# Practical No.1

**Aim:** Implementation of logic Programming using PROLOG DFS for water jug problem.

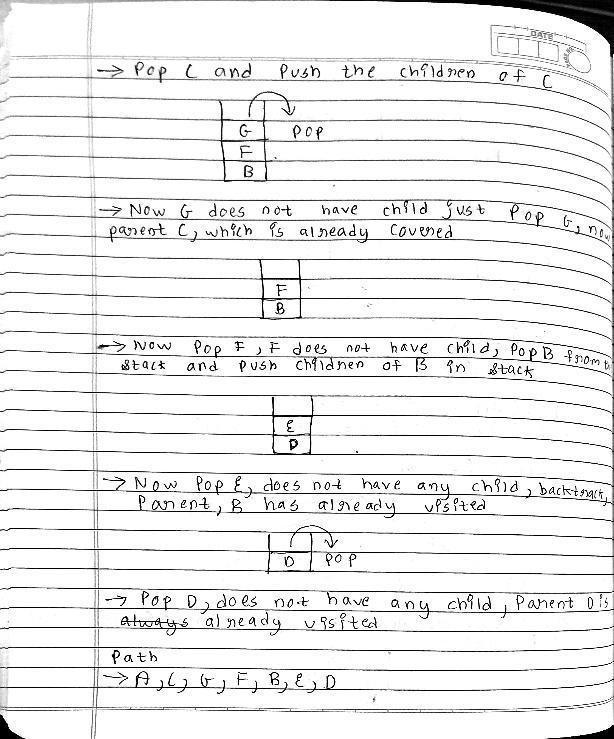
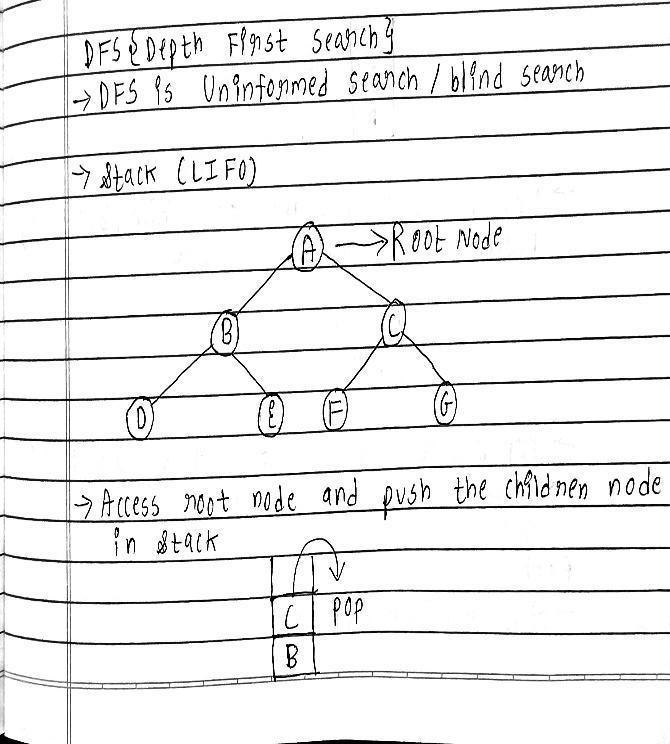
#### Objectives:

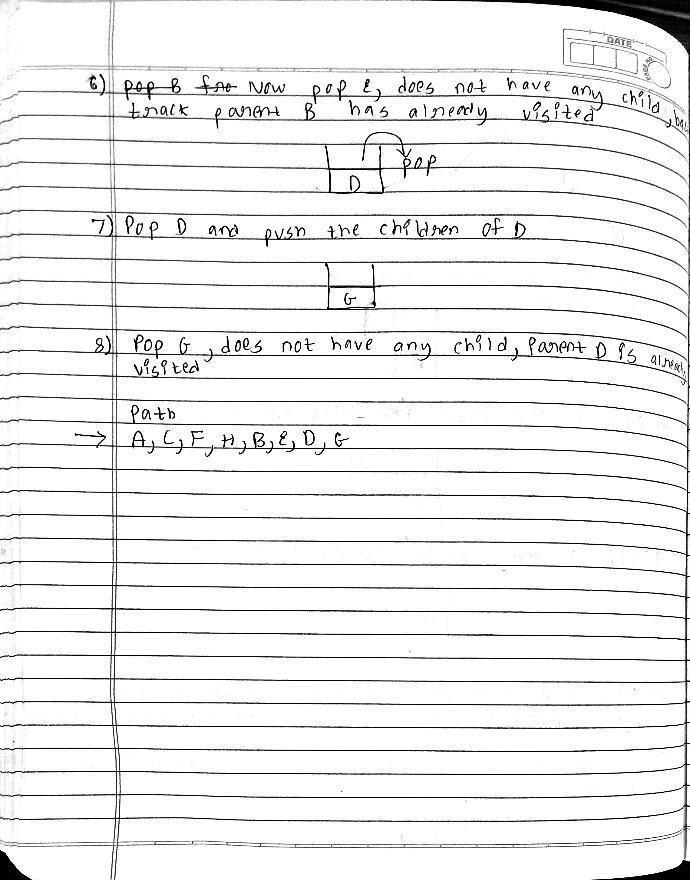
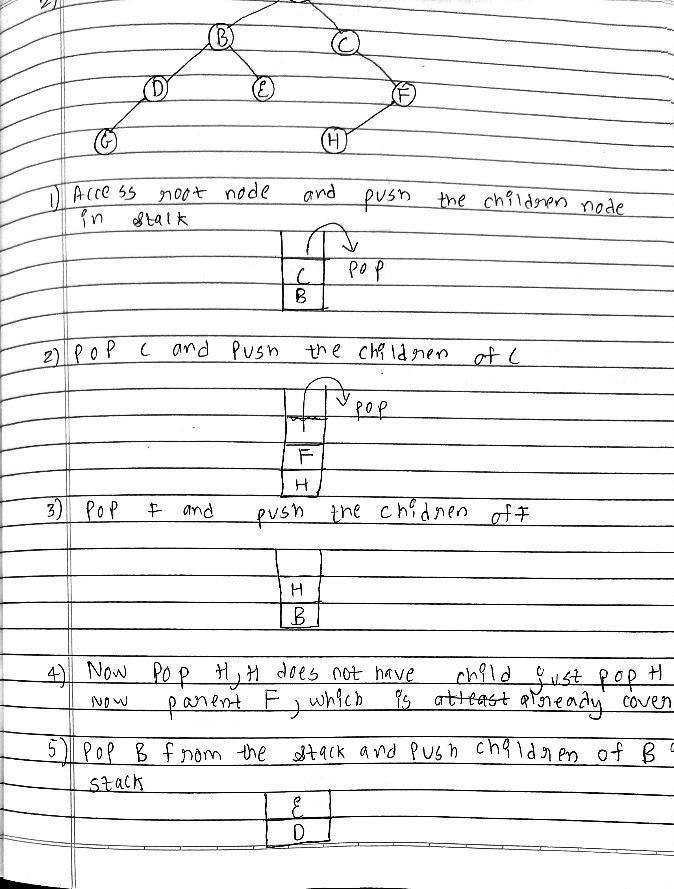
* Understanding of the DFS algorithm.
* Understanding of water-jug problem
* Analysis of water-jug Problem by using DFS.
* Implementation of DFS to solve the water jug problem.

**Theory:**



**Problems based on DFS:**





**Code:**

start(2,0):-write(' 4lit Jug: 2 | 3lit Jug: 0|\n'),

write('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n'),

write('Goal Reached! Congrats!!\n'), write('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n').

start(X,Y):-write(' 4lit Jug: '),write(X),write('| 3lit Jug: '), write(Y),write('|\n'),

write(' Enter the move::'), read(N), contains(X,Y,N).

contains(\_,Y,1):-start(4,Y).

contains(X,\_,2):-start(X,3).

contains(\_,Y,3):-start(0,Y).

contains(X,\_,4):-start(X,0).

contains(X,Y,5):-N is Y-4+X, start(4,N).

contains(X,Y,6):-N is X-3+Y, start(N,3).

contains(X,Y,7):-N is X+Y, start(N,0).

contains(X,Y,8):-N is X+Y, start(0,N).

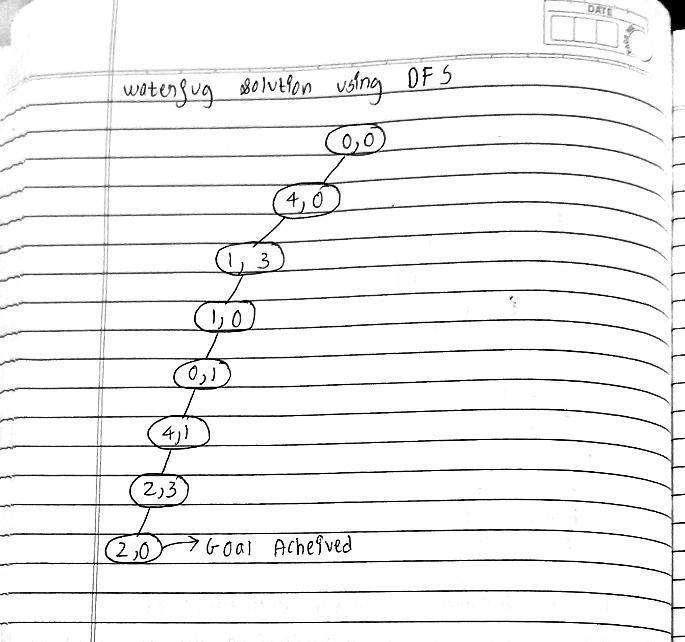
main():-write(' Water Jug Game \n'), write('Intial State: 4lit Jug- 0lit\n'), write(' 3lit Jug- 0lit\n'), write('Final State: 4lit Jug- 2lit\n'), write(' 3lit Jug- 0lit\n'), write('Follow the Rules: \n'), write('Rule 1: Fill 4lit Jug\n'), write('Rule 2: Fill 3lit Jug\n'), write('Rule 3: Empty 4lit Jug\n'), write('Rule 4: Empty 3lit Jug\n'),

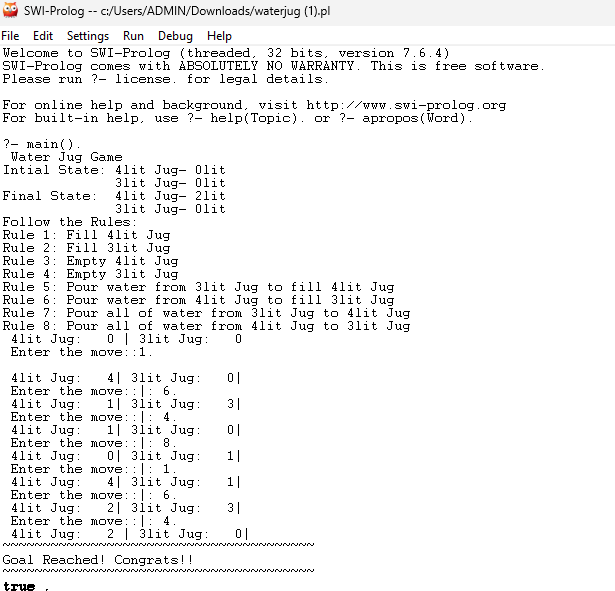
write('Rule 5: Pour water from 3lit Jug to fill 4lit Jug\n'), write('Rule 6: Pour water from 4lit Jug to fill 3lit Jug\n'), write('Rule 7: Pour all of water from 3lit Jug to 4lit Jug\n'), write('Rule 8: Pour all of water from 4lit Jug to 3lit Jug\n'), write(' 4lit Jug: 0 | 3lit Jug: 0'),nl,

write(' Enter the move::'), read(N),nl,

contains(0,0,N).

**Output:**





# Practical No.2

### Program no. 1:

**Aim:-** Design and solve Tic-Tac-Toe using BFS

**Objective** ➖

* Understand Tic-Tac-Toe game Logic.
* Understand BFS Algorithm
* Solving Tic-Tac-Toe using BFS.
* Implementation of BFS for Tic-Tac-Toe

**Theory:-**

Tic-Tac-Toe is a 2-player game played on a 3x3 grid. Players take turns marking empty spaces with "X" or "O." The goal is to get three of your marks in a row, either horizontally, vertically, or diagonally. The first to do so wins, or the game ends in a draw if all spaces are filled with no winner. Simple, yet fun!

**Introduction of BFS**

The algorithm starts from a given source and explores all reachable vertices from the given source. It is similar to the Breadth-First Traversal of a tree. Like tree, we begin with the given source (in tree, we begin with root) and traverse vertices level by level using a queue data structure. The only catch here is that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

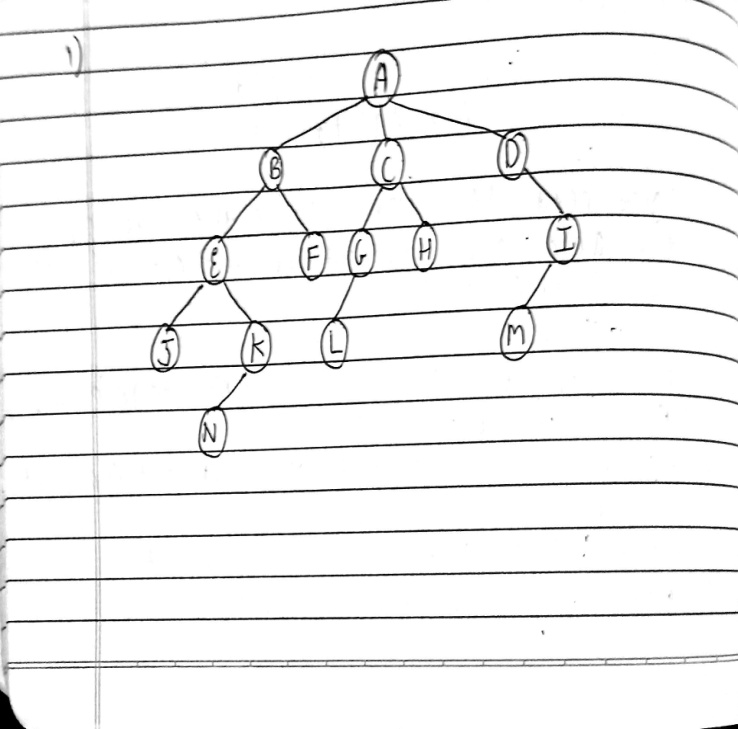
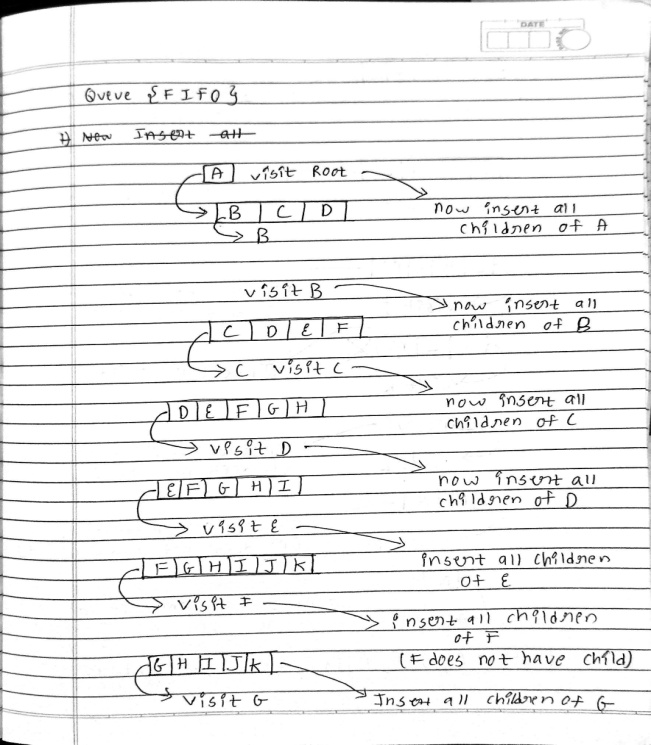
**Advantages of Breadth First Search:**

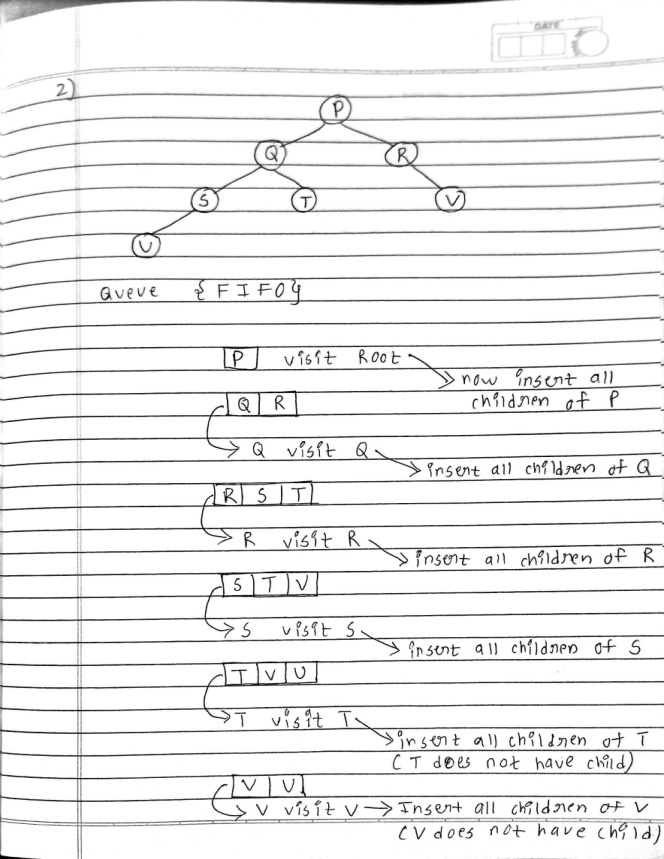
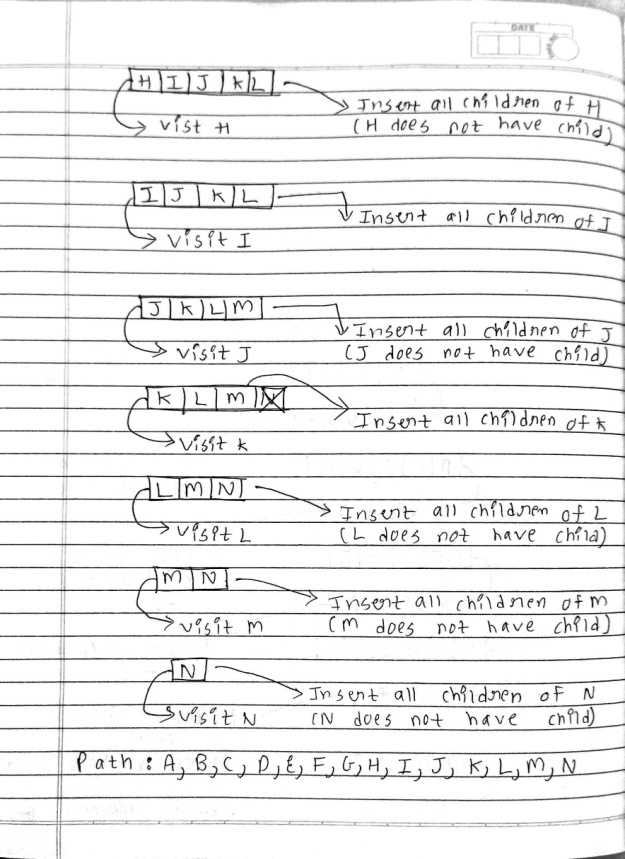
* BFS will never get trapped exploring the useful path forever.
* If there is a solution, BFS will definitely find it.
* If there is more than one solution then BFS can find the minimal one that requires less number of steps.
* Low storage requirement – linear with depth.
* Easily programmable.

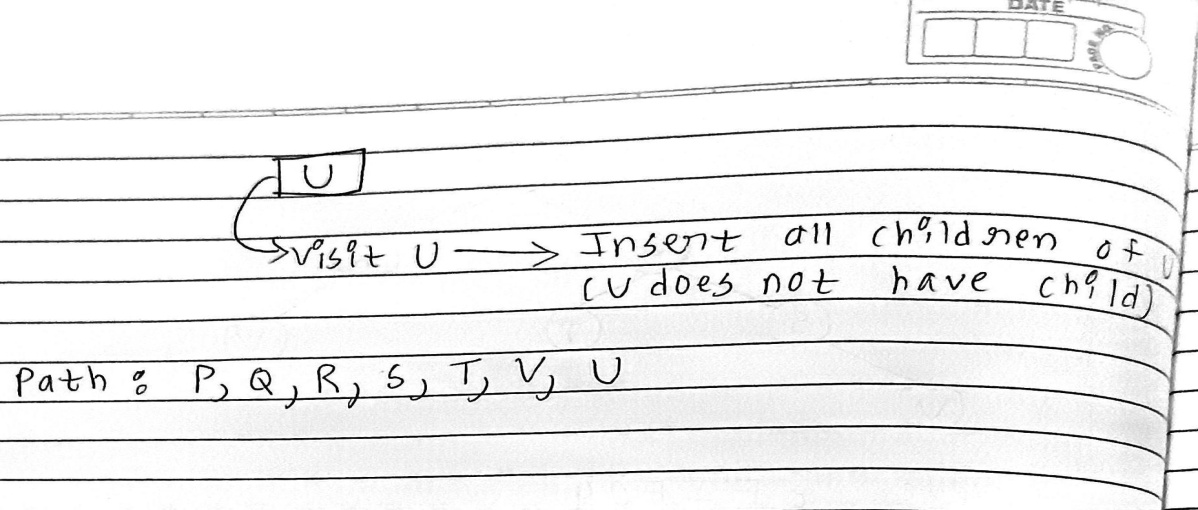
**Disadvantages of Breadth First Search:**

The main drawback of BFS is its memory requirement. Since each level of the graph must be saved in order to generate the next level and the amount of memory is proportional to the number of nodes stored the space complexity of BFS is **O(bd )**, where **b** is the branching factor(the number of children at each node, the outdegree) and **d** is the depth. As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.

#### Problems based on BFS:





#### Code:

play :- my\_turn([]).

my\_turn(Game) :- valid\_moves(ValidMoves, Game, x), any\_valid\_moves(ValidMoves, Game).

any\_valid\_moves([], \_) :- write('It is a tie'), nl.

any\_valid\_moves([\_|\_], Game) :-

findall(NextMove, game\_analysis(x, Game, NextMove), MyMoves), do\_a\_decision(MyMoves, Game).

% This can only fail in the beginning. do\_a\_decision(MyMoves, Game) :-

not(MyMoves = []), length(MyMoves, MaxMove), random(0, MaxMove, ChosenMove), nth0(ChosenMove, MyMoves, X), NextGame = [X | Game], print\_game(NextGame), (victory\_condition(x, NextGame) ->

(write('I won. You lose.'), nl); your\_turn(NextGame), !).

your\_turn(Game) :- valid\_moves(ValidMoves, Game, o), (ValidMoves = [] -> (write('It is a tie'), nl);

(write('Available moves:'), write(ValidMoves), nl, ask\_move(Y, ValidMoves),

NextGame = [Y | Game], (victory\_condition(o, NextGame) -> (write('I lose. You win.'), nl);

my\_turn(NextGame), !))).

ask\_move(Move, ValidMoves) :- write('Give your move:'), nl,

read(Move), member(Move, ValidMoves), !.

ask\_move(Y, ValidMoves) :- write('not a move'), nl, ask\_move(Y, ValidMoves).

movement\_prompt(X, Y, ValidMoves) :-

write('Give your X:'), nl, read(X), member(move(o, X, Y), ValidMoves), !, write('Give your Y:'), nl, read(Y), member(move(o, X, Y), ValidMoves).

% A routine for printing games.. Well you can use it. print\_game(Game) :-

plot\_row(0, Game), plot\_row(1, Game), plot\_row(2, Game).

plot\_row(Y, Game) :-

plot(Game, 0, Y), plot(Game, 1, Y), plot(Game, 2, Y), nl.

plot(Game, X, Y) :-

(member(move(P, X, Y), Game), ground(P)) -> write(P) ; write('.').

% This system determines whether there's a perfect play available. game\_analysis(\_, Game, \_) :-

victory\_condition(Winner, Game), Winner = x. % We do not want to lose.

% Winner = o. % We do not want to win. (egostroking mode).

% true.

% If you remove this constraint entirely, it may let you win. game\_analysis(Turn, Game, NextMove) :-

not(victory\_condition(\_, Game)), game\_analysis\_continue(Turn, Game, NextMove).

game\_analysis\_continue(Turn, Game, NextMove) :- valid\_moves(Moves, Game, Turn), game\_analysis\_search(Moves, Turn, Game, NextMove).

% Comment these away and the system refuses to play,

% because there are no ways to play this without a possibility of tie. game\_analysis\_search([], o, \_, \_). % Tie on opponent's turn. game\_analysis\_search([], x, \_, \_). % Tie on our turn.

game\_analysis\_search([X|Z], o, Game, NextMove) :- % Whatever opponent does, NextGame = [X | Game], % we desire not to lose. game\_analysis\_search(Z, o, Game, NextMove),

game\_analysis(x, NextGame, \_), !.

game\_analysis\_search(Moves, x, Game, NextMove) :- game\_analysis\_search\_x(Moves, Game, NextMove).

game\_analysis\_search\_x([X|\_], Game, X) :- NextGame = [X | Game], game\_analysis(o, NextGame, \_).

game\_analysis\_search\_x([\_|Z], Game, NextMove) :- game\_analysis\_search\_x(Z, Game, NextMove).

% This thing describes all kinds of valid games. valid\_game(Turn, Game, LastGame, Result) :-

victory\_condition(Winner, Game) ->

(Game = LastGame, Result = win(Winner)) ; valid\_continuing\_game(Turn, Game, LastGame, Result).

valid\_continuing\_game(Turn, Game, LastGame, Result) :- valid\_moves(Moves, Game, Turn), tie\_or\_next\_game(Moves, Turn, Game, LastGame, Result).

tie\_or\_next\_game([], \_, Game, Game, tie). tie\_or\_next\_game(Moves, Turn, Game, LastGame, Result) :-

valid\_gameplay\_move(Moves, NextGame, Game), opponent(Turn, NextTurn),

valid\_game(NextTurn, NextGame, LastGame, Result).

% Victory conditions for tic tac toe. victory(P, Game, Begin) :-

valid\_gameplay(Game, Begin), victory\_condition(P, Game).

victory\_condition(P, Game) :- (X = 0; X = 1; X = 2),

member(move(P, X, 0), Game),

member(move(P, X, 1), Game),

member(move(P, X, 2), Game).

victory\_condition(P, Game) :- (Y = 0; Y = 1; Y = 2),

member(move(P, 0, Y), Game),

member(move(P, 1, Y), Game),

member(move(P, 2, Y), Game).

victory\_condition(P, Game) :- member(move(P, 0, 2), Game),

member(move(P, 1, 1), Game),

member(move(P, 2, 0), Game).

victory\_condition(P, Game) :- member(move(P, 0, 0), Game),

member(move(P, 1, 1), Game),

member(move(P, 2, 2), Game).

% This describes a valid form of gameplay.

% Which player did the move is disregarded. valid\_gameplay(Start, Start).

valid\_gameplay(Game, Start) :- valid\_gameplay(PreviousGame, Start), valid\_moves(Moves, PreviousGame, \_), valid\_gameplay\_move(Moves, Game, PreviousGame).

valid\_gameplay\_move([X|\_], [X|PreviousGame], PreviousGame). valid\_gameplay\_move([\_|Z], Game, PreviousGame) :-

valid\_gameplay\_move(Z, Game, PreviousGame).

% The set of valid moves must not be affected by the decision making

% of the prolog interpreter.

% Therefore we have to retrieve them like this.

% This is equivalent to the (∀x∈0..2)(∀y∈0..2)(....

% uh wait.. There's no way to represent this using those quantifiers.

valid\_moves(Moves, Game, Turn) :- valid\_moves\_column(0, M1, [], Game, Turn), valid\_moves\_column(1, M2, M1, Game, Turn), valid\_moves\_column(2, Moves, M2, Game, Turn).

valid\_moves\_column(X, M3, M0, Game, Turn) :-

valid\_moves\_cell(X, 0, M1, M0, Game, Turn), valid\_moves\_cell(X, 1, M2, M1, Game, Turn), valid\_moves\_cell(X, 2, M3, M2, Game, Turn).

valid\_moves\_cell(X, Y, M1, M0, Game, Turn) :-

member(move(\_, X, Y), Game) -> M0 = M1 ; M1 = [move(Turn,X,Y) | M0].

% valid\_move(X, Y, Game) :-

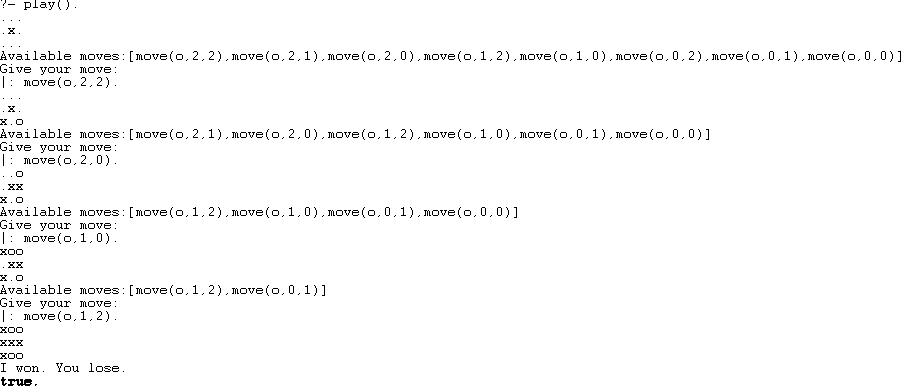
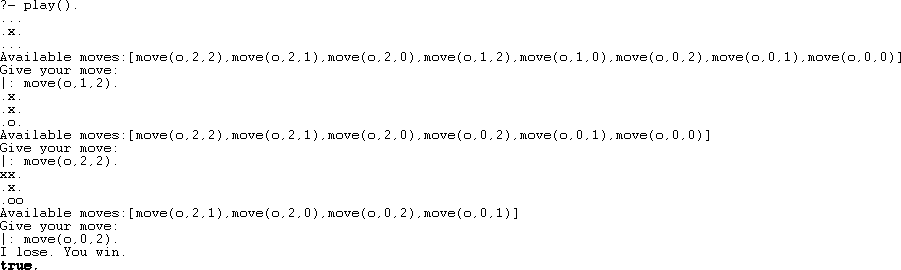
% (X = 0; X = 1; X = 2),

% (Y = 0; Y = 1; Y = 2),

% not(member(move(\_, X, Y), Game)).

opponent(x, o). opponent(o, x).

**Output:**



# Practical No. 3

**Aim:**Design Hill climbing Algorithm to solve 8-puzzle problem

#### Objectives:

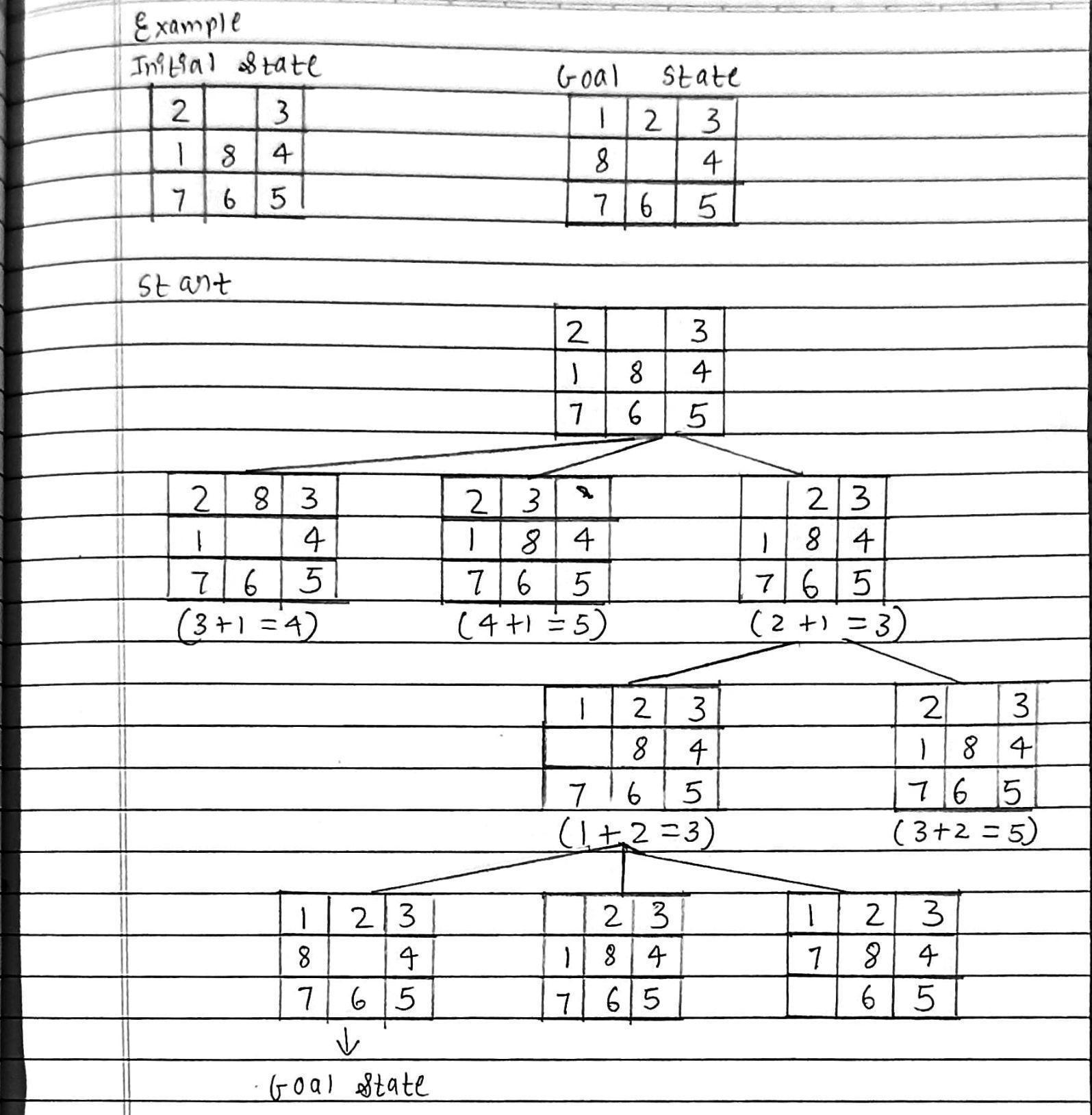
* Understand and Implement Hill Climbing algorithm
* Understand 8-puzzle problem and solve using Hill Climbing algorithm

#### Theory:

**Introduction:-**

The 8 Puzzle Problem is a classic example which is often used in the field of Artificial Intelligence (AI) to test search algorithms and problem-solving techniques. This puzzle consists of a 3x3 grid with 8 numbered tiles and one empty space

#### Examples:



**Advantages**

* Efficient: It can quickly find local optima, which is useful when time is limited
* Simple: It's easy to understand and implement
* Memory efficient: It only needs to store the current state's data
* Low computational power: It doesn't require a lot of computational power or external resources

#### Disadvantages

* Local optima: It can get stuck at locally optimal solutions that aren't the best overall
* Limited exploration: It focuses on the immediate vicinity, which can cause it to miss globally optimal solutions
* Depends on initial state: The quality of the solution depends on where the algorithm starts
* No guarantee of optimal solution: It doesn't always find the best solution

#### Code:

% 8-Puzzle Problem using Hill Climbing in Prolog

% Define goal state goal([[1,2,3], [8,0,4], [7,6,5]]).

% Move tile by swapping empty tile (0) with an adjacent tile move(State, NewState) :-

find\_blank(State, X, Y), (move\_up(State, X, Y, NewState) ; move\_down(State, X, Y, NewState) ; move\_left(State, X, Y, NewState) ; move\_right(State, X, Y, NewState)).

% Find blank (0) position

find\_blank(State, X, Y) :- nth0(X, State, Row), nth0(Y, Row, 0).

% Moves

move\_up(State, X, Y, NewState) :- X > 0, X1 is X - 1,

swap(State, X, Y, X1, Y, NewState).

move\_down(State, X, Y, NewState) :- X < 2, X1 is X + 1,

swap(State, X, Y, X1, Y, NewState).

move\_left(State, X, Y, NewState) :- Y > 0, Y1 is Y - 1,

swap(State, X, Y, X, Y1, NewState).

move\_right(State, X, Y, NewState) :- Y < 2, Y1 is Y + 1,

swap(State, X, Y, X, Y1, NewState).

% Swap tiles

swap(State, X1, Y1, X2, Y2, NewState) :- nth0(X1, State, Row1), nth0(Y1, Row1, Tile), nth0(X2, State, Row2), nth0(Y2, Row2, Tile2),

replace(State, X1, Y1, Tile2, TempState), replace(TempState, X2, Y2, Tile, NewState).

% Replace element in 2D list replace(State, X, Y, Val, NewState) :-

nth0(X, State, Row, RestRows), nth0(Y, Row, \_, RestCols), nth0(Y, NewRow, Val, RestCols),

nth0(X, NewState, NewRow, RestRows).

% Manhattan distance heuristic heuristic(State, H) :-

goal(Goal),

findall(D, (nth0(X, State, Row), nth0(Y, Row, Tile), Tile \= 0, goal\_position(Tile, Goal, GX, GY),

D is abs(X - GX) + abs(Y - GY)), Distances), sumlist(Distances, H).

% Get goal position of a tile goal\_position(Tile, Goal, X, Y) :-

nth0(X, Goal, Row), nth0(Y, Row, Tile).

% Hill Climbing search solve\_hill\_climb(State, Path) :-

solve\_hill\_climb\_helper(State, [], Path).

solve\_hill\_climb\_helper(State, Path, Path) :- goal(State). solve\_hill\_climb\_helper(State, Visited, Path) :-

move(State, NextState),

\+ member(NextState, Visited), heuristic(NextState, H),

best\_next\_state(State, H, NextState, Visited, NewState), solve\_hill\_climb\_helper(NewState, [NewState | Visited], Path).

% Choose best next state

best\_next\_state(State, BestH, BestState, Visited, NewState) :- findall(S, (move(State, S), \+ member(S, Visited)), Candidates), maplist(heuristic, Candidates, Heuristics),

min\_list(Heuristics, BestH), nth0(Index, Heuristics, BestH), nth0(Index, Candidates, NewState).

% Solve 8-puzzle with Hill Climbing solve\_puzzle(Start) :-

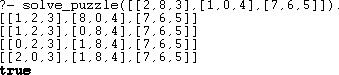
solve\_hill\_climb(Start, Path), print\_solution(Path).

% Print solution

print\_solution([]). print\_solution([S|Rest]) :-

write(S), nl, print\_solution(Rest).

**Output:**



# Practical No. 4

**Aim:**Understanding Basics of Python Programming.

**Objective:**Learn Python Libraries{NumPy,Pandas,SciPy,MatPlotlib,ScikitLearn}.

#### Theory:

**NumPy** is a library for numerical computing in Python. It provides support for

multi-dimensional arrays and a variety of mathematical functions, making it essential for

scientific computing. Its powerful array manipulation capabilities make data handling efficient and fast.

**Pandas**

**Pandas** is designed for data analysis and manipulation. It introduces structures like Series and DataFrame, which simplify data organization, filtering, and cleaning. It’s widely used in data science and integrates well with other libraries.

**Matplotlib**

**Matplotlib** is a popular visualization library used to create a variety of plots, including line graphs, bar charts, and scatter plots. It provides detailed control over visual elements, making it ideal for clear and informative data presentation.

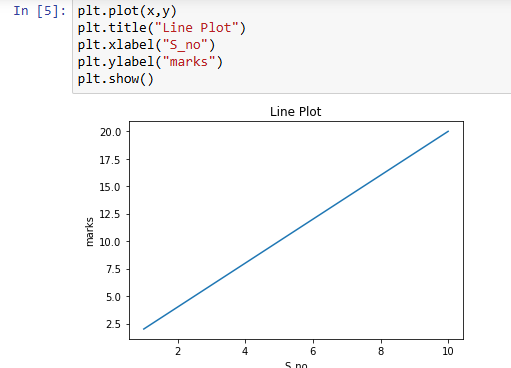
**SciPy**

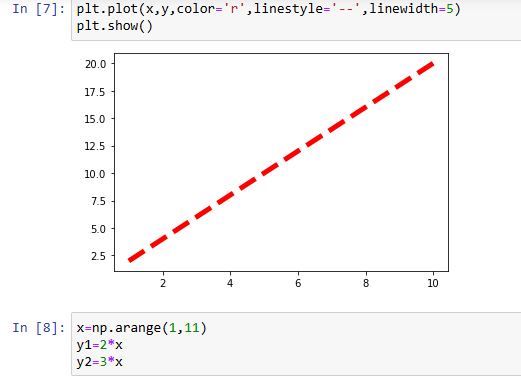
**SciPy** extends NumPy by offering advanced scientific functions such as optimization, signal processing, and integration. It is particularly useful for complex mathematical computations.

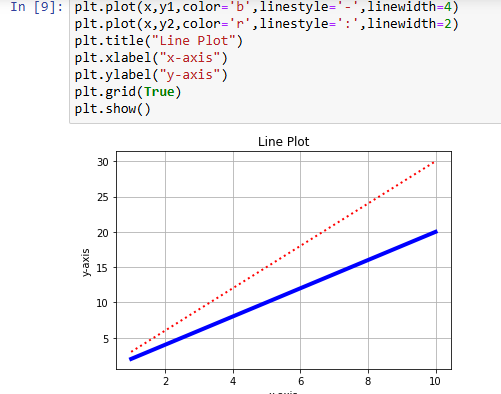
**Scikit-learn**

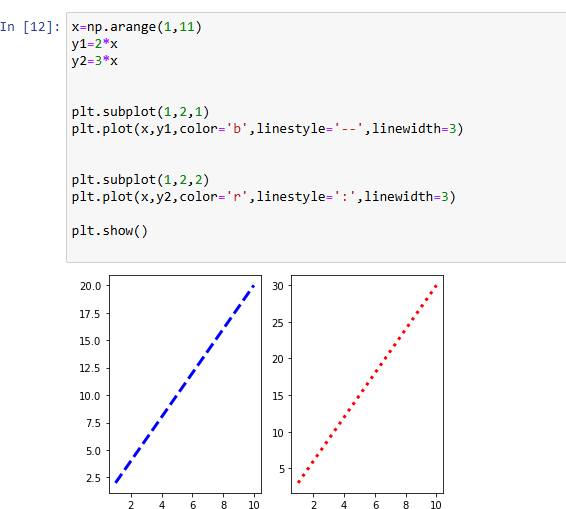
**Scikit-learn** is a machine learning library that simplifies tasks like classification, regression, and clustering. It includes powerful tools for model evaluation, data preprocessing, and feature selection, making it a go-to choice for building predictive models.

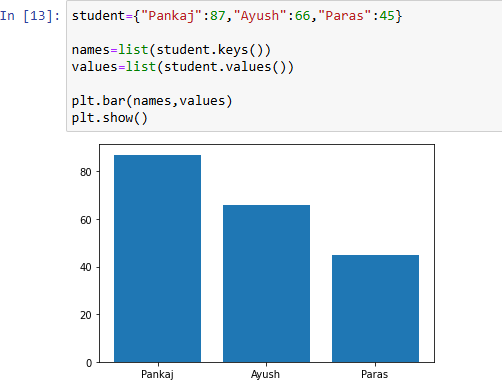
**Implementation:**

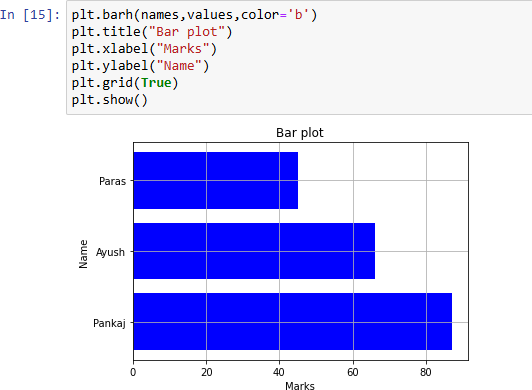


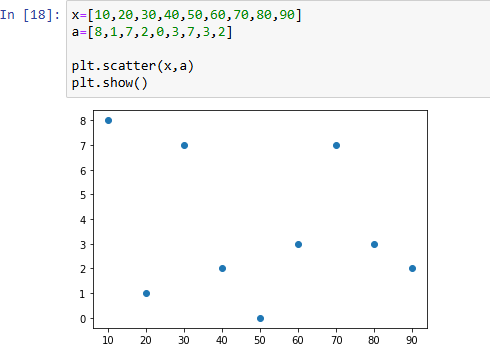




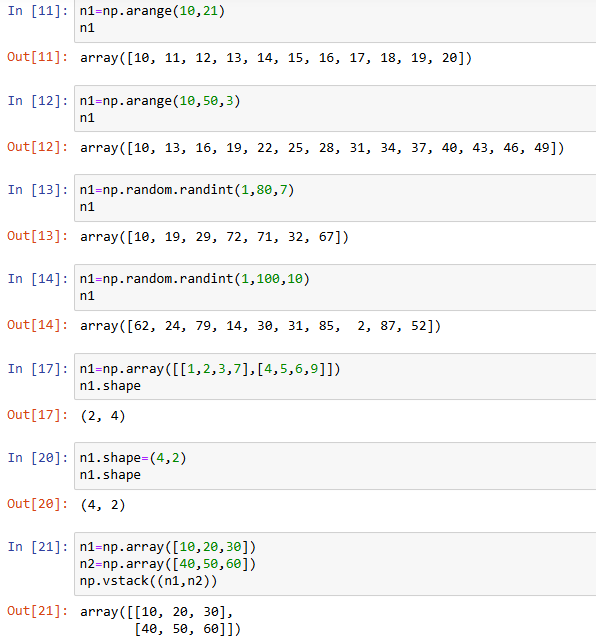
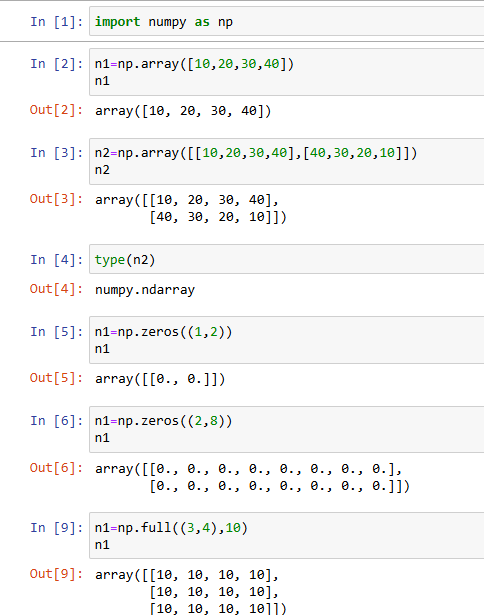


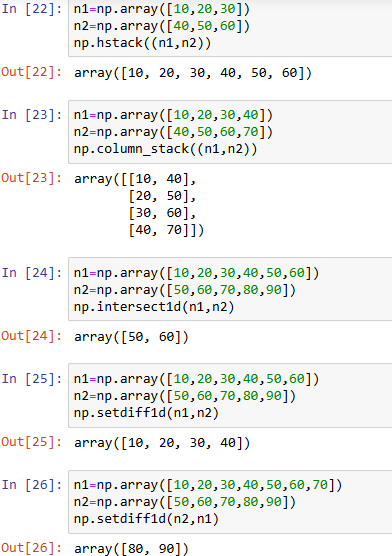


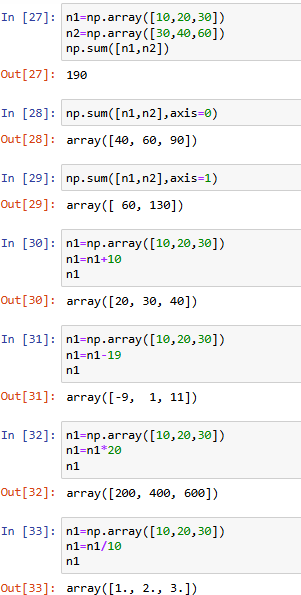




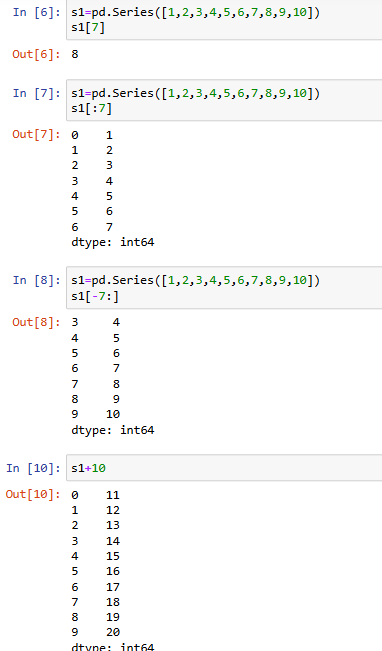
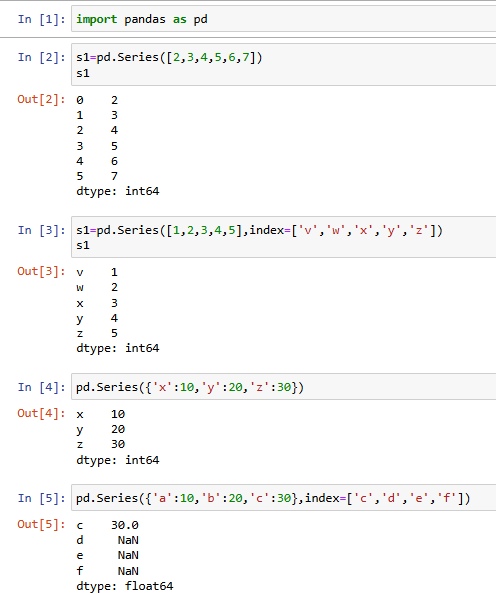
**Numpy**

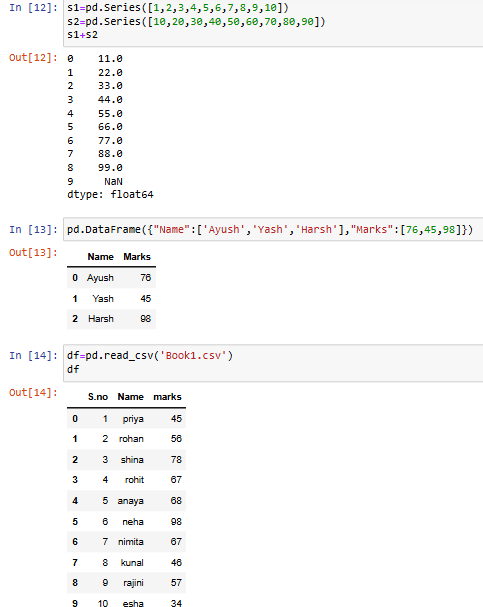


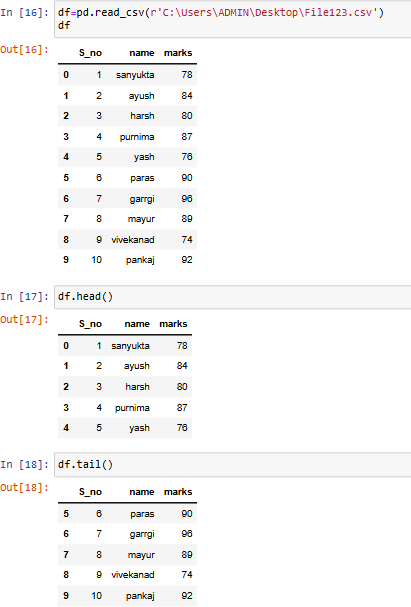


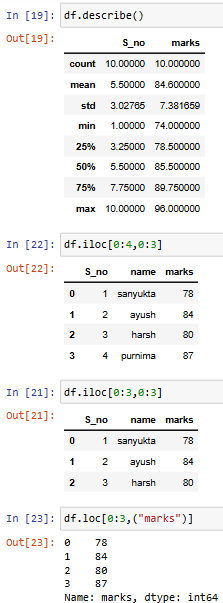


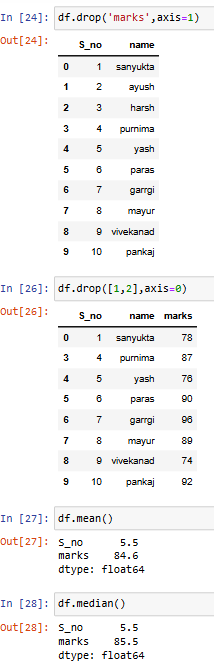
**Pandas**

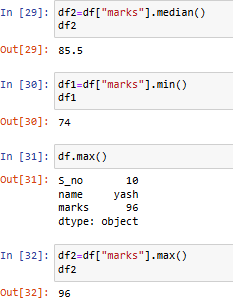




 **.**

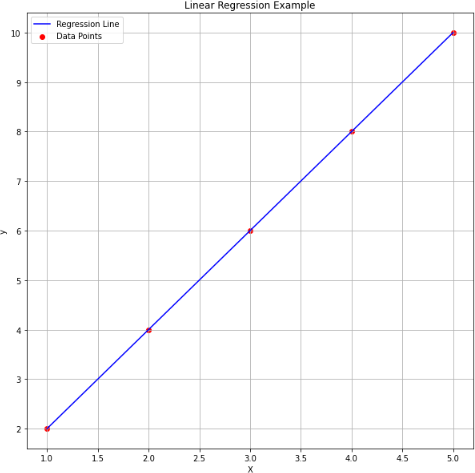




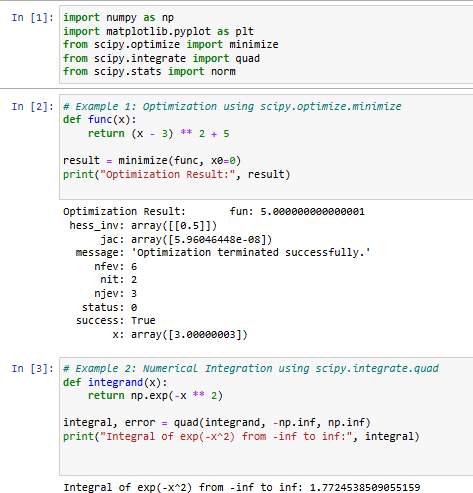


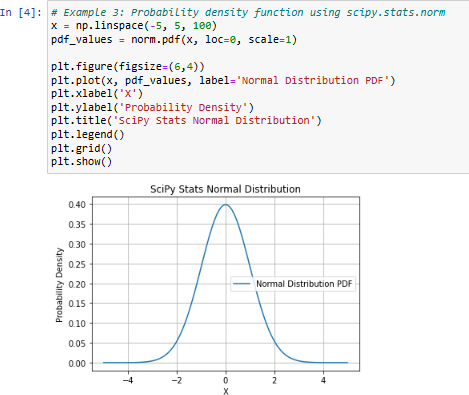
#### Scikit-Learn





**MatPlotLib**





**Conclusion:**Successfully Learned all basic xPython Libraries

# Practical No. 5

**AIM:-**Implementation of Perceptron algorithm for OR operation.

#### Objective:-

* Understanding of Perceptron.
* Understanding of OR Operation using Perceptron.
* Implementation of Perceptron for OR operation

#### Theory:-

* Introduction of Perceptron

The perceptron is a fundamental artificial neural network model introduced by Frank Rosenblatt in 1958. It serves as a binary classifier that processes input features through weighted connections, applies an activation function, and predicts output. The perceptron is the foundation of modern deep learning and machine learning advancements.

* Perceptron algorithm for OR operation



#### Code:

# required Libraries import numpy as np

import matplotlib.pyplot as plt

# input features and targets (bipolar targets) X = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])

t = np.array([1, 1, 1, -1])

# Initialization of weights and bias weights = np.zeros(2)

bias = 0

learning\_rate = 0.1

# Perceptron training algorithm for 3 epochs for epoch in range(3):

print(f"Epoch {epoch + 1}") for i in range(len(X)):

# Compute activation

activation = np.dot(X[i], weights) + bias output = 1 if activation >= 0 else -1

# Update weights and bias if there's an error if output != t[i]:

weights += learning\_rate \* t[i] \* X[i] bias += learning\_rate \* t[i]

print(f"Input: {X[i]}, Target: {t[i]}, Output: {output}, Weights: {weights}, Bias: {bias}") print("-")

# Final weights and bias after training print("Final Weights:", weights) print("Final Bias:", bias)

# Plot decision boundary x\_values = np.array([0, 1])

y\_values = (-weights[0] \* x\_values - bias) / weights[1]

plt.figure(figsize=(6,6))

plt.scatter(X[:, 0], X[:, 1], c=t, cmap='bwr', edgecolors='k', s=100) plt.plot(x\_values, y\_values, 'k--', label='Decision Boundary') plt.xlim(-0.5, 1.5)

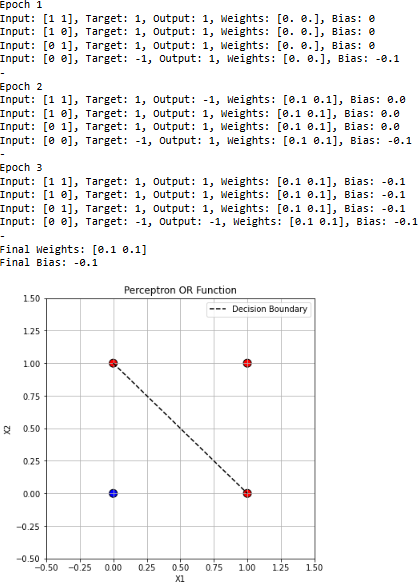
plt.ylim(-0.5, 1.5)

plt.xlabel('X1')

plt.ylabel('X2') plt.title('Perceptron OR Function') plt.legend()

plt.grid() plt.show()

#### Output:



**Conclusion:** Successfully Implemented Perceptron algorithm for OR operation.

# Practical No. 6

**AIM:-**Implementation of ADALINE algorithm for AND operation.

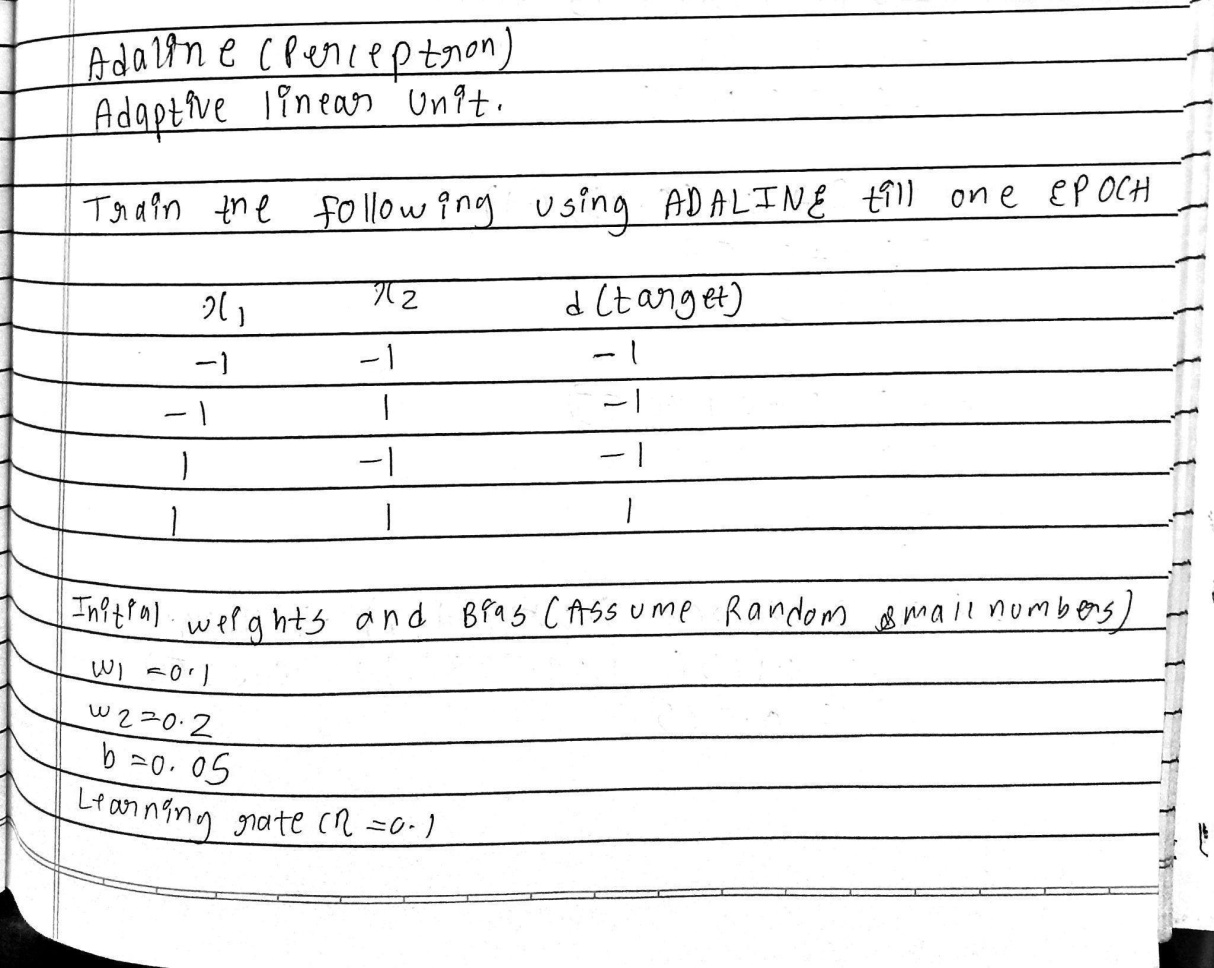
#### Objective:-

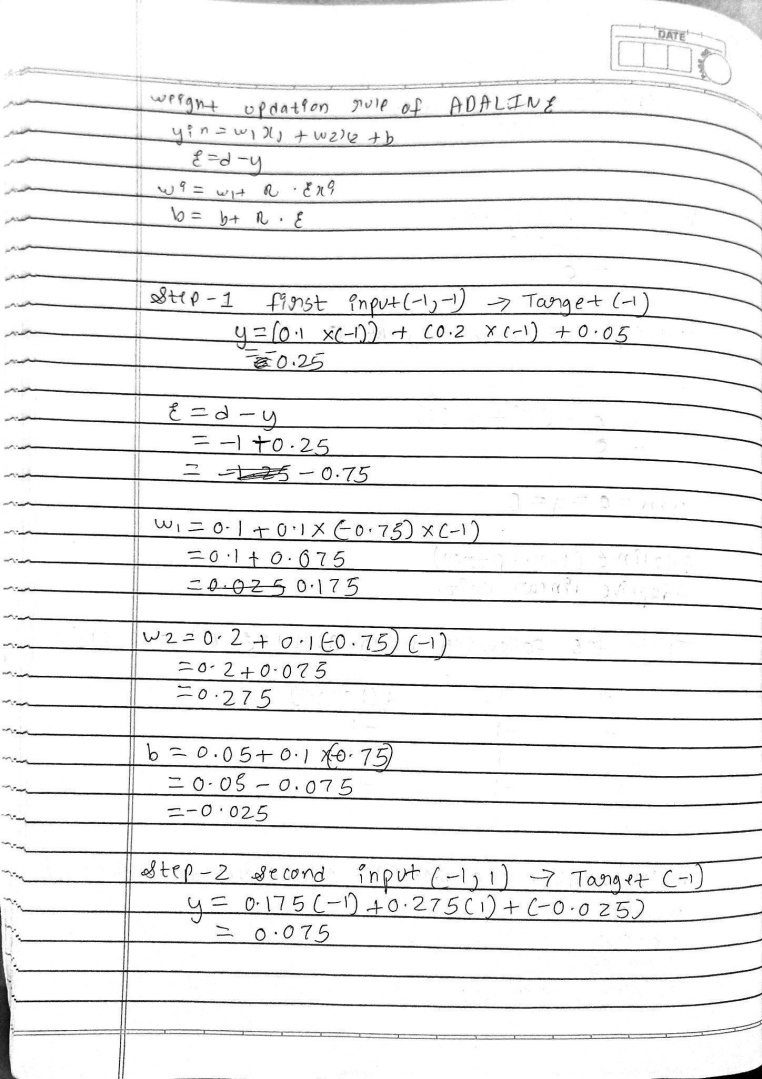
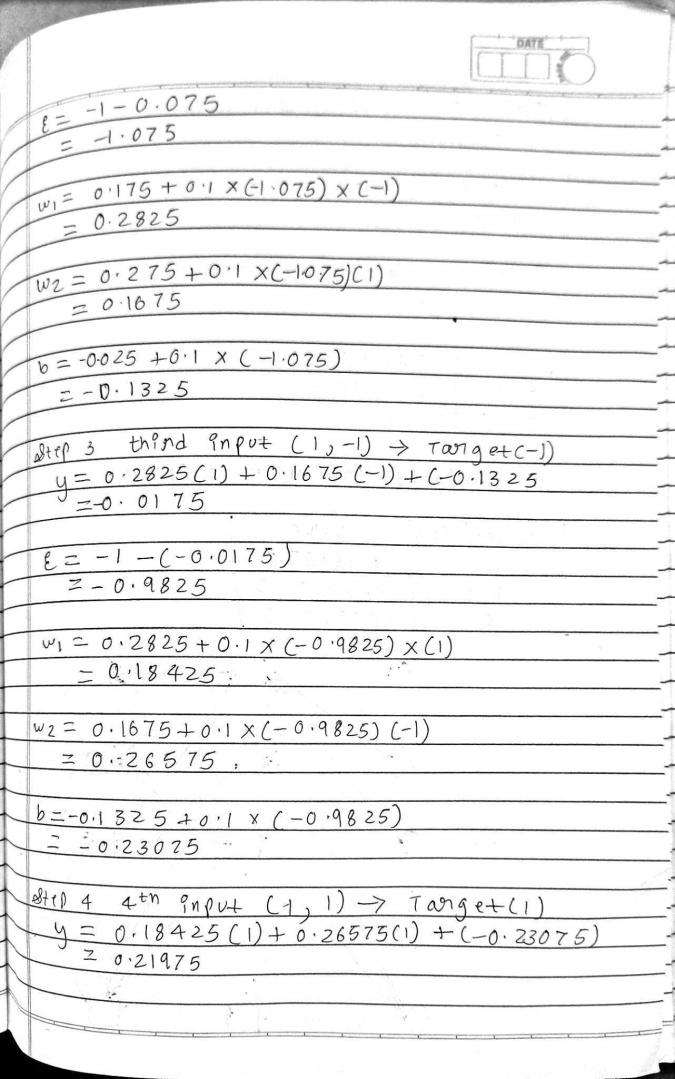
* Understanding of ADALINE.
* Understanding of AND Operation using ADALINE.
* Implementation of ADALINE for AND operation

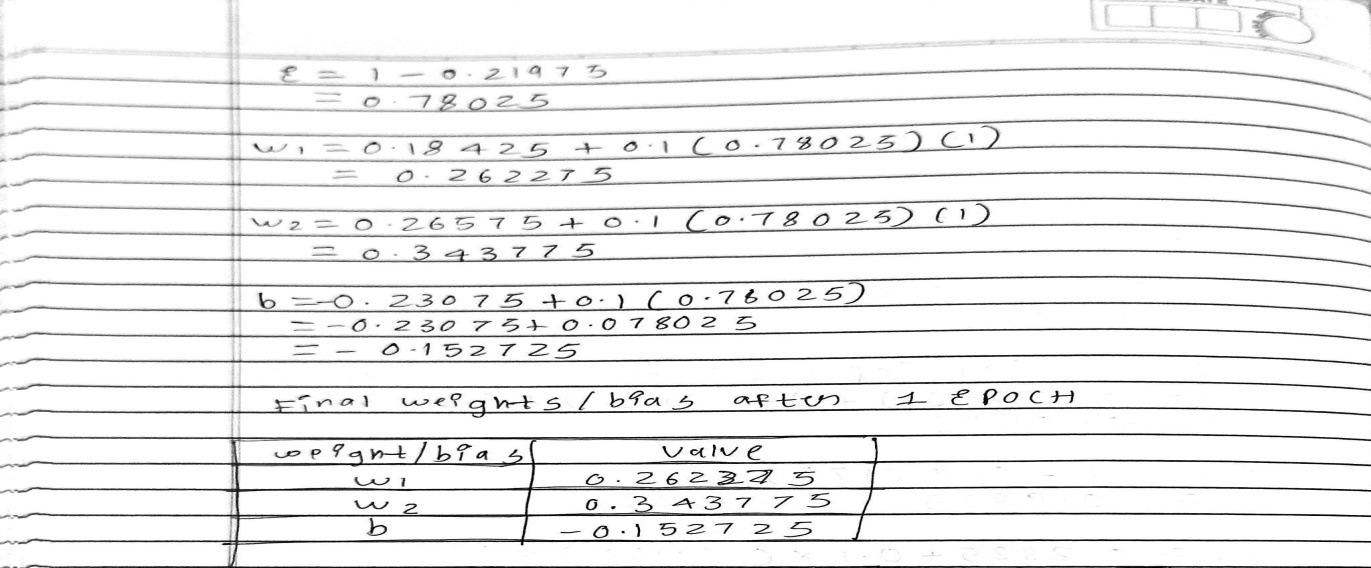
#### Theory:-

* Introduction of ADALINE

ADALINE (Adaptive Linear Neuron) is a single-layer neural network model developed by Bernard Widrow and Ted Hoff in 1960. It uses weighted inputs and applies the linear activation function before error correction via the Least Mean Squares (LMS) algorithm, making it effective in pattern recognition and signal processing tasks.

* ADALINE algorithm for AND operation



**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define AND gate inputs and expected outputs

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Input features

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

# Add bias term

X = np.c\_[np.ones((X.shape[0], 1)), X] # Adding bias as the first column

# Initialize weights

weights = np.random.rand(X.shape[1]) learning\_rate = 0.1

epochs = 20

# Activation function (Identity function for Adaline) def activation\_function(net\_input):

return net\_input

# Training Adaline using gradient descent errors = []

for epoch in range(epochs): total\_error = 0

for i in range(X.shape[0]):

net\_input = np.dot(X[i], weights) # Compute net input

output = activation\_function(net\_input) # Compute activation output error = y[i] - output # Compute error

weights += learning\_rate \* error \* X[i] # Update weights total\_error += error\*\*2 # Sum of squared errors

errors.append(total\_error)

print(f"Epoch {epoch+1}, Error: {total\_error}") # Plot error reduction

epochs\_range = range(1, epochs + 1) plt.plot(epochs\_range, errors, marker='o') plt.xlabel('Epochs')

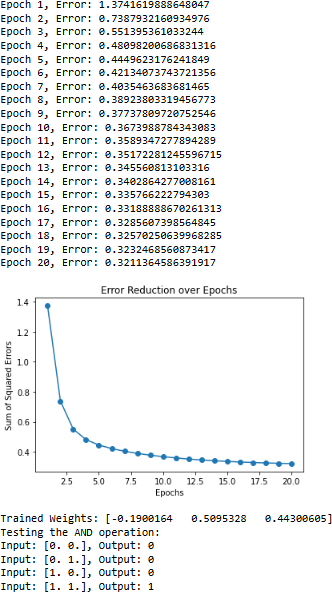
plt.ylabel('Sum of Squared Errors') plt.title('Error Reduction over Epochs') plt.show()

# Testing the trained Adaline model print("Trained Weights:", weights) print("Testing the AND operation:") for i in range(X.shape[0]):

net\_input = np.dot(X[i], weights)

output = 1 if net\_input >= 0.5 else 0 # Threshold function for classification print(f"Input: {X[i][1:]}, Output: {output}")

#### Output:



**Conclusion:** Successfully Implemented ADALINE algorithm for AND operation

# Practical No. 7

**AIM:-** Improve the prediction Accuracy by estimating the weight values for the training data using Stochastic Gradient Descent(Perceptron)

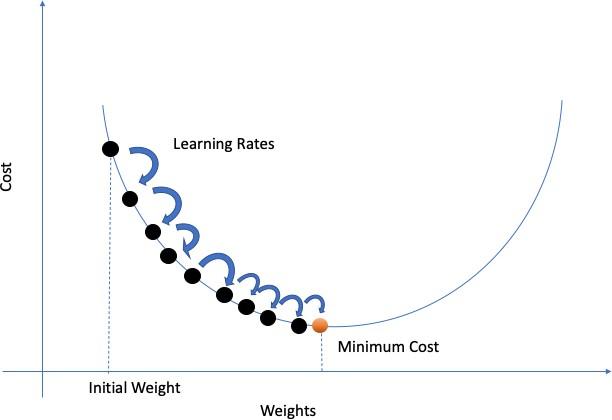
#### Objective:-

* Understanding of Stochasting Gradient Descent.
* Improving of prediction Accuracy using SGD.

#### Theory:-

* Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization algorithm used in machine learning to minimize a model's error. Unlike standard gradient descent, SGD updates model parameters using a single data sample at a time, improving efficiency for large datasets and enhancing convergence speed, especially in deep learning applications.



## Advantages of SGD

✅ Efficient for large datasets

✅ Reduces memory requirements since only one data point is processed at a time

✅ Can escape sharp local minima due to its noisy updates

## Disadvantages of SGD

❗ Fluctuations in updates may slow convergence

❗ Requires careful tuning of the learning rate

❗ May struggle with very noisy data without techniques like momentum or learning rate schedules

#### Code:

**[1]**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_classification from sklearn.model\_selection import train\_test\_split

# Generate a more complex dataset with noise and overlapping classes X, y = make\_classification(n\_samples=200, n\_features=2, n\_classes=2,

n\_redundant=0, n\_clusters\_per\_class=1, class\_sep=0.05, flip\_y=0.3, random\_state=42)

# Convert labels from {0,1} to {-1,1} for perceptron y = np.where(y == 0, -1, 1)

# Split dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

class PerceptronBatch:

def init (self, learning\_rate=0.0001, epochs=20): # Further reduced learning rate and epochs self.learning\_rate = learning\_rate

self.epochs = epochs self.weights = None self.bias = None

def fit(self, X, y):

n\_samples, n\_features = X.shape self.weights = np.zeros(n\_features) self.bias = 0

for epoch in range(self.epochs): total\_error = 0

for i in range(n\_samples):

linear\_output = np.dot(X[i], self.weights) + self.bias y\_predicted = np.where(linear\_output >= 0, 1, -1)

if y\_predicted != y[i]:

total\_error += y[i] - y\_predicted

# Apply weaker weight updates with decay

decay = 1 / (epoch + 1) # Reduce learning rate over epochs

self.weights += (self.learning\_rate \* total\_error \* np.mean(X, axis=0)) \* decay self.bias += (self.learning\_rate \* total\_error) \* decay

def predict(self, X):

linear\_output = np.dot(X, self.weights) + self.bias return np.where(linear\_output >= 0, 1, -1)

# Train Perceptron with Batch Learning

perceptron = PerceptronBatch(learning\_rate=0.0001, epochs=20) perceptron.fit(X\_train, y\_train)

y\_pred = perceptron.predict(X\_test) accuracy = np.mean(y\_pred == y\_test)

print(f"Prediction Accuracy with Batch Perceptron: {accuracy \* 100:.2f}%")

# Plot decision boundary

def plot\_decision\_boundary(X, y, model, title): x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100)) Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3) plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k') plt.title(title)

plt.show()

plot\_decision\_boundary(X\_test, y\_test, perceptron, "Perceptron with Batch Learning (Lower Accuracy)")

#### [2]

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_classification from sklearn.model\_selection import train\_test\_split

# Generate a synthetic dataset

X, y = make\_classification(n\_samples=200, n\_features=2, n\_classes=2,

n\_redundant=0, n\_clusters\_per\_class=1, class\_sep=0.5, random\_state=42)

# Convert labels from {0,1} to {-1,1} for perceptron y = np.where(y == 0, -1, 1)

# Split dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

class PerceptronSGD:

def init (self, learning\_rate=0.01, epochs=50): self.learning\_rate = learning\_rate

self.epochs = epochs self.weights = None self.bias = None

def fit(self, X, y):

n\_samples, n\_features = X.shape self.weights = np.zeros(n\_features) self.bias = 0

for epoch in range(self.epochs): for i in range(n\_samples):

linear\_output = np.dot(X[i], self.weights) + self.bias y\_predicted = np.where(linear\_output >= 0, 1, -1)

# Update rule

update = self.learning\_rate \* (y[i] - y\_predicted) self.weights += update \* X[i]

self.bias += update

def predict(self, X):

linear\_output = np.dot(X, self.weights) + self.bias return np.where(linear\_output >= 0, 1, -1)

# Train Perceptron with SGD

perceptron = PerceptronSGD(learning\_rate=0.01, epochs=50) perceptron.fit(X\_train, y\_train)

y\_pred = perceptron.predict(X\_test) accuracy = np.mean(y\_pred == y\_test)

print(f"Prediction Accuracy with Stochastic Gradient Descent: {accuracy \* 100:.2f}%")

# Plot decision boundary

def plot\_decision\_boundary(X, y, model, title): x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100)) Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

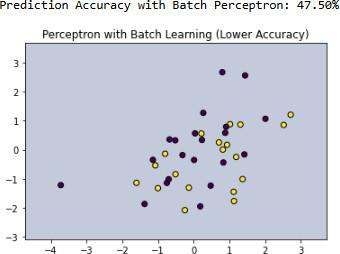
plt.contourf(xx, yy, Z, alpha=0.3) plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k') plt.title(title)

plt.show()

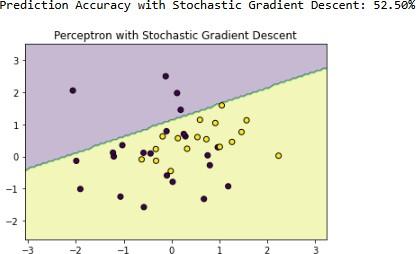
plot\_decision\_boundary(X\_test, y\_test, perceptron, "Perceptron with Stochastic Gradient Descent")

#### Output:

[1]



[2]



**Conclusion:** Successfully Improved accuracy by using Stochastic Gradient Descent

# Practical No. 8

**Aim:-**Implementation of feature Extraction,selection,Normalization,Transformation,Principal Component Analysis

#### Objectives:-

* Understanding of features Extraction,selection,Normalization,Transformation and Reduction.
* Understanding Principal component Analysis(PCA)
* Implementation of above techniques in Python.

**Theory:-**all above mentioned terms Definition with Example

#### Feature Extraction

* + The process of deriving new features from raw data to improve model performance. It involves transforming data into a format that is more meaningful for machine learning algorithms. Examples include edge detection in images or word embeddings in NLP.
  + Example:

#### Feature Selection

* + The process of selecting the most relevant features from a dataset while removing redundant or irrelevant ones. This improves model performance and reduces overfitting. Common methods include filter methods (e.g., correlation), wrapper methods (e.g., recursive feature elimination), and embedded methods (e.g., LASSO regression).

#### Feature Normalization

* + The process of scaling numerical features to a common range, typically [0,1] or [-1,1], to ensure that all features contribute equally to the model. Common techniques include Min-Max Scaling and Z-score Standardization.

#### Feature Transformation

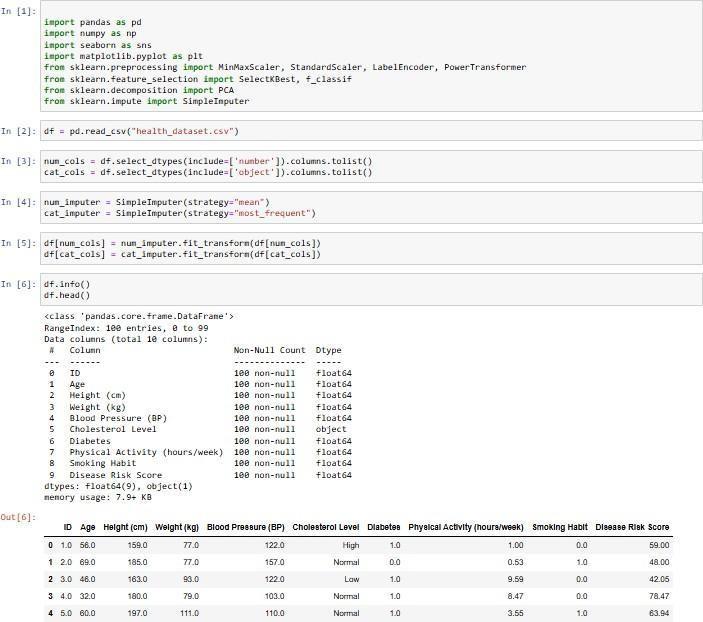
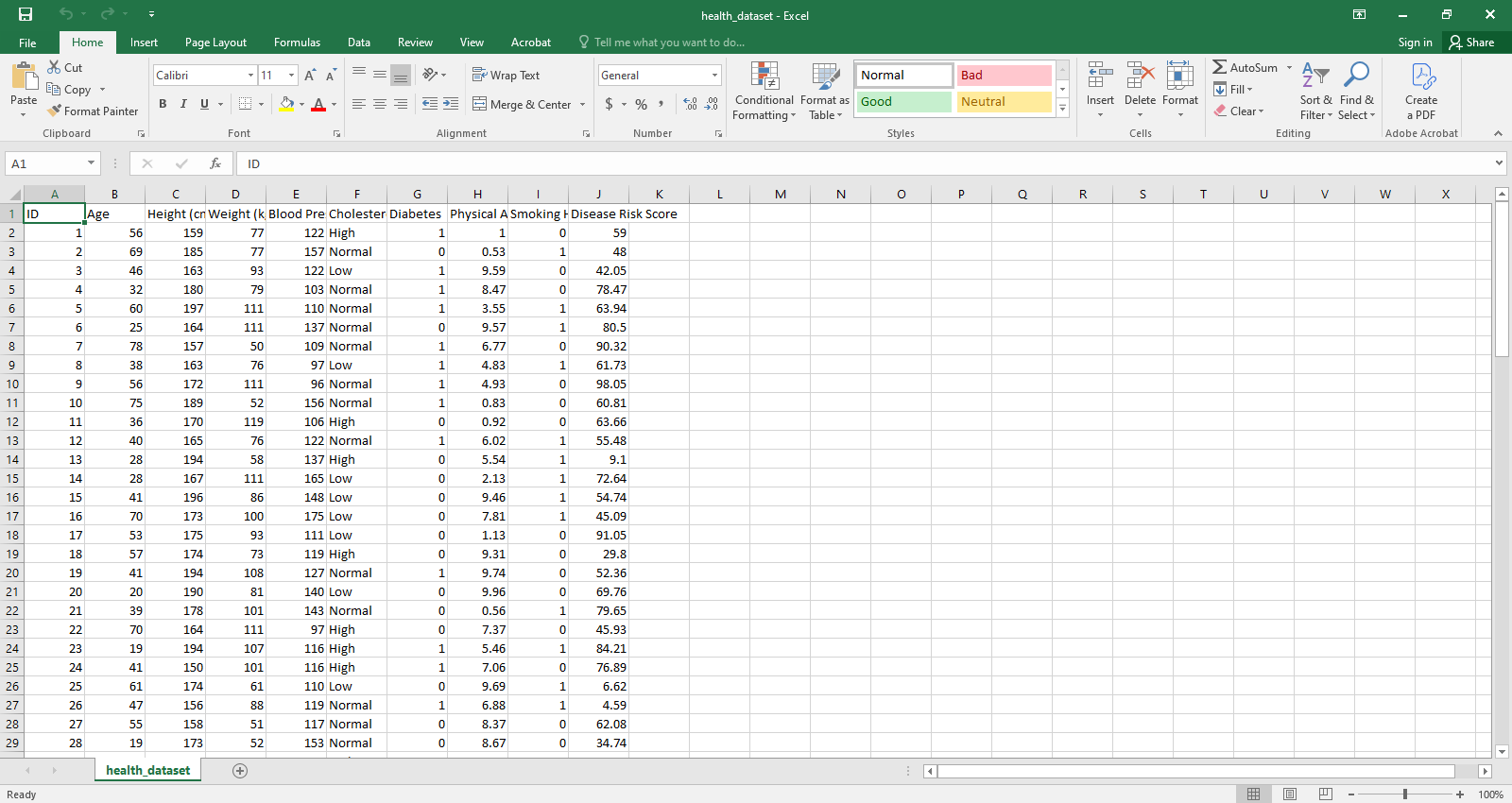
* + The process of modifying or encoding features to make them more suitable for machine learning models. This includes techniques like logarithmic scaling, polynomial transformations, encoding categorical variables, and applying mathematical functions.

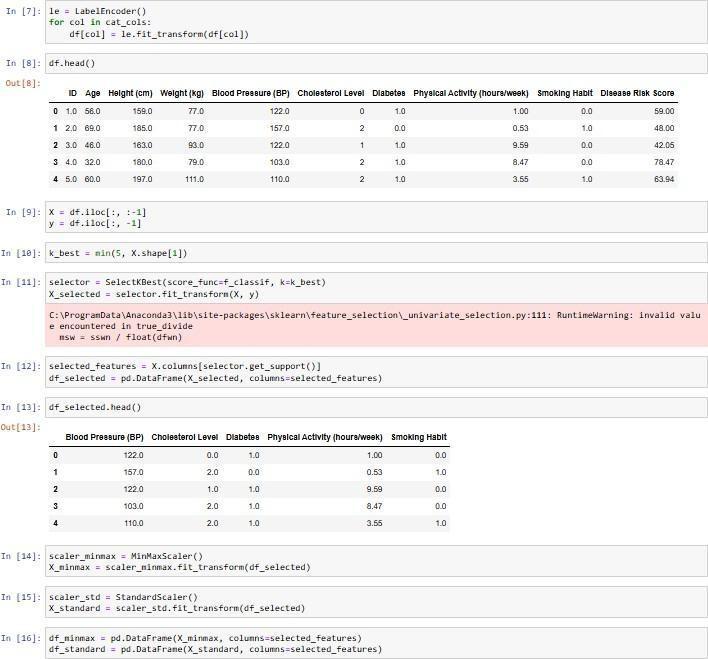
#### Principal Component Analysis (PCA)

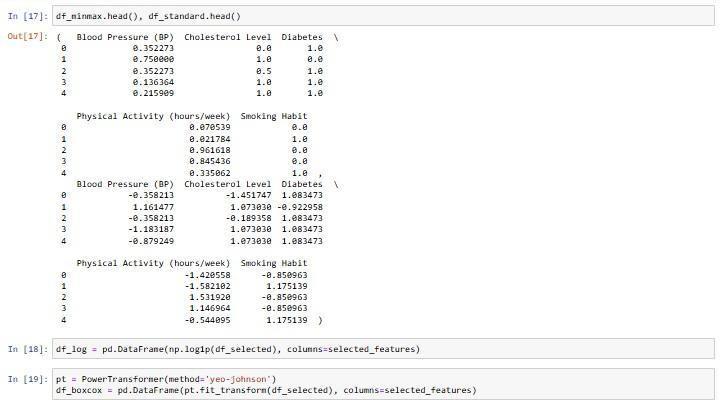
* + A dimensionality reduction technique that transforms a high-dimensional dataset into a lower-dimensional space by identifying the principal components (new axes) that capture the most variance in the data. PCA helps reduce complexity while retaining important information.

#### Code & Output:-

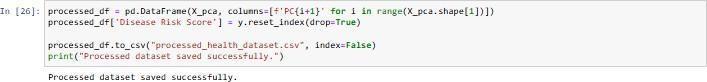
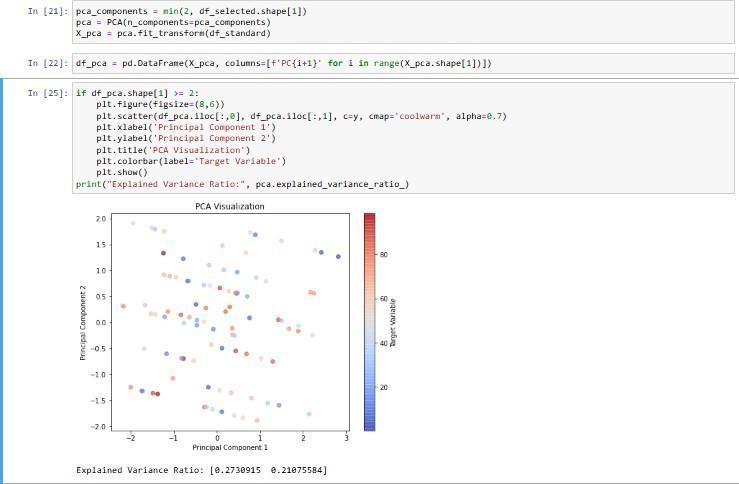
Dataset Description:



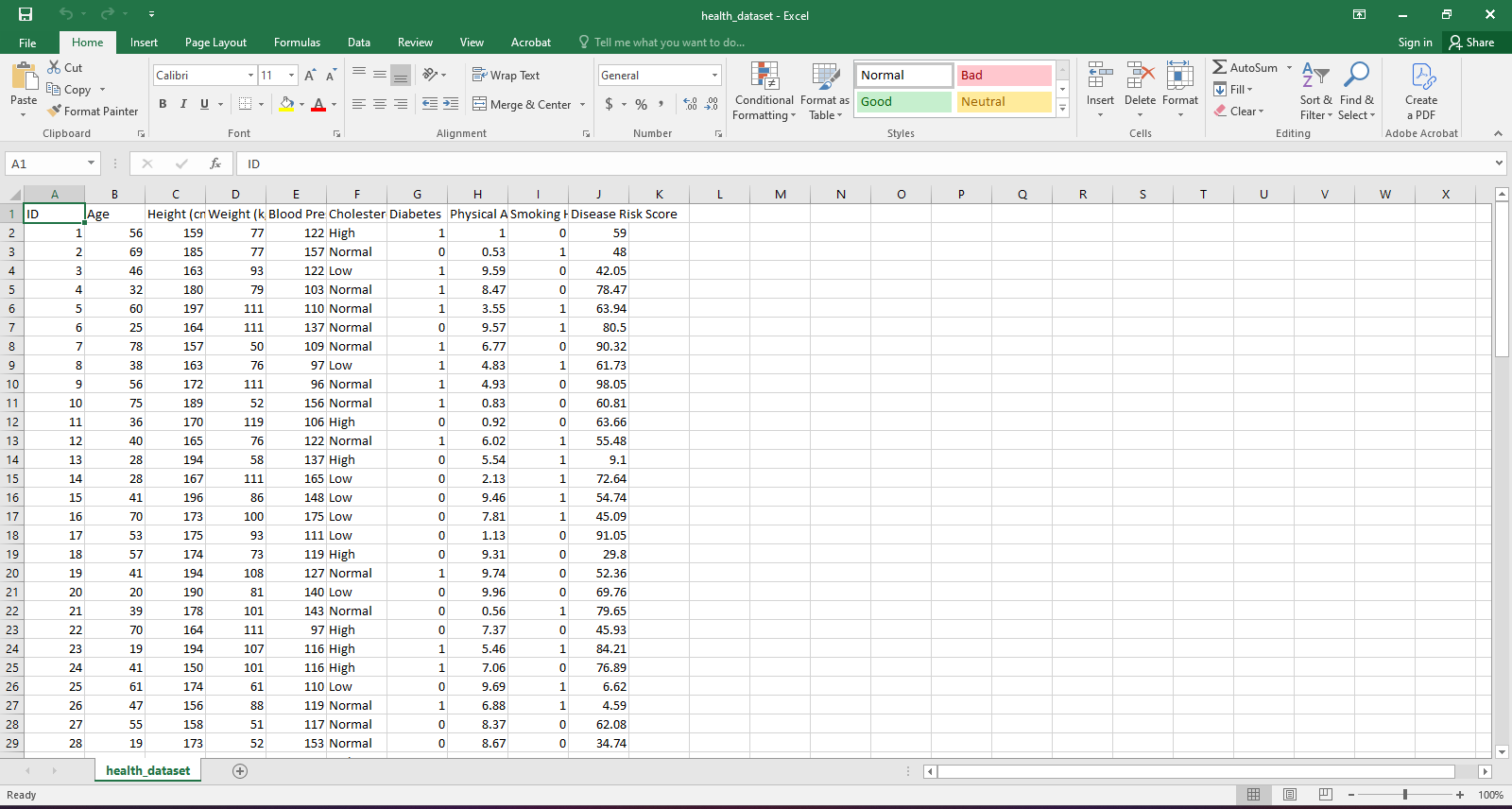








Processed Dataset:



**Conclusion:-**Successfully Implemented feature Extraction, Selection, Normalization, Transformation, PCA.

**Practical No. 9 Aim:**Implement a python program to demonstrate Logistic regression **Objectives:**

Understanding of Non-Linear Regression. Understanding of Logistic Regression.

Implementation of Logistic Regression using PYTHON

**Theory:**

## Nonlinear Regression

A type of regression where the relationship between the independent and dependent variables is modeled using a **nonlinear function**.

**Use:** When data doesn't follow a straight-line pattern.

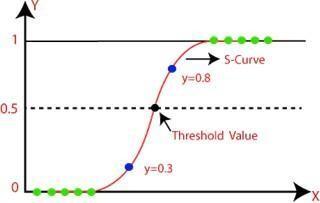
**Example:** Modeling population growth using an exponential curve:

1. y=a⋅ebx ,where y is population and x is time.

## Logistic Regression

A **classification algorithm**, not a true regression, used to predict **binary outcomes** (e.g., yes/no, 0/1).Uses the logistic (sigmoid) function to map predicted values to a probability between 0 and 1.

#### Formula:



P(y=1) = 1 /

1+e−(b0+b1x)

**Example:**Predicting whether an email is spam (1) or not spam (0) based on word features.

#### Code:

import numpy as np import pandas as pd

import matplotlib.pyplot as plt import seaborn as sns

from sklearn.model\_selection import train\_test\_split from sklearn.linear\_model import LogisticRegression from sklearn.datasets import make\_classification from sklearn.metrics import accuracy\_score

# Generate a simple dataset for S-curve visualization

X, y = make\_classification(n\_samples=500, n\_features=1, n\_informative=1, n\_redundant=0, n\_clusters\_per\_class=1, random\_state=42)

# Save dataset to CSV file

df = pd.DataFrame({'Feature': X.ravel(), 'Target': y}) df.to\_csv('logistic\_regression\_data.csv', index=False)

# Load dataset from CSV file

data = pd.read\_csv('logistic\_regression\_data.csv')

X = data[['Feature']].values y = data['Target'].values

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train logistic regression model model = LogisticRegression() model.fit(X\_train, y\_train)

# Predict probabilities

X\_range = np.linspace(X.min() - 1, X.max() + 1, 500).reshape(-1, 1) y\_prob = model.predict\_proba(X\_range)[:, 1]

# Plot S-curve plt.figure(figsize=(8, 5))

sns.scatterplot(x=X.ravel(), y=y, label='Data', alpha=0.5) plt.plot(X\_range, y\_prob, color='red', label='Logistic Regression Curve') plt.xlabel('Feature')

plt.ylabel('Probability')

plt.title('S-Curve of Logistic Regression') plt.legend()

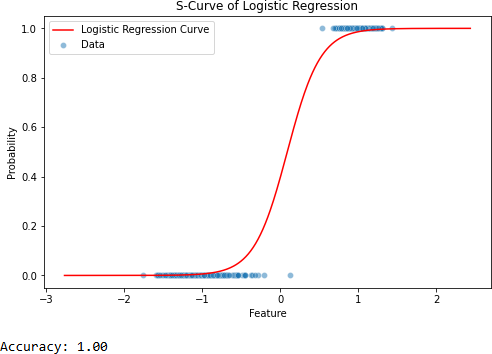
plt.show()

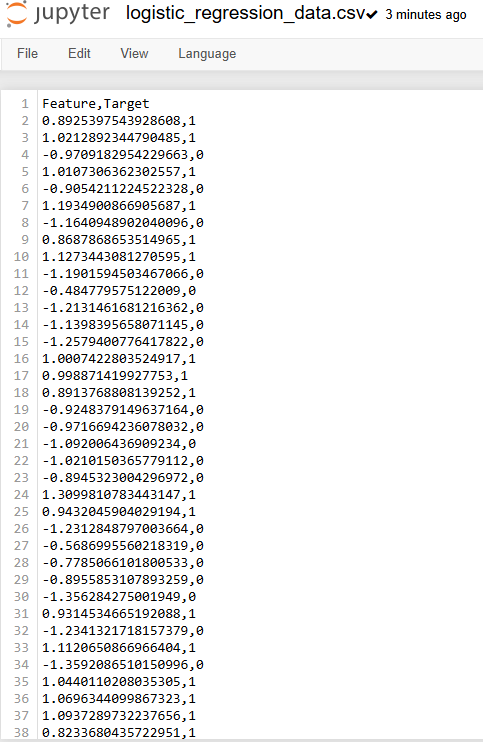
# Evaluate model

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred) print(f'Accuracy: {accuracy:.2f}')

#### Output:





**Conclusion:**Successfully demonstrated Logistic Regression.

**Practical No. 10 Aim:**Implement a python program to find Hyperplane using Linear SVM. **Objectives:**

Understanding of HyperPlane. Understanding of Linear SVM.

Implementation of Linear SVM using PYTHON

**Theory:**

### Linear SVM (Support Vector Machine):

A **Linear SVM** is a supervised learning algorithm used for binary classification. It finds the best hyperplane that separates data points of different classes.

#### Hyperplane:

A hyperplane is a decision boundary that divides the feature space into two classes.

* In 2D: it's a line
* In 3D: it's a plane
* In higher dimensions: it's a hyperplane

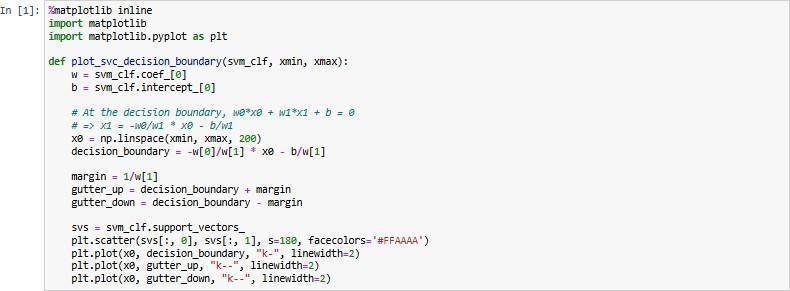
**Goal of SVM:** To maximize the margin — the distance between the hyperplane and the nearest data points from each class (called support vectors).

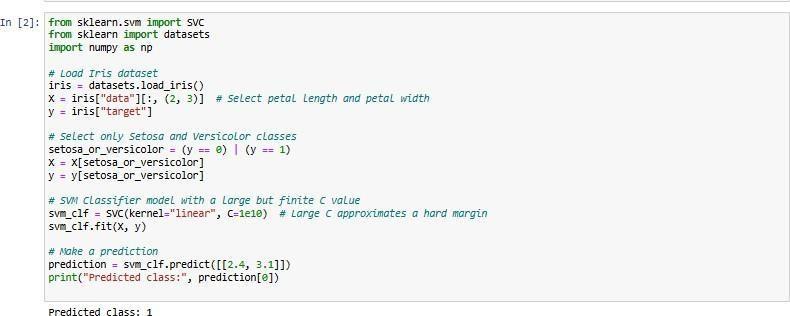
**Example**: Classifying emails as "spam" or "not spam" using word frequencies. SVM will find the line (hyperplane) that best separates the two types of emails with the widest margin.

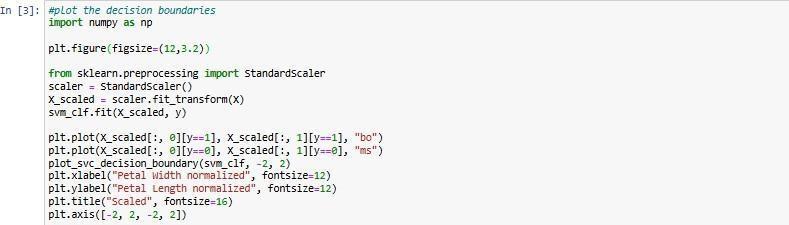
**Example:**

Classifying circular data — imagine two classes shaped like concentric circles. A linear hyperplane won’t work in 2D, but after using RBF kernel, SVM can draw a circular boundary that separates the classes.

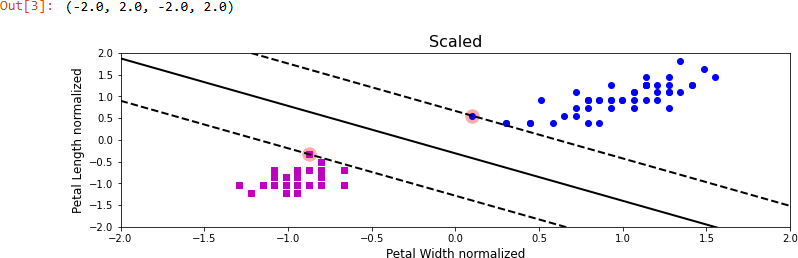
#### Code:







**Output:**



**Conclusion:**Successfully demonstrated Linear SVM.

# Practical No. 11

**Aim:**Implement a python program to find Hyperplane using Non-Linear SVM.

#### Objectives:

Understanding of HyperPlane. Understanding of Non-Linear SVM.

Implementation of Non-Linear SVM using PYTHON

#### Theory:

**Non-Linear SVM**

A Non-Linear SVM is used when data cannot be separated by a straight line (or hyperplane). It uses a kernel trick to transform the data into a higher dimension where a linear hyperplane can separate the classes.

**Hyperplane in Non-Linear SVM**

Although the separation isn't linear in original space, SVM still finds a linear hyperplane — but in the transformed feature space, not in the original one.

So, in the original space, this hyperplane may look curved or complex.

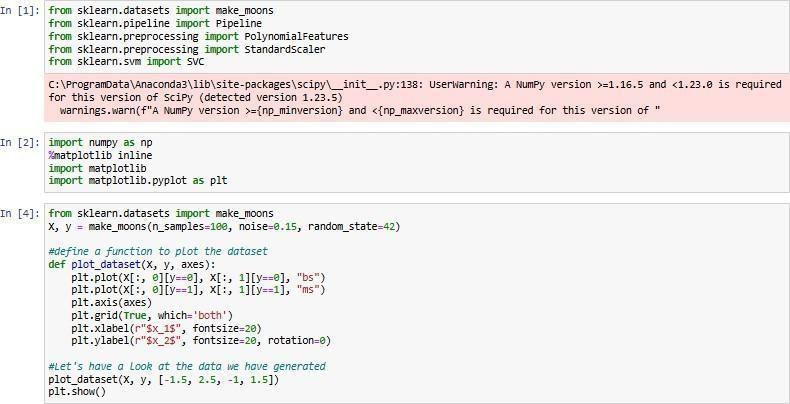
**Kernel Trick:**

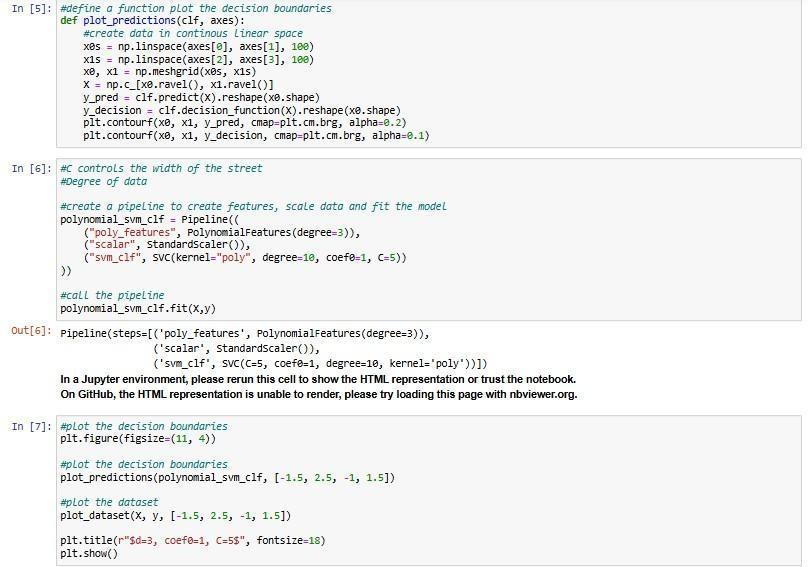
Mathematical technique used to project data into a higher-dimensional space without explicitly computing coordinates.

Common kernels:

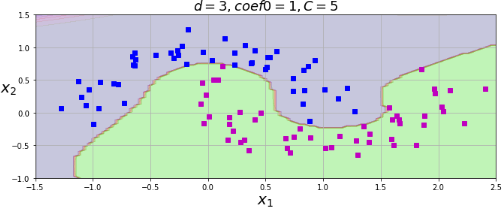
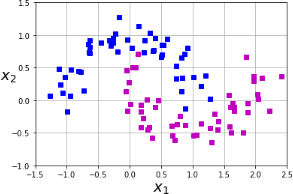
* Polynomial
* Radial Basis Function (RBF/Gaussian)

#### Code:





**Output:**



**Conclusion:**Successfully demonstrated Non-Linear SVM.

**Practical No. 12 Aim:**Implementation of k-means algorithm using Elbow method **Objectives:**

* Understandingof k-means algorithm.
* Understanding of Elbow method
* Understanding of Elbow method for k-means
* Implementation of k-means using elbow method

**Theory:**

**K-means:-**

**What is K-Means Clustering?**

K-Means is an unsupervised machine learning algorithm used to group similar data points into k clusters. It finds groupings by minimizing the distance between points and their corresponding cluster center (centroid).

#### Example

Suppose we have the following 6 data points:

(1, 2), (1, 4), (1, 0), (10, 2), (10, 4), (10, 0)

And we want to group them into **2 clusters (k=2)**.

#### 🔹 Steps of K-Means

**Step 1: Choose the number of clusters k**

Let’s choose **k = 2**

#### Step 2: Initialize centroids randomly

Pick two random points as initial centroids. For example: Centroid 1 = (1, 2)

Centroid 2 = (10, 2)

#### Step 3: Assign points to the nearest centroid

Calculate the Euclidean distance between each point and the centroids:

* (1, 2) → closer to (1, 2)
* (1, 4) → closer to (1, 2)
* (1, 0) → closer to (1, 2)
* (10, 2) → closer to (10, 2)
* (10, 4) → closer to (10, 2)
* (10, 0) → closer to (10, 2)

So, the clusters are:

Cluster 1: (1, 2), (1, 4), (1, 0)

Cluster 2: (10, 2), (10, 4), (10, 0)

#### Step 4: Recalculate the centroids

Take the average of the points in each cluster:

* Cluster 1 centroid:

((1+1+1)/3,(2+4+0)/3)=(1,2)( (1+1+1)/3 , (2+4+0)/3 ) = (1, 2)

* Cluster 2 centroid:

((10+10+10)/3,(2+4+0)/3)=(10,2)( (10+10+10)/3 , (2+4+0)/3 ) = (10, 2)

Centroids didn’t change, so the algorithm converged!

#### 🔹 Final Output

Two clusters:

* Cluster 1: All points with x = 1
* Cluster 2: All points with x = 10

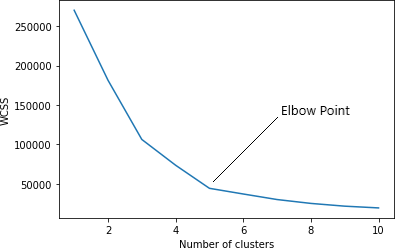
#### Elbow method:-

The Elbow Method is a graphical approach used to determine the optimal number of clusters (k) in K-Means Clustering.

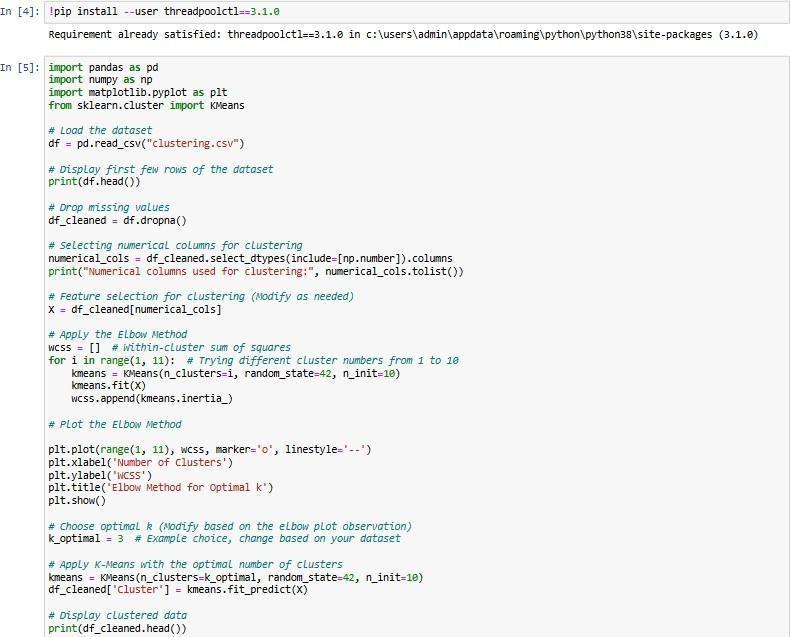
### 🔹 Step-by-Step Explanation:

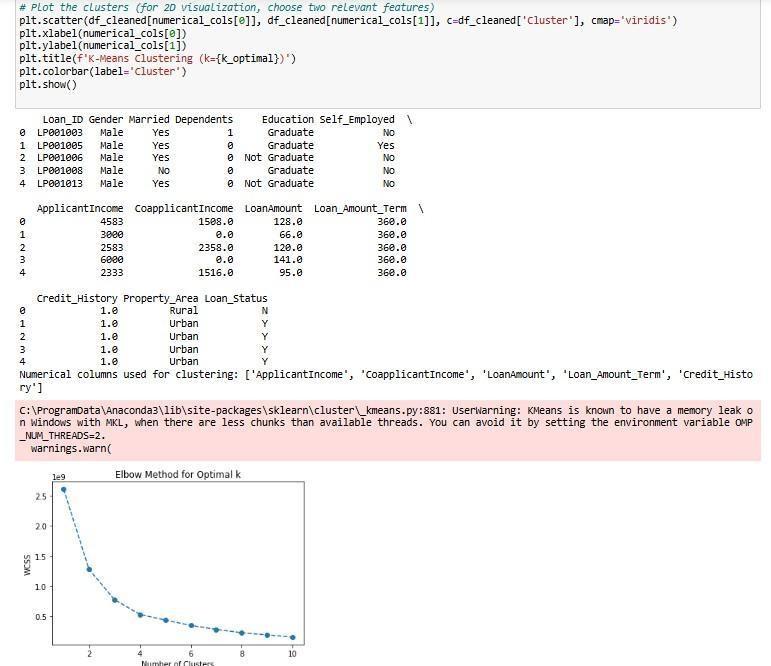
1. **Run K-Means** clustering for a range of k values (e.g., 1 to 10).
2. For each k, calculate the **WCSS (Within-Cluster Sum of Squares)**:
   * It measures the **compactness** of the clusters.
   * Lower WCSS = tighter, more defined clusters.
3. Plot **k vs. WCSS** on a graph.
4. Look for the point where the **rate of decrease sharply changes** — this is the **"elbow" point**.
5. The elbow indicates the **optimal number of clusters** — adding more clusters beyond this doesn’t improve the model much.

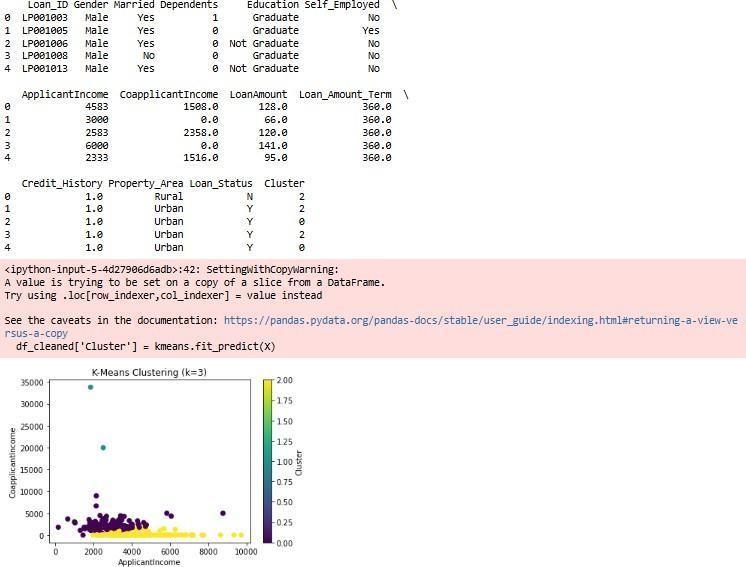
#### Diagram for Elbow Method:



**Code and Output:**







**Conclusion:**Successfully applied Elbow method and Implemented k-means algorithm.

**Practical No. 13 Aim:**Implementation of Bagging algorithm:-Random Forest **Objectives:**

* Understanding of bagging
* Understanding of Random forest
* Implementation of Random Forest

**Theory:**

1. **Bagging (Bootstrap Aggregating):**

**Definition:**

Bagging is a machine learning ensemble technique that aims to improve the stability and accuracy of algorithms by reducing variance and overfitting. It works by training multiple models (typically the same type of model) on different random subsets of the training data, then combining their predictions to make a final decision. The main idea is to average the predictions (in regression) or take a vote (in classification) to achieve a more robust model.

#### How Bagging Works:

* + **Bootstrap Sampling:** Bagging uses a technique called **bootstrap sampling** to create multiple subsets of the original dataset. Each subset is created by sampling data points **with replacement**, meaning that some points may appear multiple times in a subset, and some may not appear at all.
  + **Model Training:** Each model in the ensemble is trained on one of these bootstrap samples.

#### Aggregating Results:

* + - For regression problems, the final prediction is typically the **average** of all the model predictions.
    - For classification problems, the final prediction is made by taking the

**mode (majority vote)** of all the model predictions.

#### Advantages of Bagging:

* + **Reduces Overfitting:** By averaging multiple models, bagging reduces the likelihood of overfitting, especially with high-variance models like decision trees.
  + **Improves Accuracy:** The averaging or voting process generally results in a more accurate prediction than a single model.

**Example of Bagging:** Imagine you have a dataset to predict house prices (a regression problem). You could use bagging as follows:

1. **Step 1:** Create multiple bootstrap samples of the original dataset. For example, you might create five different subsets of the original data by sampling with replacement.
2. **Step 2:** Train a regression model (like a decision tree) on each of these bootstrap samples.
3. **Step 3:** Make predictions using each of the trained models.
4. **Step 4:** Average the predictions from each model to get the final predicted house price.

#### Random Forest:

**Definition:**

Random Forest is an extension of bagging that uses decision trees as the base learners and introduces an additional level of randomness in the model training process. It is one of the most powerful and widely used machine learning algorithms, especially for classification and regression problems.

**How Random Forest Works:** Random Forest is built on top of the bagging concept but adds an extra randomization step when splitting the nodes in the decision trees. Instead of considering all the features when making a split, Random Forest only considers a random subset of features at each node. This helps make the trees more diverse and reduces correlation between them, leading to a more robust ensemble model.

* + **Bootstrap Sampling:** Similar to bagging, Random Forest uses bootstrap sampling to create different subsets of the training data.
  + **Random Feature Selection:** When constructing each decision tree, instead of evaluating all features at each split, Random Forest randomly selects a subset of features to consider for splitting. This ensures that the trees are diverse and helps reduce overfitting.
  + **Tree Construction:** Each tree in the forest is trained on a different bootstrap sample, and at each node, only a subset of features is used for the split. This leads to more diverse trees in the forest.

#### Aggregating Results:

* + - For regression problems, the final prediction is the **average** of the predictions from all the trees.
    - For classification problems, the final prediction is the **majority vote** from all the trees.

#### Advantages of Random Forest:

* + **High Accuracy:** Random Forest generally provides very high accuracy due to the diversity of the trees and the random feature selection.
  + **Robust to Overfitting:** By averaging multiple trees, Random Forest reduces the risk of overfitting, even with deep trees.
  + **Handles High-Dimensional Data:** Random Forest is good at handling datasets with many features, as it performs random feature selection.

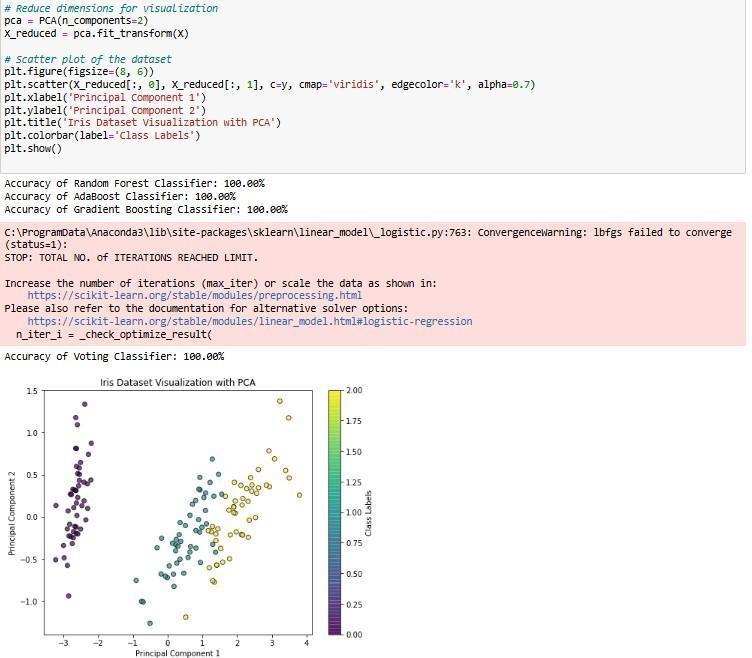
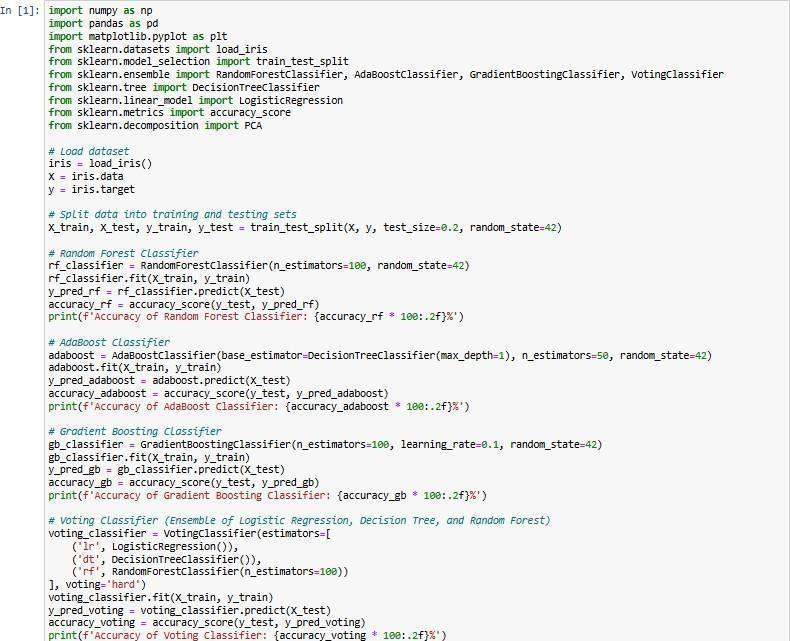
#### Example of Random Forest:

Let's say you're working with a dataset for classifying whether a customer will buy a product (binary classification).

1. **Step 1:** Create multiple bootstrap samples from the original dataset.
2. **Step 2:** For each bootstrap sample, train a decision tree. However, when constructing each tree, at each decision point (node), only a random subset of features is considered for splitting the data.
3. **Step 3:** Once all trees are built, use them to make predictions on new data. For each new observation, each tree in the forest provides a classification (e.g., "Will Buy" or "Won't Buy").
4. **Step 4:** The final classification is determined by a **majority vote** among all the trees.

For example, if you have 100 trees in the forest and 60 trees predict "Will Buy" while 40 trees predict "Won't Buy," the final prediction will be "Will Buy."

#### Code and Output:



**Conclusion:**Successfully Implemented a Bagging algorithm:Random Forest

# Practical No. 14

**Aim:**Implementation of boosting algorithm AdaBoost, stochastic gradient boosting and voting ensemble

#### Objectives:

* Understanding the concept of boosting
* Understanding of Adaboost algorithm
* Understanding of Ensemble learning Method
* Implementation of boosting algorithm

#### Theory:

**What is Boosting?**

**Boosting** is a machine learning technique used to improve the accuracy of a model by combining several weak models to create a stronger model. The key idea behind boosting is to focus on the mistakes made by previous models and correct them in subsequent iterations. This process is done by adjusting the weights of the misclassified data points, so the model learns to focus more on them.

In boosting, each model is built sequentially, and each new model tries to correct the errors made by the previous ones. Boosting is particularly effective because it helps improve the performance of weak models (e.g., decision trees with limited depth).

#### What is Ensemble Learning?

**Ensemble learning** is a method that combines multiple individual models (learners) to produce a stronger and more accurate prediction than any single model could achieve alone. The idea is that combining several models can reduce variance (overfitting) and bias (underfitting) while improving the overall predictive performance.

There are two main types of ensemble learning:

1. **Bagging** (Bootstrap Aggregating): Multiple models are trained independently on different subsets of the data, and their predictions are averaged or voted on (e.g., Random Forest).
2. **Boosting**: Models are trained sequentially, with each new model trying to correct the mistakes of the previous one (e.g., AdaBoost, Gradient Boosting).

#### AdaBoost with Example:

**AdaBoost** (Adaptive Boosting) is one of the most popular boosting algorithms. It combines weak learners (usually decision trees) to create a strong classifier. In AdaBoost, each new model focuses on the errors of the previous models.

#### Steps in AdaBoost:

1. Start with a weak model (e.g., a decision stump, which is a decision tree of height 1).
2. Assign equal weights to all training samples.
3. Train the weak model on the weighted dataset.
4. Increase the weights of the misclassified samples, so the next model focuses more on them.
5. Add the new model to the ensemble.
6. Repeat the process for a fixed number of iterations or until the model reaches the desired accuracy.

#### Example:

Imagine you have a dataset of email messages that need to be classified as spam or not spam.

1. **First iteration**: You train the first decision stump (weak model). It correctly classifies most of the emails but makes some mistakes.
2. **Weight update**: Increase the weights for the misclassified emails, so the next model will focus more on them.
3. **Second iteration**: A second decision stump is trained, and it focuses on the mistakes made by the first stump.
4. **Repeat**: This process continues until a pre-specified number of weak models are created.

Finally, AdaBoost combines the predictions of all the weak learners to make the final prediction, often by taking a weighted vote of the models' predictions.

#### Voting Ensemble Method:

The **Voting Ensemble** method combines multiple models to improve prediction accuracy by taking a vote (majority rule) for classification problems or averaging predictions for regression problems.

There are two main types of voting:

1. **Hard Voting**: The final prediction is based on the majority vote of the individual models. Each model casts a vote for a class, and the class with the most votes is chosen as the final prediction.
2. **Soft Voting**: Instead of hard class labels, models predict probabilities for each class. The probabilities from all models are averaged, and the class with the highest average probability is chosen.

#### Example of Voting Ensemble (Classification):

Suppose you have three classifiers: a Decision Tree, a K-Nearest Neighbors (KNN) model, and a Logistic Regression model.

* + Model 1 (Decision Tree) predicts: **Class A**
  + Model 2 (KNN) predicts: **Class B**
  + Model 3 (Logistic Regression) predicts: **Class A**

**Hard Voting**: Since Class A has the majority of votes (2 out of 3), the final prediction will be

#### Class A.

**Soft Voting**: If the models provide probabilities like:

* + Decision Tree: Class A: 0.7, Class B: 0.3
  + KNN: Class A: 0.4, Class B: 0.6
  + Logistic Regression: Class A: 0.5, Class B: 0.5

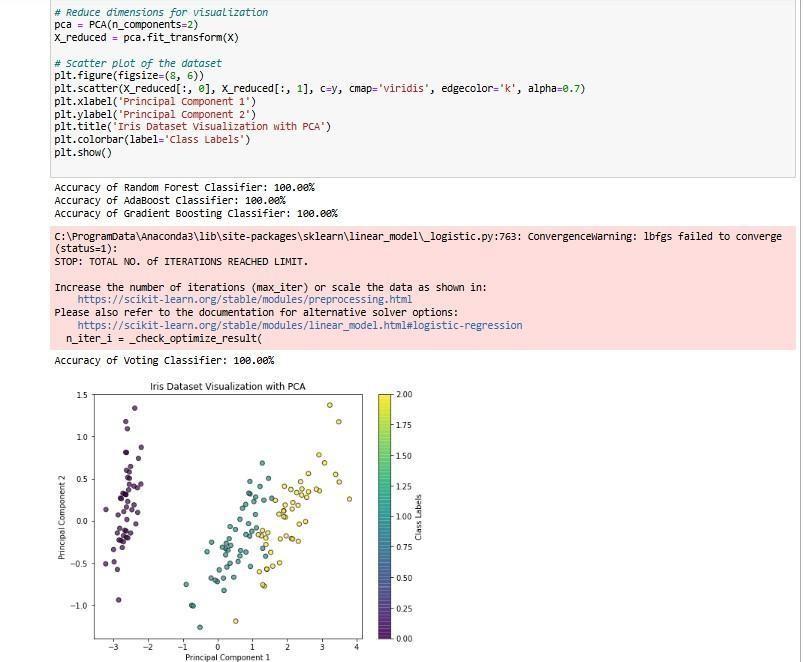
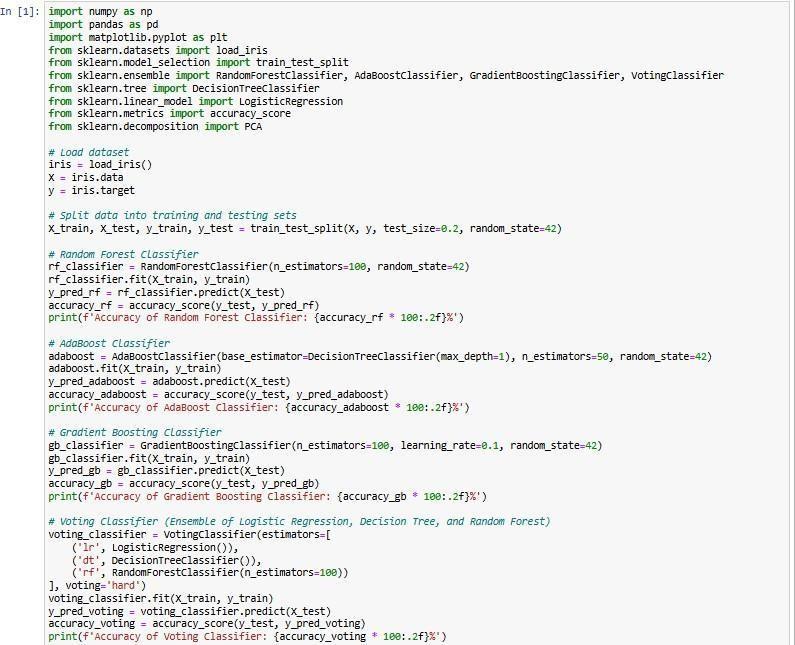
The average probabilities would be:

* + Class A: (0.7 + 0.4 + 0.5) / 3 = 0.53
  + Class B: (0.3 + 0.6 + 0.5) / 3 = 0.47

In this case, the final prediction would be **Class A**, because it has the highest average probability. To sum up:

* + **Boosting** focuses on correcting errors from previous models.
  + **Ensemble learning** combines multiple models to improve performance.
  + **AdaBoost** is a boosting method that adapts based on the errors of previous models.
  + **Voting Ensemble** combines predictions from different models using majority voting (hard voting) or probability averaging (soft voting).

#### Code & Output:



**Conclusion:**Successfully understood and implemented various Boosting algorithm