Multi-Armed Bandit Problem with UCB Algorithm P. Venkat Ravi Kumar 2211CS020418

import math import random

class MultiArmedBandit:

def init (self, n\_arms): self.n\_arms = n\_arms

self.counts = [0] \* n\_arms *# Number of times each arm is*

*pulled*

self.values = [0.0] \* n\_arms *# Estimated rewards for each arm*

def select\_arm(self):

total\_counts = sum(self.counts) if total\_counts < self.n\_arms:

return total\_counts *# Explore untried arms first*

ucb\_values = [

self.values[i] + math.sqrt(2 \* math.log(total\_counts) / self.counts[i])

for i in range(self.n\_arms)

]

return ucb\_values.index(max(ucb\_values)) *# Select the arm with max UCB*

def update(self, arm, reward): self.counts[arm] += 1

n = self.counts[arm] value = self.values[arm]

self.values[arm] = ((n - 1) / n) \* value + (1 / n) \* reward

*# Game Simulation*

def simulate\_game(bandit, reward\_probabilities, n\_rounds): total\_reward = 0

for \_ in range(n\_rounds): arm = bandit.select\_arm()

reward = 1 if random.random() < reward\_probabilities[arm] else

0

bandit.update(arm, reward) total\_reward += reward

return total\_reward

*# Example usage*

n\_arms = 3

reward\_probabilities = [0.2, 0.5, 0.8] *# Hidden probabilities of rewards for each arm*

bandit = MultiArmedBandit(n\_arms)

total\_reward = simulate\_game(bandit, reward\_probabilities, 1000) print(f"Total reward after 1000 rounds: {total\_reward}")

Total reward after 1000 rounds: 733

UCB for IoT-Based Smart Home Energy Optimization

import random import math

class SmartHomeOptimizer:

def init (self, n\_devices): self.n\_devices = n\_devices self.counts = [0] \* n\_devices self.efficiency = [0.0] \* n\_devices

def select\_device(self): total\_counts = sum(self.counts)

if total\_counts < self.n\_devices:

return total\_counts *# Explore untried devices*

ucb\_values = [

self.efficiency[i] + math.sqrt(2 \* math.log(total\_counts)

/ self.counts[i])

for i in range(self.n\_devices)

]

return ucb\_values.index(max(ucb\_values))

def update(self, device, efficiency): self.counts[device] += 1

n = self.counts[device]

value = self.efficiency[device]

self.efficiency[device] = ((n - 1) / n) \* value + (1 / n) \* efficiency

*# Simulate real-time energy usage*

def simulate\_energy\_optimizer(optimizer, energy\_efficiency, n\_rounds): total\_efficiency = 0

for \_ in range(n\_rounds):

device = optimizer.select\_device()

efficiency = random.uniform(0, energy\_efficiency[device]) *#*

*Simulate efficiency*

optimizer.update(device, efficiency) total\_efficiency += efficiency

return total\_efficiency

*# Example usage*

n\_devices = 3

energy\_efficiency = [0.6, 0.8, 0.9] *# Max efficiency levels for devices*

optimizer = SmartHomeOptimizer(n\_devices) total\_efficiency = simulate\_energy\_optimizer(optimizer, energy\_efficiency, 1000)

print(f"Total energy efficiency after 1000 rounds:

{total\_efficiency}")

Total energy efficiency after 1000 rounds: 395.82163348576444

Chess-like Game with PAC Algorithm

import numpy as np

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import train\_test\_split from sklearn.metrics import accuracy\_score

*# Simplified chess board states and optimal moves # Example: [state, optimal\_move]*

data = [

[[1, 0, 0, 0], 0], *# Move left*

[[0, 1, 0, 0], 1], *# Move right*

[[0, 0, 1, 0], 2], *# Move forward*

[[0, 0, 0, 1], 3] *# Move backward*

]

*# Prepare dataset*

X = [d[0] for d in data] *# Board states*

y = [d[1] for d in data] *# Optimal moves*

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

*# Train PAC model*

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

*# Evaluate the model*

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

print(f"Accuracy: {accuracy\_score(y\_test, y\_pred)}")

*# Predict next move*

new\_state = [0, 1, 0, 0] *# Example new board state* predicted\_move = model.predict([new\_state])[0] print(f"Predicted move for the new state: {predicted\_move}")

Accuracy: 0.0

Predicted move for the new state: 3