1. Simple python program using conditional statements,looping,performing operations such as insert,update,delete,display,sorting and searching on data types like List,Tuples,set ,dicitionary

def list\_operations(): my\_list = []

print("\n--- List Operations ---") print("1. Insert")

print("2. Update") print("3. Delete") print("4. Display") print("5. Sort") print("6. Search") print("7. Exit")

while True:

choice = input("Enter choice: ")

if choice == '1':

value = input("Enter value to insert: ") my\_list.append(value)

elif choice == '2':

old = input("Enter value to update: ") if old in my\_list:

new = input("Enter new value: ") my\_list[my\_list.index(old)] = new

else:

print("Value not found.") elif choice == '3':

value = input("Enter value to delete: ") if value in my\_list:

my\_list.remove(value) else:

print("Value not found.") elif choice == '4':

print("List contents:", my\_list) elif choice == '5':

my\_list.sort()

print("Sorted List:", my\_list) elif choice == '6':

search = input("Enter value to search: ") print("Found!" if search in my\_list else "Not found.")

elif choice == '7': break

else:

print("Invalid choice.")

def tuple\_operations():

my\_tuple = ("apple", "banana", "cherry") print("\n--- Tuple Operations (Immutable) ---") print("Original Tuple:", my\_tuple)

while True:

print("\n1. Display") print("2. Search")

print("3. Convert to List and Add Item") print("4. Exit")

choice = input("Enter choice: ")

if choice == '1':

print("Tuple Contents:", my\_tuple) elif choice == '2':

item = input("Enter item to search: ")

print("Found!" if item in my\_tuple else "Not found.") elif choice == '3':

item = input("Enter item to add: ") temp = list(my\_tuple) temp.append(item)

my\_tuple = tuple(temp) print("Updated Tuple:", my\_tuple)

elif choice == '4': break

else:

print("Invalid choice.")

def set\_operations(): my\_set = set()

print("\n--- Set Operations ---") print("1. Insert")

print("2. Delete") print("3. Display") print("4. Search") print("5. Exit")

while True:

choice = input("Enter choice: ")

if choice == '1':

value = input("Enter value to insert: ")

my\_set.add(value) elif choice == '2':

value = input("Enter value to delete: ") my\_set.discard(value)

elif choice == '3':

print("Set contents:", my\_set) elif choice == '4':

value = input("Enter value to search: ") print("Found!" if value in my\_set else "Not found.")

elif choice == '5': break

else:

print("Invalid choice.")

def dict\_operations(): my\_dict = {}

print("\n--- Dictionary Operations ---") print("1. Insert")

print("2. Update") print("3. Delete") print("4. Display") print("5. Search") print("6. Exit")

while True:

choice = input("Enter choice: ")

if choice == '1':

key = input("Enter key: ") value = input("Enter value: ") my\_dict[key] = value

elif choice == '2':

key = input("Enter key to update: ") if key in my\_dict:

value = input("Enter new value: ") my\_dict[key] = value

else:

print("Key not found.") elif choice == '3':

key = input("Enter key to delete: ") if key in my\_dict:

del my\_dict[key] else:

print("Key not found.")

elif choice == '4':

print("Dictionary contents:", my\_dict) elif choice == '5':

key = input("Enter key to search: ") print("Found!" if key in my\_dict else "Not found.")

elif choice == '6': break

else:

print("Invalid choice.")

# Main Program Loop print("\n===== Main Menu =====") print("1. List")

print("2. Tuple")

print("3. Set") print("4. Dictionary") print("5. Exit")

while True:

main\_choice = input("Enter Choice: ")

if main\_choice == '1': list\_operations()

elif main\_choice == '2': tuple\_operations()

elif main\_choice == '3': set\_operations()

elif main\_choice == '4': dict\_operations()

elif main\_choice == '5': print("Exiting Program.") break

else:

print("Invalid choice. Try again.")

1. Visualize the n-dimensional data using :
   1. Scatter plot

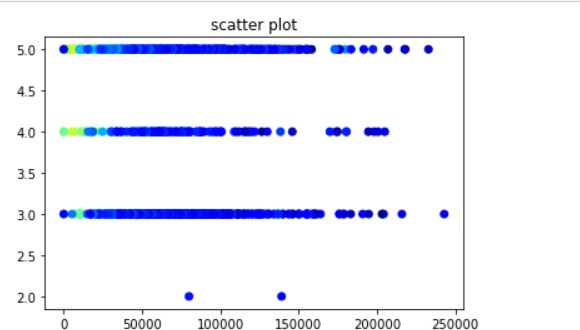
import pandas as pd

import matplotlib.pyplot as plt

data = pd.read\_csv("ToyotaCorolla.csv") x=data['KM']

y=data['Doors'] plt.scatter(x,y,c=data['Price'],cmap="jet") plt.title("scatter plot")

plt.show()



* 1. Box plot

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

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

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

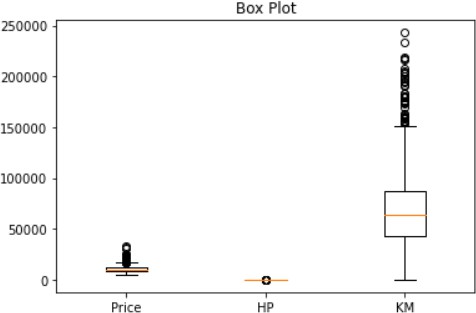
**import** seaborn **as** sns

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

*#box plot*

plt**.**title('Box Plot') plt**.**boxplot([data["Price"],data["HP"],data["KM"]])

plt**.**xticks([1,2,3],["Price","HP","KM"]) plt**.**show()



* 1. Heat map

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

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

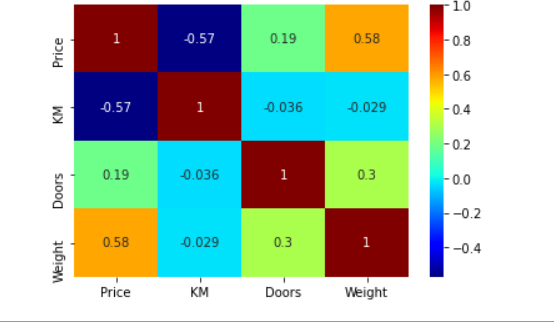
**import** matplotlib.pyplot **as** plt

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

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

*#heat map*

sns**.**heatmap(data[["Price","KM","Doors", "Weight"]]**.**corr(),cmap**=**'jet',annot**=True**) plt**.**show()



* 1. Contour plot

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

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

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

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

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

*#contour plot*

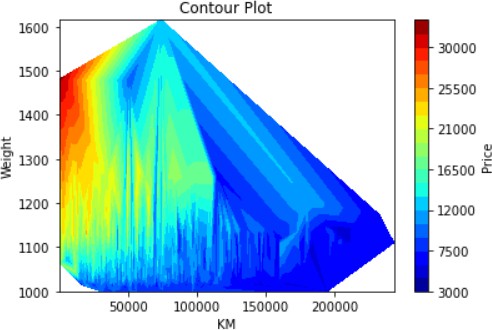
x **=** data['KM']

y **=** data['Weight']

z **=** data['Price']

plt**.**tricontourf(x, y, z, levels**=**20, cmap**=**'jet') plt**.**colorbar(label**=**'Price')

plt**.**xlabel('KM') plt**.**ylabel('Weight') plt**.**title('Contour Plot') plt**.**show()



* 1. 3D surface plot

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

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

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

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

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

*#3d surface plot*

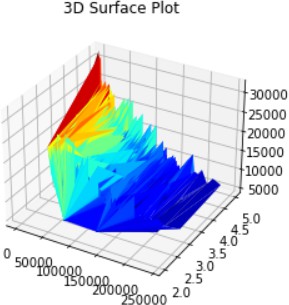
x **=** data['KM']

y **=** data['Doors']

z **=** data['Price']

ax **=** plt**.**axes(projection**=**'3d') ax**.**plot\_trisurf(x,y,z,cmap**=**"jet") ax**.**set\_title("3D Surface Plot")

plt**.**show()



1. Write a program to implement hill climbing algorithm

def hill\_climbing(graph, start, goal, heuristic): current = start

path = [current]

while current != goal: neighbors = graph[current] if not neighbors:

print(f"No more neighbors to explore from {current}. Stuck at local maxima.") return path

# Choose neighbor with lowest heuristic value (best next move) next\_node = min(neighbors, key=lambda x: heuristic[x])

# If no improvement, return current path (local maxima) if heuristic[next\_node] >= heuristic[current]:

print(f"Reached local maxima at {current}. Stopping search.") return path

current = next\_node path.append(current)

return path

graph = {

'A': ['B', 'C', 'D'],

'B': ['A', 'E'],

'C': ['A', 'E', 'D', 'F'],

'D': ['A', 'F', 'C'],

'E': ['B', 'C', 'H'],

'F': ['G', 'C', 'D'], 'G': [],

'H': ['E', 'G']

}

start = 'A' goal = 'G'

heuristic = { 'A': 40,

'B': 32,

'C': 25,

'D': 35,

'E': 19,

'F': 17,

'G': 0,

'H': 10

}

result = hill\_climbing(graph, start, goal, heuristic) if result and result[-1] == goal:

print(f"Path found from {start} to {goal}: {result}")

else:

print(f"No path found from {start} to {goal}. Reached: {result[-1]}")



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

**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(neighbour) 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}")



4.b)Write a program to implement A\* algorithm. def h(n):

H = {'A': 3, 'B': 4, 'C': 2, 'D': 6, 'G': 0, 'S': 5}

return H[n]

def a\_star\_algorithm(graph, start, goal): open\_list = [start]

closed\_list = set() g = {start:0}

parents = {start:start} while open\_list:

open\_list.sort(key=lambda v: g[v] + h(v), reverse=True) n = open\_list.pop()

# If node is goal then construct the path and return if n == goal:

reconst\_path = []

while parents[n] != n: reconst\_path.append(n) n = parents[n]

reconst\_path.append(start) reconst\_path.reverse()

print(f'Path found: {reconst\_path}')

return reconst\_path

for (m, weight) in graph[n]:

# if m is first visited, add it to open\_list and note its parent if m not in open\_list and m not in closed\_list:

open\_list.append(m) parents[m] = n

g[m] = g[n] + weight

# otherwise, check if it's quicker to first visit n, then m # and if it is, update parent and g data

# and if the node was in the closed\_list, move it to open\_list else:

if g[m] > g[n] + weight: g[m] = g[n] + weight parents[m] = n

if m in closed\_list: closed\_list.remove(m) open\_list.append(m)

# Node's neighbours are visited. Now put node to closed list. closed\_list.add(n)

print('Path does not exist!') return None

graph = {

'S': [('A', 1), ('G', 10)],

'A': [('B', 2), ('C', 1)],

'B': [('D', 5)],

'C': [('D', 3),('G', 4)], 'D': [('G', 2)]

}

a\_star\_algorithm(graph, 'S', 'G')



1. Write a program to implement Min-Max algorithm.

def minmax(depth, nodeIndex, maximizingPlayer, values, alpha, beta): if depth == 3:

return values[nodeIndex]

if maximizingPlayer: best = float('-inf') for i in range(2):

val = minmax(depth + 1, nodeIndex \* 2 + i, False, values, alpha, beta) best = max(best, val)

return best else:

best = float('inf') for i in range(2):

val = minmax(depth + 1, nodeIndex \* 2 + i, True, values, alpha, beta) best = min(best, val)

return best

# Example tree with depth 3 and 8 terminal nodes values = [3, 5, 2, 9, 12, 5, 23, 23]

# Start the Min-Max algorithm

result = minmax(0, 0, True, values, float('-inf'), float('inf')) print("The optimal value is:", result)



5 Write a program to implement Alpha-beta pruning algorithm.

def alphabeta(depth, nodeIndex, maximizingPlayer, values, alpha, beta): if depth == 3:

return values[nodeIndex]

if maximizingPlayer: best = float('-inf') for i in range(2):

val = alphabeta(depth + 1, nodeIndex \* 2 + i, False, values, alpha, beta) best = max(best, val)

alpha = max(alpha, best) if beta <= alpha:

break return best

else:

best = float('inf') for i in range(2):

val = alphabeta(depth + 1, nodeIndex \* 2 + i, True, values, alpha, beta) best = min(best, val)

beta = min(beta, best) if beta <= alpha:

break return best

# Example tree with depth 3 and 8 terminal nodes values = [3, 5, 2, 9, 12, 5, 23, 23]

# Start the Alpha-Beta Pruning algorithm

result = alphabeta(0, 0, True, values, float('-inf'), float('inf')) print("The optimal value is:", result)



1. Write a program to develop the naive bayes classifier based on split up of training and testing dataset as 90-10,70-30.
   1. Iris dataset

import numpy as np import pandas as pd

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

iris = pd.read\_csv(r"Iris.csv") iris.head()

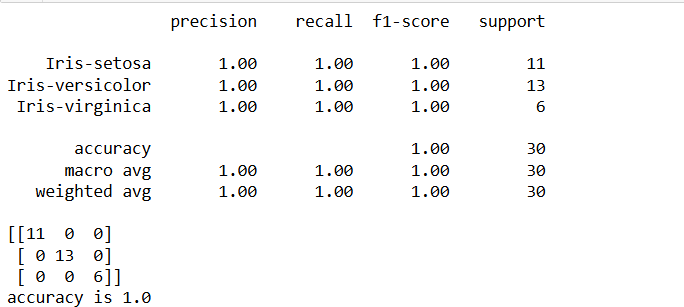
X = iris.iloc[:, :-1].values y = iris.iloc[:, -1].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0) classifier = GaussianNB()

classifier.fit(X\_train, y\_train)

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

print(classification\_report(y\_test, y\_pred)) print(confusion\_matrix(y\_test, y\_pred)) print('accuracy is', accuracy\_score(y\_test, y\_pred))



6.b) Titanic dataset

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

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

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

**from** sklearn.model\_selection **import** train\_test\_split **from** sklearn.naive\_bayes **import** GaussianNB **from** sklearn.impute **import** SimpleImputer

**from** sklearn.preprocessing **import** LabelEncoder

*# Load the dataset*

df **=** pd**.**read\_csv(r"Titanic-Dataset.csv")

df **=** df[['Survived', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']]

*# Handle missing values*

imputer **=** SimpleImputer(strategy**=**'median')

df[['Age', 'Fare']] **=** imputer**.**fit\_transform(df[['Age', 'Fare']])

df['Embarked']**.**fillna(df['Embarked']**.**mode()[0], inplace**=True**) df['Embarked'] **=** LabelEncoder()**.**fit\_transform(df['Embarked'])

*# Split the data into train and test sets*

X **=** df**.**drop('Survived', axis**=**1) y **=** df['Survived']

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

*# Initialize and fit the Gaussian Naive Bayes classifier*

classifier **=** GaussianNB() classifier**.**fit(X\_train, y\_train)

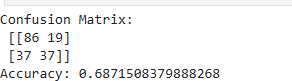
*# Make predictions on the test set*

y\_pred **=** classifier**.**predict(X\_test)

*# Evaluate the model*

cm **=** confusion\_matrix(y\_test, y\_pred) print("Confusion Matrix:\n", cm)

accuracy **=** accuracy\_score(y\_test, y\_pred) print("Accuracy:", accuracy)



1. 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,
   1. Glass dataset

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

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

**from** sklearn.model\_selection **import** train\_test\_split **from** sklearn.neighbors **import** KNeighborsClassifier **from** sklearn.metrics **import** accuracy\_score

df **=** pd**.**read\_csv('glass.csv') y **=** df['Type']**.**values

X **=** df**.**drop('Type', axis**=**1)**.**values

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X, y, test\_size**=**0.3)

*# Using scikit-learn with Euclidean distance*

clf\_euclidean **=** KNeighborsClassifier(n\_neighbors**=**3, metric**=**'euclidean') clf\_euclidean**.**fit(X\_train, y\_train)

predictions\_euclidean **=** clf\_euclidean**.**predict(X\_test) accuracy\_euclidean **=** accuracy\_score(y\_test, predictions\_euclidean) print("Accuracy with Euclidean distance:", accuracy\_euclidean)

*# Using scikit-learn with Manhattan distance*

clf\_manhattan **=** KNeighborsClassifier(n\_neighbors**=**3, metric**=**'manhattan') clf\_manhattan**.**fit(X\_train, y\_train)

predictions\_manhattan **=** clf\_manhattan**.**predict(X\_test) accuracy\_manhattan **=** accuracy\_score(y\_test, predictions\_manhattan) print("Accuracy with Manhattan distance:", accuracy\_manhattan)



7.b)Fruit dataset

import numpy as np import pandas as pd

from sklearn.model\_selection import train\_test\_split from sklearn.neighbors import KNeighborsClassifier from sklearn.metrics import accuracy\_score

# Load dataset

df = pd.read\_csv('fruits.csv') y = df['fruit\_label'].values

X = df[['mass', 'width', 'height', 'color\_score']].values

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=45)

# Using scikit-learn with Euclidean distance

clf\_euclidean = KNeighborsClassifier(n\_neighbors=5, metric='euclidean') # Fixed here clf\_euclidean.fit(X\_train, y\_train)

predictions\_euclidean = clf\_euclidean.predict(X\_test)

accuracy\_euclidean = accuracy\_score(y\_test, predictions\_euclidean) print("Accuracy with Euclidean distance (using sklearn):", accuracy\_euclidean)

# Using scikit-learn with Manhattan distance

clf\_manhattan = KNeighborsClassifier(n\_neighbors=5, metric='manhattan') # Fixed here clf\_manhattan.fit(X\_train, y\_train)

predictions\_manhattan = clf\_manhattan.predict(X\_test)

accuracy\_manhattan = accuracy\_score(y\_test, predictions\_manhattan) print("Accuracy with Manhattan distance (using sklearn):", accuracy\_manhattan)



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

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

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

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

**from** sklearn.cluster **import** KMeans

*# Load the Iris dataset*

iris **=** load\_iris()

X **=** iris**.**data

*# Number of clusters*

K **=** 3

*# K-means using scikit-learn*

kmeans **=** KMeans(n\_clusters**=**K, random\_state**=**0) labels **=** kmeans**.**fit\_predict(X)

centroids **=** kmeans**.**cluster\_centers\_

*# Print results*

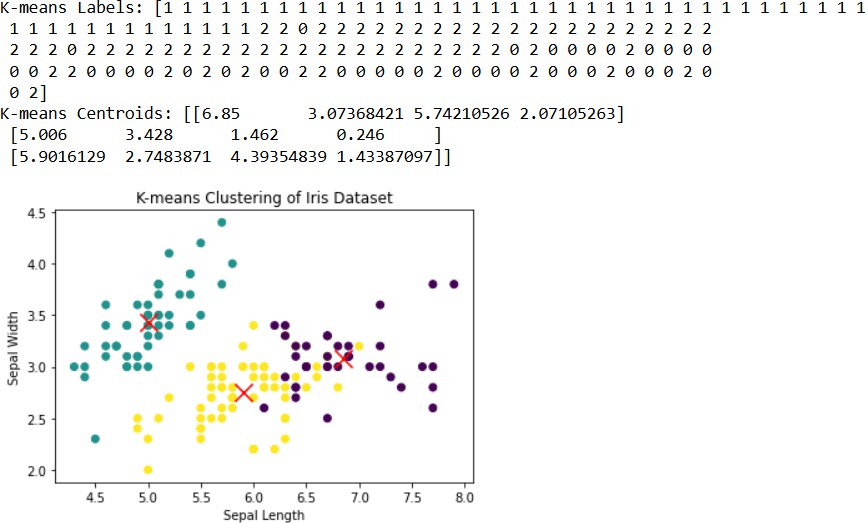
print("K-means Labels:", labels) print("K-means Centroids:", centroids)

*# Plotting K-means results*

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



1. Write a program to perform agglomerative clustering based on single-linkage

,complete-linkage criteria

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

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

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

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

iris **=** load\_iris() data **=** iris**.**data[:6]

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

n **=** data**.**shape[0] proximity\_matrix **=** np**.**zeros((n, n)) **for** i **in** range(n):

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

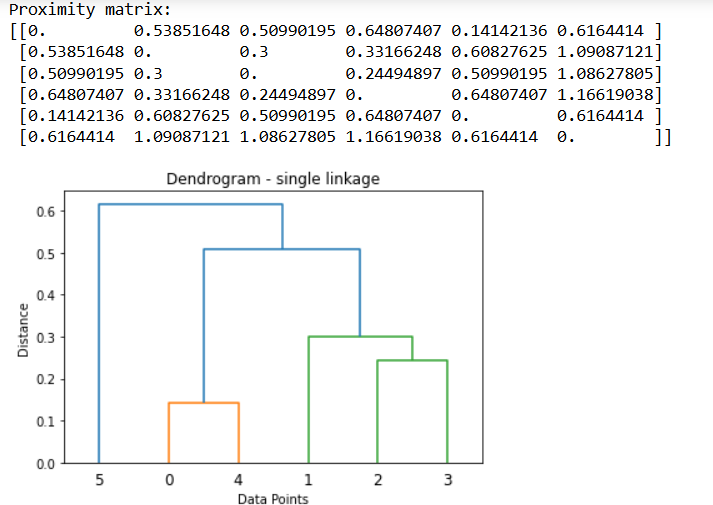
proximity\_matrix[i, j] **=** np**.**linalg**.**norm(data[i] **-** data[j]) proximity\_matrix[j, i] **=** proximity\_matrix[i, j]

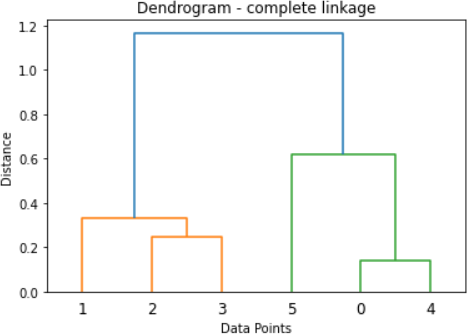
**return** proximity\_matrix

**def** plot\_dendrogram(data, method): linkage\_matrix **=** linkage(data, method**=**method) dendrogram(linkage\_matrix) plt**.**title(f'Dendrogram - {method} linkage') plt**.**xlabel('Data Points')

plt**.**ylabel('Distance') plt**.**show()

print("Proximity matrix:") print(proximity\_matrix(data)) plot\_dendrogram(data, 'single') plot\_dendrogram(data, 'complete')





1. Write a program to develop the principal component Analysis(PCA) algorithm. import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.decomposition import PCA as SklearnPCA x=load\_iris().data

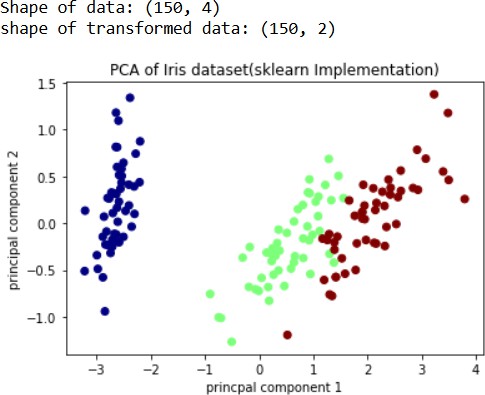
y=load\_iris().target PCA=SklearnPCA(n\_components=2) X\_projected=pca.fit\_transform(x) print("Shape of data:",x.shape)

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("princpal component 1")

plt.ylabel("principal component 2")

plt.title("PCA of Iris dataset(sklearn Implementation)") plt.show()



1. Write a program to develop the Linear Discriminant Analysis(LDA) algorithm. import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis X=load\_iris().data

y=load\_iris().target lda=LinearDiscriminantAnalysis(n\_components=2) X\_projected=lda.fit\_transform(x,y)

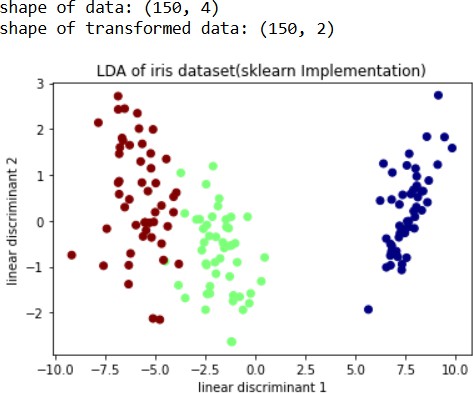
print("shape of data:",X.shape)

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 of iris dataset(sklearn Implementation)") plt.show()



1. Write a program to develop simple single layer perceptron to implement AND,OR Boolean functions.

import numpy as np def step\_function(X):

return np.where(X>=0,1,0) 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]]) class perceptron:

def init (self,input\_size,learning\_rate=0.1,epochs=1000):

self.weights=np.zeros((input\_size,1)) self.bias=0 self.learning\_rate=learning\_rate self.epochs=epochs

def train(self,X,Y):

for \_ in range(self.epochs): for inputs,label in zip(X,Y):

inputs=inputs.reshape(-1,1) linear\_output=np.dot(inputs.T,self.weights)+self.bias prediction=step\_function(linear\_output)

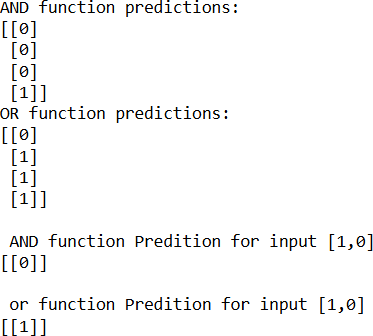
error=label-prediction self.weights+=self.learning\_rate\*error\*inputs self.bias+=self.learning\_rate\*error

def predict(self,X): linear\_output=np.dot(X,self.weights)+self.bias return step\_function(linear\_output)

perceptron\_and=perceptron(input\_size=2) perceptron\_and.train(X\_and,Y\_and) perceptron\_or=perceptron(input\_size=2) perceptron\_or.train(X\_or,Y\_or)

print("AND function predictions:") print(perceptron\_and.predict(X\_and)) print("OR function predictions:") print(perceptron\_or.predict(X\_or)) and\_test\_input=np.array([[1,0]])

print("\n AND function Predition for input [1,0]") print(perceptron\_and.predict(and\_test\_input)) or\_test\_input=np.array([[1,0]])

print("\n or function Predition for input [1,0]") print(perceptron\_or.predict(or\_test\_input))