1. The probability that it is Friday and that a student is absent is 3 %. Since there are 5 school days in a week, the probability that it is Friday is 20 %. What is theprobability that a student is absent given that today is Friday? Apply Baye's rule in python to get the result. (Ans: 15%)

```
probAbsentFriday=0.03

probFriday=0.2

# bayes Formula

#p(Absent|Friday)=p(Friday|Absent)p(Absent)/p(Friday)

#p(Friday|Abscnt)=p(Friday∩Abscnt)/p(Abscnt)

# Therefore the result is:

bayesResult=(probAbsentFriday/probFriday)

print(bayesResult * 100)
```

Output: 15

```
: probAbsentFriday=0.03
  probFriday=0.2
  # bayes Formula
  #p(Absent|Friday)=p(Friday|Absent)p(Absent)/p(Friday)
  #p(Friday|Absent)=p(FridaynAbsent)/p(Absent)
  # Therefore the result is:
  bayesResult=(probAbsentFriday/probFriday)
  print(bayesResult * 100)
```

```
15.0
```



```
import sqlite3

#function to create the tabble
def create_table():
        #Step 1: to create the connection object
        conn=sqlite3.connect("Emplite.db")
        #Step 2: to create the cursor object
        cur=conn.cursor()
        #Step 3: to execute the CREATE query
        cur.execute("CREATE TABLE IF NOT EXISTS employee(eid TEXT,ename TEXT,esal REAL)")
        #Step 4: to commit the changes
        conn.commit()
        #Step 5: to close the connection
        conn.close()

#call  to the create_table function
create_table()

# function to insert the values in the table
def insert(eid,ename,esal):
        #Step 1: to create the connection object
        conn=sqlite3.connect("Emplite.db")
        #Step 2: to create the cursor object
        cur=conn.cursor()
        #Step 3: to execute the INSERT query
        cur.execute("INSERT INTO employee VALUES(?,?,?)",(eid,ename,esal))
        #Step 4: to commit the changes
        conn.commit()
        #Step 5: to close the connection
        conn.close()

# call to insert function
insert("e02","Ramesh",500000)
insert("e03","Somesh",100000)

#function to SELECT rows from the table
def view():
        #Step 1: to create the connection object
        conn=sqlite3.connect("Emplite.db")
        #Step 2: to create the cursor object
        cur=conn.cursor()
        #Step 3: to execute the SELECT query
        cur.execute("SELECT *FROM employee")
        #Step 4: to fetch the rows from table
        rows=cur.fetchall()
        #Step 5: to close the connection
        conn.close()
        #Step 6: to print the data in the table
        for i in rows:
                print ("Employee ID: ", i[0])
                print ("Employee name: ", i[1])
                print ("Salary: ", i[2])

#call to view function
view()
```

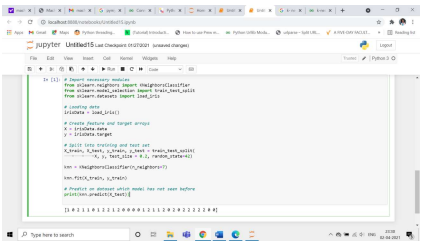
Output:

```
Employee ID:  e02
Employee name:  Ramesh
Salary:  500000.0
Employee ID:  e03
Employee name:  Somesh
Salary:  100000.0
```


3. Implement k-nearest neighbours classification using python

- The **k-nearest neighbor algorithm** is imported from the scikit-learn package.
- Create feature and target variables.
- Split data into training and test data.
- Generate a **k-NN** model using **neighbors** value.
- Train or fit the data into the model.
- Predict the future.

```
# Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# Loading data
irisData = load_iris()

# Create feature and target arrays
X = irisData.data
y = irisData.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(
                        X, y, test_size = 0.2, random_state=42)

knn = KNeighborsClassifier(n_neighbors=7)

knn.fit(X_train, y_train)

# Predict on dataset which model has not seen before
print(knn.predict(X_test))
```

4. Given the following data, which specify classifications for nine combinations of VAR1 and VAR2 predict a classification for a case where VAR1=0.906 and VAR2=0.606, using the result of kmeans clustering with 3 means (i.e., 3 centroids)

VAR1	VAR2	CLASS
1.713	1.586	0
0.180	1.786	1
0.353	1.240	1
0.940	1.566	0
1.486	0.759	1
1.266	1.106	0
1.540	0.419	1
0.459	1.799	1
0.773	0.186	1

```
from sklearn.cluster import KMeans

import numpy as np

X = np.array([[1.713,1.586], [0.180,1.786], [0.353,1.240],

        [0.940,1.566], [1.486,0.759], [1.266,1.106],[1.540,0.419],[0.459,1.799],[0.773,0.186]])

y=np.array([0,1,1,0,1,0,1,1,1])

kmeans = KMeans(n_clusters=3, random_state=0).fit(X,y)

kmeans.predict([[0.906, 0.606]])
```


5. The following training examples map descriptions of individuals onto high, medium and low credit-worthiness.

medium skiing design single twenties no -> highRisk

high golf trading married forties yes -> lowRisk

low speedway transport married thirties yes -> medRisk

medium football banking single thirties yes -> lowRisk

high flying media married fifties yes -> highRisk

low football security single twenties no -> medRisk

medium golf media single thirties yes -> medRisk

medium golf transport married forties yes -> lowRisk

high skiing banking single thirties yes -> highRisk

low golf unemployed married forties yes -> highRisk

Input attributes are (from left to right) income, recreation, job, status, age-group, home-owner. Find the unconditional probability of `golf' and the conditional probability of `single' given `medRisk' in the dataset?

```
totalRecords=10

numberGolfRecreation=4

probGolf=numberGolfRecreation/totalRecords

print("Unconditional probability of golf: = {}".format(probGolf))

#conditional probability of `single' given `medRisk'

# bayes Formula

#p(sing|medRisk)=p(medRisk|single)p(single)/p(medRisk)

#p(medRisk|single)=p(medRisk ∩ single)/p(single)

# Therefore the result is:

numberMedRiskSingle=2

numberMedRisk=3
```



```
probMedRiskSingle=numberMedRiskSingle/totalRecords

probMedRisk=numberMedRisk/totalRecords

conditionalProbability=(probMedRiskSingle/probMedRisk)

print("Conditional probability of single given medRisk: = {}".format(conditionalProbability))
```

Output:

```
Unconditional probability of golf: = 0.4

    Conditional probability of single given medRisk: = 0.6666666666666667
```


6. Implement linear regression using python.

Regression
Regression analysis is one of the most important fields in statistics and machine learning. There are many regression methods available. Linear regression is one of them

What Is Regression?
Regression analysis is one of the most important fields in statistics and machine learning. There are many regression methods available. Linear regression is one of them.

Regression searches for relationships among variables.

For example, you can observe several employees of some company and try to understand how their salaries depend on the **features**, such as experience, level of education, role, city they work in, and so on.

This is a regression problem where data related to each employee represent one **observation**. The presumption is that the experience, education, role, and city are the independent features, while the salary depends on them.

Generally, in regression analysis, you usually consider some phenomenon of interest and have a number of observations. Each observation has two or more features. Following the assumption that (at least) one of the features depends on the others, you try to establish a relation among them.

you need to find a function that maps some features or variables to others sufficiently well.

The dependent features are called the **dependent variables**, **outputs**, or **responses**.

The independent features are called the **independent variables**, **inputs**, or **predictors**.

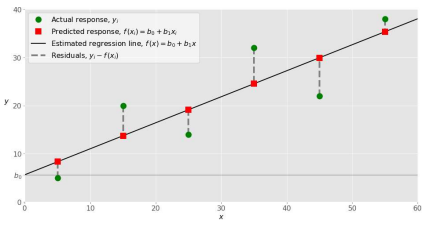
Linear Regression
Linear regression is probably one of the most important and widely used regression techniques. It's among the simplest regression methods. One of its main advantages is the ease of interpreting results.

When implementing linear regression of some dependent variable y on the set of independent variables $\mathbf{x} = (x_1, \ldots, x_r)$, where r is the number of predictors, you assume a linear relationship between y and $\mathbf{x}$: $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_r x_r + \varepsilon$. This equation is the **regression equation**. β_0, β_1, …, β_r are the **regression coefficients**, and ε is the **random error**.

Linear regression calculates the **estimators** of the regression coefficients or simply the **predicted weights**, denoted with $b_0, b_1, \ldots, b_r$. They define the **estimated regression function** $f(\mathbf{x}) = b_0 + b_1 x_1 + \cdots + b_r x_r$. This function should capture the dependencies between the inputs and output sufficiently well.

Simple Linear Regression

The following figure illustrates simple linear regression:



When implementing simple linear regression, you typically start with a given set of input-output (x-y) pairs (green circles). These pairs are your observations. For example, the leftmost observation (green circle) has the input $x = 5$ and the actual output (response) $y = 5$. The next one has $x = 15$ and $y = 20$, and so on.

The estimated regression function (black line) has the equation $f(x) = b_0 + b_1x$. Your goal is to calculate the optimal values of the predicted weights b_0 and b_1 that minimize SSR and determine the estimated regression function. The value of b_0, also called the **intercept**, shows the point where the estimated regression line crosses the y axis. It is the value of the estimated response $f(x)$ for $x = 0$. The value of b_1 determines the **slope** of the estimated regression line.

The predicted responses (red squares) are the points on the regression line that correspond to the input values. For example, for the input $x = 5$, the predicted response is $f(5) = 8.33$ (represented with the leftmost red square).

The residuals (vertical dashed gray lines) can be calculated as $y_i - f(x_i) = y_i - b_0 - b_1x_i$ for $i = 1, \ldots, n$. They are the distances between the green circles and red squares. When you implement linear regression, you are actually trying to minimize these distances and make the red squares as close to the predefined green circles as possible.

Implementing Linear Regression in Python

It's time to start implementing linear regression in Python. Basically, all you should do is apply the proper packages and their functions and classes.

Python Packages for Linear Regression

The package **NumPy** is a fundamental Python scientific package that allows many high-performance operations on single- and multi-dimensional arrays. It also offers many mathematical routines. Of course, it's open source.

The package **scikit-learn** is a widely used Python library for machine learning, built on top of NumPy and some other packages. It provides the means for preprocessing data, reducing dimensionality, implementing regression, classification, clustering, and more. Like NumPy, scikit-learn is also open source.

If you want to implement linear regression and need the functionality beyond the scope of scikit-learn, you should consider **statsmodels**. It's a powerful Python package for the estimation of statistical models, performing tests, and more. It's open source as well.

Simple Linear Regression With scikit-learn

Let's start with the simplest case, which is simple linear regression.

There are five basic steps when you're implementing linear regression:

1. Import the packages and classes you need.
2. Provide data to work with and eventually do appropriate transformations.
3. Create a regression model and fit it with existing data.
4. Check the results of model fitting to know whether the model is satisfactory.
5. Apply the model for predictions.

```
import numpy as np

import matplotlib.pyplot as plt


def estimate_coef(x, y):

    # number of observations/points

    n = np.size(x)


    # mean of x and y vector
```



```
    m_x, m_y = np.mean(x), np.mean(y)


    # calculating cross-deviation and deviation about x

    SS_xy = np.sum(y*x) - n*m_y*m_x

    SS_xx = np.sum(x*x) - n*m_x*m_x


    # calculating regression coefficients

    b_1 = SS_xy / SS_xx

    b_0 = m_y - b_1*m_x


    return(b_0, b_1)


def plot_regression_line(x, y, b):

    # plotting the actual points as scatter plot

    plt.scatter(x, y, color = "m",

                marker = "o", s = 30)


    # predicted response vector

    y_pred = b[0] + b[1]*x


    # plotting the regression line

    plt.plot(x, y_pred, color = "g")


    # putting labels
```



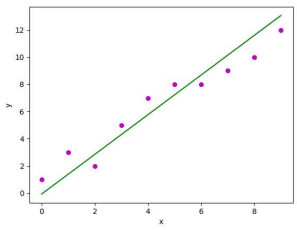
```
    plt.xlabel('x')

    plt.ylabel('y')


    # function to show plot

    plt.show()


def main():

    # observations

    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])


    # estimating coefficients

    b = estimate_coef(x, y)

    print("Estimated coefficients:\nb_0 = {} \

          \nb_1 = {}".format(b[0], b[1]))


    # plotting regression line

    plot_regression_line(x, y, b)


if __name__ == "__main__":

    main()
```

Output:

```
Estimated coefficients:
b_0 = -0.0586206896552
```



```
b_1 = 1.45747126437
```



7. Implement Naïve Bayes theorem to classify the English text

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score


msg = pd.read_csv('document.csv', names=['message', 'label'])
print("Total Instances of Dataset: ", msg.shape[0])
msg['labelnum'] = msg.label.map({'pos': 1, 'neg': 0})


X = msg.message
y = msg.labelnum


Xtrain, Xtest, ytrain, ytest = train_test_split(X, y)


count_v = CountVectorizer()
Xtrain_dm = count_v.fit_transform(Xtrain)
Xtest_dm = count_v.transform(Xtest)


df = pd.DataFrame(Xtrain_dm.toarray(),columns=count_v.get_feature_names())


clf = MultinomialNB()
clf.fit(Xtrain_dm, ytrain)
```



```
pred = clf.predict(Xtest_dm)


print('Accuracy Metrics:')
print('Accuracy: ', accuracy_score(ytest, pred))
print('Recall: ', recall_score(ytest, pred))
print('Precision: ', precision_score(ytest, pred))
print('Confusion Matrix: \n', confusion_matrix(ytest, pred))
```

document.csv:

```
I love this sandwich,pos
This is an amazing place,pos
I feel very good about these beers,pos
This is my best work,pos
What an awesome view,pos
I do not like this restaurant,neg
I am tired of this stuff,neg
I can't deal with this,neg
He is my sworn enemy,neg
My boss is horrible,neg
This is an awesome place,pos
I do not like the taste of this juice,neg
I love to dance,pos
I am sick and tired of this place,neg
What a great holiday,pos
That is a bad locality to stay,neg
```



```
We will have good fun tomorrow,pos
I went to my enemy's house today,neg
```

Output:

```
Total Instances of Dataset: 18
Accuracy Metrics:
Accuracy: 0.6
Recall: 0.6666666666666666
Precision: 0.6666666666666666
Confusion Matrix:
    [[1 1]
     [1 2]]
```


8. Implement an algorithm to demonstrate the significance of genetic algorithm

```
import numpy


def cal_pop_fitness(equation_inputs, pop):
    # Calculating the fitness value of each solution in the current population.
    # The fitness function calulates the sum of products between each input and its corresponding weight.
    fitness = numpy.sum(pop*equation_inputs, axis=1)
    return fitness


def select_mating_pool(pop, fitness, num_parents):
    # Selecting the best individuals in the current generation as parents for producing the offspring of the next generation.
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        max_fitness_idx = numpy.where(fitness == numpy.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]
        parents[parent_num, :] = pop[max_fitness_idx, :]
        fitness[max_fitness_idx] = -99999999999
    return parents


def crossover(parents, offspring_size):
    offspring = numpy.empty(offspring_size)
    # The point at which crossover takes place between two parents. Usually, it is at the center.
    crossover_point = numpy.uint8(offspring_size[1]/2)
```

```
    for k in range(offspring_size[0]):
        # Index of the first parent to mate.
        parent1_idx = k%parents.shape[0]
        # Index of the second parent to mate.
        parent2_idx = (k+1)%parents.shape[0]
        # The new offspring will have its first half of its genes taken from the first parent.
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
        # The new offspring will have its second half of its genes taken from the second parent.
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring


def mutation(offspring_crossover, num_mutations=1):
    mutations_counter = numpy.uint8(offspring_crossover.shape[1] / num_mutations)
    # Mutation changes a number of genes as defined by the num_mutations argument. The changes are random.
    for idx in range(offspring_crossover.shape[0]):
        gene_idx = mutations_counter - 1
        for mutation_num in range(num_mutations):
            # The random value to be added to the gene.
            random_value = numpy.random.uniform(-1.0, 1.0, 1)
            offspring_crossover[idx, gene_idx] = offspring_crossover[idx, gene_idx] + random_value
            gene_idx = gene_idx + mutations_counter
    return offspring_crossover
```



```
import numpy

"""
The y=target is to maximize this equation ASAP:
    y = w1x1+w2x2+w3x3+w4x4+w5x5+6wx6
    where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7)
    What are the best values for the 6 weights w1 to w6?
    We are going to use the genetic algorithm for the best possible values after a number of generations.
"""

# Inputs of the equation.
equation_inputs = [4,-2,3.5,5,-11,-4.7]

# Number of the weights we are looking to optimize.
num_weights = len(equation_inputs)

"""
Genetic algorithm parameters:
    Mating pool size
    Population size
"""
sol_per_pop = 8
num_parents_mating = 4
```



```
# Defining the population size.
pop_size = (sol_per_pop,num_weights) # The population will have sol_per_pop chromosome where each chromosome has num_weights genes.
#Creating the initial population.
new_population = numpy.random.uniform(low=-4.0, high=4.0, size=pop_size)
print(new_population)

"""
new_population[0, :] = [2.4,  0.7, 8, -2,   5,   1.1]
new_population[1, :] = [-0.4, 2.7, 5, -1,   7,   0.1]
new_population[2, :] = [-1,   2,   2, -3,   2,   0.9]
new_population[3, :] = [4,    7,   12, 6.1, 1.4, -4]
new_population[4, :] = [3.1,  4,   0,  2.4, 4.8,  0]
new_population[5, :] = [-2,   3,   -7, 6,   3,    3]
"""

best_outputs = []
num_generations = 1000
for generation in range(num_generations):
    print("Generation : ", generation)
    # Measuring the fitness of each chromosome in the population.
    fitness = cal_pop_fitness(equation_inputs, new_population)
    print("Fitness")
    print(fitness)
```



```
    best_outputs.append(numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))
    # The best result in the current iteration.
    print("Best result : ", numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))

    # Selecting the best parents in the population for mating.
    parents = select_mating_pool(new_population, fitness,
                                 num_parents_mating)
    print("Parents")
    print(parents)

    # Generating next generation using crossover.
    offspring_crossover = crossover(parents,
                                    offspring_size=(pop_size[0]-parents.shape[0], num_weights))
    print("Crossover")
    print(offspring_crossover)

    # Adding some variations to the offspring using mutation.
    offspring_mutation = mutation(offspring_crossover, num_mutations=2)
    print("Mutation")
    print(offspring_mutation)

    # Creating the new population based on the parents and offspring.
    new_population[0:parents.shape[0], :] = parents
    new_population[parents.shape[0]:, :] = offspring_mutation
```



```
# Getting the best solution after iterating finishing all generations.
#At first, the fitness is calculated for each solution in the final generation.
fitness = cal_pop_fitness(equation_inputs, new_population)
# Then return the index of that solution corresponding to the best fitness.
best_match_idx = numpy.where(fitness == numpy.max(fitness))

print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])

import matplotlib.pyplot
matplotlib.pyplot.plot(best_outputs)
matplotlib.pyplot.xlabel("Iteration")
matplotlib.pyplot.ylabel("Fitness")
matplotlib.pyplot.show()
```

Output:

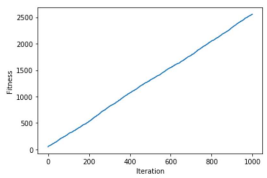
```
[[ 0.58204141 2.32880696 -2.95130209 2.57056953 3.33055238 -0.5816787] [-1.65052225 3.52263842 -2.46577305 -1.7005396 -3.80480202 0.29677167] [ 2.6239874 -2.01548549 -1.72292295 3.61090243 -1.25604726 -2.32647264] [-3.45167393 2.85771825 3.74655682 -2.01790626 0.25750106 -3.12923247] [ 2.86026334 -0.4306777 -3.26297956 1.74863348 -1.93705571 -3.18855672] [-1.70012089 0.98685104 -1.91192072 3.91873942 -0.09354385 1.43038667] [ 0.31769009 -0.87290809 3.75249785 2.57657993 0.58883082 2.83231871] [ 3.83314926 0.33838112 -2.49509594 -1.50763174 3.99440509 -0.03037715]]

Generation : 0

Fitness

[-33.70834413 9.67772594 51.30214363 -4.62383365 45.91897711 -1.56604606 9.24418172 -45.41084308]
```



```
Best result : 51.302143629097614

Parents

[[ 2.6239874 -2.01548549 -1.72292295 3.61090243 -1.25604726 -2.32647264] [ 2.86026334 -0.4306777 -3.26297956 1.74863348 -1.93705571 -3.18855672] [-1.65052225 3.52263842 -2.46577305 -1.7005396 -3.80480202 0.29677167] [ 0.31769009 -0.87290809 3.75249785 2.57657993 0.58883082 2.83231871]]

Crossover

 [[ 2.6239874 -2.01548549 -1.72292295 1.74863348 -1.93705571 -3.18855672] [ 2.86026334 -0.4306777 -3.26297956 -1.7005396 -3.80480202 0.29677167] [-1.65052225 3.52263842 -2.46577305 2.57657993 0.58883082 2.83231871] [ 0.31769009 -0.87290809 3.75249785 3.61090243 -1.25604726 -2.32647264]]

Mutation

 [[ 2.6239874 -2.01548549 -1.67896632 1.74863348 -1.93705571 -3.97789372] [ 2.86026334 -0.4306777 -3.12878279 -1.7005396 -3.80480202 -0.15430324] [-1.65052225 3.52263842 -3.37669601 2.57657993 0.58883082 2.25153466] [ 0.31769009 -0.87290809 2.93428907 3.61090243 -1.25604726 -2.71597954]]

.
.
.
.

Generation : 999

Fitness

[2554.3935562 2551.72360738 2549.40583954 2549.29931629 2552.24225166 2550.45506206 2547.1299512 2551.22467397]

Best result : 2554.3935561987346

Parents

[[ 3.17690088e-01 -8.72908094e-01 2.67689952e+02 1.74863348e+00 -1.93705571e+00 -3.37108802e+02] [ 3.17690088e-01 -8.72908094e-01 2.67638232e+02 1.74863348e+00 -1.93705571e+00 -3.36689592e+02] [ 3.17690088e-01 -8.72908094e-01 2.67254110e+02
```



```
1.74863348e+00 -1.93705571e+00 -3.36865291e+02] [ 3.17690088e-01 -8.72908094e-01 2.67370854e+02 1.74863348e+00 -1.93705571e+00 -3.36672197e+02]]

Crossover

[[ 3.17690088e-01 -8.72908094e-01 2.67689952e+02 1.74863348e+00 -1.93705571e+00 -3.36689592e+02]

[ 3.17690088e-01 -8.72908094e-01 2.67638232e+02 1.74863348e+00

-1.93705571e+00 -3.36865291e+02]

 [ 3.17690088e-01 -8.72908094e-01  2.67254110e+02  1.74863348e+00

 -1.93705571e+00 -3.36672197e+02]

 [ 3.17690088e-01 -8.72908094e-01  2.67370854e+02  1.74863348e+00

 -1.93705571e+00 -3.37108802e+02]]

Mutation

[[ 3.17690088e-01 -8.72908094e-01  2.68382875e+02  1.74863348e+00

 -1.93705571e+00 -3.36222272e+02]

 [ 3.17690088e-01 -8.72908094e-01  2.68456819e+02  1.74863348e+00

 -1.93705571e+00 -3.37417363e+02]

 [ 3.17690088e-01 -8.72908094e-01  2.67606746e+02  1.74863348e+00

 -1.93705571e+00 -3.36866918e+02]

 [ 3.17690088e-01 -8.72908094e-01  2.67051753e+02  1.74863348e+00

 -1.93705571e+00 -3.37331663e+02]]

Best solution :  [[[ 3.17690088e-01 -8.72908094e-01  2.68456819e+02  1.74863348e+00

  -1.93705571e+00 -3.37417363e+02]]]

Best solution fitness :  [2558.52782726]
```


```
import numpy as np

#Training Examples
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) #maximum of X array longitudinally
y = y/100

#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)

#Variable initialization
epoch=5 #Setting training iterations
lr=0.1 #Setting learning rate

#no of neurons in input output and hidden layerintiallization
inputlayer_neurons = 2 #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer

#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))

#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
    #Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+bout
    output = sigmoid(outinp)

    #Backpropagation
```

```
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO * outgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer wts contributed to error
    d_hiddenlayer = EH * hiddengrad

    wout += hlayer_act.T.dot(d_output) *lr   # dotproduct of nextlayererror and currentlayerop
    wh += X.T.dot(d_hiddenlayer) *lr

    print ("----------Epoch-", i+1, "Starts----------")
    print("Input: \n" + str(X))
    print("Actual Output: \n" + str(y))
    print("Predicted Output: \n" ,output)
    print ("----------Epoch-", i+1, "Ends----------\n")

  print("Input: \n" + str(X))
  print("Actual Output: \n" + str(y))
  print("Predicted Output: \n" ,output)

Output:

----------Epoch- 1 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.70605729]
 [0.6984563 ]
 [0.70378783]]
----------Epoch- 1 Ends----------

----------Epoch- 2 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.71020912]
 [0.70236232]
 [0.70781902]]
----------Epoch- 2 Ends----------

----------Epoch- 3 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.71420845]
 [0.70612764]
 [0.71170368]]
----------Epoch- 3 Ends----------

----------Epoch- 4 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.71806277]
 [0.70975913]
 [0.71544899]]
----------Epoch- 4 Ends----------

----------Epoch- 5 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.72177921]
 [0.71326327]
 [0.71906174]]
----------Epoch- 5 Ends----------

Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.72177921]
 [0.71326327]
 [0.71906174]]
```

FIND S ALGORITHM

```
import pandas as pd
import numpy as np

#to read the data in the csv file
data = pd.read_csv("data.csv")
print(data,"\n")

#making an array of all the attributes
d = np.array(data)[:,:-1]
print("\n The attributes are: ",d)

#segregating the target that has positive and negative examples
target = np.array(data)[:,-1]
print("\n The target is: ",target)

#training function to implement find-s algorithm
def train(c,t):
    for i, val in enumerate(t):
        if val == "Yes":
            specific_hypothesis = c[i].copy()
            break

    for i, val in enumerate(c):
        if t[i] == "Yes":
            for x in range(len(specific_hypothesis)):
                if val[x] != specific_hypothesis[x]:
                    specific_hypothesis[x] = '?'
                else:
                    pass

    return specific_hypothesis

#obtaining the final hypothesis
print("\n The final hypothesis is:",train(d,target))
```