Practical No:01

Aim: Design a bot using AIML.

CODE: We need to create following three files.

1) std-startup.xml

```
<aiml version="1.0.1" encoding="UTF-8">
<!-- std-startup.xml -->
        <category>
                <pattern>LOAD</pattern>
                <template>
                        <learn>basic_chat.aiml</learn>
                </template>
        </category>
</aiml>
```

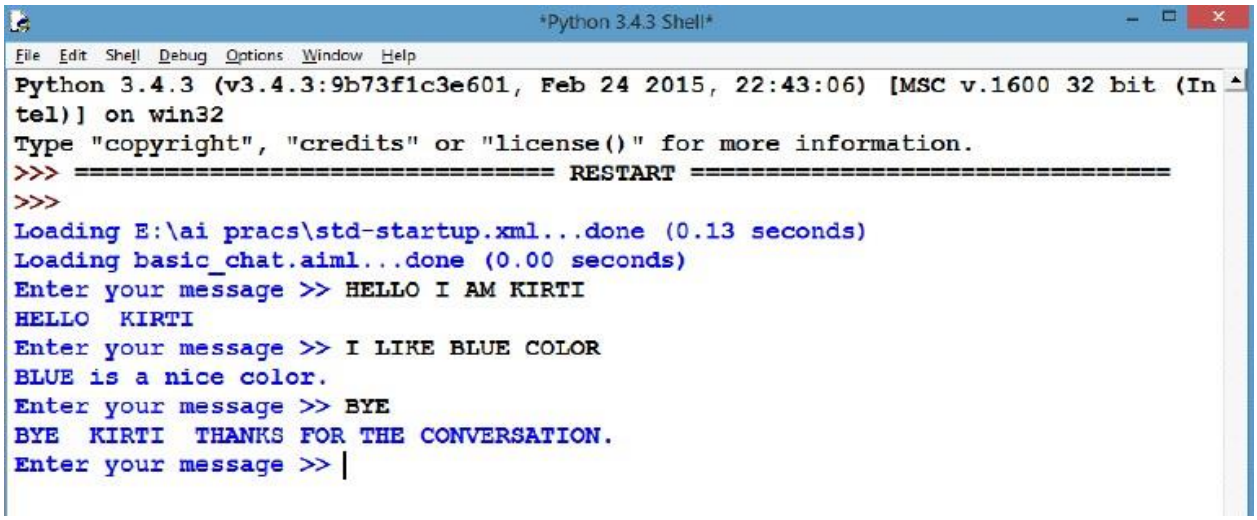
2)basic_chat.aiml

```
<aiml version="1.0.1" encoding="UTF-8">
<!-- basic_chat.aiml -->
        <category>
                <pattern>HELLO I AM *</pattern>
                <template> HELLO <set name="username"> <star/> </set> </template>
        </category>
        <category>
                <pattern>I LIKE * COLOR</pattern>
                <template><star index="1"/> is a nice color.</template>
        </category>
        <category>
                <pattern>BYE</pattern>
                <template> BYE <get name="username"/> THANKS FOR THE
        CONVERSATION.
        </template>
        </category>
</aiml>
```

3) testbot.py

```
import aiml

kernel = aiml.Kernel()
kernel.learn("F:\\aai  practs\\std-startup.xml")
kernel.learn("F:\\aai practs\\basic_chat.aiml")
kernel.respond("LOAD")
# Press CTRL-C to break this loop
while True:
        print( kernel.respond(input("Enter your message >> ")))
```

Output:



Practical No:02

Aim: Implement Bayes Theorem using Python

An excellent and widely used example of the benefit of Bayes Theorem is in the analysis of a medical diagnostic test.

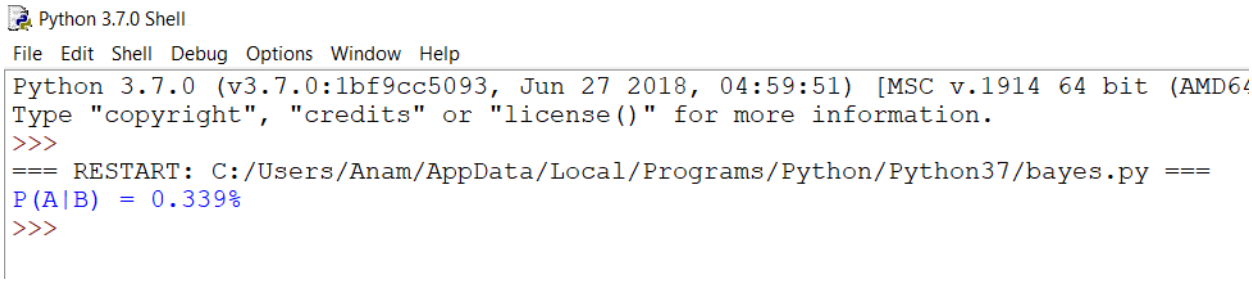
Scenario: Consider a human population that may or may not have cancer (Cancer is True or False) and a medical test that returns positive or negative for detecting cancer (Test is Positive or Negative)

Code:

```
# calculate the probability of cancer patient and diagnostic test
# calculate P(A|B) given P(A), P(B|A), P(B|not A)
def bayes_theorem(p_a, p_b_given_a, p_b_given_not_a):
        # calculate P(not A)
        not_a = 1 - p_a
        # calculate P(B)
        p_b = p_b_given_a * p_a + p_b_given_not_a * not_a
        # calculate P(A|B)
        p_a_given_b = (p_b_given_a * p_a) / p_b
        return p_a_given_b

# P(A)
p_a = 0.0002
# P(B|A)
p_b_given_a = 0.85
# P(B|not A)
p_b_given_not_a = 0.05
# calculate P(A|B)
result = bayes_theorem(p_a, p_b_given_a, p_b_given_not_a)
# summarize
print('P(A|B) = %.3f%%' % (result * 100))
```

Running this program calculates the probability that a patient has cancer.

Output:


Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64
Type "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:/Users/Anam/AppData/Local/Programs/Python/Python37/bayes.py ===
P(A|B) = 0.339%
>>>

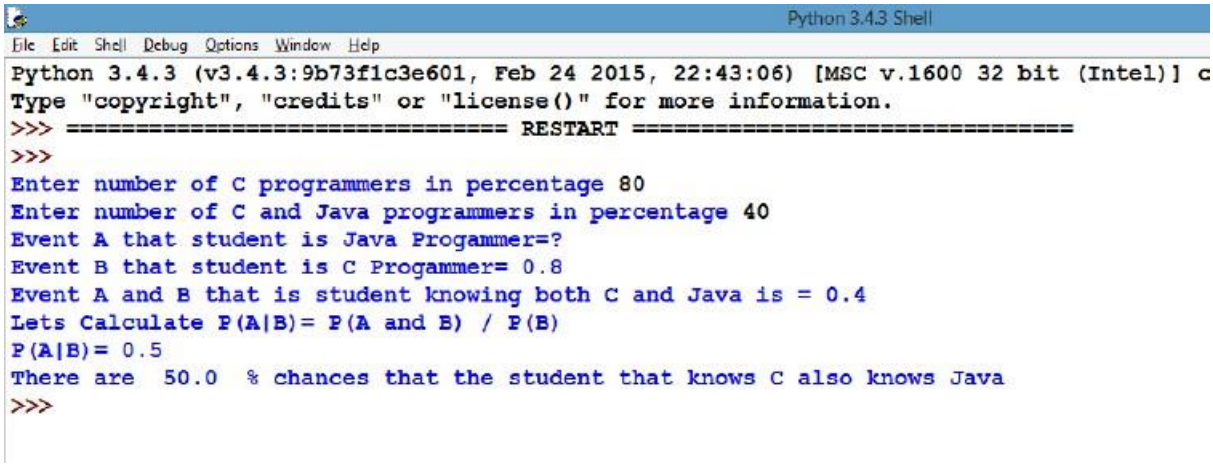
Practical No:03

Aim: Implement Conditional Probability and joint probability using Python.

Code for conditional probability:

```
pofB=float(input("Enter number of C programmers in percentage "))
pofAandB=float(input("Enter number of C and Java programmers in percentage "))
pofB=pofB/100
pofAandB=pofAandB/100
print("Event A that student is Java Progammer=?")
print("Event B that student is C Progammer=",pofB)
print("Event A and B that is student knowing both C and Java is =",pofAandB)
print("Lets Calculate P(A|B)= P(A and B) / P(B)")
pAgivenB=pofAandB / pofB
print("P(A|B)=",pAgivenB)
print("There are ",pAgivenB *100," % chances that the student that knows C also knows Java")
```

Output:

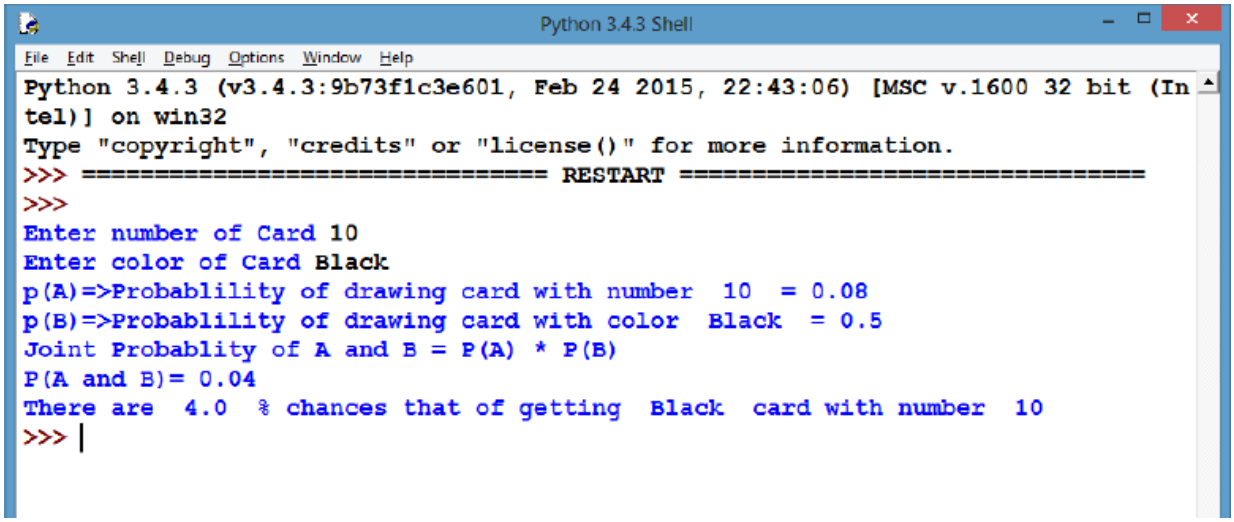


Code for Joint Probability:

```
cardnumber=input("Enter number of Card")
cardcolor=input("Enter color of Card")
pofA=4/52
pofB=26/52
```

```
print("p(A)=>Probablility of drawing card with number ",cardnumber," =",round(pofA,2))
print("p(B)=>Probablility of drawing card with color ",cardcolor," =",round(pofB,2))
print("Joint Probablity of A and B = P(A) * P(B)")
pAandB=round(pofA * pofB,2)
print("P(A and B)=",pAandB)
print("There are ",pAandB *100," % chances that of getting ",cardcolor, " card with number ",cardnumber)
```

Output:



Practical No:04

Aim: Design a Fuzzy based application using Python / R.

Code:

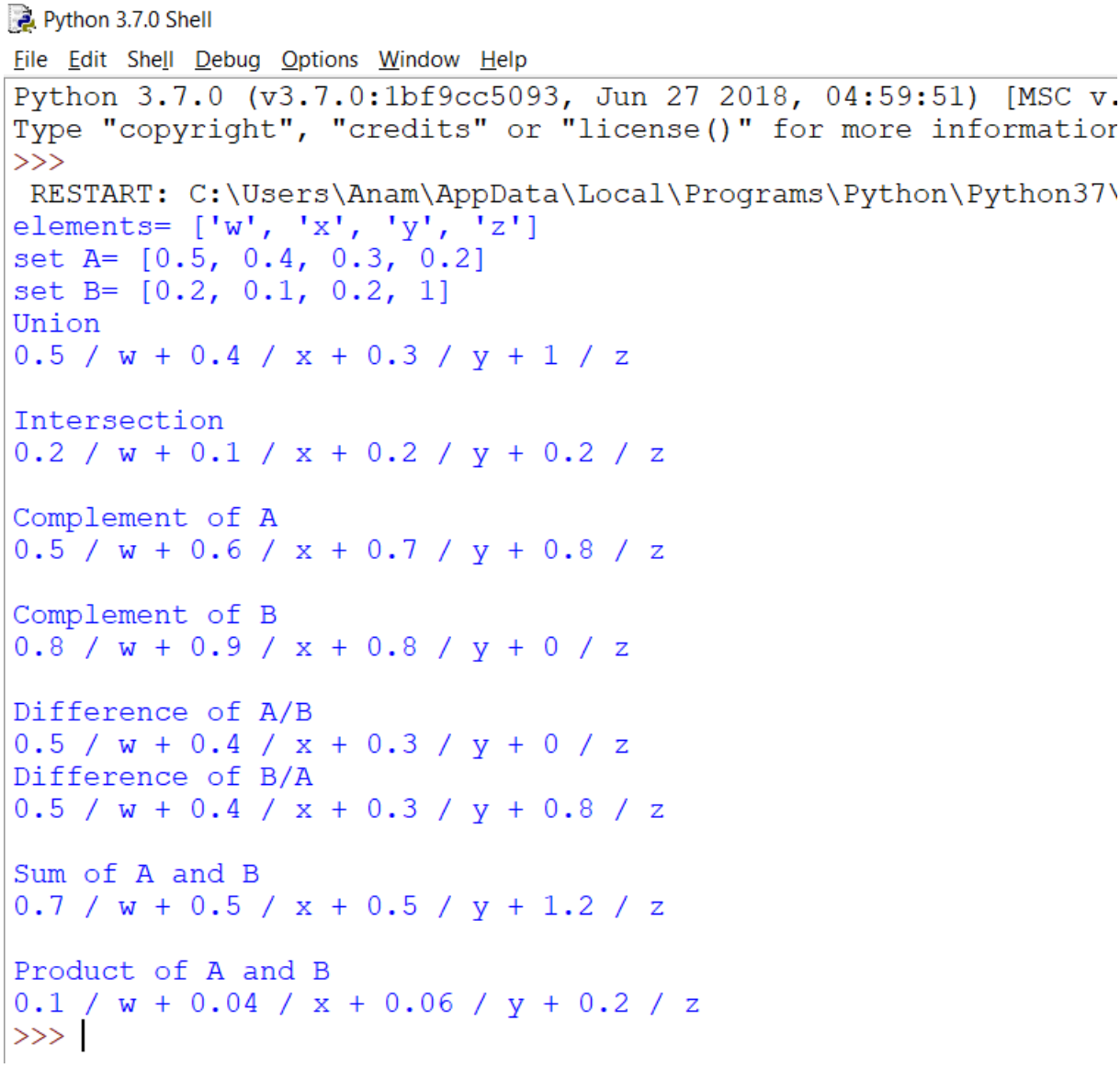
```
elt=['w','x','y','z']
A=[0.5,0.4,0.3,0.2]
B=[0.2,0.1,0.2,1]
U=[]
print("elements=",elt)
print("set A=",A)
print("set B=",B)
for i in range(0,4):
    if A[i]>B[i]:
        U.append(A[i])
    else:
        U.append(B[i])
print("Union")
for i in range(0,3):
    print(U[i] ,"/",elt[i],end=' + ')
for i in range(3,4):
    print(U[i] ,"/",elt[i],end=' ')
print()
I=[]
for i in range(0,4):
    if A[i]<B[i]:
        I.append(A[i])
    else:
        I.append(B[i])
print()
print("Intersection")
for i in range(0,3):
    print(I[i] ,"/",elt[i],end=' + ')
for i in range(3,4):
    print(I[i] ,"/",elt[i],end=' ')
```

```
print()
J=[]
K=[]
C=[1,1,1,1]
print()
print("Complement of A")
for i in range(0,4):
    J.append(C[i]-A[i])
    output=round(J[i],2)
for i in range(0,3):
    print(J[i] ,"/",elt[i],end=' + ')
for i in range(3,4):
    print(J[i] ,"/",elt[i],end=' ')
    print()
print()
print("Complement of B")
for i in range(0,4):
    K.append(C[i]-B[i])
for i in range(0,3):
    print(K[i] ,"/",elt[i],end=' + ')
for i in range(3,4):
    print(K[i] ,"/",elt[i],end=' ')
L=[]
M=[]
print()
for i in range(0,4):
    if A[i]<K[i]:
        L.append(A[i])
    else:
        L.append(K[i])
print()
print("Difference of A/B")
for i in range(0,3):
    print(L[i] ,"/",elt[i],end=' + ')
for i in range(3,4):
```

```
    print(L[i] ,"/",elt[i],end=' ')
for i in range(0,4):
    if B[i]<J[i]:
        M.append(A[i])
    else:
        M.append(J[i])
print()
print("Difference of B/A")
for i in range(0,3):
    print(M[i] ,"/",elt[i],end=' + ')
for i in range(3,4):
    print(M[i] ,"/",elt[i],end=' ')
print()
Sum=[]
Sum1=[]
print()
print("Sum of A and B")
for i in range(0,4):
    Sum.append(A[i]+B[i])
    output=round(Sum[i],2)
    Sum1.append(output)
for i in range(0,3):
    print(Sum1[i] ,"/",elt[i],end=' + ')
for i in range(3,4):
    print(Sum1[i] ,"/",elt[i],end=' ')
print()
Prod=[]
Prod1=[]
print()
print("Product of A and B")
for i in range(0,4):
    Prod.append(A[i]*B[i])
    output=round(Prod[i],2)
    Prod1.append(output)
for i in range(0,3):
```

```
    print(Prod1[i] ,"/",elt[i],end=' + ')
for i in range(3,4):
    print(Prod1[i] ,"/",elt[i],end=' ')
```

Output:



Practical No:05

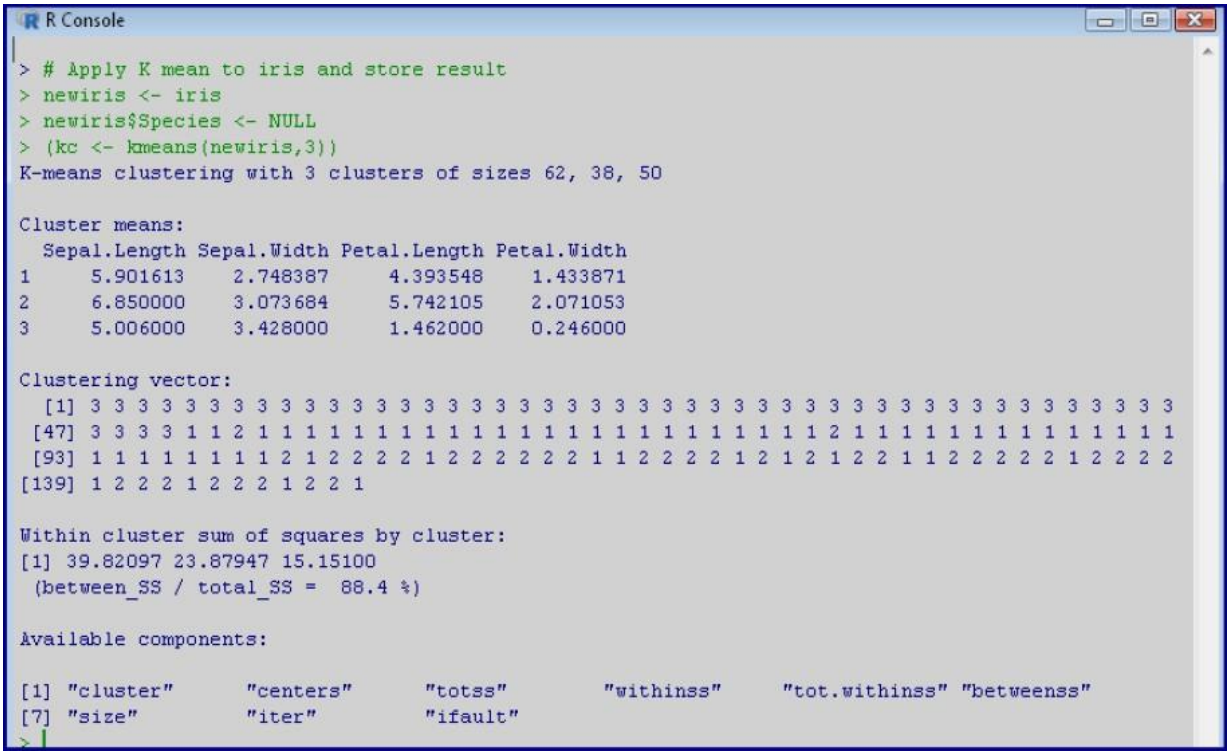
Aim: Write an application to implement clustering algorithm.

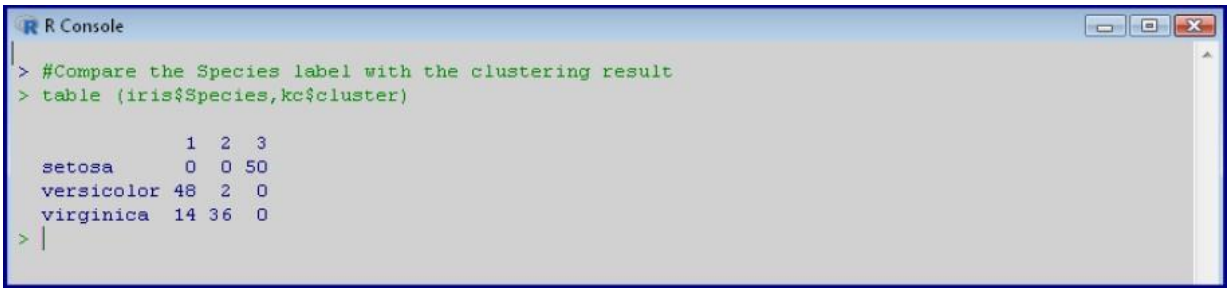
K-Means Clustering:

Code:

```
newiris <- iris
newiris$Species <- NULL
(kc <- kmeans(newiris,3))
print(kc)
# Compare the Species label with the clustering result.
table (iris$Species,kc$cluster)
plot(newiris[c("Sepal.Length","Sepal.Width")],col=kc$cluster)
points(kc$centers[,c("Sepal.Length","Sepal.Width")],col=1:3,pch=8,cex=2)
```

Output:




R Console
> #Compare the Species label with the clustering result
> table (iris$Species,kc$cluster)
1 2 3
setosa 0 0 50
versicolor 48 2 0
virginica 14 36 0
>

Practical No:06

Aim: Write an application to implement support vector machine algorithm.
Code:

Linear SVM Classifier

Let's first generate some data in 2 dimensions, and make them a little separated. After setting random seed, you make a matrix *x*, normally distributed with 20 observations in 2 classes on 2 variables. Then you make a *y* variable, which is going to be either -1 or 1, with 10 in each class. For y = 1, you move the means from 0 to 1 in each of the coordinates. Finally, you can plot the data and color code the points according to their response. The plotting character 19 gives you nice big visible dots coded blue or red according to whether the response is 1 or -1.

```
> set.seed(10111)
> x = matrix(rnorm(40), 20, 2)
> y = rep(c(-1, 1), c(10, 10))
> x[y == 1,] = x[y == 1,] + 1
> plot(x, col = y + 3, pch = 19)
```

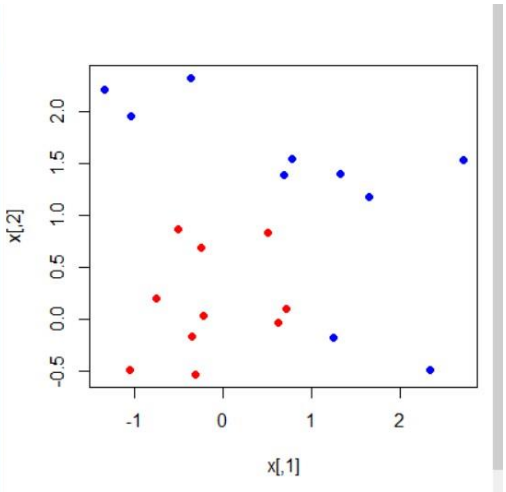
Now you load the package e1071 which contains the svm function.

```
> library(e1071)
> dat = data.frame(x, y = as.factor(y))
> svmfit = svm(y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)
```

Printing the svmfit gives its summary.

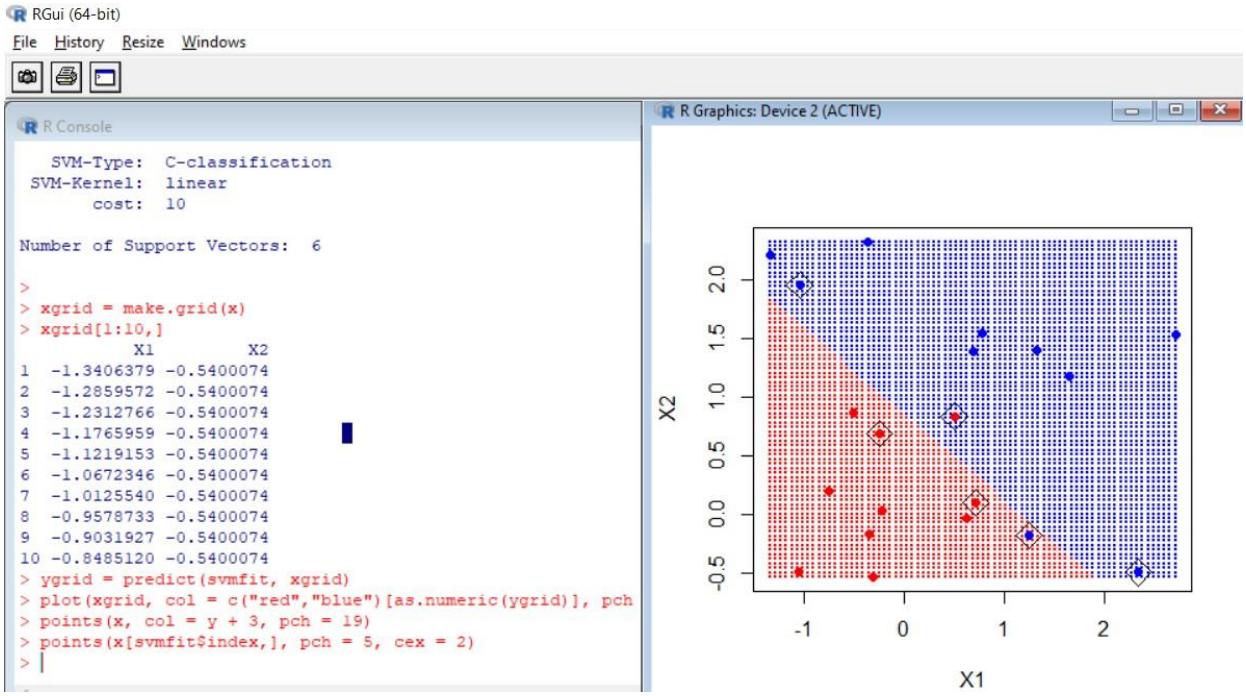
```
> print(svmfit)
```

```
> set.seed(10111)
> x = matrix(rnorm(40), 20, 2)
> y = rep(c(-1, 1), c(10, 10))
> x[y == 1,] = x[y == 1,] + 1
> plot(x, col = y + 3, pch = 19)
> library(e1071)
> dat = data.frame(x, y = as.factor(y))
> svmfit = svm(y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)
> print(svmfit)

Call:
svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  10

Number of Support Vectors:  6

> |
```




```
> plot(svmfit, dat)
```

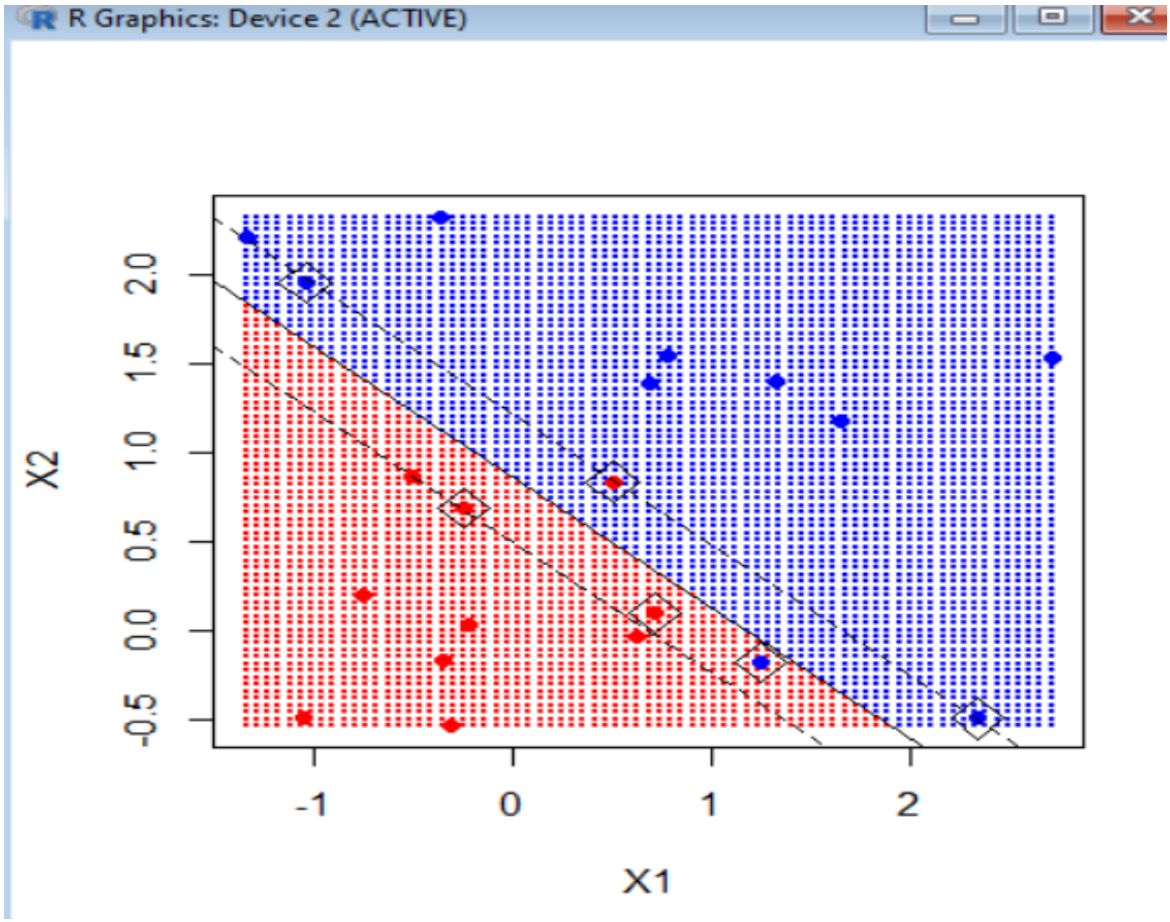
```
> make.grid = function(x, n = 75)
        {grange = apply(x, 2, range)
        x1 = seq(from = grange[1,1], to = grange[2,1], length = n)
       x2 = seq(from = grange[1,2], to = grange[2,2], length = n)
       expand.grid(X1 = x1, X2 = x2)
       }
> xgrid = make.grid(x)
> xgrid[1:10,]
> ygrid = predict(svmfit, xgrid)
> plot(xgrid, col = c("red","blue")[as.numeric(ygrid)], pch = 20, cex = .2)
> points(x, col = y + 3, pch = 19)
> points(x[svmfit$index,], pch = 5, cex = 2)
```




```
> beta = drop(t(svmfit$coefs)%*%x[svmfit$index,])
> beta0 = svmfit$rho
> plot(xgrid, col = c("red", "blue")[as.numeric(ygrid)], pch = 20, cex = .2)
> points(x, col = y + 3, pch = 19)
> points(x[svmfit$index,], pch = 5, cex = 2)
> abline(beta0 / beta[2], -beta[1] / beta[2])
> abline((beta0 - 1) / beta[2], -beta[1] / beta[2], lty = 2)
```

> abline((beta0 + 1) / beta[2], -beta[1] / beta[2], lty = 2)

Output:



Practical No:07

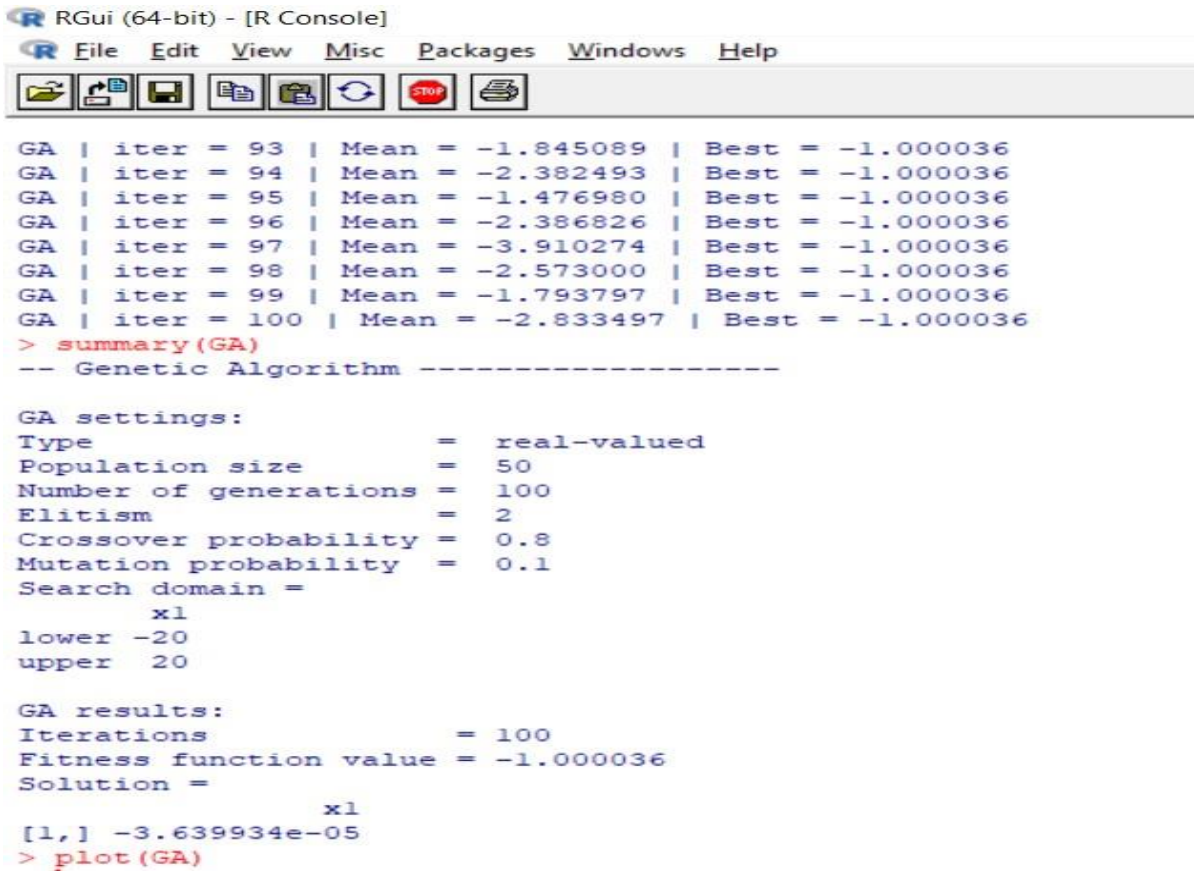
Aim: Simulate genetic algorithm with suitable example using Python / R or any other platform.

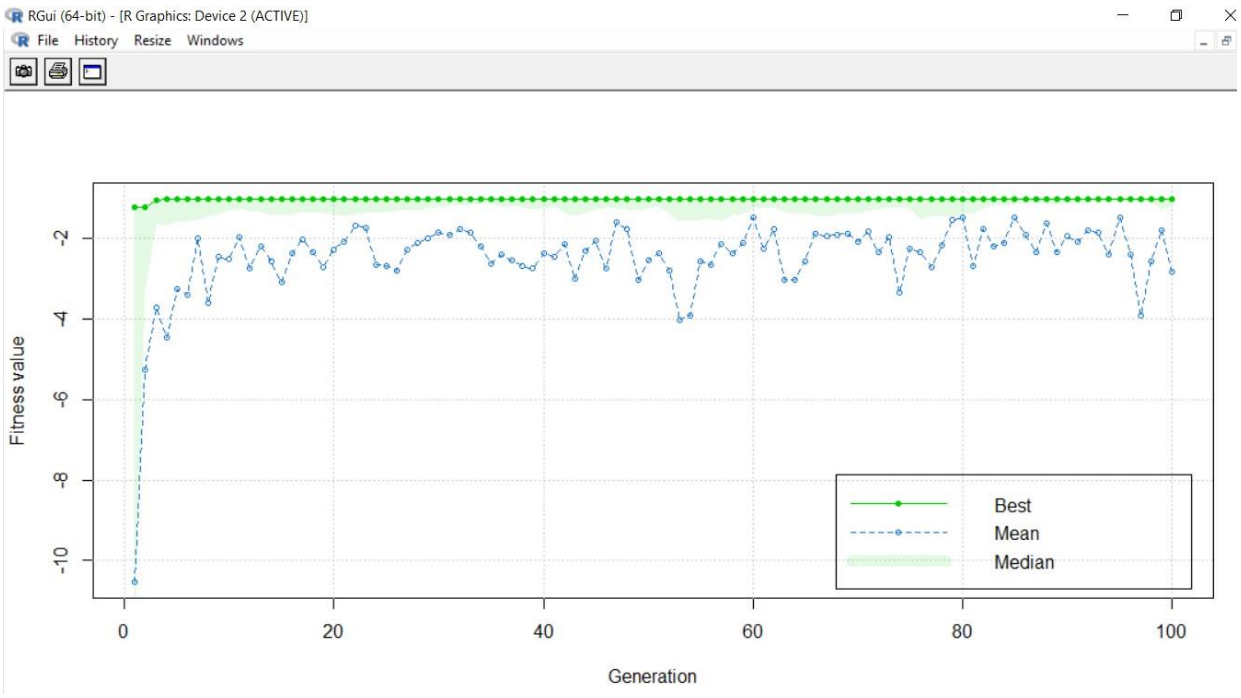
Code:

Packages: GA, Rccp, RcppArmadillo, Cli, Crayon, assertthat

```
f <- function(x) abs(x)+cos(x)
curve(f, -20, 20)
fitness <- function(x) -f(x)
GA <- ga(type = "real-valued", fitness = fitness, lower = -20, upper = 20)
summary(GA)
plot(GA)
```

Output:




RGui (64-bit) - [R Graphics: Device 2 (ACTIVE)]
File History Resize Windows
Fitness value
-2
-4
-6
-8
-10
0
20
40
60
80
100
Generation
Best
Mean
Median

Practical No:08

Aim: Design an application to simulate language parser.

Step 1: Install SLY for Python. SLY is a lexing and parsing tool which makes our process much easier.

```
pip install sly
```

Step 2: Building a Lexer

The first phase of a compiler is to convert all the character streams(the high level program that is written) to token streams. This is done by a process called lexical analysis. However, this process is simplified by using SLY.

1. **bl.py**

```
from sly import Lexer

class BasicLexer(Lexer):

    tokens = { NAME, NUMBER, STRING }

    ignore = '\t '

    literals = { '=', '+', '-', '/', '*', '(', ')', ',', ';'}

    # Define tokens as regular expressions

    # (stored as raw strings)

    NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'

    STRING = r'\".*?\"'

    # Number token

    @_(r'\d+')

    def NUMBER(self, t):

    # convert it into a python integer
```

```
        t.value = int(t.value)
        return t
    # Comment token
    @_(r'//.*')
    def COMMENT(self, t):
        pass
    # Newline token(used only for showing
    # errors in new line)
    @_(r'\n+')
    def newline(self, t):
        self.lineno = t.value.count('\n')
```

Step 3: Building a parser

2. bp.py

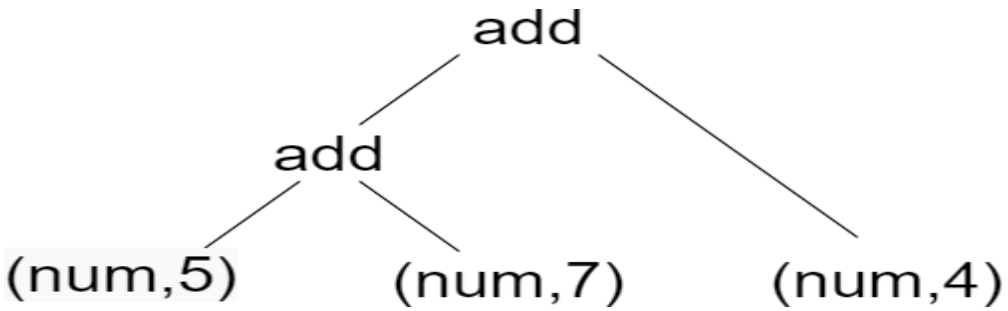
```
from sly import Parser
from bl import BasicLexer
class BasicParser(Parser):
    #tokens are passed from lexer to parser
    tokens  =  BasicLexer.tokens
    precedence = (
        ('left', '+', '-'),
        ('left', '*', '/'),
```

```
        ('right', 'UMINUS'),
    )
    def __init__(self):
        self.env = { }
    @_('')
    def statement(self, p):
        pass
    @_('var_assign')
    def statement(self, p):
        return p.var_assign
    @_('NAME "=" expr')
    def var_assign(self, p):
        return ('var_assign', p.NAME, p.expr)
    @_('NAME "=" STRING')
    def var_assign(self, p):
        return ('var_assign', p.NAME, p.STRING)
    @_('expr')
    def statement(self, p):
        return (p.expr)
    @_('expr "+" expr')
```

```
def expr(self, p):
        return ('add', p.expr0, p.expr1)
@_('expr "-" expr')
def expr(self, p):
        return ('sub', p.expr0, p.expr1)
@_('expr "*" expr')
def expr(self, p):
        return ('mul', p.expr0, p.expr1)
@_('expr "/" expr')
def expr(self, p):
        return ('div', p.expr0, p.expr1)
@_('"-" expr %prec UMINUS')
def expr(self, p):
        return p.expr
@_('NAME')
def expr(self, p):
        return ('var', p.NAME)
@_('NUMBER')
def expr(self, p):
        return ('num', p.NUMBER)
```

Step 4: Execution

Interpreting is a simple procedure. The basic idea is to take the tree and walk through it to and evaluate arithmetic operations hierarchically. This process is recursively called over and over again till the entire tree is evaluated and the answer is retrieved. Let's say, for example, 5 + 7 + 4. This character stream is first tokenized to token stream in a lexer. The token stream is then parsed to form a parse tree. The parse tree essentially returns ('add', ('add', ('num', 5), ('num', 7)), ('num', 4)). (see image below).



The interpreter is going to add 5 and 7 first and then recursively call *walkTree* and add 4 to the result of addition of 5 and 7. Thus, we are going to get 16. The below code does the same process.

3. exec.py

```
from bl import BasicLexer

from bp import BasicParser

class BasicExecute:

    def__init__(self, tree, env):

        self.env = env

        result = self.walkTree(tree)

        if result is not None and isinstance(result, int):

            print(result)
```

```
        if isinstance(result, str) and result[0] == "":
                print(result)
    def walkTree(self, node):
        if isinstance(node, int):
                return node
        if isinstance(node, str):
                return node
        if node is None:
                return None
        if node[0] == 'program':
                if node[1] == None:
                        self.walkTree(node[2])
                else:
                        self.walkTree(node[1])
                        self.walkTree(node[2])
        if node[0] == 'num':
                return node[1]
        if node[0] == 'str':
                return node[1]
```

```
            if node[0] == 'add':
                return self.walkTree(node[1]) + self.walkTree(node[2])
            elif node[0] == 'sub':
                return self.walkTree(node[1]) - self.walkTree(node[2])
            elif node[0] == 'mul':
                return self.walkTree(node[1]) * self.walkTree(node[2])
            elif node[0] == 'div':
                return self.walkTree(node[1]) / self.walkTree(node[2])
            if node[0] == 'var_assign':
                self.env[node[1]] = self.walkTree(node[2])
                return node[1]
            if node[0] == 'var':
                try:
                    return self.env[node[1]]
                except LookupError:
                    print("Undefined variable '"+node[1]+"' found!")
                    return 0
if__name__ == '__main__':
    lexer = BasicLexer()
    parser = BasicParser()
```

```
print('GFG Language')
env = {}
while True:
    try:
        text = input('GFG Language > ')
    except EOFError:
        break
    if text:
        tree = parser.parse(lexer.tokenize(text))
        BasicExecute(tree, env)
```

Output:

```
=== RESTART: C:/Users/Anam/AppData/Local/Programs/Python/Python37/exec.py ===
GFG Language
GFG Language > a=10
GFG Language > b=15
GFG Language > c=a+b
GFG Language > c
25
GFG Language > c="Hello GFG"
GFG Language > c
"Hello GFG"
GFG Language >
```