

**22CS67L**

Machine Learning Laboratory Manual

**Prof: Divya H N**

**Course instructor**

**Dept of Computer Science**

**Contents**

1. **Python Programming Basics**
2. **N-Dimensional Data Visualization**
3. **Hill Climbing Algorithm Implementation**
4. **Informed Search Algorithms**
   * Best-First Search Implementation
   * A\* Algorithm Implementation
5. **Game Theory Algorithms**
   * Minimax Algorithm Implementation
   * Alpha-Beta Pruning Implementation
6. **Naïve Bayes Classifier Implementation**
7. **K-Nearest Neighbors (KNN) Implementation**
8. **K-Means Clustering Implementation**
9. **Agglomerative Clustering Implementation**
10. **Principal Component Analysis (PCA) Implementation**
11. **Linear Discriminant Analysis (LDA) Implementation**
12. **Single-Layer Perceptron for Boolean Functions (AND, OR)**
13. **Simple python program using conditional statements, looping, performing operations such as Insert, Update, Delete, Display, Sorting and searching on data types like List, Tuple, Set, Dictionary.**

**Code:**

**def** search(data):

x**=**input("Enter search element: ")

**for** i **in** range(len(data)):

**if** data[i]**==**x:

print(f"Element found at {i}")

**return**

print("Element not Found!!")

**def** sort(data):

**for** i **in** range(len(data)**-**1):

flag**=False**

**for** j **in** range(len(data)**-**i**-**1):

**if** data[j]**>**data[j**+**1]:

temp**=**data[j]

data[j]**=**data[j**+**1]

data[j**+**1]**=**temp

flag**=True**

**if** **not** flag:

**return**

In [2]:

**def** update(data,key):

**if** key **not** **in** data:

print("Key not found")

**else**:

x**=**input("Enter a new value: ")

data[key]**=**x

In [3]:

list\_**=**[]

**def** list\_op():

print("List operations: ")

print("1.Insert\n2.Delete\n3.Update\n4.Search\n5.Sort\n6.Display\n7.Return to main")

**while** **True**:

ch**=**int(input("Enter choice: "))

**if** ch**==**1:

x**=**input("Enter element: ")

list\_**.**append(x)

**elif** ch**==**2:

ele**=**input("Enter element to be deleted: ")

**try**:

list\_**.**remove(ele)

**except**:

print(f"Element {ele} not found")

**elif** ch**==**3:

index**=**int(input("Enter index: "))

update(list\_,index)

**elif** ch**==**4:

search(list\_)

**elif** ch**==**5:

sort(list\_)

**elif** ch**==**6:

print(list\_)

**elif** ch**==**7:

**break**

**else**:

print("Invalid choice")

tup**=**tuple()

**def** tuple\_op():

print("Enter tuple: ")

tup**=**tuple(i **for** i **in** input()**.**split())

print("1.Display\n2.Search\n3.Exit")

**while** **True**:

ch**=**int(input("Enter the choice"))

**if** ch**==**1:

print(tup)

**elif** ch**==**2:

search(tup)

**elif** ch**==**3:

**break**

**else**:

Print("Invalid choice")

myset**=**set()

**def** set\_op():

print("1.Add\n2.Remove\n3.Search\n4.Display\n5.exit")

**while** **True**:

ch**=**int(input("Enter choice: "))

**if** ch**==**1:

myset**.**add(input("Enter element: "))

**elif** ch**==**2:

**try**:

myset**.**remove(input("Enter element: "))

**except**:

print("Element not found")

**elif** ch**==**3:

search(myset)

**elif** ch**==**4:

print(myset)

**elif** ch**==**5:

**return**

**else**:

print("Invalid choice")

In [6]:

dict\_**=**dict()

**def** dict\_operation():

print("1.Insert\n2.Update\n3.Delete\n4.Search\n5.Display\n6.exit")

**while** **True**:

ch**=**int(input("Enter your choice"))

**if** ch**==**1:

key**=**input("Enter key: ")

value**=**input("Enter value: ")

dict\_[key]**=**value

**elif** ch**==**2:

key**=**input("Enter the key: ")

update(dict\_,key)

**elif** ch**==**3:

key**=**input("Enter key to be deleted: ")

**if** key **in** dict\_:

**del** dict\_[key]

**else**:

print("Key not found")

**elif** ch**==**4:

ele**=**input("Enter value: ")

**for** key,value **in** dict**.**items():

**if** value**==**ele:

print(key)

**elif** ch**==**5:

print(dict\_)

**elif** ch**==**6:

**break**

**else**:

print("Invalid choice")

**while** **True**:

print("1.List\n2.Tuples\n3.Set\n4.Dictonary\n5.Exit")

ch**=**int(input("Enter choice: "))

**if** ch**==**1:

list\_op()

**elif** ch**==**2:

tuple\_op()

**elif** ch**==**3:

set\_op()

**elif** ch**==**4:

dict\_operation()

**elif** ch**==**5:

**break**

**else**:

print("Invalid choice")

**output:**

1.List

2.Tuples

3.Set

4.Dictonary

5.Exit

List operations:

1.Insert

2.Delete

3.Update

4.Search

5.Sort

6.Display

7.Return to main

Enter choice: 1

Enter element: 9

Enter choice: 2

Enter element to be deleted: 1

Element 1 not found

Enter choice: 6

['9']

Enter choice: 1

Enter element: 4

Enter choice: 1

Enter element: 5

Enter choice: 1

Enter element: 3

Enter choice: 1

Enter element: 7

Enter choice: 1

Enter element: 10

Enter choice: 6

['9', '4', '5', '3', '7', '10']

Enter choice: 2

Enter element to be deleted: 4

Enter choice: 6

['9', '5', '3', '7', '10']

**2. Visualize the n-dimensional data using Scatter plots, box plot, heat maps, contour plots, 3D surface plots using python packages.**

**Code:**

**import** pandas **as** pd

**import** numpy **as** np

**import** seaborn **as** sns

**import** matplotlib.pyplot **as** plt

**Importing the dataset:**

data **=** pd**.**read\_csv("xyz\_data.csv")

X **=** data['X']

Y **=** data['Y']

Z **=** data['Z']

**Scatter plot:**

plt**.**scatter(X,Y)

plt**.**xlabel('X')

plt**.**ylabel('Y')

plt**.**title('Scatter Plot')

**Box Plot:**

sns**.**boxplot(data**=**data)

plt**.**title('Box Plot')

**Heat Map:**

sns**.**heatmap(data**.**corr(),annot**=True**,cmap**=**'coolwarm')

plt**.**title('Heat Map')

**Contour Plot:**

plt**.**tricontour(X,Y,Z,cmap**=**'jet')

plt**.**xlabel('X')

plt**.**ylabel('Y')

plt**.**title('Contour Plot')

**3D Surface Plot:**

ax **=** plt**.**axes(projection**=**"3d")

ax**.**plot\_trisurf(X,Y,Z,cmap**=**'jet\_r')

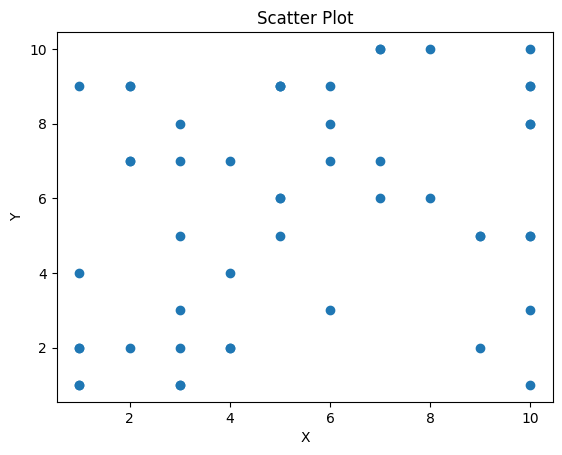
ax**.**set\_xlabel('X')

ax**.**set\_ylabel('Y')

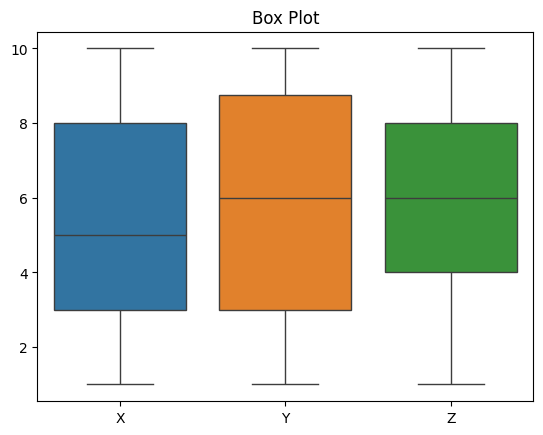
ax**.**set\_title('3D Surface Plot')

**Output:**

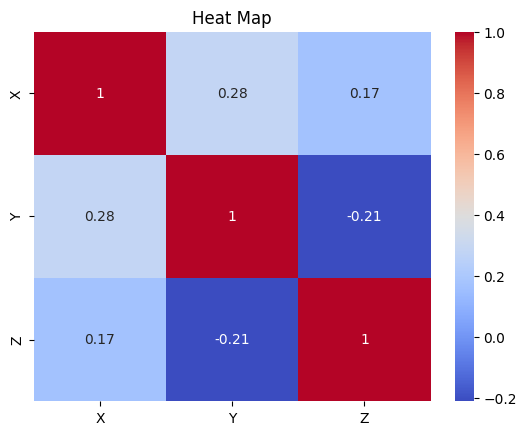
**Scatter Plot:**



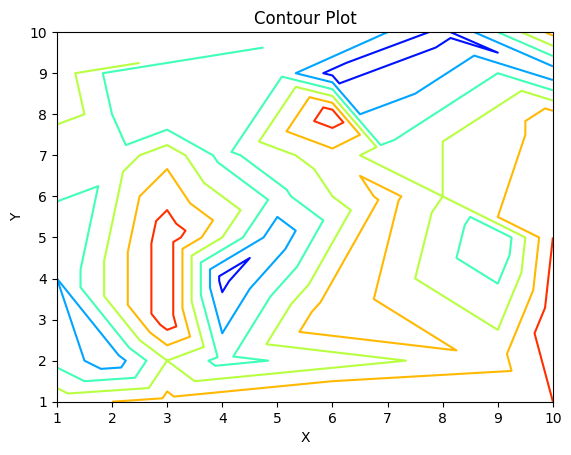
**Box Plot:**



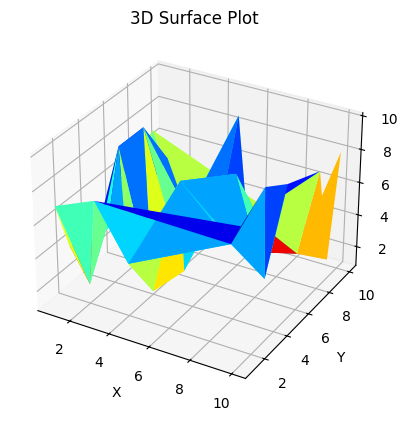
**Heat Map:**



**Contour Plot:**



**3D Surface Plot:**



**3.Write a program to implement Hill Climbing Algorithm.**

**Code:**

**import math**

**import random**

**def objective\_function(x):**

**return math.sin(x)**

**def hill\_climbing(start, step\_size, max\_iterations):**

**current = start**

**current\_value = objective\_function(current)**

**for i in range(max\_iterations):**

**next\_point = current + step\_size**

**next\_value = objective\_function(next\_point)**

**if next\_value > current\_value:**

**current = next\_point**

**current\_value = next\_value**

**else:**

**break**

**return current, current\_value**

**start = random.uniform(0, 2 \* math.pi) # Random starting point**

**step\_size = 0.01**

**max\_iterations = 1000**

**best\_solution, best\_value = hill\_climbing(start, step\_size, max\_iterations)**

**print(f"Best solution: x = {best\_solution}")**

**print(f"Objective function value: {best\_value}")**

**Output:**

**Best solution: x = 1.571807505057998**

**Objective function value: 0.9999994887593037**

**4. a) Write a program to implement the Best First Search (BFS) algorithm.**

**b) Write a program to implement the A\* algorithm.**

**4a. Best First Search algorithm: -**

**Code:**

**def** best\_first\_search(graph,start,goal,heuristic, path**=**[]):

open\_list **=** [(0,start)]

closed\_list **=** set()

closed\_list**.**add(start)

**while** open\_list:

open\_list**.**sort(key **=** **lambda** x: heuristic[x[1]], reverse**=True**)

cost, node **=** open\_list**.**pop()

path**.**append(node)

**if** node **= =** goal:

**return** cost, path

closed\_list**.**add(node)

**for** neighbour, neighbour\_cost **in** graph[node]:

**if** neighbour **not** **in** closed\_list:

closed\_list**.**add(node)

open\_list**.**append((cost**+**neighbour\_cost, neighbour))

**return** **None**

graph **=** {

'A': [('B', 11), ('C', 14), ('D',7)],

'B': [('A', 11), ('E', 15)],

'C': [('A', 14), ('E', 8), ('D',18), ('F',10)],

'D': [('A', 7), ('F', 25), ('C',18)],

'E': [('B', 15), ('C', 8), ('H',9)],

'F': [('G', 20), ('C', 10), ('D',25)],

'G': [],

'H': [('E',9), ('G',10)]

}

start **=** 'A'

goal **=** 'G'

heuristic **=** {

'A': 40,

'B': 32,

'C': 25,

'D': 35,

'E': 19,

'F': 17,

'G': 0,

'H': 10

}

result **=** best\_first\_search(graph, start, goal, heuristic)

**if** result:

print(f"Minimum cost path from {start} to {goal} is {result[1]}")

print(f"Cost: {result[0]}")

**else**:

print(f"No path from {start} to {goal}")

**Output: -**

Minimum cost path from A to G is ['A', 'C', 'F', 'G']

Cost: 44

**5.** **Write a program to implement Min-Max algorithm and Alpha-beta pruning algorithm.**

**Min-Max algorithm: -**

**class** TreeNode:

**def** \_\_init\_\_(self,value,children**=**[]):

self**.**value**=**value

self**.**children**=**children

**def** minimax(node,depth,maximizing\_player):

**if** depth**==**0 **or** **not** node**.**children:

**return** node**.**value,[node**.**value]

**if** maximizing\_player:

max\_value**=**float("-inf")

max\_path**=**[]

**for** child\_node **in** node**.**children:

child\_value,child\_path**=**minimax(child\_node,depth**-**1,**False**)

**if** child\_value**>**max\_value:

max\_value**=**child\_value

max\_path**=**[node**.**value]**+**child\_path

**return** max\_value,max\_path

**else**:

min\_value**=**float("inf")

min\_path**=**[]

**for** child\_node **in** node**.**children:

child\_value,child\_path**=**minimax(child\_node,depth**-**1,**True**)

**if** child\_value**<**min\_value:

min\_value**=**child\_value

min\_path**=**[node**.**value]**+**child\_path

**return** min\_value,min\_path

game\_tree**=**TreeNode(0,[

TreeNode(1,[TreeNode(3),TreeNode(12)]),

TreeNode(4,[TreeNode(8),TreeNode(2)])

])

optimal\_value,optimal\_path**=**minimax(game\_tree,2,**True**)

print("Optimal value: ",optimal\_value)

print("Optimal path: ",optimal\_path)

**Output: -**

**Optimal value:** 3

**Optimal path:** [0, 1, 3]

**Alpha-beta pruning algorithm: -**

**class** TreeNode:

**def** \_\_init\_\_(self,value,children**=**[]):

self**.**value**=**value

self**.**children**=**children

self**.**alpha**=**float("-inf")

self**.**beta**=**float("inf")

**def** alpha\_beta(node,depth,alpha,beta,maximizing\_player):

**global** pruned\_count

**if** depth**==**0 **or** **not** node**.**children:

**return** node**.**value,[node**.**value]

**if** maximizing\_player:

max\_value**=**float("-inf")

max\_path**=**[]

**for** child\_node **in** node**.**children:

child\_value,child\_path**=**alpha\_beta(child\_node,depth**-**1,alpha,beta,**False**)

**if** child\_value**>**max\_value:

max\_value**=**child\_value

max\_path**=**[node**.**value]**+**child\_path

alpha**=**max(alpha,max\_value)

**if** alpha**>=**beta:

pruned\_count**+=**len(child\_node**.**children)**+**1

**break**

**return** max\_value,max\_path

**else**:

min\_value**=**float("inf")

min\_path**=**[]

**for** child\_node **in** node**.**children:

child\_value,child\_path**=**alpha\_beta(child\_node,depth**-**1,alpha,beta,**True**)

**if** child\_value**<**min\_value:

min\_value**=**child\_value

min\_path**=**[node**.**value]**+**child\_path

beta**=**min(beta,min\_value)

**if** alpha**>=**beta:

pruned\_count**+=**len(child\_node**.**children)**+**1

**break**

**return** min\_value,min\_path

**def** alpha\_beta(node,depth,alpha,beta,maximizing\_player):

**global** pruned\_count

**if** depth**==**0 **or** **not** node**.**children:

**return** node**.**value,[node**.**value]

**if** maximizing\_player:

max\_value**=**float("-inf")

max\_path**=**[]

**for** child\_node **in** node**.**children:

child\_value,child\_path**=**alpha\_beta(child\_node,depth**-**1,alpha,beta,**False**)

**if** child\_value**>**max\_value:

max\_value**=**child\_value

max\_path**=**[node**.**value]**+**child\_path

alpha**=**max(alpha,max\_value)

**if** alpha**>=**beta:

pruned\_count**+=**len(child\_node**.**children)**+**1

**break**

**return** max\_value,max\_path

**else**:

min\_value**=**float("inf")

min\_path**=**[]

**for** child\_node **in** node**.**children:

child\_value,child\_path**=**alpha\_beta(child\_node,depth**-**1,alpha,beta,**True**)

**if** child\_value**<**min\_value:

min\_value**=**child\_value

min\_path**=**[node**.**value]**+**child\_path

beta**=**min(beta,min\_value)

**if** alpha**>=**beta:

pruned\_count**+=**len(child\_node**.**children)**+**1

**break**

**return** min\_value,min\_path

**Output: -**

**Pruned Count: 1**

**Optimal value: 3**

**Optimal path: [0, 1, 3]**

**6. Write a program to develop the Naive Bayes classifier based on split up of training and testing dataset as 90-10, 70-30.**

**a) Iris dataset**

**b) Titanic dataset**

**code:**

**import** numpy **as** np

**import** pandas **as** pd

**from** sklearn.model\_selection **import** train\_test\_split,cross\_val\_score

**from** sklearn.metrics **import** accuracy\_score, confusion\_matrix

**import** seaborn **as** sns

**import** matplotlib.pyplot **as** plt

In [26]:

**class** NaiveBayesclassifier:

**def** \_\_init\_\_(self):

self**.**class\_probabilities**=**{}

self**.**feature\_probabilities**=**{}

**def** fit(self,x,y):

n\_samples,n\_features**=**x**.**shape

classes**=**np**.**unique(y)

**for** class\_label **in** classes:

class\_indices**=**np**.**where(y**==**class\_label)

class\_probability**=**len(class\_indices[0])**/**n\_samples

self**.**class\_probabilities[class\_label]**=**class\_probability

class\_data**=**x[class\_indices]

feature\_probabilities**=**{}

**for** feature **in** range(n\_features):

unique\_values,counts**=**np**.**unique(class\_data[:,feature],return\_counts**=True**)

feature\_probabilities[feature]**=**{'value':unique\_values,'probabilities':counts**/**len(class\_indices[0])}

self**.**feature\_probabilities[class\_label]**=**feature\_probabilities

**def** predict(self,x):

predictons**=**[]

**for** sample **in** x:

max\_prob**=-**1

predicted\_class**=None**

**for** class\_label,class\_probability **in** self**.**class\_probabilities**.**items():

posterior**=**class\_probability

**for** feature,feature\_value **in** enumerate(sample):

**if** feature\_value **in** self**.**feature\_probabilities[class\_label][feature]['value']:

feature\_probability\_index**=**np**.**where(self**.**feature\_probabilities[class\_label][feature]['value']**==**feature\_value)[0][0]

feature\_probability**=**self**.**feature\_probabilities[class\_label][feature]['probabilities'][feature\_probability\_index]

posterior**\*=**feature\_probability

**else**:

posterior**\*=**0.01

**if** posterior**>**max\_prob:

max\_prob**=**posterior

predicted\_class**=**class\_label

predictons**.**append(predicted\_class)

**return** predictons

**def** evaluate\_naive\_bayes(dataset,target\_col,K\_values):

**for** K **in** K\_values:

print(f"\nResults for K={K}")

x\_train,x\_test,y\_train,y\_test**=**train\_test\_split(dataset**.**drop(columns**=**[target\_col]),dataset[target\_col],test\_size**=**(1**-**K**/**10),random\_state**=**42)

Clf**=**NaiveBayesclassifier()

Clf**.**fit(x\_train**.**values,y\_train**.**values)

y\_pred**=**Clf**.**predict(x\_test**.**values)

accuracy**=**accuracy\_score(y\_test,y\_pred)

cm **=** confusion\_matrix(y\_test, y\_pred)

sns**.**heatmap(cm, annot**=True**, cmap**=**'Blues')

plt**.**show()

print(f"Split {K**\***10}-{100**-**K**\***10} Accuracy:{accuracy:.2f}")

iris\_data**=**pd**.**read\_csv('iris.csv')

titanic\_data**=**pd**.**read\_csv('titanic.csv')

In [28]:

titanic\_data**=**titanic\_data**.**drop(columns**=**['PassengerId','Name','Ticket','Cabin'])

titanic\_data['Age']**.**fillna(titanic\_data['Age']**.**median(),inplace**=True**)

titanic\_data['Embarked']**.**fillna(titanic\_data['Embarked']**.**mode()[0],inplace**=True**)

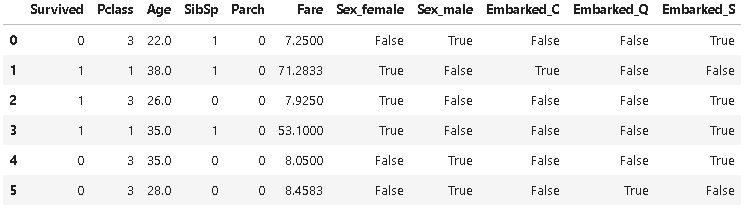
titanic\_data**=**pd**.**get\_dummies(titanic\_data,columns**=**['Sex','Embarked'])

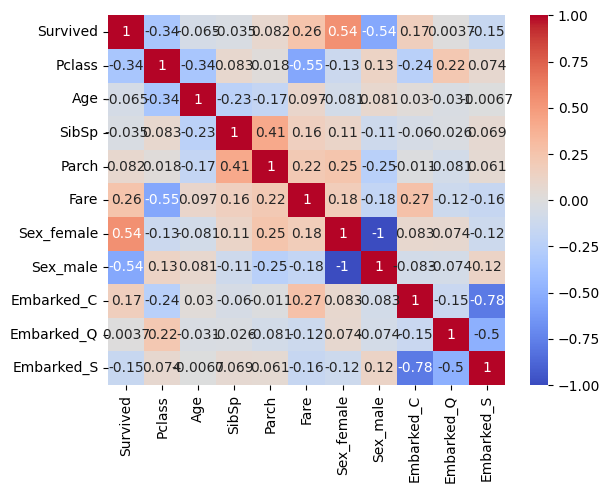
titanic\_data**.**head(6)

correlation\_matrix **=** titanic\_data**.**corr()

sns**.**heatmap(correlation\_matrix, annot**=True**, cmap**=**'coolwarm')

**Output:**



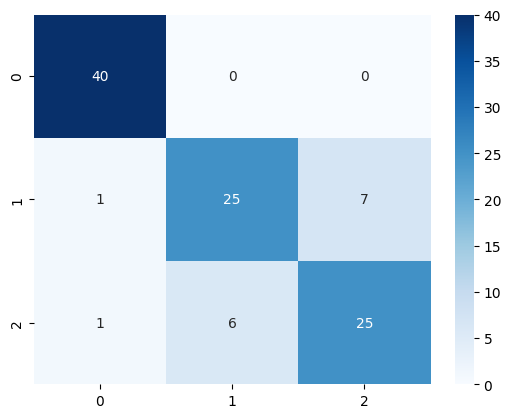


print("Iris Dataset Naive Bayes Classifier Results")

evaluate\_naive\_bayes(iris\_data,'Species',[3,5,7,9])

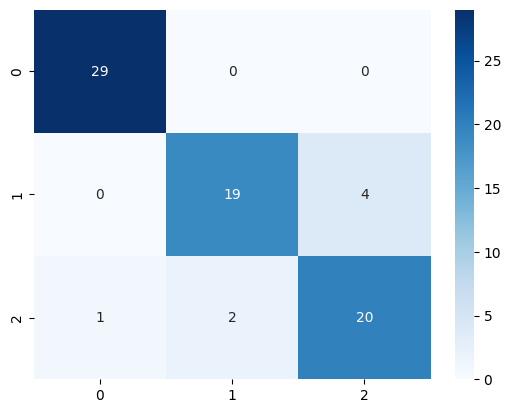
Iris Dataset Naive Bayes Classifier Results

Results for K=3



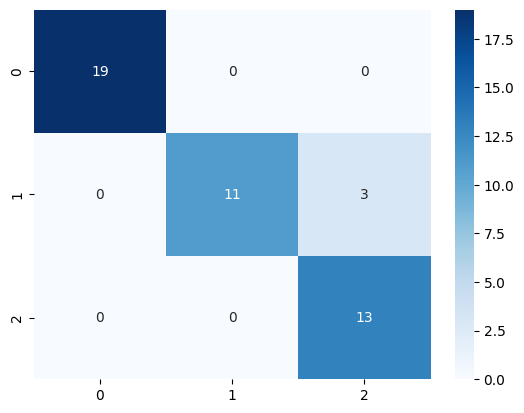
Split 30-70 Accuracy:0.86

Results for K=5



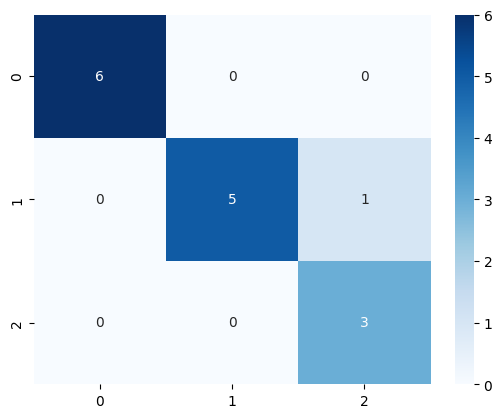
Split 50-50 Accuracy:0.91

Results for K=7



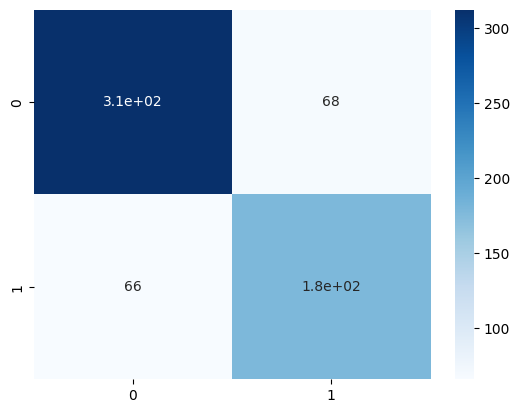
Split 70-30 Accuracy:0.93

Results for K=9



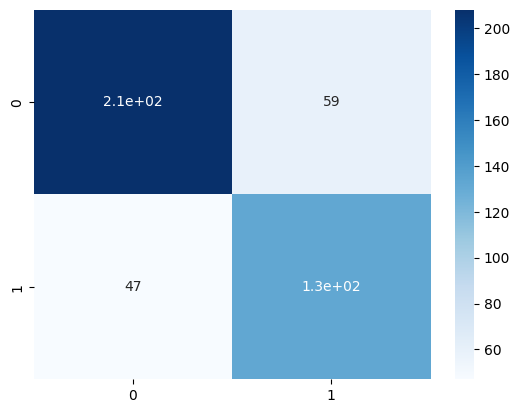
print("\nTitanic dataset Naive Bayes Classifier Results: ")

evaluate\_naive\_bayes(titanic\_data,'Survived',[3,5])



Split 30-70 Accuracy:0.79

Results for K=5



Split 70-30 Accuracy:0.79

**7. Write a program to develop the KNN classifier for the k values as 3,5,7 based on split up of training and testing dataset as 90-10, 70-30,**

**a) Glass dataset**

**b) Fruit dataset**

**using the different distance metrics like Euclidean and Manhattan distance.**

**Code:**

**import pandas as pd**

**from sklearn.model\_selection import train\_test\_split**

**from sklearn.neighbors import KNeighborsClassifier as KNN**

**from sklearn.metrics import accuracy\_score**

**In [2]:**

**glass\_data=pd.read\_csv("glass.csv")**

**fruit\_data=pd.read\_csv("fruits.csv")**

**datasets=['Glass','Fruit']**

**Xg = glass\_data.drop('Type',axis=1).values**

**yg = glass\_data['Type'].values**

**Xf = fruit\_data[['mass','width','height','color\_score']].values**

**yf = fruit\_data['fruit\_label'].values**

**In [3]:**

**k\_values=[3,5,7]**

**distance\_metrics= ['manhattan','euclidean']**

**test\_splits = [0.1,0.3]**

**In [4]:**

**for dataset in datasets:**

**print(f"Datasets:{dataset}")**

**for k in k\_values:**

**for p,distance\_metric in enumerate(distance\_metrics):**

**for ts in test\_splits:**

**print(f"K={k},Distance Metric={distance\_metric}: ")**

**print(f"Split = {ts}")**

**knn=KNN(n\_neighbors=k,p=1,weights="distance")**

**if dataset == 'Glass':**

**X\_train,X\_test, y\_train,y\_test=train\_test\_split(Xg,yg,test\_size=ts,random\_state=42)**

**else:**

**X\_train,X\_test, y\_train,y\_test=train\_test\_split(Xf,yf,test\_size=ts,random\_state=42)**

**knn.fit(X\_train,y\_train)**

**accuracy=knn.score(X\_test,y\_test)**

**print(f"accuracy on test set: {accuracy:.2f}")**

**print("------------")**

**print("----------------------------")**

**print("------------------------------------------------")**

**Output:**

**Datasets:Glass**

**K=3,Distance Metric=manhattan:**

**Split = 0.1**

**accuracy on test set: 0.77**

**------------**

**K=3,Distance Metric=manhattan:**

**Split = 0.3**

**accuracy on test set: 0.69**

**------------**

**----------------------------**

**K=3,Distance Metric=euclidean:**

**Split = 0.1**

**accuracy on test set: 0.77**

**------------**

**K=3,Distance Metric=euclidean:**

**Split = 0.3**

**accuracy on test set: 0.69**

**------------**

**----------------------------**

**K=5,Distance Metric=manhattan:**

**Split = 0.1**

**accuracy on test set: 0.77**

**------------**

**K=5,Distance Metric=manhattan:**

**Split = 0.3**

**accuracy on test set: 0.71**

**------------**

**----------------------------**

**K=5,Distance Metric=euclidean:**

**Split = 0.1**

**accuracy on test set: 0.77**

**------------**

**K=5,Distance Metric=euclidean:**

**Split = 0.3**

**accuracy on test set: 0.71**

**------------**

**----------------------------**

**K=7,Distance Metric=manhattan:**

**Split = 0.1**

**accuracy on test set: 0.73**

**------------**

**K=7,Distance Metric=manhattan:**

**Split = 0.3**

**accuracy on test set: 0.62**

**------------**

**----------------------------**

**K=7,Distance Metric=euclidean:**

**Split = 0.1**

**accuracy on test set: 0.73**

**------------**

**K=7,Distance Metric=euclidean:**

**Split = 0.3**

**accuracy on test set: 0.62**

**------------**

**----------------------------**

**------------------------------------------------**

**Datasets:Fruit**

**K=3,Distance Metric=manhattan:**

**Split = 0.1**

**accuracy on test set: 0.50**

**------------**

**K=3,Distance Metric=manhattan:**

**Split = 0.3**

**accuracy on test set: 0.67**

**------------**

**----------------------------**

**K=3,Distance Metric=euclidean:**

**Split = 0.1**

**accuracy on test set: 0.50**

**------------**

**K=3,Distance Metric=euclidean:**

**Split = 0.3**

**accuracy on test set: 0.67**

**------------**

**----------------------------**

**K=5,Distance Metric=manhattan:**

**Split = 0.1**

**accuracy on test set: 0.67**

**------------**

**K=5,Distance Metric=manhattan:**

**Split = 0.3**

**accuracy on test set: 0.67**

**------------**

**----------------------------**

**K=5,Distance Metric=euclidean:**

**Split = 0.1**

**accuracy on test set: 0.67**

**------------**

**K=5,Distance Metric=euclidean:**

**Split = 0.3**

**accuracy on test set: 0.67**

**------------**

**----------------------------**

**K=7,Distance Metric=manhattan:**

**Split = 0.1**

**accuracy on test set: 0.67**

**------------**

**K=7,Distance Metric=manhattan:**

**Split = 0.3**

**accuracy on test set: 0.56**

**------------**

**----------------------------**

**K=7,Distance Metric=euclidean:**

**Split = 0.1**

**accuracy on test set: 0.67**

**------------**

**K=7,Distance Metric=euclidean:**

**Split = 0.3**

**accuracy on test set: 0.56**

**------------**

**----------------------------**

**8. Write a program to perform unsupervised K-means clustering techniques**

**Code:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from sklearn.datasets import load\_iris**

**import warnings**

**warnings.filterwarnings('ignore')**

**In [2]:**

**def kmeans(X,K,max\_iters=100):**

**centroids=X[:K]**

**for \_ in range(max\_iters):**

**expanded\_x = X[:,np.newaxis]**

**euc\_dist = np.linalg.norm(expanded\_x-centroids,axis=2)**

**labels = np.argmin(euc\_dist,axis=1)**

**new\_centroids = np.array([X[labels==k].mean(axis=0) for k in range(K)])**

**if np.all(centroids==new\_centroids):**

**break**

**centroids = new\_centroids**

**return labels,centroids**

**In [3]:**

**X=load\_iris().data**

**K=3**

**labels,centroids=kmeans(X,K)**

**print("Labels: ",labels)**

**print("Centroids: ",centroids)**

**Labels: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2**

**2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**

**1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0**

**0 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0**

**0 1]**

**Centroids: [[6.85384615 3.07692308 5.71538462 2.05384615]**

**[5.88360656 2.74098361 4.38852459 1.43442623]**

**[5.006 3.428 1.462 0.246 ]]**

**plt.scatter(X[:,0],X[:,1],c=labels)**

**plt.scatter(centroids[:,0],centroids[:,1],marker='x',color='red',s=200)**

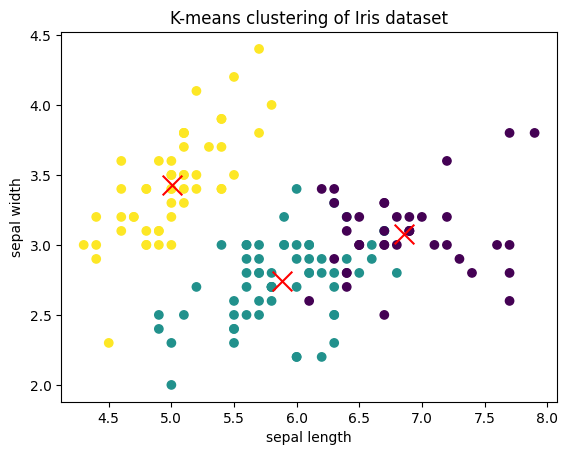
**plt.xlabel('sepal length')**

**plt.ylabel('sepal width')**

**plt.title('K-means clustering of Iris dataset')**

**plt.show()**

**Output:**



**9.Write a program to perform agglomerative clustering based on single linkage, complete-linkage criteria.**

**Code:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from scipy.cluster.hierarchy import dendrogram,linkage**

**from sklearn.datasets import load\_iris**

**data = load\_iris().data**

**data = data[:6]**

**In [2]:**

**def proximity\_matrix(data):**

**n = data.shape[0]**

**pm = np.zeros((n,n))**

**for i in range(n):**

**for j in range(i+1,n):**

**pm[i,j]=pm[j,i]=np.linalg.norm(data[i]-data[j])**

**return pm**

**In [3]:**

**def plot\_deno(data,method):**

**link\_mat = linkage(data,method=method)**

**dendrogram(link\_mat)**

**plt.title(method.capitalize()+" Linkage")**

**plt.show()**

**In [4]:**

**print('Proximity Matrix is : ')**

**print(proximity\_matrix(data))**

**Proximity Matrix is :**

**[[0. 0.53851648 0.50990195 0.64807407 0.14142136 0.6164414 ]**

**[0.53851648 0. 0.3 0.33166248 0.60827625 1.09087121]**

**[0.50990195 0.3 0. 0.24494897 0.50990195 1.08627805]**

**[0.64807407 0.33166248 0.24494897 0. 0.64807407 1.16619038]**

**[0.14142136 0.60827625 0.50990195 0.64807407 0. 0.6164414 ]**

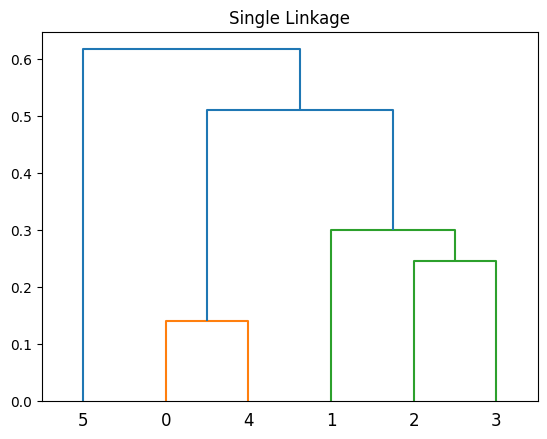
**[0.6164414 1.09087121 1.08627805 1.16619038 0.6164414 0. ]]**

**plot\_deno(data,'single')**

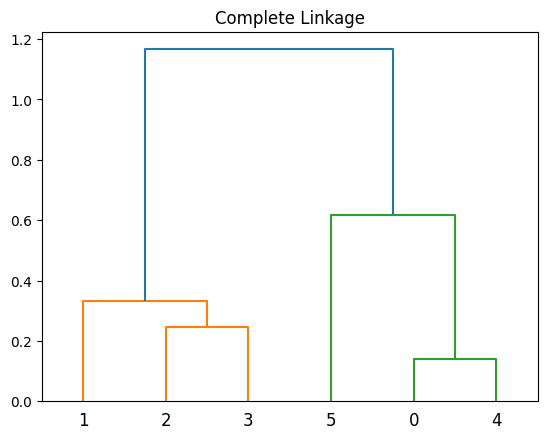
**plot\_deno(data,'complete')**

**Output:**

**Single Linkage**



**Complete Linkage**



**10. Write a program to develop Principal Component Analysis (PCA) algorithms.**

**Code:**

**import matplotlib.pyplot as plt**

**from sklearn.decomposition import PCA**

**from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis as LDA**

**from sklearn.datasets import load\_iris**

**In [2]:**

**X = load\_iris().data**

**y = load\_iris().target**

**print("Shape of Data:", X.shape)**

**plt.scatter(X[:,0], X[:,1], c=y, cmap="jet")**

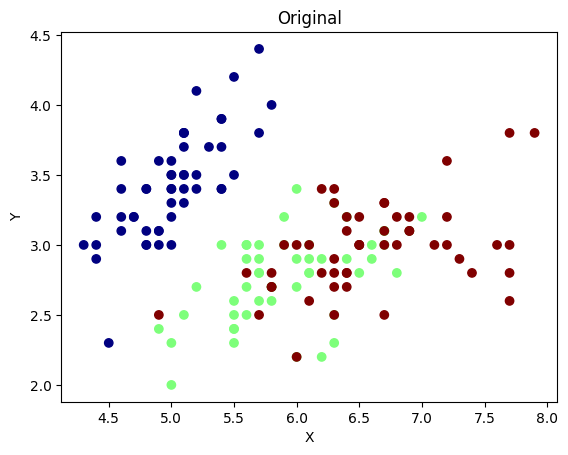
**plt.xlabel("X")**

**plt.ylabel("Y")**

**plt.title("Original")**

**plt.show()**

**Shape of Data: (150, 4)**

****

**pca = PCA(n\_components=2)**

**pca.fit(X)**

**X\_projected = pca.transform(X)**

**print("Shape of transformed Data:", X\_projected.shape)**

**pc1 = X\_projected[:, 0]**

**pc2 = X\_projected[:, 1]**

**plt.scatter(pc1, pc2, c=y, cmap="jet")**

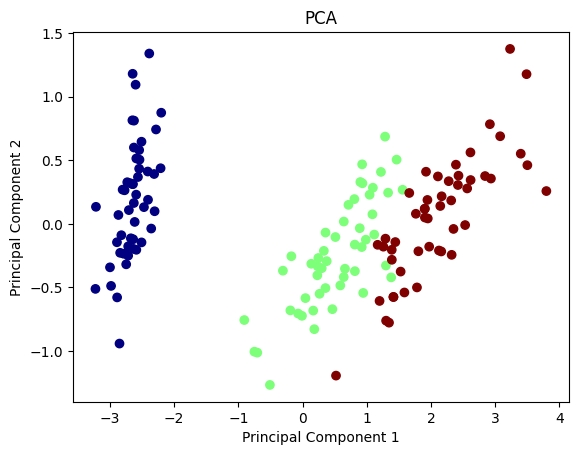
**plt.xlabel("Principal Component 1")**

**plt.ylabel("Principal Component 2")**

**plt.title("PCA")**

**Shape of transformed Data: (150, 2)**

**Text(0.5, 1.0, 'PCA')**



**lda = LDA(n\_components=2)**

**lda.fit(X,y)**

**X\_projected = lda.transform(X)**

**print("Shape of transformed Data:", X\_projected.shape)**

**ld1 = X\_projected[:, 0]**

**ld2 = X\_projected[:, 1]**

**plt.scatter(ld1, ld2, c=y, cmap="jet")**

**plt.xlabel("Linear Discriminant 1")**

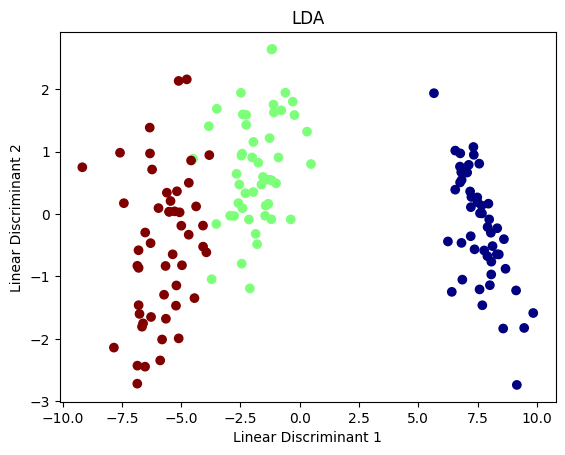
**plt.ylabel("Linear Discriminant 2")**

**plt.title("LDA")**

**Shape of transformed Data: (150, 2)**

**Out[4]:**

**Text(0.5, 1.0, 'LDA')**



**11.Write a program to develop Linear Discriminant Analysis (LDA) algorithms.**

**Code:**

**import matplotlib.pyplot as plt**

**from sklearn.decomposition import PCA**

**from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis as LDA**

**from sklearn.datasets import load\_iris**

**In [2]:**

**X = load\_iris().data**

**y = load\_iris().target**

**print("Shape of Data:", X.shape)**

**plt.scatter(X[:,0], X[:,1], c=y, cmap="jet")**

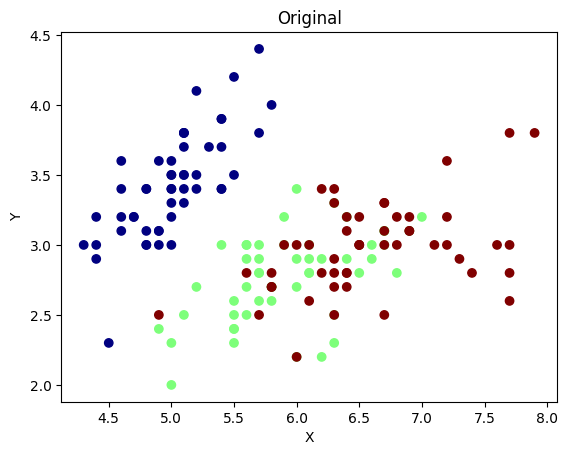
**plt.xlabel("X")**

**plt.ylabel("Y")**

**plt.title("Original")**

**plt.show()**

**Shape of Data: (150, 4)**



lda **=** LDA(n\_components**=**2)

lda**.**fit(X,y)

X\_projected **=** lda**.**transform(X)

print("Shape of transformed Data:", X\_projected**.**shape)

ld1 **=** X\_projected[:, 0]

ld2 **=** X\_projected[:, 1]

plt**.**scatter(ld1, ld2, c**=**y, cmap**=**"jet")

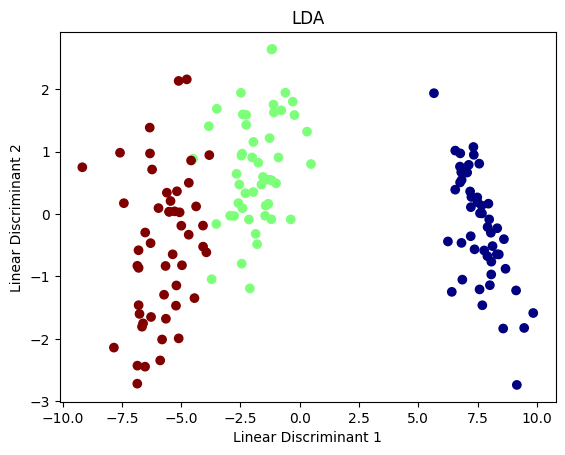
plt**.**xlabel("Linear Discriminant 1")

plt**.**ylabel("Linear Discriminant 2")

plt**.**title("LDA")

Shape of transformed Data: (150, 2)

Text(0.5, 1.0, 'LDA')



**12 Write a Program to develop simple single layer perceptron to implement AND, OR Boolean functions.**

**Code:**

**import numpy as np**

**In [2]:**

**def unit\_step(x):**

**return np.where(x > 0 , 1 ,0)**

**In [15]:**

**class SLP:**

**def \_\_init\_\_(self, lr=0.01, epochs=1000):**

**self.lr = lr**

**self.epochs = epochs**

**self.weights = None**

**self.bias = 0**

**def fit(self, X, y):**

**self.weights = np.zeros((X.shape[1], 1))**

**for \_ in range(self.epochs):**

**for ip, label in zip(X, y):**

**ip = ip.reshape(-1, 1)**

**linear\_op = np.dot(ip.T, self.weights) + self.bias**

**prediction = unit\_step(linear\_op)**

**err = label - prediction**

**self.weights += self.lr \* err \* ip**

**self.bias += self.lr \* err**

**def predict(self, X):**

**X = X.reshape(-1,1)**

**linear\_op = np.dot(X.T, self.weights) + self.bias**

**prediction = unit\_step(linear\_op)**

**return prediction**

**X\_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])**

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

**X\_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])**

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

**perceptron\_and = SLP()**

**perceptron\_and.fit(X\_and, y\_and)**

**print("AND gate")**

**print(perceptron\_and.predict(np.array([0, 0])))**

**print(perceptron\_and.predict(np.array([0, 1])))**

**print(perceptron\_and.predict(np.array([1, 0])))**

**print(perceptron\_and.predict(np.array([1, 1])))**

**output:**

**AND gate**

**[[0]]**

**[[0]]**

**[[0]]**

**[[1]]**

**perceptron\_or = SLP()**

**perceptron\_or.fit(X\_or, y\_or)**

**print("OR gate")**

**print(perceptron\_or.predict(np.array([0, 0])))**

**print(perceptron\_or.predict(np.array([0, 1])))**

**print(perceptron\_or.predict(np.array([1, 0])))**

**print(perceptron\_or.predict(np.array([1, 1])))**

**output:**

**OR gate**

**[[0]]**

**[[1]]**

**[[1]]**

**[[1]]**