Each cell contains each week's program AIML

```
# Week 1
# BFS Implementation
graph = {
    '5' : ['3', '7'],
    '3' : ['2', '4'],
    '7' : ['8'],
    '2' : [],
    '4' : ['8'],
    '8' : [],
}

visited = []
queue = []

def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    result = []  # To store the order of traversal

    while queue:
        m = queue.pop(0)
        result.append(m)  # Append to result list

        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

    # Print the traversal order with "->"
    print(" -> ".join(result))

print("Following is the breadth-first search:")
bfs(visited, graph, '5')

# DFS Implementation
graph = {
    '5' : ['3', '7'],
    '3' : ['2', '4'],
    '7' : ['8'],
    '2' : [],
    '4' : ['8'],
    '8' : [],
}
visited = set()

def dfs(node, visited, graph):
    result = []  # To store the order of traversal
    def dfs_helper(node):
        if node not in visited:
            visited.add(node)
            result.append(node)  # Append to result list
            for i in graph[node]:
                dfs_helper(i)

    dfs_helper(node)
    # Print the traversal order with "->"
    print(" -> ".join(result))

print("\nFollowing is the depth-first search:")
dfs("5", visited, graph)
```

```
Following is the breadth-first search:
5 -> 3 -> 7 -> 2 -> 4 -> 8

Following is the depth-first search:
5 -> 3 -> 2 -> 4 -> 8 -> 7
```

```
# Week 2
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}  # Store distance from starting node
    parents = {}  # Store parent relationships

    g[start_node] = 0  # Distance from start to itself is 0
    parents[start_node] = start_node  # Root node points to itself
```

```
    while len(open_set) > 0:
        # Find the node with the lowest f(n) = g(n) + h(n)
        n = None
        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node:
            # Reconstruct path from start_node to stop_node
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print(f"Path found: {path}")
            return path

        for (m, weight) in get_neighbors(n):
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            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)

        open_set.remove(n)
        closed_set.add(n)

    print("Path does not exist!")
    return None


def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    return None


def heuristic(n):
    # Heuristic values (Manhattan or straight-line distance to goal)
    H_dist = {
        'A': 6,
        'B': 4,
        'C': 2,
        'D': 1,
        'E': 0
    }
    return H_dist[n]


# Graph definition
Graph_nodes = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 3)],
    'C': [('D', 1), ('E', 5)],
    'D': [('E', 2)],
}

# Perform A* Search from 'A' to 'E'
aStarAlgo('A', 'E')
```

```
Path found: ['A', 'C', 'D', 'E']
['A', 'C', 'D', 'E']
```

```
# Week 3
from sys import maxsize
from itertools import permutations

V = 4

def travellingSalesmanProblem(graph, s):
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
```

```
    min_path = maxsize
    next_permutation = permutations(vertex)

    for i in next_permutation:
        current_pathweight = 0
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]
        min_path = min(min_path, current_pathweight)

    return min_path

if __name__ == "__main__":
    graph = [[0, 10, 15, 20],
             [10, 0, 35, 25],
             [15, 35, 0, 30],
             [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))

colors = ['red', 'blue', 'green', 'yellow', 'black']
states = ['andhra', 'karnataka', 'tamilnadu', 'kerala']
neighbors = {
    'andhra': ['karnataka', 'tamilnadu'],
    'karnataka': ['andhra', 'tamilnadu', 'kerala'],
    'tamilnadu': ['andhra', 'karnataka', 'kerala'],
    'kerala': ['karnataka', 'tamilnadu']
}
colors_of_states = {}

def promising(state, color):
    for neighbor in neighbors.get(state):
        color_of_neighbor = colors_of_states.get(neighbor)
        if color_of_neighbor == color:
            return False
    return True

def get_color_for_state(state):
    for color in colors:
        if promising(state, color):
            return color
[ ]

def main():
    for state in states:
        colors_of_states[state] = get_color_for_state(state)
    print(colors_of_states)

main()
```

```
80
{'andhra': 'red', 'karnataka': 'blue', 'tamilnadu': 'green', 'kerala': 'red'}
```

```
# Week 4
from sympy import symbols, Or, Not, Implies, satisfiable

Rain = symbols('Rain')
Harry_Visited_Hagrid = symbols('Harry_Visited_Hagrid')
Harry_Visited_Dumbledore = symbols('Harry_Visited_Dumbledore')

sentence_1 = Implies(Rain, Harry_Visited_Hagrid)
sentence_2 = (
    Or(Harry_Visited_Hagrid, Harry_Visited_Dumbledore)
    & Not(Harry_Visited_Hagrid & Harry_Visited_Dumbledore)
)
sentence_3 = Harry_Visited_Dumbledore

knowledge_base = sentence_1 & sentence_2 & sentence_3
solution = satisfiable(knowledge_base, all_models=True)

for model in solution:
    if model[Rain]:
        print("It rained today.")
    else:
        print("There is no rain today.")
```

```
There is no rain today.
```

```
# Week 5
from itertools import product
p_burglary = 0.002
p_earthquake = 0.001


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

p_david_calls_given_alarm = 0.91
p_david_does_not_call_given_alarm = 0.09
p_david_calls_given_no_alarm = 0.05
p_david_does_not_call_given_no_alarm = 0.95

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


def joint_probability(alarm,burglary,earthquake,david_calls,sophia_calls):
    if alarm:
      if burglary and earthquake:
        p_alarm = p_alarm_given_burglary_and_earthquake
      elif burglary and not earthquake:
         p_alarm = p_alarm_given_burglary_and_no_earthquake
      elif not burglary and  earthquake:
        p_alarm = p_alarm_given_no_burglary_and_earthquake
      else:
         p_alarm = p_alarm_given_no_burglary_and_no_earthquake


    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_no_alarm)
    p_sophia= (p_sophia_calls_given_alarm if sophia_calls else p_sophia_does_not_call_given_no_alarm)
    return (p_burglary if burglary else 1 - p_burglary) * (p_earthquake if earthquake else 1 - p_earthquake)*p_alarm*p_david*p_sophia


result = joint_probability(
    alarm = True,
    burglary=False,
    earthquake=False,
    david_calls=True,
    sophia_calls=True
)
print(f"The probanility that the alarm has sounded,there is neither a burglary nor earthquake,and both david and sophia called marry is:
```

```
The probanility that the alarm has sounded,there is neither a burglary nor earthquake,and both david and sophia called marry is:0.00
```

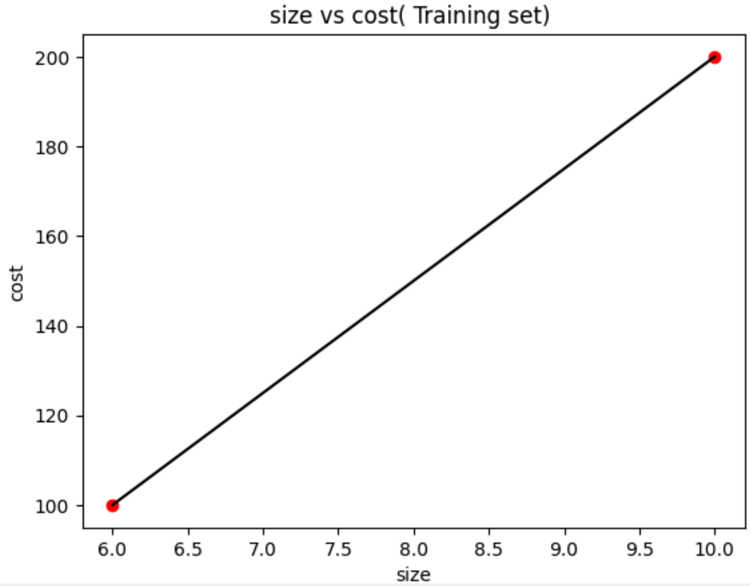
```
# Week 6
import numpy as np
class HMM:
    def __init__(self,states,observations):
         self.states=states
         self.n_states=len(states)
         self.n_obs=len(observations)
         self.State_index={state:i for i,state in enumerate(states)}
         self.obs_index={obs:i for i,obs in enumerate(observations)}
         self.A=np.array([
             [0.6,0.3,0.1],
             [0.2,0.5,0.3],
             [0.1,0.4,0.5]
         ])
         self.B=np.array([
             [0.8,0.15,0.05],
             [0.3,0.4,0.3],
             [0.1,0.2,0.7]
```

```
        ])
        self.pi=np.array([0.5,0.3,0.2])
    def forward(self,obs_seq):
        n=len(obs_seq)
        alpha=np.zeros((n,self.n_states))
        alpha[0]=self.pi * self.B[:,obs_seq[0]]
        for t in range(1,n):
            for j in range(self.n_states):
                alpha[t,j]=(alpha[t-1] @ self.A[:,j]) * self.B[j, obs_seq[t]]
        return alpha.sum(axis=1)[-1]


states=['Sunny','Cloudy','Rainy']
observations=['Umbrella','Normal','Raincoat']
hmm=HMM(states, observations)
obs_seq=['Umbrella','Normal','Umbrella','Raincoat']
obs_seq_indices=[hmm.obs_index[obs] for obs in obs_seq]
prob=hmm.forward(obs_seq_indices)
print(f"Probability of the observation sequence'{obs_seq}':{prob:.4f}")
```

```
Probability of the observation sequence'['Umbrella', 'Normal', 'Umbrella', 'Raincoat']':0.0133
```

```
# Week 7
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pandas as pd

# Data with 3 values
data = {
    'Size': [6, 10, 14],
    'Cost': [100, 200, 300]
}

# Create DataFrame
df = pd.DataFrame(data)

# Save to CSV
df.to_csv('pizza.csv', index=False)

dataset=pd.read_csv('pizza.csv')
dataset.head()

X=dataset.iloc[:,0:-1].values
y=dataset.iloc[:,1].values
X

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)
X_train

from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train,y_train)
regressor

regressor.predict([[30]])
y_pred=regressor.predict(X_test)
y_pred

df1=pd.DataFrame({'Actual':y_test,'Prediction':y_pred})
df1

regressor.score(X_test,y_test)
plt.scatter(X_train,y_train ,color='red')
plt.plot(X_train,regressor.predict(X_train),color='black')
plt.title("size vs cost( Training set) ")
plt.xlabel("size")
plt.ylabel("cost")
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_regression.py:1211: UndefinedMetricWarning: R^2 score is not well-defined w
  warnings.warn(msg, UndefinedMetricWarning)
```




```
# Week 8
import pandas as pd
from collections import Counter
import math
from pprint import pprint

#Entropy calculation function
def entropy(probs):
  return sum(-prob * math.log(prob, 2) for prob in probs if prob > 0)

def entropy_of_list(a_list):
  cnt=Counter(a_list)
  num_instances=len(a_list)
  probs=[x/num_instances for x in cnt.values()]
  return entropy(probs)


# Corrected Information Gain function
def information_gain(df, split_attribute_name, target_attribute_name):

    df_split = df.groupby(split_attribute_name)


    nobs = len(df.index) * 1.0


    df_agg_ent = df_split[target_attribute_name].agg([entropy_of_list, lambda x:len(x) / nobs])
    df_agg_ent.columns = ['entropy', 'prob']


    avg_info = sum(df_agg_ent['entropy'] * df_agg_ent['prob'])


    old_entropy = entropy_of_list(df[target_attribute_name])


    return old_entropy - avg_info

#ID3 Decision Tree algorithm
def id3DT(df,target_attribute_name, attribute_names, default_class=None):#PlayTennis
  cnt  = Counter(df[target_attribute_name])#yes:9,no:5
  if len(cnt) == 1:
    return next(iter(cnt))
  elif df.empty or not attribute_names:
    return default_class
  else:
    default_class = max(cnt, key=cnt.get)#yes
    gain = [information_gain(df, attr, target_attribute_name) for attr in attribute_names]

    index_of_max = gain.index(max(gain))#0.2464
    best_attr = attribute_names[index_of_max]#outlook
    tree = {best_attr: {}}
    remaining_attributes = [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, default_class)
```

```
      tree[best_attr][attr_val] = subtree

    return tree

  tree

def classify(instance,tree,default=None):
  attribute=next(iter(tree))
  if instance[attribute] in tree[attribute]:
    result=tree[attribute][instance[attribute]]
    if isinstance(result,dict):
      return classify(instance,result)
    else:
      return result

  else:
    return default

data={
    'Outlook':['Sunny','Sunny','Overcast','Rain','Rain','Rain','Overcast','Sunny','Sunny','Rain','Sunny','Overcast','Overcast','Rain'],
    'Temperature':['Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild','Cool','Mild','Mild','Mild','Hot','Mild'],
    'Humidity':['High','High','High','High','Normal','Normal','Normal','High','Normal','Normal','Normal','High','Normal','High'],
    'Wind':['Weak','Strong','Weak','Weak','Weak','Strong','Strong','Weak','Weak','Weak','Strong','Strong','Weak','Strong'],
    'PlayTennis':['No','No','Yes','Yes','Yes','No','Yes','No','Yes','Yes','Yes','Yes','Yes','No']

}

df=pd.DataFrame(data)
df

attribute_names=list(df.columns)
attribute_names.remove('PlayTennis')
attribute_names

tree=id3DT(df,'PlayTennis',attribute_names)
print("The Resultant Decision Tree is:")
pprint(tree)

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

df2=pd.DataFrame(new_data)
df2

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

```
The Resultant Decision Tree is:
{'Outlook': {'Overcast': 'Yes',
             'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},
             'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}
```

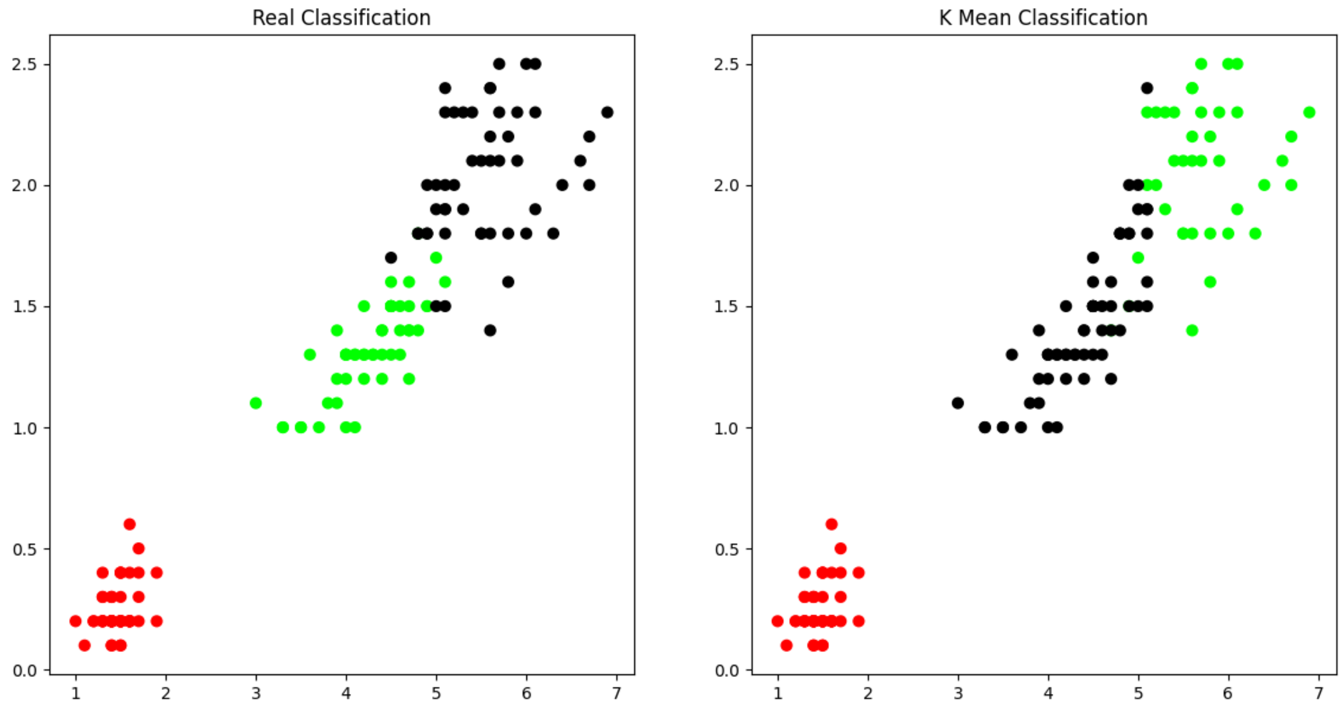
	Outlook	Temperature	Humidity	Wind	Predicted
0	Rain	Mild	High	Weak	Yes
1	Sunny	Hot	Normal	Strong	Yes

Next steps: Generate code with df2 View recommended plots New interactive sheet

```
# Week 9

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']
y=pd.DataFrame(iris.target)
y.columns=['target']
plt.figure(figsize=(14,7))
colormap=np.array(['red','lime','black'])
'''plt.subplot(1,2,1)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y.target],s=40)
```

```
plt.title('Sepal')
plt.subplot(1,2,2)
plt.scatter(X.Sepal_Length,X.Sepal_Width,c=colormap[y.target],s=40)
plt.title('Petal')'''

model=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('K Mean Classification')
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 1 1 1 1 2 1 1 1 1
 1 1 2 2 1 1 1 1 2 1 2 1 2 1 1 2 2 1 1 1 1 1 2 1 1 1 1 2 1 1 1 2 1 1 1 2 1
 1 2]
Text(0.5, 1.0, 'K Mean Classification')
```




```
# Week 10

import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score

iris= load_iris()

print("Dataset keys:",iris.keys())

df = pd.DataFrame(iris['data'], columns=iris['feature_names'])
print("Feature Data\n",df.head())

print("Target names:", iris['target_names'])
print("Feature names:",iris['feature_names'])

print("Target array:\n", iris['target'])
X=df
y=iris['target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

knn= KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train,y_train)
print("length",len(X_test))
y_pred=knn.predict(X_test)
```

```
cm_test = confusion_matrix(y_test,y_pred)
print("Confusion Matrix (Test Data):\n",cm_test)

accuracy_test= accuracy_score(y_test,y_pred)
print("Correct prediction on test data:", accuracy_test)

print("length",len(X_train))
y_train_pred=knn.predict(X_train)

cm_train=confusion_matrix(y_train,y_train_pred)
print("Confusion Matrix (Training Data):\n",cm_train)
```

```
Dataset keys: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename', 'data_module'])
Feature Data
    sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
Target names: ['setosa' 'versicolor' 'virginica']
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target array:
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
length 50
Confusion Matrix (Test Data):
 [[19  0  0]
 [ 0 15  0]
 [ 0  1 15]]
Correct prediction on test data: 0.98
length 100
Confusion Matrix (Training Data):
 [[31  0  0]
 [ 0 33  2]
 [ 0  2 32]]
```

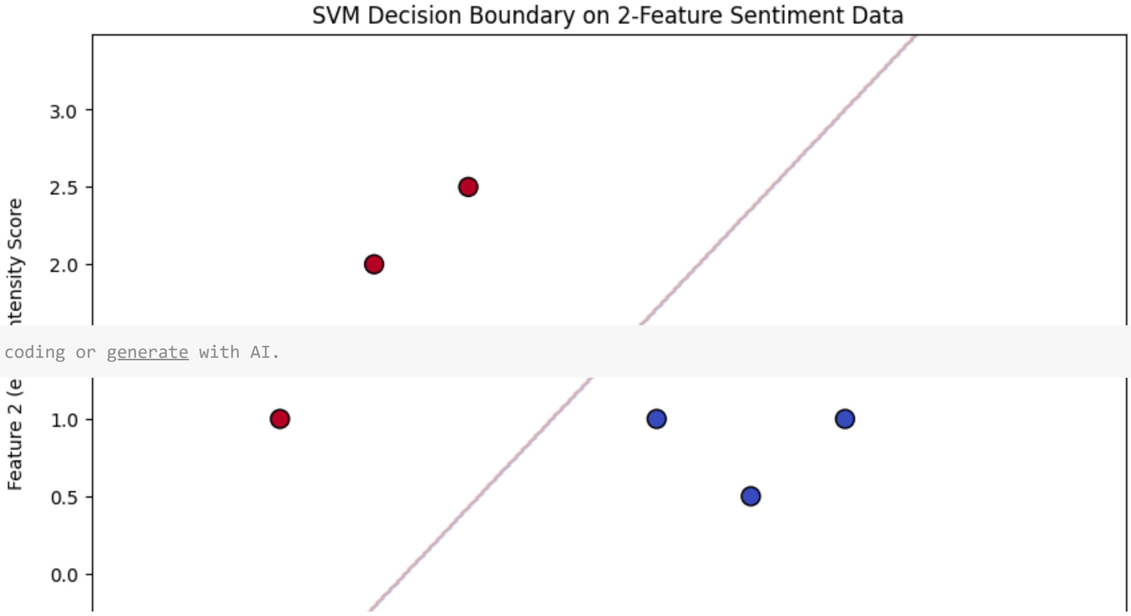
```
# Week 11
import numpy as np

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

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

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

input_layer_neurons=2
hidden_layer_neurons=4
output_layer_neurons=1
epochs=10000
learning_rate=0.1
np.random.seed(42)
wh=np.random.uniform(size=(input_layer_neurons,hidden_layer_neurons))
bh=np.random.uniform(size=(1,hidden_layer_neurons))

wout=np.random.uniform(size=(hidden_layer_neurons,output_layer_neurons))
bout=np.random.uniform(size=(1,output_layer_neurons))

for epoch in range(epochs):
    hidden_layer_input=np.dot(X,wh)+bh
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input=np.dot(hidden_layer_output,wout)+bout
    output=sigmoid(output_layer_input)
    error=y-output

    hidden_layer_gradient=sigmoid_derivative(output)
    output_layer_gradient=sigmoid_derivative(hidden_layer_output)
    d_output=error*hidden_layer_gradient

    hidden_layer_gradient=sigmoid_derivative(hidden_layer_output)
    d_hidden_layer=d_output.dot(wout.T)*hidden_layer_gradient

    wout+=hidden_layer_output.T.dot(d_output)*learning_rate
    bout+=np.sum(d_output,axis=0,keepdims=True)*learning_rate
    wh+=X.T.dot(d_hidden_layer)*learning_rate
    bh+=np.sum(d_hidden_layer,axis=0,keepdims=True)*learning_rate
```

```
    if(epoch %1000 ==0):
      print(f"Epoch {epoch},Error: {np.mean(np.abs(error))}")

print("Final predictions after training:")
print(output)
```

```
Epoch 0,Error: 0.49914791405546904
Epoch 1000,Error: 0.4989908274224632
Epoch 2000,Error: 0.49392112204426847
Epoch 3000,Error: 0.46086324847622695
Epoch 4000,Error: 0.37081148754970494
Epoch 5000,Error: 0.2293685934150816
Epoch 6000,Error: 0.1411700792664044
Epoch 7000,Error: 0.10187019467760619
Epoch 8000,Error: 0.08085064924133495
Epoch 9000,Error: 0.06790718296112089
Final predictions after training:
[[0.04690963]
 [0.95663392]
 [0.92548675]
 [0.07177571]]
```

```
# Week 12

import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score


data = np.array([
    [1.5,2.0,1],
    [1.0,1.0,1],
    [2.0,2.5,1],
    [2.5,1.5,1],
    [3.0,1.0,0],
    [3.5,0.5,0],
    [4.0,1.0,0],
    [4.5,1.5,0]
])
X= data[:,:2]
y=data[:,2]

svm_model =SVC(kernel='linear')
svm_model.fit(X,y)

y_pred = svm_model.predict(X)
print("Accuracy:", accuracy_score(y, y_pred))
print(" \nClassification Report:\n",  classification_report(y, y_pred))

x_min, x_max =X[:, 0].min()-1, X[:, 0].max()+1
y_min, y_max =X[:, 1].min()-1, X[:, 1].max()+1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
print("xx",xx)
print("yy", yy)
Z = svm_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
print("Z=",Z)
plt.figure(figsize=(10, 6))
plt.contour(xx, yy, Z, alpha=0.2, cmap='coolwarm')
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm',edgecolors='k',s=100)
plt.xlabel("Feature 1 (e.g)., Positivity Score")
plt.ylabel("Feature 2 (e.g)., Intensity Score")
plt.title('SVM Decision Boundary on 2-Feature Sentiment Data')
plt.show()
```

```
Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00         4
         1.0       1.00      1.00      1.00         4

    accuracy                           1.00         8
   macro avg       1.00      1.00      1.00         8
weighted avg       1.00      1.00      1.00         8

xx [[0.   0.01 0.02 ... 5.47 5.48 5.49]
 [0.   0.01 0.02 ... 5.47 5.48 5.49]
 [0.   0.01 0.02 ... 5.47 5.48 5.49]
 ...
 [0.   0.01 0.02 ... 5.47 5.48 5.49]
 [0.   0.01 0.02 ... 5.47 5.48 5.49]
 [0.   0.01 0.02 ... 5.47 5.48 5.49]]
yy [[-0.5  -0.5  -0.5  ... -0.5  -0.5  -0.5 ]
 [-0.49 -0.49 -0.49 ... -0.49 -0.49 -0.49]
 [-0.48 -0.48 -0.48 ... -0.48 -0.48 -0.48]
 ...
 [ 3.47  3.47  3.47 ...  3.47  3.47  3.47]
 [ 3.48  3.48  3.48 ...  3.48  3.48  3.48]
 [ 3.49  3.49  3.49 ...  3.49  3.49  3.49]]
Z= [[1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]
 ...
 [1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]]
```



Start coding or generate with AI.