# Simulation of Apriori Algorithm on Groceries Dataset

Ashishkumar Rana 09

2022-07-21

## Simulation of Apriori Algorithm on Groceries Dataset

---

```
#install.packages("arules")
#install.packages("arulesViz")
```

---

```
library(arules)
```

```
## Warning: package 'arules' was built under R version 4.2.1

## Loading required package: Matrix

##
## Attaching package: 'arules'

## The following objects are masked from 'package:base':
##
##     abbreviate, write
```

```
library(arulesViz)
```

```
## Warning: package 'arulesViz' was built under R version 4.2.1
```

- In R Programming language for the application of Apriori Algorithms arules is used.
- This library provides inbuilt datset and also many function that are suitable for Apriori processes.
- ArulesViz is another libaray that is used for visualization of apriori model.

```
data("Groceries")
```

```
Groceries
```

```
## transactions in sparse format with
##  9835 transactions (rows) and
##  169 items (columns)
```

- The dataset that we use in this practical is a groceries dataset.
- This dataset is provided by the arules package of R programming langauge. This dataset is collected from 30 days of transaction of a groceries store.
- The dataset contain 9835 transactions and the item are aggregated into 169 categories.
- Summary of above dataset will give the information about most frequent item.

```
summary(Groceries)

## transactions as itemMatrix in sparse format with
##  9835 rows (elements/itemsets/transactions) and
##  169 columns (items) and a density of 0.02609146
##
## most frequent items:
##       whole milk other vegetables       rolls/buns             soda
##             2513             1903             1809             1715
##           yogurt          (Other)
##             1372            34055
##
## element (itemset/transaction) length distribution:
## sizes
##    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
16
## 2159 1643 1299 1005  855  645  545  438  350  246  182  117   78   77   55
46
##   17   18   19   20   21   22   23   24   26   27   28   29   32
##   29   14   14    9   11    4    6    1    1    1    1    3    1
##
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   2.000   3.000   4.409   6.000  32.000
##
## includes extended item information - examples:
##        labels  level2           level1
## 1 frankfurter sausage meat and sausage
## 2     sausage sausage meat and sausage
## 3  liver loaf sausage meat and sausage
```

## Frequent itemsets generation

```
itemsets=apriori(Groceries,
              parameter = list(minlen = 1, maxlen = 1, support = 0.02, target
= "frequent itemsets"))
```

```
## Apriori
##
```

```
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##          NA    0.1    1 none FALSE            TRUE       5    0.02      1
##  maxlen            target  ext
##       1 frequent itemsets TRUE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 196
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [59 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1
```

```
## Warning in apriori(Groceries, parameter = list(minlen = 1, maxlen = 1, sup
port
## = 0.02, : Mining stopped (maxlen reached). Only patterns up to a length of
1
## returned!
```

```
##  done [0.00s].
## sorting transactions ... done [0.00s].
## writing ... [59 set(s)] done [0.00s].
## creating S4 object  ... done [0.00s].
```

```
inspect(head(sort(itemsets, by = "support"), 10))
```

```
##      items              support    count
## [1]  {whole milk}       0.25551601 2513
## [2]  {other vegetables} 0.19349263 1903
## [3]  {rolls/buns}       0.18393493 1809
## [4]  {soda}             0.17437722 1715
## [5]  {yogurt}           0.13950178 1372
## [6]  {bottled water}    0.11052364 1087
## [7]  {root vegetables}  0.10899847 1072
## [8]  {tropical fruit}   0.10493137 1032
## [9]  {shopping bags}    0.09852567  969
## [10] {sausage}          0.09395018  924
```

```
itemsets=apriori(Groceries,
              parameter = list(minlen = 1, maxlen = 1, support = 0.02, target
= "frequent itemsets"))
```

```
## Apriori
##
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
```

```
##          NA    0.1    1 none FALSE            TRUE       5    0.02      1
##  maxlen            target  ext
##       1 frequent itemsets TRUE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 196
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [59 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1

## Warning in apriori(Groceries, parameter = list(minlen = 1, maxlen = 1, sup
port
## = 0.02, : Mining stopped (maxlen reached). Only patterns up to a length of
1
## returned!

##  done [0.00s].
## sorting transactions ... done [0.00s].
## writing ... [59 set(s)] done [0.00s].
## creating S4 object  ... done [0.00s].
```

```
inspect(head(sort(itemsets, by = "support"), 10))
```

```
##      items              support    count
## [1]  {whole milk}       0.25551601 2513
## [2]  {other vegetables} 0.19349263 1903
## [3]  {rolls/buns}       0.18393493 1809
## [4]  {soda}             0.17437722 1715
## [5]  {yogurt}           0.13950178 1372
## [6]  {bottled water}    0.11052364 1087
## [7]  {root vegetables}  0.10899847 1072
## [8]  {tropical fruit}   0.10493137 1032
## [9]  {shopping bags}    0.09852567  969
## [10] {sausage}          0.09395018  924
```

- The apriori algorithm from the arules package impliments apriori algorithms to create frequent itemsets.
- In the parameter list we have set the support thresold to 0.02 which means that for an item to be considered as frequent it must appear atleast 198 times.
- Display the first top 10 frequent itemsets arranged in the descending order of support.

```
itemsets=apriori(Groceries,
              parameter = list(minlen = 2, maxlen = 2, support = 0.02, target
= "frequent itemsets"))
```

```
## Apriori
##
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##          NA    0.1    1 none FALSE            TRUE       5    0.02      2
##  maxlen            target  ext
##       2 frequent itemsets TRUE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 196
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [59 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2

## Warning in apriori(Groceries, parameter = list(minlen = 2, maxlen = 2, sup
port
## = 0.02, : Mining stopped (maxlen reached). Only patterns up to a length of
2
## returned!

##  done [0.00s].
## sorting transactions ... done [0.00s].
## writing ... [61 set(s)] done [0.00s].
## creating S4 object  ... done [0.00s].
```

```
inspect(head(sort(itemsets, by = "support"), 10))
```

```
##      items                               support    count
## [1]  {other vegetables, whole milk}      0.07483477 736
## [2]  {whole milk, rolls/buns}            0.05663447 557
## [3]  {whole milk, yogurt}                0.05602440 551
## [4]  {root vegetables, whole milk}       0.04890696 481
## [5]  {root vegetables, other vegetables} 0.04738180 466
## [6]  {other vegetables, yogurt}          0.04341637 427
## [7]  {other vegetables, rolls/buns}      0.04260295 419
## [8]  {tropical fruit, whole milk}        0.04229792 416
## [9]  {whole milk, soda}                  0.04006101 394
## [10] {rolls/buns, soda}                  0.03833249 377
```

```
itemsets=apriori(Groceries,
              parameter = list(minlen = 3, maxlen = 3, support = 0.02, target
= "frequent itemsets"))
```

```
## Apriori
##
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##          NA    0.1    1 none FALSE            TRUE       5    0.02      3
##  maxlen            target  ext
##       3 frequent itemsets TRUE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 196
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [59 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3

## Warning in apriori(Groceries, parameter = list(minlen = 3, maxlen = 3, sup
port
## = 0.02, : Mining stopped (maxlen reached). Only patterns up to a length of
3
## returned!

##  done [0.00s].
## sorting transactions ... done [0.00s].
## writing ... [2 set(s)] done [0.00s].
## creating S4 object  ... done [0.00s].

inspect(head(sort(itemsets, by = "support"), 10))

##     items                                         support    count
## [1] {root vegetables, other vegetables, whole milk} 0.02318251 228
## [2] {other vegetables, whole milk, yogurt}          0.02226741 219
```

## Rule generation

```
rules = apriori(Groceries,
              parameter = list(support = 0.001,confidence = 0.6,  target = "r
ules"))
```

```
## Apriori
##
## Parameter specification:
```

```
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##         0.6    0.1    1 none FALSE            TRUE       5   0.001      1
##  maxlen target  ext
##      10  rules TRUE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 9
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [157 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 done [0.01s].
## writing ... [2918 rule(s)] done [0.00s].
## creating S4 object  ... done [0.00s].
```

```
inspect(head(sort(rules, by = "support"), 10))
```

```
##      lhs                        rhs               support confidence    covera
ge     lift count
## [1]  {butter,
##       yogurt}                => {whole milk} 0.009354347  0.6388889 0.0146415
86 2.500387    92
## [2]  {root vegetables,
##       butter}                => {whole milk} 0.008235892  0.6377953 0.0129130
66 2.496107    81
## [3]  {root vegetables,
##       other vegetables,
##       yogurt}                => {whole milk} 0.007829181  0.6062992 0.0129130
66 2.372842    77
## [4]  {tropical fruit,
##       other vegetables,
##       yogurt}                => {whole milk} 0.007625826  0.6198347 0.0123029
99 2.425816    75
## [5]  {tropical fruit,
##       domestic eggs}         => {whole milk} 0.006914082  0.6071429 0.0113879
00 2.376144    68
## [6]  {butter,
##       whipped/sour cream} => {whole milk} 0.006710727  0.6600000 0.0101677
68 2.583008    66
## [7]  {tropical fruit,
##       curd}                  => {whole milk} 0.006507372  0.6336634 0.0102694
46 2.479936    64
## [8]  {tropical fruit,
##       butter}                => {whole milk} 0.006202339  0.6224490 0.0099644
13 2.436047    61
## [9]  {butter,
```

```
##        domestic eggs}      => {whole milk} 0.005998983  0.6210526 0.0096593
80 2.430582    59
## [10] {pip fruit,
##        whipped/sour cream} => {whole milk} 0.005998983  0.6483516 0.0092526
69 2.537421    59
```

- the apriori function can also be used to generate rules.
- In the given code the minimum support thresold is taken as 0.001 and the mininmum confidence thresold is taken as 0.6.
- The inspect function gives the list of top 10 association rule in the descending order of support.

```
rules = apriori(Groceries,
              parameter = list(support = 0.001,confidence = 0.6,  target = "r
ules"))
```

```
## Apriori
##
## Parameter specification:
##  confidence minval smax arem  aval originalSupport maxtime support minlen
##         0.6    0.1    1 none FALSE            TRUE       5   0.001      1
##  maxlen target  ext
##      10  rules TRUE
##
## Algorithmic control:
##  filter tree heap memopt load sort verbose
##     0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 9
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [157 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 done [0.01s].
## writing ... [2918 rule(s)] done [0.00s].
## creating S4 object  ... done [0.00s].
```

```
inspect(head(sort(rules, by = "confidence"), 10))
```

```
##      lhs                     rhs                       support confidence
coverage     lift count
## [1]  {rice,
##        sugar}              => {whole milk}      0.001220132          1 0.
001220132 3.913649    12
## [2]  {canned fish,
##        hygiene articles}    => {whole milk}      0.001118454          1 0.
001118454 3.913649    11
```
8 | Page

```
## [3]  {root vegetables,
```

```
##       butter,
##       rice}                   => {whole milk}       0.001016777          1 0.
001016777 3.913649    10
## [4]  {root vegetables,
##       whipped/sour cream,
##       flour}                  => {whole milk}       0.001728521          1 0.
001728521 3.913649    17
## [5]  {butter,
##       soft cheese,
##       domestic eggs}          => {whole milk}       0.001016777          1 0.
001016777 3.913649    10
## [6]  {citrus fruit,
##       root vegetables,
##       soft cheese}            => {other vegetables} 0.001016777          1 0.
001016777 5.168156    10
## [7]  {pip fruit,
##       butter,
##       hygiene articles}       => {whole milk}       0.001016777          1 0.
001016777 3.913649    10
## [8]  {root vegetables,
##       whipped/sour cream,
##       hygiene articles}       => {whole milk}       0.001016777          1 0.
001016777 3.913649    10
## [9]  {pip fruit,
##       root vegetables,
##       hygiene articles}       => {whole milk}       0.001016777          1 0.
001016777 3.913649    10
## [10] {cream cheese ,
##       domestic eggs,
##       sugar}                  => {whole milk}       0.001118454          1 0.
001118454 3.913649    11
```

**Write a python code to deal with good or bad csv file by removing that column in which all the values are missing and for others columns using average values to fill the missing values**

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call d
rive.mount("/content/drive", force_remount=True).
```

In [2]:
```
import numpy as np
import pandas as pd
```

In [3]:
```
data = pd.read_csv("/content/drive/MyDrive/MSC_Cariculum/SEM3/Data Mining/Practic
data
```

Out[3]:

| | ID | FieldA | FieldB | FieldC | FieldD | FieldE | FieldF | FieldG |
|---|---|---|---|---|---|---|---|---|
| **0** | 1.0 | Good | Better | Best | 1024.0 | NaN | 10241.0 | 1 |
| **1** | 2.0 | Good | NaN | Best | 512.0 | NaN | 5121.0 | 2 |
| **2** | 3.0 | Good | Better | NaN | 256.0 | NaN | 256.0 | 3 |
| **3** | 4.0 | Good | Better | Best | NaN | NaN | 211.0 | 4 |
| **4** | 5.0 | Good | Better | NaN | 64.0 | NaN | 6411.0 | 5 |
| **5** | 6.0 | Good | NaN | Best | 32.0 | NaN | 32.0 | 6 |
| **6** | 7.0 | NaN | Better | Best | 16.0 | NaN | 1611.0 | 7 |
| **7** | 8.0 | NaN | NaN | Best | 8.0 | NaN | 8111.0 | 8 |
| **8** | 9.0 | NaN | NaN | NaN | 4.0 | NaN | 41.0 | 9 |
| **9** | 10.0 | A | B | C | 2.0 | NaN | 21111.0 | 10 |
| **10** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 11 |
| **11** | 10.0 | Good | Better | Best | 1024.0 | NaN | 102411.0 | 12 |
| **12** | 10.0 | Good | NaN | Best | 512.0 | NaN | 512.0 | 13 |
| **13** | 10.0 | Good | Better | NaN | 256.0 | NaN | 1256.0 | 14 |
| **14** | 10.0 | Good | Better | Best | NaN | NaN | NaN | 15 |
| **15** | 10.0 | Good | Better | NaN | 64.0 | NaN | 164.0 | 16 |
| **16** | 10.0 | Good | NaN | Best | 32.0 | NaN | 322.0 | 17 |
| **17** | 10.0 | NaN | Better | Best | 16.0 | NaN | 163.0 | 18 |
| **18** | 10.0 | NaN | NaN | Best | 8.0 | NaN | 844.0 | 19 |
| **19** | 10.0 | NaN | NaN | NaN | 4.0 | NaN | 4555.0 | 20 |
| **20** | 10.0 | A | B | C | 2.0 | NaN | 111.0 | 21 |

In [4]:
```
df = data.copy()
df.head(10)
```

Out[4]:

| | ID | FieldA | FieldB | FieldC | FieldD | FieldE | FieldF | FieldG |
|---|---|---|---|---|---|---|---|---|
| **0** | 1.0 | Good | Better | Best | 1024.0 | NaN | 10241.0 | 1 |
| **1** | 2.0 | Good | NaN | Best | 512.0 | NaN | 5121.0 | 2 |
| **2** | 3.0 | Good | Better | NaN | 256.0 | NaN | 256.0 | 3 |
| **3** | 4.0 | Good | Better | Best | NaN | NaN | 211.0 | 4 |
| **4** | 5.0 | Good | Better | NaN | 64.0 | NaN | 6411.0 | 5 |
| **5** | 6.0 | Good | NaN | Best | 32.0 | NaN | 32.0 | 6 |
| **6** | 7.0 | NaN | Better | Best | 16.0 | NaN | 1611.0 | 7 |
| **7** | 8.0 | NaN | NaN | Best | 8.0 | NaN | 8111.0 | 8 |
| **8** | 9.0 | NaN | NaN | NaN | 4.0 | NaN | 41.0 | 9 |
| **9** | 10.0 | A | B | C | 2.0 | NaN | 21111.0 | 10 |

In [5]:
```
df.shape
```

Out[5]:
```
(21, 8)
```

In [6]:
```
df.isnull().sum()
```

Out[6]:
```
ID         1
FieldA     7
FieldB     9
FieldC     7
FieldD     3
FieldE    21
FieldF     2
FieldG     0
dtype: int64
```

In [7]:
```
df[df.isnull().any(axis=1)] # inorder to check the row which is having the missir
```

Out[7]:

| | ID | FieldA | FieldB | FieldC | FieldD | FieldE | FieldF | FieldG |
|---|---|---|---|---|---|---|---|---|
| **0** | 1.0 | Good | Better | Best | 1024.0 | NaN | 10241.0 | 1 |
| **1** | 2.0 | Good | NaN | Best | 512.0 | NaN | 5121.0 | 2 |
| **2** | 3.0 | Good | Better | NaN | 256.0 | NaN | 256.0 | 3 |
| **3** | 4.0 | Good | Better | Best | NaN | NaN | 211.0 | 4 |
| **4** | 5.0 | Good | Better | NaN | 64.0 | NaN | 6411.0 | 5 |
| **5** | 6.0 | Good | NaN | Best | 32.0 | NaN | 32.0 | 6 |
| **6** | 7.0 | NaN | Better | Best | 16.0 | NaN | 1611.0 | 7 |
| **7** | 8.0 | NaN | NaN | Best | 8.0 | NaN | 8111.0 | 8 |
| **8** | 9.0 | NaN | NaN | NaN | 4.0 | NaN | 41.0 | 9 |
| **9** | 10.0 | A | B | C | 2.0 | NaN | 21111.0 | 10 |
| **10** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 11 |
| **11** | 10.0 | Good | Better | Best | 1024.0 | NaN | 102411.0 | 12 |
| **12** | 10.0 | Good | NaN | Best | 512.0 | NaN | 512.0 | 13 |
| **13** | 10.0 | Good | Better | NaN | 256.0 | NaN | 1256.0 | 14 |
| **14** | 10.0 | Good | Better | Best | NaN | NaN | NaN | 15 |
| **15** | 10.0 | Good | Better | NaN | 64.0 | NaN | 164.0 | 16 |
| **16** | 10.0 | Good | NaN | Best | 32.0 | NaN | 322.0 | 17 |
| **17** | 10.0 | NaN | Better | Best | 16.0 | NaN | 163.0 | 18 |
| **18** | 10.0 | NaN | NaN | Best | 8.0 | NaN | 844.0 | 19 |
| **19** | 10.0 | NaN | NaN | NaN | 4.0 | NaN | 4555.0 | 20 |
| **20** | 10.0 | A | B | C | 2.0 | NaN | 111.0 | 21 |

In [8]:
```
df1 = df.dropna(axis= 1, how = "all")
df1
```

Out[8]:

| | ID | FieldA | FieldB | FieldC | FieldD | FieldF | FieldG |
|---|---|---|---|---|---|---|---|
| **0** | 1.0 | Good | Better | Best | 1024.0 | 10241.0 | 1 |
| **1** | 2.0 | Good | NaN | Best | 512.0 | 5121.0 | 2 |
| **2** | 3.0 | Good | Better | NaN | 256.0 | 256.0 | 3 |
| **3** | 4.0 | Good | Better | Best | NaN | 211.0 | 4 |
| **4** | 5.0 | Good | Better | NaN | 64.0 | 6411.0 | 5 |
| **5** | 6.0 | Good | NaN | Best | 32.0 | 32.0 | 6 |
| **6** | 7.0 | NaN | Better | Best | 16.0 | 1611.0 | 7 |
| **7** | 8.0 | NaN | NaN | Best | 8.0 | 8111.0 | 8 |
| **8** | 9.0 | NaN | NaN | NaN | 4.0 | 41.0 | 9 |
| **9** | 10.0 | A | B | C | 2.0 | 21111.0 | 10 |
| **10** | NaN | NaN | NaN | NaN | NaN | NaN | 11 |
| **11** | 10.0 | Good | Better | Best | 1024.0 | 102411.0 | 12 |
| **12** | 10.0 | Good | NaN | Best | 512.0 | 512.0 | 13 |
| **13** | 10.0 | Good | Better | NaN | 256.0 | 1256.0 | 14 |
| **14** | 10.0 | Good | Better | Best | NaN | NaN | 15 |
| **15** | 10.0 | Good | Better | NaN | 64.0 | 164.0 | 16 |
| **16** | 10.0 | Good | NaN | Best | 32.0 | 322.0 | 17 |
| **17** | 10.0 | NaN | Better | Best | 16.0 | 163.0 | 18 |
| **18** | 10.0 | NaN | NaN | Best | 8.0 | 844.0 | 19 |
| **19** | 10.0 | NaN | NaN | NaN | 4.0 | 4555.0 | 20 |
| **20** | 10.0 | A | B | C | 2.0 | 111.0 | 21 |

In [9]:
```
df1.isnull().sum()
```

Out[9]:
```
ID        1
FieldA    7
FieldB    9
FieldC    7
FieldD    3
FieldF    2
FieldG    0
dtype: int64
```

In [10]:

```
df2 = df1.dropna(axis = 1, how = "any")
df2
```

Out[10]:

| | FieldG |
|---|---|
| **0** | 1 |
| **1** | 2 |
| **2** | 3 |
| **3** | 4 |
| **4** | 5 |
| **5** | 6 |
| **6** | 7 |
| **7** | 8 |
| **8** | 9 |
| **9** | 10 |
| **10** | 11 |
| **11** | 12 |
| **12** | 13 |
| **13** | 14 |
| **14** | 15 |
| **15** | 16 |
| **16** | 17 |
| **17** | 18 |
| **18** | 19 |
| **19** | 20 |
| **20** | 21 |

In [11]:

```
df2.isnull().sum()
```

Out[11]:

```
FieldG    0
dtype: int64
```

In [12]:
```
df2.fillna(df2.mode(), inplace= True)
```

```
/usr/local/lib/python3.7/dist-packages/pandas/core/frame.py:5182: SettingWithCo
pyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/sta
ble/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pyd
ata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-c
opy)
  downcast=downcast,
```

In [13]:
```
df2.isnull().sum()
```

Out[13]:
```
FieldG    0
dtype: int64
```

In [13]:

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call d
rive.mount("/content/drive", force_remount=True).
```

In [ ]:
```
import numpy as np
import pandas as pd
```

In [ ]:
```
data = pd.read_csv("/content/drive/MyDrive/MSC_Cariculum/SEM3/Data Mining/Practic
                   header = 0, low_memory= False, usecols= ["Country", "Place Nam
                   encoding= "latin-1")
data
```

Out[3]:

| | Country | Place Name | Latitude | Longitude |
|---|---|---|---|---|
| **0** | US | New York | 40.7528 | -73.9725 |
| **1** | US | New York | 40.7528 | -73.9725 |
| **2** | US | New York | 40.7528 | -73.9725 |
| **3** | US | New York | 40.7528 | -73.9725 |
| **4** | US | New York | 40.7528 | -73.9725 |
| **...** | ... | ... | ... | ... |
| **3557** | DE | Munich | 48.0915 | 11.5392 |
| **3558** | DE | Munich | 48.1833 | 11.7500 |
| **3559** | DE | Munich | 48.1000 | 11.4667 |
| **3560** | DE | Munich | 48.1480 | 11.7434 |
| **3561** | DE | Munich | 48.1480 | 11.7434 |

3562 rows × 4 columns

In [ ]:
```
df = data.copy()
df.head(10)
```

Out[4]:

| | Country | Place Name | Latitude | Longitude |
|---|---|---|---|---|
| **0** | US | New York | 40.7528 | -73.9725 |
| **1** | US | New York | 40.7528 | -73.9725 |
| **2** | US | New York | 40.7528 | -73.9725 |
| **3** | US | New York | 40.7528 | -73.9725 |
| **4** | US | New York | 40.7528 | -73.9725 |
| **5** | US | New York | 40.7528 | -73.9725 |
| **6** | US | New York | 40.7528 | -73.9725 |
| **7** | US | New York | 40.7528 | -73.9725 |
| **8** | US | New York | 40.7528 | -73.9725 |
| **9** | US | New York | 40.7528 | -73.9725 |

In [ ]:
```
df.rename(columns= {"Place Name":"Place_Name"}, inplace= True)
```

In [ ]:
```
df1 = df[["Country", "Place_Name","Latitude"]]
df1
```

Out[6]:

| | Country | Place_Name | Latitude |
|---|---|---|---|
| **0** | US | New York | 40.7528 |
| **1** | US | New York | 40.7528 |
| **2** | US | New York | 40.7528 |
| **3** | US | New York | 40.7528 |
| **4** | US | New York | 40.7528 |
| **...** | ... | ... | ... |
| **3557** | DE | Munich | 48.0915 |
| **3558** | DE | Munich | 48.1833 |
| **3559** | DE | Munich | 48.1000 |
| **3560** | DE | Munich | 48.1480 |
| **3561** | DE | Munich | 48.1480 |

3562 rows × 3 columns

In [ ]:
```
mean_data = df1.groupby(["Country", "Place_Name"])[ "Latitude"].mean()
mean_data
```

Out[7]:
```
Country  Place_Name
DE       Munich         48.143223
GB       London         51.509406
US       New York       40.747044
Name: Latitude, dtype: float64
```

In [ ]:
```
final_df = pd.DataFrame(mean_data)
final_df
```

Out[8]:

| Country | Place_Name | Latitude |
|---|---|---|
| DE | Munich | 48.143223 |
| GB | London | 51.509406 |
| US | New York | 40.747044 |

In [ ]:
```
final_df.to_csv("/content/drive/MyDrive/MSC_Cariculum/SEM3/Data Mining/Practical4
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.tree import DecisionTreeClassifier
```

In [2]:
```
df = pd.read_csv("null_data_binary.csv")
df.head()
```

Out[2]:

| | Temprature | FFMC | DMC | DC | ISI | Class |
|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | fire |
| **1** | 0 | 0 | 0 | 0 | 0 | fire |
| **2** | 0 | 0 | 0 | 0 | 0 | fire |
| **3** | 0 | 0 | 0 | 0 | 0 | fire |
| **4** | 0 | 0 | 0 | 0 | 0 | notfire |

In [3]:
```
X = df.iloc[:,:-1]
X
```

Out[3]:

| | Temprature | FFMC | DMC | DC | ISI |
|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 |
| **9** | 0 | 0 | 0 | 0 | 0 |

In [4]:
```
y = df.iloc[:,-1]
y
```

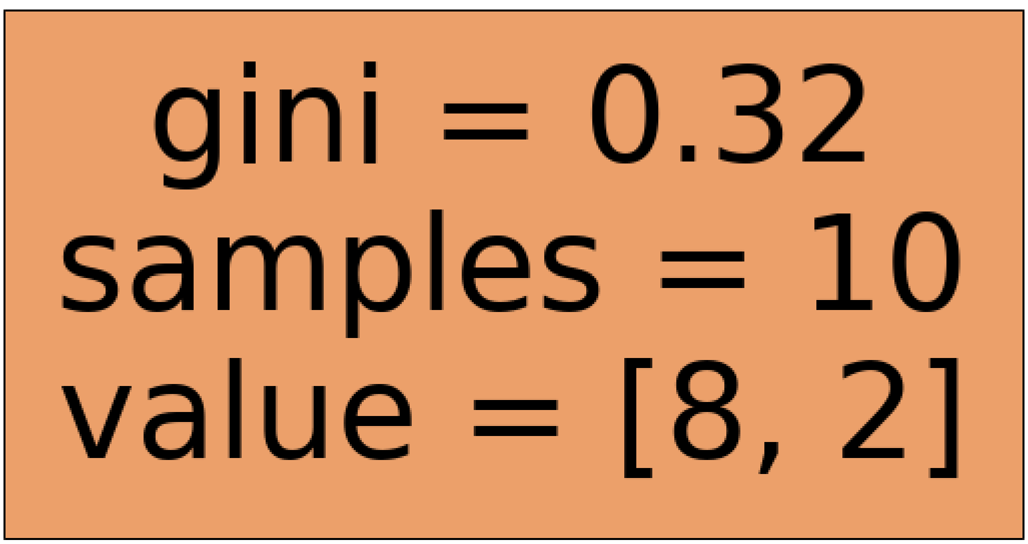
Out[4]:
```
0       fire
1       fire
2       fire
3       fire
4    notfire
5       fire
6       fire
7    notfire
8       fire
9       fire
Name: Class, dtype: object
```

In [5]:
```
Decisiontree = DecisionTreeClassifier()
```

In [6]:
```
Decisiontree.fit(X,y)
```

Out[6]:
```
DecisionTreeClassifier()
```

In [7]:
```
from sklearn import tree
plt.figure(figsize = (10,5))
tree.plot_tree(Decisiontree,filled = True)
plt.show()
```




In [ ]:

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.tree import DecisionTreeClassifier
```

In [2]:
```
df = pd.read_csv("same_data.csv")
df.head()
```

Out[2]:

| | Temprature | FFMC | DMC | DC | ISI | Class |
|---|---|---|---|---|---|---|
| **0** | 72 | 48 | 64 | 66 | 47 | fire |
| **1** | 45 | 56 | 68 | 29 | 30 | fire |
| **2** | 73 | 76 | 71 | 36 | 24 | fire |
| **3** | 57 | 66 | 77 | 49 | 46 | fire |
| **4** | 51 | 50 | 40 | 66 | 72 | fire |

In [3]:
```
X = df.iloc[:,:-1]
X
```

Out[3]:

| | Temprature | FFMC | DMC | DC | ISI |
|---|---|---|---|---|---|
| **0** | 72 | 48 | 64 | 66 | 47 |
| **1** | 45 | 56 | 68 | 29 | 30 |
| **2** | 73 | 76 | 71 | 36 | 24 |
| **3** | 57 | 66 | 77 | 49 | 46 |
| **4** | 51 | 50 | 40 | 66 | 72 |
| **5** | 80 | 61 | 75 | 58 | 27 |
| **6** | 71 | 44 | 24 | 29 | 54 |
| **7** | 66 | 77 | 35 | 61 | 73 |
| **8** | 76 | 39 | 23 | 65 | 79 |
| **9** | 73 | 24 | 65 | 53 | 22 |

In [4]:
```
y = df.iloc[:,-1]
y
```

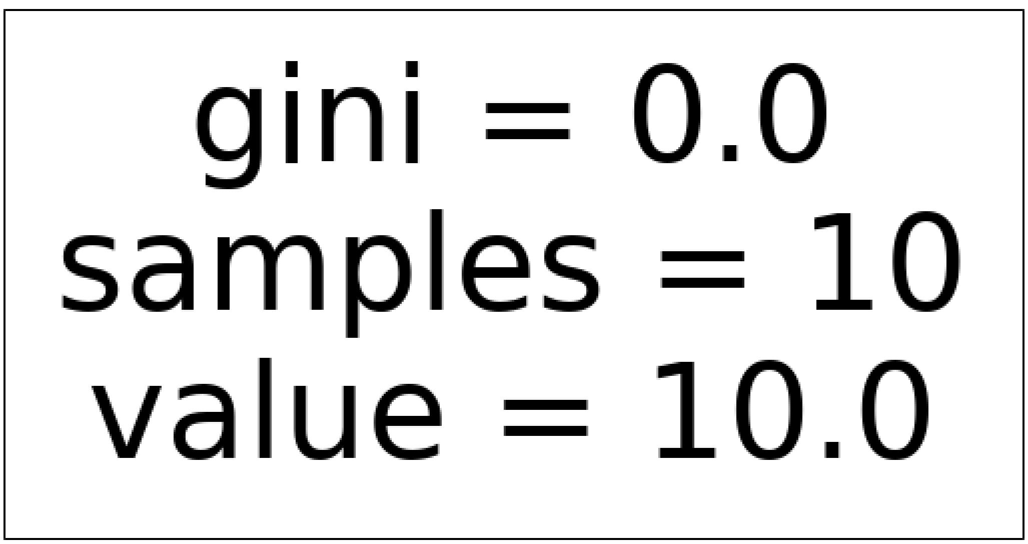
Out[4]:
```
0    fire
1    fire
2    fire
3    fire
4    fire
5    fire
6    fire
7    fire
8    fire
9    fire
Name: Class, dtype: object
```

In [5]:
```
Decisiontree = DecisionTreeClassifier()
```

In [6]:
```
Decisiontree.fit(X,y)
```

Out[6]:
```
DecisionTreeClassifier()
```

In [7]:
```
from sklearn import tree
plt.figure(figsize = (10,5))
tree.plot_tree(Decisiontree,filled = True)
plt.show()
```




In [ ]:

# Naive Bayes

2022-09-15

---

## Name:- Ashishkumar Rana

## Roll No:- 903

## Subject:- Data Mining

## Practical:- Naive Bayes

---

## Question 1

```
df = read.csv("C:\\Users\\User39\\Desktop\\Data Minig\\Naive Bayes.csv")
df
```

```
##          Age Income JobSatisfaction    Desire Enrolls
## 1       <=30   High              No      Fair      No
## 2       <=30   High              No Exceelent      No
## 3   31 to 40   High              No      Fair     Yes
## 4        >40 Medium              No      Fair     Yes
## 5        >40    Low             Yes Excellent     Yes
## 6        >40    Low             Yes Excellent      No
## 7   31 to 40    Low             Yes Excellent     Yes
## 8       <=30 Medium              No      Fair      No
## 9       <=30    Low             Yes      Fair     Yes
## 10       >40 Medium             Yes      Fair     Yes
## 11      <=30 Medium             Yes Excellent     Yes
## 12  31 to 40 Medium              No Excellent     Yes
## 13  31 to 40   High             Yes      Fair     Yes
## 14       >40 Medium              No Excellent      No
## 15      <=30 Medium             Yes      Fair
```

```
head(df)
```

```
##         Age Income JobSatisfaction    Desire Enrolls
## 1      <=30   High              No      Fair      No
## 2      <=30   High              No Exceelent      No
```

```
## 3 31 to 40   High              No      Fair     Yes
## 4      >40 Medium              No      Fair     Yes
## 5      >40    Low             Yes Excellent     Yes
## 6      >40    Low             Yes Excellent      No
```

```
traindata = as.data.frame(df[1:14,])
traindata
```

```
##         Age Income JobSatisfaction    Desire Enrolls
## 1      <=30   High              No      Fair      No
## 2      <=30   High              No Exceelent      No
## 3  31 to 40   High              No      Fair     Yes
## 4       >40 Medium              No      Fair     Yes
## 5       >40    Low             Yes Excellent     Yes
## 6       >40    Low             Yes Excellent      No
## 7  31 to 40    Low             Yes Excellent     Yes
## 8      <=30 Medium              No      Fair      No
## 9      <=30    Low             Yes      Fair     Yes
## 10      >40 Medium             Yes      Fair     Yes
## 11     <=30 Medium             Yes Excellent     Yes
## 12 31 to 40 Medium              No Excellent     Yes
## 13 31 to 40   High             Yes      Fair     Yes
## 14      >40 Medium              No Excellent      No
```

```
testdata = as.data.frame(df[15,])
testdata
```

```
##     Age Income JobSatisfaction Desire Enrolls
## 15 <=30 Medium             Yes   Fair
```

```
library(e1071) # Packge for Naive Bayes
model = naiveBayes(Enrolls~Age+Income+JobSatisfaction+Desire, data=
traindata)
model
```

```
##
## Naive Bayes Classifier for Discrete Predictors
##
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
##
## A-priori probabilities:
## Y
##        No       Yes
## 0.3571429 0.6428571
##
## Conditional probabilities:
##      Age
## Y          <=30       >40  31 to 40
##   No  0.6000000 0.4000000 0.0000000
##   Yes 0.2222222 0.3333333 0.4444444
```

```
## 
##      Income
## Y           High       Low    Medium
##   No  0.4000000 0.2000000 0.4000000
##   Yes 0.2222222 0.3333333 0.4444444
## 
##      JobSatisfaction
## Y             No       Yes
##   No  0.8000000 0.2000000
##   Yes 0.3333333 0.6666667
## 
##      Desire
## Y     Exceelent Excellent      Fair
##   No  0.2000000 0.4000000 0.4000000
##   Yes 0.0000000 0.4444444 0.5555556
```

```
model2 = naiveBayes(Enrolls~Age+Income+JobSatisfaction+Desire, data=
traindata, laplace = 0.01)
model2
```

```
## 
## Naive Bayes Classifier for Discrete Predictors
## 
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
## 
## A-priori probabilities:
## Y
##        No       Yes
## 0.3571429 0.6428571
## 
## Conditional probabilities:
##      Age
## Y          <=30       >40  31 to 40
##   No  0.6020000 0.4020000 0.0020000
##   Yes 0.2233333 0.3344444 0.4455556
## 
##      Income
## Y           High       Low    Medium
##   No  0.4020000 0.2020000 0.4020000
##   Yes 0.2233333 0.3344444 0.4455556
## 
##      JobSatisfaction
## Y             No       Yes
##   No  0.8020000 0.2020000
##   Yes 0.3344444 0.6677778
## 
##      Desire
## Y       Exceelent   Excellent        Fair
```

```
##   No  0.202000000 0.402000000 0.402000000
##   Yes 0.001111111 0.445555556 0.556666667
```

## Question 2

```
library(readxl)
Data <- read_excel("C:/Users/User39/Desktop/Data Minig/Naive Bayes
Data.xlsx")
Data

## # A tibble: 11 × 5
##    Play  Outlook  Temperature Humidity Wind
##    <chr> <chr>    <chr>       <chr>    <lgl>
##  1 yes   rainy    cool        normal   FALSE
##  2 no    rainy    cool        normal   TRUE
##  3 yes   overcast hot         high     FALSE
##  4 no    sunny    mild        high     FALSE
##  5 yes   rainy    cool        normal   FALSE
##  6 yes   sunny    cool        normal   FALSE
##  7 yes   rainy    cool        normal   FALSE
##  8 yes   sunny    hot         normal   FALSE
##  9 yes   overcast mild        high     TRUE
## 10 no    sunny    mild        high     TRUE
## 11 yes   rainy    hot         normal   NA

traindata = as.data.frame(Data[1:10,])
traindata

##    Play  Outlook Temperature Humidity  Wind
## 1   yes    rainy        cool   normal FALSE
## 2    no    rainy        cool   normal  TRUE
## 3   yes overcast         hot     high FALSE
## 4    no    sunny        mild     high FALSE
## 5   yes    rainy        cool   normal FALSE
## 6   yes    sunny        cool   normal FALSE
## 7   yes    rainy        cool   normal FALSE
## 8   yes    sunny         hot   normal FALSE
## 9   yes overcast        mild     high  TRUE
## 10   no    sunny        mild     high  TRUE

testdata = as.data.frame(Data[11,])
testdata

##   Play Outlook Temperature Humidity Wind
## 1  yes   rainy         hot   normal   NA

model = naiveBayes(Wind ~ Play+Outlook+Temperature+Humidity, data = Data)
model

##
## Naive Bayes Classifier for Discrete Predictors
```

```
## 
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
## 
## A-priori probabilities:
## Y
## FALSE  TRUE 
##   0.7   0.3 
## 
## Conditional probabilities:
##        Play
## Y              no       yes
##   FALSE 0.1428571 0.8571429
##   TRUE  0.6666667 0.3333333
## 
##        Outlook
## Y         overcast     rainy     sunny
##   FALSE 0.1428571 0.4285714 0.4285714
##   TRUE  0.3333333 0.3333333 0.3333333
## 
##        Temperature
## Y            cool       hot      mild
##   FALSE 0.5714286 0.2857143 0.1428571
##   TRUE  0.3333333 0.0000000 0.6666667
## 
##        Humidity
## Y            high    normal
##   FALSE 0.2857143 0.7142857
##   TRUE  0.6666667 0.3333333
```

```
model = naiveBayes(Wind ~ Play+Outlook+Temperature+Humidity, data = Data,
laplace = 0.01)
model
```

```
## 
## Naive Bayes Classifier for Discrete Predictors
## 
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
## 
## A-priori probabilities:
## Y
## FALSE  TRUE 
##   0.7   0.3 
## 
## Conditional probabilities:
##        Play
## Y              no       yes
##   FALSE 0.1442857 0.8585714
##   TRUE  0.6700000 0.3366667
```

```
## 
##        Outlook
## Y         overcast     rainy     sunny
##   FALSE 0.1442857 0.4300000 0.4300000
##   TRUE  0.3366667 0.3366667 0.3366667
## 
##        Temperature
## Y               cool         hot        mild
##   FALSE 0.572857143 0.287142857 0.144285714
##   TRUE  0.336666667 0.003333333 0.670000000
## 
##        Humidity
## Y            high    normal
##   FALSE 0.2871429 0.7157143
##   TRUE  0.6700000 0.3366667
```

**Name: Ashishkumar Rana**

**Roll no: 903**

**Subject: Data Mining**

**Practical: Support Vector Machine**

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call d
rive.mount("/content/drive", force_remount=True).
```

In [2]:
```
from sklearn import datasets
```

In [3]:
```
cancer_data = datasets.load_breast_cancer()
print(cancer_data.feature_names)
print(cancer_data.target_names)
```

```
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
['malignant' 'benign']
```

In [4]:
```
cancer_data.data.shape
```

Out[4]:
```
(569, 30)
```

In [5]:
```
cancer_data.data[0]
```

Out[5]:
```
array([1.799e+01, 1.038e+01, 1.228e+02, 1.001e+03, 1.184e-01, 2.776e-01,
       3.001e-01, 1.471e-01, 2.419e-01, 7.871e-02, 1.095e+00, 9.053e-01,
       8.589e+00, 1.534e+02, 6.399e-03, 4.904e-02, 5.373e-02, 1.587e-02,
       3.003e-02, 6.193e-03, 2.538e+01, 1.733e+01, 1.846e+02, 2.019e+03,
       1.622e-01, 6.656e-01, 7.119e-01, 2.654e-01, 4.601e-01, 1.189e-01])
```

In [6]:
```
cancer_data.target[0]
```

Out[6]:
```
0
```

In [7]:
```
x = cancer_data.data
y = cancer_data.target
```

In [8]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.3, random_st
```

In [9]:
```
x_train.shape
```
Out[9]:
```
(398, 30)
```

In [10]:
```
x_test.shape
```
Out[10]:
```
(171, 30)
```

In [11]:
```
from sklearn.svm import SVC
Classifier = SVC(kernel = "linear")
Classifier.fit(x_train,y_train)
```
Out[11]:
```
SVC(kernel='linear')
```

In [12]:
```
y_pred = Classifier.predict(x_test)
```

In [13]:
```
import pandas as pd
df = pd.DataFrame({"Actual Value":y_test, "Predicted_value":y_pred})
df.head(10)
```
Out[13]:

| | Actual Value | Predicted_value |
|---|---|---|
| **0** | 1 | 1 |
| **1** | 0 | 0 |
| **2** | 1 | 1 |
| **3** | 0 | 0 |
| **4** | 0 | 0 |
| **5** | 0 | 0 |
| **6** | 0 | 0 |
| **7** | 0 | 0 |
| **8** | 1 | 1 |
| **9** | 1 | 1 |

In [14]:
```
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report, accuracy_sco
```

In [15]:
```
print(confusion_matrix(y_test, y_pred))
```
```
[[ 57   6]
 [  2 106]]
```

In [16]:
```
accuracy_score(y_test, y_pred)
```
Out[16]:
```
0.9532163742690059
```

In [17]:
```
metrics.precision_score(y_test, y_pred)
```
Out[17]:
```
0.9464285714285714
```

In [18]:
```
metrics.recall_score(y_test, y_pred)
```

Out[18]:
```
0.9814814814814815
```

In [19]:
```
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.97      0.90      0.93        63
           1       0.95      0.98      0.96       108

    accuracy                           0.95       171
   macro avg       0.96      0.94      0.95       171
weighted avg       0.95      0.95      0.95       171
```

**Name: Ashishkumar Rana**

**Roll no: 903**

**Subject: Data Mining**

**Practical: Confusion Matrix**

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call d
rive.mount("/content/drive", force_remount=True).
```

### Import the library for ignorning the warnings

In [2]:
```
import warnings as warnings
warnings.filterwarnings("ignore")
```

In [3]:
```
import numpy as np
import pandas as pd
```

In [4]:
```
data = {"y_actual":[1,0,0,1,0,1,0,0,1,0,1,0],
        "y_pred":[1,1,0,1,0,1,1,0,1,0,0,0]}
```

In [5]:
```
df = pd.DataFrame(data,columns=["y_actual","y_pred"])
df.head()
```

Out[5]:

| | y_actual | y_pred |
|---|---|---|
| **0** | 1 | 1 |
| **1** | 0 | 1 |
| **2** | 0 | 0 |
| **3** | 1 | 1 |
| **4** | 0 | 0 |

### Creating Confusion matrix using pandas library

In [6]:
```
Confusion_matrix = pd.crosstab(df["y_actual"],df["y_pred"], rownames=["Actual"],
Confusion_matrix
```

Out[6]:

| Predicted | 0 | 1 |
|---|---|---|
| **Actual** | | |
| **0** | 5 | 2 |
| **1** | 1 | 4 |

### Creating Confusion matrix with marginal

In [7]:
```
Confusion_matrix_marginal = pd.crosstab(df["y_actual"],df["y_pred"], rownames=["A
Confusion_matrix_marginal
```

Out[7]:

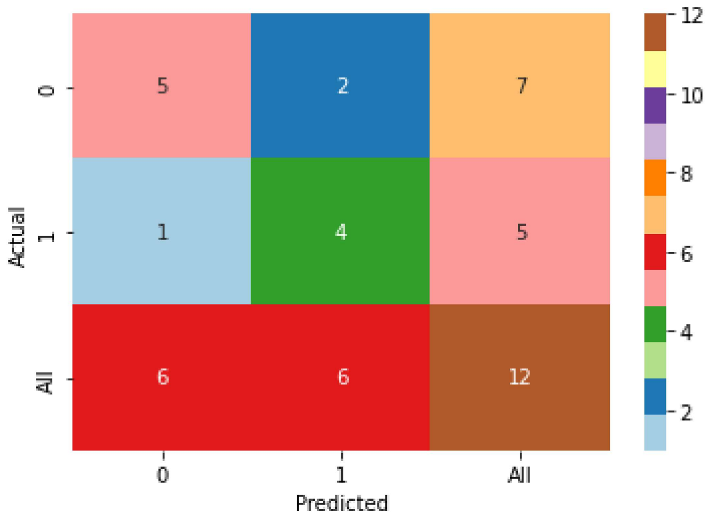
| Predicted | 0 | 1 | All |
|---|---|---|---|
| **Actual** | | | |
| **0** | 5 | 2 | 7 |
| **1** | 1 | 4 | 5 |
| **All** | 6 | 6 | 12 |

### Using seaborn libraray to create Confusion matrix

In [8]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

In [9]:
```
sns.heatmap(Confusion_matrix_marginal,annot = True, cmap = "Paired")
```

Out[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f19ab27ed50>
```




### Creating Confusion matrix using sklearn library

In [10]:
```
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report
```

In [11]:
```
print("::::::::::Confusion Matrix::::::::")
print(confusion_matrix(df["y_actual"],df["y_pred"]))
```

```
::::::::::Confusion Matrix::::::::
[[5 2]
 [1 4]]
```

In [12]:
```
print("::::::::::Classification Report::::::::")
print(classification_report(df["y_actual"],df["y_pred"]))
```

```
::::::::::Classification Report::::::::
              precision    recall  f1-score   support

           0       0.83      0.71      0.77         7
           1       0.67      0.80      0.73         5

    accuracy                           0.75        12
   macro avg       0.75      0.76      0.75        12
weighted avg       0.76      0.75      0.75        12
```

In [12]:

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call d
rive.mount("/content/drive", force_remount=True).
```

In [2]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn import metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error,mean_absolute
```

In [3]:
```
from  sklearn.datasets import fetch_california_housing
```

In [4]:
```
data = fetch_california_housing(as_frame = True)
```

In [5]:
```
df = data.frame
```

In [6]:
```
df.head(10)
```

Out[6]:

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude | MedHo |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 8.3252 | 41.0 | 6.984127 | 1.023810 | 322.0 | 2.555556 | 37.88 | -122.23 | |
| **1** | 8.3014 | 21.0 | 6.238137 | 0.971880 | 2401.0 | 2.109842 | 37.86 | -122.22 | |
| **2** | 7.2574 | 52.0 | 8.288136 | 1.073446 | 496.0 | 2.802260 | 37.85 | -122.24 | |
| **3** | 5.6431 | 52.0 | 5.817352 | 1.073059 | 558.0 | 2.547945 | 37.85 | -122.25 | |
| **4** | 3.8462 | 52.0 | 6.281853 | 1.081081 | 565.0 | 2.181467 | 37.85 | -122.25 | |
| **5** | 4.0368 | 52.0 | 4.761658 | 1.103627 | 413.0 | 2.139896 | 37.85 | -122.25 | |
| **6** | 3.6591 | 52.0 | 4.931907 | 0.951362 | 1094.0 | 2.128405 | 37.84 | -122.25 | |
| **7** | 3.1200 | 52.0 | 4.797527 | 1.061824 | 1157.0 | 1.788253 | 37.84 | -122.25 | |
| **8** | 2.0804 | 42.0 | 4.294118 | 1.117647 | 1206.0 | 2.026891 | 37.84 | -122.26 | |
| **9** | 3.6912 | 52.0 | 4.970588 | 0.990196 | 1551.0 | 2.172269 | 37.84 | -122.25 | |

In [7]:
```
df.tail(10)
```

Out[7]:

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude | Me |
|---|---|---|---|---|---|---|---|---|---|
| **20630** | 3.5673 | 11.0 | 5.932584 | 1.134831 | 1257.0 | 2.824719 | 39.29 | -121.32 | |
| **20631** | 3.5179 | 15.0 | 6.145833 | 1.141204 | 1200.0 | 2.777778 | 39.33 | -121.40 | |
| **20632** | 3.1250 | 15.0 | 6.023377 | 1.080519 | 1047.0 | 2.719481 | 39.26 | -121.45 | |
| **20633** | 2.5495 | 27.0 | 5.445026 | 1.078534 | 1082.0 | 2.832461 | 39.19 | -121.53 | |
| **20634** | 3.7125 | 28.0 | 6.779070 | 1.148256 | 1041.0 | 3.026163 | 39.27 | -121.56 | |
| **20635** | 1.5603 | 25.0 | 5.045455 | 1.133333 | 845.0 | 2.560606 | 39.48 | -121.09 | |
| **20636** | 2.5568 | 18.0 | 6.114035 | 1.315789 | 356.0 | 3.122807 | 39.49 | -121.21 | |
| **20637** | 1.7000 | 17.0 | 5.205543 | 1.120092 | 1007.0 | 2.325635 | 39.43 | -121.22 | |
| **20638** | 1.8672 | 18.0 | 5.329513 | 1.171920 | 741.0 | 2.123209 | 39.43 | -121.32 | |
| **20639** | 2.3886 | 16.0 | 5.254717 | 1.162264 | 1387.0 | 2.616981 | 39.37 | -121.24 | |

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   MedInc       20640 non-null  float64
 1   HouseAge     20640 non-null  float64
 2   AveRooms     20640 non-null  float64
 3   AveBedrms    20640 non-null  float64
 4   Population   20640 non-null  float64
 5   AveOccup     20640 non-null  float64
 6   Latitude     20640 non-null  float64
 7   Longitude    20640 non-null  float64
 8   MedHouseVal  20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
```

In [9]:
```
df.shape
```

Out[9]: (20640, 9)

In [10]:
```
df.size
```

Out[10]: 185760

In [11]:
```
x = df.iloc[:,:-1]
x
```

Out[11]:

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude |
|---|---|---|---|---|---|---|---|---|
| **0** | 8.3252 | 41.0 | 6.984127 | 1.023810 | 322.0 | 2.555556 | 37.88 | -122.23 |
| **1** | 8.3014 | 21.0 | 6.238137 | 0.971880 | 2401.0 | 2.109842 | 37.86 | -122.22 |
| **2** | 7.2574 | 52.0 | 8.288136 | 1.073446 | 496.0 | 2.802260 | 37.85 | -122.24 |
| **3** | 5.6431 | 52.0 | 5.817352 | 1.073059 | 558.0 | 2.547945 | 37.85 | -122.25 |
| **4** | 3.8462 | 52.0 | 6.281853 | 1.081081 | 565.0 | 2.181467 | 37.85 | -122.25 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **20635** | 1.5603 | 25.0 | 5.045455 | 1.133333 | 845.0 | 2.560606 | 39.48 | -121.09 |
| **20636** | 2.5568 | 18.0 | 6.114035 | 1.315789 | 356.0 | 3.122807 | 39.49 | -121.21 |
| **20637** | 1.7000 | 17.0 | 5.205543 | 1.120092 | 1007.0 | 2.325635 | 39.43 | -121.22 |
| **20638** | 1.8672 | 18.0 | 5.329513 | 1.171920 | 741.0 | 2.123209 | 39.43 | -121.32 |
| **20639** | 2.3886 | 16.0 | 5.254717 | 1.162264 | 1387.0 | 2.616981 | 39.37 | -121.24 |

20640 rows × 8 columns

In [12]:
```
x.shape
```

Out[12]:
```
(20640, 8)
```

In [13]:
```
y = df.iloc[:,-1]
y
```

Out[13]:
```
0        4.526
1        3.585
2        3.521
3        3.413
4        3.422
         ...
20635    0.781
20636    0.771
20637    0.923
20638    0.847
20639    0.894
Name: MedHouseVal, Length: 20640, dtype: float64
```

In [14]:
```
y.shape
```

Out[14]:
```
(20640,)
```

In [15]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(x)
```

In [16]:
```
# separate dataset into train and test
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.25,random_sta
X_train.shape, X_test.shape
```

Out[16]:
```
((15480, 8), (5160, 8))
```

In [17]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(15480, 8)
(5160, 8)
(15480,)
(5160,)
```

In [18]:
```
KNN_Regressor = KNeighborsRegressor(n_neighbors=5)
KNN_Regressor.fit(X_train,y_train)
```

Out[18]:
```
KNeighborsRegressor()
```

In [19]:
```
print("Training Score:",KNN_Regressor.score(X_train, y_train))
print("Test Score:",KNN_Regressor.score(X_test,y_test))
```

```
Training Score: 0.7964929714092259
Test Score: 0.6737730861932387
```

In [20]:
```
KNN_Regressor_Prediction = KNN_Regressor.predict(X_test)
KNN_Regressor_Prediction
```

Out[20]:
```
array([3.5354, 0.6036, 2.5062, ..., 0.737 , 2.3528, 1.0536])
```

In [21]:
```
Actual_predicted = pd.DataFrame({'Actual Values': y_test, 'Predicted Values': KNN
Actual_predicted.head(10)
```

Out[21]:

| | Actual Values | Predicted Values |
|---|---|---|
| **4712** | 3.550 | 3.535400 |
| **2151** | 0.707 | 0.603600 |
| **15927** | 2.294 | 2.506200 |
| **82** | 1.125 | 0.991400 |
| **8161** | 2.254 | 2.336800 |
| **6636** | 2.630 | 3.835404 |
| **17333** | 2.268 | 2.396400 |
| **19081** | 1.662 | 1.290800 |
| **13298** | 1.180 | 1.639200 |
| **7157** | 1.563 | 2.222000 |

In [22]:
```
meanAbErr = metrics.mean_absolute_error(y_test, KNN_Regressor_Prediction)
meanSqErr = metrics.mean_squared_error(y_test, KNN_Regressor_Prediction)
rootMeanSqErr = np.sqrt(metrics.mean_squared_error(y_test, KNN_Regressor_Predicti

print('Mean Absolute Error:', meanAbErr)
print('Mean Square Error:', meanSqErr)
print('Root Mean Square Error:', rootMeanSqErr)
```

```
Mean Absolute Error: 0.4443545546511627
Mean Square Error: 0.42933392108343804
Root Mean Square Error: 0.6552357751858777
```

In [22]:

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call d
rive.mount("/content/drive", force_remount=True).
```

In [2]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn import metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error,mean_absolute
```

In [3]:
```
from  sklearn.datasets import fetch_california_housing
```

In [4]:
```
data = fetch_california_housing(as_frame = True)
```

In [5]:
```
df = data.frame
```

In [6]:
```
df.head(10)
```

Out[6]:

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude | MedHo |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 8.3252 | 41.0 | 6.984127 | 1.023810 | 322.0 | 2.555556 | 37.88 | -122.23 | |
| 1 | 8.3014 | 21.0 | 6.238137 | 0.971880 | 2401.0 | 2.109842 | 37.86 | -122.22 | |
| 2 | 7.2574 | 52.0 | 8.288136 | 1.073446 | 496.0 | 2.802260 | 37.85 | -122.24 | |
| 3 | 5.6431 | 52.0 | 5.817352 | 1.073059 | 558.0 | 2.547945 | 37.85 | -122.25 | |
| 4 | 3.8462 | 52.0 | 6.281853 | 1.081081 | 565.0 | 2.181467 | 37.85 | -122.25 | |
| 5 | 4.0368 | 52.0 | 4.761658 | 1.103627 | 413.0 | 2.139896 | 37.85 | -122.25 | |
| 6 | 3.6591 | 52.0 | 4.931907 | 0.951362 | 1094.0 | 2.128405 | 37.84 | -122.25 | |
| 7 | 3.1200 | 52.0 | 4.797527 | 1.061824 | 1157.0 | 1.788253 | 37.84 | -122.25 | |
| 8 | 2.0804 | 42.0 | 4.294118 | 1.117647 | 1206.0 | 2.026891 | 37.84 | -122.26 | |
| 9 | 3.6912 | 52.0 | 4.970588 | 0.990196 | 1551.0 | 2.172269 | 37.84 | -122.25 | |

In [7]:
```
df.tail(10)
```

Out[7]:

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude | Me |
|---|---|---|---|---|---|---|---|---|---|
| **20630** | 3.5673 | 11.0 | 5.932584 | 1.134831 | 1257.0 | 2.824719 | 39.29 | -121.32 | |
| **20631** | 3.5179 | 15.0 | 6.145833 | 1.141204 | 1200.0 | 2.777778 | 39.33 | -121.40 | |
| **20632** | 3.1250 | 15.0 | 6.023377 | 1.080519 | 1047.0 | 2.719481 | 39.26 | -121.45 | |
| **20633** | 2.5495 | 27.0 | 5.445026 | 1.078534 | 1082.0 | 2.832461 | 39.19 | -121.53 | |
| **20634** | 3.7125 | 28.0 | 6.779070 | 1.148256 | 1041.0 | 3.026163 | 39.27 | -121.56 | |
| **20635** | 1.5603 | 25.0 | 5.045455 | 1.133333 | 845.0 | 2.560606 | 39.48 | -121.09 | |
| **20636** | 2.5568 | 18.0 | 6.114035 | 1.315789 | 356.0 | 3.122807 | 39.49 | -121.21 | |
| **20637** | 1.7000 | 17.0 | 5.205543 | 1.120092 | 1007.0 | 2.325635 | 39.43 | -121.22 | |
| **20638** | 1.8672 | 18.0 | 5.329513 | 1.171920 | 741.0 | 2.123209 | 39.43 | -121.32 | |
| **20639** | 2.3886 | 16.0 | 5.254717 | 1.162264 | 1387.0 | 2.616981 | 39.37 | -121.24 | |

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   MedInc       20640 non-null  float64
 1   HouseAge     20640 non-null  float64
 2   AveRooms     20640 non-null  float64
 3   AveBedrms    20640 non-null  float64
 4   Population   20640 non-null  float64
 5   AveOccup     20640 non-null  float64
 6   Latitude     20640 non-null  float64
 7   Longitude    20640 non-null  float64
 8   MedHouseVal  20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
```

In [9]:
```
df.shape
```

Out[9]: (20640, 9)

In [10]:
```
df.size
```

Out[10]: 185760

In [11]:
```
x = df.iloc[:,:-1]
x
```

Out[11]:

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude |
|---|---|---|---|---|---|---|---|---|
| **0** | 8.3252 | 41.0 | 6.984127 | 1.023810 | 322.0 | 2.555556 | 37.88 | -122.23 |
| **1** | 8.3014 | 21.0 | 6.238137 | 0.971880 | 2401.0 | 2.109842 | 37.86 | -122.22 |
| **2** | 7.2574 | 52.0 | 8.288136 | 1.073446 | 496.0 | 2.802260 | 37.85 | -122.24 |
| **3** | 5.6431 | 52.0 | 5.817352 | 1.073059 | 558.0 | 2.547945 | 37.85 | -122.25 |
| **4** | 3.8462 | 52.0 | 6.281853 | 1.081081 | 565.0 | 2.181467 | 37.85 | -122.25 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **20635** | 1.5603 | 25.0 | 5.045455 | 1.133333 | 845.0 | 2.560606 | 39.48 | -121.09 |
| **20636** | 2.5568 | 18.0 | 6.114035 | 1.315789 | 356.0 | 3.122807 | 39.49 | -121.21 |
| **20637** | 1.7000 | 17.0 | 5.205543 | 1.120092 | 1007.0 | 2.325635 | 39.43 | -121.22 |
| **20638** | 1.8672 | 18.0 | 5.329513 | 1.171920 | 741.0 | 2.123209 | 39.43 | -121.32 |
| **20639** | 2.3886 | 16.0 | 5.254717 | 1.162264 | 1387.0 | 2.616981 | 39.37 | -121.24 |

20640 rows × 8 columns

In [12]:
```
x.shape
```

Out[12]:
```
(20640, 8)
```

In [13]:
```
y = df.iloc[:,-1]
y
```

Out[13]:
```
0        4.526
1        3.585
2        3.521
3        3.413
4        3.422
         ...
20635    0.781
20636    0.771
20637    0.923
20638    0.847
20639    0.894
Name: MedHouseVal, Length: 20640, dtype: float64
```

In [14]:
```
y.shape
```

Out[14]:
```
(20640,)
```

In [15]:
```
from sklearn.preprocessing import MinMaxScaler
min_max=MinMaxScaler()
df_minmax=pd.DataFrame(min_max.fit_transform(x),columns=x.columns)
df_minmax.head()
```

Out[15]:

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude |
|---|---|---|---|---|---|---|---|---|
| **0** | 0.539668 | 0.784314 | 0.043512 | 0.020469 | 0.008941 | 0.001499 | 0.567481 | 0.211155 |
| **1** | 0.538027 | 0.392157 | 0.038224 | 0.018929 | 0.067210 | 0.001141 | 0.565356 | 0.212151 |
| **2** | 0.466028 | 1.000000 | 0.052756 | 0.021940 | 0.013818 | 0.001698 | 0.564293 | 0.210159 |
| **3** | 0.354699 | 1.000000 | 0.035241 | 0.021929 | 0.015555 | 0.001493 | 0.564293 | 0.209163 |
| **4** | 0.230776 | 1.000000 | 0.038534 | 0.022166 | 0.015752 | 0.001198 | 0.564293 | 0.209163 |

In [16]:
```
# separate dataset into train and test
X_train, X_test, y_train, y_test = train_test_split(df_minmax,y,test_size=0.25,ra
X_train.shape, X_test.shape
```

Out[16]: ((15480, 8), (5160, 8))

In [17]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(15480, 8)
(5160, 8)
(15480,)
(5160,)
```

In [18]:
```
KNN_Regressor = KNeighborsRegressor(n_neighbors=5)
KNN_Regressor.fit(X_train,y_train)
```

Out[18]: KNeighborsRegressor()

In [19]:
```
print("Training Score:",KNN_Regressor.score(X_train, y_train))
print("Test Score:",KNN_Regressor.score(X_test,y_test))
```

```
Training Score: 0.8041397300295833
Test Score: 0.6976447422494866
```

In [20]:
```
KNN_Regressor_Prediction = KNN_Regressor.predict(X_test)
KNN_Regressor_Prediction
```

Out[20]: array([2.201 , 0.6466, 2.5836, ..., 0.977 , 2.2138, 1.0994])

In [21]:
```
Actual_predicted = pd.DataFrame({'Actual Values': y_test, 'Predicted Values': KNN
Actual_predicted.head(10)
```

Out[21]:

| | Actual Values | Predicted Values |
|---|---|---|
| **4712** | 3.550 | 2.201000 |
| **2151** | 0.707 | 0.646600 |
| **15927** | 2.294 | 2.583600 |
| **82** | 1.125 | 1.832800 |
| **8161** | 2.254 | 2.311000 |
| **6636** | 2.630 | 3.777002 |
| **17333** | 2.268 | 2.733400 |
| **19081** | 1.662 | 1.391400 |
| **13298** | 1.180 | 1.489200 |
| **7157** | 1.563 | 1.931600 |

In [22]:
```
meanAbErr = metrics.mean_absolute_error(y_test, KNN_Regressor_Prediction)
meanSqErr = metrics.mean_squared_error(y_test, KNN_Regressor_Prediction)
rootMeanSqErr = np.sqrt(metrics.mean_squared_error(y_test, KNN_Regressor_Predicti

print('Mean Absolute Error:', meanAbErr)
print('Mean Square Error:', meanSqErr)
print('Root Mean Square Error:', rootMeanSqErr)
```

```
Mean Absolute Error: 0.4225393023255814
Mean Square Error: 0.3979174092518759
Root Mean Square Error: 0.6308069508588788
```

In [22]: