AIML LAB MANUAL

IoT –CSBC

```
import networkx as nx
import matplotlib.pyplot as plt

# Create a graph representing the city map
G = nx.Graph()

# Add cities and roads to the graph
cities = {'A': 'SEACET', 'B': 'K.R.Puram', 'C': 'ITPL', 'D': 'RM nagar', 'E': 'Tin Factory'}
roads = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'E'), ('D', 'E')]

G.add_nodes_from(cities.keys())
G.add_edges_from(roads)
```

```
# Visualize the graph with city names
pos = nx.spring_layout(G)
nx.draw(G, pos, labels=cities, with_labels=True, font_weight='bold', node_size=700, node_color='skyblue', font_size=10, edge_colo
plt.title('City Map')
plt.show()
```

```
# Breadth-First Search (BFS) implementation
def bfs(graph, start):
    visited = set()
    queue = [start]
    bfs_result = []

    while queue:
        node = queue.pop(0)
        if node not in visited:
            bfs_result.append(node)
            visited.add(node)
            queue.extend(neighbor for neighbor in graph.neighbors(node) if neighbor not in visited)

    return bfs_result
```

```
# Depth-First Search (DFS) implementation
def dfs(graph, start):
    visited = set()
    dfs_result = []

    def dfs_recursive(node):
        nonlocal visited
        visited.add(node)
        dfs_result.append(node)
        for neighbor in graph.neighbors(node):
            if neighbor not in visited:
                dfs_recursive(neighbor)

    dfs_recursive(start)
    return dfs_result
```

```
# Perform BFS and DFS starting from City A
start_city = 'A'
bfs_result = bfs(G, start_city)
dfs_result = dfs(G, start_city)

print('BFS Result:', [cities[node] for node in bfs_result])
print('DFS Result:', [cities[node] for node in dfs_result])
```

Program 1: Aim: To implement and evaluate DFS and BFS algorithm.

Given city map compare the following uniform search algorithm DFS and BFS

Introduction

Breadth-First Search is an algorithm for traversing or searching tree or graph data structures. In the context of a city map, BFS explores the city network level by level, starting from a specific city. It systematically visits all the neighbors of a city before moving on to their neighbors.

Depth-First Search is another algorithm for traversing or searching tree or graph data structures. In DFS, the algorithm explores as far as possible along each branch before backtracking. In the context of a city map, DFS may explore a single route deeply before exploring other alternatives.

Graph Representation

The city map is represented as a graph where cities are nodes, and roads are edges. NetworkX is used to create and visualize the graph.

City Map Significance

The code highlights the significance of using graph algorithms in the context of a city map, showcasing their utility in navigation, route planning, and network analysis.

Algorithmic Exploration

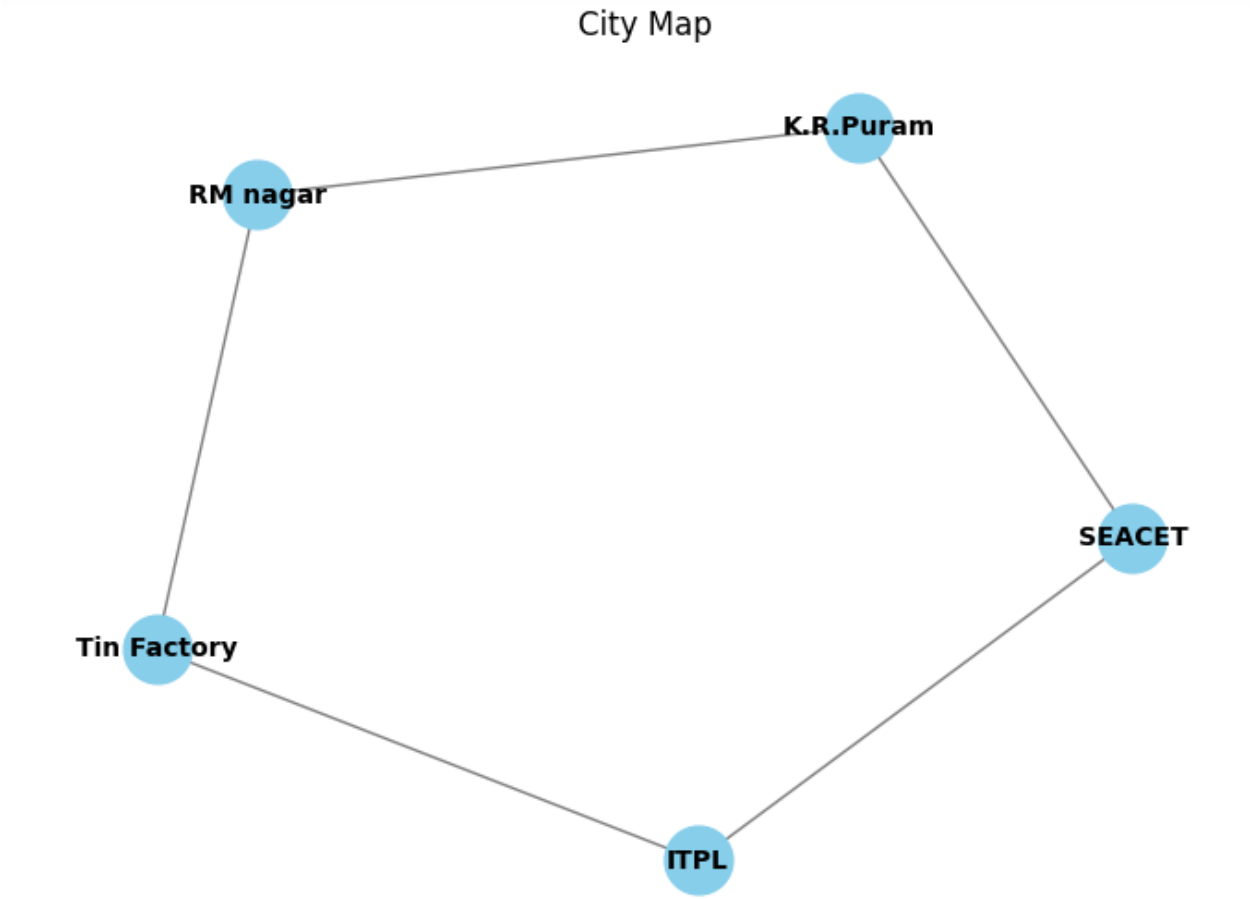
The BFS and DFS algorithms are applied to explore the city map, providing insights into different aspects of the network.

Visualization

The code visualizes the city map, providing a clear representation of the connectivity between cities and visualize the graph.

Use case in City Map ---BFS

1. **Starting Point**:
    - Choose a starting City (e.g., City A)
2. **Exploration**:
    - Visit all the neighboring cities of the starting city.
    - Move on to the neighbor's neighbors, exploring each level of the city map before moving deeper.



Result: -

```
BFS Result: ['SEACET', 'K.R.Puram', 'ITPL', 'RM nagar', 'Tin Factory']
DFS Result: ['SEACET', 'K.R.Puram', 'RM nagar', 'Tin Factory', 'ITPL']
```

3. **Shortest Paths:**
    - BFS naturally finds the shortest paths between the starting city and all other reachable cities. This is particularly useful for finding the shortest routes in a city map by exploring the city map level by level.
4. **Applications:**
    - BFS can be applied to tasks like finding the shortest route from one city to another, determining the connectivity of the city network, and identifying all cities reachable within a certain distance
5. **Queue Mechanism:**
    - BFS uses a queue to keep track of the order in which cities are visited. The starting city is enqueued, and then its neighbors are enqueued one by one, ensuring a level-wise exploration.

Use case in City Map ---BFS

1. **Starting Point**:
    - Choose a starting City (e.g., City A)
2. **Exploration**:
    - Explore as deeply as possible along each road before backtracking to explore other roads.
3. **Alternative routes.**
    - DFS is suitable for exploring alternative routes and uncovering various paths in the city network. It may not guarantee the shortest path but can uncover different paths.
4. **Applications:**
    - DFS can be applied to tasks like identifying connected components in the city map, exploring all possible routes, and detecting cycles in the network.
5. **Stack Mechanism:**
    - DFS often uses a stack or recursion to keep track of the exploration. It explores a path until it reaches a dead-end and then backtracks to explore other paths.

Conclusion

The provided code offers a hands-on demonstration of BFS and DFS algorithms applied to a city map, emphasizing their importance in solving graph-related problems. It serves as an educational tool for understanding these algorithms and their practical applications in the context of urban networks. The visualization aspect enhances the code's effectiveness in conveying the connectivity and exploration process within the city map.

```python
# Import necessary libraries
import networkx as nx
import matplotlib.pyplot as plt
import heapq

# Create a graph representing the city map
G = nx.Graph()

# Add cities and roads to the graph
cities = {'A': 'SEACET', 'B': 'K.R.Puram', 'C': 'ITPL', 'D': 'RM Nagar', 'E': 'Tin Factory','F':'Whitefield','G': 'HOSKOTE'}
roads = [('A', 'B', 5), ('A', 'C', 3), ('B', 'D', 8), ('C', 'E', 2), ('D', 'E', 4),('F','A',2),('G','C',3)]
G.add_weighted_edges_from(roads)

# Visualize the graph with city names and edge weights
pos = nx.spring_layout(G)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw(G, pos, labels=cities, with_labels=True, font_weight='bold', node_size=700, node_color='skyblue', font_size=10, edge_
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.title('City Map with Weights')
plt.show()

# Define heuristic function
def heuristic(node, goal):
    return nx.shortest_path_length(G, node, goal, weight='weight')
```

```python
# Breadth-First Search (BFS) implementation
def bfs(graph, start, goal):
    visited = set()
    queue = [start]
    bfs_result = []

    while queue:
        node = queue.pop(0)
        if node == goal:
            return visited
        if node not in visited:
            visited.add(node)
            bfs_result.append(node)
            queue.extend(neighbor for neighbor in graph.neighbors(node) if neighbor not in visited)
```

```python
# Depth-First Search (DFS) implementation
def dfs(graph, start, goal):
    visited = set()
    dfs_result = []

    def dfs_recursive(node):
        nonlocal visited
        visited.add(node)
        dfs_result.append(node)
        for neighbor in graph.neighbors(node):
            if neighbor not in visited:
                dfs_recursive(neighbor)

    dfs_recursive(start)
    return dfs_result
```

Program 2: Aim: To implement, evaluate A* algorithm and identify the difference between BFS/DFS and A*.

Given city map and heuristic values implement A* algorithm and identify the difference between uniform search and heuristic search algorithm.

Introduction

A* algorithm, along with comparisons to Breadth-First Search (BFS) and Depth-First Search (DFS), on a city map represented as a graph. This implementation is done using the NetworkX library for graph creation and manipulation and Matplotlib for visualization.

Use case in City Map

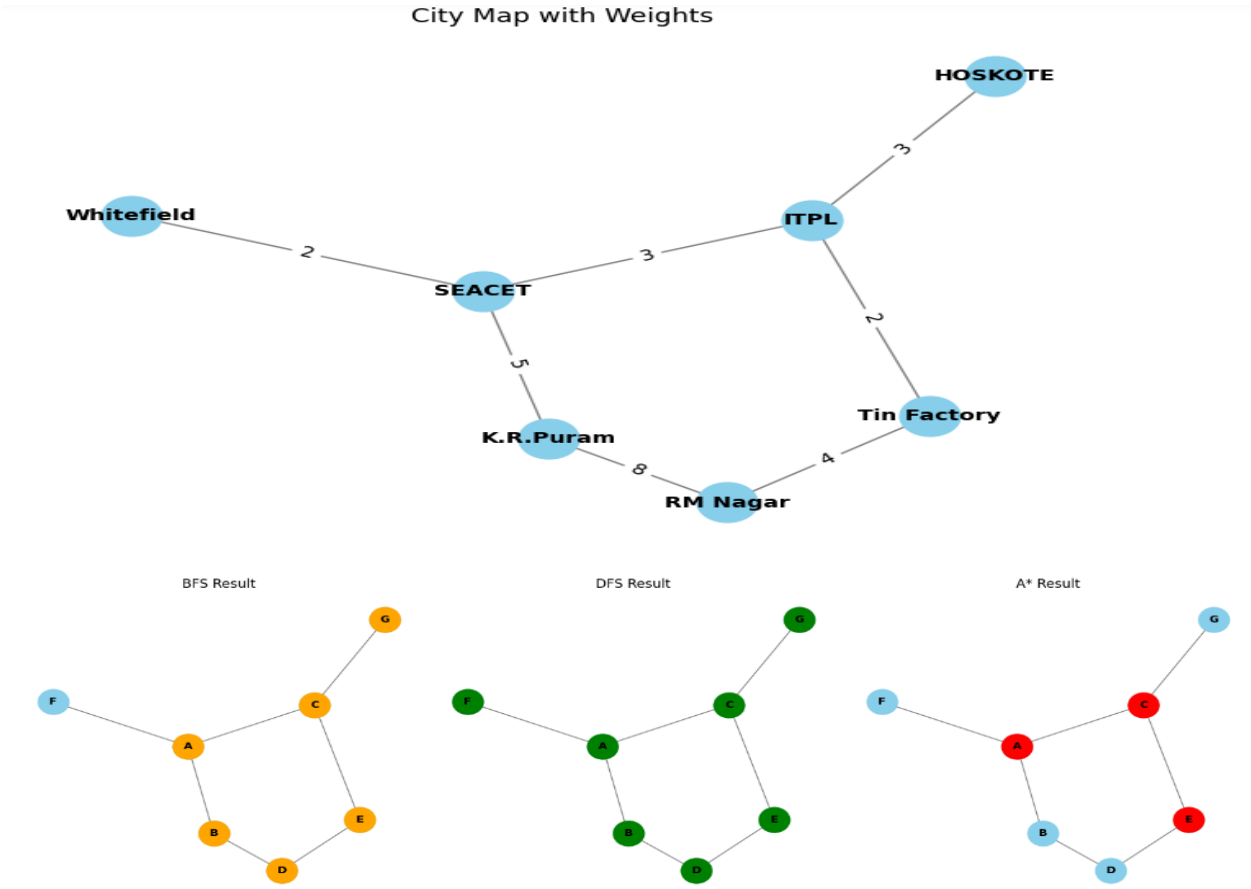
1. **Import Libraries:**
   - Import necessary libraries like NetworkX, Matplotlib, and heapq.
2. **Create City Map Graph:**
   - Use NetworkX to create a graph representing the city map.
   - Add cities as nodes and roads as edges with weights.
   - Visualize the city map using Matplotlib.
3. **Define Heuristic Function:**
   - Define a heuristic function for A* search.
   - The heuristic should estimate the cost from a given city to the goal city.
4. **Breadth-First Search (BFS) Function:**
   - Implement a function for Breadth-First Search.
   - Use a queue data structure to explore the city map.
   - Visualize the visited nodes during BFS.
5. **Depth-First Search (DFS) Function:**
   - Implement a function for Depth-First Search.
   - Use a stack data structure to explore the city map.
   - Visualize the visited nodes during DFS.
6. *A Search Function:**
   - Implement the A* search algorithm.
   - Use a priority queue (heap) to explore the city map based on total cost (path cost + heuristic).
   - Visualize the visited nodes during A* search.
7. **Run and Compare Algorithms:**
   - Choose a start city and a goal city for exploration.
   - Run BFS, DFS, and A* searches from the start city to the goal city.
   - Collect and display the paths explored, total distances, and efficiency of each algorithm.

```python
# A* Search implementation
def astar(graph, start, goal, h_func):
    priority_queue = [(0, start)]
    visited = set()

    while priority_queue:
        current_cost, current_node = heapq.heappop(priority_queue)

        if current_node == goal:
            return visited

        if current_node not in visited:
            visited.add(current_node)

            for neighbor, weight in graph[current_node].items():
                if neighbor not in visited:
                    total_cost = current_cost + weight['weight'] + h_func(neighbor, goal)
                    heapq.heappush(priority_queue, (total_cost, neighbor))
```

```python
# Perform BFS, DFS, and A* searches
start_city = 'E'
goal_city = 'F'

visited_bfs = bfs(G, start_city, goal_city)
visited_dfs = dfs(G, start_city, goal_city)
visited_astar = astar(G, start_city, goal_city, heuristic)
```

```python
# Visualization
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
nx.draw(G, pos, with_labels=True, font_weight='bold', node_size=700, node_color='skyblue', font_size=10, edge_color='gray',
nx.draw_networkx_nodes(G, pos, nodelist=visited_bfs, node_color='orange', node_size=700)
plt.title('BFS Result')

plt.subplot(1, 3, 2)
nx.draw(G, pos, with_labels=True, font_weight='bold', node_size=700, node_color='skyblue', font_size=10, edge_color='gray',
nx.draw_networkx_nodes(G, pos, nodelist=visited_dfs, node_color='green', node_size=700)
plt.title('DFS Result')

plt.subplot(1, 3, 3)
nx.draw(G, pos, with_labels=True, font_weight='bold', node_size=700, node_color='skyblue', font_size=10, edge_color='gray',
nx.draw_networkx_nodes(G, pos, nodelist=visited_astar, node_color='red', node_size=700)
plt.title('A* Result')

plt.tight_layout()
plt.show()
print('A* Search Result:', [cities[node] for node in visited_astar])
print('BFS Search Result:', [cities[node] for node in visited_bfs])
print('DFS Search Result:', [cities[node] for node in visited_dfs])
```

8. **Visualize Results:**
   - Visualize the paths explored by each algorithm on the city map.
   - Highlight the optimal path found by A* search.
9. **Analytical Comparison:**
   - Compare the efficiency of BFS, DFS, and A* in terms of explored paths, computational complexity, and optimality.
   - Discuss the strengths and weaknesses of each algorithm.
10. **Conclusion:**
    - Summarize the findings and insights gained from the comparison.
    - Discuss scenarios where each algorithm might be preferred.



Result:-

```
A* Search Result: ['SEACET', 'ITPL', 'Tin Factory']
BFS Search Result: ['RM Nagar', 'Tin Factory', 'HOSKOTE', 'K.R.Puram', 'SEACET', 'ITPL']
DFS Search Result: ['Tin Factory', 'ITPL', 'SEACET', 'K.R.Puram', 'RM Nagar', 'Whitefield', 'HOSKOTE']
```

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, export_text
from sklearn import metrics
import math
from sklearn.preprocessing import LabelEncoder

def calculate_entropy(y):
    """
    Calculate entropy for a given target variable.
    """
    unique_labels = y.unique()
    entropy = 0
    total_samples = len(y)

    for label in unique_labels:
        label_count = len(y[y == label])
        probability = label_count / total_samples
        entropy -= probability * math.log2(probability)

    return entropy
```

```python
def calculate_information_gain(X, y, feature):
    """
    Calculate information gain for a given feature.
    """
    entropy_parent = calculate_entropy(y)
    unique_values = X[feature].unique()
    total_samples = len(y)
    weighted_entropy_child = 0

    for value in unique_values:
        subset_y = y[X[feature] == value]
        weight = len(subset_y) / total_samples
        weighted_entropy_child += weight * calculate_entropy(subset_y)

    information_gain = entropy_parent - weighted_entropy_child
    return information_gain
```

```python
# Load the tennis dataset
# Replace 'tennisdata.csv' with the actual filename
df = pd.read_csv(r'C:\Users\lenovo\Desktop\tennisdata.csv')

# Assuming the target variable is 'PlayTennis'. Adjust this based on your dataset.
target_variable = 'PlayTennis'

# Split the dataset into features and target variable
X = df.drop(target_variable, axis=1)
y = df[target_variable]

# Convert categorical variables to numerical using Label Encoding
label_encoder = LabelEncoder()
for column in X.select_dtypes(include=['object']).columns:
    X[column] = label_encoder.fit_transform(X[column])

# Calculate entropy for the entire dataset
entropy_before_split = calculate_entropy(y)
print(f"Entropy before split: {entropy_before_split:.4f}")

# Calculate information gain for each feature
for feature in X.columns:
    information_gain = calculate_information_gain(X, y, feature)
    print(f"Information Gain for {feature}: {information_gain:.4f}")
```

Program 3: **Aim: To Construct the Decision tree using the training data sets under Supervised learning Concept.**

Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

Introduction

Decision trees are powerful machine learning models capable of performing both classification and regression tasks. The ID3 algorithm, standing for Iterative Dichotomiser 3, is a popular decision tree algorithm that recursively splits the dataset based on the features to create a tree-like structure.

Decision tree classifier using the ID3 algorithm to predict whether to play tennis based on certain weather conditions. Here's an overview of the algorithmic steps involved:

1. **Load Dataset:**
   The code begins by loading a dataset from a CSV file using the pandas library. The dataset contains information about weather conditions and whether or not tennis was played. The filename is assumed to be 'tennisdata.csv'.

2. **Preprocess Dataset:**
   The target variable for prediction is assumed to be 'PlayTennis'. The code checks if this variable is present in the dataset columns. Categorical variables in the dataset are converted to numerical values using Label Encoding to make them suitable for training a machine learning model.

3. **Split Dataset:**
   The dataset is split into features (X) and the target variable (y). Then, the data is further split into training and testing sets using the `train_test_split` function from scikit-learn.
4. **Build Decision Tree Model(ID3 Algorithm):**
   A decision tree classifier is instantiated with the ID3 algorithm (`criterion='entropy'`). The model is then trained using the training data.
5. **Make Predicitons on Test Set:**
   The trained decision tree model is used to make predictions on the test set (`X_test).`

```python
# 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)

# Build the decision tree model (ID3 algorithm)
model = DecisionTreeClassifier(criterion='entropy')  # ID3 uses entropy as the criterion
model.fit(X_train, y_train)

# Visualize the decision tree (optional)
tree_rules = export_text(model, feature_names=list(X.columns))
print("Decision Tree Rules:\n", tree_rules)

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

# Evaluate the model
accuracy = metrics.accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

```
Tennis Dataset:
     Outlook Temperature Humidity  Windy PlayTennis
0      Sunny         Hot     High  False         No
1      Sunny         Hot     High   True         No
2   Overcast         Hot     High  False        Yes
3      Rainy        Mild     High  False        Yes
4      Rainy        Cool   Normal  False        Yes
5      Rainy        Cool   Normal   True         No
6   Overcast        Cool   Normal   True        Yes
7      Sunny        Mild     High  False         No
8      Sunny        Cool   Normal  False        Yes
9      Rainy        Mild   Normal  False        Yes
10     Sunny        Mild   Normal   True        Yes
11  Overcast        Mild     High   True        Yes
12  Overcast         Hot   Normal  False        Yes
13     Rainy        Mild     High   True         No
Decision Tree Rules:
|--- Outlook <= 0.50
|    |--- class: Yes
|--- Outlook >  0.50
|    |--- Windy <= 0.50
|    |    |--- Temperature <= 1.00
|    |    |    |--- class: Yes
|    |    |--- Temperature >  1.00
|    |    |    |--- Outlook <= 1.50
|    |    |    |    |--- class: Yes
|    |    |    |--- Outlook >  1.50
|    |    |    |    |--- class: No
|    |--- Windy >  0.50
|    |    |--- Outlook <= 1.50
|    |    |    |--- class: No
|    |    |--- Outlook >  1.50
|    |    |    |--- Temperature <= 1.50
|    |    |    |    |--- class: No
|    |    |    |--- Temperature >  1.50
|    |    |    |    |--- class: Yes

Accuracy: 66.67%
```

6. **Evaluate the Model:**

The accuracy of the model is evaluated using the `accuracy_score` function from scikit-learn, comparing the predicted values (`y_pred`) with the actual values in the test set (`y_test`). The accuracy is then printed to assess the performance of the decision tree classifier.

This algorithmic flow demonstrates the typical steps involved in implementing a decision tree classifier using the ID3 algorithm, including data loading, preprocessing, model training, visualization, and model evaluation. The choice of the ID3 algorithm is indicated by specifying 'entropy' as the criterion for the decision tree classifier.

Conclusion:

The decision tree classifier successfully learned from the training data and made predictions on new, unseen data. The accuracy score provides an indication of how well the model generalizes to new instances. The ID3 algorithm, with its use of entropy as the criterion, is suitable for building decision trees that can make predictions based on categorical features.

In [1]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import style
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

In [5]:
```
df=dataset = pd.read_csv(r'C:\Users\lenovo\Desktop\insurance.csv')
```

In [6]:
```
df.head()
```

Out[6]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

In [7]:
```
df['sex'] = df['sex'].apply({'male':0,'female':1}.get)
df['smoker']= df['smoker'].apply({'yes':1,'no':0}.get)
df['region'] = df['region'].apply({'southwest':1,'southeast':2,'northwest':3,'northeast':4}.get)
```

In [8]:
```
df.head()
```

Out[8]:

	age	sex	bmi	children	smoker	region	charges
0	19	1	27.900	0	1	1	16884.92400
1	18	0	33.770	1	0	2	1725.55230
2	28	0	33.000	3	0	2	4449.46200
3	33	0	22.705	0	0	3	21984.47061
4	32	0	28.880	0	0	3	3866.85520

In [9]:
```
x = df.drop(['charges','sex'], axis=1)
y = df.charges
```

In [10]:
```
x_train,x_test,y_train,y_test = train_test_split(x,y, test_size=0.3, random_state =42)
print("x_train shape: ",x_train.shape)
print("x_test shape: ",x_test.shape)
print("y_train shape: ",y_train.shape)
print("y_test shape: ",y_test.shape)
```

```
x_train shape:  (936, 5)
x_test shape:  (402, 5)
y_train shape:  (936,)
y_test shape:  (402,)
```

Program 4 : Aim : To evaluate linear regression for prediction.

Demonstrate and analyse the application of Linear regression model for predicting insurance cost.

Introduction

Linear regression is a statistical technique used for modeling the relationship between a dependent variable (target) and one or more independent variables (features). In simple linear regression, there is one independent variable, while multiple independent variables lead to multiple linear regression

This Python code uses a linear regression model to predict medical insurance costs based on features such as age, BMI, number of children, smoking status, and region

1. **Predictive Modeling:**
   - Linear regression is a popular choice for predictive modeling when there is a linear relationship between the independent variables (features) and the dependent variable (target). In this case, the code aims to predict medical insurance costs based on features such as age, BMI, number of children, smoking status, and region.
2. **Model Training:**
   - The linear regression model (`Linreg`) is trained on the training set (`x_train` and `y_train`). The model learns the relationship between the input features and the target variable from the provided data. The `fit` method is used to find the coefficients (weights) that minimize the difference between the predicted and actual values.
3. **Prediction:**
   - After training, the model is used to make predictions (`pred`) on the test set (`x_test`). The predicted values represent the estimated medical insurance costs based on the learned relationships from the training data.
4. **Evaluation:**
   - The code uses the R2 score (`r2_score` from scikit-learn) to evaluate the performance of the linear regression model. The R2 score measures how well the model explains the variability in the target variable. A higher R2 score indicates a better fit of the model to the data.
5. **New Customer Predictions:**
   - The trained linear regression model is applied to predict medical insurance costs for new customer scenarios (`Cost_pred1` and `Cost_pred2`). These predictions demonstrate the model's ability to generalize to unseen data and provide estimates for different customer profiles.

In [11]:
```
Linreg = LinearRegression()
Linreg.fit(x_train,y_train)
pred = Linreg.predict(x_test)
from sklearn.metrics import r2_score
print("R2 score: ",(r2_score(y_test,pred)))
```

```
R2 score:  0.7697211527941855
```

In [14]:
```
Data = {'age':50,'bmi':25,'children':12 ,'smoker':0,'region':2}
Index =[0]
Cust_df = pd.DataFrame(Data,Index)
Cust_df
Cost_pred1 = Linreg.predict(Cust_df)
print("The medical insurance cost of the new customer category Non Smoker : ",Cost_pred1)
Data = {'age':50,'bmi':25,'children':12 ,'smoker':1,'region':2}
Index =[0]
Cust_df = pd.DataFrame(Data,Index)
Cust_df
Cost_pred2 = Linreg.predict(Cust_df)
print("The medical insurance cost of the new customer category Smoker : ",Cost_pred2)
```

```
The medical insurance cost of the new customer category Non Smoker :  [13831.50207896]
The medical insurance cost of the new customer category Smoker :  [37458.56459121]
```

1. **Interpretability:**
   Linear regression models offer interpretability, as the coefficients associated with each feature provide insights into the strength and direction of their influence on the target variable. This interpretability is valuable for understanding how specific factors contribute to predicted outcomes.
2. **Limitations:**
   a. It's important to note that linear regression assumes a linear relationship between features and the target. If the relationship is more complex or nonlinear, other models may be more suitable. Additionally, linear regression is sensitive to outliers and may not perform well in the presence of multicollinearity.

Use case

1. **Dataset Loading and Preprocessing:**
   - The dataset is loaded from a CSV file ('insurance.csv') using pandas. Categorical variables like 'sex', 'smoker', and 'region' are converted to numerical values for model training.
2. **Data Splitting:**
   - The dataset is split into features (`x`) and the target variable (`y`). The data is further divided into training and testing sets using the `train_test_split` function.
3. **Linear Regression Model Training:**
   - A linear regression model is created using the `LinearRegression` class from scikit-learn. The model is trained on the training set.
4. **Prediction and Model Evaluation:**
   - The trained model is used to make predictions (`pred`) on the test set. The R2 score, a measure of how well the model fits the data, is calculated using the `r2_score` function.
5. **Prediction for New Customer Categories:**
   - Two new customer scenarios are created with different smoking statuses. Predictions for medical insurance costs for these new customers are made using the trained linear regression model.
6. **Conclusion:**
   - The code successfully preprocesses the data, trains a linear regression model, and evaluates its performance using the R2 score. The predictions for new customer scenarios demonstrate the model's ability to estimate medical insurance costs based on the provided features.

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) #maximum of X array longitudinally
y = y/100

#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)

#Variable initialization
epoch=5 #Setting training iterations
lr=0.1 #Setting learning rate

inputlayer_neurons = 2 #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
#weight and bias initialization
```

```
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))

#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
    #Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+bout
    output = sigmoid(outinp)

    #Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO * outgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer wts contributed to error
    d_hiddenlayer = EH * hiddengrad

    wout += hlayer_act.T.dot(d_output) *lr   # dotproduct of nextlayererror and currentlayerop
    wh += X.T.dot(d_hiddenlayer) *lr
```

```
    print ("----------Epoch-", i+1, "Starts----------")
    print("Input: \n" + str(X))
    print("Actual Output: \n" + str(y))
    print("Predicted Output: \n" ,output)
    print ("----------Epoch-", i+1, "Ends----------\n")

print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
print("Final Error in predicted output :\n" ,str(y-output))
```

Program 5: To understand the working principle of ANN with feedforward and backward principle.

Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets.

Demonstrate simple neural network with one hidden layer using the sigmoid activation function for binary classification.

Let's break down the key components and provide an introduction:

1. **Data Preparation:**
   - The code initializes input data `X` and target output `y`. The input data is normalized by dividing it by the maximum value along each column, and the target output is scaled by dividing it by 100. This preprocessing helps ensure that the data is within a suitable range for the neural network to learn effectively.
2. **Neural Network Architecture:**
   - The neural network architecture is defined with an input layer consisting of 2 neurons, one hidden layer with 3 neurons, and an output layer with 1 neuron. Weights (`wh`, `wout`) and biases (`bh`, `bout`) are initialized with random values. This constitutes a simple feedforward neural network.
3. **Sigmoid Activation Function:**
   - The sigmoid activation function is defined to introduce non-linearity to the network. The sigmoid function squashes the output between 0 and 1, making it suitable for binary classification problems.
4. **Training (Forward Propagation and Backpropagation):**
   - The training loop (`for i in range(epoch)`) iterates over a specified number of epochs. During each epoch, the neural network undergoes forward propagation to compute the predicted output. Subsequently, backpropagation is performed to update the weights and biases based on the error. The learning rate (`lr`) determines the step size during weight updates.
5. **Printing Epoch Information:**
   - For each epoch, the input data, actual output (`y`), and predicted output are printed. This information provides insights into how the neural network is learning and adjusting its weights to minimize the error.
6. **Final Output and Error:**
   - After completing the training process, the final input data, actual output, predicted output, and the error in the predicted output are printed. This helps assess the performance of the neural network on the given binary classification task.

```
-----------Epoch- 1 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.83170693]
 [0.82299861]
 [0.83108381]]
-----------Epoch- 1 Ends----------
-----------Epoch- 2 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.83249847]
 [0.82376119]
 [0.83187046]]
-----------Epoch- 2 Ends----------
```

```
----------Epoch- 4 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.83403481]
 [0.8252421 ]
 [0.8333974 ]]
----------Epoch- 4 Ends----------

----------Epoch- 5 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.83478052]
 [0.82596128]
 [0.83413859]]
----------Epoch- 5 Ends----------
```

-till

-

```
----------Epoch- 5 Ends----------

Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.83478052]
 [0.82596128]
 [0.83413859]]
Final Error in predicted output :
 [[0.08521948]
 [0.03403872]
 [0.05586141]]
```

-

In [16]:
```
import pandas as pd
from sklearn import tree
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB

# Load data from CSV
data = pd.read_csv(r'C:\Users\lenovo\Desktop\tennisdata.csv')
print("THe first 5 values of data is :\n",data.head())
```

```
THe first 5 values of data is :
     Outlook Temperature Humidity  Windy PlayTennis
0      Sunny         Hot     High  False         No
1      Sunny         Hot     High   True         No
2   Overcast         Hot     High  False        Yes
3      Rainy        Mild     High  False        Yes
4      Rainy        Cool   Normal  False        Yes
```

In [18]:
```
# obtain Train data and Train output
X = data.iloc[:,:-1]
print("\nThe First 5 values of train data is\n",X.head())
```

```
The First 5 values of train data is
     Outlook Temperature Humidity  Windy
0      Sunny         Hot     High  False
1      Sunny         Hot     High   True
2   Overcast         Hot     High  False
3      Rainy        Mild     High  False
4      Rainy        Cool   Normal  False
```

In [19]:
```
y = data.iloc[:,-1]
print("\nThe first 5 values of Train output is\n",y.head())
```

```
The first 5 values of Train output is
 0     No
1     No
2    Yes
3    Yes
4    Yes
Name: PlayTennis, dtype: object
```

In [20]:
```
# Convert then in numbers
le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)

le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)

le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)

le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)

print("\nNow the Train data is :\n",X.head())
```

Program 6: Demonstrate the text classifier using Naïve bayes classifier algorithm.

Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

The provided Python code utilizes the Gaussian Naive Bayes (GNB) classifier to perform classification on the famous tennis dataset. The task involves predicting whether tennis can be played based on various weather conditions. The GNB algorithm is particularly well-suited for this task due to its assumption of conditional independence between features, making it efficient for classification problems.

Algorithm:

1. **Loading and Displaying Data:**
   - The code begins by loading the tennis dataset from a CSV file using the pandas library. This dataset contains information about different weather conditions and the corresponding decision to play tennis.
2. **Data Preparation:**
   - The input features (`X`) and target variable (`y`) are extracted from the dataset. Categorical variables are transformed into numerical format using `LabelEncoder`.
3. **Label Encoding:**
   - For each categorical feature, a separate `LabelEncoder` is applied to convert categorical labels into numerical values. This preprocessing step is crucial for training machine learning models.
4. **Transforming Target Variable:**
   - The target variable ('PlayTennis') is encoded using `LabelEncoder` to convert class labels into numerical format (0 and 1).
5. **Train-Test Split:**
   - The dataset is split into training and testing sets using the `train_test_split` function from scikit-learn. This ensures an unbiased evaluation of the model's performance on unseen data.
6. **Model Training:**
   - The Gaussian Naive Bayes classifier is instantiated and trained on the training set using the `fit` method. The GNB algorithm assumes that features are conditionally independent given the class.
7. **Model Evaluation:**
   - The accuracy of the trained model is evaluated using the testing set. The `accuracy_score` function from scikit-learn compares the predicted labels with the true labels, providing a metric for the model's performance.

```
Now the Train data is :
    Outlook  Temperature  Humidity  Windy
0        2            1         0      0
1        2            1         0      1
2        0            1         0      0
3        1            2         0      0
4        1            0         1      0
```

[21]:
```
le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n",y)
```

```
Now the Train output is
 [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
```

[22]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.20)

classifier = GaussianNB()
classifier.fit(X_train,y_train)

from sklearn.metrics import accuracy_score
print("Accuracy is:",accuracy_score(classifier.predict(X_test),y_test))
```

```
Accuracy is: 0.6666666666666666
```

Significance:

1. **Efficiency and Simplicity:**
   - Gaussian Naive Bayes is known for its simplicity and efficiency. It's particularly useful for smaller datasets and problems where the conditional independence assumption holds.
2. **Categorical Data Handling:**
   - GNB handles categorical data well, making it suitable for tasks with categorical features, such as the weather conditions in this tennis dataset.
3. **Interpretability:**
   - The model is interpretable, and the probabilities it outputs can be used to understand the confidence in predictions.

Use Cases:

1. **Weather-Based Decision Making:**
   - The model can be used in scenarios where decisions depend on weather conditions, such as deciding whether outdoor events or activities should proceed.
2. **Customer Behavior Prediction:**
   - In business, the GNB algorithm can be applied to predict customer behavior based on various factors, aiding in targeted marketing strategies.
3. **Email Spam Detection:**
   - GNB is commonly used in email spam detection, where the algorithm can analyze features of an email to predict whether it is spam or not.

In conclusion, the Gaussian Naive Bayes classifier proves to be a valuable tool for certain types of classification tasks, providing efficiency, simplicity, and interpretability. Its application extends to various domains where conditional independence assumptions align with the nature of the data.

Program 7 : Aim : Demonstate and Analyse the results sets obtained from Bayesian belief network Principle.

Write a program to construct a Bayesian network considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard Heart Disease Data Set. You can use Java/Python ML library classes/API.

```
from pgmpy.models import BayesianModel
model=BayesianModel([
('age','Lifestyle'),
('Gender','Lifestyle'),
('Family','heartdisease'),
('diet','cholestrol'),
('Lifestyle','diet'),
('cholestrol','heartdisease'),
('diet','cholestrol')
])

from pgmpy.estimators import MaximumLikelihoodEstimator
model.fit(heart_disease, estimator=MaximumLikelihoodEstimator)

from pgmpy.inference import VariableElimination
HeartDisease_infer = VariableElimination(model)
```

```
import pandas as pd
data=pd.read_csv(r"C:\Users\lenovo\Desktop\heartdisease.csv")
heart_disease=pd.DataFrame(data)
print(heart_disease)
```

```
    age  Gender  Family  diet  Lifestyle  cholestrol  heartdisease
0     0       0       1     1          3           0             1
1     0       1       1     1          3           0             1
2     1       0       0     0          2           1             1
3     4       0       1     1          3           2             0
4     3       1       1     0          0           2             0
5     2       0       1     1          1           0             1
6     4       0       1     0          2           0             1
7     0       0       1     1          3           0             1
8     3       1       1     0          0           2             0
9     1       1       0     0          0           2             1
10    4       1       0     1          2           0             1
11    4       0       1     1          3           2             0
12    2       1       0     0          0           0             0
13    2       0       1     1          1           0             1
14    3       1       1     0          0           1             0
15    0       0       1     0          0           2             1
16    1       1       0     1          2           1             1
17    3       1       1     1          0           1             0
18    4       0       1     1          3           2             0
```

```
print('For age Enter { SuperSeniorCitizen:0, SeniorCitizen:1, MiddleAged:2, Youth:3, Teen:4 }')
print('For Gender Enter { Male:0, Female:1 }')
print('For Family History Enter { yes:1, No:0 }')
print('For diet Enter { High:0, Medium:1 }')
print('For lifeStyle Enter { Athlete:0, Active:1, Moderate:2, Sedentary:3 }')
print('For cholesterol Enter { High:0, BorderLine:1, Normal:2 }')

q = HeartDisease_infer.query(variables=['heartdisease'], evidence={
    'age':int(input('Enter age :')),
    'Gender':int(input('Enter Gender :')),
    'Family':int(input('Enter Family history :')),
    'diet':int(input('Enter diet :')),
    'Lifestyle':int(input('Enter Lifestyle :')),
    'cholestrol':int(input('Enter cholestrol :'))
    })
print(q)
```

```
For age Enter { SuperSeniorCitizen:0, SeniorCitizen:1, MiddleAged:2, Youth:3, Teen:4 }
For Gender Enter { Male:0, Female:1 }
For Family History Enter { yes:1, No:0 }
For diet Enter { High:0, Medium:1 }
For lifeStyle Enter { Athlete:0, Active:1, Moderate:2, Sedentary:3 }
For cholesterol Enter { High:0, BorderLine:1, Normal:2 }
Enter age :3
Enter Gender :0
Enter Family history :0
Enter diet :1
Enter Lifestyle :0
Enter cholestrol :1
```

heartdisease	phi(heartdisease)
heartdisease(0)	0.0000
heartdisease(1)	1.0000

Program 8: Bayesian belief network Principle.

Write a Program to implement k-Nearest algorithm to classify; the Iris data set. Print both correct and wrong predicitons.

```
import numpy as np
import pandas as pd
```

```
dataset = pd.read_csv(r'C:\Users\lenovo\Desktop\Iris.csv')
```

```
dataset.shape
```

```
(150, 6)
```

```
dataset.head(5)
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

```
dataset.describe()
```

```
dataset.describe()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	75.500000	5.843333	3.054000	3.758667	1.198667
std	43.445368	0.828066	0.433594	1.764420	0.763161
min	1.000000	4.300000	2.000000	1.000000	0.100000
25%	38.250000	5.100000	2.800000	1.600000	0.300000
50%	75.500000	5.800000	3.000000	4.350000	1.300000
75%	112.750000	6.400000	3.300000	5.100000	1.800000
max	150.000000	7.900000	4.400000	6.900000	2.500000

```
dataset.groupby('Species').size()
```

```
Species
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
dtype: int64
```

```
feature_columns = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm','PetalWidthCm']
X = dataset[feature_columns].values
y = dataset['Species'].values
```

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import cross_val_score
```

```
classifier = KNeighborsClassifier(n_neighbors=3)
```

```
classifier.fit(X_train, y_train)
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

```
y_pred = classifier.predict(X_test)
```

```
cm = confusion_matrix(y_test, y_pred)
cm
```

```
array([[11,  0,  0],
       [ 0, 12,  1],
       [ 0,  0,  6]], dtype=int64)
```

```
accuracy = accuracy_score(y_test, y_pred)*100
print('Accuracy of our model is equal ' + str(round(accuracy, 2)) + ' %.')
```

```
Accuracy of our model is equal 96.67 %.
```

[20]:
```
from sklearn.metrics import classification_report,confusion_matrix
```

[21]:
```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      0.92      0.96        13
           2       0.86      1.00      0.92         6

    accuracy                           0.97        30
   macro avg       0.95      0.97      0.96        30
weighted avg       0.97      0.97      0.97        30
```

Confusion matrix results needs to be calculated.

K Nearest Neighbor

1. KNN Theory

1.1 Type of algorithm

KNN can be used for both classification and regression predictive problems. KNN falls in the supervised learning family of algorithms. Informally, this means that we are given a labelled dataset consiting of training observations (x, y) and would like to capture the relationship between x and y. More formally, our goal is to learn a function $h : X \rightarrow Y$ so that given an unseen observation x, $h(x)$ can confidently predict the corresponding output y.

1.2 Distance measure

In the classification setting, the K-nearest neighbor algorithm essentially boils down to forming a majority vote between the K most similar instances to a given "unseen" observation. Similarity is defined according to a distance metric between two data points. The k-nearest-neighbor classifier is commonly based on the Euclidean distance between a test sample and the specified training samples. Let x_i be an input sample with p features $(x_{i1}, x_{i2}, \ldots, x_{ip})$, n be the total number of input samples $(i = 1, 2, \ldots, n)$. The Euclidean distance between sample x_i and x_l is is defined as:

$$d(x_i, x_l) = \sqrt{(x_{i1} - x_{l1})^2 + (x_{i2} - x_{l2})^2 + \ldots + (x_{ip} - x_{lp})^2}$$

Sometimes other measures can be more suitable for a given setting and include the Manhattan, Chebyshev and Hamming distance.

1.3 Algorithm steps

STEP 1: Cgoose the number K of neighbors

STEP 2: Take the K nearest neighbors of the new data point, according to your distance metric

STEP 3: Among these K neighbors, count the number of data points to each category

STEP 4: Assign the new data point to the category where you counted the most neighbors

2. Importing and preperation of data

2.1 Import libraries

In [1]:
```
import numpy as np
import pandas as pd
```

2.2 Load dataset

NOTE: Iris dataset 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.

In [2]:
```
# Importing the dataset
dataset = pd.read_csv('../input/Iris.csv')
```

2.3 Summarize the Dataset

In [3]:
```
# We can get a quick idea of how many instances (rows) and how many at
tributes (columns) the data contains with the shape property.
dataset.shape
```

Out[3]:
```
(150, 6)
```

In [4]:
```
dataset.head(5)
```

Out[4]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

In [5]:
```
dataset.describe()
```

Out[5]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	75.500000	5.843333	3.054000	3.758667	1.198667
std	43.445368	0.828066	0.433594	1.764420	0.763161
min	1.000000	4.300000	2.000000	1.000000	0.100000
25%	38.250000	5.100000	2.800000	1.600000	0.300000
50%	75.500000	5.800000	3.000000	4.350000	1.300000
75%	112.750000	6.400000	3.300000	5.100000	1.800000
max	150.000000	7.900000	4.400000	6.900000	2.500000

In [6]:
```
# Let's now take a look at the number of instances (rows) that belong
to each class. We can view this as an absolute count.
dataset.groupby('Species').size()
```

Out[6]:
```
Species
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
dtype: int64
```

2.4 Dividing data into features and labels

NOTE: As we can see dataset contain six columns: Id, SepalLengthCm, SepalWidthCm, PetalLengthCm, PetalWidthCm and Species. The actual features are described by columns 1-4. Last column contains labels of samples. Firstly we need to split data into two arrays: X (features) and y (labels).

In [7]:
```
feature_columns = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthC
m','PetalWidthCm']
X = dataset[feature_columns].values
y = dataset['Species'].values

# Alternative way of selecting features and labels arrays:
# X = dataset.iloc[:, 1:5].values
# y = dataset.iloc[:, 5].values
```

In [8]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
```

2.6 Spliting dataset into training set and test set

Let's split dataset into training set and test set, to check later on whether or not our classifier works correctly.

In [9]:
```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.2, random_state = 0)
```

4.1. Making predictions

In [16]:
```
# Fitting clasifier to the Training set
# Loading libraries
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import cross_val_score

# Instantiate learning model (k = 3)
classifier = KNeighborsClassifier(n_neighbors=3)

# Fitting the model
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)
```

Building confusion matrix:

In [17]:
```
cm = confusion_matrix(y_test, y_pred)
cm
```

Out[17]:
```
array([[11,  0,  0],
       [ 0, 12,  1],
       [ 0,  0,  6]])
```

In [18]:
```
accuracy = accuracy_score(y_test, y_pred)*100
print('Accuracy of our model is equal ' + str(round(accuracy, 2)) +
' %.')
```

```
Accuracy of our model is equal 96.67 %.
```

The most commonly calculated statistical measures

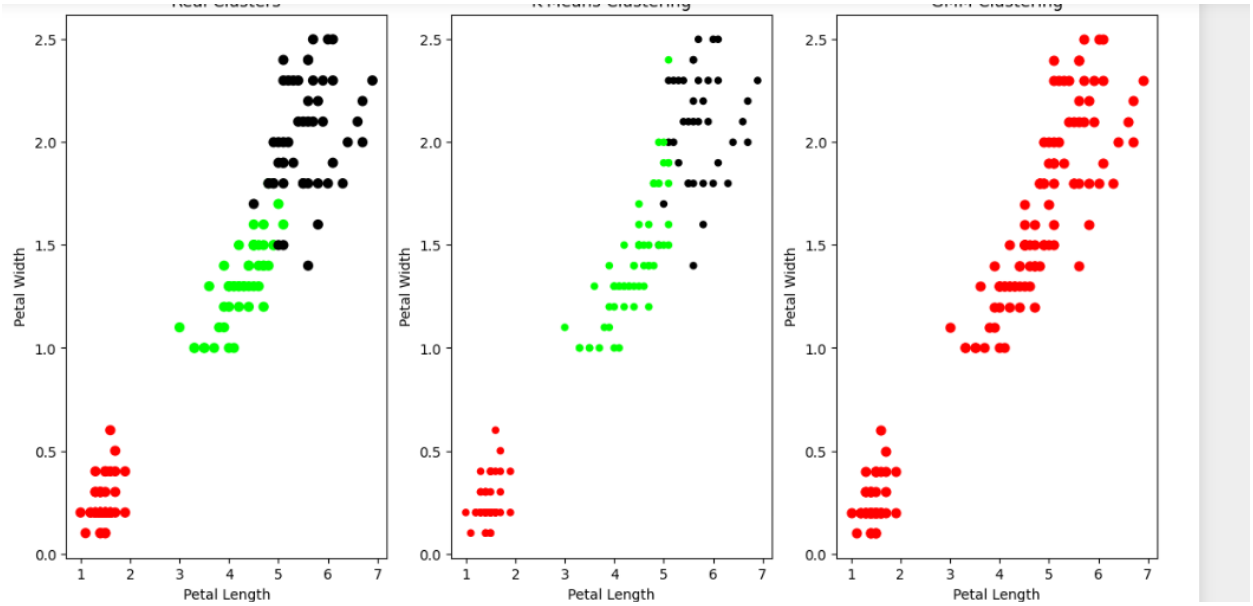
Measure	Formula
Sensitivity	TPR = TP / (TP + FN)
Specificity	SPC = TN / (FP + TN)
Positive Predictive Value (Precision)	PPV = TP / (TP + FP)
Negative Predictive Value	NPV = TN / (TN + FN)
False Positive Rate	FPR = FP / (FP + TN)
False Discovery Rate	FDR = FP / (FP + TP)
False Negative Rate	FNR = FN / (FN + TP)
Accuracy	ACC = (TP + TN) / (TP + TN + FP + FN)
F1 Score	F1 = 2TP / (2TP + FP + FN)
Matthews Correlation Coefficient	MCC = (TP x TN – FP x FN) / (sqrt((TP + FP) x (TP + FN) x (TN + FP) x (TN + FN)))

Program 9: Implement and demonstate the working model o K-means clustering algorithm with Expectation Maximization Concept.

Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

In [1]:
```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import pandas as pd
import numpy as np

# import some data to play with
iris = datasets.load_iris()
X = pd.DataFrame(iris.data)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']
y = pd.DataFrame(iris.target)
y.columns = ['Targets']

# Build the K Means Model
model = KMeans(n_clusters=3)
model.fit(X) # model.labels_ : Gives cluster no for which samples belongs to

# # Visualise the clustering results
plt.figure(figsize=(14,7))
colormap = np.array(['red', 'lime', 'black'])

# Plot the Original Classifications using Petal features
plt.subplot(1, 3, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
```

```
# Plot the Models Classifications
plt.subplot(1, 3, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K-Means Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

# General EM for GMM
from sklearn import preprocessing

# transform your data such that its distribution will have a # mean value 0 and standard deviation of 1.
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=40)
gmm.fit(xs)
plt.subplot(1, 3, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[0], s=40)
plt.title('GMM Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

print('Observation: The GMM using EM algorithm based clustering matched the true labels more closely than the Kmeans.')
```


Petal Width
Petal Length
Petal Width
Petal Length
Petal Width
Petal Length