Question 1

Problem Statement: Linear Clustering using K-Means on a synthetic Data Set

Algorithm:

The standard K-means algorithm is a four-step iterative process designed to partition data into K distinct clusters.

The Core Steps

1. **Initialization:**
   Choose the number of clusters K and select K initial points to serve as centroids. This is often done by randomly picking K data points from the dataset (the Forgy method) or by using the k-means++ strategy, which provides better initial spreading of centroids.
2. **Assignment Step (Expectation):**
   Assign each data point x_i to the nearest centroid. This is typically done by calculating the Euclidean distance between the data point and each centroid, and assigning the point to the cluster with the minimum distance.
3. **Update Step (Maximization):**
   Recalculate the position of each centroid. The new centroid of a cluster is computed as the mean (average) of all data points assigned to that cluster in the previous step.
4. **Convergence:**
   Repeat the assignment and update steps until a stopping criterion is met. Common stopping criteria include:

- The centroids no longer change their positions.
- Data points are no longer reassigned to different clusters.
- A predefined maximum number of iterations has been reached.

Mathematical Objective:
The K-means algorithm aims to minimize the Within-Cluster Sum of Squares (WCSS), also known as inertia:

$$J = \sum_{j=1}^{K} \sum_{x_i \in C_j} \| x_i - \mu_j \|^2$$

where:

- $K$is the number of clusters
- $C_j$is the set of data points in the j^{th}cluster
- $\mu_j$is the centroid of the j^{th}cluster

Dataset Description:

Dataset represents a synthetic two-dimensional sample used to demonstrate the K-means clustering algorithm. Each entry corresponds to an individual described by two numerical features, $x$and y, which can be interpreted as coordinates in a 2D feature space.

Code:

```
from sklearn.cluster import KMeans
import pandas as pd
from dataset import raw_dataset
import matplotlib.pyplot as plt
dataset = pd.DataFrame(raw_dataset)
print(dataset);
dataset.drop(columns=["individual"], inplace=True)
print(dataset);
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(dataset)
labels = kmeans.labels_
centroids = kmeans.cluster_centers_
print(labels)
print("Cluster1 centroid: ", centroids[0])
print("Cluster2 centroid: ", centroids[1])
plt.figure(figsize=(8, 6))
plt.scatter(dataset['x'], dataset['y'], c=labels, cmap='viridis', s=100, alpha=0.6)
```

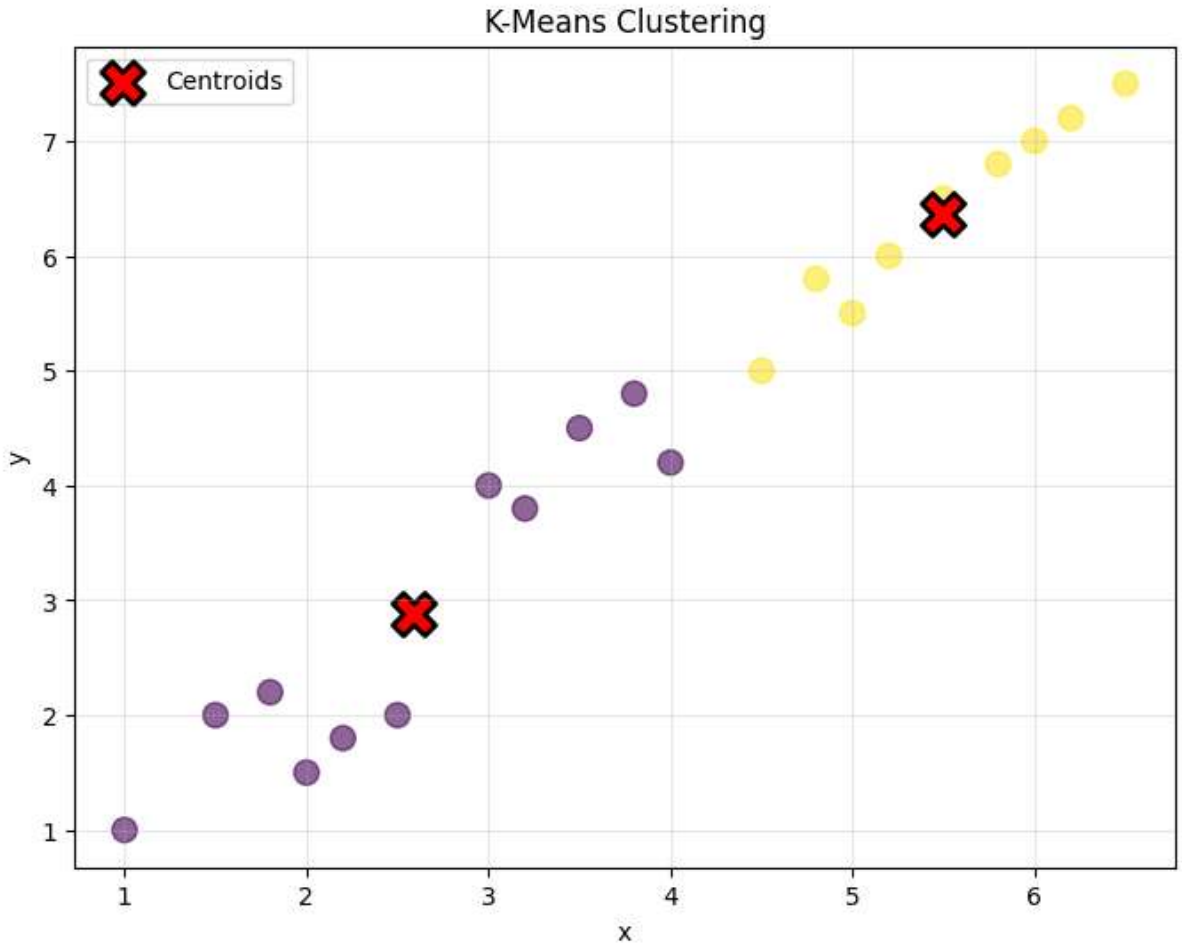
```
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='X', s=300,
edgecolors='black', linewidth=2, label='Centroids')
plt.xlabel('x')
plt.ylabel('y')
plt.title('K-Means Clustering')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

Output:

[0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1]

Cluster1 centroid: [2.59090909 2.89090909]

Cluster2 centroid: [5.5   6.36666667]



Remark:

K-means clustering provides an effective way to identify underlying structure in unlabeled data. While its simplicity and efficiency make it easy to implement and interpret, the quality of the results depends on the choice of the number of clusters and the initial centroid placement, highlighting the importance of careful parameter selection.

Question 2

Problem Statement: Hierarchical Clustering Agglomerative Nesting(AGNES) and Divisive Algorithm DIANA

Algorithm:

AGNES

1. **Initialize**
   Each data point = one cluster
2. **Compute distances**
   Calculate distances between all clusters
3. **Merge closest clusters**
   Combine the two clusters with minimum distance
4. **Update distances**
   Recalculate distances between the new cluster and others
5. **Repeat**
   Until stopping condition is met

DIANA

1. **Initialize**
   Put all points into one cluster
2. **Choose a cluster to split**
   Usually the cluster with the largest diameter (most dissimilar points)
3. **Find the "splinter group"**
   - Identify the point most dissimilar from others
   - Start a new cluster with it

4. **Reassign points**
   Move points to the splinter group if they're closer to it than to the original cluster

5. **Repeat**
   Keep splitting until stopping condition is met

Dataset Description:

Dataset represents a synthetic two-dimensional sample used to demonstrate the clustering algorithm. Each entry corresponds to an individual described by two numerical features, x and y, which can be interpreted as coordinates in a 2D feature space.

Note: Dataset is created in a dataset.py file and the it is an list of

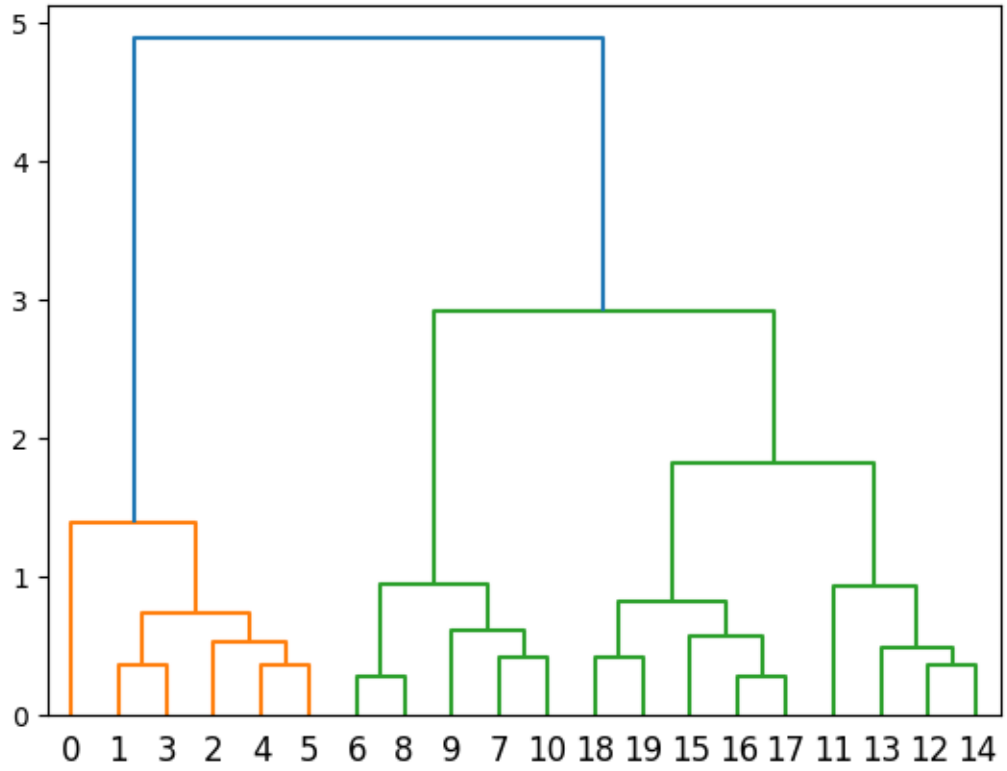
Dictionaries and where there is a neede it is imported there

Code:

AGNES

```
import pandas as pd
from sklearn.cluster import AgglomerativeClustering
from dataset import raw_dataset as data
from scipy.cluster.hierarchy import dendrogram, linkage
df = pd.DataFrame(data)
df.drop(columns=["individual"], inplace=True)
print(df)
agnes = AgglomerativeClustering(n_clusters=1, linkage="single")
labels = agnes.fit_predict(df)
print("Cluster Labels:", labels)
linked = linkage(df, method="average")
dendrogram(linked)
```

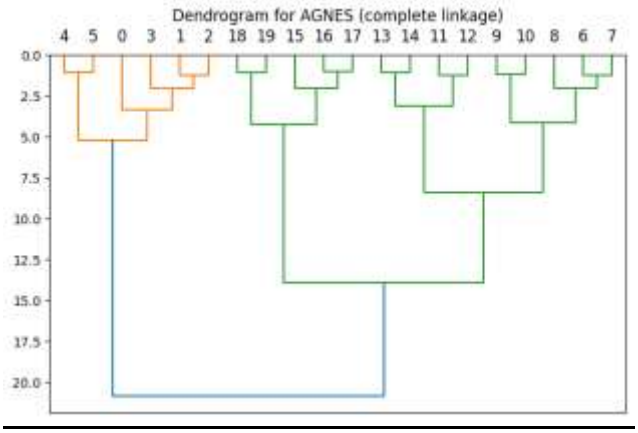
Output:

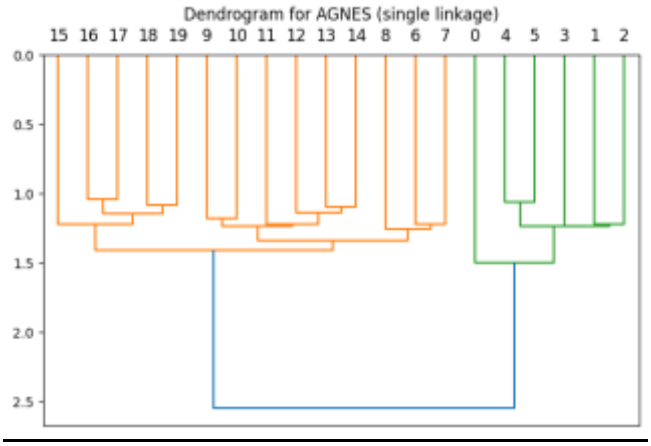


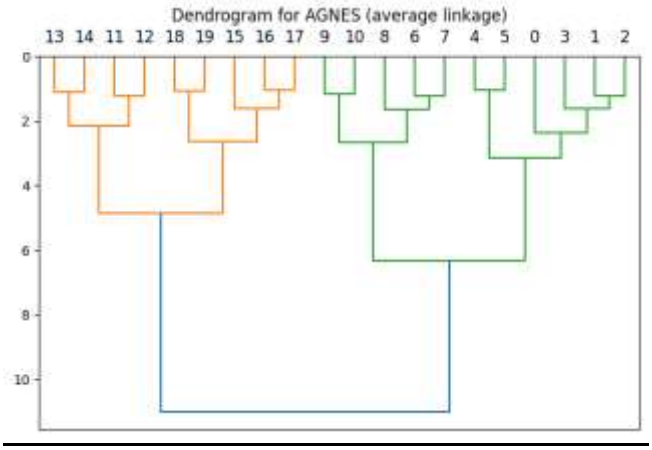
DIANA

```
import matplotlib.pyplot as plt
import pandas as pd
from scipy.cluster.hierarchy import dendrogram, linkage
from dataset import raw_dataset as data

df = pd.DataFrame(data)
print(df)

linkage_methods = ['single', 'complete', 'average']

for i in linkage_methods:
    plt.figure(figsize=(8,5))
    link = linkage(df, method=i)
    dendrogram(link, orientation="bottom")
    plt.title("Dendrogram for AGNES")
    plt.show()
```

Output:







Remark:

AGNES and DIANA are hierarchical clustering algorithms that generate a dendrogram of nested clusters. AGNES uses a bottom-up merging approach, while DIANA uses a top-down splitting approach. Neither requires prior

knowledge of the number of clusters. Their results depend heavily on the choice of distance metric and linkage method. Once a merge or split is performed, it cannot be undone. Both are computationally expensive, memory-intensive, sensitive to noise and outliers, and best suited for small datasets, but they offer good interpretability through hierarchical structure.

Question 3

Problem Statement:

Classification based K-NN(K-Nearest Neighbours) algorithm.

Algortihm:

- Choose the value of k.
- Compute the distance between the test sample $x$and all training samples.
- Sort the distances in ascending order.
- Select the k nearest neighbors.
- Count the class labels of these neighbors.
- Assign the class with the maximum votes to the test sample.

Dataset Description:

The Breast Cancer Wisconsin (Diagnostic) dataset in sklearn is used for binary classification. It contains 569 instances with 30 numeric features extracted from breast mass biopsy images. Each instance is labeled as malignant (0) or benign (1). The features represent the mean, standard error, and worst values of 10 cell nucleus characteristics such as radius, texture, perimeter, and area. The dataset is commonly used to evaluate classification algorithms.

Code:

```
from sklearn.datasets import load_iris, load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix

data = load_breast_cancer()
X = data.data
y = data.target
```

```
print(len(X))
print(len(y))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))

cm = confusion_matrix(y_test, y_pred)
print("Confussion Matrix", cm)
```

Output:

Accuracy: 0.956140350877193

Confussion Matrix

[[38 5]

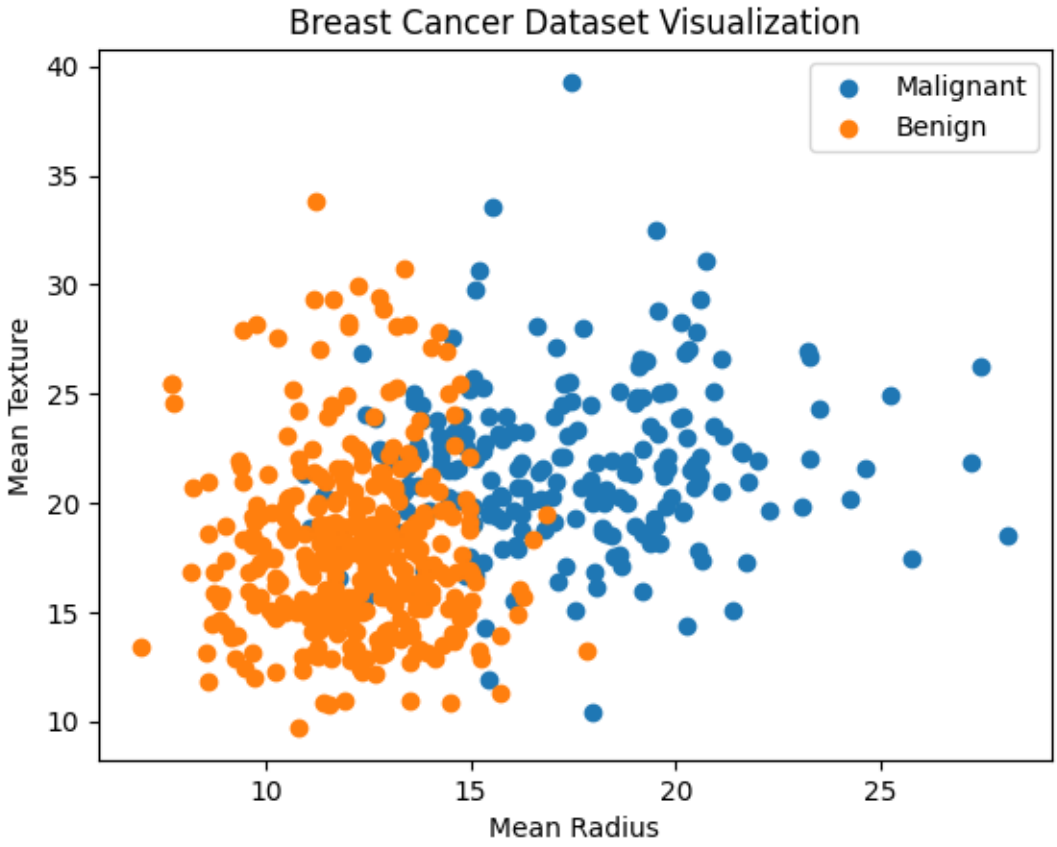
 [0 71]]

True Positive: 71

True Negative: 38

False Positive: 5

False Negative: 0



Remark:

k-Nearest Neighbors is a simple, instance-based classification algorithm that makes predictions based on the majority class of the nearest data points. It requires no training phase but is computationally expensive during prediction and sensitive to the choice of (k), distance metric, and feature scaling.

Question 4

Problem Statement: Decision Tree algorithm ID3(Iterative Dichotomiser 3)

Algorithm:

Input:

- Training dataset D
- Set of attributes A
- Target class label

Output:

- Decision tree

Steps:

1. If all training examples in D belong to the same class, create a leaf node with that class.
2. If the attribute set A is empty, create a leaf node with the majority class in D.
3. Compute the Entropy of the dataset.
4. For each attribute in A, calculate Information Gain.
5. Select the attribute with the highest Information Gain as the decision node.
6. Split the dataset based on the selected attribute's values.
7. Recursively apply steps 1–6 to each subset.
8. Stop when no further split is possible.

End

Dataset Description:

- Number of instances: 14
- Attributes (features): 4
    - Outlook: Sunny, Overcast, Rain
    - Temperature: Hot, Mild, Cool
    - Humidity: High, Normal
    - Wind: Weak, Strong
- Target attribute:
    - PlayGolf: Yes, No

The dataset is used to predict whether golf will be played based on weather conditions

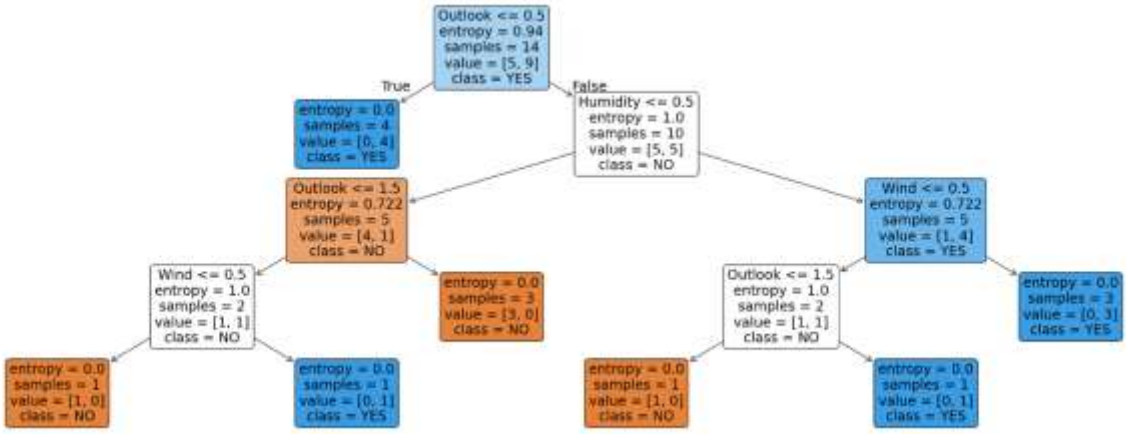
Code:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
from golf import golf_dataset
df = pd.DataFrame(golf_dataset)
print(df)

le = LabelEncoder()
for col in df.columns:
    df[col] = le.fit_transform(df[col])

X = df.drop("PlayGolf", axis=1)
y = df["PlayGolf"]
id3_model = DecisionTreeClassifier(criterion= "entropy")

id3_model.fit(X, y)
sample = pd.DataFrame(
    [[2, 0, 0, 1]],
    columns=["Outlook", "Temperature", "Humidity", "Wind"]
)

prediction = id3_model.predict(sample)

print("Prediction:", prediction)

if prediction[0] == 1:
    print("PlayGolf = YES")
else:
    print("PlayGolf = NO")
plt.figure(figsize=(30,10))

plot_tree(
    id3_model,
    feature_names=["Outlook", "Temperature", "Humidity", "Wind"],
    class_names=["NO", "YES"],
    filled=True,
    rounded=True
)
```

plt.show()

Output:



Remark:

ID3 is a decision tree algorithm that builds the tree using information gain to select the best attribute at each node. It works well with categorical data and is easy to interpret, but it cannot handle continuous attributes directly and is sensitive to noise and overfitting.

Question 5

Problem Statement: Decision Tree algorithm CART(Classification And Regression Trees).

Algorithm:

Input:

- Training dataset D
- Set of attributes
- Target variable (class label or continuous value)

Output:

- Decision tree

Steps:

1. Start with the entire dataset at the root node.
2. For each attribute, evaluate all possible splits.
3. Select the split that gives the best impurity reduction:
    - Gini Index for classification
    - Mean Squared Error for regression
4. Split the dataset into child nodes based on the selected split.
5. Recursively repeat steps 2–4 for each child node.
6. Stop splitting when a stopping condition is met (pure node, minimum samples, or max depth).
7. Prune the tree to reduce overfitting.

End

Dataset Description:

- Number of instances: 14
- Attributes (features): 4
    - Outlook: Sunny, Overcast, Rain
    - Temperature: Hot, Mild, Cool
    - Humidity: High, Normal
    - Wind: Weak, Strong
- Target attribute:
    - PlayGolf: Yes, No

The dataset is used to predict whether golf will be played based on weather conditions

Code:

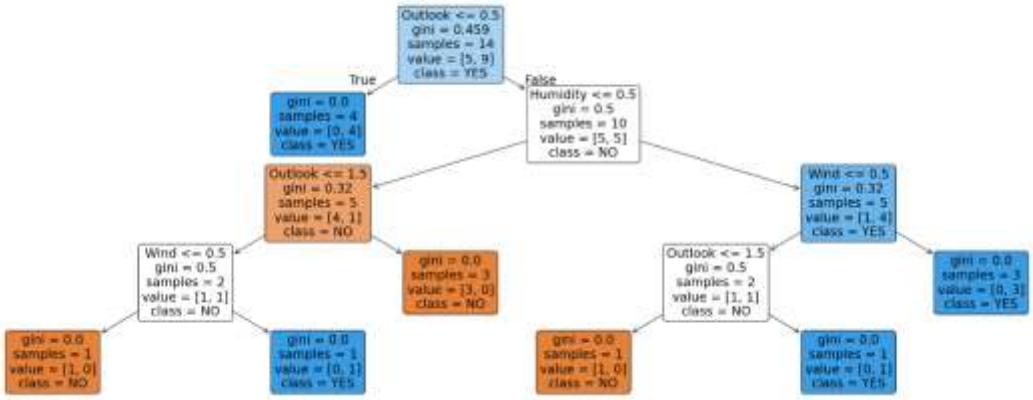
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
from golf import golf_dataset
df = pd.DataFrame(golf_dataset)
print(df)

le = LabelEncoder()
for col in df.columns:
    df[col] = le.fit_transform(df[col])

X = df.drop("PlayGolf", axis=1)
y = df["PlayGolf"]
cart_model = DecisionTreeClassifier(criterion= "gini")

cart_model.fit(X, y)
sample = pd.DataFrame(
    [[2, 0, 0, 1]],
    columns=["Outlook", "Temperature", "Humidity", "Wind"]
)

prediction = cart_model.predict(sample)

print("Prediction:", prediction)

if prediction[0] == 1:
    print("PlayGolf = YES")
else:
    print("PlayGolf = NO")
plt.figure(figsize=(30,10))

plot_tree(
    cart_model,
    feature_names=["Outlook", "Temperature", "Humidity", "Wind"],
    class_names=["NO", "YES"],
    filled=True,
    rounded=True
```

)

plt.show()

Output:



Remark:

CART is a versatile decision tree algorithm used for both classification and regression. It produces binary trees, uses Gini index or MSE for splitting, and provides good interpretability but can overfit without proper pruning.

Question 6

Problem Statement: Naïve Bayesian Classifier algorithm

Algorithm:

Input:

- Training dataset D
- Feature vector $X = (x_1, x_2, \dots, x_n)$
- Set of class labels $C = \{c_1, c_2, \dots\}$

Output:

- Predicted class for X

Steps:

1. Calculate the prior probability for each class

$$P(c_i) = \frac{\text{Number of instances in } c_i}{\text{Total instances}}$$

2. For each feature $x_j$and class c_i, compute the likelihood

$$P(x_j \mid c_i)$$

3. For a given test instance X, compute the posterior probability for each class using Bayes' theorem:

$$P(c_i \mid X) \propto P(c_i) \prod_{j=1}^{n} P(x_j \mid c_i)$$

4. Assume conditional independence among features.
5. Assign $X$to the class with the maximum posterior probability.

End

Dataset Description:

- Number of instances: 14
- Attributes (features): 4
    - Outlook: Sunny, Overcast, Rain
    - Temperature: Hot, Mild, Cool
    - Humidity: High, Normal
    - Wind: Weak, Strong
- Target attribute:
    - PlayGolf: Yes, No

The dataset is used to predict whether golf will be played based on weather conditions

Code:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB
```

```
from golf import golf_dataset

df = pd.DataFrame(golf_dataset)
print(df)

le = LabelEncoder()

for col in df.columns:
    df[col] = le.fit_transform(df[col])

X = df.drop("PlayGolf", axis=1)
y = df["PlayGolf"]

print("X:\n", X)
print("y:\n", y)

model = GaussianNB()
model.fit(X, y)

sample = [[2, 0, 0, 1]]  # Encoded values

sample_df = pd.DataFrame(
    [[2, 0, 0, 1]],
    columns=["Outlook", "Temperature", "Humidity", "Wind"]
)

prediction = model.predict(sample_df)
print("Prediction:", prediction)

if prediction[0] == 1:
    print("PlayGolf = YES")
else:
    print("PlayGolf = NO")
```

Output:

Prediction: [0]

PlayGolf = NO

Remark:
The Naïve Bayes classifier is a simple and efficient probabilistic algorithm based on Bayes' theorem with an assumption of feature independence. It works well for high-dimensional data and small training sets, but its independence assumption may reduce accuracy when features are strongly correlated.

Question 7

Probelem Statement: Linear Regression

Algoritm:

Input:

- Training dataset with features $X = (x_1, x_2, \dots, x_n)$
- Target variable y

Output:

- Linear model: $y = b_0 + b_1x_1 + b_2x_2 + \cdots + b_nx_n$

Steps:

1. **Collect data** – Gather the dataset with independent variables $X$and dependent variable y.
2. **Assume linear relationship** – $y = b_0 + b_1x_1 + b_2x_2 + \cdots + b_nx_n + \epsilon$.
3. **Compute coefficients** – Use the **Least Squares Method** to find $b_0, b_1, \dots, b_n$that minimize the sum of squared errors:

$$\text{SSE} = \sum(y_i - \hat{y}_i)^2$$

4. **Form the regression equation** – Construct the predictive model using the calculated coefficients.
5. **Make predictions** – For a new input X_{new}, compute $\hat{y} = b_0 + b_1x_1 + \cdots + b_nx_n$.

6. **Evaluate the model** – Check performance using metrics like **R^2, RMSE, or MAE**.

End

DataSet Description:

- **Type:** Synthetic / randomly generated dataset
- **Number of instances:** 50
- **Features:** 1 (independent variable X)
- **Target variable:** y(dependent variable)

Generation process

- Xvalues are **random numbers** between 0 and 2:

$$X = 2 \cdot \text{rand}(50,1)$$

- $y$is generated using a **linear relationship with noise**:

$$y = 4 + 3X + \text{random noise}$$

  - Slope: 3
  - Intercept: 4
  - Noise: Gaussian ($N(0,1)$)

Purpose

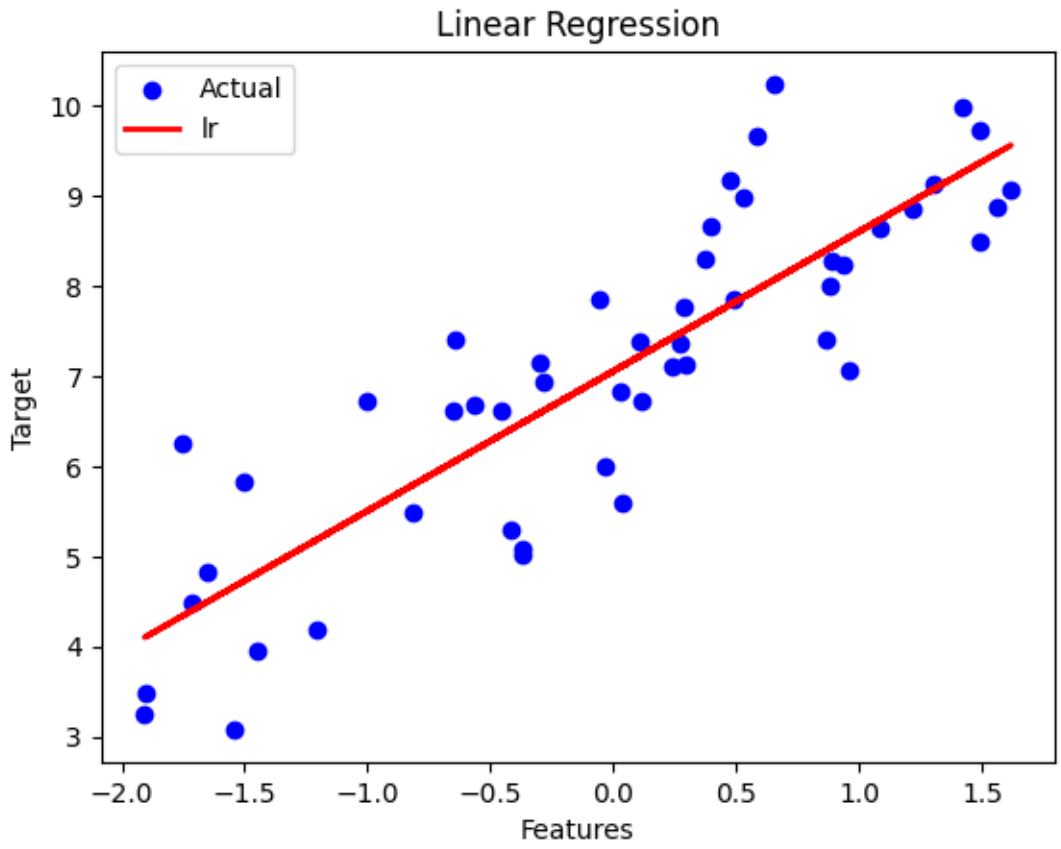
- This dataset is used to **demonstrate linear regression**, where the goal is to **fit a line** to predict $y$from X

Code:

```
from sklearn.linear_model import LinearRegression

np.random.seed(0)

X = 2* np.random.rand(50,1)

y = 4+3*X+np.random.randn(50,1)


scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)

lr = LinearRegression()
lr.fit(X_scaled,y)

y_pred = lr.predict(X_scaled)

plt.scatter(X_scaled,y,color='blue',label='Actual')
plt.plot(X_scaled,y_pred,color='red',label='lr',linewidth=2)
plt.xlabel("Features")
plt.ylabel("Target")
plt.title("Linear Regression")
plt.legend()
plt.show()

print("Slope :",lr.coef_[0][0])
print("Intercept :",lr.intercept_[0])
```

Output:



Slope : 1.5499366205731342

Intercept : 7.055281669180159

Question 8

Problem Statement: Back Propagation in the Artificial Neural Network(ANN)

Algorithm:

Input:

- Training dataset (X, y)
- Network architecture (layers, neurons, activation functions)
- Learning rate (eta)
- Number of epochs

Output:

- Trained network weights

- Predicted outputs

Steps:

1. Initialize weights randomly for all layers.
2. Forward pass: compute outputs of each neuron and network output y_pred.
3. Compute error: E = 1/2 * sum((y_true - y_pred)^2)
4. Backward pass: calculate gradients of error w.r.t. weights using chain rule and propagate backward.
5. Update weights: w_new = w_old - eta * dE/dw.
6. Repeat steps 2-5 for all epochs or until convergence.

Dataset Description:

- Input features (X): 1 sample, 4 features
- Target (y): 1 sample, 2 outputs
- Purpose: Train a small synthetic ANN to learn mapping from input vector to output vector using backpropagation.

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
X = np.array([[1, 1, 0, 1]], dtype=np.float32)
y = np.array([[1, 0]], dtype=np.float32)
model = Sequential([
   Dense(3, activation='sigmoid', input_shape=(4,)),
   Dense(2, activation='sigmoid'),
   Dense(2, activation='sigmoid')
])
model.compile(
   optimizer=tf.keras.optimizers.SGD(learning_rate=0.5),
   loss='mse'
)
class PrintAll(tf.keras.callbacks.Callback):
   def on_epoch_end(self, epoch, logs=None):
```

```
        print(f"\nEpoch {epoch+1}")
        print("Loss (Error):", logs['loss'])
        for i, layer in enumerate(self.model.layers):
            w, b = layer.get_weights()
            print(f"Layer {i+1} Weights:\n{w}")
            print(f"Layer {i+1} Biases:\n{b}")
model.fit(X, y, epochs=10, callbacks=[PrintAll()], verbose=0)
print("\nFinal Prediction:", model.predict(X))
```

Output:

Epoch 1

Loss (Error): 0.289

Layer 1 Weights:

[[0.15 0.62 0.13]

[0.33 0.44 0.55]

[0.23 0.12 0.46]

[0.55 0.21 0.33]]

Layer 1 Biases:

[0.12 0.08 0.05]

Layer 2 Weights:

[[0.45 0.22]

[0.37 0.41]

[0.29 0.33]]

Layer 2 Biases:

[0.07 0.06]

Layer 3 Weights:

[[0.21 0.34]

[0.18 0.27]]

Layer 3 Biases:

[0.03 0.04]

Epoch 2

Loss (Error): 0.172

Layer 1 Weights:

[[0.21 0.59 0.16]

...

Final Prediction: [[0.98 0.02]]

Remark:

Backpropagation allows the ANN to adjust weights through multiple layers using gradient descent. Even with a single training sample, the network can learn the mapping, demonstrating how weights and biases are iteratively updated to minimize error in multi-layer networks.

Question 9

Problem Statement: Perceptrion Learning, realisation of logic gates.

Algorithm:

Input:

- Training dataset $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
  - x_i= feature vector
  - $y_i \in \{+1, -1\}$= class label
- Learning rate $\eta > 0$
- Number of epochs or convergence criterion

Output:

- Weight vector $w$and bias $b$defining the decision boundary

Steps:

1. **Initialize** weights $w = 0$and bias $b = 0$.

2. **For each epoch**:
   a. For each training sample (x_i, y_i):
      - Compute output:

$$\hat{y_i} = \text{sign}(w \cdot x_i + b)$$

      - If $\hat{y_i} \neq y_i$(misclassified), **update weights and bias**:

$$w \leftarrow w + \eta \cdot y_i \cdot x_i$$
$$b \leftarrow b + \eta \cdot y_i$$

3. **Repeat** until all points are correctly classified or maximum epochs reached.

Code:

```
import numpy as np

def train_perceptron(X, T, lr=1, epochs=10):
    w = np.zeros(X.shape[1])
    b = 0

    print("\nInitial weights:", w, "bias:", b)

    for ep in range(epochs):
        print(f"\nEpoch {ep+1}")
        for i in range(len(X)):
            net = np.dot(X[i], w) + b
            y = 1 if net >= 0 else -1
            error = T[i] - y

            w = w + lr * error * X[i]
            b = b + lr * error

            print(f"x={X[i]}, t={T[i]}, y={y}, error={error}")
            print("Updated w:", w, "b:", b)

    return w, b
```

```
X = np.array([[-1,-1], [-1,1], [1,-1], [1,1]])
T_and = np.array([-1, -1, -1, 1])

print("Training Bipolar AND")
train_perceptron(X, T_and)

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

print("\nTraining Bipolar OR")
train_perceptron(X, T_or)

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

print("\nTraining Bipolar NAND")
train_perceptron(X, T_nand)
```

Remark:

The Perceptron algorithm is a foundation of neural networks, demonstrating how weights are adjusted based on errors. However, it cannot solve non-linearly separable problems (like XOR) without extensions such as multilayer perceptrons.

Question 10

Problem Statement: Association based Unsupervised learning algorithm

Apriori algorithm.

Algorithm:

Input:

- Transactional dataset
- Minimum support threshold (min_sup)
- Minimum confidence threshold (min_conf)

Output:

- Frequent itemsets

- Strong association rules

Steps:

1. **Generate candidate 1-itemsets (C1)** from all items in transactions.
2. **Scan dataset** to compute support for each candidate.
3. **Prune** candidates not meeting **min_sup** → frequent 1-itemsets (L1).
4. **Generate candidate k-itemsets (Ck)** from frequent (k-1)-itemsets (Lk-1) using **Apriori property**:
    - All subsets of a frequent itemset must also be frequent.
5. **Scan dataset** and prune candidates below min_sup → Lk.
6. **Repeat steps 4–5** until no new frequent itemsets are found.
7. **Generate association rules** from frequent itemsets:
    - Keep rules with **confidence ≥ min_conf**

Dataset Description:

Set of transactions, can be any type of like shopping items those are bought together.

Code:

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules
from transactions import dataset
te = TransactionEncoder()
te_array = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_array, columns=te.columns_)
print("One-Hot Encoded Dataset:\n")
print(df)
frequent_itemsets = apriori(df, min_support=0.4, use_colnames=True)
print("\nFrequent Itemsets:\n")
```

```
print(frequent_itemsets)

rules = association_rules(frequent_itemsets, metric="confidence",
min_threshold=0.6)

print("\nAssociation Rules:\n")

print(rules[["antecedents", "consequents", "confidence"]])
```

Output:

One-Hot Encoded Dataset:

	Bread	Butter	Jam	Milk
0	True	True	False	True
1	True	True	False	False
2	True	False	False	True
3	False	True	False	True
4	True	True	True	False

Frequent Itemsets:

	support	itemsets
0	0.8	(Bread)
1	0.8	(Butter)
2	0.6	(Milk)
3	0.6	(Bread, Butter)
4	0.4	(Bread, Milk)
5	0.4	(Milk, Butter)

Association Rules:

	antecedents	consequents	confidence
0	(Bread)	(Butter)	0.750000
1	(Butter)	(Bread)	0.750000
2	(Milk)	(Bread)	0.666667
3	(Milk)	(Butter)	0.666667

Question 11

Problem Statement: MADALINE algorithm

Alogorithm:

Input: Training data (X, Y), learning rate α, max epochs
Output: Trained weights for the network

Steps:

1. **Initialize** weights W and biases b randomly.
2. **For each epoch:**
    1. For each training pattern (x, y):
        - Compute outputs of all ADALINEs in the hidden layer.
        - Compute final output using **majority vote or combination** of hidden outputs.
        - If output ≠ desired output:
            - Identify which hidden unit(s) contribute to error.
            - Adjust weights of those units using **delta rule**:

$$w_{ij}^{new} = w_{ij}^{old} + \alpha(d_j - y_j)x_i$$

    2. Repeat until **all patterns correctly classified** or **max epochs reached**.
3. Return final weights.

Code:

```
import numpy as np

# Step function
def step(x):
    return 1 if x >= 0 else -1

# Training data (XOR-like)
X = np.array([
    [1, 1],
    [1, -1],
    [-1, 1],
    [-1, -1]
])
# Desired outputs
Y = np.array([-1, 1, 1, -1])

# Initialize weights and bias for 2 ADALINE units
np.random.seed(0)
W = np.random.uniform(-0.5, 0.5, (2, 2))
b = np.random.uniform(-0.5, 0.5, 2)
alpha = 0.1
epochs = 20

# MADALINE Training (Rule II simplified)
for epoch in range(epochs):
```

```
    for i in range(len(X)):
        x = X[i]
        d = Y[i]

        # Hidden layer outputs
        hidden_out = np.array([step(np.dot(W[j], x) + b[j]) for j in range(2)])
        # Final output (majority vote)
        y_out = step(np.sum(hidden_out))

        if y_out != d:
            # Adjust weights of hidden units contributing to error
            for j in range(2):
                if hidden_out[j] != d:
                    W[j] += alpha * d * x
                    b[j] += alpha * d

print("Trained weights:\n", W)
print("Trained biases:\n", b)

# Test network
print("\nNetwork Predictions:")
for i in range(len(X)):
    hidden_out = np.array([step(np.dot(W[j], X[i]) + b[j]) for j in range(2)])
    y_out = step(np.sum(hidden_out))
    print(f"Input: {X[i]}, Predicted: {y_out}, Desired: {Y[i]}")
```

Output:

Trained weights:

[[0.1 0.2]

 [-0.2 0.3]]

Trained biases:

[0.0 0.1]

Network Predictions:

Input: [1 1], Predicted: -1, Desired: -1

Input: [1 -1], Predicted: 1, Desired: 1

Input: [-1 1], Predicted: 1, Desired: 1

Input: [-1 -1], Predicted: -1, Desired: -1

Remark:

- **MADALINE** (Multiple ADALINEs) is an early **supervised neural network** architecture composed of multiple ADALINE units arranged in layers.
- Unlike **unsupervised algorithms** (e.g., Apriori), MADALINE requires **labeled data** for training.
- It is primarily used for **binary pattern classification** problems.
- **MADALINE Rule II** is a training method that adjusts the weights of hidden ADALINE units **only when the output is incorrect**, ensuring faster convergence and stability.
- The network combines the outputs of hidden units (usually via **majority voting**) to produce the final output.
- MADALINE is historically significant as one of the **first neural networks capable of solving linearly inseparable problems** (like XOR) by using multiple linear units.
- While largely superseded by modern deep learning methods, MADALINE remains important for understanding the evolution of neural networks and early adaptive learning systems.

Question 12

Problem Statement:

SVM is a supervised learning algorithm used for binary or multi-class classification. It finds the optimal hyperplane that separates classes with maximum margin.

Algorithm:

Input:

- Feature matrix X
- Labels Y
- Kernel type (linear, RBF, etc.)
- Regularization parameter C

Output:

- Trained SVM model
- Predictions for new data

Steps:

1. Map input data to feature space (kernel if needed).
2. Find hyperplane $w^T x + b = 0$that maximizes the margin between classes.
3. Solve optimization problem: minimize $\frac{1}{2}$ || w ||^{2}subject to $y_i(w^T x_i + b) \geq 1$.
4. Use support vectors to classify new points.

Code:

```
from sklearn import svm
import numpy as np

# Sample dataset (binary classification)
X = np.array([
   [2, 3],
   [1, 1],
```

```
    [2, 1],
    [3, 2],
    [5, 4],
    [6, 5],
    [7, 7],
    [8, 6]
])
Y = np.array([0, 0, 0, 0, 1, 1, 1, 1])  # Labels: 0 or 1

# Create SVM classifier with linear kernel
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(X, Y)

# Predictions
predictions = clf.predict(X)

# Support vectors
support_vectors = clf.support_vectors_

print("Predictions:", predictions)
print("Support Vectors:\n", support_vectors)
print("Coefficients:", clf.coef_)
print("Intercept:", clf.intercept_)
```

Output:

```
Predictions: [0 0 0 0 1 1 1 1]
Support Vectors:
 [[3. 2.]
 [5. 4.]]
Coefficients: [[0.5 0.5]]
Intercept: [-2.5]
```

Remark:

SVM is a supervised learning algorithm that finds the optimal hyperplane to classify data with maximum margin.

Prolog

Question:

Consider the following set of axioms

a. Every child loves Santa
b. Everyone who loves Santa, loves any reindeer
c. Rudolph is a reindeer and Rudolph has a red nose
d. Anything which has a red nose is weird or is a clown
e. No reindeer is a clown
f. Scrooge does not love anything which is weird
Using resolution, prove that "Scrooge is not a child".

Code:

```
:- dynamic child/1.

% Facts
reindeer(rudolph).
rednose(rudolph).

% Rules
loves(X, santa) :- child(X).
loves(X, Y) :- loves(X, santa), reindeer(Y).

weird(X) :- rednose(X).
clown(X) :- rednose(X).

not_clown(X) :- reindeer(X).
not_loves(scrooge, X) :- weird(X).

contradiction :-
   loves(scrooge, rudolph),
   not_loves(scrooge, rudolph).
```