Week1)**Week 1 Dynamic BFS and DFS Traversal problem**

graph = {}

edge\_set = set()

# Add a node only if it doesn't already exist

def add\_node(node):

    if node in graph:

        print(f"'{node}' already exists. Please enter a different node.")

        return False

    graph[node] = []

    return True

# Add edge only if not a duplicate

def add\_edge(u, v):

    edge = tuple(sorted((u, v)))

    if edge in edge\_set:

        print(f"Edge {u}-{v} already exists. Please enter a different edge.")

        return False

    if u not in graph or v not in graph:

        print("Both nodes must be added before connecting them with an edge.")

        return False

    graph[u].append(v)

    graph[v].append(u)

    edge\_set.add(edge)

    return True

# BFS

def bfs(start):

    visited = []

    queue = [start]

    print("BFS:", end=" ")

    while queue:

        node = queue.pop(0)

        if node not in visited:

            print(node, end=" ")

            visited.append(node)

            for neighbor in graph[node]:

                if neighbor not in visited:

                    queue.append(neighbor)

    print()

# DFS

def dfs(node, visited=None):

    if visited is None:

        visited = []

        print("DFS:", end=" ")

    if node not in visited:

        print(node, end=" ")

        visited.append(node)

        for neighbor in graph[node]:

            dfs(neighbor, visited)

# === Input Section ===

# Add unique nodes

n = int(input("Enter number of nodes: "))

i = 0

while i < n:

    node = input(f"Enter node {i + 1}: ").strip()

    if add\_node(node):

        i += 1

# Add edges without duplication

e = int(input("Enter number of edges: "))

for i in range(e):

    while True:

        u, v = input(f"Enter edge {i + 1} (two nodes): ").split()

        if add\_edge(u, v):

            break

# Start traversal

start = input("Enter starting node: ").strip()

if start in graph:

    bfs(start)

    dfs(start)

    print()

else:

    print("Starting node not found in the graph.")

**OUT PUT:**

Enter number of nodes: 7

Enter node 1: 5 Enter node 2: 2

Enter node 3: 3 Enter node 4: 4

Enter node 5: 8 Enter node 6: 6

Enter node 7: 3

'3' already exists. Please enter a different node.

Enter node 7: 7

Enter number of edges: 6

Enter edge 1 (two nodes): 5 2

Enter edge 2 (two nodes): 5 3

Enter edge 3 (two nodes): 2 4

Enter edge 4 (two nodes): 2 4

Edge 2-4 already exists. Please enter a different edge.

Enter edge 4 (two nodes): 2 8

Enter edge 5 (two nodes): 3 6

Enter edge 6 (two nodes): 8 7

Enter starting node: 5

BFS: 5 2 3 4 8 6 7

DFS: 5 2 4 8 7 3 6

WEEK2)Week2 A \* **algorithm Implementation**

def aStarAlgo(start\_node, stop\_node):

    open\_set = {start\_node}  # Set of nodes to be evaluated

    closed\_set = set()  # Set of nodes already evaluated

    g = {}  # Dictionary to store the distance from the start node

    parents = {}  # Dictionary to store the parent of each node

    # The distance from the start node to itself is zero

    g[start\_node] = 0

    # The start node has no parent (it is the root)

    parents[start\_node] = start\_node

    while open\_set:

        n = None

        # Node with the lowest f() = g + heuristic() value is chosen

        for v in open\_set:

            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):

                n = v

        # Print current node being evaluated, heuristic value and the sets

        print(f"\nEvaluating node: {n} (g: {g[n]}, h: {heuristic(n)}, f: {g[n] + heuristic(n)})")

        print(f"Open Set: {open\_set}")

        print(f"Closed Set: {closed\_set}")

        # If the goal is reached or no more nodes can be explored

        if n == stop\_node:

            path = []

            while parents[n] != n:

                path.append(n)

                n = parents[n]

            path.append(start\_node)

            path.reverse()  # Reverse the path to get from start to goal

            print('Path found: {}'.format(path))

            return path

        # Explore neighbors of the current node

        print(f"Exploring neighbors of {n}:")

        for (m, weight) in get\_neighbors(n):

            h\_m = heuristic(m)  # Heuristic value of the neighbor

            print(f"  Neighbor: {m} with weight: {weight} and h({m}): {h\_m}")

            if m not in open\_set and m not in closed\_set:

                open\_set.add(m)

                parents[m] = n

                g[m] = g[n] + weight

                print(f"  Added {m} to open set with g({m}) = {g[m]} and f({m}) = {g[m] + h\_m}")

            else:

                if g[m] > g[n] + weight:

                    g[m] = g[n] + weight

                    parents[m] = n

                    if m in closed\_set:

                        closed\_set.remove(m)

                        open\_set.add(m)

                        print(f"  Updated {m} to have a shorter path with g({m}) = {g[m]} and f({m}) = {g[m] + h\_m}")

        open\_set.remove(n)

        closed\_set.add(n)

    print('Path does not exist!')

    return None

# Function to return neighbors and their distances

def get\_neighbors(v):

    if v in Graph\_nodes:

        return Graph\_nodes[v]

    return []

# Heuristic function for each node (Manhattan or other heuristic values)

def heuristic(n):

    H\_dist = {

        'S': 5,

        'A': 3,

        'B': 4,

        'C': 2,

        'D': 6,

        'G': 0,

    }

    return H\_dist.get(n, 0)

# Graph representation (node -> list of (neighbor, weight))

Graph\_nodes = {

    'S': [('A', 1), ('G', 10)],

    'A': [('B', 2), ('C', 1)],

    'B': [('D', 5)],

    'C': [('D', 3), ('G', 4)],

    'D': [('G', 2)],

}

# Run the algorithm

aStarAlgo('S', 'G')

OUTPUT:

Evaluating node: S (g: 0, h: 5, f: 5)

Open Set: {'S'}

Closed Set: set()

Exploring neighbors of S:

Neighbor: A with weight: 1 and h(A): 3

Added A to open set with g(A) = 1 and f(A) = 4

Neighbor: G with weight: 10 and h(G): 0

Added G to open set with g(G) = 10 and f(G) = 10

Evaluating node: A (g: 1, h: 3, f: 4)

Open Set: {'G', 'A'}

Closed Set: {'S'}

Exploring neighbors of A:

Neighbor: B with weight: 2 and h(B): 4

Added B to open set with g(B) = 3 and f(B) = 7

Neighbor: C with weight: 1 and h(C): 2

Added C to open set with g(C) = 2 and f(C) = 4

Evaluating node: C (g: 2, h: 2, f: 4)

Open Set: {'B', 'G', 'C'}

Closed Set: {'S', 'A'}

Exploring neighbors of C:

Neighbor: D with weight: 3 and h(D): 6

Added D to open set with g(D) = 5 and f(D) = 11

Neighbor: G with weight: 4 and h(G): 0

Evaluating node: G (g: 6, h: 0, f: 6)

Open Set: {'B', 'G', 'D'}

Closed Set: {'C', 'S', 'A'}

Path found: ['S', 'A', 'C', 'G']

['S', 'A', 'C', 'G']

WEEK3)**Travelling Salesman Problem Implementaion using Python**

from sys import maxsize  # Import maxsize to represent an infinitely large value (for comparison)

from itertools import permutations  # Import permutations to generate all possible orders of cities

v = 4  # Number of cities (vertices), here v = 4 (cities labeled 0, 1, 2, 3)

# Function to find the shortest path for the Traveling Salesman Problem

def travellingSalesmanProblem(graph, s):

    vertex = []  # List to store all cities excluding the start city 's'

    # Loop through all the cities and add those that are not the starting city 's'

    for i in range(v):

        if i != s:

            vertex.append(i)  # Add city i to the list 'vertex' if it's not the start city

    # Initialize min\_path to a very large value (infinity), so it can be updated with valid paths

    min\_path = maxsize

    # Generate all possible permutations of the cities in 'vertex'

    # This will give all possible orders of visiting the cities

    next\_permutation = permutations(vertex)

    # Iterate through each permutation of cities

    for i in next\_permutation:

        current\_pathweight = 0  # Initialize the total distance of the current path

        k = s  # Start at the initial city (starting city is 's')

        # Loop through the cities in the current permutation

        for j in i:

            current\_pathweight += graph[k][j]  # Add the distance from the current city 'k' to the next city 'j'

            k = j  # Update the current city to the next city

        # After visiting all cities, add the distance to return to the starting city

        current\_pathweight += graph[k][s]  # Add the distance from the last city back to the start city

        # Update min\_path if the current path weight is smaller than the previous min\_path

        min\_path = min(min\_path, current\_pathweight)

    # Return the shortest path found after evaluating all permutations

    return min\_path

# Example graph where the value graph[i][j] represents the distance from city 'i' to city 'j'

graph = [

    [0, 10, 15, 20],  # Distances from city 0 to others

    [10, 0, 35, 25],  # Distances from city 1 to others

    [15, 35, 0, 30],  # Distances from city 2 to others

    [20, 25, 30, 0]   # Distances from city 3 to others

]

s = 0  # Set the starting city (index 0, which is city 0)

# Call the function and print the result

print(travellingSalesmanProblem(graph, s))  # Output: 80

OUTPUT:

80

WEEK3)**Graph Coloring Problem Implementaion using Python**

Graph Coloring Problem

colors=['Red','Blue','Green']

states=['a','b','c','d']

neighbors={}

neighbors['a']=['b','c','d']

neighbors['b']=['a','d']

neighbors['c']=['a','d']

neighbors['d']=['c','b','a']

colors\_of\_states={}

def promising(state,color):#d,green

    for neighbor in neighbors.get(state):#c,b,a

        color\_of\_neighbor=colors\_of\_states.get(neighbor)#blue

        if color\_of\_neighbor==color:#b==b

            return False

    return True

def get\_color\_for\_state(state):#d

    for color in colors:#Red,Blue,Green

        if promising(state,color):#d,Red

            return color

def main():

    for state in states:#c,d

        colors\_of\_states[state]=get\_color\_for\_state(state)#a:Red,b:blue,c:blue,d:green

    print(colors\_of\_states)

main()

OUTPUT:

{'a': 'Red', 'b': 'Blue', 'c': 'Blue', 'd': 'Green'}

**Week 4) Implementation of Knowledge Representation**

from sympy import symbols, Or, Not, Implies,Xor,satisfiable

# Define propositional variables

Rain = symbols('Rain')

Harry\_Visited\_Hagrid = symbols('Harry\_Visited\_Hagrid')

Harry\_Visited\_Dumbledore = symbols('Harry\_Visited\_Dumbledore')

# Define the logical expressions based on the given statements

sentence\_1 = Implies(Not(Rain), Harry\_Visited\_Hagrid)

sentence\_2 = Xor(Harry\_Visited\_Hagrid, Harry\_Visited\_Dumbledore)

sentence\_3 = Harry\_Visited\_Dumbledore

# Combine the statements

knowledge\_base = sentence\_1 & sentence\_2 & sentence\_3

#Finding the solution

solution = satisfiable(knowledge\_base, all\_models=True)

#To print the output

for model in solution:

    if model[Rain]:

        print("It rained today.")

    else:

        print("There is no rain today.")

**OUTPUT:**

It rained today.

**Week 5) Implementation of Bayesian Network.**

#week5

#Bayesian Network

# Define conditional probability tables (CPTs)

P\_burglary = 0.002#t

P\_earthquake = 0.001#t

# Probability of alarm given burglary and earthquake

P\_alarm\_given\_burglary\_and\_earthquake = 0.94

P\_alarm\_given\_burglary\_and\_no\_earthquake = 0.95

P\_alarm\_given\_no\_burglary\_and\_earthquake = 0.31

P\_alarm\_given\_no\_burglary\_and\_no\_earthquake = 0.001

# Probability of David calling given alarm

P\_david\_calls\_given\_alarm = 0.91#t

P\_david\_does\_not\_call\_given\_alarm = 0.09

P\_david\_calls\_given\_no\_alarm = 0.05#t

P\_david\_does\_not\_call\_given\_no\_alarm = 0.95

# Probability of Sophia calling given alarm

P\_sophia\_calls\_given\_alarm = 0.75

P\_sophia\_does\_not\_call\_given\_alarm = 0.25

P\_sophia\_calls\_given\_no\_alarm = 0.02

P\_sophia\_does\_not\_call\_given\_no\_alarm = 0.98

# Calculate joint probability

def joint\_probability(alarm, burglary, earthquake, david\_calls, sophia\_calls):#(t,f,f,t,t)

    if alarm:

        if burglary and earthquake:

            P\_alarm = P\_alarm\_given\_burglary\_and\_earthquake

        elif burglary:

            P\_alarm = P\_alarm\_given\_burglary\_and\_no\_earthquake

        elif earthquake:

            P\_alarm = P\_alarm\_given\_no\_burglary\_and\_earthquake

        else:

            P\_alarm = P\_alarm\_given\_no\_burglary\_and\_no\_earthquake#0.001

    else:

        if burglary and earthquake:

            P\_alarm = 1 - P\_alarm\_given\_burglary\_and\_earthquake

        elif burglary:

            P\_alarm = 1 - P\_alarm\_given\_burglary\_and\_no\_earthquake

        elif earthquake:

            P\_alarm = 1 - P\_alarm\_given\_no\_burglary\_and\_earthquake

        else:

            P\_alarm = 1 - P\_alarm\_given\_no\_burglary\_and\_no\_earthquake

    P\_david = (P\_david\_calls\_given\_alarm if david\_calls else P\_david\_does\_not\_call\_given\_alarm) if alarm else (P\_david\_calls\_given\_no\_alarm if david\_calls else P\_david\_does\_not\_call\_given\_no\_alarm)#0.91

    P\_sophia = (P\_sophia\_calls\_given\_alarm if sophia\_calls else P\_sophia\_does\_not\_call\_given\_alarm) if alarm else (P\_sophia\_calls\_given\_no\_alarm if sophia\_calls else P\_sophia\_does\_not\_call\_given\_no\_alarm)#0.75

    return (P\_burglary if burglary else 1 - P\_burglary) \* (P\_earthquake if earthquake else 1 - P\_earthquake) \* P\_alarm \* P\_david \* P\_sophia#0.75\*0.91\*0.001\*0.998\*0.999

# Calculate the probability for the given scenario

result = joint\_probability(

    alarm=True,

    burglary=False,

    earthquake=False,

    david\_calls=True,

    sophia\_calls=True

)

# Print the result

print(f'The probability that the alarm has sounded, there is neither a burglary nor an earthquake, and both David and Sophia called Harry is: {result:.8f}')

**OUTPUT:**

The probability that the alarm has sounded, there is neither a burglary nor an earthquake, and both David and Sophia called Harry is: 0.00068045

**Week 6 )To implement Hidden Markov Model**

import numpy as np

import itertools

import pandas as pd

# Define state space and probabilities

states = ['sleeping', 'eating', 'pooping']

hidden\_states = ['healthy', 'sick']

pi = [0.5, 0.5]  # Initial state probabilities

# Initial state distribution

state\_space = pd.Series(pi, index=hidden\_states, name='states')

print("Initial Probabilities:\n", state\_space, "\n")

# Transition probabilities (hidden -> hidden)

a\_df = pd.DataFrame(columns=hidden\_states, index=hidden\_states)

a\_df.loc['healthy'] = [0.7, 0.3]

a\_df.loc['sick'] = [0.4, 0.6]

print("Transition Probabilities:\n", a\_df, "\n")

# Emission probabilities (hidden -> observable)

b\_df = pd.DataFrame(columns=states, index=hidden\_states)

b\_df.loc['healthy'] = [0.2, 0.6, 0.2]

b\_df.loc['sick'] = [0.4, 0.1, 0.5]

print("Emission Probabilities:\n", b\_df, "\n")

# Forward algorithm: Total probability of observation sequence

def forward\_algorithm(obs\_seq, a\_df, b\_df, pi, hidden\_states):

    total\_prob = 0

    all\_state\_paths = list(itertools.product(hidden\_states, repeat=len(obs\_seq)))

    for path in all\_state\_paths:

        prob = pi[hidden\_states.index(path[0])] \* b\_df.loc[path[0], obs\_seq[0]]

        for t in range(1, len(obs\_seq)):

            prev\_state = path[t - 1]

            curr\_state = path[t]

            prob \*= a\_df.loc[prev\_state, curr\_state] \* b\_df.loc[curr\_state, obs\_seq[t]]

        total\_prob += prob

    return total\_prob

# Viterbi algorithm: Most likely hidden state sequence

def viterbi\_algorithm(obs\_seq, a\_df, b\_df, pi, hidden\_states):

    max\_prob = 0

    best\_path = None

    all\_state\_paths = list(itertools.product(hidden\_states, repeat=len(obs\_seq)))

    for path in all\_state\_paths:

        prob = pi[hidden\_states.index(path[0])] \* b\_df.loc[path[0], obs\_seq[0]]

        for t in range(1, len(obs\_seq)):

            prev\_state = path[t - 1]

            curr\_state = path[t]

            prob \*= a\_df.loc[prev\_state, curr\_state] \* b\_df.loc[curr\_state, obs\_seq[t]]

        if prob > max\_prob:

            max\_prob = prob

            best\_path = path

    return max\_prob, best\_path

# Example observation sequence

obsq = ['sleeping', 'eating', 'sleeping']

# Run and print

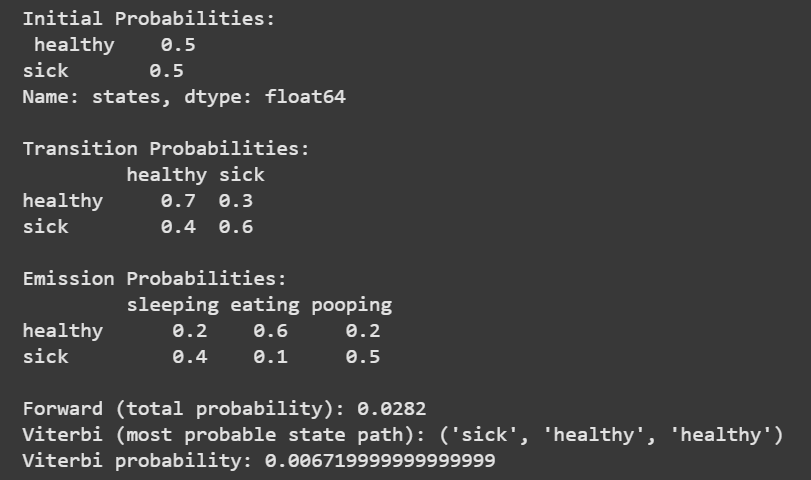
print("Forward (total probability):", forward\_algorithm(obsq, a\_df, b\_df, pi, hidden\_states))

v\_prob, v\_path = viterbi\_algorithm(obsq, a\_df, b\_df, pi, hidden\_states)

print("Viterbi (most probable state path):", v\_path)

print("Viterbi probability:", v\_prob)

**OUTPUT:**



**Week 7 )To implement Regression algorithm**

# Importing the libraries

import numpy as np

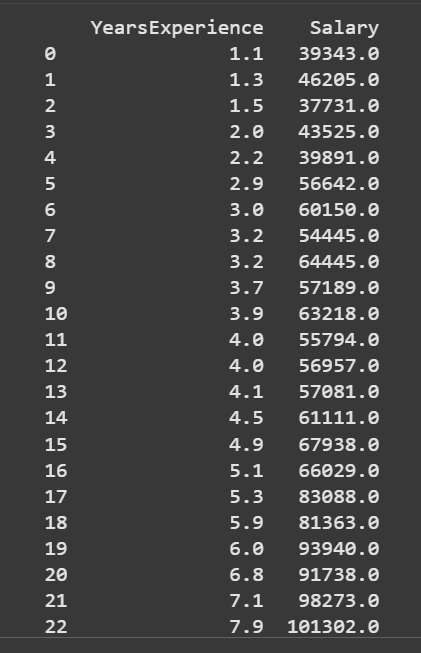
import pandas as pd

import matplotlib.pyplot as plt

# Importing the dataset

dataset = pd.read\_csv('/salary\_data.csv')

print(dataset)



# data preprocessing

X = dataset.iloc[:, :-1].values  #independent variable array

y = dataset.iloc[:,1].values  #dependent variable vector

# Splitting the dataset into the Training set and Test set

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=1/3,random\_state=0)

# fitting the regression model

from sklearn.linear\_model import LinearRegression

regressor = LinearRegression()

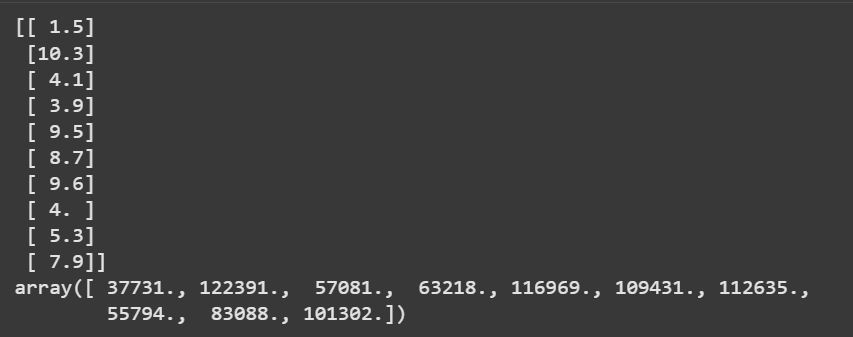
regressor.fit(X\_train,y\_train) #actually produces the linear eqn for the data

# predicting the test set results

y\_pred = regressor.predict(X\_test)

print(X\_test)

y\_test



# visualizing the results

#plot for the TRAIN

plt.scatter(X\_train, y\_train, color='red') # plotting the observation line

plt.plot(X\_train, regressor.predict(X\_train), color='blue') # plotting the regression line

plt.title("Salary vs Experience (Training set)") # stating the title of the graph

plt.xlabel("Years of experience") # adding the name of x-axis

plt.ylabel("Salaries") # adding the name of y-axis

plt.show() # specifies end of graph



#plot for the TEST

plt.scatter(X\_test, y\_test, color='red')

plt.plot(X\_train, regressor.predict(X\_train), color='blue') # plotting the regression line

plt.title("Salary vs Experience (Testing set)")# stating the title of the graph

plt.xlabel("Years of experience")# adding the name of x-axis

plt.ylabel("Salaries")# adding the name of y-axis

plt.show()# specifies end of graph



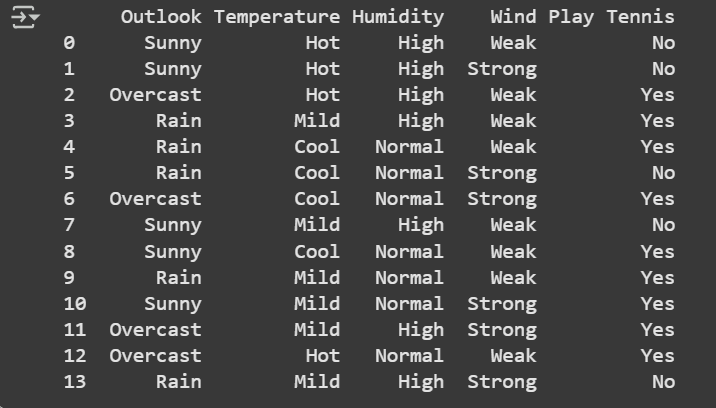
**Week 8 To implement decision tree based ID3 algorithm**

#Import Play Tennis Data

import pandas as pd

df=pd.read\_csv("/content/PlayTennis.csv")

print(df)



#Function to calculate the entropy of probaility of observations

# -p\*log2\*p

def entropy(probs):

 import math

 return sum(-prob\*math.log(prob,2) for prob in probs)

#Function to calulate the entropy of the given Data Sets/List with respect to target attributes

def entropy\_of\_list(a\_list):

 from collections import Counter

 cnt = Counter (x for x in a\_list) # Counter calculates the propotion of class

 print(cnt)

 num\_instances =len(a\_list)

 probs=[x/num\_instances for x in cnt.values()]

 print(num\_instances)

 print(probs)

 return entropy(probs)

total\_entropy= entropy\_of\_list(df['Play Tennis'])

print(total\_entropy)

Output

Counter({'Yes': 9, 'No': 5}) 14 [0.35714285714285715, 0.6428571428571429] 0.9402859586706309

def information\_gain(df,split\_attribute\_name, target\_attribute\_name, trace=0):

  df\_split =df.groupby(split\_attribute\_name)

  print(df\_split)

  for name,group in df\_split:

    print("Name",name)

    print("Group",group)

    nobs=len(df.index)\*1.0

    print(nobs)

    print("NOBS",nobs)

    df\_agg\_ent=df\_split.agg({target\_attribute\_name: [entropy\_of\_list,lambda x: len(x)/nobs] })[target\_attribute\_name]

    avg\_info=sum(df\_agg\_ent['entropy\_of\_list'] \* df\_agg\_ent['<lambda\_0>'])

    old\_entropy=entropy\_of\_list(df[target\_attribute\_name])

    return old\_entropy-avg\_info

def id3DT(df, target\_attribute\_name, attribute\_names, default\_class=None):

  from collections import Counter

  cnt = Counter(x for x in df[target\_attribute\_name])

  if len(cnt)==1:

     return next(iter(cnt))

  elif df.empty or (not attribute\_names):

     return default\_class

  else:

     default\_class =max(cnt.keys())

#print("attributes\_names:",attribute\_names)

     gainz=[information\_gain(df,attr, target\_attribute\_name) for attr in attribute\_names]

     index\_of\_max=gainz.index(max(gainz))

     best\_attr=attribute\_names[index\_of\_max]

     tree={best\_attr:{}}

     remaining\_attributes\_names=[i for i in attribute\_names if i != best\_attr]

     for attr\_val, data\_subset in df.groupby(best\_attr):

      subtree=id3DT(data\_subset,target\_attribute\_name,remaining\_attributes\_names,default\_class)

      tree[best\_attr][attr\_val]=subtree

     return tree

# Predicting Attributes

attribute\_names = list(df)

print("List of Attributes:", attribute\_names)

attribute\_names.remove('Play Tennis') #Remove the class attribute

print("Predicting Attributes:", attribute\_names)

Out put

List of Attributes: ['Outlook', 'Temperature', 'Humidity', 'Wind', 'Play Tennis'] Predicting Attributes: ['Outlook', 'Temperature', 'Humidity', 'Wind']

from pprint import pprint

tree= id3DT(df,'Play Tennis',attribute\_names)

print("The Resultant Decision Tree is ")

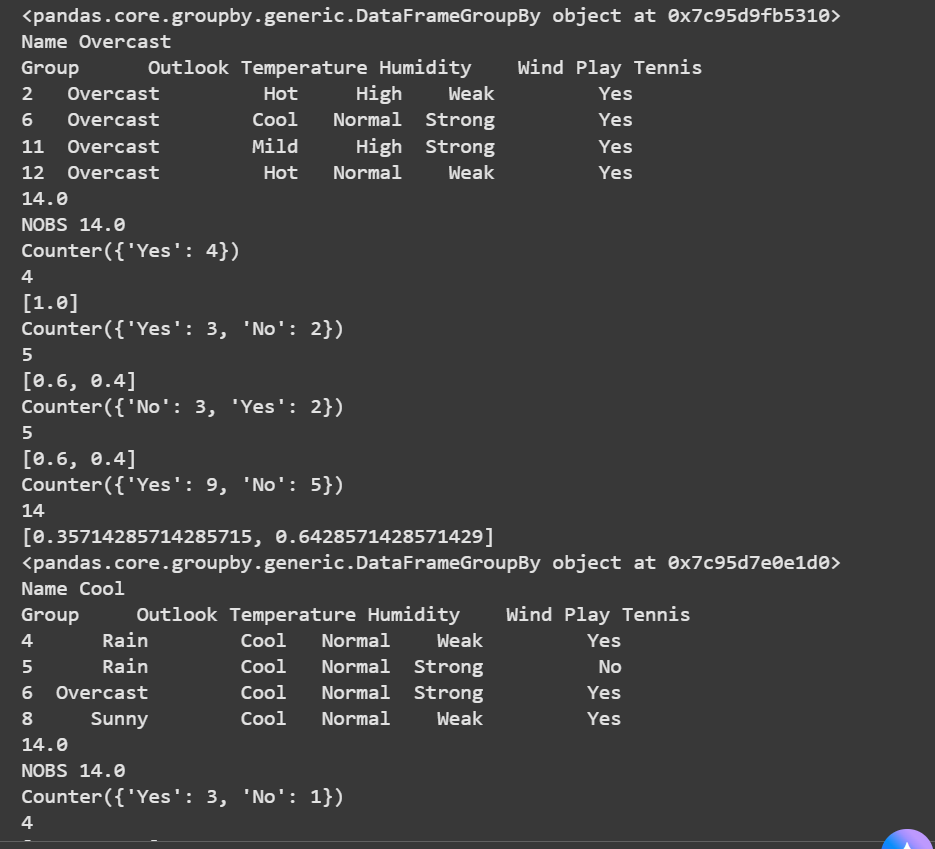
pprint(tree)

attribute=next(iter(tree))

print("Best Attribute: \n", attribute)

print("Tree Keys\n ", tree[attribute].keys())

Output:



def classify(instance, tree, default=None):

  attribute=next(iter(tree))

  print("Key:",tree.keys())

  print("Attribute",attribute)

  if instance[attribute] in tree[attribute].keys():

    result=tree[attribute][instance[attribute]]

    print("Instance Attribute:",instance[attribute], "TreeKeys:",tree[attribute].keys())

    if isinstance(result,dict):

       return classify(instance,result)

    else:

       return result

  else:

    return default

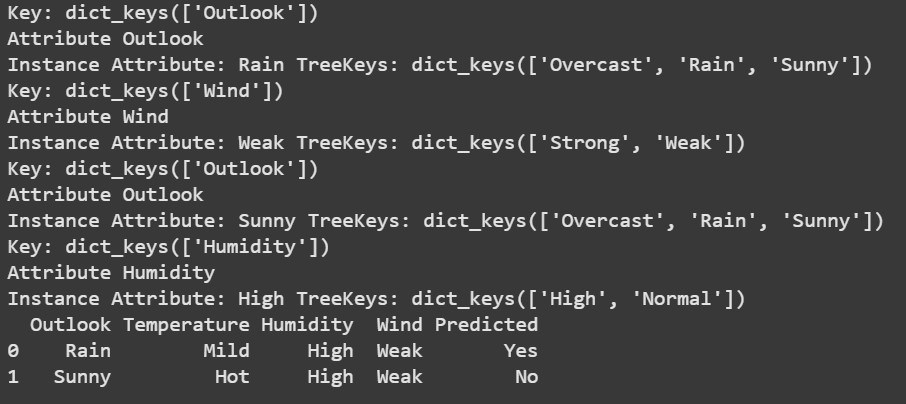
tree1={'Outlook':['Rain','Sunny'],'Temperature':['Mild','Hot'],'Humidity':['High','High'],'Wind':['Weak','Weak']}

df2=pd.DataFrame(tree1)

df2['Predicted']=df2.apply(classify,axis=1, args=(tree,'No'))

print(df2)

Output:



**Week 9 To implement K-Means Clustering algorithm**

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.cluster import KMeans

import sklearn.metrics as sm

import pandas as pd

import numpy as np

iris =datasets.load\_iris()

X=pd.DataFrame(iris.data)

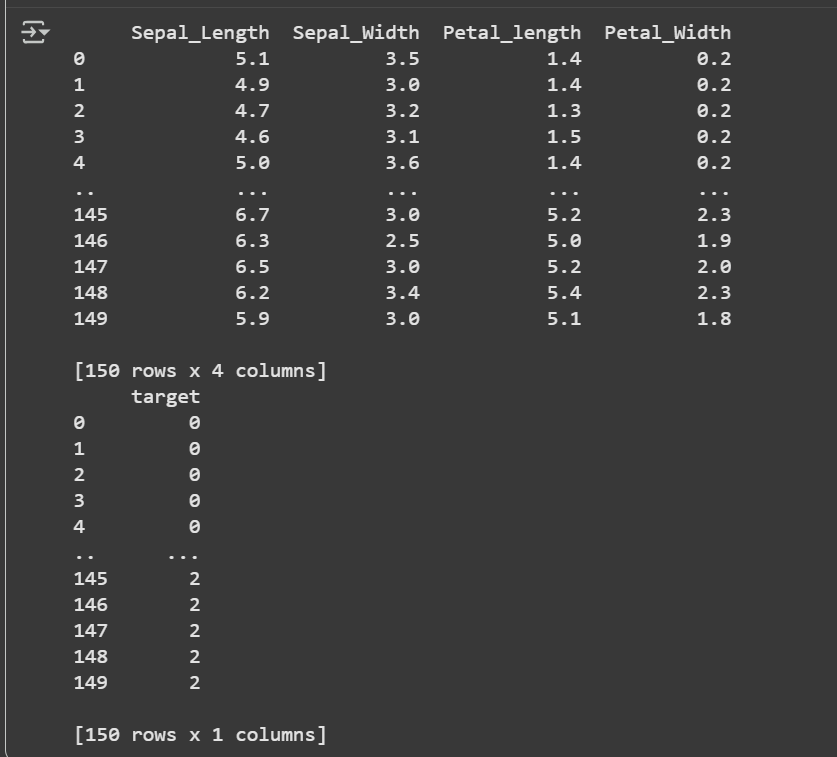
X.columns=['Sepal\_Length','Sepal\_Width', 'Petal\_length', 'Petal\_Width']

print(X)

y=pd.DataFrame(iris.target)

y.columns=['target']

print(y)



plt.figure(figsize=(14,7))

colormap=np.array(['red','lime','black'])

plt.subplot(1,2,1)

plt.scatter(X.Sepal\_Length,X.Sepal\_Width,c=colormap[y.target],s=40)

plt.title('Sepal')

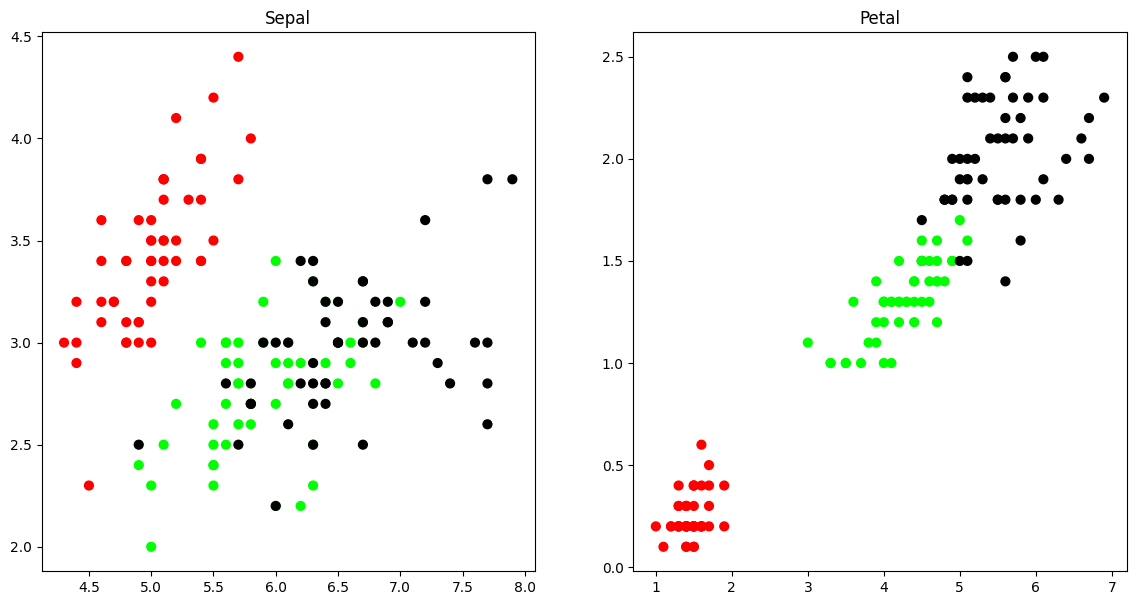
plt.subplot(1,2,2)

plt.scatter(X.Petal\_length,X.Petal\_Width,c=colormap[y.target],s=40)

plt.title('Petal')

plt.show()

Output:



odel=KMeans(n\_clusters=3)

model.fit(X)

print(model.labels\_)

plt.subplot(1,2,1)

plt.scatter(X.Petal\_length,X.Petal\_Width,c=colormap[y.target],s=40)

plt.title('Real Classification')

plt.subplot(1,2,2)

plt.scatter(X.Petal\_length,X.Petal\_Width,c=colormap[model.labels\_],s=40)

plt.title( 'KMEANS Classfication')

plt.show()

print('Accuracy')

print(sm.accuracy\_score(y,model.labels\_))

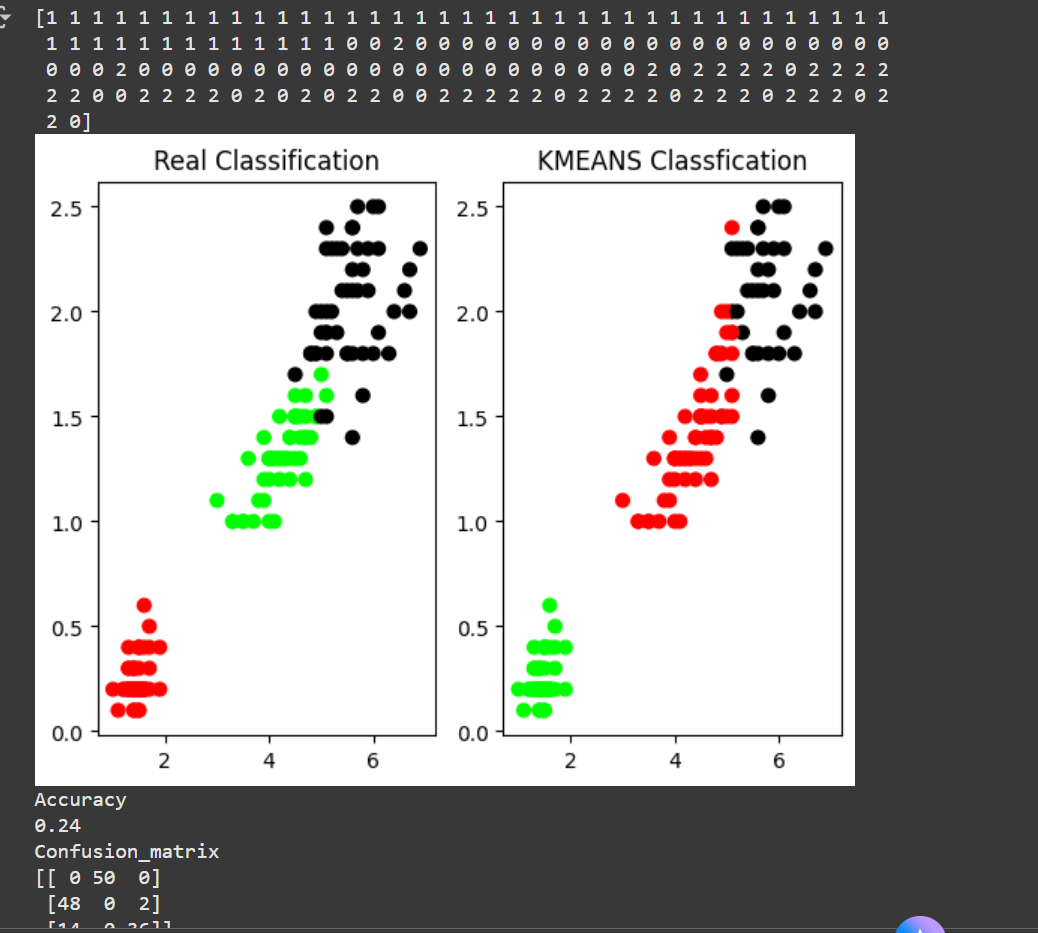
print('Confusion\_matrix')

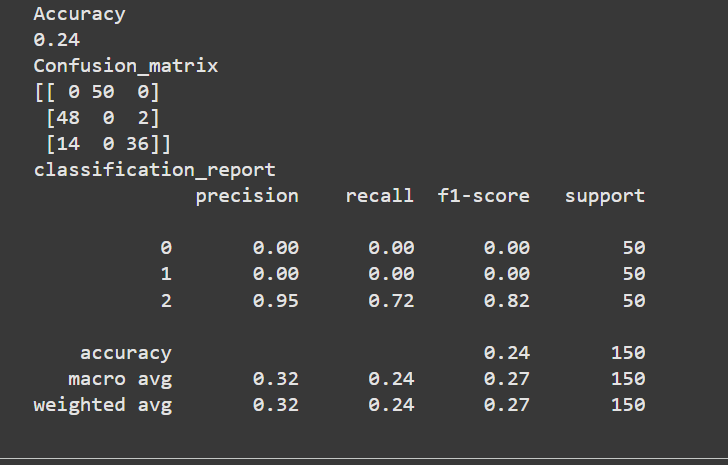
print(sm.confusion\_matrix(y,model.labels\_))

print('classification\_report')

print(sm.classification\_report(y,model.labels\_))

Output:





**Week 10 To implement K-Nearest Neighbor algorithm (K-NN)**

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn import datasets

import pandas as pd

import numpy as np

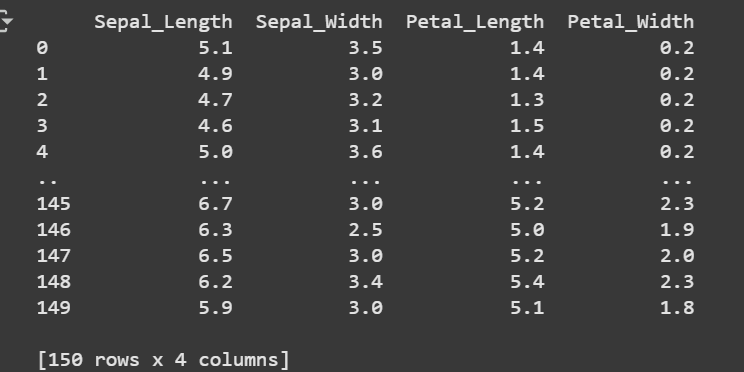
iris = datasets.load\_iris()

X = pd.DataFrame(iris.data)

X.columns = ['Sepal\_Length', 'Sepal\_Width', 'Petal\_Length', 'Petal\_Width']

print(X)

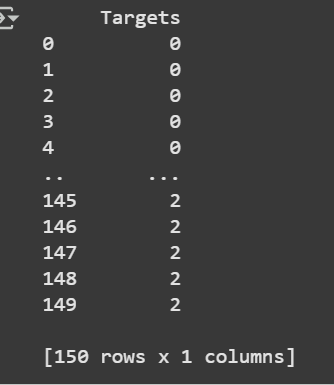
**Output:**



y = pd.DataFrame(iris.target)

y.columns = ['Targets']

print(y)



#Split the data into train and test samples

x\_train,x\_test,y\_train,y\_test = train\_test\_split(iris.data,iris.target,test\_size=0.1)

print("Dataset is split into training and testing...")

print("Size of training data and its label",x\_train.shape,y\_train.shape)

print("Size of testing data and its label",x\_test.shape,y\_test.shape)

Output:

Dataset is split into training and testing...

Size of training data and its label (135, 4) (135,)

Size of testing data and its label (15, 4) (15,)

# prints Label no. and their names

for i in range(len(iris.target\_names)):

  print("Label", i , "-",str(iris.target\_names[i]))

#create object of KNN classifer

classifer = KNeighborsClassifier(n\_neighbors=3)

#perform Training

classifer.fit(x\_train, y\_train)#perform teating

y\_pred=classifer.predict(x\_test)

#Display the results

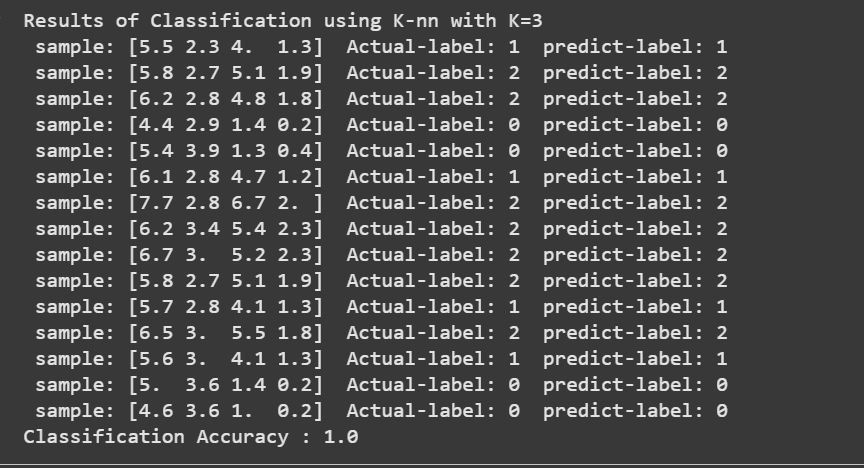
print("Results of Classification using K-nn with K=3")

for r in range(0,len(x\_test)):

  print(" sample:", str(x\_test[r]), " Actual-label:",str(y\_test[r]), " predict-label:", str(y\_pred[r]))

print("Classification Accuracy :" , classifer.score(x\_test,y\_test))

Output:



from sklearn.metrics import classification\_report, confusion\_matrix

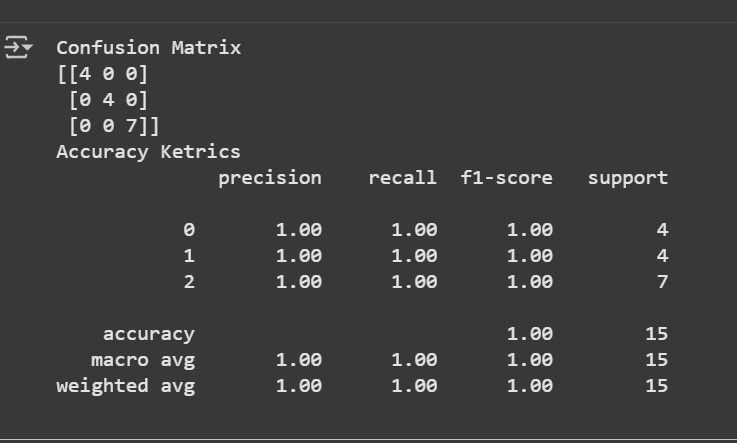
print('Confusion Matrix')

print(confusion\_matrix(y\_test,y\_pred))

print('Accuracy Ketrics')

print(classification\_report(y\_test,y\_pred))

Output:



**Week 11 To implement Back Propagation Algorithm**

import numpy as np

X = np.array(([2,9],[1,5],[3,6])) #Hours Studied,Hours Slept

y=np.array(([92],[86],[89])) #Test Score

y=y/100 #Max Test Score is 100

#Sigmoid Function

def sigmoid(x):

  return 1/(1+ np.exp(-x))

#Derivatives of Sigmoid function

def derivatives\_sigmoid(x):

  return x\*(1-x)

#Variable initialization

epoch=10000 #Setting training iterations

lr=0.1 #Setting learning rate

inputlayer\_neurons = 2 #number of features in data set

hiddenlayers\_neurons = 3 #number of hidden layers neurons

output\_neurons = 1 #number of neurons of output layer

#weight and bias initialization

wh=np.random.uniform(size=(inputlayer\_neurons,hiddenlayers\_neurons))

bias\_hidden=np.random.uniform(size=(1,hiddenlayers\_neurons))  #bias matrix to the hidden layer

weight\_hidden=np.random.uniform(size=(hiddenlayers\_neurons,output\_neurons)) #weight matrix to the output layer

bias\_output=np.random.uniform(size=(1,output\_neurons)) #matrix to output layer

for i in range(epoch):

  hinp1=np.dot(X,wh)

  hinp=hinp1+ bias\_hidden

  hlayer\_activation = sigmoid(hinp)

  outinp1=np.dot(hlayer\_activation,weight\_hidden)

  outinp = outinp1+bias\_output

  output = sigmoid(outinp)

EO = y-output

outgrad=derivatives\_sigmoid(output)

d\_output = EO \* outgrad

EH = d\_output.dot(weight\_hidden.T)

hiddengrad=derivatives\_sigmoid(hlayer\_activation)

d\_hiddenlayer = EH \* hiddengrad

weight\_hidden += hlayer\_activation.T.dot(d\_output) \* lr

bias\_hidden += np.sum(d\_hiddenlayer, axis=0,keepdims=True) \* lr

wh += X.T.dot(d\_hiddenlayer) \* lr

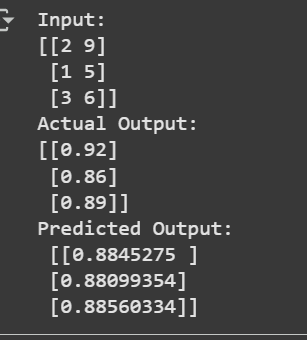
bias\_output += np.sum(d\_output,axis=0,keepdims=True) \*lr

print("Input: \n"+str(X))

print("Actual Output: \n"+str(y))

print("Predicted Output: \n",output)

**Output:**



**Week 12 To implement Support Vector Machine**

from sklearn import datasets

import pandas as pd

import numpy as np

iris = datasets.load\_iris()

iris = datasets.load\_iris()

X = pd.DataFrame(iris.data)

X.columns = ['Sepal\_Length', 'Sepal\_Width', 'Petal\_Length', 'Petal\_Width']

print(X)

y = pd.DataFrame(iris.target)

y.columns = ['Targets']

print(y)

#Split the data into train and test samples

x\_train,x\_test,y\_train,y\_test = train\_test\_split(iris.data,iris.target,test\_size=0.1)

#Fitting the model

from sklearn.svm import SVC

model=SVC()

SVC=model.fit(x\_train,y\_train)

#Predictions from the trained model

y\_pred=SVC.predict(x\_test)

#Model Evaluation

from sklearn.metrics import classification\_report, confusion\_matrix

print('Confusion Matrix')

print(confusion\_matrix(y\_test,y\_pred))

print('Accuracy Ketrics')

print(classification\_report(y\_test,y\_pred))

Output:

