**1) scatter plot and hill climbing**  
  
x1 = np.random.rand(25)

y1 = np.random.rand(25)

x2 = np.random.rand(25)

y2 = np.random.rand(25)

plt.scatter(x1, y1, color='blue', label='Group 1')

plt.scatter(x2, y2, color='red', label='Group 2')

plt.title("Scatter plot with two colors")

plt.xlabel("X\_axis")

plt.ylabel("Y\_axis")

plt.legend()

plt.show()

**Hill climbing:**  
  
def hill\_climb(start, step\_size, max\_iterations):

def fitness(x):

return -x\*\*2 + 4 # The function we want to maximize

current = start

for \_ in range(max\_iterations):

neighbor = current + step\_size

if fitness(neighbor) > fitness(current):

current = neighbor

else:

break # No better neighbor found

return current, fitness(current)

solution, value = hill\_climb(start=2, step\_size=-0.1, max\_iterations=100)

print("Best solution:", solution)

print("Best value:", value)

**2) 3d plot and BFS**

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D # Needed for 3D plotting

# Creating meshgrid for X and Y

x = np.linspace(-5, 5, 50)

y = np.linspace(-5, 5, 50)

X, Y = np.meshgrid(x, y)

# Define Z as a function of X and Y

Z = np.sin(np.sqrt(X\*\*2 + Y\*\*2)) # Example surface

# Plotting

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

ax.plot\_surface(X, Y, Z, cmap='viridis') # You can change color map

ax.set\_title("3D Surface Plot")

ax.set\_xlabel("X axis")

ax.set\_ylabel("Y axis")

ax.set\_zlabel("Z axis")

plt.show()

**BFS**

from collections import deque

# BFS function

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

print(node, end=' ')

visited.add(node)

queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

# Example graph (adjacency list)

graph = {

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

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

# Call BFS

bfs(graph, 'A')

**3) contour plot and A\***  
  
contour plot :  
import numpy as np

import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 100)

y = np.linspace(-5, 5, 100)

X, Y = np.meshgrid(x, y)

Z = np.sin(np.sqrt(X\*\*2 + Y\*\*2)) # function of X, Y

plt.contourf(X, Y, Z, cmap='plasma') # filled contour

plt.colorbar()

plt.title("Contour Plot")

plt.xlabel("X axis")

plt.ylabel("Y axis")

plt.show()

**A\***

import heapq

def heuristic(a, b):

# Manhattan distance

return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(grid, start, goal):

rows, cols = len(grid), len(grid[0])

open\_list = []

heapq.heappush(open\_list, (0 + heuristic(start, goal), 0, start, [start]))

visited = set()

while open\_list:

f, cost, current, path = heapq.heappop(open\_list)

if current == goal:

return path

if current in visited:

continue

visited.add(current)

for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]: # 4 directions

nx, ny = current[0] + dx, current[1] + dy

if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 0:

next\_node = (nx, ny)

heapq.heappush(open\_list, (cost + 1 + heuristic(next\_node, goal), cost + 1, next\_node, path + [next\_node]))

return None # No path found

# Example grid (0 = open, 1 = wall)

grid = [

[0, 0, 0, 0],

[1, 1, 0, 1],

[0, 0, 0, 0],

[0, 1, 1, 0],

[0, 0, 0, 0]

]

start = (0, 0)

goal = (4, 3)

path = astar(grid, start, goal)

print("Path:", path)

**4) Heat map and Min-max algorithm**   
  
import seaborn as sns

import numpy as np

import matplotlib.pyplot as plt

data = np.random.rand(10, 5) # 2D data: 10 rows × 5 columns

sns.heatmap(data, cmap='viridis')

plt.title("Heat Map")

plt.show()

**min –max**

# Minimax Algorithm Implementation for a Game Tree

def minimax(depth, node\_index, is\_maximizing\_player, scores, height):

# Base case: leaf node

if depth == height:

return scores[node\_index]

if is\_maximizing\_player:

return max(

minimax(depth + 1, node\_index \* 2, False, scores, height),

minimax(depth + 1, node\_index \* 2 + 1, False, scores, height)

)

else:

return min(

minimax(depth + 1, node\_index \* 2, True, scores, height),

minimax(depth + 1, node\_index \* 2 + 1, True, scores, height)

)

# Example leaf node scores

scores = [3, 5, 6, 9, 1, 2, 0, -1]

# Height of the tree = log2(len(scores)) = 3

tree\_height = 3

# Call minimax starting from root (depth=0), root index=0, maximizing player's turn

best\_value = minimax(0, 0, True, scores, tree\_height)

print("The optimal value is:", best\_value)

**5) Box plot and Alpha beta pruning**  
  
import seaborn as sns

import numpy as np

import matplotlib.pyplot as plt

data = np.random.rand(100, 101) # 100 samples, 101 features

sns.boxplot(data=data)

plt.title("Box Plot")

plt.xlabel("Features")

plt.ylabel("Values")

plt.show()

**Alpha beta**

def alphabeta(depth, node\_index, is\_max, scores, alpha, beta, height):

if depth == height:

return scores[node\_index]

if is\_max:

best = float('-inf')

# Explore left and right child

for i in range(2):

val = alphabeta(depth + 1, node\_index \* 2 + i, False, scores, alpha, beta, height)

best = max(best, val)

alpha = max(alpha, best)

if beta <= alpha:

break # β cut-off

return best

else:

best = float('inf')

# Explore left and right child

for i in range(2):

val = alphabeta(depth + 1, node\_index \* 2 + i, True, scores, alpha, beta, height)

best = min(best, val)

beta = min(beta, best)

if beta <= alpha:

break # α cut-off

return best

# Example leaf node values

scores = [3, 5, 6, 9, 1, 2, 0, -1]

# Tree height = 3 (log2 of 8 leaf nodes)

tree\_height = 3

# Start at root: depth=0, node\_index=0, maximizing=True

optimal = alphabeta(0, 0, True, scores, float('-inf'), float('inf'), tree\_height)

print("The optimal value is:", optimal)

**6) Navy base classifier**

import pandas as pd

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

from sklearn.preprocessing import LabelEncoder

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

# Load Titanic dataset from CSV

# Replace with your actual path if needed

data = pd.read\_csv('titanic.csv')

# Select features and target (you can customize as needed)

features = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare']

target = 'Survived'

# Handle missing values (simple strategy)

data = data[features + [target]].dropna()

# Encode categorical columns

le = LabelEncoder()

data['Sex'] = le.fit\_transform(data['Sex']) # male=1, female=0

# Split into input (X) and output (y)

X = data[features]

y = data[target]

# Split into training and testing (70-30)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=0)

# Train Naive Bayes classifier

model = GaussianNB()

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

# Predict and evaluate

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

print("Accuracy of Naive Bayes classifier:", accuracy\_score(y\_test, y\_pred))

7) KNN

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

# Load the glass dataset from local CSV file

# Replace 'glass.csv' with the correct filename or path if needed

data = pd.read\_csv('glass.csv')

# Separate features (X) and target (y)

X = data.iloc[:, :-1] # All columns except the last (features)

y = data.iloc[:, -1] # Last column (target class)

# Split dataset into 70% training and 30% testing

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=0)

# Standardize the feature values

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# KNN with Euclidean distance (default)

knn\_euclidean = KNeighborsClassifier(n\_neighbors=3, metric='euclidean')

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

y\_pred\_euclidean = knn\_euclidean.predict(X\_test)

print("Accuracy with Euclidean distance:", accuracy\_score(y\_test, y\_pred\_euclidean))

# KNN with Manhattan distance

knn\_manhattan = KNeighborsClassifier(n\_neighbors=3, metric='manhattan')

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

y\_pred\_manhattan = knn\_manhattan.predict(X\_test)

print("Accuracy with Manhattan distance:", accuracy\_score(y\_test, y\_pred\_manhattan))

8) K means

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler

# Load and scale data

iris = load\_iris()

X = StandardScaler().fit\_transform(iris.data)

# Apply K-Means (3 clusters for 3 Iris species)

kmeans = KMeans(n\_clusters=3, random\_state=0)

labels = kmeans.fit\_predict(X)

# Scatter plot using first two features

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

plt.title("K-Means Clustering (Iris Dataset)")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

9) single and complete linkage

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

from scipy.cluster.hierarchy import dendrogram, linkage

# Load and standardize the Iris dataset

iris = load\_iris()

X = StandardScaler().fit\_transform(iris.data)

# Function to plot dendrogram

def plot\_dendrogram(X, method):

Z = linkage(X, method=method)

plt.figure(figsize=(6, 4))

plt.title(f'Dendrogram - {method} linkage')

dendrogram(Z)

plt.xlabel("Samples")

plt.ylabel("Distance")

plt.tight\_layout()

plt.show()

# Plot dendrograms for both linkage methods

for method in ['single', 'complete']:

plot\_dendrogram(X, method)

11)PCA and LDA

from sklearn.datasets import load\_iris

from sklearn.decomposition import PCA

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis as LDA

import matplotlib.pyplot as plt

# Load the Iris dataset

iris = load\_iris()

X = iris.data # Features (4 columns)

y = iris.target # Labels (0 = setosa, 1 = versicolor, 2 = virginica)

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

# -----------------------------

# PCA (Principal Component Analysis)

# -----------------------------

print("\n--- PCA ---")

pca = PCA(n\_components=2) # Reduce to 2 dimensions

X\_pca = pca.fit\_transform(X)

print("Shape after PCA:", X\_pca.shape)

print("Explained Variance Ratio:", pca.explained\_variance\_ratio\_)

# Plot PCA result

plt.figure(figsize=(6, 4))

plt.title("PCA - Iris Dataset")

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=y, cmap='viridis')

plt.xlabel("PCA 1")

plt.ylabel("PCA 2")

plt.show()

# -----------------------------

# LDA (Linear Discriminant Analysis)

# -----------------------------

print("\n--- LDA ---")

lda = LDA(n\_components=2)

X\_lda = lda.fit\_transform(X, y)

print("Shape after LDA:", X\_lda.shape)

# Plot LDA result

plt.figure(figsize=(6, 4))

plt.title("LDA - Iris Dataset")

plt.scatter(X\_lda[:, 0], X\_lda[:, 1], c=y, cmap='plasma')

plt.xlabel("LDA 1")

plt.ylabel("LDA 2")

plt.show()

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

# Step function

def step(x):

return 1 if x >= 0 else 0

# Training function

def train(X, y):

w1, w2 = 0, 0 # weights

b = 0 # bias

lr = 1 # learning rate

for \_ in range(10): # 10 iterations

for i in range(4): # for all 4 input examples

x1, x2 = X[i]

z = w1 \* x1 + w2 \* x2 + b

y\_pred = step(z)

error = y[i] - y\_pred

# update weights and bias

w1 += lr \* error \* x1

w2 += lr \* error \* x2

b += lr \* error

return w1, w2, b

# Prediction

def predict(X, w1, w2, b):

results = []

for x1, x2 in X:

z = w1 \* x1 + w2 \* x2 + b

results.append(step(z))

return results

# Input and Output for AND

X = [[0,0], [0,1], [1,0], [1,1]]

y = [0, 0, 0, 1]

# Train and predict

w1, w2, b = train(X, y)

print("AND Gate Output:", predict(X, w1, w2, b))

output:

AND Predictions: [0, 0, 0, 1]

OR Predictions: [0, 1, 1, 1]