UNIVERSITY OF CALCUTTA

Artificial Intelligence Assignment

RAHUL BISWAS
MSc Computer Science Semester 3

ASSIGNMENT NO.	DESCRIPTION
1	K-means Clustering on the Iris Dataset
2	K-medoid Clustering on Iris dataset.
3	Apply AGNES (single linkage, complete linkage, average linkage) on Iris dataset for clustering.
4	Apply DIANA (single linkage, complete linkage, average linkage) on Iris dataset for clustering.
5	Draw decision tree using ID3 algorithm on golf playing dataset.
6	Apply CART on buy computer dataset.
7	Apply naïve Bayesian algorithm on buy computer dataset to identify class label of unknown samples.
8	Apply back propagation algorithm on sample {1,0,1} with the class label {1,0}. (Where network topology: 3-2-2-2, all biases and weights are initialized at 0)
9	Apply fuzzy c means algorithm on Boston Housing Dataset.
10	Apply perceptron for realization of logic gates. (bias= 1)
11	Apply Madeline algorithm for Bipolar XOR gates. (Weights are randomly initialized, v0, v1, v2 = 0.5, bias values are set to 1, learning rate = 0.5, Network topology: 2-2-1)
12	Apply Madeline algorithm for variable network topology.

Problem Statement:

K-means Clustering on the Iris Dataset.

Data Description:

It includes three iris species with 50 samples each as well as some properties about each flower. One flower species is linearly separable from the other two, but the other two are not linearly separable from each other.
The columns in this dataset are:
SepalLengthCm
SepalWidthCm
PetalLengthCm
PetalWidthCm

Algorithm:

Algorithm: K-Means

Input: Unlabeled Dataset, Number of clusters/group(K)

Output: K number of groups of given dataset

Steps:

1. Start.
2. Select randomly K numbers points/centroid from the given dataset.
3. For each point, calculate the distance from the centroids and assign then to the closest centroid cluster.
4. Calculate the variance and place a new centroid of each cluster.
5. Repeat step 2 for the new centroids.
6. If any changes occur on any cluster, then do step 3. else Stop.
7. End.

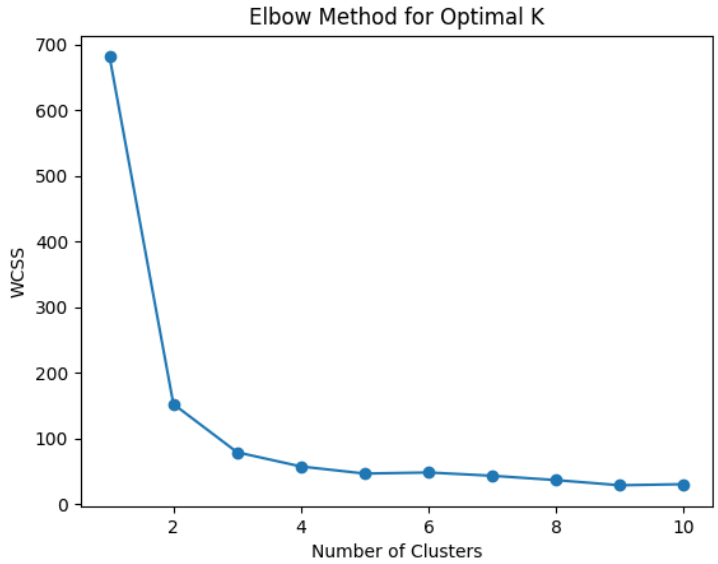
Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def print_group(clusters):
        for cluster_id, cluster_points in clusters.items():
                print(f"\nCluster {cluster_id} ({len(cluster_points)}
points):")
                for index, point in cluster_points:
                        print(f"  Index: {index}, Data Point:
{point}")

def calculate_wcss(dataset, centroids, clusters):
        wcss = 0
        for cluster_id, cluster_points in clusters.items():
                centroid = centroids[cluster_id]
                for index, point in cluster_points:
                # Use the data point directly for WCSS calculation
                        wcss += np.sum((point - centroid) ** 2)
        return wcss

def  kmeans(dataset,no_of_centroid,test = 0):
    # Ensure dataset is a NumPy array for efficient computation
        if isinstance(dataset, pd.DataFrame):
                dataset = dataset.to_numpy()

    # Choosing initial centroids
        random_indices =
np.random.randint(0,len(dataset),no_of_centroid)
        centroids = dataset[random_indices,:]
        if test ==0:
# If this is not test case then print
                print(f"Initial centroids are : {centroids}")

    # K means algorithm
        clusters = {i: [] for i in range(no_of_centroid)}
# Creating clusters / groups
        tolerance = 1e-2                                # Convergence
tolerance (distance change between centroids)
        shifted = True
# Variable to track either centroids are shifted
        iter = 0
        while shifted:
                iter = iter + 1
                shifted = False
                distances = np.sqrt(np.sum((dataset[:, np.newaxis, :]
- centroids[np.newaxis, :, :])**2, axis=2))
                if test ==0:
# If this is not test case then print
                        print(f"distance from the centroids:")
                        print(distances)
                labels = np.argmin(distances, axis=1)
# Assign points to the nearest centroid
                clusters = {i: [] for i in range(no_of_centroid)}
# Creating clusters / groups
```

```
                for i, label in enumerate(labels):
                        clusters[label].append((i, dataset[i]))  #
Append (index, data point) to the cluster
                if test == 0:
                        print_group(clusters)

                new_centroids = np.array([np.mean([point[1] for point
in clusters[i]], axis=0) if len(clusters[i]) > 0 else centroids[i]
for i in range(no_of_centroid)])
                if test ==0:
                        print(f"New centroids are : {new_centroids}")
                centroid_shift = np.linalg.norm(new_centroids -
centroids)  # Calculate the Euclidean distance between centroids

                # If the centroid shift is smaller than the
tolerance, stop
                if centroid_shift >= tolerance:
                        centroids = new_centroids
                        shifted = True
                else:
                        if test ==0:
                                print(f"Convergence reached after
{iter} iterations.")

        wcss = calculate_wcss(dataset, centroids, clusters)
        return centroids, clusters, wcss

if __name__ == '__main__':
        np.random.seed(42)

        dataset_name = 'Iris.csv'
    # Read Input data
        csv_file  =  pd.read_csv(f'Dataset/{dataset_name}')
# Read Dataset
        dataset = csv_file.iloc[:,1:-1]
# Exclusion of "ID" and "Species" column

        # csv_file = pd.read_csv('Dataset/kmeans_test.csv')
        # dataset = csv_file.iloc[:, 1:]

    # # K means Clustering
    #     no_of_centroid = 2
    #     k_means_centroids,k_means_clusters,wcss =
kmeans(dataset,no_of_centroid,test = 0)
    # K-means Clustering
        max_clusters = 10
        wcss_values = []
        for no_of_centroid in range(1, max_clusters + 1):
            centroids, clusters, wcss = kmeans(dataset,
no_of_centroid,test = 1)
            wcss_values.append(wcss)

        # Plot WCSS to find the "elbow point"
        plt.plot(range(1, max_clusters + 1), wcss_values, marker='o')
        plt.xlabel('Number of Clusters')
        plt.ylabel('WCSS')
        plt.title('Elbow Method for Optimal K')
        plt.savefig(f"Elbow Method for Optimal K {dataset_name}.png")
        plt.show()
```

```
        no_of_centroid = 2
        print(f"After Elbow Method, the optimal number of centroid is
{no_of_centroid}");
        centroids, clusters, wcss = kmeans(dataset,
no_of_centroid,test = 0)
```

Output:




```
Cluster 0 (97 points):
  Index: 50, Data Point: [7.  3.2 4.7 1.4]
  Index: 51, Data Point: [6.4 3.2 4.5 1.5]
  Index: 52, Data Point: [6.9 3.1 4.9 1.5]
  Index: 53, Data Point: [5.5 2.3 4.  1.3]
  Index: 54, Data Point: [6.5 2.8 4.6 1.5]
  Index: 55, Data Point: [5.7 2.8 4.5 1.3]
  Index: 56, Data Point: [6.3 3.3 4.7 1.6]
  Index: 58, Data Point: [6.6 2.9 4.6 1.3]
  Index: 59, Data Point: [5.2 2.7 3.9 1.4]
  Index: 60, Data Point: [5.  2.  3.5 1. ]
  Index: 61, Data Point: [5.9 3.  4.2 1.5]
  Index: 62, Data Point: [6.  2.2 4.  1. ]
  Index: 63, Data Point: [6.1 2.9 4.7 1.4]
  Index: 64, Data Point: [5.6 2.9 3.6 1.3]
  Index: 65, Data Point: [6.7 3.1 4.4 1.4]
  Index: 66, Data Point: [5.6 3.  4.5 1.5]
  Index: 67, Data Point: [5.8 2.7 4.1 1. ]
  Index: 68, Data Point: [6.2 2.2 4.5 1.5]
  Index: 69, Data Point: [5.6 2.5 3.9 1.1]
  Index: 70, Data Point: [5.9 3.2 4.8 1.8]
  Index: 71, Data Point: [6.1 2.8 4.  1.3]
  Index: 72, Data Point: [6.3 2.5 4.9 1.5]
  Index: 73, Data Point: [6.1 2.8 4.7 1.2]
```

```
Index: 74, Data Point: [6.4 2.9 4.3 1.3]
Index: 75, Data Point: [6.6 3.  4.4 1.4]
Index: 76, Data Point: [6.8 2.8 4.8 1.4]
Index: 77, Data Point: [6.7 3.  5.  1.7]
Index: 78, Data Point: [6.  2.9 4.5 1.5]
Index: 79, Data Point: [5.7 2.6 3.5 1. ]
Index: 80, Data Point: [5.5 2.4 3.8 1.1]
Index: 81, Data Point: [5.5 2.4 3.7 1. ]
Index: 82, Data Point: [5.8 2.7 3.9 1.2]
Index: 83, Data Point: [6.  2.7 5.1 1.6]
Index: 84, Data Point: [5.4 3.  4.5 1.5]
Index: 85, Data Point: [6.  3.4 4.5 1.6]
Index: 86, Data Point: [6.7 3.1 4.7 1.5]
Index: 87, Data Point: [6.3 2.3 4.4 1.3]
Index: 88, Data Point: [5.6 3.  4.1 1.3]
Index: 89, Data Point: [5.5 2.5 4.  1.3]
Index: 90, Data Point: [5.5 2.6 4.4 1.2]
Index: 91, Data Point: [6.1 3.  4.6 1.4]
Index: 92, Data Point: [5.8 2.6 4.  1.2]
Index: 94, Data Point: [5.6 2.7 4.2 1.3]
Index: 95, Data Point: [5.7 3.  4.2 1.2]
Index: 96, Data Point: [5.7 2.9 4.2 1.3]
Index: 97, Data Point: [6.2 2.9 4.3 1.3]
Index: 99, Data Point: [5.7 2.8 4.1 1.3]
Index: 100, Data Point: [6.3 3.3 6.  2.5]
Index: 101, Data Point: [5.8 2.7 5.1 1.9]
Index: 102, Data Point: [7.1 3.  5.9 2.1]
Index: 103, Data Point: [6.3 2.9 5.6 1.8]
Index: 104, Data Point: [6.5 3.  5.8 2.2]
Index: 105, Data Point: [7.6 3.  6.6 2.1]
Index: 106, Data Point: [4.9 2.5 4.5 1.7]
Index: 107, Data Point: [7.3 2.9 6.3 1.8]
Index: 108, Data Point: [6.7 2.5 5.8 1.8]
Index: 109, Data Point: [7.2 3.6 6.1 2.5]
Index: 110, Data Point: [6.5 3.2 5.1 2. ]
Index: 111, Data Point: [6.4 2.7 5.3 1.9]
Index: 112, Data Point: [6.8 3.  5.5 2.1]
Index: 113, Data Point: [5.7 2.5 5.  2. ]
Index: 114, Data Point: [5.8 2.8 5.1 2.4]
Index: 115, Data Point: [6.4 3.2 5.3 2.3]
Index: 116, Data Point: [6.5 3.  5.5 1.8]
Index: 117, Data Point: [7.7 3.8 6.7 2.2]
Index: 118, Data Point: [7.7 2.6 6.9 2.3]
Index: 119, Data Point: [6.  2.2 5.  1.5]
Index: 120, Data Point: [6.9 3.2 5.7 2.3]
Index: 121, Data Point: [5.6 2.8 4.9 2. ]
Index: 122, Data Point: [7.7 2.8 6.7 2. ]
Index: 123, Data Point: [6.3 2.7 4.9 1.8]
Index: 124, Data Point: [6.7 3.3 5.7 2.1]
Index: 125, Data Point: [7.2 3.2 6.  1.8]
Index: 126, Data Point: [6.2 2.8 4.8 1.8]
Index: 127, Data Point: [6.1 3.  4.9 1.8]
Index: 128, Data Point: [6.4 2.8 5.6 2.1]
Index: 129, Data Point: [7.2 3.  5.8 1.6]
Index: 130, Data Point: [7.4 2.8 6.1 1.9]
Index: 131, Data Point: [7.9 3.8 6.4 2. ]
Index: 132, Data Point: [6.4 2.8 5.6 2.2]
Index: 133, Data Point: [6.3 2.8 5.1 1.5]
Index: 134, Data Point: [6.1 2.6 5.6 1.4]
Index: 135, Data Point: [7.7 3.  6.1 2.3]
Index: 136, Data Point: [6.3 3.4 5.6 2.4]
```

```
  Index: 137, Data Point: [6.4 3.1 5.5 1.8]
  Index: 138, Data Point: [6.  3.  4.8 1.8]
  Index: 139, Data Point: [6.9 3.1 5.4 2.1]
  Index: 140, Data Point: [6.7 3.1 5.6 2.4]
  Index: 141, Data Point: [6.9 3.1 5.1 2.3]
  Index: 142, Data Point: [5.8 2.7 5.1 1.9]
  Index: 143, Data Point: [6.8 3.2 5.9 2.3]
  Index: 144, Data Point: [6.7 3.3 5.7 2.5]
  Index: 145, Data Point: [6.7 3.  5.2 2.3]
  Index: 146, Data Point: [6.3 2.5 5.  1.9]
  Index: 147, Data Point: [6.5 3.  5.2 2. ]
  Index: 148, Data Point: [6.2 3.4 5.4 2.3]
  Index: 149, Data Point: [5.9 3.  5.1 1.8]

Cluster 1 (53 points):
  Index: 0, Data Point: [5.1 3.5 1.4 0.2]
  Index: 1, Data Point: [4.9 3.  1.4 0.2]
  Index: 2, Data Point: [4.7 3.2 1.3 0.2]
  Index: 3, Data Point: [4.6 3.1 1.5 0.2]
  Index: 4, Data Point: [5.  3.6 1.4 0.2]
  Index: 5, Data Point: [5.4 3.9 1.7 0.4]
  Index: 6, Data Point: [4.6 3.4 1.4 0.3]
  Index: 7, Data Point: [5.  3.4 1.5 0.2]
  Index: 8, Data Point: [4.4 2.9 1.4 0.2]
  Index: 9, Data Point: [4.9 3.1 1.5 0.1]
  Index: 10, Data Point: [5.4 3.7 1.5 0.2]
  Index: 11, Data Point: [4.8 3.4 1.6 0.2]
  Index: 12, Data Point: [4.8 3.  1.4 0.1]
  Index: 13, Data Point: [4.3 3.  1.1 0.1]
  Index: 14, Data Point: [5.8 4.  1.2 0.2]
  Index: 15, Data Point: [5.7 4.4 1.5 0.4]
  Index: 16, Data Point: [5.4 3.9 1.3 0.4]
  Index: 17, Data Point: [5.1 3.5 1.4 0.3]
  Index: 18, Data Point: [5.7 3.8 1.7 0.3]
  Index: 19, Data Point: [5.1 3.8 1.5 0.3]
  Index: 20, Data Point: [5.4 3.4 1.7 0.2]
  Index: 21, Data Point: [5.1 3.7 1.5 0.4]
  Index: 22, Data Point: [4.6 3.6 1.  0.2]
  Index: 23, Data Point: [5.1 3.3 1.7 0.5]
  Index: 24, Data Point: [4.8 3.4 1.9 0.2]
  Index: 25, Data Point: [5.  3.  1.6 0.2]
  Index: 26, Data Point: [5.  3.4 1.6 0.4]
  Index: 27, Data Point: [5.2 3.5 1.5 0.2]
  Index: 28, Data Point: [5.2 3.4 1.4 0.2]
  Index: 29, Data Point: [4.7 3.2 1.6 0.2]
  Index: 30, Data Point: [4.8 3.1 1.6 0.2]
  Index: 31, Data Point: [5.4 3.4 1.5 0.4]
  Index: 32, Data Point: [5.2 4.1 1.5 0.1]
  Index: 33, Data Point: [5.5 4.2 1.4 0.2]
  Index: 34, Data Point: [4.9 3.1 1.5 0.1]
  Index: 35, Data Point: [5.  3.2 1.2 0.2]
  Index: 36, Data Point: [5.5 3.5 1.3 0.2]
  Index: 37, Data Point: [4.9 3.1 1.5 0.1]
  Index: 38, Data Point: [4.4 3.  1.3 0.2]
  Index: 39, Data Point: [5.1 3.4 1.5 0.2]
  Index: 40, Data Point: [5.  3.5 1.3 0.3]
  Index: 41, Data Point: [4.5 2.3 1.3 0.3]
  Index: 42, Data Point: [4.4 3.2 1.3 0.2]
  Index: 43, Data Point: [5.  3.5 1.6 0.6]
  Index: 44, Data Point: [5.1 3.8 1.9 0.4]
  Index: 45, Data Point: [4.8 3.  1.4 0.3]
```

```
        Index: 46, Data Point: [5.1 3.8 1.6 0.2]
        Index: 47, Data Point: [4.6 3.2 1.4 0.2]
        Index: 48, Data Point: [5.3 3.7 1.5 0.2]
        Index: 49, Data Point: [5.  3.3 1.4 0.2]
        Index: 57, Data Point: [4.9 2.4 3.3 1. ]
        Index: 93, Data Point: [5.  2.3 3.3 1. ]
        Index: 98, Data Point: [5.1 2.5 3.  1.1]
      New centroids are : [[6.30103093 2.88659794 4.95876289 1.69587629]
       [5.00566038 3.36037736 1.56226415 0.28867925]]
      Convergence reached after 5 iterations.
**This is the final clusters only.
```

Conclusion:

K-Means Clustering is an Unsupervised Machine Learning algorithm which groups the unlabeled dataset into different clusters. Here Iris dataset is clustered in 2 groups as per the elbow finding method.

Problem Statement:

K-medoids Clustering on the Iris Dataset.

Data Description:

It includes three iris species with 50 samples each as well as some properties about each flower. One flower species is linearly separable from the other two, but the other two are not linearly separable from each other.
The columns in this dataset are:

1. SepalLengthCm
2. SepalWidthCm
3. PetalLengthCm
4. PetalWidthCm

Algorithm:

1. Start with an initial set of medoids (selected randomly or using an initialization method).
2. Iteratively:
   - Replace a medoid with a non-medoid data point if it reduces the total sum of distances (sum of squared errors, SSE) within the resulting cluster.
   - Continue this process until no further improvement can be made.

   The objective is to minimize the Sum of Squared Errors (SSE), which is defined as:

$$\boldsymbol{SSE} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{K}} \sum_{x \in C_i} \text{dist}^2(m_i, x)$$

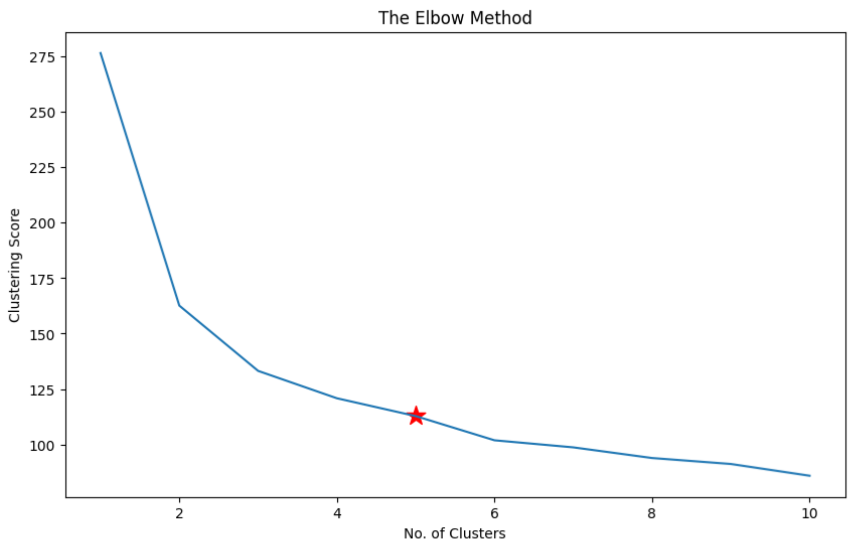
Program:

```
import numpy as np
```

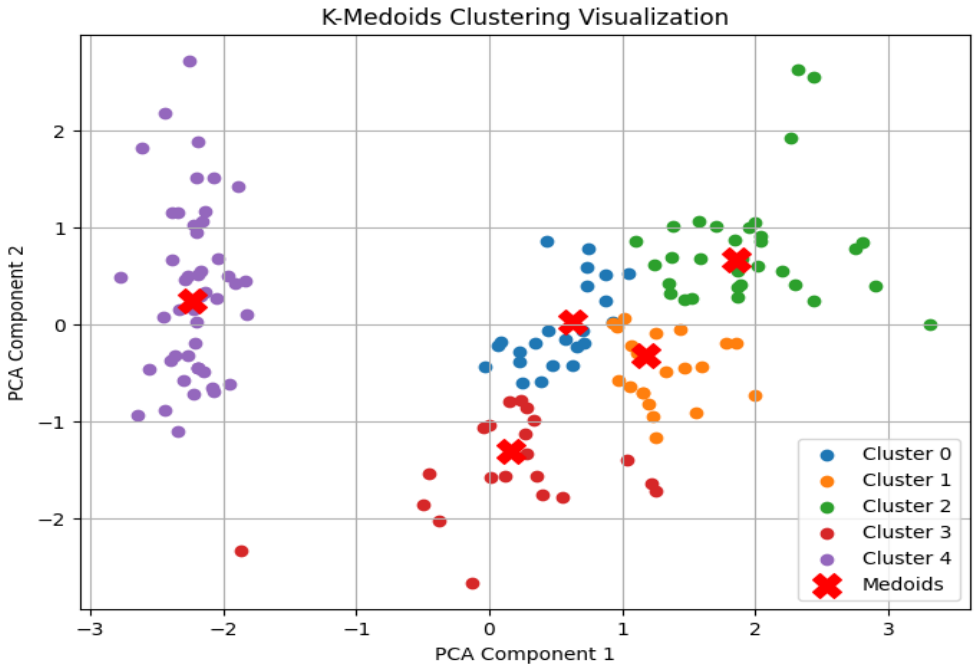
```
from sklearn_extra.cluster import KMedoids
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import pandas as pd


# Sample data with 4 columns
dataset = pd.read_csv("/content/Iris.csv")
X_data = dataset.iloc[:,1:-1]
data = X_data.to_numpy()

# Step 1: Normalize the data
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)


clustering_score = []
for i in range(1, 11):
    kmedoids = KMedoids(n_clusters = i, init = 'random',
random_state = 42)
    kmedoids.fit(data_scaled)
    clustering_score.append(kmedoids.inertia_) # inertia_ = Sum
of squared distances of samples to their closest cluster center.


plt.figure(figsize=(10,6))
plt.plot(range(1, 11), clustering_score)
plt.scatter(5,clustering_score[4], s = 200, c = 'red',
marker='*')
plt.title('The Elbow Method')
plt.xlabel('No. of Clusters')
plt.ylabel('Clustering Score')
plt.show()


# Step 2: Apply K-medoids
k = 5  # Choose the number of clusters
kmedoids = KMedoids(n_clusters=k, metric='euclidean',
random_state=42)
labels = kmedoids.fit_predict(data_scaled)

# Step 3: Evaluate the clustering
print("Cluster labels:", labels)
print("Medoids:", kmedoids.cluster_centers_)
print("Silhouette Score:", silhouette_score(data_scaled, labels))

# Step 4: Reduce dimensions for visualization
pca = PCA(n_components=2)
data_2d = pca.fit_transform(data_scaled)

# Step 5: Plot the clusters
```

```
plt.figure(figsize=(8, 6))
for cluster in np.unique(labels):
    plt.scatter(data_2d[labels == cluster, 0], data_2d[labels ==
cluster, 1], label=f'Cluster {cluster}')

# Plot the medoids
medoid_2d = pca.transform(kmedoids.cluster_centers_)
plt.scatter(medoid_2d[:, 0], medoid_2d[:, 1], c='red',
marker='X', s=200, label='Medoids')

plt.title('K-Medoids Clustering Visualization')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.legend()
plt.grid()
plt.show()
```

Output:




```
Cluster labels: [4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4
 4 4 4 4 3 4 4 4 4 4 4 4 4 2 0 2 3 1 0 0 3 0 3 3 0 3 0 0 0 0 3 3 3
0 0 1 0
 0 0 1 2 0 3 3 3 3 1 0 0 0 3 0 3 3 0 3 3 3 0 0 0 3 0 2 1 2 1 2 2 3
2 1 2 2
 1 2 1 1 2 2 2 2 3 2 1 2 1 2 2 1 1 1 2 2 2 1 1 1 2 2 2 1 2 2 2 1 2
2 2 1 2
 2 1]
Medoids: [[ 0.31099753 -0.1249576   0.47843012  0.26469891]
 [ 0.4321654  -0.58776353  0.59216153  0.79059079]
 [ 1.2803405   0.10644536  0.93335575  1.1850097 ]
 [-0.29484182 -1.28197243  0.08037019 -0.12972   ]
 [-1.02184904  0.80065426 -1.2844067  -1.31297673]]
Silhouette Score: 0.3759993156661742
```



Conclusion:

K-Medoids clustering mechanism in Partition Clustering. First, Clustering is the process of breaking down an abstract group of data points/ objects into classes of similar objects such that all the objects in one cluster have similar traits. A group of n objects is broken down into k number of clusters based on their similarities. Here I have use PCA(Principal Component Analysis) for dimension reduction, which makes it easier to represent in 2D space.

Problem Statement:

Apply AGNES (single linkage, complete linkage, average linkage) on Iris dataset for clustering.

Dataset Description:

The Iris dataset is one of the most famous datasets in machine learning and statistics. It is widely used for classification and clustering tasks. The dataset consists of measurements of 150 iris flowers, where each sample corresponds to a single flower from one of three species.

Details of the Iris Dataset:

Number of samples: 150 (50 samples from each of 3 species)

Number of features: 4 features (measurements of the flowers)

Number of classes (species): 3 species

Species: Setosa, Versicolor, Virginica

Features (Columns):

Sepal Length (cm) – The length of the sepal.

Sepal Width (cm) – The width of the sepal.

Petal Length (cm) – The length of the petal.

Petal Width (cm) – The width of the petal.

Procedure:

Step 1: Initialize Each Data Point as a Separate Cluster.

Step 2: Compute the Distance Between Clusters using euclidean distance.

Step 3: Merge the Two Closest Clusters.

Step 4: Update the Distance Matrix according to the linkage criteria (Single, complete, average).

Step 5: Repeat steps 3 and 4 until all points are in a single cluster.

Step 6: The final result of AGNES is a dendrogram, which shows how clusters are merged over time.

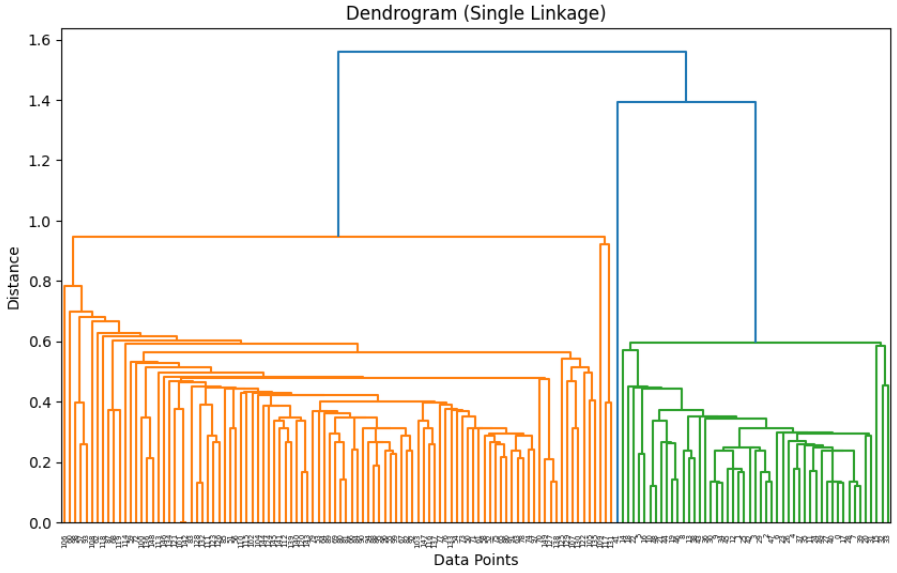
Source code:

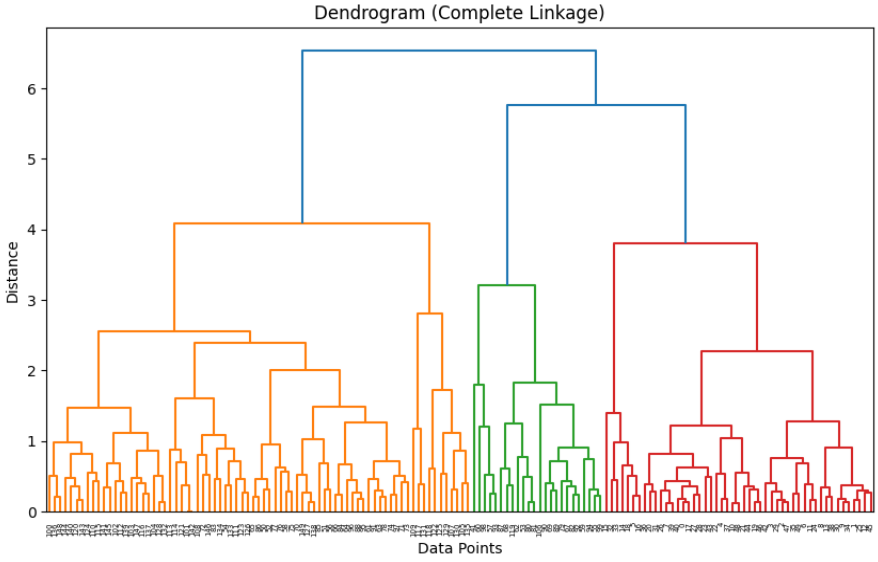
```
import numpy as np

import pandas as pd

from sklearn import datasets

from sklearn.preprocessing import StandardScaler,
MinMaxScaler

from sklearn.cluster import AgglomerativeClustering

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

from scipy.cluster.hierarchy import linkage, dendrogram


# Load the Iris dataset

iris = datasets.load_iris()

X = iris.data  # Features (Sepal Length, Sepal Width,
Petal Length, Petal Width)

y = iris.target  # True labels (not used in unsupervised
learning)


# Convert to DataFrame for better preprocessing

df = pd.DataFrame(X, columns=iris.feature_names)


# Handle missing values (if any) by filling with mean
values

df.fillna(df.mean(), inplace=True)
```

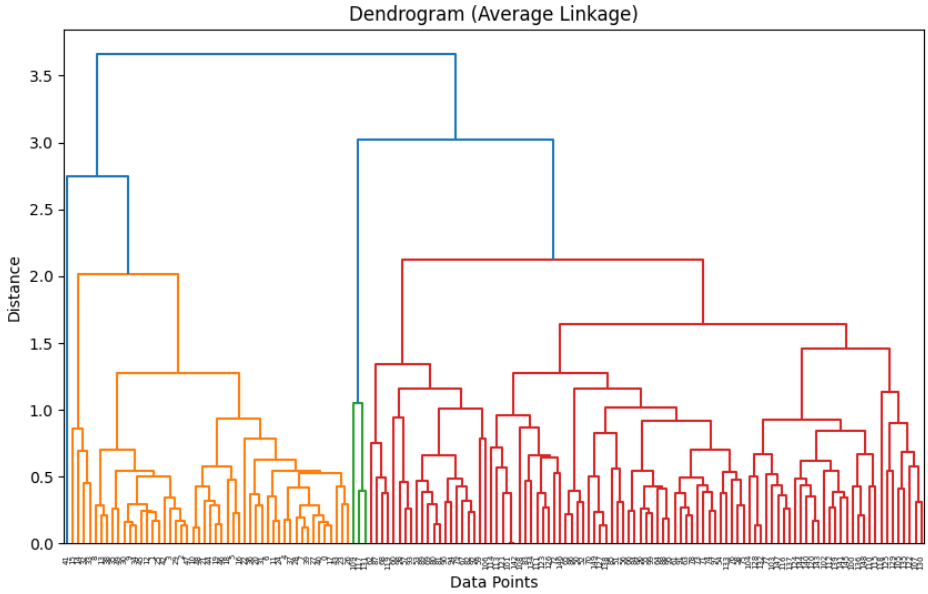
```
# Remove duplicate rows (if any)
df.drop_duplicates(inplace=True)

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df)

# Define a function to apply AGNES with different linkage
methods
def agnes_clustering(linkage_method):
    agnes = AgglomerativeClustering(n_clusters=3,
linkage=linkage_method)
    cluster_labels = agnes.fit_predict(X_scaled)

    # Compute linkage matrix for dendrogram
    Z = linkage(X_scaled, method=linkage_method)

    # Reduce dimensionality for visualization using PCA
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X_scaled)

    # Plot the clusters
    plt.figure(figsize=(8, 6))
    for cluster in range(3):
        plt.scatter(
            X_pca[np.where(cluster_labels == cluster), 0],
            X_pca[np.where(cluster_labels == cluster), 1],
            label=f"Cluster {cluster + 1}"
        )
```

```
    plt.title(f"AGNES Clustering
({linkage_method.capitalize()} Linkage)")
    plt.xlabel("Principal Component 1")
    plt.ylabel("Principal Component 2")
    plt.legend()
    plt.show()

    # Plot the dendrogram
    plt.figure(figsize=(10, 6))
    dendrogram(Z)
    plt.title(f"Dendrogram ({linkage_method.capitalize()}
Linkage)")
    plt.xlabel("Data Points")
    plt.ylabel("Distance")
    plt.show()

# Apply AGNES with single linkage
agnes_clustering("single")

# Apply AGNES with complete linkage
agnes_clustering("complete")

# Apply AGNES with average linkage
agnes_clustering("average")
```

Output:


Dendrogram (Single Linkage)
Distance
Data Points


Dendrogram (Complete Linkage)
Distance
Data Points


Dendrogram (Average Linkage)
Distance
Data Points

Discussion:

Agglomerative Clustering (AGNES) is a hierarchical clustering algorithm. Here we have used PCA (Principal component analysis) component to reduce the dimension of the problem and make visualization easier. You can achieve same with Euclidian distance but you can not visualize the dataset.

Problem statement:

Apply DIANA (single linkage, complete linkage, average linkage) on Iris dataset for clustering.

Dataset Description:

The Iris dataset is one of the most famous datasets in machine learning and statistics. It is widely used for classification and clustering tasks. The dataset consists of measurements of 150 iris flowers, where each sample corresponds to a single flower from one of three species.

Details of the Iris Dataset:

Number of samples: 150 (50 samples from each of 3 species)

Number of features: 4 features (measurements of the flowers)

Number of classes (species): 3 species

Species: Setosa, Versicolor, Virginica

Features (Columns):

Sepal Length (cm) – The length of the sepal.

Sepal Width (cm) – The width of the sepal.

Petal Length (cm) – The length of the petal.

Petal Width (cm) – The width of the petal.

Procedure:

Step 1: Start with all data points as one cluster.

Step 2: Compute the dissimilarity or distance matrix for all data points.

Step 3: Identify the two most dissimilar (or least similar) observations within the cluster (using the dissimilarity matrix).

Step 4: Split the cluster into two smaller clusters. This can be done by selecting the most dissimilar point and assigning it to one of the new clusters, while the remaining points are assigned to the other cluster.

Step 5: Once the cluster is split, update the dissimilarity matrix for the remaining clusters.

Step 6: Repeat steps 3-5 iteratively for the newly created clusters. At each step, find the most dissimilar cluster, and then split it into two smaller clusters.

Step 7: The process continues until all individual data points are in their own clusters.

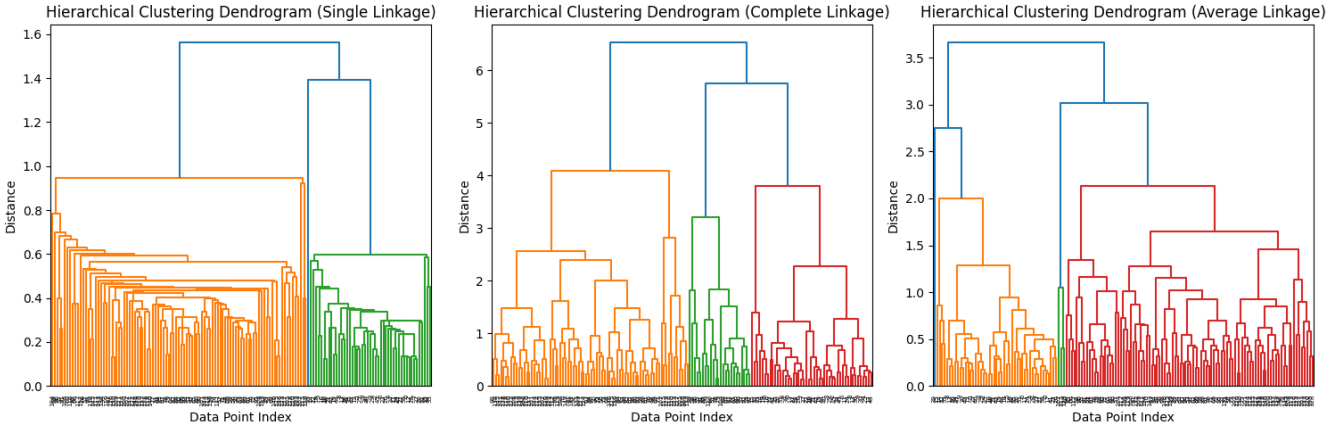
Step 8: The output of DIANA is a hierarchical clustering tree (dendrogram), which shows how clusters were split at each step.

SOURCE CODE:

```
import numpy as np
import pandas as pd
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler


# Load the Iris dataset
df = pd.read_csv('Iris.csv')


# Drop the 'Id' and 'Species' columns as they are not part
of the feature set
X = df.drop(columns=['Id', 'Species'])


# Handle missing values by filling with mean values
X.fillna(X.mean(), inplace=True)
```

```
# Remove duplicate rows
X.drop_duplicates(inplace=True)

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Linkage methods to evaluate
linkage_methods = ['single', 'complete', 'average']

# Plot dendrograms for each linkage method
plt.figure(figsize=(15, 5))
for i, method in enumerate(linkage_methods, 1):
    # Perform hierarchical clustering
    Z = linkage(X_scaled, method)

    # Plot dendrogram
    plt.subplot(1, 3, i)
    dendrogram(Z)
    plt.title(f'Hierarchical Clustering Dendrogram
({method.capitalize()} Linkage)')
    plt.xlabel('Data Point Index')
    plt.ylabel('Distance')

plt.tight_layout()
plt.show()
```

Output:



Discussion:

Divisive Analysis clustering (DIANA) is a hierarchical clustering algorithm. Here we have used PCA (Principal component analysis) component to reduce the dimension of the problem and make visualization easier. You can achieve same with Euclidian distance but you cannot visualize the dataset.

Problem Statement:

Draw decision tree using ID3 algorithm on golf playing dataset.

Dataset Description:

The dataset you provided appears to be related to predicting whether or not people will play golf based on different weather conditions.

Columns:

Outlook: Describes the weather outlook.

Categories: Sunny, Overcast, Rain.

Temp. (Temperature): Describes the temperature during the day.

Categories: Hot, Mild, Cool.

Humidity: Describes the humidity level.

Categories: High, Normal.

Wind: Describes the wind speed.

Categories: Weak, Strong.

Decision: The decision of whether or not to play golf.

Categories: Yes, No.

The dataset contains 14 records, each representing one instance of weather conditions and the corresponding decision regarding playing golf. The columns are a mix of categorical features (Outlook, Temp., Humidity, Wind) and a binary target variable (Decision).

Procedure:

Step 1: Begin with all the data points in the root of the tree.

Step 2: Calculate entropy for the dataset.

Step 3: For each attribute, calculate the information gain.

Step 4: Select the attribute with the highest information gain to split the dataset.

Step 5: Split the dataset based on the chosen attribute's values.

Step 6: Repeat the process recursively for each subset.

Step 7: Stop when one of the stopping conditions is met (pure subsets or no attributes left to split).

Source code:

```
import numpy as np

import pandas as pd

import math

from collections import Counter

import graphviz


# Load dataset

df_data = pd.read_csv('GolfPlay.csv')


# Handle missing values by filling with the mode of each
column

df_data.fillna(df_data.mode().iloc[0], inplace=True)


# Remove duplicate rows

df_data.drop_duplicates(inplace=True)


# Convert categorical data into numerical format

df = df_data.apply(lambda x: x.astype('category').cat.codes if
x.dtype == 'object' else x)


# Entropy Calculation

def entropy(y):

    counter = Counter(y)

    total = len(y)

    return -sum((count/total) * math.log2(count/total) for
count in counter.values())


# Information Gain Calculation

def information_gain(df, feature, target):
```

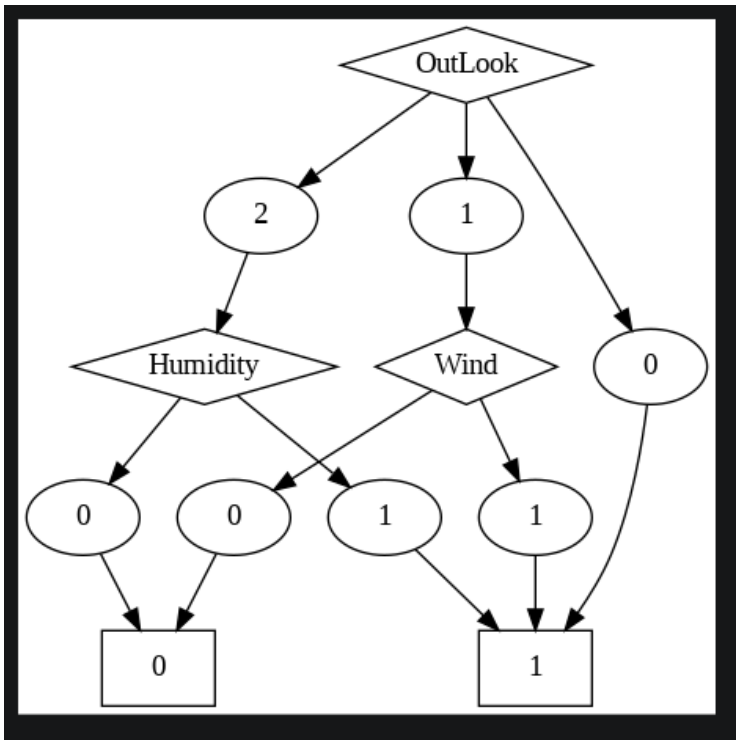
```
    total_entropy = entropy(df[target])

    values = df[feature].unique()

    weighted_entropy = sum((len(df[df[feature] == v]) /
len(df)) * entropy(df[df[feature] == v][target]) for v in
values)

    return total_entropy - weighted_entropy


# ID3 Algorithm - Recursively Build the Decision Tree

def id3(df, features, target, tree=None):

    if len(df[target].unique()) == 1:

        return df[target].iloc[0]

    if len(features) == 0:

        return df[target].mode()[0]


    best_feature = max(features, key=lambda f:
information_gain(df, f, target))

    tree = {best_feature: {}}


    for value in df[best_feature].unique():

        subset = df[df[best_feature] == value]

        tree[best_feature][value] = id3(subset, [f for f in
features if f != best_feature], target)


    return tree


# Build the decision tree

features = df.columns[:-1]

target = 'Decision'

decision_tree = id3(df, features, target)
```

```
# Print Decision Tree
import pprint
pprint.pprint(decision_tree)


# Function to visualize the tree
def visualize_tree(tree, parent=None, graph=None):
    if graph is None:
        graph = graphviz.Digraph(format="png")

    if isinstance(tree, dict):
        for node, sub_tree in tree.items():
            if parent is None:
                graph.node(node, label=node, shape="diamond")
            else:
                graph.node(node, label=node, shape="diamond")
                graph.edge(parent, node)

            for value, branch in sub_tree.items():
                child_name = f"{node}_{value}"
                graph.node(child_name, label=str(value))
                graph.edge(node, child_name)
                visualize_tree(branch, parent=child_name,
graph=graph)
    else:
        graph.node(str(tree), label=str(tree), shape="box")
        graph.edge(parent, str(tree))

    return graph
```

```
# Draw the decision tree
graph = visualize_tree(decision_tree)
graph.render("decision_tree", view=True)
```

Output:

```
{'OutLook': {0: 1, 1: {'Wind': {0: 0, 1: 1}}, 2: {'Humidity': {0: 0, 1: 1}}}}
```



Discussion:

ID3 is a simple yet powerful decision tree algorithm that can be an excellent choice for classification tasks where interpretability is important. While it has some limitations, such as a tendency to overfit and favor features with more categories.

Problem Statement:

Apply CART on buy computer dataset.

Dataset Description:

The dataset contains 14 records and is used to predict whether a person will buy a computer based on various features:

Columns:

id: Unique identifier.

age: Age group (youth, middle_age, senior).

income: Income level (high, medium, low).

student: Whether the person is a student (yes/no).

credit_rating: Credit rating (fair, excellent).

Buy_Computer: Target variable (yes/no), indicating if the person buys a computer.

Objective: Predict whether an individual will buy a computer based on age, income, student status, and credit rating.

Target: Buy_Computer (yes/no).

Procedure:

Step 1: Select the best feature and threshold to split the data at each node based on Gini impurity (for classification).

Step 2: Split the data into two subsets based on the chosen feature and threshold.

Step 3: Repeat the splitting process recursively for each subset until one of the stopping conditions is met.

Step 4: Assign class labels or target values to leaf nodes.

Step 5: Prune the tree to avoid overfitting, if necessary.

Step 6: Repeat the process recursively for each subset.

Step 7: Stop when one of the stopping conditions is met (pure subsets or no attributes left to split).

Source code:

```
import numpy as np
import pandas as pd
import math
from collections import Counter
import graphviz


# Load dataset
df = pd.read_csv('Buy_Computer.csv')
df_data = df.iloc[:, 1:]


# Handle missing values by filling with the mode of each
column
df_data.fillna(df_data.mode().iloc[0], inplace=True)


# Remove duplicate rows
df_data.drop_duplicates(inplace=True)


# Convert categorical data into numerical format
```

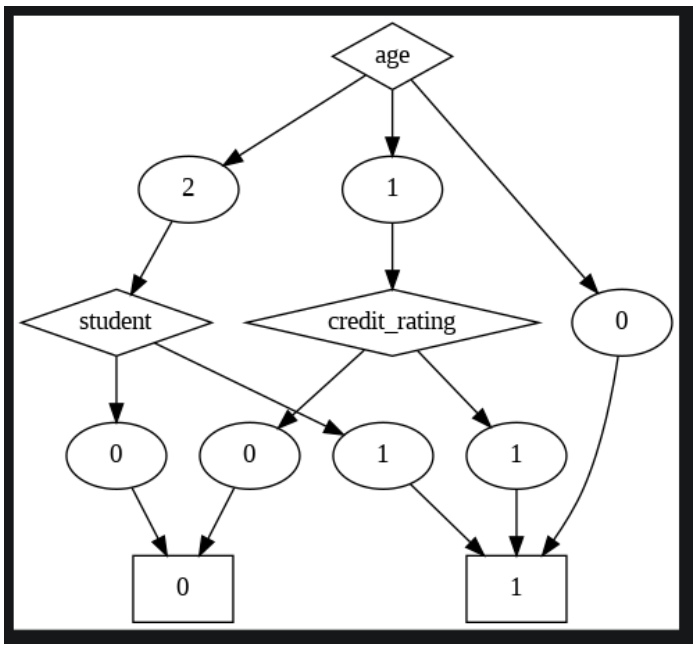
```
df = df_data.apply(lambda x: x.astype('category').cat.codes if
x.dtype == 'object' else x)


# Entropy Calculation

def entropy(y):

    counter = Counter(y)

    total = len(y)

    return -sum((count/total) * math.log2(count/total) for
count in counter.values())


# Information Gain Calculation

def information_gain(df, feature, target):

    total_entropy = entropy(df[target])

    values = df[feature].unique()

    weighted_entropy = sum((len(df[df[feature] == v]) /
len(df)) * entropy(df[df[feature] == v][target]) for v in
values)

    return total_entropy - weighted_entropy


# ID3 Algorithm - Recursively Build the Decision Tree

def id3(df, features, target, tree=None):

    if len(df[target].unique()) == 1:

        return df[target].iloc[0]

    if len(features) == 0:

        return df[target].mode()[0]


    best_feature = max(features, key=lambda f:
information_gain(df, f, target))

    tree = {best_feature: {}}


    for value in df[best_feature].unique():

        subset = df[df[best_feature] == value]

        tree[best_feature][value] = id3(subset, [f for f in
features if f != best_feature], target)
```

```
    return tree

# Build the decision tree
features = df.columns[:-1]
target = 'Buy_Computer'
decision_tree = id3(df, features, target)

# Print Decision Tree
import pprint
pprint.pprint(decision_tree)

# Function to visualize the tree
def visualize_tree(tree, parent=None, graph=None):
    if graph is None:
        graph = graphviz.Digraph(format="png")

    if isinstance(tree, dict):
        for node, sub_tree in tree.items():
            if parent is None:
                graph.node(node, label=node, shape="diamond")
            else:
                graph.node(node, label=node, shape="diamond")
                graph.edge(parent, node)

            for value, branch in sub_tree.items():
                child_name = f"{node}_{value}"
                graph.node(child_name, label=str(value))
                graph.edge(node, child_name)
                visualize_tree(branch, parent=child_name,
graph=graph)
```

```
        else:
            graph.node(str(tree), label=str(tree), shape="box")
            graph.edge(parent, str(tree))

        return graph

    # Draw the decision tree
    graph = visualize_tree(decision_tree)
    graph.render("decision_tree", view=True)
```

Output:

```
{'age': {0: 1,
         1: {'credit_rating': {0: 0, 1: 1}},
         2: {'student': {0: 0, 1: 1}}}}
```



Discussion:

This process involves efficient preprocessing, where missing values are filled with the mode, duplicate rows are removed, and categorical features are encoded using LabelEncoder. The decision tree is both **visualized using graphviz** for a graphical representation and

printed as a **text-based tree** for easier interpretation. This workflow ensures that the data is clean, the model is simple yet effective, and both graphical and textual outputs are provided for better transparency.

Problem Statement:

Apply naïve Bayesian algorithm on buy computer dataset to identify class label of unknown samples.

Dataset Description:

The dataset you provided appears to be related to predicting whether or not people will play golf based on different weather conditions.

Columns:

Outlook: Describes the weather outlook.

Categories: Sunny, Overcast, Rain.

Temp. (Temperature): Describes the temperature during the day.

Categories: Hot, Mild, Cool.

Humidity: Describes the humidity level.

Categories: High, Normal.

Wind: Describes the wind speed.

Categories: Weak, Strong.

Decision: The decision of whether or not to play golf.

Categories: Yes, No.

The dataset contains 10 records, each representing one instance of weather conditions and the corresponding decision regarding playing golf. The columns are a mix of categorical features (Outlook, Temp., Humidity, Wind) and a binary target variable (Decision).

Procedure:

Step 1: Prepare the Data: Handle missing values, encode categorical variables using LabelEncoder.

Step 2: Divide the dataset into training and testing sets.

Step 3:

Calculate the prior probability of each class in the target variable:

$$Total\ number\ of\ \text{samples}\mathrm{P(C)} = \frac{\text{Number of samples in class C}}{\text{Total number of samples}}$$

Step 4: Calculate Conditional Probabilities.

Step 5: Calculate the probabilities using Bayesian formula.

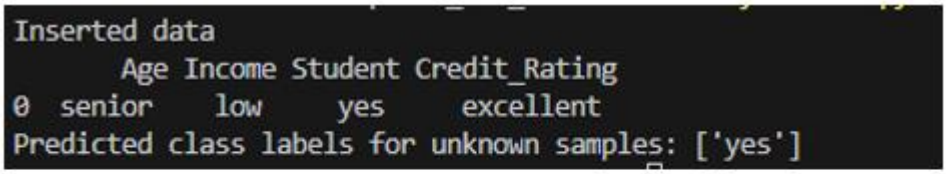
Source Code:

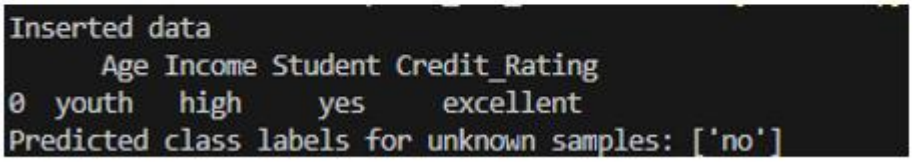
```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB

from sklearn.preprocessing import LabelEncoder


# Dataset

data = {

    'Age': ['youth', 'youth', 'youth', 'middle', 'senior', 'senior',
'middle', 'youth', 'middle', 'senior'],

    'Income': ['low', 'high', 'high', 'medium', 'low', 'low', 'high',
'medium', 'low', 'medium'],

    'Student': ['no', 'yes', 'no', 'yes', 'yes', 'yes', 'yes', 'no',
'no', 'yes'],

    'Credit_Rating': ['fair', 'excellent', 'fair', 'fair', 'fair',
'excellent', 'excellent', 'fair', 'excellent', 'fair'],

    'Buy_Computer': ['yes', 'no', 'yes', 'yes', 'yes', 'yes', 'yes',
'no', 'yes', 'yes']

}


# Create DataFrame

df = pd.DataFrame(data)


# Initialize LabelEncoder for each column separately

le_age = LabelEncoder()

le_income = LabelEncoder()

le_student = LabelEncoder()

le_credit_rating = LabelEncoder()

le_buy_computer = LabelEncoder()
```

```
# Fit the LabelEncoder on each column of the training data

df['Age'] = le_age.fit_transform(df['Age'])

df['Income'] = le_income.fit_transform(df['Income'])

df['Student'] = le_student.fit_transform(df['Student'])

df['Credit_Rating'] =
le_credit_rating.fit_transform(df['Credit_Rating'])

df['Buy_Computer'] =
le_buy_computer.fit_transform(df['Buy_Computer'])


# Features and target

X = df[['Age', 'Income', 'Student', 'Credit_Rating']]

y = df['Buy_Computer']


# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)


# Train a Naïve Bayes classifier

nb = GaussianNB()

nb.fit(X_train, y_train)


# Predict class labels for test samples

y_pred = nb.predict(X_test)


# Predict for unknown samples

unknown_samples = pd.DataFrame({

    'Age': ['senior'],

    'Income': ['low'],

    'Student': ['yes'],

    'Credit_Rating': ['excellent']

})


print(f"Inserted data\n", unknown_samples)
```

```
# Use the same LabelEncoder transformations to the unknown samples

unknown_samples['Age'] = le_age.transform(unknown_samples['Age'])

unknown_samples['Income'] =
le_income.transform(unknown_samples['Income'])

unknown_samples['Student'] =
le_student.transform(unknown_samples['Student'])

unknown_samples['Credit_Rating'] =
le_credit_rating.transform(unknown_samples['Credit_Rating'])


# Make predictions using the Naïve Bayes classifier

predicted_labels = nb.predict(unknown_samples[['Age', 'Income',
'Student', 'Credit_Rating']])

print("Predicted class labels for unknown samples:",
le_buy_computer.inverse_transform(predicted_labels))
```

Output:





Discussion:

The implementation of the Naive Bayes algorithm in code is a straightforward and efficient way to apply the model to classification tasks. The code typically involves key steps such as preprocessing the data, calculating prior probabilities, computing conditional probabilities (likelihoods), and applying Bayes' Theorem to predict the most likely class for new data instances.

Problem Statement:

Apply back propagation algorithm on sample {1,0,1} with the class label {1,0}. (Where network topology: 3-2-2-2, all biases and weights are initialized at 0)

Procedure:

Step 0: Initialize weights and biases to random number between 0 and 1 for all layers.

Step 1: Forward Pass (Propagation)

For each input neuron X_i calculate the weighted sum for the hidden layer neuron z_i :

$$z_i = \sum Xi * Wi + \theta$$

Where Wi is the corresponding weight and θ is the baising factor

For z_i output y_i will be:

$$\sigma(zi) = \frac{1}{1+e^{-z_i}}$$

Step 2: Calculate the Error (Loss)

Compute the error between the predicted output and the actual target label using the Mean Squared Error (MSE):

$$L = \frac{1}{2}\sum (yi - ai)^2$$

Step 4: Update Weights and Biases

Update the weights and biases using the Gradient Descent rule:

Update weights for each layer:

$$w_i = learning\_rate * \ change\ of\ w$$

Update biases for each layer:

$$b_i = leanring_rate * Errorofthenode$$

Step 5: Repeat the Process

Repeat the process for several iterations (epochs), updating weights and biases until the error converges to a minimal value, or until a predefined number of epochs is reached.

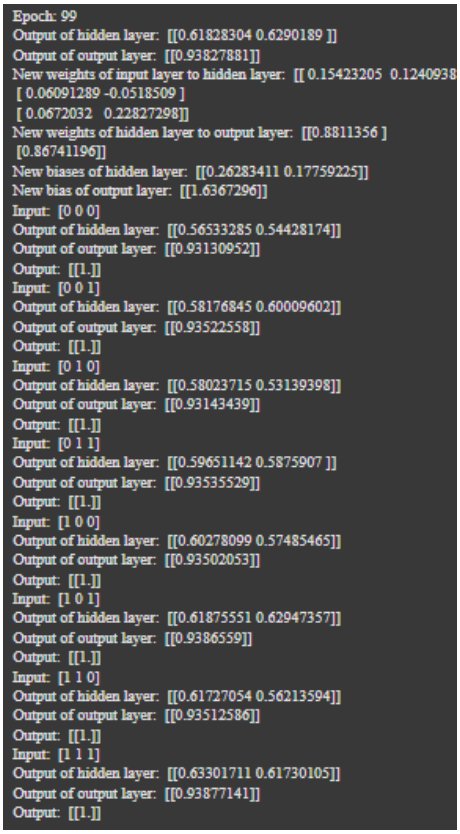
Source code:

```
import numpy as np


class NeuralNetwork():

    def __init__(self, input_layer_size, hidden_layer_size,
output_layer_size, learning_rate):

        self.input_size = input_layer_size

        self.hidden_size = hidden_layer_size

        self.output_size = output_layer_size

        #self.w1 = np.round(np.random.rand(self.input_size,
self.hidden_size) - 0.5, 3)

        #self.w2 = np.round(np.random.rand(self.hidden_size,
self.output_size) - 0.5, 3)

        #self.b1 = np.round(np.random.rand(1,
self.hidden_size) - 0.5,3)

        #self.b2 = np.round(np.random.rand(1,
self.output_size) - 0.5,3)

        self.w1 = np.array([[0.2,-0.3],[0.4, 0.1],[-
0.5,0.2]])

        self.w2 = np.array([-0.3,-0.2])

        self.b1 = np.array([-0.4,0.2])

        self.b2 = np.array([0.1])

        self.learning_rate = learning_rate

        self.error_list = []

        self.limit = 0.5

        self.true_positives = 0
```

```
            self.false_positives = 0

            self.true_negatives = 0

            self.false_negatives = 0

            print("Initial Input layer to Hidden layer weights:
", self.w1)

            print("Initial Hidden layer to output layer weights:
", self.w2)

            print("Initial Biases of Hidden layer: ", self.b1)

            print("Initial Bias of Output layer: ",self.b2)

        def __sigmoid(self, x):

            return 1 / (1 + np.exp(-x))

        def __sigmoid_derivative(self, x):

            return x * (1 - x)

        def __forward(self, X):

            self.z1 = np.dot(X, self.w1) + self.b1

            self.a1 = self.__sigmoid(self.z1)

            print("Output of hidden layer: ",self.a1)

            self.z2 = np.dot(self.a1, self.w2) + self.b2

            a2 =  self.__sigmoid(self.z2)

            print("Output of output layer: ",a2)

            return a2

        def __backpropagation(self, X, target, output):

          self.o_error = (target - output)

          self.o_delta = self.o_error *
    self.__sigmoid_derivative(output)

          print('delta Output: ',self.o_delta)

          self.z2_error = self.o_delta*self.w2

          print('Error of output layer : ',self.z2_error)

          self.z2_delta = self.z2_error *
    self.__sigmoid_derivative(self.a1)

          print('Delta error of output layer: ',self.z2_delta)

          self.w1 += self.learning_rate * np.outer(X,
    self.z2_delta)
```

```
      # Use np.dot to get the correct dimensions for
updating w2

      self.w2 += self.learning_rate *
np.dot(self.a1.reshape(self.hidden_size,self.output_size),
self.o_delta)

      self.b1 += self.learning_rate * self.z2_delta

      self.b2 += self.learning_rate * self.o_delta

      print('New weight of input layer to hidden layer:
',self.w1)

      print('New weight of hidden layer to output layer:
',self.w2)

      print('New baises of hidden layer: ',self.b1)

      print('New bias of output layer: ',self.b2)

    def train(self, X, y, epochs):

        for epoch in range(epochs):

            print('Epoch: ',epoch)

            o = self.__forward(X)

            self.__backpropagation(X, y, o)


    def predict(self, x_predicted):

        return self.__forward(x_predicted).item()


if __name__ == "__main__":

  n = NeuralNetwork(3,2,1,0.9)

  input = np.array([1,0,1])

  n.train(input,1,100)

  test =
np.array([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1
,1,0],[1,1,1]])

  for test in test:

    print('Input: ',test)

    print('Output: ',np.round(n.predict(test)))
```

Output:



Discussion:

This code implements a simple neural network with one hidden layer, using the sigmoid activation function and backpropagation for training. It includes methods for forward propagation (calculating the output of the network) and backpropagation (adjusting weights and biases based on error). The network is trained for a fixed number of epochs and can predict binary outputs (0 or 1) for given input data.

Problem Statement:

Apply fuzzy c means algorithm on Boston Housing Dataset.

Dataset Description:

The dataset contains 506 samples (rows) with 13 features (columns) and 1 target variable. Each sample represents a particular census tract in the Boston area, with attributes related to housing and demographics. The target variable is the median value of owner-occupied homes in thousands of dollars.

Features (Input Variables):

CRIM: Crime rate per capita (higher value means more crime).

ZN: Proportion of residential land zoned for large lots (in proportion).

INDUS: Proportion of non-retail business acres per town.

CHAS: Charles River dummy variable (1 if tract bounds the river; 0 otherwise).

NOX: Nitrogen oxides concentration (parts per 10 million).

RM: Average number of rooms per dwelling.

AGE: Proportion of owner-occupied units built before 1940.

DIS: Weighted distance to employment centers.

RAD: Index of accessibility to radial highways (higher values mean more access).

TAX: Property tax rate (per $10,000).

PTRATIO: Pupil-teacher ratio by town.

B: Proportion of Black residents by town (calculated as 1000(Bk − 0.63)^2 where Bk is the proportion of Black residents).

LSTAT: Percentage of lower status population.

Target Variable:

MEDV: Median value of homes in thousands of dollars

Procedure:

Step 1:

Set the number of clusters (C).

Set the fuzziness parameter (m, usually 2).

Set a convergence criterion or tolerance (epsilon) for when the algorithm will stop.

Step 2: Initialize Membership Matrix:

The sum of membership values for each data point should be 1, i.e.,

Step 3: Initialize Cluster Centers:

Initialize the C cluster centers (centroids) randomly or by selecting random data points as the initial centroids.

These centers represent the central point for each cluster in the feature space.

Step 4: Update Membership Values:

Calculate the degree of membership for each data point in each cluster. The membership values indicate the degree to which each data point belongs to each cluster.

Update the membership matrix using the degree of membership based on the current positions of the centroids and the distances between data points and cluster centers.

Step 5: Update Cluster Centers:

Calculate new cluster centers based on the updated membership values. Each centroid is the weighted average of all data points, where the weights are the membership values raised to a power based on the fuzziness parameter.

Step 6: Check for Convergence:

Check if the algorithm has converged. Convergence is typically determined when:

The change in the membership matrix between iterations is smaller than a predefined threshold (epsilon).

The cluster centers have stabilized (i.e., no significant change in their positions).

Step 7:

If convergence has not been reached, repeat steps 4 through 6. Update the membership matrix and cluster centers iteratively until the algorithm converges or reaches the maximum number of iterations.

Output Results:

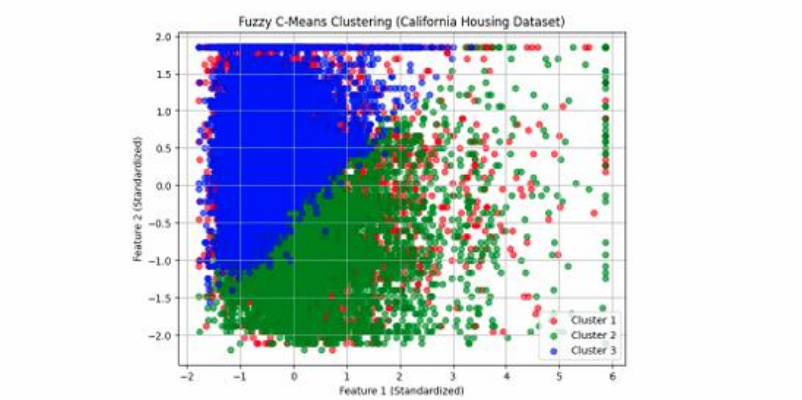
Once the algorithm converges, the final cluster centers and membership matrix are obtained.

Source Code:

```
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler
import skfuzzy as fuzz
import matplotlib.pyplot as plt

# Load California Housing Dataset
data = fetch_california_housing()
X = data.data  # Features

# Normalize the data for better clustering performance
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define the number of clusters
n_clusters = 3

# Apply Fuzzy C-Means algorithm
cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
    X_scaled.T,  # Transpose to shape (features, samples)
    c=n_clusters,  # Number of clusters
    m=2.0,         # Fuzziness parameter
    error=0.005,   # Stopping criterion
    maxiter=1000,  # Maximum number of iterations
```

```
    init=None      # Initial guess for membership values
)

# Assign each data point to the cluster with the highest
membership value
cluster_membership = np.argmax(u, axis=0)

# Visualize the clustering result (using first two features
for simplicity)
plt.figure(figsize=(8, 6))
colors = ['r', 'g', 'b']
for i in range(n_clusters):
    plt.scatter(
        X_scaled[cluster_membership == i, 0],
        X_scaled[cluster_membership == i, 1],
        label=f'Cluster {i+1}',
        color=colors[i],
        alpha=0.6
    )
plt.title('Fuzzy C-Means Clustering (California Housing
Dataset)')
plt.xlabel('Feature 1 (Standardized)')
plt.ylabel('Feature 2 (Standardized)')
plt.legend()
plt.grid()
plt.show()

# Print final cluster centers
print("Cluster Centers:")
print(cntr)
```

Output:




```
Cluster Centers:
[[-0.08712011  0.10727301  0.01305073 -0.02471359 -0.17622414 -0.02098736
   1.02632233 -1.03479205]
 [ 0.31929463 -0.5382695   0.13526595  0.00454979  0.23990346  0.00171917
  -0.50671729  0.51966092]
 [-0.21952402  0.38799397 -0.19338494 -0.04828853 -0.10182158  0.00881839
  -0.6281649   0.60604465]]
```

Discussion:

Fuzzy C-Means (FCM) is a robust clustering algorithm that allows for soft cluster assignments, meaning each data point can belong to multiple clusters with varying degrees of membership. Unlike traditional hard clustering methods like K-means, where each point is assigned to a single cluster, FCM provides a more flexible approach, making it particularly useful for datasets with overlapping or ambiguous cluster boundaries. By iteratively updating the membership values and cluster centers, FCM minimizes an objective function that reflects both the distances between data points and cluster centers and the degree of membership.

Problem Statement:

Apply perceptron for realization of logic gates. (bias = 1)

Procedure:

Step 0: Initialize weights and biases to random number between 0 and 1 for all

layers.

Step 1: Forward Pass (Propagation)

For each input neuron X_i calculate the weighted sum for the hidden layer neuron z_i :

$$z_i = \sum Xi * Wi + \theta$$

Where Wi is the corresponding weight and θ is the baising factor

For z_i output y_i will be:

$$\sigma(zi) = \frac{1}{1+e^{-z_i}}$$

Step 2: Calculate the Error (Loss)

Compute the error between the predicted output and the actual target label using the Mean Squared Error (MSE):

$$L = \frac{1}{2}\sum (yi - ai)^2$$

Step 4: Update Weights and Biases

Update the weights and biases using the Gradient Descent rule:

Update weights for each layer:

$$w_i = learning\_rate * \ \ change\ of\ w$$

Update biases for each layer:

$$b_i = leanring_rate * Erroofthenode$$

Step 5: Repeat the Process

Repeat the process for several iterations (epochs), updating weights and biases until the error converges to a minimal value, or until a predefined number of epochs is reached.

Source code:

```
import numpy as np


w = np.array([0,0,0,0])
eta = int(1)
theta = int(0)


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


target = [1,-1,-1,-1]


print("Inputs", inputs)
print("Target", target)
print("Weight", w)
flag = True
epoch = int(1)
while(flag):
    print("Epoch",epoch)
    epoch +=1
    flag = False
    for i in range(inputs.shape[0]):
        print("For input ",i)
        net_input = sum(inputs[i,:]*w)
        print("net_input",net_input)
        if(net_input > theta):
```

```
            y_out = 1
        elif(net_input>= -theta and net_input <= theta):
            y_out = 0
        else:
            y_out = -1
        print("Y_out", y_out)
        if(y_out != target[i]):
            flag = True
            w[:]+=eta*target[i]*inputs[i,:]
            print("Updated w", w)
```

Output:

Inputs [[1 1 1 1]

[1 -1 1 1]

[1 1 -1 1]

[1 1 1 -1]]

Target [1, -1, -1, -1]

Weight [0 0 0 0]

Epoch 1

For input 0

net_input 0

Y_out 0

Updated w [1 1 1 1]

For input 1

net_input 2

Y_out 1

Updated w [0 2 0 0]

For input 2

net_input 2

Y_out 1

Updated w [-1 1 1 -1]

For input 3

net_input 2

Y_out 1

Updated w [-2 0 0 0]

Epoch 2

For input 0

net_input -2

Y_out -1

Updated w [-1 1 1 1]

For input 1

net_input 0

Y_out 0

Updated w [-2 2 0 0]

For input 2

net_input 0

Y_out 0

Updated w [-3 1 1 -1]

For input 3

net_input 0

Y_out 0

Updated w [-4 0 0 0]

Epoch 3

For input 0

net_input -4

Y_out -1

Updated w [-3 1 1 1]

For input 1

net_input -2

Y_out -1

For input 2

net_input -2

Y_out -1

For input 3

net_input -2

Y_out -1

Epoch 4

For input 0

net_input 0

Y_out 0

Updated w [-2 2 2 2]

For input 1

net_input 0

Y_out 0

Updated w [-3 3 1 1]

For input 2

net_input 0

Y_out 0

Updated w [-4 2 2 0]

For input 3

net_input 0

Y_out 0

Updated w [-5 1 1 1]

Epoch 5

For input 0

net_input -2

Y_out -1

Updated w [-4 2 2 2]

For input 1

net_input -2

Y_out -1

For input 2

net_input -2

Y_out -1

For input 3

net_input -2

Y_out -1

Epoch 6

For input 0

net_input 2

Y_out 1

For input 1

net_input -2

Y_out -1

For input 2

net_input -2

Y_out -1

For input 3

net_input -2

Y_out -1

Discussion:

This code implements a simple Perceptron algorithm for binary classification using a sign activation function. The weights are initially set to zero, and the learning rate is 1. The input vectors, which include a bias term, are processed in a loop for multiple epochs. The training continues until no more weight updates are needed, indicating convergence.

Problem Statement:

Apply Madeline algorithm for Bipolar XOR gates. (Weights are randomly initialized, v0, v1, v2 = 0.5, bias values are set to 1, learning rate = 0.5, Network topology: 2-2-1).

Data Description:

Here Dataset is Bipolar XOR gate with 4 rows as the input of the network.

Procedure:

Step 1: Initialize V0, V, V1 with 0.5 and other weights w0, w1, w2, w01, w02 by small random values. All bias inputs are set to 1.

Step 2: Set the learning rate h to a suitable value.

Step 3: For each bipolar training pair (s : t), do Steps 4-6.

Step 4: Activate the input units: x1 = s1, x2 = s2, all biases are set to 1 permanently.

Step 5:

Propagate the input signals through the net to the output unit Y.

5.1 Compute net inputs to the hidden units.

$z\_in1 = 1 \times w0 + x1 \times w1 + x2 \times w2$

$z\_in2 = 1 \times w01 + x1 \times w02 + x2 \times w02$

5.2 Compute activations of the hidden units z_out using the bipolar step function

If $z\_in \geq 0$, then $z\_out = 1$

If $z\_in < 0$, then $z\_out = -1$

5.3 Compute net input to the output unit

$y\_in = 1 \times v0 + z\_out1 \times v1 + z\_out2 \times v2$

5.4 Find the activation of the output unit y_out using the same activation function as in Step 5.2

If $y\_in \geq 0$, then $y\_out = 1$

If $y\_in < 0$, then $y\_out = -1$

Step 6:

Adjust the weights of the hidden units, if required, according to the following rules:

(i) If y_out = t, then the net yields the expected result. Weights need not be updated.

(ii) If $y\_out \neq t$, then apply one of the following rules whichever is applicable.

Case I: t = 1

Find the hidden unit z whose net input z_in is closest to 0. Adjust the weights attached to z according to the formula:

$w\_j(new) = w\_j(old) + h \times (1 - z\_in) \times x\_j$, for all j

Case II: t = -1

Adjust the weights attached to those hidden units z that have positive net input.

$w\_j(new) = w\_j(old) + h \times (-1 - z\_in) \times x\_j$, for all j

Step 7:

Test for stopping condition. It can be any one of the following:

(i) No change of weight occurs in Step 6.

(ii) The weight adjustments have reached an acceptable level.

(iii) A predefined number of iterations have been carried out.

If the stopping condition is satisfied, then stop. Otherwise, go to Step 3.

Source Code:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

"""Madaline procedure"""

class Madaline:
    # All methods of the Madaline algorithm
    def __init__(self,
                 dataset: pd.DataFrame,
                 input_layer_size: int,
                 hidden_layer_size: int,
                 output_layer_size: int,
                 v0: float = 0.5,
                 v1: float = 0.5,
                 v2: float = 0.5,
                 bias: list = [1, 1, 1],
                 learning_rate: float = 0.5):
            """
            Initialize the Madaline network parameters.

            Args:
                dataset: Input dataset in the form of a Pandas
DataFrame.
                input_layer_size: Size of the input layer.
                hidden_layer_size: Size of the hidden layer.
                output_layer_size: Size of the output layer.
                v0, v1, v2: Bias values for each layer.
                bias: List of bias terms for each hidden unit.
                learning_rate: Learning rate for weight updates.
```

```
            """
            self.v0 = v0
            self.v1 = v1
            self.v2 = v2
            self.bias = bias
            self.dataset = dataset
            self.learning_rate = learning_rate

            # Ensure learning rate is within the correct range
            if learning_rate > 1:
                    print('Learning rate exceeds 1')
                    exit()

            self.stop = 0

            # Initialize weights randomly between 0 and 1
            self.input_weights = np.random.uniform(0, 1,
(input_layer_size, hidden_layer_size))
    def train(self):
            """
            Train the Madaline network on the dataset.
            """
            epoch = 0
            while not self.stop:
                    print('Epoch:', epoch)
                    epoch += 1
                    print('Input weights:', self.input_weights)
                    self.y = []  # Array to store the output for each
data point

                    # Iterate through each data point in the dataset
                    for i in range(len(self.dataset)):
                            print('iteration:', i,'----------------
----------')
```

```
                            # Step 5.1: Calculate net input to the
hidden units
                            self.z_in =
np.matmul(self.dataset.iloc[i, :-1].to_numpy(), self.input_weights)
                            print("Net input to hidden units
(z_in):", self.z_in)


                            # Step 5.2: Apply the bipolar step
function to get hidden layer activations
                            z_out =
self.__sign_activation_function(self.z_in)
                            print("Hidden layer activations
(z_out):", z_out)


                            # Step 5.3: Compute net input to the
output unit
                            y_in = self.v0 + np.dot(z_out,
[self.v1,self.v2])
                            print("Net input to output unit (y_in):",
y_in)


                            # Step 5.4: Apply activation function to
output unit
                            self.y_out =
self.__sign_activation_function(y_in)
                            print("Output unit activation (y_out):",
self.y_out)


                            # Store the output of the current data
point
                            self.y.append(self.y_out)


                            # Step 6: Update weights if the output is
incorrect
                            if self.y_out != self.dataset.iloc[i, -
1]:  # Check if output matches the expected output
                                print("Output not matched!!")
                                self.__update_weight_matrix(i)
```

```
                # Check for convergence (if all outputs match target
outputs)
                    self.stop = self.__stop_function()


    def __sign_activation_function(self, x):
            """
            Bipolar step activation function.


            Args:
                x: Input array.


            Returns:
                Array with elements set to 1 if >= 0, else -1.
            """
            return np.where(x >= 0, 1, -1)


    def __update_weight_matrix(self, i):
            """
            Update the weight matrix based on error correction.


            Args:
                i: Index of the current data point.
            """
        # Case I: Desired output (target) is +1
            if self.dataset.iloc[i, -1] == 1:
                    # Find hidden unit closest to zero net input and
adjust weights
                    j = np.argmin(np.abs(self.z_in))
                    # Update weight for the chosen hidden unit
                    self.input_weights[:,j] = self.input_weights[:,j]
+ \
                                              self.learning_rate *
(self.bias[j] - self.z_in[j]) * \
                                              self.dataset.iloc[i, :-
1].to_numpy()
```

```
                    print('Updated weights are: ',self.input_weights)
            else:
                # Case II: Desired output is -1
                    indices = np.where(self.z_in >= 0)[0]  # Find
hidden units with positive net input
                    for j in indices:
                        # Update weights for each of those units
                            self.input_weights[:,j] =
self.input_weights[:,j] + \

self.learning_rate * (-self.bias[j] - self.z_in[j]) * \

self.dataset.iloc[i, :-1].to_numpy()
                            print('Updated weights are:
',self.input_weights)


    def __stop_function(self):
            """
            Check if the network has converged by comparing all
outputs to targets.


            Returns:
                1 if all outputs match targets, 0 otherwise.
            """
            for i in range(len(self.dataset)):
                if self.y[i] != self.dataset.iloc[i, -1]:  # If any
output is incorrect, do not stop
                    return 0
            return 1  # Stop if all outputs are correct


# Main execution
if __name__ == '__main__':
          np.random.seed(7)
          dataset = pd.read_csv('Bipolar XOR.csv')
          train_data, test_data = train_test_split(dataset,
test_size=0.2,random_state=40)
```

```
            print("Training Data:")

            print(train_data)

            print("\nTesting Data:")

            print(test_data)

            mad =
      Madaline(train_data,input_layer_size=3,hidden_layer_size=2,output_lay
      er_size=1)

            mad.train()
```

Output:

```
Training Data:


  x0  x1  x2  t
0  1   1   1 -1
1  1   1  -1  1
2  1  -1   1  1

Testing Data:
  x0  x1  x2  t
3  1  -1  -1 -1
Epoch: 0
Input weights: [[0.07630829 0.77991879]
 [0.43840923 0.72346518]
 [0.97798951 0.53849587]]
iteration: 0 ----------------------------
Net input to hidden units (z_in): [1.49270703 2.04187984]
Hidden layer activations (z_out): [1 1]
Net input to output unit (y_in): 1.5
Output unit activation (y_out): 1
Output not matched!!
Updated weights are:  [[-1.17004523  0.77991879]
 [-0.80794428  0.72346518]
 [-0.268364    0.53849587]]
Updated weights are:  [[-1.17004523 -0.74102113]
 [-0.80794428 -0.79747474]
 [-0.268364   -0.98244405]]
iteration: 1 ----------------------------
Net input to hidden units (z_in): [-1.70962551 -0.55605182]
Hidden layer activations (z_out): [-1 -1]
Net input to output unit (y_in): -0.5
Output unit activation (y_out): -1
Output not matched!!
Updated weights are:  [[-1.17004523  0.03700478]
 [-0.80794428 -0.01944883]
 [-0.268364   -1.76046996]]
iteration: 2 ----------------------------
Net input to hidden units (z_in): [-0.63046495 -1.70401635]
Hidden layer activations (z_out): [-1 -1]
Net input to output unit (y_in): -0.5
Output unit activation (y_out): -1
```

```
Output not matched!!
Updated weights are:  [[-0.35481275  0.03700478]
 [-1.62317676 -0.01944883]
 [ 0.54686847 -1.76046996]]
Epoch: 1
Input weights: [[-0.35481275  0.03700478]
 [-1.62317676 -0.01944883]
 [ 0.54686847 -1.76046996]]
iteration: 0 ----------------------------
Net input to hidden units (z_in): [-1.43112104 -1.74291401]
Hidden layer activations (z_out): [-1 -1]
Net input to output unit (y_in): -0.5
Output unit activation (y_out): -1
iteration: 1 ----------------------------
Net input to hidden units (z_in): [-2.52485798  1.77802591]
Hidden layer activations (z_out): [-1  1]
Net input to output unit (y_in): 0.5
Output unit activation (y_out): 1
iteration: 2 ----------------------------
Net input to hidden units (z_in): [ 1.81523247 -1.70401635]
Hidden layer activations (z_out): [ 1 -1]
Net input to output unit (y_in): 0.5
Output unit activation (y_out): 1
```

Discussion:

This code implements a Madaline (Multiple Adaptive Linear Element) neural network that is trained to learn from a dataset. The Madaline network uses a simple architecture consisting of an input layer, hidden layer, and output layer. The training process involves iterating through the data points, calculating the net input to the hidden and output layers, applying activation functions (bipolar step function), and updating the weights based on errors using the Perceptron learning rule.