In [1]:

```python
#Find-S Algorithm
import csv

def find_s_algorithm(training_data):
    # Initialize the hypothesis with the most specific values
    hypothesis = training_data[0][:-1]  # Assuming the last column is the class label

    # Iterate through the training data
    for instance in training_data:
        if instance[-1] == '1':  # Positive instance
            for i in range(len(hypothesis)):
                if instance[i] != hypothesis[i]:
                    hypothesis[i] = '?'

    return hypothesis

def load_data(file_path):
    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        data = [row for row in reader]
    return data

if __name__ == "__main__":
    # Example usage
    data_path = "C:/Users/GS.Devarayulu/OneDrive/Desktop/Iris.csv"
    training_data = load_data(data_path)

    # Apply Find-S algorithm
    hypothesis = find_s_algorithm(training_data)

    # Display the hypothesis
    print("Find-S Hypothesis:", hypothesis)
```

```
Find-S Hypothesis: ['Id', 'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']
```

In [2]:

```
#Candidate Elimination Algorithm
import copy

def initialize_hypotheses(attributes):
    # Initialize the version space with the most specific and most general hypotheses
    specific_hypothesis = ['0'] * len(attributes)
    general_hypothesis = ['?'] * len(attributes)
    return specific_hypothesis, general_hypothesis

def is_consistent(instance, hypothesis):
    # Check if the instance is consistent with the hypothesis
    for attr, value in zip(hypothesis, instance):
        if attr != '?' and attr != value:
            return False
    return True

def candidate_elimination(training_data):
    attributes = training_data[0][:-1]  # Assuming the last column is the class label
    specific_hypothesis, general_hypothesis = initialize_hypotheses(attributes)

    for instance in training_data:
        if instance[-1] == '1':  # Positive instance
            for i in range(len(attributes)):
                if specific_hypothesis[i] == '0':
                    specific_hypothesis[i] = instance[i]
                elif specific_hypothesis[i] != instance[i]:
                    specific_hypothesis[i] = '?'

            for i in range(len(attributes)):
                if specific_hypothesis[i] != '?' and specific_hypothesis[i] != general_hypothesis[i]:
                    general_hypothesis[i] = '?'
        elif instance[-1] == '0':  # Negative instance
            for i in range(len(attributes)):
                if specific_hypothesis[i] == instance[i]:
                    general_hypothesis[i] = specific_hypothesis[i]
                    specific_hypothesis[i] = '?'

    return specific_hypothesis, general_hypothesis

if __name__ == "__main__":
    # Example usage
```

```
    data_path = "C:/Users/GS.Devarayulu/OneDrive/Desktop/Iris.csv"
    training_data = load_data(data_path)

    # Apply Candidate Elimination algorithm
    specific_hypothesis, general_hypothesis = candidate_elimination(training_data)

    # Display the hypotheses
    print("Specific Hypothesis:", specific_hypothesis)
    print("General Hypothesis:", general_hypothesis)
```

```
Specific Hypothesis: ['0', '0', '0', '0', '0']
General Hypothesis: ['?', '?', '?', '?', '?']
```

In [3]:

```
#Decision Tree Based ID3 Algorithm
import math
from collections import Counter

class Node:
    def __init__(self, attribute=None, value=None, results=None, children=None):
        self.attribute = attribute    # Attribute to split on
        self.value = value            # Value of the attribute for the split
        self.results = results        # Class labels at this node if it's a leaf
        self.children = children      # Child nodes in the decision tree

def entropy(data):
    # Calculate the entropy of a set of instances
    labels = [instance[-1] for instance in data]
    label_counts = Counter(labels)
    total_instances = len(data)

    entropy_value = 0
    for count in label_counts.values():
        probability = count / total_instances
        entropy_value -= probability * math.log2(probability)

    return entropy_value

def information_gain(data, attribute_index):
    # Calculate the information gain for a specific attribute
    total_entropy = entropy(data)

    # Group instances by the values of the selected attribute
    attribute_values = set([instance[attribute_index] for instance in data])
    weighted_entropy = 0

    for value in attribute_values:
        subset = [instance for instance in data if instance[attribute_index] == value]
        probability = len(subset) / len(data)
        weighted_entropy += probability * entropy(subset)

    return total_entropy - weighted_entropy

def get_best_attribute(data, attributes):
    # Select the attribute with the highest information gain
```

```python
    information_gains = [information_gain(data, i) for i in range(len(attributes)-1)]
    best_attribute_index = information_gains.index(max(information_gains))
    return attributes[best_attribute_index]

def build_tree(data, attributes):
    # Recursively build the decision tree
    labels = [instance[-1] for instance in data]

    # If all instances have the same class label, create a leaf node
    if len(set(labels)) == 1:
        return Node(results=labels[0])

    # If there are no more attributes to split on, create a leaf node with the majority class label
    if len(attributes) == 1:
        majority_label = Counter(labels).most_common(1)[0][0]
        return Node(results=majority_label)

    # Select the best attribute to split on
    best_attribute = get_best_attribute(data, attributes)
    node = Node(attribute=best_attribute)

    # Recursively build the tree for each value of the selected attribute
    values = set([instance[attributes.index(best_attribute)] for instance in data])
    for value in values:
        subset = [instance[:-1] for instance in data if instance[attributes.index(best_attribute)] == valu
        child_attributes = [attr for attr in attributes if attr != best_attribute]
        child_node = build_tree(subset, child_attributes)
        node.children = node.children or {}
        node.children[value] = child_node

    return node

def print_tree(node, indent=""):
    # Print the decision tree
    if node.results is not None:
        print(indent + "Class:", node.results)
    else:
        print(indent + "Attribute:", node.attribute)
        for value, child_node in node.children.items():
            print(indent + "|-" + str(value))
            print_tree(child_node, indent + "  ")
```

```python
if __name__ == "__main__":
    # Example usage
    data_path = "C:/Users/GS.Devarayulu/OneDrive/Desktop/Iris.csv"
    training_data = load_data(data_path)
    attributes = training_data[0][:-1]  # Assuming the last column is the class label

    # Build the decision tree
    root_node = build_tree(training_data, attributes)

    # Print the decision tree
    print_tree(root_node)
```

```
Attribute: Id
|-29
  Class: 0.2
|-124
  Class: 1.8
|-48
  Class: 0.2
|-60
  Class: 1.4
|-21
  Class: 0.2
|-108
  Class: 1.8
|-53
  Class: 1.5
|-127
  Class: 1.8
|-123
  Class: 2.0
```

In [4]:
```
#ANN by using Back Propagation Algorithm
import numpy as np

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases
        self.weights_input_hidden = np.random.rand(self.input_size, self.hidden_size)
        self.biases_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.rand(self.hidden_size, self.output_size)
        self.biases_output = np.zeros((1, self.output_size))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward(self, X):
        # Forward pass
        self.hidden_layer_input = np.dot(X, self.weights_input_hidden) + self.biases_hidden
        self.hidden_layer_output = self.sigmoid(self.hidden_layer_input)

        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_hidden_output) + self.bia
        self.output_layer_output = self.sigmoid(self.output_layer_input)

        return self.output_layer_output

    def backward(self, X, y, output):
        # Backward pass
        error_output = y - output
        delta_output = error_output * self.sigmoid_derivative(output)

        error_hidden = delta_output.dot(self.weights_hidden_output.T)
        delta_hidden = error_hidden * self.sigmoid_derivative(self.hidden_layer_output)

        # Update weights and biases
```

```
        self.weights_hidden_output += self.hidden_layer_output.T.dot(delta_output) * self.learning_rate
        self.biases_output += np.sum(delta_output, axis=0, keepdims=True) * self.learning_rate

        self.weights_input_hidden += X.T.dot(delta_hidden) * self.learning_rate
        self.biases_hidden += np.sum(delta_hidden, axis=0, keepdims=True) * self.learning_rate

    def train(self, X, y, epochs):
        for epoch in range(epochs):
            output = self.forward(X)
            self.backward(X, y, output)

            if epoch % 1000 == 0:
                loss = np.mean(np.square(y - output))
                print(f"Epoch {epoch}, Loss: {loss}")

    def predict(self, X):
        return self.forward(X)

# Example usage
if __name__ == "__main__":
    # Sample dataset (XOR problem)
    X = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])

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

    # Initialize and train the neural network
    input_size = 2
    hidden_size = 4
    output_size = 1
    learning_rate = 0.1

    neural_network = NeuralNetwork(input_size, hidden_size, output_size, learning_rate)
    neural_network.train(X, y, epochs=10000)

    # Test the trained neural network
    predictions = neural_network.predict(X)
```

```
    print("\nPredictions:")
    print(predictions)
```

```
Epoch 0, Loss: 0.27700271974174406
Epoch 1000, Loss: 0.2438537727821578
Epoch 2000, Loss: 0.2000426242867202
Epoch 3000, Loss: 0.11370106229236214
Epoch 4000, Loss: 0.03605996921610707
Epoch 5000, Loss: 0.014833609708491957
Epoch 6000, Loss: 0.008398486720701702
Epoch 7000, Loss: 0.005626972221129503
Epoch 8000, Loss: 0.004149231463463141
Epoch 9000, Loss: 0.00325023314190095

Predictions:
[[0.0575738 ]
 [0.951083  ]
 [0.95158723]
 [0.0506053 ]]
```

In [6]:

```
#Naive Bayes Classifier
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics

# Sample data (you can replace this with your own dataset)
data = [
    {'text': 'I love programming', 'label': 'positive'},
    {'text': 'Python is great', 'label': 'positive'},
    {'text': 'I dislike bugs', 'label': 'negative'},
    {'text': 'Programming is challenging', 'label': 'negative'},
]

# Extract features and labels
texts = [entry['text'] for entry in data]
labels = [entry['label'] for entry in data]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(texts, labels, test_size=0.2, random_state=42)

# Convert text data to a bag-of-words representation
vectorizer = CountVectorizer()
X_train_vectorized = vectorizer.fit_transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)

# Train a Naive Bayes classifier
classifier = MultinomialNB()
classifier.fit(X_train_vectorized, y_train)

# Make predictions on the test set
y_pred = classifier.predict(X_test_vectorized)

# Evaluate the classifier
accuracy = metrics.accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')

# Example prediction
new_text = 'I enjoy coding'
new_text_vectorized = vectorizer.transform([new_text])
prediction = classifier.predict(new_text_vectorized)[0]
```

```
print(f'Prediction for "{new_text}": {prediction}')
```

```
Accuracy: 0.0
Prediction for "I enjoy coding": negative
```

In [9]:

```python
#EM Algorithm
import pandas as pd
import numpy as np
from sklearn.mixture import GaussianMixture
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

# Load data from CSV file
data = pd.read_csv("C:/Users/GS.Devarayulu/OneDrive/Desktop/Iris.csv")

# Assuming your data has features you want to cluster
X = data.iloc[:, :-1].values  # Assuming the last column contains labels

# Standardize the data
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)

# Apply Gaussian Mixture Model with Expectation-Maximization
n_clusters = 3  # Specify the number of clusters
gmm = GaussianMixture(n_components=n_clusters, random_state=42)
gmm.fit(X_standardized)

# Get cluster assignments for each data point
cluster_assignments = gmm.predict(X_standardized)

# Add cluster assignments to the original DataFrame
data['Cluster'] = cluster_assignments

# Visualize the results (for 2D data)
if X.shape[1] == 2:
    plt.scatter(X_standardized[:, 0], X_standardized[:, 1], c=cluster_assignments, cmap='viridis')
    plt.title('EM Clustering Results')
    plt.show()

# Print cluster sizes
cluster_sizes = np.bincount(cluster_assignments)
for i, size in enumerate(cluster_sizes):
    print(f"Cluster {i}: {size} instances")

# Print cluster means and covariances
for i in range(n_clusters):
```

```
    print(f"\nCluster {i}:\nMean: {gmm.means_[i]}\nCovariance:\n{gmm.covariances_[i]}")
```

```
Cluster 0: 50 instances
Cluster 1: 50 instances
Cluster 2: 50 instances

Cluster 0:
Mean: [ 1.1532549   0.9012241  -0.1860831   1.01939967  1.08686022]
Covariance:
[[ 0.11231949 -0.00213875  0.03310702 -0.02073082  0.00516509]
 [-0.00213875  0.58162439  0.25791603  0.20475835  0.07713933]
 [ 0.03310702  0.25791603  0.54555029  0.09215373  0.14233109]
 [-0.02073082  0.20475835  0.09215373  0.09648023  0.03594225]
 [ 0.00516509  0.07713933  0.14233109  0.03594225  0.12807347]]

Cluster 1:
Mean: [-1.1547262  -1.01457897  0.84230679 -1.30487835 -1.25512862]
Covariance:
[[ 0.1110726  -0.00472915 -0.01939923  0.00173354  0.00406865]
 [-0.00472915  0.17877068  0.27559705  0.01089769  0.01646556]
 [-0.01939923  0.27559705  0.76185102  0.0150643   0.03409836]
 [ 0.00173354  0.01089769  0.0150643   0.00954173  0.00417477]
 [ 0.00406865  0.01646556  0.03409836  0.00417477  0.01947098]]

Cluster 2:
Mean: [-2.75591681e-04  1.12159924e-01 -6.56936747e-01  2.84365555e-01
  1.66875189e-01]
Covariance:
[[ 0.11120556 -0.0556294  -0.01959485 -0.01686189 -0.01459697]
 [-0.0556294   0.38394242  0.23445733  0.12363649  0.08716777]
 [-0.01959485  0.23445733  0.51748676  0.10687147  0.12312558]
 [-0.01686189  0.12363649  0.10687147  0.06974231  0.05338602]
 [-0.01459697  0.08716777  0.12312558  0.05338602  0.06615774]]
```

In [10]:

```
#kNN Algorithm
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
import pandas as pd

# Load your dataset (replace "your_dataset.csv" with the actual file path)
data = pd.read_csv("C:/Users/GS.Devarayulu/OneDrive/Desktop/Iris.csv")

# Assuming the last column is the target variable (class label)
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features (optional, but often recommended)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Choose the value of k (number of neighbors)
k = 3

# Create kNN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=k)

# Train the classifier
knn_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn_classifier.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Classification Report:\n", classification_rep)
```

```
Accuracy: 1.0
Classification Report:
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      1.00      1.00         9
 Iris-virginica       1.00      1.00      1.00        11

       accuracy                           1.00        30
      macro avg       1.00      1.00      1.00        30
   weighted avg       1.00      1.00      1.00        30
```

In [ ]: