1.Scatter plot+hill climbing

```python
import matplotlib.pyplot as plt
import numpy as np

# Scatter Plot
x_points = np.random.rand(10)
y_points = np.random.rand(10)
plt.scatter(x_points, y_points)
plt.title("Scatter Plot")
plt.show()

# Hill Climbing: Maximize f(x) = -x^2 + 4
def f(x):
    return -x**2 + 4

# Start point
x = float(input("Enter start x: "))

step_size = 0.1
for _ in range(50):
    next_x1 = x + step_size
    next_x2 = x - step_size

    if f(next_x1) > f(x):
        x = next_x1
    elif f(next_x2) > f(x):
        x = next_x2
    else:
        # No better neighbor → stop
        break

print("Best x:", x, "Value:", f(x))
```

2. 3d plot and bfs

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from queue import PriorityQueue

# 3D Surface Plot
x = y = np.linspace(-3, 3, 30)
x, y = np.meshgrid(x, y)
z = x**2 + y**2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, cmap='viridis')
plt.show()

# Best First Search
graph = {
    'S': ['A', 'B'],
    'A': ['G'],
    'B': ['G'],
    'G': []
}
h = {'S': 3, 'A': 2, 'B': 1, 'G': 0}

def best_first(start, goal):
    pq = PriorityQueue()
    pq.put((h[start], start))
    visited = set()

    while not pq.empty():
        _, node = pq.get()
        if node in visited:
            continue
        print("Visited:", node)
        visited.add(node)

        if node == goal:
            print("Goal Reached:", node)
            break

        for n in graph[node]:
            pq.put((h[n], n))

# Run Best First Search
best_first('S', 'G')
```

4. Contour plot with a*

```
import numpy as np
import matplotlib.pyplot as plt
from queue import PriorityQueue

# Contour Plot
x = y = np.linspace(-2, 2, 30)
x, y = np.meshgrid(x, y)
z = x**2 + y**2
plt.contour(x, y, z)
plt.title("Contour Plot")
plt.show()

# A* Search
graph = {
    'S': [('A', 1), ('B', 2)],
    'A': [('G', 3)],
    'B': [('G', 1)],
    'G': []
}
h = {'S': 4, 'A': 2, 'B': 1, 'G': 0}

def astar(start, goal):
    pq = PriorityQueue()
    pq.put((0, start))
    cost = {start: 0}
    parent = {start: None}

    while not pq.empty():
        _, node = pq.get()
        print("Visited:", node)

        if node == goal:
            print("Goal Reached:", node)
            break

        for n, c in graph[node]:
            new_cost = cost[node] + c
            if n not in cost or new_cost < cost[n]:
                cost[n] = new_cost
                pq.put((new_cost + h[n], n))
                parent[n] = node

    # reconstruct path
    if goal in parent:
        path = []        node = goal
```

```
        while node is not None:
            path.append(node)
            node = parent[node]
        path.reverse()
        print("Path:", " -> ".join(path))
        print("Total Cost:", cost[goal])

# Run A* Search
astar('S', 'G')
```

4. Heatmap and min max

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Heatmap
data = np.random.rand(4, 4)
sns.heatmap(data, annot=True)
plt.title("Heatmap")
plt.show()

# Minimax (2-level tree demo)
def minmax(values, is_max):
    a = values[0:2]  # first branch
    b = values[2:4]  # second branch
    return max(min(a), min(b)) if is_max else min(max(a), max(b))

# Example run
vals = list(map(int, input("Enter 4 values (space-separated): ").split()))
print("Result:", minmax(vals, True))
```

```
5. box plot a* alg

import matplotlib.pyplot as plt

# Box Plot
data = list(map(int, input("Enter values for box plot: ").split()))
plt.boxplot(data)
plt.title("Box Plot")
plt.show()

# Simplified Alpha-Beta (2-level tree)
def alphabeta(vals):
```

```
    a = min(vals[0], vals[1])
    b = min(vals[2], vals[3])
    return max(a, b)

v = list(map(int, input("Enter 4 leaf values: ").split()))
print("Alpha-Beta Result:", alphabeta(v))
```

7. KNN

```
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("glass.csv")
X, y = df.drop("Type", axis=1), df["Type"]

# Train-test split (70-30)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Run KNN with different distance metrics
for metric in ["euclidean", "manhattan"]:
    clf = KNeighborsClassifier(n_neighbors=3, metric=metric)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"Accuracy with {metric.capitalize()} distance: {acc:.4f}")
```

6. Naïve bayes on titanic dataset

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, accuracy_score

# Load dataset
df = pd.read_csv("Titanic-
Dataset.csv")[['Survived','Pclass','Age','SibSp','Parch','Fare','Embarked']]
```

```
# Handle missing values
df[['Age','Fare']] =
SimpleImputer(strategy='median').fit_transform(df[['Age','Fare']])
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].mode()[0])
df['Embarked'] = LabelEncoder().fit_transform(df['Embarked'])

# Features and target
X, y = df.drop('Survived', axis=1), df['Survived']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train Naive Bayes model
model = GaussianNB().fit(X_train, y_train)

# Predictions & evaluation
y_pred = model.predict(X_test)
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Accuracy:", round(accuracy_score(y_test, y_pred), 4))
```

8. unsupervised k means clustering

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

# Load Iris dataset
X, y = load_iris(return_X_y=True)

# K-means with 3 clusters
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
labels = kmeans.fit_predict(X)
# Print results
print("Cluster Centers:\n", kmeans.cluster_centers_)

# Plot (using first two features for visualization)
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', alpha=0.7)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='x', color='red', s=200, label="Centroids")
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('K-means Clustering (Iris Dataset)')
plt.legend()
plt.show()
```

9. agglomerative clustering

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.datasets import load_iris

# Load first 6 samples of Iris dataset
iris = load_iris()
data = iris.data[:6]

# Function to compute proximity matrix
def proximity_matrix(X):
    n = X.shape[0]
    mat = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1, n):
            dist = np.linalg.norm(X[i] - X[j])  # Euclidean distance
            mat[i, j] = mat[j, i] = dist
    return mat

# Function to plot dendrogram
def plot_dendrogram(X, method):
    Z = linkage(X, method=method)
    dendrogram(Z, labels=[f"Point {i}" for i in range(len(X))])
    plt.title(f"Hierarchical Clustering ({method.capitalize()} Linkage)")
    plt.xlabel("Data Points")
    plt.ylabel("Distance")
    plt.show()

# Show proximity matrix
print("Proximity matrix:\n", proximity_matrix(data))

# Plot dendrograms
plot_dendrogram(data, "single")
plot_dendrogram(data, "complete")
```

10.PCA and LDA

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Load Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# ----- PCA -----
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

print("PCA: Original shape:", X.shape)
print("PCA: Transformed shape:", X_pca.shape)

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap="jet", alpha=0.7)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA on Iris Dataset")
plt.show()

# ----- LDA -----
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit_transform(X, y)

print("LDA: Original shape:", X.shape)
print("LDA: Transformed shape:", X_lda.shape)

plt.scatter(X_lda[:, 0], X_lda[:, 1], c=y, cmap="jet", alpha=0.7)
plt.xlabel("Linear Discriminant 1")
plt.ylabel("Linear Discriminant 2")
plt.title("LDA on Iris Dataset")
plt.show()
```