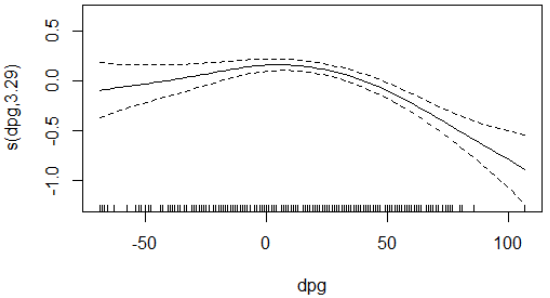

s(vh,1)
0.5
0.0
-0.5
-1.0
5300
5400
5500
5600
5700
5800
5900
vh

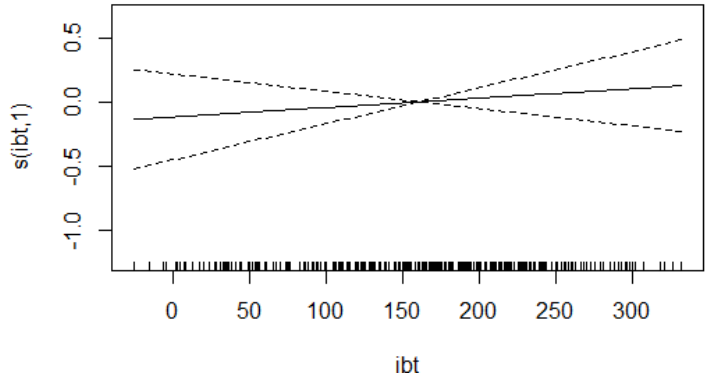

s(wind,1.02)
0.5
0.0
-0.5
-1.0
0
2
4
6
8
10
wind

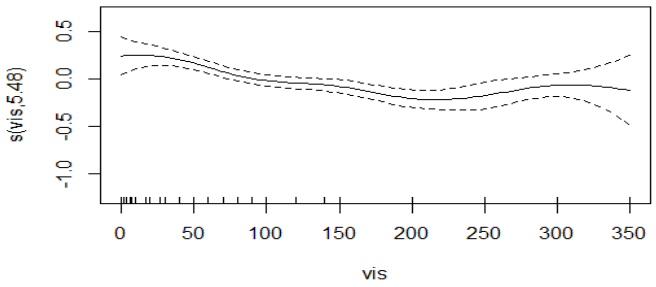

s(humidity,2.41)
0.5
0.0
-0.5
-1.0
20
40
60
80
humidity


s(temp,3.8)
0.5
0.0
-0.5
-1.0
30
40
50
60
70
80
90
temp


s(ibh,2.77)
0.5
0.0
-0.5
-1.0
0
1000
2000
3000
4000
5000
ibh


s(dpg,3.29)
0.5
0.0
-0.5
-1.0
-50
0
50
100
dpg


s(ibt,1)
0.5
0.0
-0.5
-1.0
0
50
100
150
200
250
300
ibt


s(vis,5.48)
0.5
0.0
-0.5
-1.0
0
50
100
150
200
250
300
350
vis

3. Décision Tree

- Decision tree is a graph to represent choices and their results in form of a tree. The nodes in the graph represent an event or choice and the edges of the graph represent the decision rules or conditions.
- It is mostly used in Machine Learning and Data Mining applications using R.
- Examples of use of decision tress is − predicting an email as spam or not spam, predicting of a tumor is cancerous or predicting a loan as a good or bad credit risk based on the factors in each of these.
- Generally, a model is created with observed data also called training data. Then a set of validation data is used to verify and improve the model. R has packages which are used to create and visualize decision trees.
- For new set of predictor variable, we use this model to arrive at a decision on the category (yes/No, spam/not spam) of the data.
- The R package "party" is used to create decision trees.

Code:

```
>install.packages("party")
>library(party)
```

- The package "party" has the function ctree() which is used to create and analyze decision tree.

Syntax:

ctree(formula, data)

Where

- formula is a formula describing the predictor and response variables.
- data is the name of the data set used.

Input Data

- We will use the R in-built data set named readingSkills to create a decision tree. It describes the score of someone's readingSkills if we know the variables "age","shoesize","score" and whether the person is a native speaker or not.

```
>print(head(readingSkills))
  nativeSpeaker age shoeSize    score
1           yes   5  24.83189 32.29385
2           yes   6  25.95238 36.63105
3            no  11  30.42170 49.60593
4           yes   7  28.66450 40.28456
5           yes  11  31.88207 55.46085
6           yes  10  30.07843 52.83124
```

```
# Create the input data frame.
>input.dat <- readingSkills[c(1:105),]
>input.dat
 nativeSpeaker age shoeSize    score
1             yes   5 24.83189 32.29385
2             yes   6 25.95238 36.63105
3              no  11 30.42170 49.60593
4             yes   7 28.66450 40.28456
5             yes  11 31.88207 55.46085
6             yes  10 30.07843 52.83124
7              no   7 27.25963 34.40229
8             yes  11 30.72398 55.52747
9             yes   5 25.64411 32.49935
10    no   7 26.69835 33.93269
11    And so on till 105
# Give the chart file a name.
>png(file = "decision_tree.png")
```

```
# Create the tree.
>output.tree <- ctree(nativeSpeaker ~ age + shoeSize + score, data = input.dat)
>output.tree
        Conditional inference tree with 4 terminal nodes
Response:  nativeSpeaker
Inputs:  age, shoeSize, score
Number of observations:  105
1) score <= 38.30587; criterion = 1, statistic = 24.932
  2) age <= 6; criterion = 0.993, statistic = 9.361
    3) score <= 30.76591; criterion = 0.999, statistic = 14.093
      4)*  weights = 13
    3) score > 30.76591
      5)*  weights = 9
  2) age > 6
    6)*  weights = 21
1) score > 38.30587
  7)*  weights = 62
```

Plot the tree.

>plot(output.tree)

Save the file.

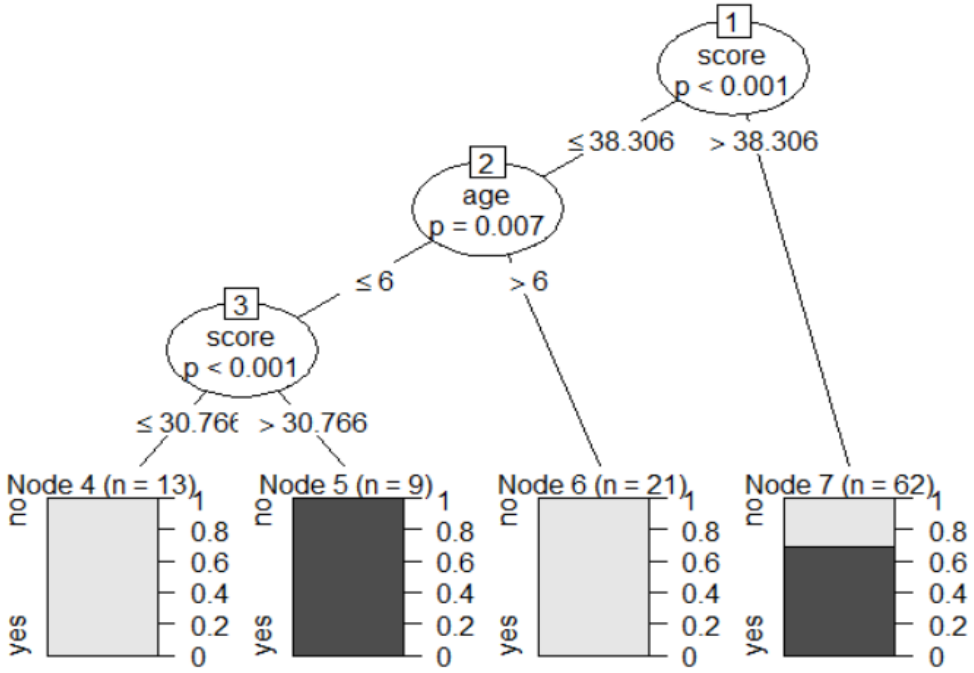
>dev.off()

RStudioGD

2

Conclusion

From the decision tree shown above we can conclude that anyone whose readingSkills score is less than 38.3 and age is more than 6 is not a native Speaker.



4. Random Forest

- In the random forest approach, a large number of decision trees are created. Every observation is fed into every decision tree. The most common outcome for each observation is used as the final output. A new observation is fed into all the trees and taking a majority vote for each classification model.
- An error estimate is made for the cases which were not used while building the tree. That is called an OOB (Out-of-bag) error estimate which is mentioned as a percentage.
- The R package "randomForest" is used to create random forests.

Syntax:

randomForest(formula,data)

- Formula : describing the predictor and response variables.
- Data is the name of the data set used.

Install R Package

>install.packages(« randomForest)

>library(randomForest)

>print(head(readingSkills))

```
>head(readingSkills)
  nativeSpeaker age shoeSize    score
1           yes   5 24.83189 32.29385
2           yes   6 25.95238 36.63105
3            no  11 30.42170 49.60593
4           yes   7 28.66450 40.28456
5           yes  11 31.88207 55.46085
6           yes  10 30.07843 52.83124
# Create the forest.
>output.forest <- randomForest(nativeSpeaker ~ age + shoeSize + score,
        data = readingSkills)
>output.forest
Call:
 randomForest(formula = nativeSpeaker ~ age + shoeSize + score,       data =
      readingSkills)
              Type of random forest: classification
                    Number of trees: 500
```

No. of variables tried at each split: 1

OOB estimate of error rate: 1%

Confusion matrix:

	no	yes	class.error
no	99	1	0.01
yes	1	99	0.01

Conclusion

- From the random forest shown above we can conclude that the shoesize and score are the important factors deciding if someone is a native speaker or not. Also the model has only 1% error which means we can predict with 99% accuracy.

3. Time Series and Autocorrelation

- A big part of statistics, particularly for financial and econometric data, is analyzing time series, data that are autocorrelated over time.
- That is, one observation depends on previous observations and the order matters.

1. Autoregressive Moving Average

- One of the most common ways of fitting time series models is to use autoregressive (AR), moving average(MA) or both (ARMA).
- AR model can be thought of as linear regressions of the current value of the time series against previous values.
- MA models are, similarly, linear regressions of the current value of the time series against current and previous residuals.

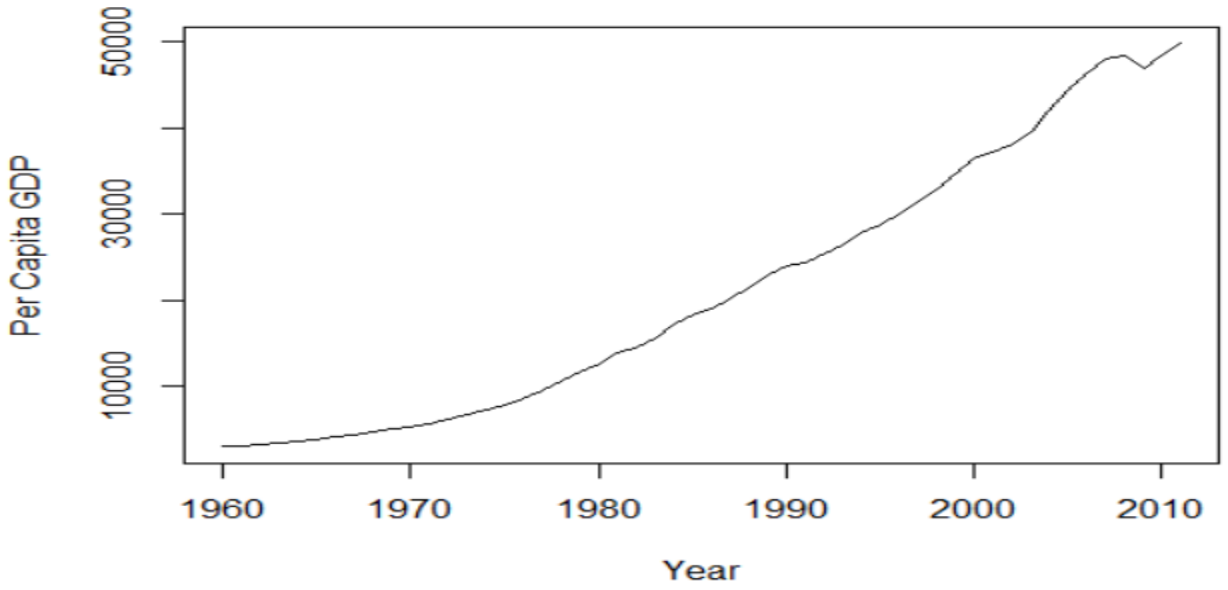
EG: We will make use of the world bank API to download gross domestic product (GDP) for a number of countries from 1960 through 2011.

```
>#load the World Bank API Package
>install.packages("WDI")
>require(WDI)
>gdp<-WDI(country=c("US","CA","GB","DE","CN","JP","SG","IL"),indicator =
    c("NY.GDP.PCAP.CD","NY.GDP.MKTP.CD"),start=1960,end=2011)
```

```
>names(gdp)<-c("iso2c","Country","Year","PerCapGDP","GDP")
>head(gdp)
   iso2c Country Year PerCapGDP        GDP
1    CA  Canada 1960  2294.569          41093453545
2    CA  Canada 1961  2231.294          40767969454
3    CA  Canada 1962  2255.230          41978852041
4    CA  Canada 1963  2354.839          44657169109
5    CA  Canada 1964  2529.518          48882938810
6     CA  Canada 1965  2739.586          53909570342
>install.packages(“ggplot2”)
>library(ggplot2)
>install.packages(“scales”)
>library(scales)
#per capita GDP
>ggplot(gdp,aes(Year,PerCapGDP,color=Country,linetype=Country))+geom_line()+scale
      _y_continuous(label=dollar)
>install.packages(“useful”)
>require(useful)
```

```
# absolute GDP
>ggplot(gdp,aes(Year,GDP,color=Country,linetype=Country))+geom_line()+sca
      le_y_continuos(label=multiple_format(extra=dollar,multiple="M"))


# get US data
>us<-gdp$PerCapGDP[gdp$Country=="United States"]
#convert it to a time series
>us<-ts(us,start=min(gdp$Year), end=max(gdp$Year))
>us
Time Series:
Start = 1960
End = 2011
Frequency = 1
 [1]  3007.123  3066.563  3243.843  3374.515  3573.941
 [6]  3827.527  4146.317  4336.427  4695.923  5032.145
[11]  5246.884  5623.444  6109.926  6741.332  7242.441
[16]  7820.065  8611.402  9471.306 10587.286 11695.554
[21] 12597.668 13993.167 14438.976 15561.426 17134.286
[26] 18269.422 19115.053 20100.859 21483.233 22922.437
[31] 23954.479 24405.165 25492.952 26464.853 27776.636
[36] 28782.175 30068.231 31572.690 32949.198 34620.929
[41] 36449.855 37273.618 38166.038 39677.198 41921.810
[46] 44307.921 46437.067 48061.538 48401.427 47001.555
[51] 48375.407 49793.714
```

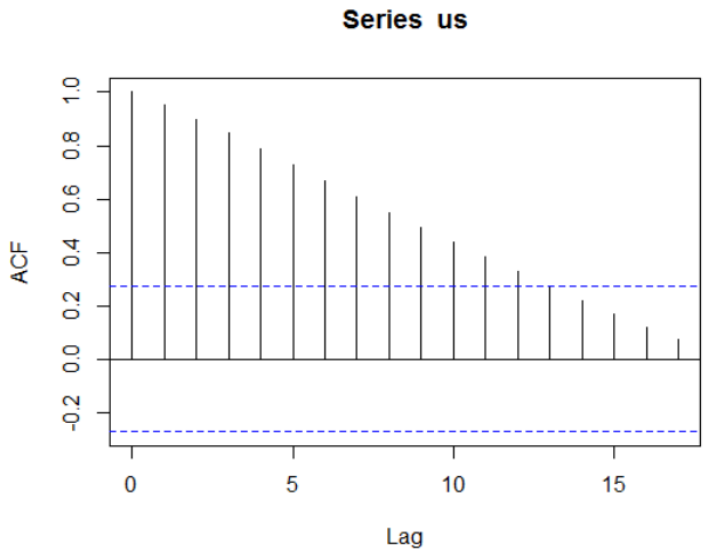
```
>plot(us,ylab="Per Capita GDP",xlab="Year")
```



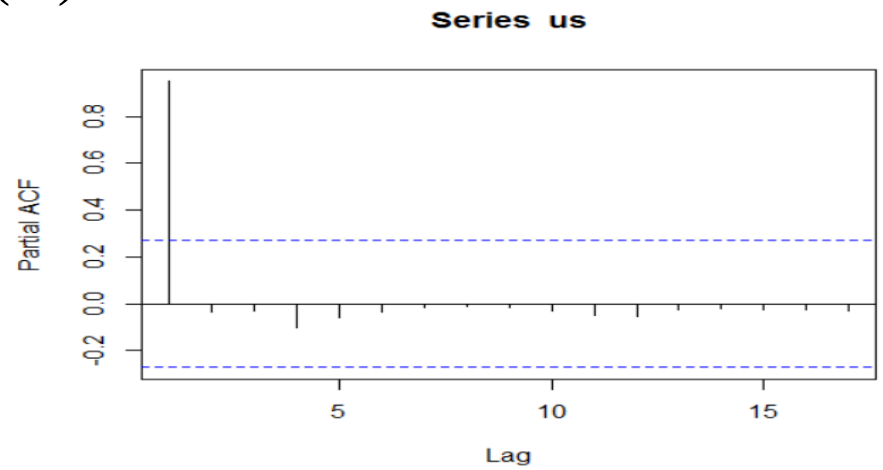
- Another way to assess a time series is to view its autocovariance function (ACF) and partial autocovariance function (PACF).
- In R this is done with the appropriately named acf and pacf functions.
- the ACF shows the correlation of a time series with lags of itself. That is, how much the time series is correlated with itself at one lag, at two lags, at three lags and so on.
- The PACF is a little more complicated. The autocorrelation at lag one can have lingering effects on the autocorrelation at lag two and onward.
- The partial autocorrelation is the amount of correlation between a time series and lags of itself that is not explained by the previous lag.

- So, the partial autocorrelation at lag two is the correlation between the time series and its second lag that is not explained by the first lag.
- The ACF and PACF for the U.S Per Capita GDP data (vertical lines that extend beyond the horizontal line indicates autocorrelations and partial autocorrelations that are significant at those lags.

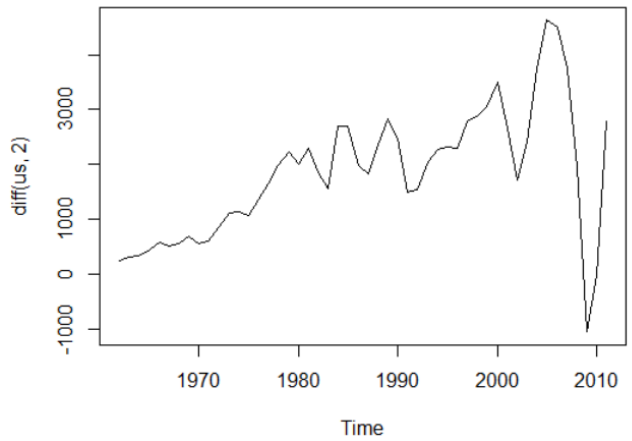
>acf(us)



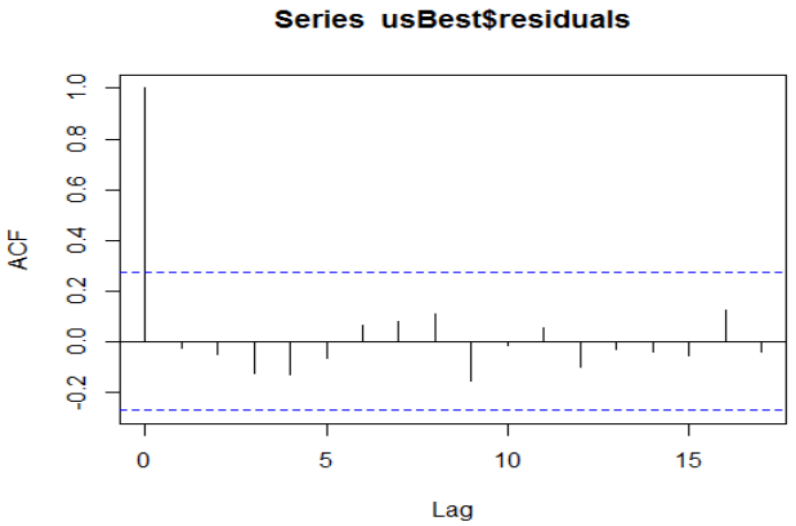
>pacf(us)



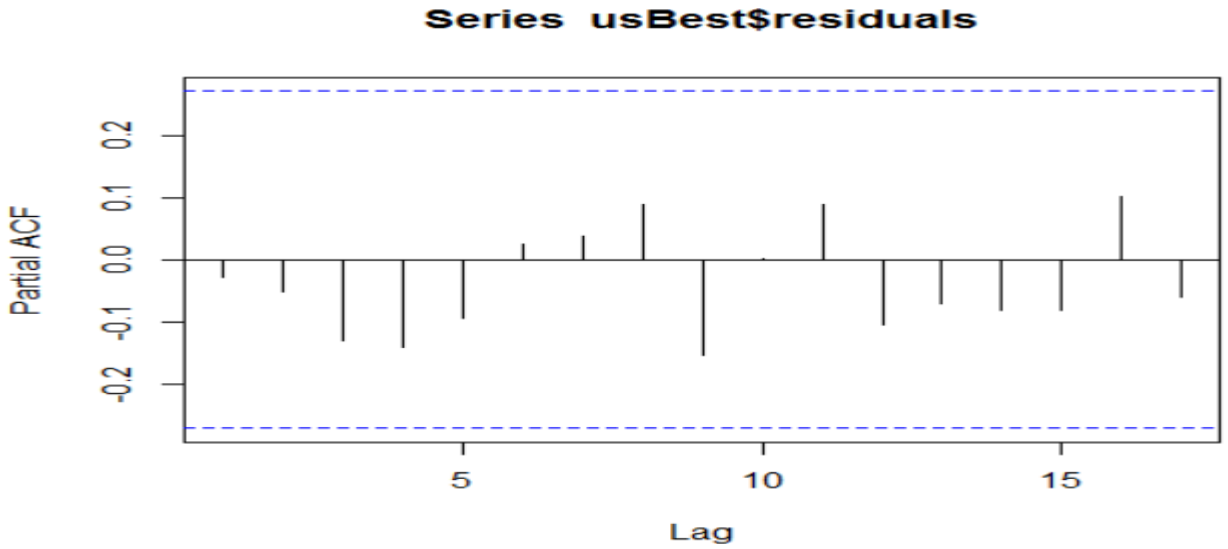
```
>x<-c(1,4,8,2,6,6,5,3)
>diff(x,difference=1)
[1] 3 4 -6 4 0 -1 -2
>diff(x,difference=2)
[1] 1 -10 10 -4 -1 -1
>diff(x,lag=1)
[1] 3 4 -6 4 0 -1 -2
>diff(x,lag=2)
[1] 7 -2 -2 4 -1 -3
>install.packages("forecast")
>require(forecast)
>ndiffs(x=us)
[1] 2
>plot(diff(us,2))
```



```
>usBest<-auto.arima(x=us)
>usBest
Series:
ARIMA(0,2,2)
Coefficients:
          ma1      ma2
      -0.3898  -0.3875
s.e.   0.1339   0.1267
sigma^2 estimated as 296081:  log likelihood=-385.31
AIC=776.62   AICc=777.14   BIC=782.35
>acf(usBest$residuals)
```

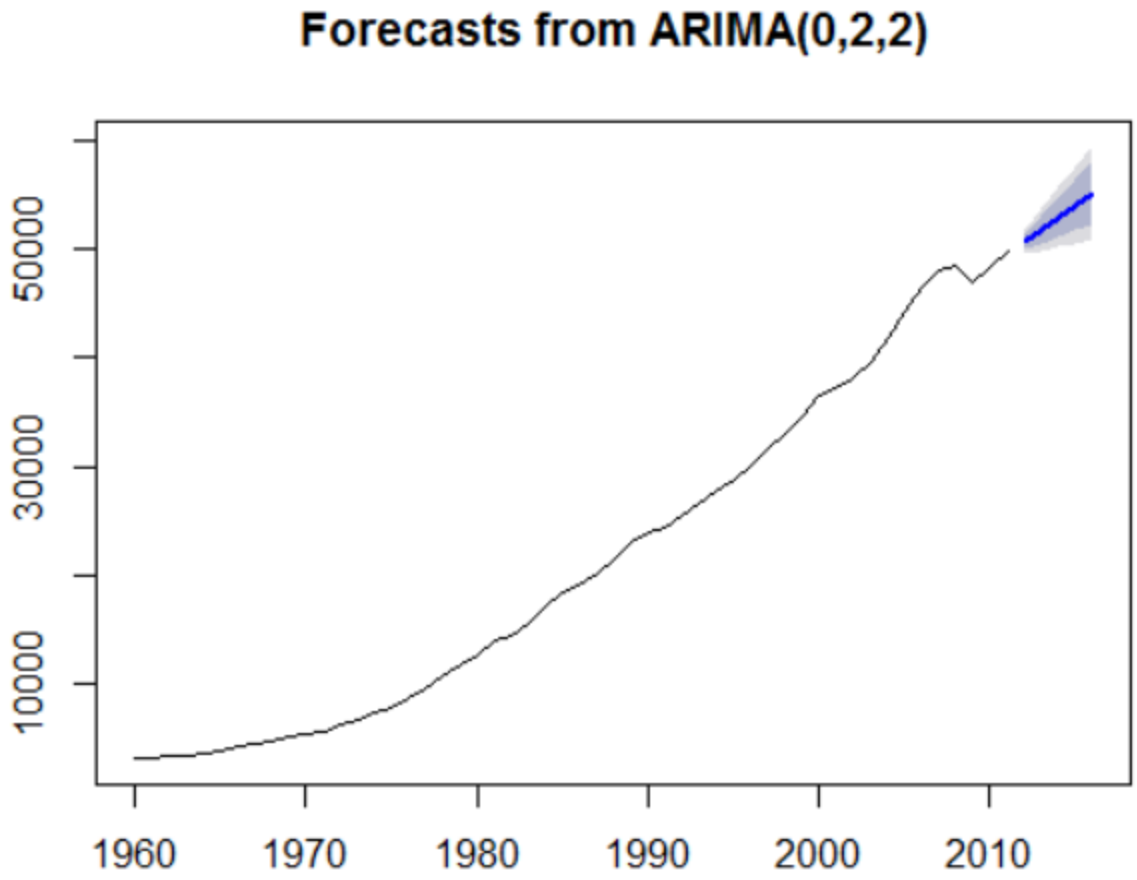


>pacf(usBest$residuals)



>coef(usBest)
 ma1 ma2
-0.3898261 -0.3874942
>predict(usBest,n.ahead=5,se.fit=TRUE)
$pred
Time Series:
Start = 2012
End = 2016
Frequency = 1
[1] 50778.46 51875.18 52971.90 54068.62 55165.35
$se
Time Series:
Start = 2012
End = 2016
Frequency = 1
[1] 544.1332 1031.3671 1434.6979 1819.1658 2201.3858

```
>theForecast<-forecast(object=usBest,h=5)
>plot(theForecast)
```

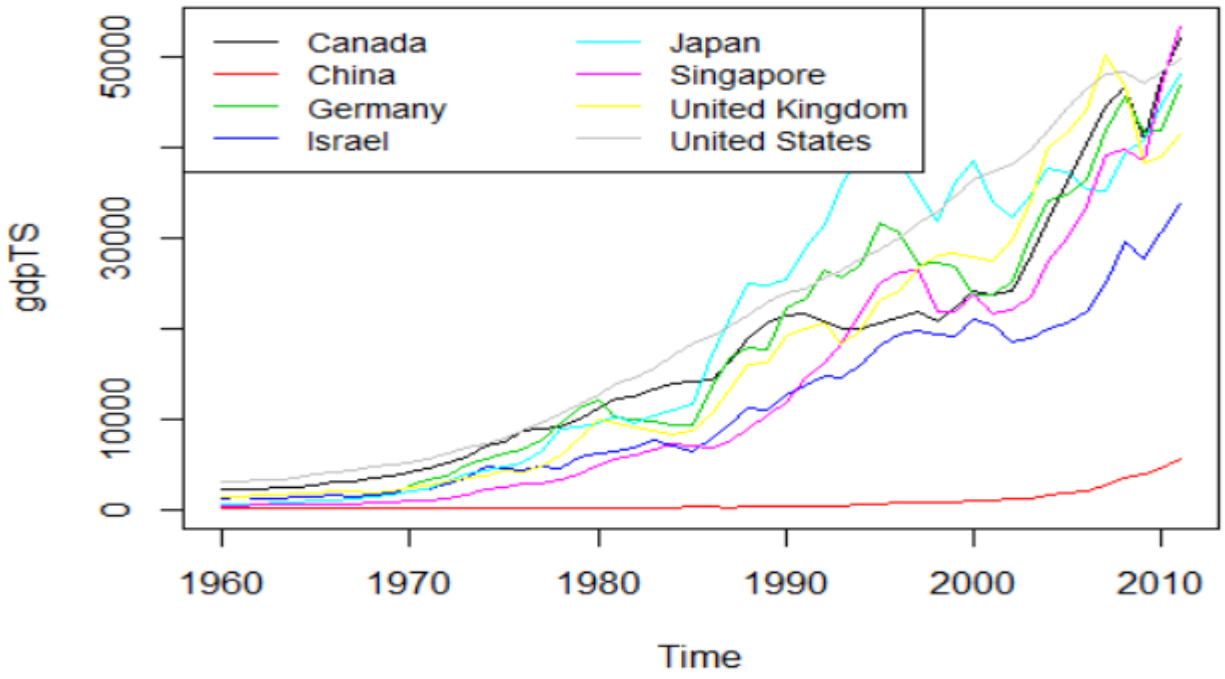


2. VAR

- When dealing with multiple time series where each depends on its own past, others pasts and other presents, things gets more complicated.
- The first thing we will do is convert all of the GDP data into a multivariate time series.
- To do this we first cast the data.frame to wide format then call ts to convert it.

```
>#load reshape2
>require(reshape2)
>#cast the data.frame to wide format
gdpCast<-dcast(Year~Country,
    data=gdp[,c("Country","Year","PerCapGDP")],value.var = "PerCapGDP")
>head(gdpCast)
>#convert to time series
>gdpTS<-ts(data=gdpCast[,-
    1],start=min(gdpCast$Year),end=max(gdpCast$Year))
>#build a plot and legend using base graphics
>plot(gdpTS,plot.type = "single",col=1:8)
```

>legend("topleft",legend=colnames(gdpTS),ncol=2,lty=1,col=1:8,cex=.9)

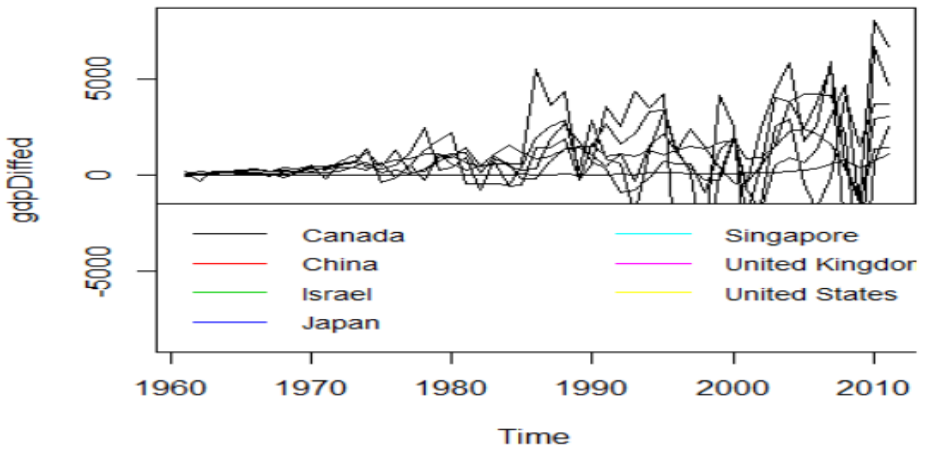


• Before proceeding we have to deal with the NA's for Germany. For some reason the World Bank does not have data on Germany's GDP before 1970.
• There are other resources, such as St.Louis Federal Reserve Economic Data(FRED), but their data do not agree well with the World Bank data, so we remove Germany from our data.

>gdpTS<-gdpTS[,which(colnames(gdpTS)!= "Germany")]

- The most common way of fitting a model to multiple time series is to use a vector autoregressive (VAR) model.
- While ar can compute a VAR, it often has problems with singular matrices when the AR order is high, so it is better to use VAR from the vars package.
- To check whether the data should be diffed, we use the ndiffes function on gdpTS and then apply that number of diffs.

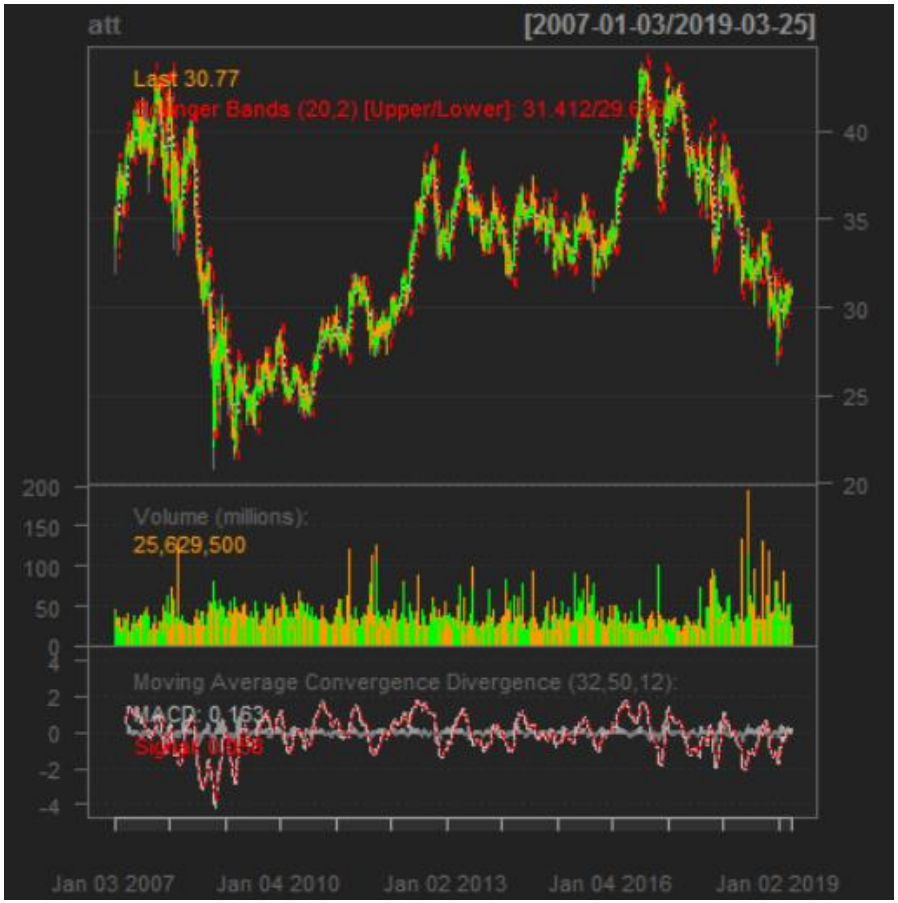
```
>numDiffs<-ndiffs(gdpTS)
>numDiffs
[1] 1
>gdpDiffed<-diff(gdpTS,differences=numDiffs)
>plot(gdpDiffed,plot.type = "single",cols=1:7)
>legend("bottomleft",legend=colnames(gdpDiffed),ncol=2,lty=1,col=1:7,cex=.9
      )
```



3. GARCH

- A problem with ARMA models is that they do not handle extreme events or high volatility well.
- To overcome this a good tool to use is generalized autoregressive conditional heteroskedasticity or the GARCH family of models, which in addition to modeling the mean of the process also model the variance.

```
>require(quantmod)
>att<-getSymbols("T",auto.assign=FALSE)
>att
>require(xts)
>head(att)
>plot(att)
>chartSeries(att)
>addBBands()
>addMACD(32,50,12)
```



Series chart for AT&T

- We are only interested in the closing price, so we create a variable holding just that.

```
>attClose<-att$T.Close
>class(attClose)
[1] "xts" "zoo“
>head(attClose)
           T.Close
2007-01-03   34.95
2007-01-04   34.50
2007-01-05   33.96
2007-01-08   33.81
2007-01-09   33.94
2007-01-10   34.03
```

- The package most widely considered to be the best for fitting GARCH models is rugarch.
- There are other packages for fitting GARCH models, such as tseries, fGarch and bayesGARCH, but we will focus on rugarch.
- Generally GARCH(1,1) will be sufficient so we will fit that model to the data.

- The first step is setting up the model specification using ugarchspec. We specify the volatility to be modeled as a GARCH(1,1) and the mean to be modeled as an ARMA(1,1).
- We also specify that the innovation distribution should be the t distribution.

```
>install.package("rugarch")
>require(rugarch)
>attSpec<-ugarchspec(variance.model=list(model="sGARCH",
    garchOrder=c(1,1)),mean.model=list(armaOrder=c(1,1)),distribution.model
    = "std")
```

- The next step is to fit the model using ugarchfit.

```
>attGarch<-ugarchfit(spec=attSpec,data=attClose)
>attGarch
*---------------------------------*
*          GARCH Model Fit        *
*---------------------------------*
Conditional Variance Dynamics
-----------------------------------
GARCH Model    : sGARCH(1,1)
```

Mean Model : ARFIMA(1,0,1)
Distribution : std

Optimal Parameters

	Estimate	Std. Error	t value	Pr(>|t|)
mu	35.020833	0.324477	107.93021	0.000000
ar1	0.996755	0.001160	859.33792	0.000000
ma1	-0.003829	0.018109	-0.21143	0.832551
omega	0.001567	0.000550	2.84737	0.004408
alpha1	0.044523	0.008178	5.44435	0.000000
beta1	0.947374	0.009382	100.97567	0.000000
shape	5.317563	0.492993	10.78628	0.000000

Robust Standard Errors:

	Estimate	Std. Error	t value	Pr(>|t|)
mu	35.020833	0.093926	372.85481	0.000000
ar1	0.996755	0.001081	922.20587	0.000000
ma1	-0.003829	0.018810	-0.20355	0.838707
omega	0.001567	0.000610	2.56855	0.010213
alpha1	0.044523	0.010298	4.32348	0.000015
beta1	0.947374	0.011461	82.66368	0.000000
shape	5.317563	0.516419	10.29699	0.000000

LogLikelihood : -1385.668

Information Criteria

Akaike 0.90521
Bayes 0.91893
Shibata 0.90520
Hannan-Quinn 0.91014

Weighted Ljung-Box Test on Standardized Residuals
---

	statistic	p-value
Lag[1]	0.078	0.7800
Lag[2*(p+q)+(p+q)-1][5]	2.594	0.7264
Lag[4*(p+q)+(p+q)-1][9]	4.332	0.6119

d.o.f=2
H0 : No serial correlation
Weighted Ljung-Box Test on Standardized Squared Residuals
---

	statistic	p-value
Lag[1]	0.7637	0.3822
Lag[2*(p+q)+(p+q)-1][5]	1.4044	0.7634
Lag[4*(p+q)+(p+q)-1][9]	3.4425	0.6842

d.o.f=2

Weighted ARCH LM Tests

	Statistic	Shape	Scale	P-Value
ARCH Lag[3]	0.3219	0.500	2.000	0.5705
ARCH Lag[5]	0.4133	1.440	1.667	0.9089
ARCH Lag[7]	2.7459	2.315	1.543	0.5628

Nyblom stability test

Joint Statistic: 4.6242

Individual Statistics:

mu	0.47721
ar1	0.08977
ma1	0.09391
omega	0.10109
alpha1	1.03553
beta1	0.47932
shape	1.74524

Asymptotic Critical Values (10% 5% 1%)

Joint Statistic:	1.69 1.9 2.35
Individual Statistic:	0.35 0.47 0.75

Sign Bias Test

---

	t-value	prob sig
Sign Bias	0.08041	0.9359
Negative Sign Bias	0.36999	0.7114
Positive Sign Bias	0.62487	0.5321
Joint Effect	0.85683	0.8358

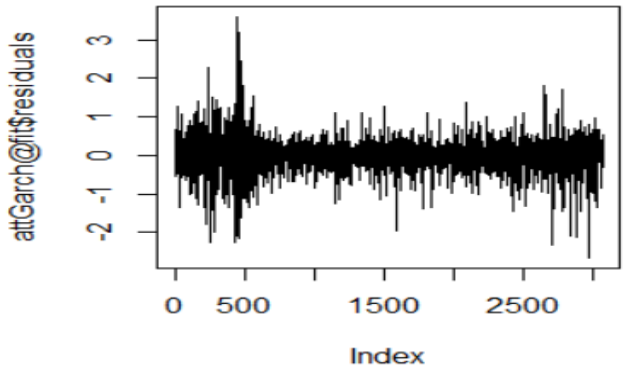
Adjusted Pearson Goodness-of-Fit Test:

---

	group	statistic	p-value(g-1)
1	20	26.94	0.10608
2	30	40.12	0.08193
3	40	43.69	0.27898
4	50	62.15	0.09838

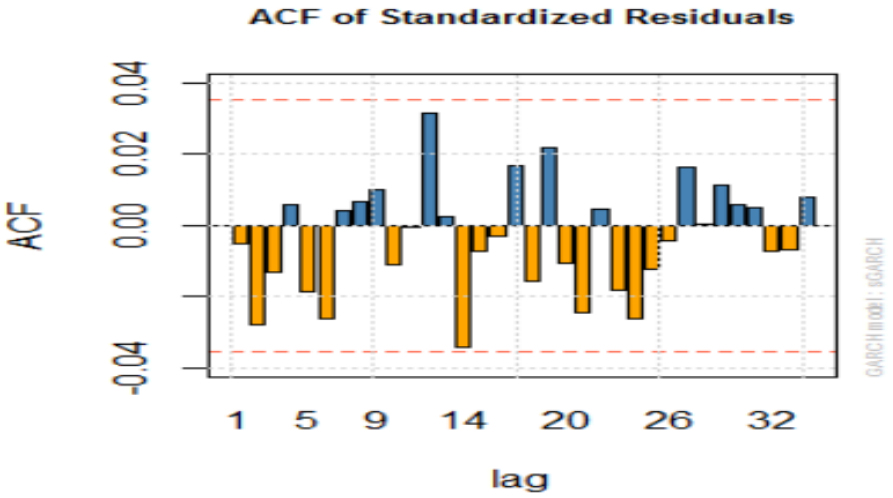
Elapsed time : 1.180067

>plot(attGarch@fit$residuals,type="l")



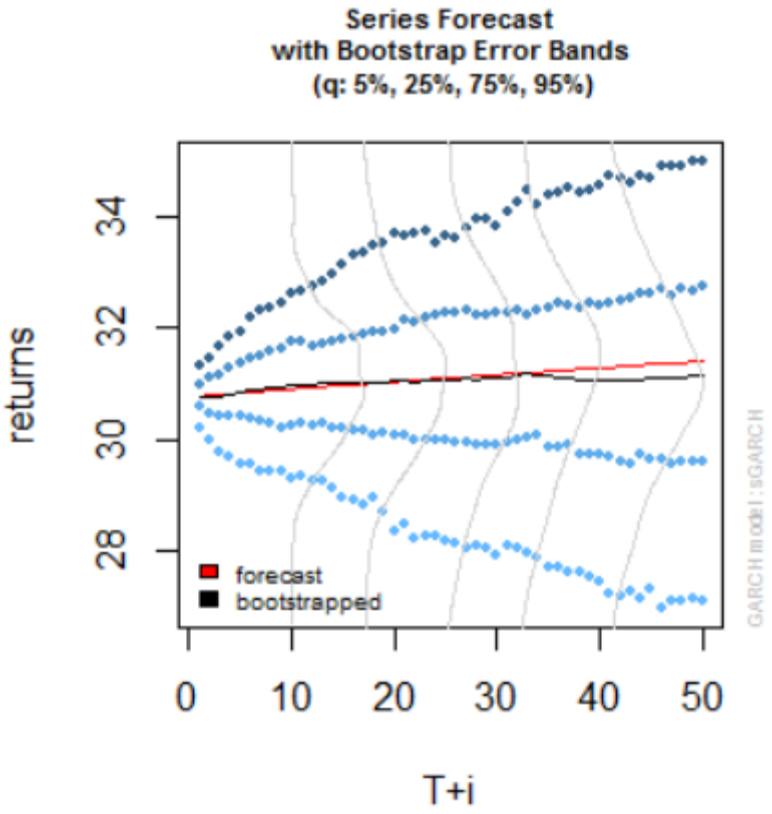
Residuals

>plot(attGarch, which=10)



ACF of residuals

```
>attPred<-ugarchboot(attGarch,n.ahead = 50,method=c("Partial","Full")[1])
>plot(attPred,which=2)
```



Predictions for GARCH model on AT&T data

4. Clustering

- Play's a big role in modern machine learning, is the partitioning of data into groups.
- This can be done in a number of ways, the two most popular being K-means and hierarchical clustering.
- In terms of a data.frame, a clustering algorithm finds out which rows are similar to each other.
- Rows that are grouped together are supposed to have high similarity to each other and low similarity with rows outside the grouping.

1. K-means

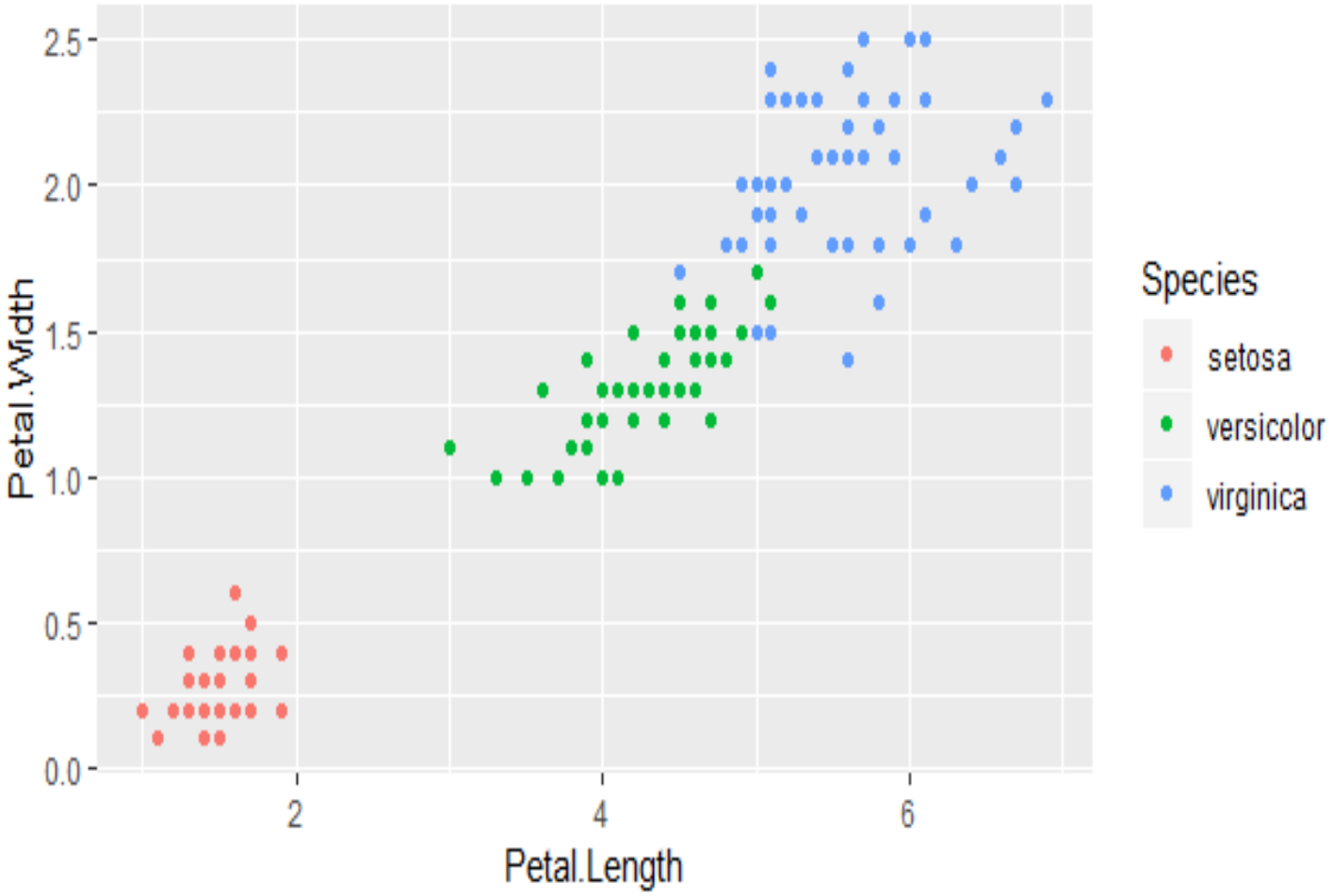
- One of the more popular algorithms for clustering is K-means.
- It divides the observations into discrete groups based on some distance metric.
- For K-means we need to specify the number of clusters, and then the algorithm assigns observations into that many clusters.
- In R, K-means is done with the aptly named K-means function. The first 2 arguments are the data to be clustered, which must be all numeric(K-means does not work with categorical data), and the number of centers(clusters).

- Printing the K-means objects displays the size of the clusters, the cluster mean for each column, the cluster membership for each row and similarity measures.
- Plotting the results of K-means clustering can be difficult because of high dimensional nature of the data. To overcome this, the plot.kmeans function in useful performs multidimensional scaling to project the data into two dimensions, and then color codes the points according to cluster membership.

Code:

```
> library(datasets)
>head(iris)
Petal.Width Species
1         0.2  setosa
2         0.2  setosa
3         0.2  setosa
4         0.2  setosa
5         0.2  setosa
6         0.4  setosa
```

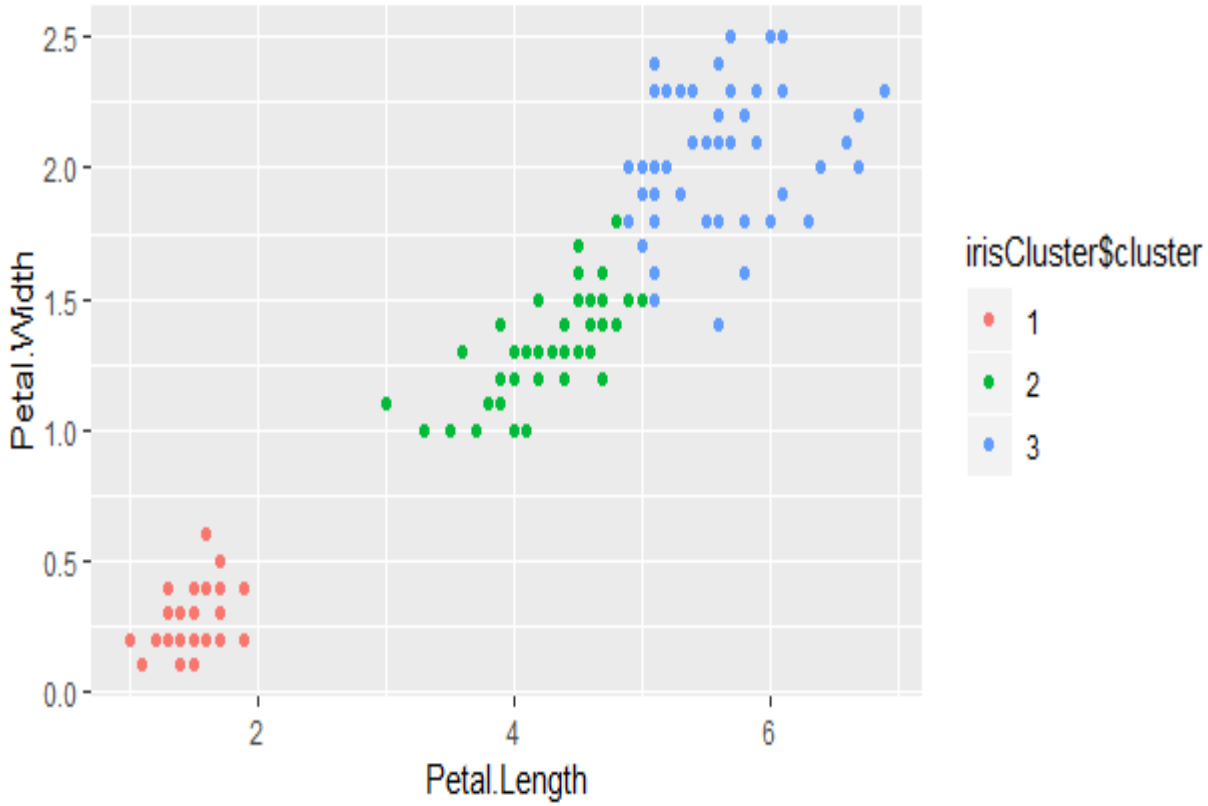
```
>library(ggplot2)
>ggplot(iris,aes(Petal.Length,Petal.Width,color=Species))+geom_point()
```



```
>set.seed(20)
>irisCluster<-kmeans(iris[,3:4],3,nstart=20)
>irisCluster
Clustering vector:
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [25] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [49] 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [73] 2 2 2 2 2 3 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [97] 2 2 2 2 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 2
[121] 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3
[145] 3 3 3 3 3 3
Within cluster sum of squares by cluster:
[1]  2.02200 13.05769 16.29167
 (between_SS / total_SS =  94.3 %)
Available components:
[1] "cluster"      "centers"      "totss"
[4] "withinss"     "tot.withinss" "betweenss"
[7] "size"         "iter"         "ifault"
>
```

```
>table(irisCluster$cluster,iris$Species)
        setosa          versicolor          virginica
  1     50                 0                   0
  2      0                 48                  4
  3      0                 2                   46
> irisCluster$cluster<-as.factor(irisCluster$cluster)
>irisCluster$cluster
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [25] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [49] 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [73] 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2
 [97] 2 2 2 2 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 2
[121] 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3
[145] 3 3 3 3 3 3
Levels: 1 2 3
```

```
>ggplot(iris,aes(Petal.Length,Petal.Width,
        color=irisCluster$cluster))+geom_point()
```



2. PAM

- Two problems with K-means clustering are that it does not work with categorical data and it is susceptible to outliers.
- An alternative is K-medoids. Instead of the center of a cluster being the mean of the cluster, the center is one of the actual observations in the cluster.
- The most common K-mediods algorithm is Partitioning Around Mediods(PAM). The cluster package contains the pam().
- If the data has a few missing values, pam handles missing values well. Before we run the clustering algorithm we clean up the data some more.
- Now we fit the clustering using pam from the cluster package. Each line represents an observation, and each grouping of lines is a cluster.
- Observations that fit the cluster well have large positive lines and observations that do not fit well have small or negative lines.
- A bigger average width for a cluster means a better clustering.

Code:

```
>x<-rbind(cbind(rnorm(10,0,0.5),rnorm(10,0,0.5)),
        cbind(rnorm(15,5,0.5),rnorm(15,5,0.5)))
```

```
>x
            [,1]        [,2]
 [1,] -0.10815576 -0.409241508
 [2,]  0.79507288 -0.771283919
 [3,]  0.77807164  0.277941073
 [4,]  0.55422545 -0.184514485
 [5,] -0.54867092 -0.523669140
 [6,] -0.93030286  0.009089958
 [7,] -0.45678942  0.440938753
 [8,]  0.62278446  0.440930748
 [9,]  0.04392736  0.513121593
[10,]  0.21174095 -0.190654590
[11,]  5.54971762  4.169400227
[12,]  4.98454143  5.582606039
[13,]  5.09516971  4.465169011
[14,]  5.66760327  5.454441770
[15,]  5.36527617  4.341246290
[16,]  5.02810095  5.327714978
[17,]  5.66465281  4.669814088
[18,]  4.79594003  5.495790384
[19,]  4.59087145  4.658388232
[20,]  5.17947283  5.560660598
[21,]  5.02935803  5.194275645
[22,]  4.96528171  4.924616605
[23,]  4.80759187  4.767011671
[24,]  4.56283827  4.319742181
[25,]  5.59816533  4.985207120
```

```
>library(pamr)
>pamx<-pam(x,2)
>pamx
Medoids:
     ID
[1,] 10 0.2117409 -0.1906546
[2,] 22 4.9652817  4.9246166
Clustering vector:
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Objective function:
    build      swap
0.8236515 0.6037261

Available components:
 [1] "medoids"    "id.med"     "clustering" "objective"
 [5] "isolation"  "clusinfo"   "silinfo"    "diss"
 [9] "call"       "data"
```

```
>summary(pamx)
Medoids:
     ID
[1,] 10 0.2117409 -0.1906546
[2,] 22 4.9652817  4.9246166
Clustering vector:
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Objective function:
    build      swap
0.8236515 0.6037261
Numerical information per cluster:
     size  max_diss    av_diss diameter separation
[1,]   10 1.1593800 0.6674374 1.893649   5.528942
[2,]   15 0.9549435 0.5612518 1.583682   5.528942
Isolated clusters:
 L-clusters: character(0)
 L*-clusters: [1] 1 2
```

Silhouette plot information:

	cluster	neighbor	sil_width
10	1	2	0.8956637
1	1	2	0.8881902
4	1	2	0.8783324
9	1	2	0.8655514
5	1	2	0.8633453
7	1	2	0.8573630
8	1	2	0.8474618
3	1	2	0.8442775
6	1	2	0.8406352
2	1	2	0.8265019
21	2	1	0.9142036
22	2	1	0.9137526
16	2	1	0.9092210
23	2	1	0.9010146
25	2	1	0.8990212
20	2	1	0.8938403
13	2	1	0.8921443
12	2	1	0.8898658
17	2	1	0.8866756
18	2	1	0.8865927
14	2	1	0.8801415
15	2	1	0.8787858
19	2	1	0.8751181
11	2	1	0.8520542
24	2	1	0.8412541

```
Average silhouette width per cluster:
[1] 0.8607322 0.8875790
Average silhouette width of total data set:
[1] 0.8768403

300 dissimilarities, summarized :
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.1335  0.8335  3.7113  3.9761  7.0918  8.6244
Metric :  euclidean
Number of objects : 25

Available components:
 [1] "medoids"    "id.med"     "clustering" "objective"
 [5] "isolation"  "clusinfo"   "silinfo"    "diss"
 [9] "call"       "data"
```

```
>(p2m<-pam(x,2,medoids=c(1,16)))
Medoids:
     ID
[1,] 10 0.2117409 -0.1906546
[2,] 22 4.9652817  4.9246166
Clustering vector:
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Objective function:
    build      swap
0.6728364 0.6037261

Available components:
 [1] "medoids"    "id.med"     "clustering" "objective"
 [5] "isolation"  "clusinfo"   "silinfo"    "diss"
 [9] "call"       "data"
```

```
>p2.s<-pam(x,2,medoids=c(1,16),do.swap=FALSE)
>p2.s
Medoids:
     ID
[1,]  1 -0.1081558 -0.4092415
[2,] 16  5.0281009  5.3277150
Clustering vector:
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Objective function:
    build      swap
0.6728364 0.6728364

Available components:
 [1] "medoids"    "id.med"     "clustering" "objective"
 [5] "isolation"  "clusinfo"   "silinfo"    "diss"
 [9] "call"       "data"
```

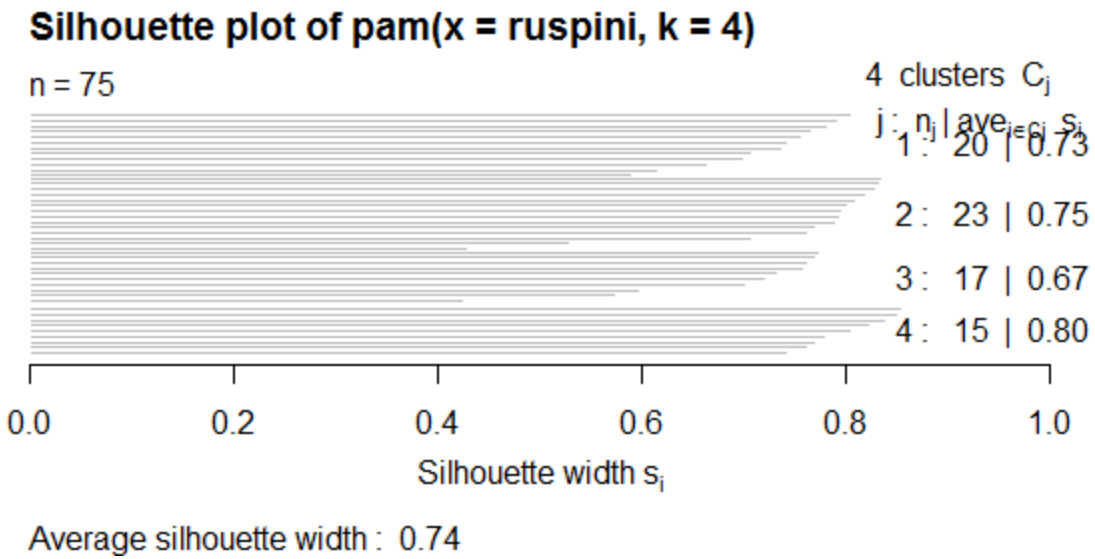
```
>p3m<-pam(x,3,trace=2)
C pam(): computing 301 dissimilarities from  25 x 2  matrix: [Ok]
pam()'s bswap(*, s=8.62438, pamonce=0): build 3 medoids:
     new repr. 24
     new repr. 10
     new repr. 16
  after build: medoids are 10 16 24
   swp new 13 <-> 24 old; decreasing diss. 13.4737 by -1.16452
end{bswap()}, end{cstat()}
>(p3m.<-pam(x,3,medoids = 3:1,trace=1))
C pam(): computing 301 dissimilarities from  25 x 2  matrix: [Ok]
pam()'s bswap(*, s=8.62438, pamonce=0): medoids given
  after build: medoids are  1  2  3
end{bswap()}, end{cstat()}
```

```
C pam(): computing 301 dissimilarities from  25 x 2  matrix: [Ok]
pam()'s bswap(*, s=8.62438, pamonce=0): medoids given
  after build: medoids are  1  2  3
end{bswap()}, end{cstat()}
Medoids:
      ID
[1,]  1 -0.1081558 -0.4092415
[2,]  8  0.6227845  0.4409307
[3,] 22  4.9652817  4.9246166
Clustering vector:
 [1] 1 1 2 2 1 1 1 2 2 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
Objective function:
    build      swap
3.9953828 0.5405376
Available components:
 [1] "medoids"    "id.med"     "clustering" "objective"
 [5] "isolation"  "clusinfo"   "silinfo"    "diss"
 [9] "call"       "data"
```

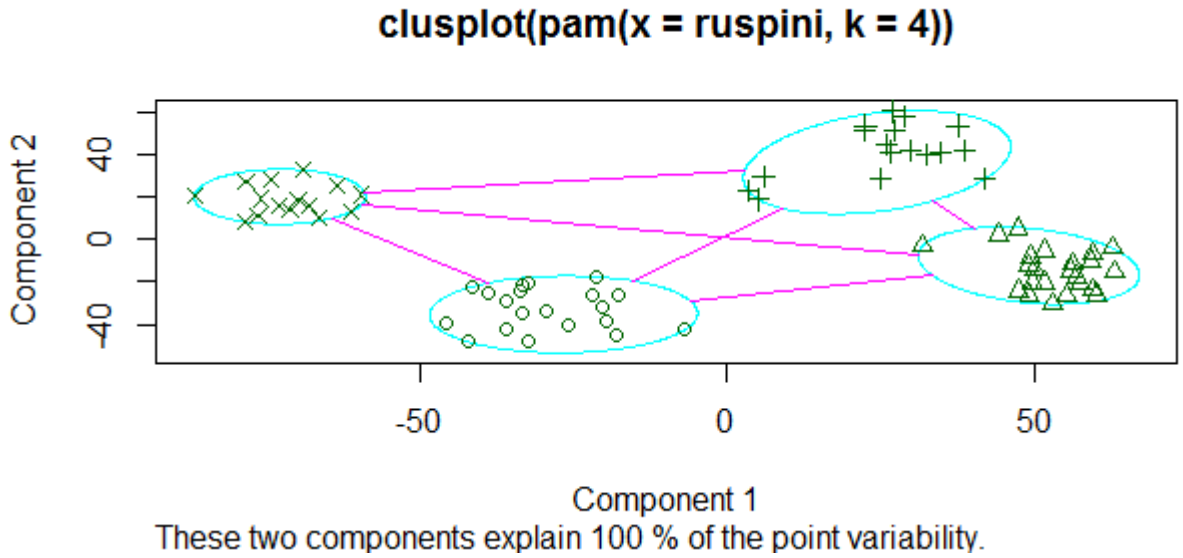
```
>pam(daisy(x,metric = "manhattan"),2,diss=TRUE)
Medoids:
     ID
[1,] 10 10
[2,] 22 22
Clustering vector:
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Objective function:
    build      swap
1.0639279 0.7823442

Available components:
[1] "medoids"    "id.med"     "clustering" "objective"
[5] "isolation"  "clusinfo"   "silinfo"    "diss"
[9] "call"
```

```
>data("ruspini")
>plot(pam(ruspini,4),ask=TRUE)
Make a plot selection (or 0 to exit):
 1: plot  All
2: plot  Clusplot
3: plot  Silhouette Plot
```

Selection: 1



Selection:2


clusplot(pam(x = ruspini, k = 4))
Component 2
40
0
-40
-50
0
50
Component 1
These two components explain 100 % of the point variability.

3. Hierarchical Clustering:

- Hierarchical Clustering builds clusters within clusters, and does not require a prespecified number of cluster like K-means and K-medoids do.
- A Hierarchical Clustering can be thought of as a tree and displayed as a dendogram, at the top there is just one cluster consisting of all the observations, and at the bottom each observation is an entire cluster.
- In between are varying levels of clustering.
- Hierarchical clustering also works on categorical data like the country information data. However, its dissimilarity matrix must be calculated differently.
- There are a number of ways to compute the distance between clusters and they can have a significant impact on the results of a hierarchical clustering.

Code:

```
>clusters<-hclust(dist(iris[,3:4]))
>clusters
Call:
hclust(d = dist(iris[, 3:4]))
```

Cluster method : complete

Distance : euclidean

Number of objects: 150

>plot(clusters)