

Analysis of variance (ANOVA) is a statistical technique that is used to check if the means of two or more groups are significantly different from each other. ANOVA checks the impact of one or more factors by comparing the means of different samples.

Example:

ANOVA can be used to prove/disprove if all the medication treatments were equally effective or not.

ANOVA can be used to study of the effects of fertilizer type and planting density on crop yield.

The null hypothesis (H_0) of the ANOVA is no difference in means, and the alternate hypothesis (H_a) is that the means are different from one another.

Sample dataset contains observations from an imaginary study of the effects of fertilizer type and planting density on crop yield.

One-way ANOVA example

In the one-way ANOVA, we test the effects of 3 types of fertilizer on crop yield.

Two-way ANOVA example

In the two-way ANOVA, we add an additional independent variable: planting density. We test the effects of 3 types of fertilizer and 2 different planting densities on crop yield.

Install and load the packages

```
install.packages(c("ggplot2", "ggpubr", "tidyverse", "broom", "AICcmodavg"))
```

Then load these packages into your R environment (do this every time you restart the R program):

```
library(ggplot2)
library(ggpubr) library(ggpubr)
library(tidyverse)
library(broom)
library(AICcmodavg)
```

Step 1: Load the data into R

```
one.way <- aov(yield ~ fertilizer, data = crop.data)
```

```
summary(one.way)
```

density	block	fertilizer	yield
1:48	1:24	1:32	Min. :175.4
2:48	2:24	2:32	1st Qu.:176.5
	3:24	3:32	Median :177.1
	4:24		Mean :177.0
			3rd Qu.:177.4
			Max. :179.1

You should see 'density', 'block', and 'fertilizer' listed as categorical variables with the number of observations at each level (i.e. 48 observations at density 1 and 48 observations at density 2).

'Yield' should be a quantitative variable with a numeric summary (minimum, median, mean, maximum).

Step 2: Perform the ANOVA test

ANOVA tests whether any of the group means are different from the overall mean of the data by checking the variance of each individual group against the overall variance of the data. If one or more groups falls outside the range of variation predicted by the null hypothesis (all group means are equal), then the test is statistically significant.

We can perform an ANOVA in R using the `aov()` function. This will calculate the test statistic for ANOVA and determine whether there is significant variation among the groups formed by the levels of the independent variable.

One-way ANOVA

In the one-way ANOVA example, we are modeling crop yield as a function of the type of fertilizer used. First we will use `aov()` to run the model, then we will use `summary()` to print the summary of the model.

```
one.way <- aov(yield ~ fertilizer, data = crop.data)
```

```
summary(one.way)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
fertilizer	2	6.07	3.0340	7.863	7e-04	***
Residuals	93	35.89	0.3859			

 signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

The model summary first lists the independent variables being tested in the model (in this case we have only one, 'fertilizer') and the model residuals ('Residual'). All of the variation that is not explained by the independent variables is called residual variance.

The rest of the values in the output table describe the independent variable and the residuals:

The Df column displays the degrees of freedom for the independent variable (the number of levels in the variable minus 1), and the degrees of freedom for the residuals (the total number of observations minus one and minus the number of levels in the independent variables).

The Sum Sq column displays the sum of squares (a.k.a. the total variation between the group means and the overall mean).

The Mean Sq column is the mean of the sum of squares, calculated by dividing the sum of squares by the degrees of freedom for each parameter.

The F-value column is the test statistic from the F test. This is the mean square of each independent variable divided by the mean square of the residuals. The larger the F value, the more likely it is that the variation caused by the independent variable is real and not due to chance.

The Pr(>F) column is the p-value of the F-statistic. This shows how likely it is that the F-value calculated from the test would have occurred if the null hypothesis of no difference among group means were true.

The p-value of the fertilizer variable is low ($p < 0.001$), so it appears that the type of fertilizer used has a real impact on the final crop yield.

Two-way ANOVA

In the [two-way ANOVA](#) example, we are modeling crop yield as a function of type of fertilizer and planting density. First we use `aov()` to run the model, then we use `summary()` to print the summary of the model.

```
two.way <- aov(yield ~ fertilizer + density, data = crop.data)
```

```
summary(two.way)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
fertilizer	2	6.068	3.034	9.073	0.000253	***
density	1	5.122	5.122	15.316	0.000174	***
Residuals	92	30.765	0.334			

signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Adding planting density to the model seems to have made the model better: it reduced the residual variance (the residual sum of squares went from 35.89 to 30.765), and both planting density and fertilizer are statistically significant (p-values < 0.001).

Adding interactions between variables

Sometimes you have reason to think that two of your independent variables have an interaction effect rather than an additive effect.

For example, in our crop yield experiment, it is possible that planting density affects the plants' ability to take up fertilizer. This might influence the effect of fertilizer type in a way that isn't accounted for in the two-way model.

To test whether two variables have an interaction effect in ANOVA, simply use an asterisk instead of a plus-sign in the model:

```
interaction <- aov(yield ~ fertilizer*density, data = crop.data)
```

```
summary(interaction)
```

In the output table, the 'fertilizer:density' variable has a low sum-of-squares value and a high p-value, which means there is not much variation that can be explained by the interaction between fertilizer and planting density.

Adding a blocking variable

If you have grouped your experimental treatments in some way, or if you have a confounding variable that might affect the relationship you are interested in testing, you should include that element in the model as a blocking variable. The simplest way to do this is just to add the variable into the model with a '+’.

For example, in many crop yield studies, treatments are applied within ‘blocks’ in the field that may differ in soil texture, moisture, sunlight, etc. To control for the effect of differences among planting blocks we add a third term, ‘block’, to our ANOVA.

```
blocking <- aov(yield ~ fertilizer + density + block, data = crop.data)
```

```
summary(blocking)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
fertilizer	2	6.068	3.034	9.018	0.000269	***
density	1	5.122	5.122	15.224	0.000184	***
block	2	0.486	0.243	0.723	0.488329	
Residuals	90	30.278	0.336			

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

The ‘block’ variable has a low sum-of-squares value (0.486) and a high p-value ($p = 0.48$), so it’s probably not adding much information to the model. It also doesn’t change the sum of squares for the two independent variables, which means that it’s not affecting how much variation in the dependent variable they explain.

Step 3: Find the best-fit model

There are now four different ANOVA models to explain the data. How do you decide which one to use? Usually you’ll want to use the ‘best-fit’ model – the model that best explains the variation in the dependent variable.

The [Akaike information criterion](#) (AIC) is a good test for model fit. AIC calculates the information value of each model by balancing the variation explained against the number of parameters used.

In AIC model selection, we compare the information value of each model and choose the one with the lowest AIC value (a lower number means more information explained!)

```
library(AICcmodavg)
```

```
model.set <- list(one.way, two.way, interaction, blocking)  
model.names <- c("one.way", "two.way", "interaction", "blocking")
```

```
aictab(model.set, modnames = model.names)
```

The model with the lowest AIC score (listed first in the table) is the best fit for the data:

Model selection based on AICc:

	K	AICc	Delta_AICc	AICcwt	Cum.wt	LL
two.way	5	173.86	0.00	0.71	0.71	-81.59
blocking	7	176.93	3.08	0.15	0.86	-80.83
interaction	7	177.12	3.26	0.14	1.00	-80.92
one.way	4	186.41	12.56	0.00	1.00	-88.99

From these results, it appears that the two.way model is the best fit. The two-way model has the lowest AIC value, and 71% of the AIC weight, which means that it explains 71% of the total variation in the dependent variable that can be explained by the full set of models.

The model with blocking term contains an additional 15% of the AIC weight, but because it is more than 2 delta-AIC worse than the best model, it probably isn't good enough to include in your results.

Step 4: Check for homoscedasticity

To check whether the model fits the assumption of homoscedasticity, look at the model diagnostic plots in R using the `plot()` function:

```
par(mfrow=c(2,2))
plot(two.way)
par(mfrow=c(1,1))
```

##K-means clustering Code##

Use built in data set in R - iris

help(iris) - To understand the data set.

```
> newiris <- iris
```

```
> newiris$Species <- NULL
```

#Apply kmeans to newiris, and store the

#clustering result in kc.

#The cluster number is set to 3.

```
->(kc <- kmeans(newiris, 3))
```

```
K-means clustering with 3 clusters of sizes 38, 50, 62
Cluster means:
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1      6.850000      3.073684      5.742105      2.071053
2      5.006000      3.428000      1.462000      0.246000
3      5.901613      2.748387      4.393548      1.433871
Clustering vector:
 [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[30] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 1 3 3 3 3 3
[59] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 3 3 3 3 3 3 3 3 3
[88] 3 3 3 3 3 3 3 3 3 3 3 3 3 1 3 1 1 1 1 3 1 1 1 1 1 1 3 3 1
[117] 1 1 1 3 1 3 1 3 1 1 3 3 1 1 1 1 1 3 1 1 1 1 3 1 1 1 3 1 1
[146] 1 3 1 1 3
Within cluster sum of squares by cluster:
 [1] 23.87947 15.15100 39.82097
Available components:
 [1] "cluster" "centers" "withinss" "size"
```

Compare the Species label with the clustering result

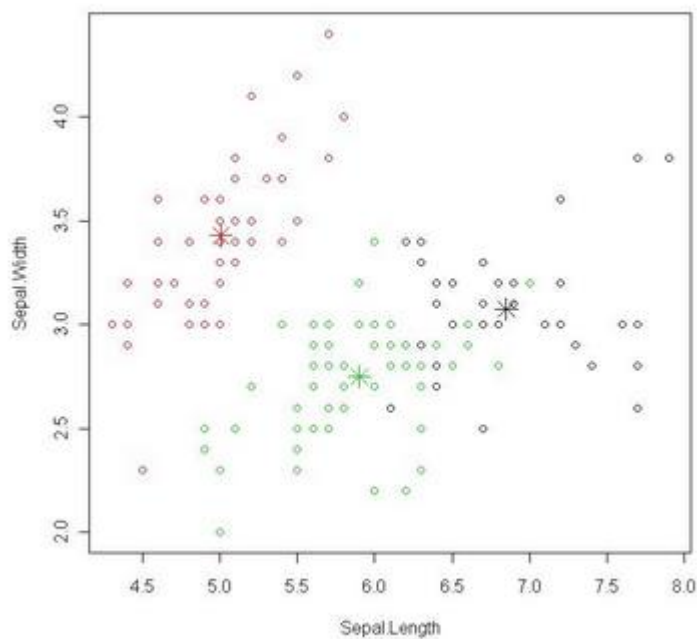
```
> table(iris$Species, kc$cluster)
```

	1	2	3
setosa	0	50	0
versicolor	2	0	48
virginica	36	0	14

Plot the clusters and their centres. Note that there are four dimensions in the data and that only the first two dimensions are used to draw the plot below. Some black points close to the green centre (asterisk) are actually closer to the black centre in the four dimensional space.

```
> plot(newiris[c("Sepal.Length", "Sepal.Width")], col=kc$cluster)
> points(kc$centers[,c("Sepal.Length", "Sepal.Width")], col=1:3, pch=8, cex=2)
```

This page demonstrates k-means clustering with R.




```
####Decision Tree###
```

```
library(party)
```

```
print(head(readingSkills)) # Load the party package. It will automatically load other  
# dependent packages.
```

```
library(party)
```

```
# Create the input data frame.
```

```
input.dat <- readingSkills[c(1:105),]
```

```
# Give the chart file a name.
```

```
png(file = "decision_tree.png")
```

```
# Create the tree.
```

```
output.tree <- ctree(  
  nativeSpeaker ~ age + shoeSize + score,  
  data = input.dat)
```

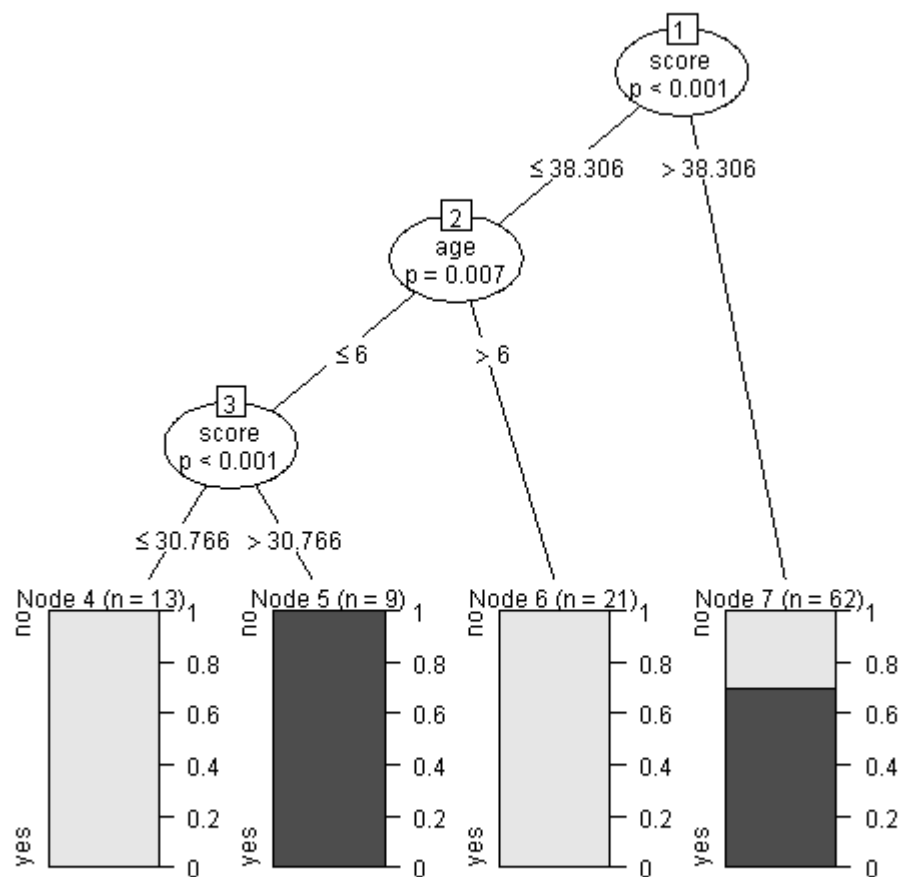
```
# Plot the tree.
```

```
plot(output.tree)
```

```
# Save the file.
```

```
dev.off()
```

	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



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.

use the **randomForest()** function to create the decision tree and see it's graph.

Load the party package. It will automatically load other

required packages.

```
library(party)
```

```
library(randomForest)
```

Create the forest.

```
output.forest <- randomForest(nativeSpeaker ~ age + shoeSize + score,
  data = readingSkills)
```

View the forest results.

```
print(output.forest)
```

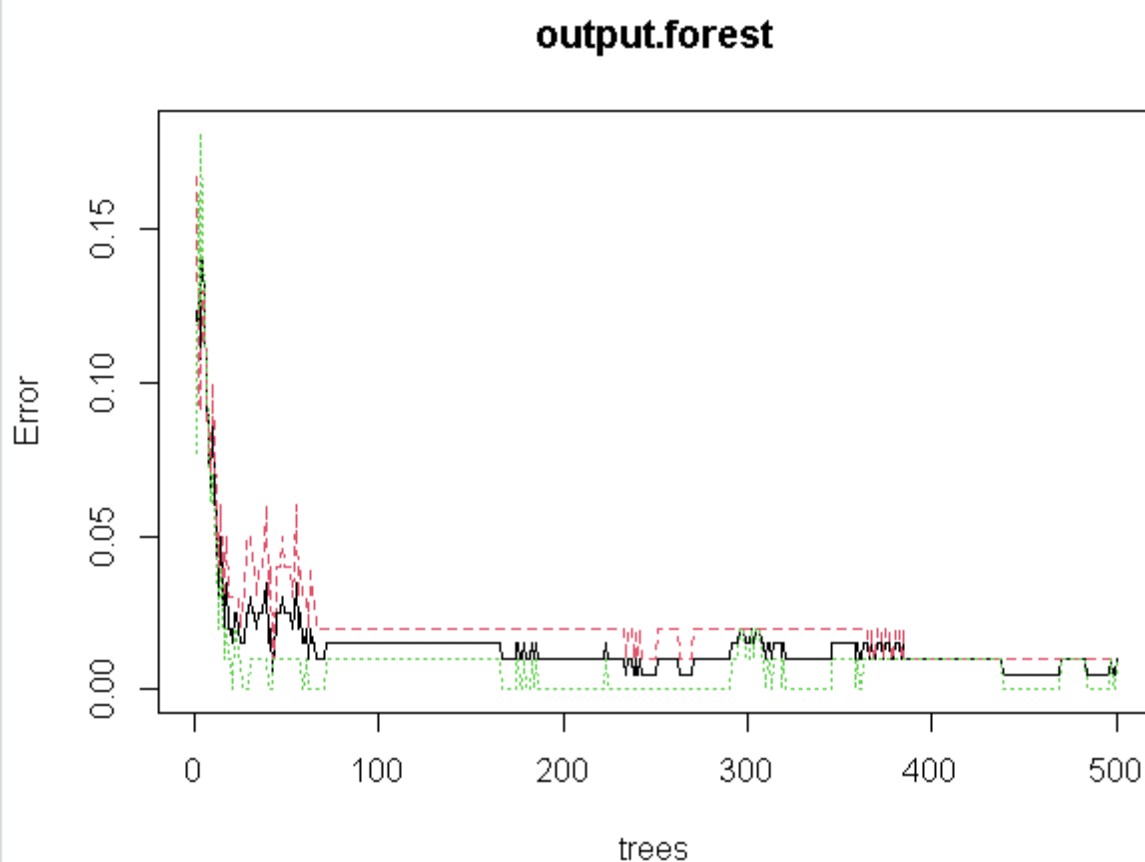
```
# Importance of each predictor.  
print(importance(fit,type = 2))
```

When we execute the above code, it produces the following result –

```
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.



Practical No 10: Practical of Hypothesis testing

Basic terminology

- Variable (under study) – What you measure (like marks of students)
- Population- Set of all units in the study
- Sample- Subset of units selected from population
- Distribution-How values of variable are distributed in the population (like normal distribution)
- Factor- Defines subgroups in the study(like gender)
- Descriptive Statistics- mean, median, standard deviation etc.

What is Statistical Inference?

Research Question: What is the average salary of junior data scientists in India?

Sample of 250 junior data scientists observed

Sample mean is computed

Average salary of population of junior data scientists is estimated using sample of size 250

Research Question: What is the percentage of data scientists in India who use R for data analysis?

Sample of 430 data scientists observed

Sample proportion (or percentage) is computed

Percentage of data scientists using R is estimated using sample of size 430

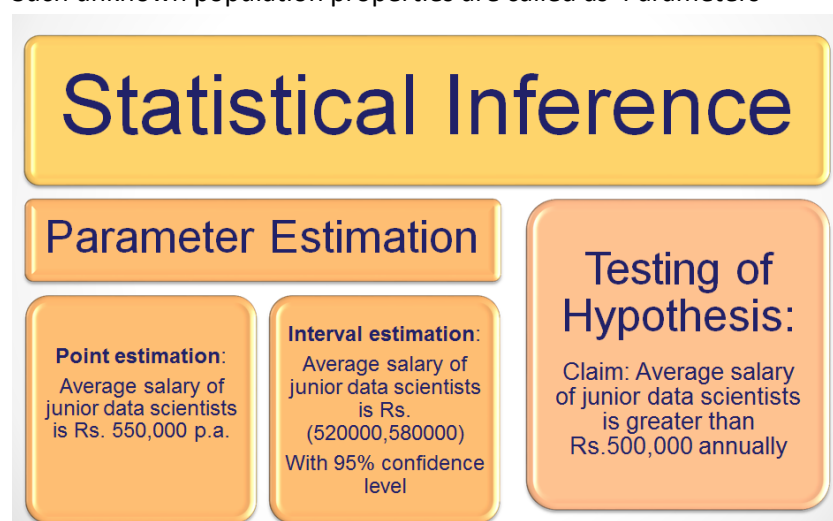
Statistical Inference

Statistical inference is making probabilistic statements about unknown properties of the population.

These unknown population properties can be:

- Mean
- Proportion
- Variance Variance, Proportions etc.

Such unknown population properties are called as 'Parameters'

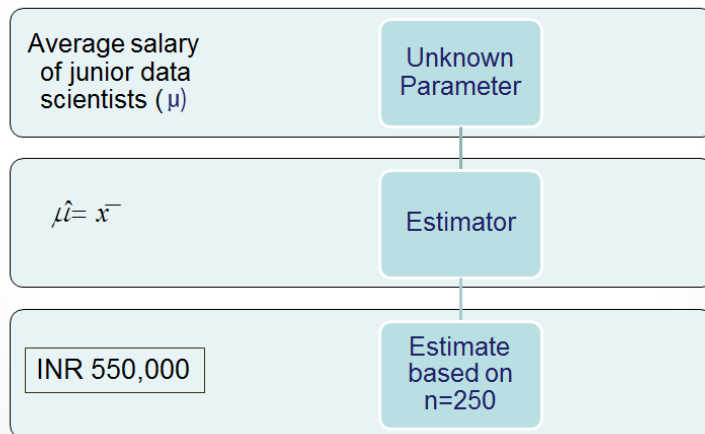


Parameter: Unknown property or characteristic of population - (population mean, variance, proportion)

Estimator: A rule or function based on sample observations which is used to estimate the parameter (sample mean, sample variance, sample proportion)

Estimate: A particular value computed by substituting the sample observations into an Estimator.

- **Research Question: What is the average salary of junior data scientists in India?**



Testing of Hypothesis

Hypothesis: An assertion about the distribution / parameter of the distribution of one or more random variables.

Null Hypothesis: An assertion which is generally believed to be true until researcher rejects it with evidence
Alternative Hypothesis: A researcher's claim which contradicts null hypothesis

In simple words, testing of hypothesis is to decide whether a statement regarding population parameter is true or false, based on sample data.

Example:

A consumer protection agency wants to test a paint manufacturer's claim, that average drying time of their new paint is less than 20 minutes.

Sample: n=36 boards were painted from 36 different cans and the drying time was observed.

Estimator of mean drying time is sample mean

Null Hypothesis: $H_0: \mu = 20$; Alternative Hypothesis: $H_a: \mu < 20$

Decision rule: Reject null hypothesis $\hat{\mu} = \bar{x}$

Test Statistic:

The statistic on which decision rule is defined. In this case the test statistic is: \bar{x}

- Rejecting the null hypothesis when it is in fact true is called a Type I error.
- Not rejecting (accepting) the null hypothesis when in fact it is false is called a Type II error.

$$\alpha = \text{Pr} [\text{Type I Error}] = \text{Pr} [\text{Reject } H_0 \mid H_0 \text{ is True}]$$

$$\beta = \text{Pr} [\text{Type II Error}] = \text{Pr} [\text{Accept } H_0 \mid H_0 \text{ is False}]$$

Probability of Type I error is called as 'Level of Significance (α)' generally set as 5% ($\alpha=0.05$) and null hypothesis is rejected if observed risk(p value) is less than 0.05

Power of the test is given by: $(1 - \beta)$

One Sample t test (for mean)

- One sample t test is used to test the hypothesis about a single population mean.
- We use one-sample t-test when we collect data on a single sample drawn from a defined population.
- In this design, we have one group of subjects, collect data on these subjects and compare our sample statistic to the hypothesized value of population parameter.
- Subjects in the study can be patients, customers, retail stores etc.

Example

A large company is concerned about time taken by employees to complete weekly MIS report. The general manager claims that '**average time taken to complete the MIS report is more than 90 minutes**'.

Time taken to complete MIS report is observed for 12 randomly selected employees.

Time
85
95
105
85
90
97
104
95
88
90
94
95

Null Hypothesis $H_0: \mu = \mu_0$

Alternative Hypothesis $H_1: \mu > \mu_0$

Here $\mu_0 = 90$ (test value)

Assumptions for one sample t test

The assumptions of the one-sample t-test are listed below:

- Random sampling from a defined population
- Population is normally distributed

The validity of the test is not seriously affected by moderate deviations from 'Normality' assumption.

Test Statistic

- The test statistic for one sample t test is given below:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

- Where \bar{x} is the sample mean, s is the sample standard deviation and n is the sample size and μ_0 is the test value.
- The quantity t follows a distribution called as 't distribution' with $n-1$ degrees of freedom.

Most of the software's provide p value.

If p value is less than 0.05 (Level of Significance) then Reject H0 (Here p value is 0.0407)

Inference: Reject H0 and conclude that time taken to complete MIS report is significantly more than 90 minutes.

One Sample t test using R

#Import csv data set ONE SAMPLE t TEST

```
mistime<-read.csv(file.choose(),header=T)
```

Use t.test by specifying variable, alternate hypothesis and μ_0

```
t.test(mistime$Time, alternative="greater", mu=90)
```

#mistime\$Time is the variable under study

alternative="greater" and mu=90 specify:

Null Hypothesis: $H_0: \mu=90$

Alternative Hypothesis: $H_1: \mu > 90$,

One Sample t-test

data: mistime\$Time

t = 1.9176, df = 11, p-value = 0.04074 → Reject H_0

alternative hypothesis: true mean is greater than 90

95 percent confidence interval:

90.22748 Inf

sample estimates:

mean of x

93.58333

t is the calculated value of the test statistic and p-value is used to make inference about acceptance or rejection of null hypothesis (if p-value < 0.05 reject H_0)

Inference: Reject H_0 and conclude that time taken to complete MIS report is significantly more than 90 minutes.

11

Independent Samples t test Equality of two means

The independent-samples t-test compares the means of two independent groups on the same continuous variable.

Following hypotheses are tested in independent samples t test

H0: Two population means are equal

H1: Two population means are not equal

$$H_0 : \mu_1 = \mu_2$$

$$H_1 : \mu_1 \neq \mu_2$$

Independent Samples t test Example

- **The company is assessing the difference in time to complete MIS report between two groups of employees**

Group I: Experience(0-1 years)

Group II: Experience(1-2 years)

Time_G1	Time_G2
85	83
95	85
105	96
85	94
90	102
97	100
104	94
95	95
88	88
90	92
94	95
95	94
	95
	90

Null Hypothesis H0: Average time is equal for 2 groups

Alternative Hypothesis H1: Not H0

Assumptions for independent samples t test

- The assumptions for independent samples t-test are listed below.
- The samples drawn are random samples.
- The populations from which samples are drawn have equal &
- unknown variances.(there is a test to check this assumption)
- The populations follow normal distribution.(there is a test to check this assumption)

Independent Samples t test using R

#Import csv data set INDEPENDENT SAMPLE t TEST

```
mistime2<-read.csv(file.choose(),header=T)
```

#Use t.test with variables ,alternate hypothesis and equal variance assumption


```
t.test(mistime2$time_g1,mistime2$time_g2, alternative="two.sided",  
       var.equal=TRUE)
```

alternative="two.sided" specify: $H_0: \mu_1 = \mu_2$ and $H_1: \mu_1 \neq \mu_2$,

var.equal=TRUE assumes variances in 2 groups are equal

- Two Sample t-test

data: time_g1 and time_g2

t = 0.2235, df = 24, p-value = 0.8251  Do not reject H_0

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

-4.216185 5.239994

sample estimates:

mean of x mean of y

93.58333 93.07143

p-value > 0.05, therefore we do not reject H_0

Inference: there is no significant difference in time taken to complete MIS report between two groups of employees

```
###LOGISTICS REGRESSION###
```

```
# Installing the package
```

```
install.packages("caTools") # For Logistic regression
```

```
install.packages("ROCR")# For ROC curve to evaluate model
```

```
# Loading package
```

```
library(caTools)
```

```
library(ROCR)
```

```
# Splitting dataset
```

```
split <- sample.split(mtcars, SplitRatio = 0.8)
```

```
split
```

```
  [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  
 [11] FALSE
```

```
train_reg <- subset(mtcars, split == "TRUE")
```

```
test_reg <- subset(mtcars, split == "FALSE")
```

```
# Training model
```

```
logistic_model <- glm(vs ~ wt + disp, data = train_reg, family = "binomial")
```

```
logistic_model
```

```
Call: glm(formula = vs ~ wt + disp, family = "binomial", data = train_reg)
```

```
Coefficients:
```

```
(Intercept)          wt          disp  
    2.27215      0.90217     -0.02608
```

```
Degrees of Freedom: 23 Total (i.e. Null); 21 Residual
```

```
Null Deviance:      32.6
```

```
Residual Deviance: 16.92      AIC: 22.92
```

```
# Summary
```

```
summary(logistic_model)
```

```
Call:
glm(formula = vs ~ wt + disp, family = "binomial", data = train_reg)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.6504  -0.3964  -0.1307   0.5787   1.8689

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)  2.27215    2.64176   0.860   0.390
wt           0.90217    1.64319   0.549   0.583
disp        -0.02608    0.01562  -1.670   0.095 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 32.601  on 23  degrees of freedom
Residual deviance: 16.920  on 21  degrees of freedom
AIC: 22.92

Number of Fisher Scoring iterations: 6
```

```
# Predict test data based on model
```

```
predict_reg <- predict(logistic_model, test_reg, type = "response")
```

```
predict_reg
```

```
Mazda RX4 Wag           Merc 280           Merc 280C
 0.66681476           0.73211231           0.73211231
Merc 450SL      Toyota Corona Dodge Challenger
 0.17445655           0.79646316           0.05498165
Camaro Z28      Volvo 142E
 0.03261184           0.83549031
```

```
# Changing probabilities
```

```
predict_reg <- ifelse(predict_reg > 0.5, 1, 0)
```

```
# Evaluating model accuracy
```

```
# using confusion matrix
```

```
table(test_reg$vs, predict_reg)
```

```

predict_reg
  0 1
0 3 1
1 0 4

```

```
missing_classerr <- mean(predict_reg != test_reg$vs)
```

```
print(paste('Accuracy =', 1 - missing_classerr))
```

```
[1] "Accuracy = 0.875"
```

```
# ROC-AUC Curve
```

```
ROCPred <- prediction(predict_reg, test_reg$vs)
```

```
ROCPer <- performance(ROCPred, measure = "tpr", x.measure = "fpr")
```

```
auc <- performance(ROCPred, measure = "auc")
```

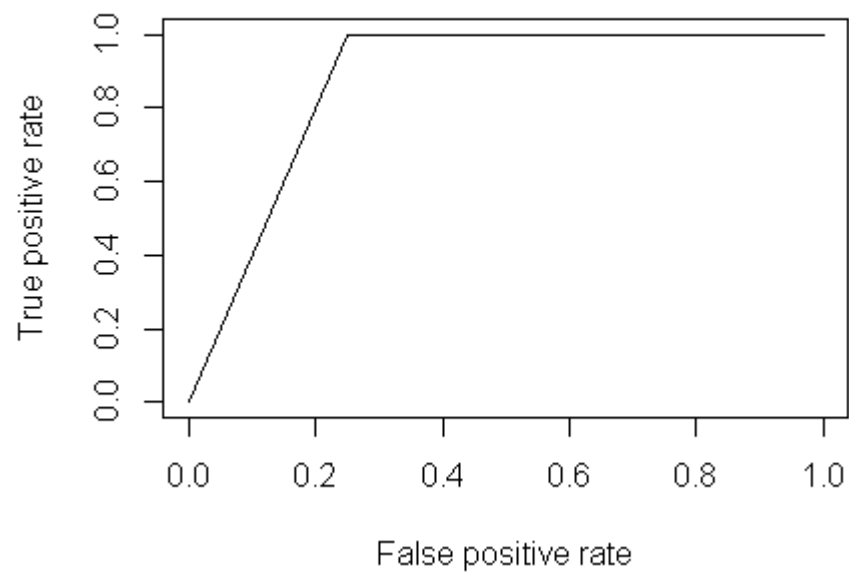
```
auc <- auc@y.values[[1]]
```

```
auc
```

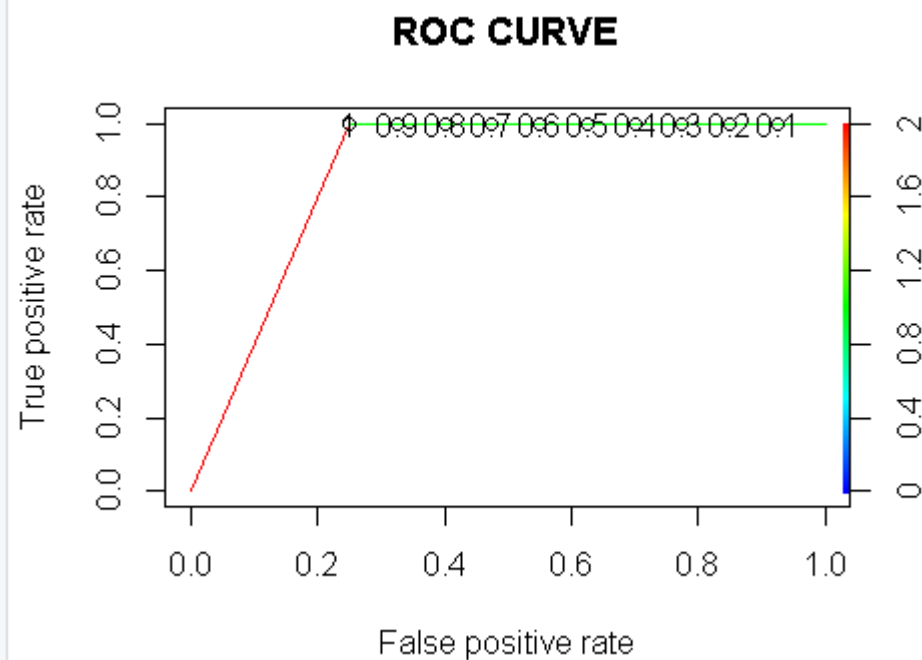
```
[1] 0.875
```

```
# Plotting curve
```

```
plot(ROCPer)
```



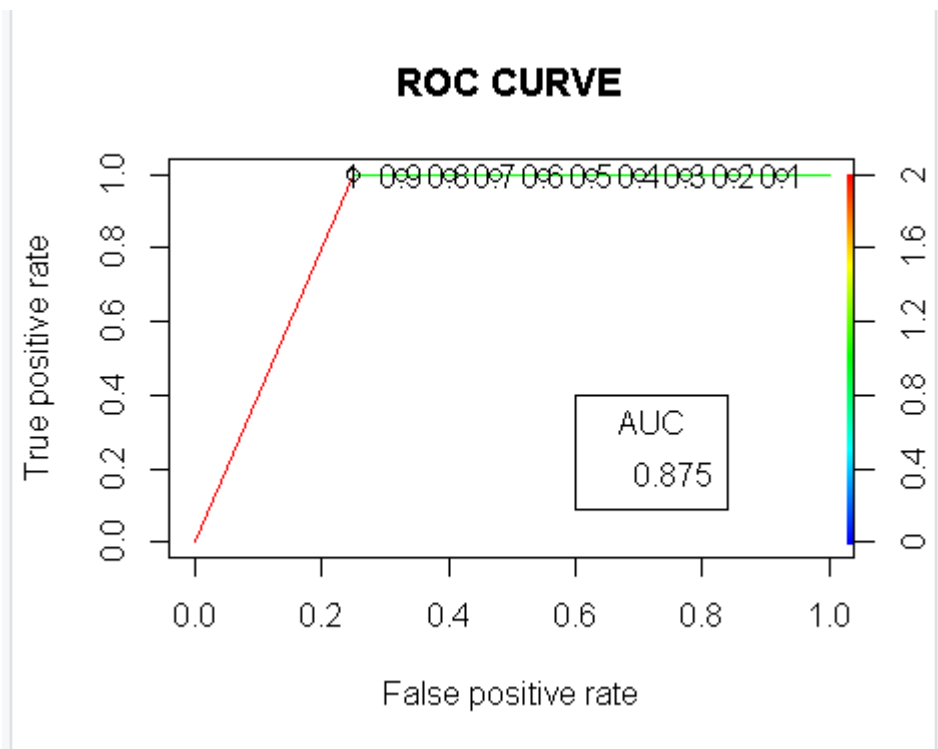
```
plot(ROCPer, colorize = TRUE,
     print.cutoffs.at = seq(0.1, by = 0.1),
     main = "ROC CURVE")
```



```
abline(a = 0, b = 1)
```

```
auc <- round(auc, 4)
```

```
legend(.6, .4, auc, title = "AUC", cex = 1)
```



```
###Simple/Multiple Linear Regression###
```

```
#####CODE FOR SIMPLE LINEAR REGRESSION###
```

```
#Upload the data from csv
```

```
input = read.csv("G:/TYCS DATA SCIENCE SEM 6/Data Sets/ageheight.csv")
```

```
#Create the linear regression
```

```
lmHeight = lm(height~age,data=input)
```

```
#get the summary
```

```
summary(lmHeight)
```

```
Call:
lm(formula = height ~ age, data = input)

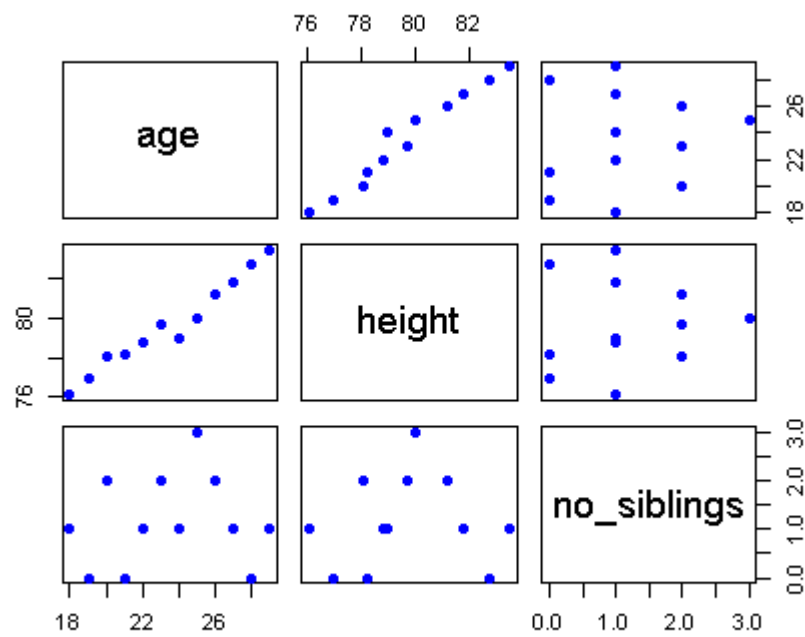
Residuals:
    Min       1Q   Median       3Q      Max
-0.99347 -0.08368  0.05723  0.32576  0.58765

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  65.10676    0.91839   70.89 7.61e-15 ***
age           0.62028    0.03867   16.04 1.83e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4624 on 10 degrees of freedom
Multiple R-squared:  0.9626,    Adjusted R-squared:  0.9589
F-statistic: 257.4 on 1 and 10 DF,  p-value: 1.83e-08
```

```
#Plot
```

```
plot(input,pch=16,col="blue")
```

#predict the value

```
lmHeight=lm(height~age,data=input)
```

```
> a = data.frame(age=20.5)
```

```
>result = predict(lmHeight,a)
```

```
>print(result)
```

```
1
77.82249
```

Result:

```
Call:
lm(formula = height ~ age, data = input)

Residuals:
    Min       1Q   Median       3Q      Max
-66.059   5.193   5.810   6.617   7.533

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  64.8445    43.3815   1.495   0.166
age           0.3835     1.8264   0.210   0.838

Residual standard error: 21.84 on 10 degrees of freedom
Multiple R-squared:  0.00439,    Adjusted R-squared:  -0.09517
F-statistic: 0.0441 on 1 and 10 DF,  p-value: 0.8379
```

#####CODE FOR MULTIPLE LINEAR REGRESSION###

```
>input = read.csv("G:/TYCS DATA SCIENCE SEM 6/Data Sets/ageheight.csv")
```

```
>lmHeight = lm(height~age + no_sibling,data=input)
```

```
>summary(lmHeight)
```

Result:

```
Call:
lm(formula = height ~ age + no_siblings, data = input)

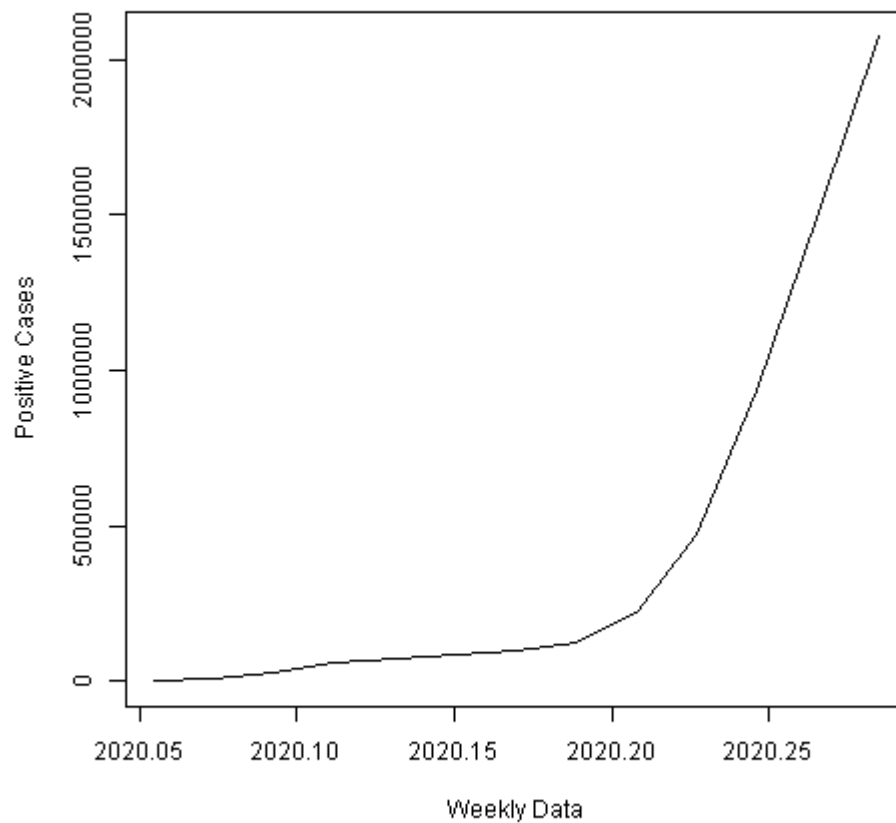
Residuals:
    Min       1Q   Median       3Q      Max
-65.795   4.334   5.484   6.924   9.104

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  64.1008    45.7871   1.400   0.195
age           0.3427     1.9323   0.177   0.863
no_siblings   1.4601     7.4320   0.196   0.849

Residual standard error: 22.97 on 9 degrees of freedom
Multiple R-squared:  0.008642, Adjusted R-squared:  -0.2117
F-statistic: 0.03923 on 2 and 9 DF,  p-value: 0.9617
```

```
###CODE FOR UNIVARIATE LINEAR REGRESSION###  
  
#weekly data as an input  
  
>x = c(580,7813,28266,59287,75700,87820,95314,126214,  
218843,471497,936851,1508725,2072113)  
  
#load package  
  
> library(lubridate)  
  
#create a file  
  
>png(file="TimeSeriesCovid.png")  
  
#create a time series object  
  
mts = ts(x,start=decimal_date(ymd("2020-01-21")),frequency=365.25/7)  
  
>plot(mts,xlab="Weekly Data",ylab="Positive Cases",main="COVID-19  
Pademic",col.main="darkgreen")  
  
> dev.off()
```

COVID-19 Pademic



```
###CODE FOR MULTIVARIATE LINEAR REGRESSION###
```

```
> positivecases =  
c(580,7813,28266,59287,75700,87820,95314,126214,218843,471497,936851,1508725,207  
2113)
```

```
>  
recoveredcases=c(17,270,565,1261,2126,2800,3285,4628,8951,21283,47210,88480,138475  
)
```

```
> library(lubridate)
```

```
> png(file="Multivariate14.png")
```

```
> mts=ts(cbind(positivecases,recoveredcases),start=decimal_date
```

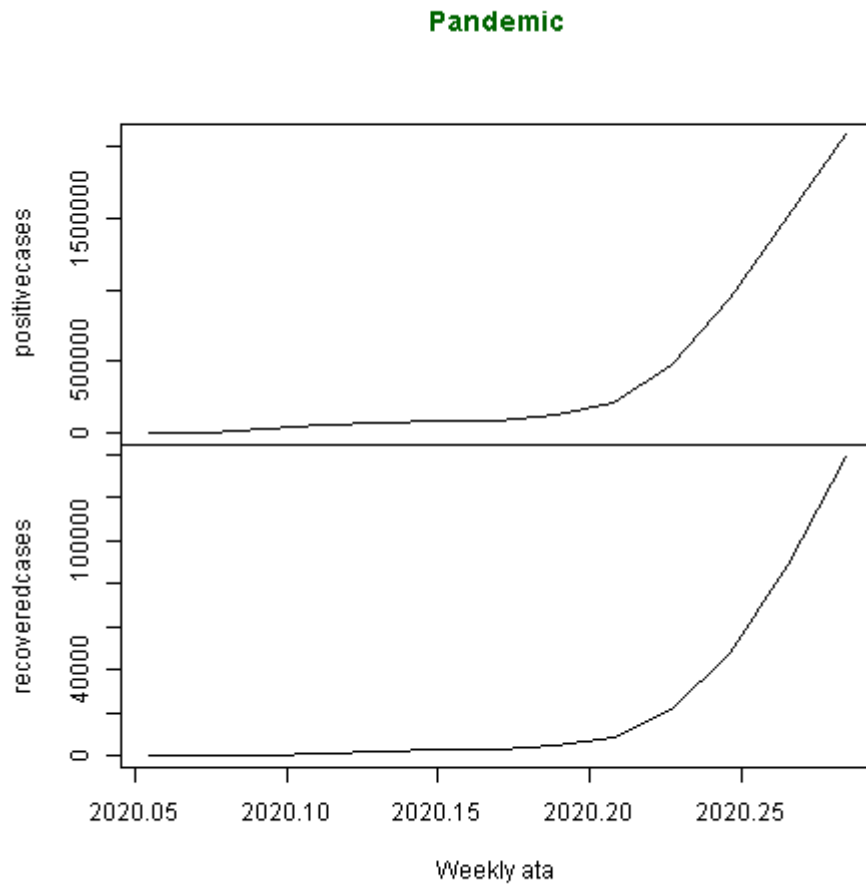
```

+ (ymd("2020-01-21")),frequency = 365.25/7)

> plot(mts,xlab="Weekly ata",main="Pandemic",col.main="darkGreen")

> dev.off()

```



```
#####CODE FOR FORECASTING#####
```

```

x <- c(580, 7813, 28266, 59287, 75700,87820, 95314, 126214, 218843,
      471497, 936851, 1508725, 2072113)

```

```
# library required for decimal_date() function
```

```
library(lubridate)
```

```
# library required for forecasting
```

```
library(forecast)
```

```
# output to be created as png file
```

```
png(file = "forecastTimeSeries.png")
```

```
# creating time series object
```

```
# from date 22 January, 2020
```

```
mts <- ts(x, start = decimal_date(ymd("2020-01-22")),  
         frequency = 365.25 / 7)
```

```
# forecasting model using arima model
```

```
fit <- auto.arima(mts)
```

```
# Next 5 forecasted values
```

```
forecast(fit, 5)
```

```
      Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95  
2020.307      2547989 2491957 2604020 2462296 2633682  
2020.326      2915130 2721277 3108983 2618657 3211603  
2020.345      3202354 2783402 3621307 2561622 3843087  
2020.364      3462692 2748533 4176851 2370480 4554904  
2020.383      3745054 2692884 4797225 2135898 5354210  
> |
```

```
# plotting the graph with next
```

```
# 5 weekly forecasted values
```

```
plot(forecast(fit, 5), xlab = "Weekly Data",
```

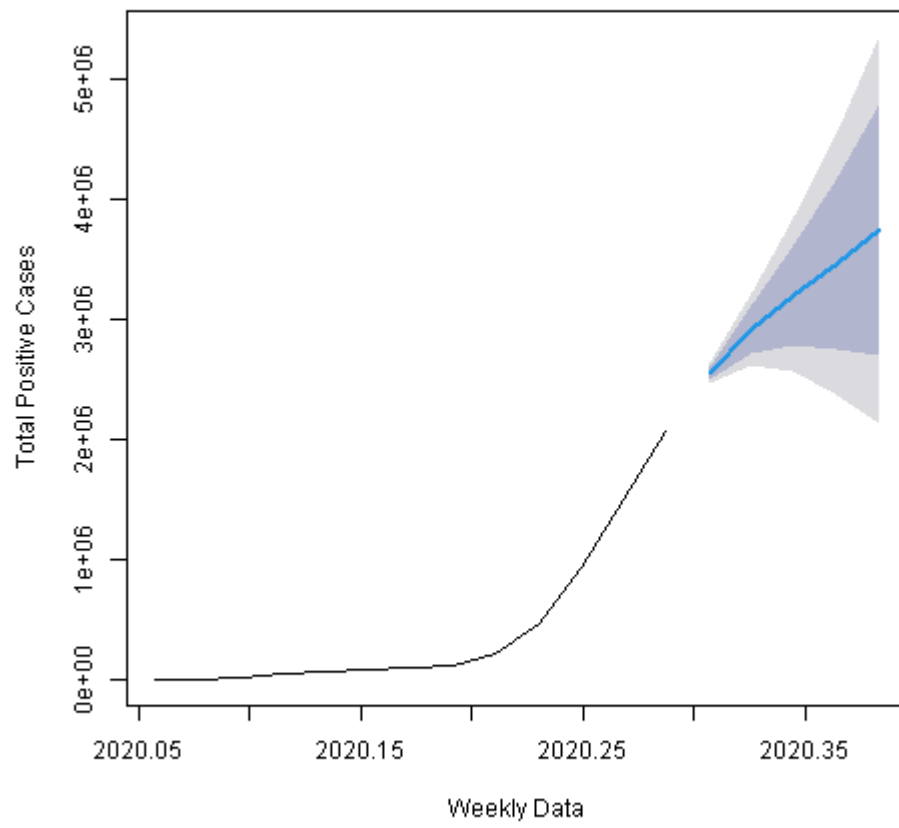
```
ylab = "Total Positive Cases",
```

```
main = "COVID-19 Pandemic", col.main = "darkgreen")
```

```
# saving the file
```

```
dev.off()
```

COVID-19 Pandemic



Ex. 2: Practical of Data collection, Data curation and management for Large-scale Data system (such as MongoDB)

Data curation is management of the data lifecycle. The term is typically applied to efforts to collect and preserve historical data. The following are common elements of data curation.

Overview: Data Curation	
Type	<u>Data</u>
Definition	Management of the data lifecycle especially in regards to historical data.
Related Concepts	Data Governance Data Management Data Profiling Data Backup Data Lineage Retention Schedule Compliance Data Quality Data Availability Data Disposal

1. MongoDB Create database
2. MongoDB Drop Database
3. MongoDB Create collection
4. MongoDB Drop collection
5. MongoDB Insert Document
6. MongoDB Query Document
7. MongoDB Update Document
8. Delete document in MongoDB
9. MongoDB Projection
10. limit() and skip() method in MongoDB
11. Sorting of Documents in MongoDB
12. MongoDB Indexing

Practical slip may contain experiment on above topics.


```
C:\>net start MongoDB
```

```
The MongoDB service is starting..  
The MongoDB service was started successfully.
```

```
C:\>mongo
```

To display the name of current database:

```
>db
```

```
test
```

```
>
```

Create Database in MongoDB

Once you are in the MongoDB shell, create the database in MongoDB by typing this command:

```
use database_name
```

For example: create a database “beginnersbook”:

```
use beginnersbook
```

```
> use beginnersbook
```

```
switched to db beginnersbook
```

```
>
```

Note: If the database name you mentioned is already present then this command will connect you to the database. However if the database doesn't exist then this will create the database with the given name and connect you to it.

At any point if you want to check the currently connected database just type the command **db**. This command will show the database name to which you are currently connected. This is really helpful command when you are working with several databases so that before creating a collection or inserting a document in database, you may want to ensure that you are in the right database.

```
> db
```

```
beginnersbook
```

```
>
```

To list down all the databases, use the command **show dbs**. This command lists down all the databases and their size on the disk.

```
> show dbs
```

```
EmployeeDB    0.000GB
```

```
Testdb        0.000GB
```

```
admin          0.000GB
```

```
beginnersbookdb 0.000GB
```

```
config         0.000GB
```

```
local      0.000GB
test       0.000GB
>
```

As you can see that the database “beginnersbook” that we have created is not present in the list of all the databases. This is because a database is not created until you save a document in it.

Now let us create a collection **user** and insert a document in it.

```
> db.user.insert({name: "Chaitanya", age: 30})
WriteResult({ "nInserted" : 1 })
> show dbs
EmployeeDB   0.000GB
Testdb       0.000GB
admin        0.000GB
beginnersbook 0.000GB
beginnersbookdb 0.000GB
config       0.000GB
local        0.000GB
test         0.000GB
>
```

Drop Database in MongoDB

We use `db.dropDatabase()` command to delete a database. You should be very careful when deleting a database as this will remove all the data inside that database, including collections and documents stored in the database.

MongoDB Drop Database

The syntax to drop a Database is:

```
db.dropDatabase()
```

We do not specify any database name in this command, because this command deletes the currently selected database. Lets see the steps to drop a database in MongoDB.

1. See the list of databases using **show dbs** command.

```
> show dbs
admin      0.000GB
beginnersbook 0.000GB
local      0.000GB
```

2. Switch to the database that needs to be dropped by typing this command.

```
use database_name
```

For example delete the database “beginnersbook”.

```
> use beginnersbook
switched to db beginnersbook
```

Note: Change the database name in the above command, from beginnersbook to the database that needs to be deleted.

3. Now, the currently selected database is beginnersbook so the command `db.dropDatabase()` would delete this database.

```
> db.dropDatabase()  
{ "dropped" : "beginnersbook", "ok" : 1 }
```

The command executed successfully and showing the operation “dropped” and status “ok” which means that the database is dropped.

```
> use beginnersbook  
switched to db beginnersbook
```

```
> db.dropDatabase()  
{ "dropped" : "beginnersbook", "ok" : 1 }
```

```
> show dbs  
EmployeeDB    0.000GB  
Testdb        0.000GB  
admin         0.000GB  
beginnersbookdb 0.000GB  
config        0.000GB  
local         0.000GB  
test          0.000GB  
>
```

Create Collection in MongoDB

We know that the data in MongoDB is stored in form of documents. These documents are stored in Collection and Collection is stored in Database.

Method 1: Creating the Collection in MongoDB on the fly

The cool thing about MongoDB is that you need not to create collection before you insert document in it. With a single command you can insert a document in the collection and the MongoDB creates that collection on the fly.

Syntax: **`db.collection_name.insert({key:value, key:value...})`**

For example:

We don't have a collection beginnersbook in the database beginnersbookdb. This command will create the collection named “beginnersbook” on the fly and insert a document in it with the specified key and value pairs.

```
> use beginnersbookdb  
switched to db beginnersbookdb  
> db.beginnersbook.insert({  
...  name: "Chaitanya",  
...  age: 30,  
...  website: "beginnersbook.com"  
... })
```

```
WriteResult({ "nInserted" : 1 })
>
```

To check whether the document is successfully inserted, type the following command. It shows all the documents in the given collection.

Syntax: **db.collection_name.find()**

```
> db.beginnersbook.find()
{ "_id" : ObjectId("5c281a2dc23e08d1515fd9ca"), "name" : "Chaitanya", "age" : 30,
  "website" : "beginnersbook.com" }
>
```

To check whether the collection is created successfully, use the following command.

```
show collections
```

This command shows the list of all the collections in the currently selected database.

```
> show collections
beginnersbook
students
>
```

Method 2: Creating collection with options before inserting the documents

We can also create collection before we actually insert data in it. This method provides you the options that you can set while creating a collection.

Syntax:

```
db.createCollection(name, options)
```

name is the collection name

and **options** is an optional field that we can use to specify certain parameters such as size, max number of documents etc. in the collection.

First lets see how this command is used for creating collection without any parameters:

```
> db.createCollection("students")
{
  "ok" : 0,
  "errmsg" : "a collection 'beginnersbookdb.students' already exists",
  "code" : 48,
  "codeName" : "NamespaceExists"
}

> db.createCollection("students11")
{ "ok" : 1 }
>
```

Lets see the options that we can provide while creating a collection:

capped: type: boolean.

This parameter takes only true and false. This specifies a cap on the max entries a collection can have. Once the collection reaches that limit, it starts overwriting old entries.

The point to note here is that when you set the capped option to true you also have to specify the size parameter.

size: type: number.

This specifies the max size of collection (capped collection) in bytes.

max: type: number.

This specifies the max number of documents a collection can hold.

autoIndexId: type: boolean

The default value of this parameter is false. If you set it true then it automatically creates index field `_id` for each document. We will learn about index in the MongoDB indexing tutorial.

Lets see an example of capped collection:

```
> db.createCollection("teachers", { capped : true, size : 9232768 } )
{ "ok" : 1 }
>
```

This command will create a collection named “teachers” with the max size of 9232768 bytes. Once this collection reaches that limit it will start overwriting old entries.

```
> show collections
beginnersbook
students
students11
teachers
>
```

Drop collection in MongoDB

To drop a collection , first connect to the database in which you want to delete collection and then type the following command to delete the collection:

```
db.collection_name.drop()
```

Note: Once you drop a collection all the documents and the indexes associated with them will also be dropped. To preserve the indexes we use `remove()` function that only removes the documents in the collection but doesn't remove the collection itself and the indexes created on it.

MongoDB drop collection Example

For example delete a collection names "teachers" in my database "beginnersbookdb". Write the following commands in the given sequence.

```
> use beginnersbookdb
switched to db beginnersbookdb
> show collections
beginnersbook
students
students11
teachers
> db.teachers.drop()
true
> show collections
beginnersbook
students
students11
>
```

As you can see that the command `db.teachers.drop()` returned true which means that the collection is deleted successfully.

MongoDB Insert Document

Syntax to insert a document into the collection:

```
db.collection_name.insert()
```

Lets take an example to understand this .

MongoDB Insert Document using insert() Example

Here we are inserting a document into the collection named "beginnersbook". The field "course" in the example below is an array that holds the several key-value pairs.

You should see a successful write message like this:

```
WriteResult({ "nInserted" : 1 })
```

The `insert()` method creates the collection if it doesn't exist but if the collection is present then it inserts the document into it

```

> db.beginnersbook.insert(
...   {
...     name: "Chaitanya",
...     age: 30,
...     email: "admin@beginnersbook.com",
...     course: [ { name: "MongoDB", duration: 7 }, { name: "Java", duration: 30 } ]
...   }
... )
WriteResult({ "nInserted" : 1 })
>

```

Verification:

You can also verify whether the document is successfully inserted by typing following command:

```
db.collection_name.find()
```

In the above example, we inserted the document in the collection named “beginnersbook” so the command should be:

```

> db.beginnersbook.find()
{ "_id" : ObjectId("5c2d37734fa204bd77e7fc1c"), "name" : "Chaitanya", "age" : 30, "email" :
"admin@beginnersbook.com", "course" : [ { "name" : "MongoDB", "duration" : 7 }, { "name"
: "Java", "duration" : 30 } ] }
>

```

MongoDB Example: Insert Multiple Documents in collection

To insert multiple documents in collection, we define an array of documents and later we use the insert() method on the array variable as shown in the example below. Here we are inserting three documents in the collection named “students”. This command will insert the data in “students” collection, if the collection is not present then it will create the collection and insert these documents.

```

> var beginners =
... [
...   {
...     "StudentId" : 1001,
...     "StudentName" : "Steve",
...     "age": 30
...   },
...   {
...     "StudentId" : 1002,
...     "StudentName" : "Negan",
...     "age": 42
...   },
...   {

```

```

... "StudentId" : 3333,
... "StudentName" : "Rick",
...   "age": 35
... },
... ];
> db.students.insert(beginners);

```

Output:

```

BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 3,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
>

```

As you can see that it shows number 3 in front of **nInserted**. this means that the 3 documents have been inserted by this command.

```

> db.students.find()
{ "_id" : ObjectId("5c281c90c23e08d1515fd9cc"), "StudentId" : 1001, "StudentName" :
"Steve", "age" : 30 }
{ "_id" : ObjectId("5c281c90c23e08d1515fd9cd"), "StudentId" : 1002, "StudentName" :
"Negan", "age" : 42 }
{ "_id" : ObjectId("5c281c90c23e08d1515fd9ce"), "StudentId" : 3333, "StudentName" :
"Rick", "age" : 35 }
{ "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"), "StudentId" : 1001, "StudentName" :
"Steve", "age" : 30 }
{ "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"), "StudentId" : 1002, "StudentName" :
"Negan", "age" : 42 }
{ "_id" : ObjectId("5c2d38934fa204bd77e7fc1f"), "StudentId" : 3333, "StudentName" :
"Rick", "age" : 35 }
>

```

You can print the output data in a JSON format so that you can read it easily. To print the data in JSON format run the command **db.collection_name.find().forEach(printjson)**

So in our case the command would be this:

```

> db.students.find().forEach(printjson)
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
  "StudentId" : 1001,
  "StudentName" : "Steve",

```



```

    "age" : 30
  }
  {
    "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
    "StudentId" : 1002,
    "StudentName" : "Negan",
    "age" : 42
  }
  {
    "_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
    "StudentId" : 3333,
    "StudentName" : "Rick",
    "age" : 35
  }
  {
    "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
    "StudentId" : 1001,
    "StudentName" : "Steve",
    "age" : 30
  }
  {
    "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
    "StudentId" : 1002,
    "StudentName" : "Negan",
    "age" : 42
  }
  {
    "_id" : ObjectId("5c2d38934fa204bd77e7fc1f"),
    "StudentId" : 3333,
    "StudentName" : "Rick",
    "age" : 35
  }
}
>

```

MongoDB Query Document using find() method

Querying all the documents in JSON format

Lets say we have a collection `students` in a database named `beginnersbookdb`. To get all the documents we use this command:

```
db.students.find()
```

However the output we get is not in any format and less-readable. To improve the readability, we can format the output in JSON format with this command:

```
db.students.find().forEach(printjson);
```

OR simply use `pretty()` – It does the same thing.

```

> db.students.find().pretty()
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
  "StudentId" : 1002,
  "StudentName" : "Negan",
  "age" : 42
}
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
  "StudentId" : 3333,
  "StudentName" : "Rick",
  "age" : 35
}
{
  "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
  "StudentId" : 1002,
  "StudentName" : "Negan",
  "age" : 42
}
{
  "_id" : ObjectId("5c2d38934fa204bd77e7fc1f"),
  "StudentId" : 3333,
  "StudentName" : "Rick",
  "age" : 35
}

```

Query Document based on the criteria

Instead of fetching all the documents from collection, we can fetch selected documents based on a criteria.

Equality Criteria:

For example: To fetch the data of "Steve" from students collection. The command for this should be:

```
> db.students.find({StudentName : "Steve"}).pretty()
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
>
```

Greater Than Criteria:

Syntax:

```
db.collection_name.find({"field_name":{$gt:criteria_value}}).pretty()
```

For example: To fetch the details of students having age > 32 then the query should be:

```
> db.students.find({"age":{$gt:32}}).pretty()
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
  "StudentId" : 1002,
  "StudentName" : "Negan",
  "age" : 42
}
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
  "StudentId" : 3333,
  "StudentName" : "Rick",
  "age" : 35
}
>
```

Less than Criteria:

Syntax:

```
db.collection_name.find({"field_name":{$lt:criteria_value}}).pretty()
```

Example: Find all the students having id less than 3000. The command for this criteria would be:

```
> db.students.find({"StudentId":{$lt:3000}}).pretty()
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
```

```
"StudentId" : 1002,
"StudentName" : "Negan",
"age" : 42
}
>
```

Not Equals Criteria:

Syntax:

```
db.collection_name.find({"field_name":{"$ne:criteria_value"}}).pretty()
```

Example: Find all the students where id is not equal to 1002. The command for this criteria would be:

```
> db.students.find({"StudentId":{"$ne:1002"}}).pretty()
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
  "StudentId" : 3333,
  "StudentName" : "Rick",
  "age" : 35
}
>
```

Greater than equals Criteria:

```
db.collection_name.find({"field_name":{"$gte:criteria_value"}}).pretty()
```

Less than equals Criteria:

```
db.collection_name.find({"field_name":{"$lte:criteria_value"}}).pretty()
```

The pretty() method that we have added at the end of all the commands is not mandatory. It is just used for formatting purposes.

MongoDB – Update Document in a Collection

In MongoDB, we have two ways to update a document in a collection. 1) update() method 2) save() method. Although both the methods update an existing document, they are being used in different scenarios. The update() method is used when we need to update the values of an existing document while save() method is used to replace the existing document with the document that has been passed in it.

To update a document in MongoDB, we provide a criteria in command and the document that matches that criteria is updated.

Updating Document using update() method

Syntax:

```
db.collection_name.update(criteria, update_data)
```

Example:

For example: Use a collection named "got" in the database "beginnersbookdb". The documents inside "got" are:

```
> use beginnersbookdb
switched to db beginnersbookdb
> show collections
beginnersbook
students
students11
> db.createCollection("got")
{ "ok" : 1 }
>

> var abc = [
... {
...   "_id" : ObjectId("59bd2e73ce524b733f14dd65"),
...   "name" : "Jon Snow",
...   "age" : 32
... },
... {
...   "_id" : ObjectId("59bd2e8bce524b733f14dd66"),
...   "name" : "Khal Drogo",
...   "age" : 36
... },
... {
...   "_id" : ObjectId("59bd2e9fce524b733f14dd67"),
...   "name" : "Sansa Stark",
...   "age" : 20
... },
... {
...   "_id" : ObjectId("59bd2ec5ce524b733f14dd68"),
...   "name" : "Lord Varys",
...   "age" : 42
... },
... ];
> db.got.insert(abc);
BulkWriteResult({
```

```

    "writeErrors" : [ ],
    "writeConcernErrors" : [ ],
    "nInserted" : 4,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
  })

> db.got.find().pretty()
{
  "_id" : ObjectId("59bd2e73ce524b733f14dd65"),
  "name" : "Jon Snow",
  "age" : 32
}
{
  "_id" : ObjectId("59bd2e8bce524b733f14dd66"),
  "name" : "Khal Drogo",
  "age" : 36
}
{
  "_id" : ObjectId("59bd2e9fce524b733f14dd67"),
  "name" : "Sansa Stark",
  "age" : 20
}
{
  "_id" : ObjectId("59bd2ec5ce524b733f14dd68"),
  "name" : "Lord Varys",
  "age" : 42
}
>

```

To update the name of Jon Snow with the name “Kit Harington”. The command for this would be:

```
db.got.update({"name":"Jon Snow"},{$set:{"name":"Kit Harington"}})
```

```

> db.got.update({"name":"Jon Snow"},{$set:{"name":"Kit Harington"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.got.find().pretty()
{
  "_id" : ObjectId("59bd2e73ce524b733f14dd65"),
  "name" : "Kit Harington",
  "age" : 32
}
{

```

```

    "_id" : ObjectId("59bd2e8bce524b733f14dd66"),
    "name" : "Khal Drogo",
    "age" : 36
  }
  {
    "_id" : ObjectId("59bd2e9fce524b733f14dd67"),
    "name" : "Sansa Stark",
    "age" : 20
  }
  {
    "_id" : ObjectId("59bd2ec5ce524b733f14dd68"),
    "name" : "Lord Varys",
    "age" : 42
  }
}
>

```

By default the update method updates a single document. In the above example we had only one document matching with the criteria, however if there were more then also only one document would have been updated. To enable update() method to update multiple documents you have to set “multi” parameter of this method to true as shown below.

To update multiple documents with the update() method:

```

db.got.update({"name":"Jon Snow"},
{$set:{"name":"Kit Harington"}},{multi:true})

```

Updating Document using save() method

Syntax:

```

db.collection_name.save( {_id:ObjectId(), new_document} )

```

Lets take the same example that we have seen above. Now we want to update the name of “Kit Harington” to “Jon Snow”. To work with save() method you should know the unique _id field of that document.

A very important **point to note** is that when you do not provide the _id field while using save() method, it calls insert() method and the passed document is inserted into the collection as a new document

To get the _id of a document, you can either type this command:

```

db.got.find().pretty()

```

Here got is a collection name. This method of finding unique _id is only useful when you have few documents, otherwise scrolling and searching for that _id in huge number of documents is tedious.

If you have huge number of documents then, to get the _id of a particular document use the criteria in find() method. For example: To get the _id of the document that we want to update using save() method, type this command:

```
> db.got.find({"name": "Kit Harington"}).pretty()
{
  "_id" : ObjectId("59bd2e73ce524b733f14dd65"),
  "name" : "Kit Harington",
  "age" : 32
}
> db.got.save({"_id" : ObjectId("59bd2e73ce524b733f14dd65"), "name":
... "Jon Snow", "age": 30})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.got.find().pretty()
{
  "_id" : ObjectId("59bd2e73ce524b733f14dd65"),
  "name" : "Jon Snow",
  "age" : 30
}
{
  "_id" : ObjectId("59bd2e8bce524b733f14dd66"),
  "name" : "Khal Drogo",
  "age" : 36
}
{
  "_id" : ObjectId("59bd2e9fce524b733f14dd67"),
  "name" : "Sansa Stark",
  "age" : 20
}
{
  "_id" : ObjectId("59bd2ec5ce524b733f14dd68"),
  "name" : "Lord Varys",
  "age" : 42
}
>
```

MongoDB Delete Document from a Collection

In this tutorial we will learn how to delete documents from a collection. The remove() method is used for removing the documents from a collection in MongoDB.

Syntax of remove() method:


```
db.collection_name.remove(delete_criteria)
```

Delete Document using remove() method

Lets use a collection students in my MongoDB database named beginnersbookdb. The documents in students collection are:

```
> db.students.find().pretty()
{
  "_id" : ObjectId("59bcecc7668dcce02aaa6fed"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("59bcecc7668dcce02aaa6fef"),
  "StudentId" : 3333,
  "StudentName" : "Rick",
  "age" : 35
}
```

To remove the student from this collection who has a student id equal to 3333. Write a command using remove() method like this:

```
db.students.remove({"StudentId": 3333})
```

Output:

```
WriteResult({ "nRemoved" : 1 })
```

To verify whether the document is actually deleted. Type the following command:

```
db.students.find().pretty()
```

It will list all the documents of students collection.

```
> use beginnersbookdb
switched to db beginnersbookdb
> db.students.find().pretty()
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
```

```

    "StudentId" : 1002,
    "StudentName" : "Negan",
    "age" : 42
  }
  {
    "_id" : ObjectId("5c281c90c23e08d1515fd9ce"),
    "StudentId" : 3333,
    "StudentName" : "Rick",
    "age" : 35
  }
  {
    "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
    "StudentId" : 1001,
    "StudentName" : "Steve",
    "age" : 30
  }
  {
    "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
    "StudentId" : 1002,
    "StudentName" : "Negan",
    "age" : 42
  }
  {
    "_id" : ObjectId("5c2d38934fa204bd77e7fc1f"),
    "StudentId" : 3333,
    "StudentName" : "Rick",
    "age" : 35
  }
}

```

```

> db.students.remove({"StudentId": 3333})
WriteResult({ "nRemoved" : 2 })

```

```

> db.students.find().pretty()
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
  "StudentId" : 1002,
  "StudentName" : "Negan",
  "age" : 42
}
{
  "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),

```

```

    "StudentId" : 1001,
    "StudentName" : "Steve",
    "age" : 30
  }
  {
    "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
    "StudentId" : 1002,
    "StudentName" : "Negan",
    "age" : 42
  }
>

```

How to remove only one document matching your criteria?

When there are more than one documents present in collection that matches the criteria then all those documents will be deleted if you run the remove command. However there is a way to limit the deletion to only one document so that even if there are more documents matching the deletion criteria, only one document will be deleted.

```
db.collection_name.remove(delete_criteria, justOne)
```

Here justOne is a Boolean parameter that takes only 1 and 0, if you give 1 then it will limit the the document deletion to only 1 document. This is an optional parameters as we have seen above that we have used the remove() method without using this parameter.

Example:

```
db.walkingdead.remove({"age": 32}, 1)
```

Remove all Documents

If you want to remove all the documents from a collection but does not want to remove the collection itself then you can use remove() method like this:

```
db.collection_name.remove({})
```

MongoDB Projection

This is used when we want to get the selected fields of the documents rather than all fields.

For example, we have a collection where we have stored documents that have the fields: StudentId, StudentName, age but we want to see only the StudentId of all the students then in that case we can use projection to get only the StudentId.

Syntax:

```
db.collection_name.find({}, {field_key: 1 or 0})
```

```

> db.students.find().pretty()
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cc"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("5c281c90c23e08d1515fd9cd"),
  "StudentId" : 1002,
  "StudentName" : "Negan",
  "age" : 42
}
{
  "_id" : ObjectId("5c2d38934fa204bd77e7fc1d"),
  "StudentId" : 1001,
  "StudentName" : "Steve",
  "age" : 30
}
{
  "_id" : ObjectId("5c2d38934fa204bd77e7fc1e"),
  "StudentId" : 1002,
  "StudentName" : "Negan",
  "age" : 42
}
> db.students.find({}, {"_id": 0, "StudentId" : 1})
{ "StudentId" : 1001 }
{ "StudentId" : 1002 }
{ "StudentId" : 1001 }
{ "StudentId" : 1002 }
>

```

Value 1 means show that field and 0 means do not show that field. When we set a field to 1 in Projection other fields are automatically set to 0, except `_id`, so to avoid the `_id` we need to specifically set it to 0 in projection. The vice versa is also true when we set few fields to 0, other fields set to 1 automatically.

```

> db.students.find({}, {"_id": 0, "StudentName" : 0, "age" : 0})
{ "StudentId" : 1001 }
{ "StudentId" : 1002 }
{ "StudentId" : 1001 }
{ "StudentId" : 1002 }
>

```

```

> db.students.find({}, {"_id": 0, "StudentName" : 0, "age" : 1})
Error: error: {

```

```

    "ok" : 0,
    "errmsg" : "Projection cannot have a mix of inclusion and exclusion.",
    "code" : 2,
    "codeName" : "BadValue"
  }
>

```

This is because we have set student_name to 0 and other field student_age to 1. We can't mix these. You either set those fields that you don't want to display to 0 or set the fields to 1 that you want to display.

MongoDB – limit() and skip() method

The limit() method in MongoDB

This method limits the number of documents returned in response to a particular query.

Syntax:

```

db.collection_name.find().limit(number_of_documents)
db.studentdata.find({student_id : {$gt:2002}}).pretty()
db.studentdata.find({student_id : {$gt:2002}}).limit(1).pretty()

```

MongoDB Skip() Method

The skip() method is used for skipping the given number of documents in the Query result.

To understand the use of skip() method, let's take the same example that we have seen above. In the above example we can see that by using limit(1) we managed to get only one document, which is the first document that matched the given criteria. What if you do not want the first document matching your criteria. For example we have two documents that have student_id value greater than 2002 but when we limited the result to 1 by using limit(1), we got the first document, in order to get the second document matching this criteria we can use skip(1) here which will skip the first document.

```

db.studentdata.find({student_id : {$gt:2002}}).limit(1).skip(1).pretty()

```

MongoDB sort() method

Sorting Documents using sort() method

Using sort() method, you can sort the documents in ascending or descending order based on a particular field of document.

Syntax of sort() method:

```

db.collection_name.find().sort({field_key:1 or -1})

```

1 is for ascending order and -1 is for descending order. The default value is 1.

For example: collection `studentdata` contains following documents:

```
> db.studentdata.find().pretty()
{
  "_id" : ObjectId("59bf63380be1d7770c3982af"),
  "student_name" : "Steve",
  "student_id" : 2002,
  "student_age" : 22
}
{
  "_id" : ObjectId("59bf63500be1d7770c3982b0"),
  "student_name" : "Carol",
  "student_id" : 2003,
  "student_age" : 22
}
{
  "_id" : ObjectId("59bf63650be1d7770c3982b1"),
  "student_name" : "Tim",
  "student_id" : 2004,
  "student_age" : 23
}
```

Let's display the `student_id` of all the documents in **descending order**:

```
> db.studentdata.find({}, {"student_id": 1, _id:0}).sort({"student_id": -1})
{ "student_id" : 2004 }
{ "student_id" : 2003 }
{ "student_id" : 2002 }
```

To display the `student_id` field of all the students in **ascending order**:

```
> db.studentdata.find({}, {"student_id": 1, _id:0}).sort({"student_id": 1})
{ "student_id" : 2002 }
{ "student_id" : 2003 }
{ "student_id" : 2004 }
```

Default: The default is ascending order so if any value is not provided in the `sort()` method then it will sort the records in ascending order as shown below:

```
> db.studentdata.find({}, {"student_id": 1, _id:0}).sort({})
{ "student_id" : 2002 }
{ "student_id" : 2003 }
{ "student_id" : 2004 }
```

You can also sort the documents based on the field that you don't want to display: For example, you can sort the documents based on `student_id` and display the `student_age` and `student_name` fields.

```
> db.studentdata.find({}, {"student_id": 0, _id:0}).sort({"student_id": 1})
{ "student_name" : "Steve", "student_age" : 22 }
{ "student_name" : "Carol", "student_age" : 22 }
{ "student_name" : "Tim", "student_age" : 23 }
>
```

MongoDB Indexing Tutorial with Example

An **index** in MongoDB is a special data structure that holds the data of few fields of documents on which the index is created. Indexes improve the speed of search operations in database because instead of searching the whole document, the search is performed on the indexes that hold only few fields. On the other hand, having too many indexes can hamper the performance of insert, update and delete operations because of the additional write and additional data space used by indexes.

How to create index in MongoDB

```
db.collection_name.createIndex({field_name: 1 or -1})
```

The value 1 is for ascending order and -1 is for descending order.

For example, use collection `studentdata`. The documents inside this collection have following fields:

`student_name`, `student_id` and `student_age`

Let's create the index on `student_name` field in ascending order:

```
db.studentdata.createIndex({student_name: 1})
```

Output:

```
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

```
> db.studentdata.createIndex({student_name: 1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

Command is Successful

Number of indexes after command execution (`_id` and the one we have created)

Number of indexes before the command is executed (`_id`)

We have created the index on `student_name` which means when someone searches the document based on the `student_name`, the search will be faster because the index will be used for this search. So this is important to create the index on the field that will be frequently searched in a collection.

MongoDB – Finding the indexes in a collection

We can use `getIndexes()` method to find all the indexes created on a collection. The syntax for this method is:

```
db.collection_name.getIndexes()
```

So to get the indexes of `studentdata` collection, the command would be:

```
> db.studentdata.getIndexes()
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "test.studentdata"
  },
  {
    "v" : 2,
    "key" : {
      "student_name" : 1
    },
    "name" : "student_name_1",
    "ns" : "test.studentdata"
  }
]
```

The output shows that we have two indexes in this collection. The default index created on `_id` and the index that we have created on `student_name` field.

MongoDB – Drop indexes in a collection

You can either drop a particular index or all the indexes.

Dropping a specific index:

For this purpose the `dropIndex()` method is used.

```
db.collection_name.dropIndex({index_name: 1})
```

Lets drop the index that we have created on `student_name` field in the collection `studentdata`. The command for this:

```
db.studentdata.dropIndex({student_name: 1})
```

```
> db.studentdata.dropIndex({student_name: 1})
{ "nIndexesWas" : 2, "ok" : 1 }
>
```

`nIndexesWas`: It shows how many indexes were there before this command got executed
`ok: 1`: This means the command is executed successfully.

Dropping all the indexes:

To drop all the indexes of a collection, we use `dropIndexes()` method.

Syntax of `dropIndexes()` method:

```
db.collection_name.dropIndexes()
```

Lets say we want to drop all the indexes of `studentdata` collection.

```
db.studentdata.dropIndexes()
```

```
> db.studentdata.dropIndexes()
{
  "nIndexesWas" : 1,
  "msg" : "non-_id indexes dropped for collection",
  "ok" : 1
}
>
```

The message “non-_id indexes dropped for collection” indicates that the default index `_id` will still remain and cannot be dropped. This means that using this method we can only drop indexes that we have created, we can’t drop the default index created on `_id` field.