**Cryptairthmetic**

import itertools

import re

#Faster Implementation

def compile\_formula(formula, verbose=False):

formula = formula.replace(' = ', ' == ')

letters = cat(sorted(set(re.findall('[A-Z]', formula))))

firstletters = sorted(set(re.findall(r'\b([A-Z])[A-Z]', formula)))

body = re.sub('[A-Z]+', compile\_word, formula)

body = ' and '.join(firstletters + [body])

fn = 'lambda {}: {}'.format(','.join(letters), body)

if verbose: print(fn)

assert len(letters) <= 10

return eval(fn), letters

def compile\_word(matchobj):

word = matchobj.group()

terms = reversed([mult(10\*\*i, L) for (i, L) in enumerate(reversed(word))])

return '(' + '+'.join(terms) + ')'

def mult(factor, var): return var if factor == 1 else str(factor) + '\*' + var

def faster\_solve(formula):

fn, letters = compile\_formula(formula)

for digits in itertools.permutations((1,2,3,4,5,6,7,8,9,0), len(letters)):

try:

if fn(\*digits):

yield formula.translate(str.maketrans(letters, cat(map(str, digits))))

except ArithmeticError:

pass

#Standard Solve Implementations

def solve(formula):

return filter(valid, letter\_replacements(formula))

#

def letter\_replacements(formula):

formula = formula.replace(' = ', ' == ') # Allow = or ==

letters = cat(set(re.findall('[A-Z]', formula)))

for digits in itertools.permutations('1234567890', len(letters)):

yield formula.translate(str.maketrans(letters, cat(digits)))

def valid(exp):

try:

return not leading\_zero(exp) and eval(exp) is True

except ArithmeticError:

return False

cat = ''.join # Function to concatenate strings

leading\_zero = re.compile(r'\b0[0-9]').search # Function to check for illegal number

print(next(faster\_solve('SEND + MORE = MONEY')))

Graph Coloring Constraint

from constraint import \*

def graph\_coloring(graph, num\_colors):

problem = Problem()

# Create a variable for each vertex and add its domain

for vertex in graph.keys():

problem.addVariable(vertex, range(num\_colors))

# Add a constraint for each pair of adjacent vertices

for vertex, neighbors in graph.items():

for neighbor in neighbors:

problem.addConstraint(lambda x, y: x != y, (vertex, neighbor))

# Find a solution that satisfies all constraints

solution = problem.getSolutions()

return solution

# Example usage:

graph = {

'A': ['B', 'C'],

'B': ['A', 'C', 'D'],

'C': ['A', 'B', 'D'],

'D': ['B', 'C']

}

num\_colors = 3

solution = graph\_coloring(graph, num\_colors)

print(solution) # {'A': 0, 'B': 1, 'C': 2, 'D': 0}

Graph Color

class CSPSolver:

def \_\_init\_\_(self, variables, domains, constraints):

self.variables = variables

self.domains = domains

self.constraints = constraints

def solve(self):

assignment = {}

return self.backtrack\_search(assignment)

def backtrack\_search(self, assignment):

if len(assignment) == len(self.variables):

return assignment

var = self.select\_unassigned\_variable(assignment)

for value in self.order\_domain\_values(var, assignment):

if self.is\_consistent(var, value, assignment):

assignment[var] = value

result = self.backtrack\_search(assignment)

if result is not None:

return result

del assignment[var]

return None

def select\_unassigned\_variable(self, assignment):

unassigned = set(self.variables) - set(assignment.keys())

return min(unassigned, key=lambda var: len(self.domains[var]))

def order\_domain\_values(self, var, assignment):

return self.domains[var]

def is\_consistent(self, var, value, assignment):

assignment[var] = value

for constraint in self.constraints:

if not constraint(assignment):

return False

return True

# Define variables and domains

variables = ['WA', 'NT', 'Q', 'NSW', 'V', 'SA', 'T']

domains = {

'WA': ['red', 'green', 'blue'],

'NT': ['red', 'green', 'blue'],

'Q': ['red', 'green', 'blue'],

'NSW': ['red', 'green', 'blue'],

'V': ['red', 'green', 'blue'],

'SA': ['red', 'green', 'blue'],

'T': ['red', 'green', 'blue'],

}

# Define constraints

def adjacent\_regions\_must\_have\_different\_colors(assignment):

if 'WA' in assignment and 'NT' in assignment:

if assignment['WA'] == assignment['NT']:

return False

# Define similar constraints for other adjacent regions

return True

def no\_constraint\_violations(assignment):

return True

constraints = [adjacent\_regions\_must\_have\_different\_colors, no\_constraint\_violations]

# Create the CSPSolver instance and solve the problem

solver = CSPSolver(variables, domains, constraints)

solution = solver.solve()

print(solution)

N Queens

class CSPSolver:

def \_\_init\_\_(self, variables, domains, constraints):

self.variables = variables

self.domains = domains

self.constraints = constraints

def solve(self):

assignment = {}

return self.backtrack\_search(assignment)

def backtrack\_search(self, assignment):

if len(assignment) == len(self.variables):

return assignment

var = self.select\_unassigned\_variable(assignment)

for value in self.order\_domain\_values(var, assignment):

if self.is\_consistent(var, value, assignment):

assignment[var] = value

result = self.backtrack\_search(assignment)

if result is not None:

return result

del assignment[var]

return None

def select\_unassigned\_variable(self, assignment):

unassigned = set(self.variables) - set(assignment.keys())

return min(unassigned, key=lambda var: len(self.domains[var]))

def order\_domain\_values(self, var, assignment):

return self.domains[var]

def is\_consistent(self, var, value, assignment):

# Check row and column constraints

if value in assignment.values():

return False

for k, v in assignment.items():

if abs(k - var) == abs(v - value):

return False

return True

def print\_solution(solution):

if solution is None:

print("No solution found")

else:

for i in range(len(solution)):

row = ""

for j in range(len(solution)):

if solution[i] == j:

row += "Q "

else:

row += ". "

print(row)

# Define variables and domains

n = 8 # Change n to the desired value of N

variables = list(range(n))

domains = {i: list(range(n)) for i in range(n)}

# Define constraints

constraints = []

# Create the CSPSolver instance and solve the problem

solver = CSPSolver(variables, domains, constraints)

solution = solver.solve()

print(solution)

print\_solution(solution)

Job Scheduling Constraint Library

from constraint import \*

# Define the job scheduling problem

problem = Problem()

# Define the variables and domains

# Each variable represents a job, and the domain of each variable is the time required to complete the job

job1 = [3, 5, 1]

job2 = [2, 4, 6]

job3 = [4, 3, 2]

problem.addVariables(["job1", "job2", "job3"], [job1, job2, job3])

# Define the constraints

# Each constraint enforces that no two jobs overlap in time

for i in range(3):

for j in range(i+1, 3):

problem.addConstraint(lambda j1, j2: j1[i]+j1[j] <= j2[i] or j2[i]+j2[j] <= j1[i], ("job"+str(i+1), "job"+str(j+1)))

# Solve the problem

solutions = problem.getSolutions()

# Print the solutions

for solution in solutions:

print(solution)

TSP Constraint

from constraint import Problem, AllDifferentConstraint

# Define the distance matrix between cities

distances = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

# Define the number of cities

n = len(distances)

# Create a problem instance

problem = Problem()

# Create variables for the order in which cities are visited

order = list(range(n))

problem.addVariables(order, range(n))

# Add constraint to ensure each city is visited exactly once

problem.addConstraint(AllDifferentConstraint(), order)

# Define a function to calculate the total distance of a given route

def calculate\_distance(route):

distance = 0

for i in range(n-1):

distance += distances[route[i]][route[i+1]]

distance += distances[route[n-1]][route[0]]

return distance

# Add constraint to minimize the total distance of the route

problem.addConstraint(lambda \*route: calculate\_distance(route), order)

# Solve the problem

solution = problem.getSolution()

# Print the solution

print("Shortest route:", [solution[i] for i in range(n)])

print("Total distance:", calculate\_distance([solution[i] for i in range(n)]))

Latin Problem Constraint

from constraint import Problem, AllDifferentConstraint

# Define the size of the Latin square

n = 5

# Create a problem instance

problem = Problem()

# Create variables for each cell in the Latin square

variables = []

for i in range(n):

for j in range(n):

variables.append((i, j))

# Add domain constraints for each variable

for variable in variables:

problem.addVariable(variable, range(1, n+1))

# Add row constraints

for i in range(n):

row\_variables = [(i, j) for j in range(n)]

problem.addConstraint(AllDifferentConstraint(), row\_variables)

# Add column constraints

for j in range(n):

column\_variables = [(i, j) for i in range(n)]

problem.addConstraint(AllDifferentConstraint(), column\_variables)

# Add diagonal constraints

diagonal1\_variables = [(i, i) for i in range(n)]

problem.addConstraint(AllDifferentConstraint(), diagonal1\_variables)

diagonal2\_variables = [(i, n-i-1) for i in range(n)]

problem.addConstraint(AllDifferentConstraint(), diagonal2\_variables)

# Solve the problem

solutions = problem.getSolutions()

# Print the solutions

for solution in solutions:

for i in range(n):

for j in range(n):

print(solution[(i, j)], end=' ')

print()

print()

Tic Tac Toe

class TicTacToe:

def \_\_init\_\_(self):

self.board = [" " for x in range(9)]

self.player = "X"

def display\_board(self):

row1 = "| {} | {} | {} |".format(self.board[0], self.board[1], self.board[2])

row2 = "| {} | {} | {} |".format(self.board[3], self.board[4], self.board[5])

row3 = "| {} | {} | {} |".format(self.board[6], self.board[7], self.board[8])

print()

print(row1)

print(row2)

print(row3)

def player\_move(self):

print("Player {} turn.".format(self.player))

choice = int(input("Enter your move (1-9): ").strip())

if self.board[choice - 1] == " ":

self.board[choice - 1] = self.player

else:

print("That space is already taken.")

self.player\_move()

def check\_for\_winner(self):

winning\_combinations = [

[0, 1, 2], [3, 4, 5], [6, 7, 8], # rows

[0, 3, 6], [1, 4, 7], [2, 5, 8], # columns

[0, 4, 8], [2, 4, 6] # diagonals

]

for combination in winning\_combinations:

if self.board[combination[0]] == self.board[combination[1]] == self.board[combination[2]] != " ":

return self.board[combination[0]]

if " " not in self.board:

return "Tie"

return None

def minimax(self, player):

result = self.check\_for\_winner()

if result is not None:

if result == "X":

return -1

elif result == "O":

return 1

else:

return 0

if player == "X":

best = [None, -2]

for i in range(9):

if self.board[i] == " ":

self.board[i] = "X"

score = self.minimax("O")

self.board[i] = " "

if score > best[1]:

best = [i, score]

else:

best = [None, 2]

for i in range(9):

if self.board[i] == " ":

self.board[i] = "O"

score = self.minimax("X")

self.board[i] = " "

if score < best[1]:

best = [i, score]

return best[1]

def computer\_move(self):

print("Computer turn.")

move = self.minimax("O")

print(move)

if move == 1:

for i in range(9):

if self.board[i] == " ":

self.board[i] = "O"

if self.check\_for\_winner() == "O":

print("Computer chooses {}".format(i + 1))

return

self.board[i] = " "

elif move == -1:

for i in range(9):

if self.board[i] == " ":

self.board[i] = "X"

if self.check\_for\_winner() == "X":

self.board[i] = "O"

print("Computer chooses {}".format(i + 1))

return

self.board[i] = " "

for i in range(9):

if self.board[i] == " ":

self.board[i] = "O"

print("Computer chooses {}".format(i + 1))

break

game = TicTacToe()

while True:

game.display\_board()

game.player\_move()

result = game.check\_for\_winner()

if result is not None:

game.display\_board()

if result == "Tie":

print("Tie game.")

else:

print("{} wins!".format(result))

break

game.computer\_move()

result = game.check\_for\_winner()

if result is not None:

game.display\_board()

if result == "Tie":

print("Tie game.")

else:

print("{} wins!".format(result))

break

Min Max

# Returns optimal value for current player

# (Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer, values):

# Terminating condition. i.e

# leaf node is reached

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = -float('inf')

# Recur for left and right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

False, values)

best = max(best, val)

return best

else:

best = float('inf')

# Recur for left and

# right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

True, values)

best = min(best, val)

return best

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

values = [2, 4, 6, 8, 1, 2, 10, 12]

print("The optimal value is:", minimax(0, 0, True, values))

Alpha Beta Pruning

# Initial values of Alpha and Beta

MAX, MIN = 1000, -1000

# Returns optimal value for current player

# (Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer,

values, alpha, beta):

# Terminating condition. i.e

# leaf node is reached

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

# Recur for left and right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

# Alpha Beta Pruning

if beta <= alpha:

break

return best

else:

best = MAX

# Recur for left and

# right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

# Alpha Beta Pruning

if beta <= alpha:

break

return best

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

values = [2, 4, 6, 8, 1, 2, 10, 12]

print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))

Linear Regression

import pandas as pd

from sklearn.linear\_model import LinearRegression

data = pd.read\_csv('data.csv')

X = data[['x1', 'x2', ... , 'xn']] # independent variables

y = data['y'] # dependent variable

model = LinearRegression()

model.fit(X, y)

model = LinearRegression()

model.fit(X, y)

predictions = model.predict(new\_data)

#Least Squares

slope, intercept = np.polyfit(x, y, 1)

Evaluation Metrics

#R2 score

r2 = r2\_score(y\_actual, y\_pred)

print('R-squared score:', r2)

#MSE & RSE

rmse = np.sqrt(mean\_squared\_error(y\_actual, y\_pred))

rse = np.sqrt(sum((y\_pred - y\_actual)\*\*2) / (len(y\_actual) - 2))

print('RMSE:', rmse)

print('RSE:', rse)

#accuracy\_score

from sklearn.metrics import roc\_curve, roc\_auc\_score, accuracy\_score

acc = accuracy\_score(y\_test, preds)

print('Accuracy:', acc)

#ROC

from sklearn.metrics import roc\_curve, roc\_auc\_score, accuracy\_score

probs = clf.predict\_proba(X\_test)[:, 1]

# Calculate false positive rate, true positive rate, and thresholds for ROC curve

fpr, tpr, thresholds = roc\_curve(y\_test, probs)

# Plot ROC curve

plt.plot(fpr, tpr, linestyle='--', label='Logistic Regression')

# Add labels and legend to plot

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.legend()

#confusion matrix

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(y\_true, y\_pred)

Decision Tree

from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier()

clf.fit(X\_train, y\_train)

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

K means Classifier

from sklearn.neighbors import KNeighborsClassifier

# Create the classifier object

clf = KNeighborsClassifier(n\_neighbors=5)

# Train the model on the data

clf.fit(X\_train, y\_train)

# Predict the output for the test data

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

SVM

from sklearn.svm import SVC

# Create the classifier object

clf = SVC(kernel='linear', C=1.0)

# Train the model on the data

clf.fit(X\_train, y\_train)

# Predict the output for the test data

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

svm = SVC(kernel='rbf')

# train the SVM classifier

svm.fit(X\_train, y\_train)

# make predictions on test set

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

K Means Clustering

kmeans = KMeans(n\_clusters=2)

# fit the K-Means model on the data

kmeans.fit(X)

# get the coordinates of the centroids

centroids = kmeans.cluster\_centers\_

# get the labels of each point

labels = kmeans.labels\_

Reinforcement Learning

Frozen Lake

import numpy as np

import gym

#setting environment

env = gym.make('FrozenLake-v1')

#set hyperparams

alpha = 0.1

gamma = 0.99

num\_episodes = 10000 #doubled the training

#initialize Q table

num\_states = env.observation\_space.n

num\_actions = env.action\_space.n

Q = np.zeros((env.observation\_space.n,env.action\_space.n))

#run Q learning algorithm now

for i in range(num\_episodes):

#reset env

state = env.reset()

done = False

total\_reward = 0

while not done:

if np.random.uniform(0,1) < 0.5:

action = env.action\_space.sample()

else:

action = np.argmax(Q[state,:])

#take action and observe next state and reward

next\_state,reward,done,info = env.step(action)

# Update the Q-value of the (state, action) pair

Q[state,action] = Q[state,action] + alpha \*(reward + gamma \*np.max(Q[next\_state,:])-Q[state,action])

state = next\_state

total\_reward += reward

print(f"Episode {i}: Total reward = {total\_reward}")

#tesing

num\_test\_episodes = 500 #increased from 100 to 500

num\_test\_steps = 500 #increased from 100 to 500

num\_successes = 0

for i in range(num\_test\_episodes):

state=env.reset()

done = False

steps = 0

while not done and steps < num\_test\_steps:

action = np.argmax(Q[state,:])

next\_state,reward,done,info = env.step(action)

state=next\_state

steps+=1

if state==15:

num\_successes+=1

print("Success rate:", num\_successes/num\_test\_episodes)

Traffic V3

env = gym.make('Taxi-v3')

Q = np.zeros((env.observation\_space.n, env.action\_space.n))

learning\_rate = 0.8

discount\_factor = 0.95

num\_episodes = 5000

max\_steps = 100

for episode in range(num\_episodes):

state = env.reset()

total\_reward = 0

done = False

for step in range(max\_steps):

epsilon = 0.1

if np.random.uniform(0, 1) < epsilon:

action = env.action\_space.sample()

else:

action = np.argmax(Q[state, :])

next\_state, reward, done, \_ = env.step(action)

Q[state, action] = Q[state, action] + learning\_rate \* (

reward + discount\_factor \* np.max(Q[next\_state, :]) - Q[state, action])

total\_reward += reward

state = next\_state

if done:

break

print("Episode:", episode, "Total Reward:", total\_reward)

num\_eval\_episodes = 100

eval\_rewards = []

for episode in range(num\_eval\_episodes):

state = env.reset()

total\_reward = 0

done = False

while not done:

action = np.argmax(Q[state, :])

next\_state, reward, done, \_ = env.step(action)

total\_reward += reward

state = next\_state

eval\_rewards.append(total\_reward)

print("Average Evaluation Reward:", np.mean(eval\_rewards))

Task 3 RL

num\_nodes = 5

sink\_node = 0

cost\_matrix = np.array([[0, 1, 2, 3, 4],

[1, 0, 1, 2, 3],

[2, 1, 0, 1, 2],

[3, 2, 1, 0, 1],

[4, 3, 2, 1, 0]])

Q = np.zeros((num\_nodes, num\_nodes))

learning\_rate = 0.8

discount\_factor = 0.95

num\_episodes = 2000

max\_steps = 100

for episode in range(num\_episodes):

state = np.random.randint(0, num\_nodes)

total\_cost = 0

done = False

for step in range(max\_steps):

epsilon = 0.1

if np.random.uniform(0, 1) < epsilon:

action = np.random.randint(0, num\_nodes)

else:

action = np.argmax(Q[state, :])

cost = cost\_matrix[state, action]

Q[state, action] = Q[state, action] + learning\_rate \* (

cost + discount\_factor \* np.min(Q[action, :]) - Q[state, action])

total\_cost += cost

state = action

if state == sink\_node:

break

print("Episode:", episode, "Total Cost:", total\_cost)

num\_eval\_episodes = 10

eval\_costs = []

for episode in range(num\_eval\_episodes):

state = np.random.randint(0, num\_nodes)

total\_cost = 0

done = False

while not done:

action = np.argmax(Q[state, :])

cost = cost\_matrix[state, action]

total\_cost += cost

state = action

if state == sink\_node:

break

eval\_costs.append(total\_cost)

print("Average Evaluation Cost:", np.mean(eval\_costs))

Epsilon Greedy

#another way for Q-learning

import numpy as np

import gym

# Create the FrozenLake-v0 environment

env = gym.make('FrozenLake-v1')

# Set the hyperparameters

num\_episodes = 10000

max\_steps = 100

alpha = 0.1

gamma = 0.99

epsilon = 1.0

epsilon\_min = 0.01

epsilon\_decay = 0.99

# Initialize the Q-table

num\_states = env.observation\_space.n

num\_actions = env.action\_space.n

Q = np.zeros((num\_states, num\_actions))

# Define the epsilon-greedy policy

def epsilon\_greedy\_policy(state, epsilon):

if np.random.rand() < epsilon:

# Take a random action

action = env.action\_space.sample()

else:

# Choose the best action from the Q-table

action = np.argmax(Q[state, :])

return action

# Loop over episodes

for episode in range(num\_episodes):

# Reset the environment and get the initial state

state = env.reset()

# Choose the initial action using the epsilon-greedy policy

action = epsilon\_greedy\_policy(state, epsilon)

# Initialize the total reward for the episode

total\_reward = 0

# Loop over steps within this episode

for t in range(max\_steps):

# Take the chosen action and observe the next state and reward

next\_state, reward, done, info = env.step(action)

# Choose the next action using the epsilon-greedy policy

next\_action = epsilon\_greedy\_policy(next\_state, epsilon)

# Update the Q-table

td\_error = reward + gamma \* Q[next\_state, next\_action] - Q[state, action]

Q[state, action] += alpha \* td\_error

# Update the state, action, and total reward

state = next\_state

action = next\_action

total\_reward += reward

# If the episode is complete, break out of the loop

if done:

break

# Decay the epsilon value for the next episode

epsilon = max(epsilon\_min, epsilon \* epsilon\_decay)

# Print the total reward for this episode

print(f"Episode {episode}: Total reward = {total\_reward}")

# Print the final Q-table

print("Final Q-table:")

print(Q)

SARSA

import numpy as np

import gym

# FrozenLake-v0 gym environment

env = gym.make('FrozenLake-v1')

# Parameters

epsilon = 0.9

total\_episodes = 10000

max\_steps = 100

alpha = 0.05

gamma = 0.95

#Initializing the Q-vaue

Q = np.zeros((env.observation\_space.n, env.action\_space.n))

# Function to choose the next action with episolon greedy

def choose\_action(state):

action=0

if np.random.uniform(0, 1) < epsilon:

action = env.action\_space.sample()

else:

action = np.argmax(Q[state, :])

return action

#Initializing the reward

reward=0

# Starting the SARSA learning

for episode in range(total\_episodes):

t = 0

state1 = env.reset()

action1 = choose\_action(state1)

while t < max\_steps:

# Visualizing the training

# env.render()

# Getting the next state

state2, reward, done, info = env.step(action1)

#Choosing the next action

action2 = choose\_action(state2)

#Learning the Q-value

Q[state1, action1] = Q[state1, action1] + alpha \* (reward + gamma \* Q[state2, action2] - Q[state1, action1])

state1 = state2

action1 = action2

#Updating the respective vaLues

t += 1

reward += 1

#If at the end of learning process

if done:

break

#Evaluating the performance

print ("Performace : ", reward/total\_episodes)

#Visualizing the Q-matrix

print(Q)

Cart Pole

!pip install stable-baselines3

!pip install pyglet

############################

import gym

from stable\_baselines3 import PPO

from stable\_baselines3.common.vec\_env import DummyVecEnv

from stable\_baselines3.common.evaluation import evaluate\_policy

###########################

environment\_name = "CartPole-v0"

env = gym.make(environment\_name)

episodes = 5

for episode in range(1, episodes+1):

state = env.reset()

done = False

score = 0

while not done:

#env.render()

action = env.action\_space.sample()

n\_state, reward, done, info = env.step(action)

score+=reward

print('Episode:{} Score:{}'.format(episode, score))

env.close()

#########################

env = gym.make(environment\_name)

env = DummyVecEnv([lambda: env])

model = PPO('MlpPolicy', env, verbose = 1)

model.learn(total\_timesteps=20000)

##########################

obs = env.reset()

while True:

action, \_states = model.predict(obs)

obs, rewards, done, info = env.step(action)

#env.render()

if done:

print('info', info)

break

env.close()

## **Probability**

### **Rolling a dice twice**

import itertools

dice = [1, 2, 3, 4, 5, 6]

sample\_space = list(itertools.product(dice, repeat=2))

count = 0

for outcome in sample\_space:

total = sum(outcome)

if total % 2 == 0 or total > 7:

count += 1

probability = count / len(sample\_space)

### **balls with replacement**

import itertools

colors = ['W', 'R', 'G']

sample\_space = list(itertools.product(colors, repeat=5))

count = 0

for outcome in sample\_space:

white\_count = outcome.count('W')

red\_count = outcome.count('R')

if white\_count == 3 or red\_count == 2:

count += 1

probability = count / len(sample\_space)

print("Probability of drawing 3 white or 2 red balls:", probability)

### **All 5 are the same color**

import itertools

colors = ['W', 'R', 'G']

sample\_space = list(itertools.product(colors, repeat=5))

count = 0

for color in colors:

outcome = (color,) \* 5

if outcome in sample\_space:

count += 1

probability = count / len(colors)

print("Probability of drawing all 5 balls of the same color:", probability)

## **A U B and A ∩ B**

sample\_space = [(i, j) for i in range(1, 7) for j in range(1, 7)]

A = [(i, j) for (i, j) in sample\_space if (i+j) % 2 == 0]

B = [(i, j) for (i, j) in sample\_space if (i % 3 == 0) or (j % 3 == 0)]

P\_A = len(A) / len(sample\_space)

P\_B = len(B) / len(sample\_space)

A\_and\_B = [outcome for outcome in sample\_space if outcome in A and outcome in B]

P\_B\_given\_A = len(A\_and\_B) / len(A)

P\_A\_or\_B = P\_A + P\_B - len(A\_and\_B) / len(sample\_space)

print("The probability of A ∪ B is:", P\_A\_or\_B)

print("The probability of A ∩ B is:", P\_A \* P\_B\_given\_A)

Possible outcomes of rolling a fair six-sided die

outcomes = (1, 2, 3, 4, 5, 6

# Event A: even numbers

event\_A - (num for num in outcomes if num % 2 -- 0)

# Event B: numbers divisible by 3

event\_B - (num for num in outcomes if num % 3 =- 0)

# Probability of event A

PA- len(event\_ A) / len(outcomes

# Probability of event B

P\_B- len(event\_B) / len(outcomes

# Probability of event A and B

P\_Aand\_B - len(event\_A.intersection(event\_B)) / len(outcones)

# Probability of event A or B

P\_Aor\_B= len(event\_A.union(event\_B)) / len(outcomes

# Print the probabilities

print("P(A U B):", PAor\_B)

print("P(A n B):", P\_A\_and\_B

## **Conditional Prob**

#### **Defective product**

P = [0.3, 0.2, 0.5]

PD = [0.01, 0.03, 0.02]

D = sum([P[j] \* PD[j] for j in range(3)])

P\_Dj = [PD[j] \* P[j] / D for j in range(3)]

most\_likely\_plan = P\_Dj.index(max(P\_Dj)) + 1

print("Plan", most\_likely\_plan, "is most likely responsible for the defective product.")

#### **Other 1**

P\_P1 = 0.3

P\_P2 = 0.2

P\_P3 = 0.5

P\_D\_given\_P1 = 0.01

P\_D\_given\_P2 = 0.03

P\_D\_given\_P3 = 0.02

P\_D = P\_D\_given\_P1 \* P\_P1 + P\_D\_given\_P2 \* P\_P2 + P\_D\_given\_P3 \* P\_P3

P\_P1\_given\_D = P\_D\_given\_P1 \* P\_P1 / P\_D

P\_P2\_given\_D = P\_D\_given\_P2 \* P\_P2 / P\_D

P\_P3\_given\_D = P\_D\_given\_P3 \* P\_P3 / P\_D

print("Probability of using Plan 1 given a defective product:", P\_P1\_given\_D)

print("Probability of using Plan 2 given a defective product:", P\_P2\_given\_D)

print("Probability of using Plan 3 given a defective product:", P\_P3\_given\_D)

#### **Other 2**

prior\_a1 = 0.65

prior\_a2 = 0.35

good\_given\_a1 = 1

bad\_given\_a1 = 0

good\_given\_a2 = 0.9

bad\_given\_a2 = 0.1

prob\_bad = prior\_a1 \* bad\_given\_a1 + prior\_a2 \* bad\_given\_a2

post\_a1\_given\_b = prior\_a1 \* bad\_given\_a1 / prob\_bad

print(f"The probability that the bad part was supplied by supplier 1 is: {post\_a1\_given\_b:.2f}")

prob\_good = prior\_a1 \* good\_given\_a1 + prior\_a2 \* good\_given\_a2

post\_a2\_given\_g = prior\_a2 \* good\_given\_a2 / prob\_good

print(f"The probability that the good part was supplied by supplier 2 is: {post\_a2\_given\_g:.2f}")

## **Bayesian Networks**

#Import required packages

import math

from pomegranate import \*

# Initially the door selected by the guest is completely random

guest =DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )

# The door containing the prize is also a random process

prize =DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )

# The door Monty picks, depends on the choice of the guest and the prize door

monty =ConditionalProbabilityTable(

[[ 'A', 'A', 'A', 0.0 ],

[ 'A', 'A', 'B', 0.5 ],

[ 'A', 'A', 'C', 0.5 ],

[ 'A', 'B', 'A', 0.0 ],

[ 'A', 'B', 'B', 0.0 ],

[ 'A', 'B', 'C', 1.0 ],

[ 'A', 'C', 'A', 0.0 ],

[ 'A', 'C', 'B', 1.0 ],

[ 'A', 'C', 'C', 0.0 ],

[ 'B', 'A', 'A', 0.0 ],

[ 'B', 'A', 'B', 0.0 ],

[ 'B', 'A', 'C', 1.0 ],

[ 'B', 'B', 'A', 0.5 ],

[ 'B', 'B', 'B', 0.0 ],

[ 'B', 'B', 'C', 0.5 ],

[ 'B', 'C', 'A', 1.0 ],

[ 'B', 'C', 'B', 0.0 ],

[ 'B', 'C', 'C', 0.0 ],

[ 'C', 'A', 'A', 0.0 ],

[ 'C', 'A', 'B', 1.0 ],

[ 'C', 'A', 'C', 0.0 ],

[ 'C', 'B', 'A', 1.0 ],

[ 'C', 'B', 'B', 0.0 ],

[ 'C', 'B', 'C', 0.0 ],

[ 'C', 'C', 'A', 0.5 ],

[ 'C', 'C', 'B', 0.5 ],

[ 'C', 'C', 'C', 0.0 ]], [guest, prize] )

d1 = State( guest, name="guest" )

d2 = State( prize, name="prize" )

d3 = State( monty, name="monty" )

#Building the Bayesian Network

network = BayesianNetwork( "Solving the Monty Hall Problem With Bayesian Networks" )

network.add\_states(d1, d2, d3)

network.add\_edge(d1, d3)

network.add\_edge(d2, d3)

network.bake()

beliefs = network.predict\_proba({ 'guest' : 'A' })

beliefs = map(str, beliefs)

print("n".join( "{}t{}".format( state.name, belief ) for state, belief in zip( network.states, beliefs ) ))

## **Bayesian Model**

import pandas as pd

from pgmpy.models import BayesianModel

from pgmpy.estimators import MaximumLikelihoodEstimator, BayesianEstimator

from pgmpy.inference import VariableElimination

data = pd.read\_csv('heart.csv')

model = BayesianModel([('age', 'chol'), ('sex', 'chol'), ('chol', 'target')])

model.fit(data, estimator=MaximumLikelihoodEstimator)

inference = VariableElimination(model)

evidence = {'age': 50, 'sex': 1}

probability = inference.query(variables=['target'], evidence=evidence)

print(probability)

Q1

Linear Regression: Complicated

import numpy as np

import pandas as pd

df1 = pd.read\_csv(r'C:\Users\Mohsin\Desktop\Dataset\_Q1.csv')

df1.head()

####################################################################

# Analyzing Data

####################################################################

import matplotlib.pyplot as plt

# plot the relationship between size and price

plt.scatter(df1['Size (sq ft)'], df1['Price (USD)'])

plt.xlabel('Size (sq ft)')

plt.ylabel('Price (USD)')

plt.show()

####################################################################

# Correlation

###################################################################

import seaborn as sns

plt.figure(figsize=(7,4))

correlation = df1.corr()

sns.heatmap(data=correlation, annot=True)

plt.show()

####################################################################

# Group By

####################################################################

price\_by\_bedrooms = df1.groupby('Bedrooms')['Price (USD)'].mean()

print(price\_by\_bedrooms)

####################################################################

# Standardize

####################################################################

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

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df1\_scaled = pd.DataFrame(scaler.fit\_transform(df1), columns=df1.columns)

# assume your data is stored in a variable called "data"

#X = df1.iloc[:, :-1] # select all columns except the last one as features

#y = df1.iloc[:, -1] # select the last column as the target variable

#X = df1.drop('Price (USD)', axis=1)

# if 1D X = df1\_scaled['Size (sq ft)'].values.reshape(-1,1)

#####################################################################

# Set X & Y

######################################################################

X = df1\_scaled[['Size (sq ft)','Bedrooms']]

y = df1\_scaled['Price (USD)'].values.reshape(-1,1)

#####################################################################

# Split Data

######################################################################

# split the data into training and test sets with a 70/30 ratio

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

X = df1\_scaled[['Size (sq ft)', 'Bedrooms']]

y = df1\_scaled['Price (USD)'].values.reshape(-1, 1)

data\_X = X.values

data\_Y = np.tile(y, 2).reshape(-1, 1)

# plot the relationship between size and price

plt.scatter(data\_X, data\_Y)

plt.xlabel('Size (sq ft) and Bedrooms')

plt.ylabel('Price (USD)')

plt.show()

from sklearn.linear\_model import LinearRegression

model = LinearRegression()

model.fit(X\_train, y\_train)

# Plot the scatter plot

plt.scatter(data\_X, data\_Y, color='blue', label='Actual')

plt.xlabel('Size (sq ft) & Bedrooms')

plt.ylabel('Price (USD)')

# Create a range of values for Size (sq ft) feature

x\_range = np.arange(data\_X.min(), data\_X.max(), 0.1).reshape(-1, 1)

y\_range\_pred = model.predict(np.concatenate((x\_range, np.ones\_like(x\_range)), axis=1))

# Plot the line plot

plt.plot(x\_range, y\_range\_pred, color='red', label='Predicted')

plt.legend()

plt.show()

y\_pred = model.predict(X\_test) # make predictions

# evaluate the model

from sklearn.metrics import mean\_squared\_error, r2\_score,mean\_absolute\_error

mse = mean\_absolute\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print('Mean Absolute error:

Q2 DECISION TREE CLASSIFIER

import seaborn as sns

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

categorical\_cols = df2.select\_dtypes(include=['object'])

ENCODING Categorical To Numeric

from sklearn.preprocessing import LabelEncoder

for col in categorical\_cols:

le = LabelEncoder()

df2[col] = le.fit\_transform(df2[col])

df2

STANDARDIZE

# Scale the numeric features using StandardScaler or MinMaxScaler

num\_cols = ['Age', 'Monthly Charges']

scaler = StandardScaler()

df2\_scaled = df2

df2\_scaled[num\_cols] = pd.DataFrame(scaler.fit\_transform(df2[num\_cols]), columns=num\_cols)

df2\_scaled

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y, test\_size=0.3,random\_state=42)

dtc = DecisionTreeClassifier(criterion = 'entropy', max\_depth=10, random\_state=42)

dtc.fit(X\_train,y\_train)

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

score = accuracy\_score(y\_pred, y\_test)

print(score)

Q3 KNN Algorithim

import seaborn as sns

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

X = df3.iloc[:,1:-1]

y = df3.iloc[:,-1]

x\_train,x\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=0.3,random\_state=42)

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion\_matrix

#Odd Points so take square root of 9 and always take odd points i.e sqrt(144)=12 but take 11

#What is n\_jobs=-1

#what is the plot showing?

knn = KNeighborsClassifier(n\_neighbors=3, n\_jobs=-1)

knn.fit(x\_train,y\_train)

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

score = accuracy\_score(y\_test, y\_pred)

mat = confusion\_matrix(y\_pred, y\_test)

print('Score:', score)

print(mat)

sns.scatterplot(data=x\_train, x='Sepal Length', y='Petal Width', hue=y\_train)

sns.scatterplot(data=x\_test, x='Sepal Length', y='Petal Width', hue=y\_pred, palette='Reds')

Q4 SVM Vector

import seaborn as sns

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

from sklearn.preprocessing import LabelEncoder

df4.drop(columns='ID',inplace=True)

print(df4)

cat\_cols = df4.select\_dtypes(include=['object'])

for col in cat\_cols:

enc = LabelEncoder()

df4[col] = enc.fit\_transform(df4[col])

X = df4.drop(columns=['Fraudulent'])

y = df4['Fraudulent']

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=0.3,random\_state=42)

from sklearn.svm import SVC

#what is c=?

svc = SVC(kernel='linear')

svc.fit(X\_train, y\_train)

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

print(accuracy\_score(y\_pred, y\_test))

Q5

import seaborn as sns

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

from sklearn.preprocessing import LabelEncoder

cat\_cols = df5.select\_dtypes(include=['object'])

for col in cat\_cols:

enc = LabelEncoder()

df5[col] = enc.fit\_transform(df5[col])

X = df5.drop(columns=[' Items Purchased'])

from sklearn.cluster import KMeans

plt.figure(figsize=(5,5))

wss = []

for i in range(1,6):

kmeans = KMeans(n\_clusters=i, init='k-means++' ,random\_state=42)

kmeans.fit\_predict(X)

wss.append(kmeans.inertia\_)

plt.plot(range(1,6),wss)

plt.title("The Elbow Method")

plt.xlabel("Number of clusters")

plt.show()

from sklearn.metrics import silhouette\_score

for i in range(2,7):

kmeans = KMeans(n\_clusters=i)

kmeans.fit(X)

score = silhouette\_score(X,kmeans.labels\_)

print(f"for cluster {i}: The Score is {score}")

km = KMeans(n\_clusters=6, init='k-means++')

y\_means = km.fit\_predict(X)

plt.scatter(X.loc[:,' Age'], X.loc[:,' Total Spending'], c=km.labels\_)

plt.xlabel('Age')

plt.ylabel('Total Spending')

plt.show()