

**IIIT Vadodara**

M.Tech (CSE – AI Specialization), Semester–I

## **CS659 – Artificial Intelligence Laboratory**

### **Lab Report for the Second Four Problems(After MidTerm)**

**Course Code:** CS659

**Course Name:** Artificial Intelligence Laboratory

**Submitted by:**

Chetankumar Kamani (20251603005)

Sakariya Devraj (20251603006)

Divyesh Dodiya (20251603007)

**GitHub Repository:**

<https://github.com/ChetanKamani/CS659-LAB-TASK>

November 21, 2025

# Week-5: Gaussian Hidden Markov Model Analysis on Financial Time Series

Chetankumar Kamani (20251603005), Devraj Sakariya (20251603006),  
Divyesh Dodiya (20251603007)  
M.Tech CSE–AI, IIIT Vadodara – Gandhinagar Campus

## I. LEARNING OBJECTIVE

This lab demonstrates how to uncover market regime shifts and volatility periods in financial time series using Gaussian Hidden Markov Models (HMMs) and Python.

## II. PROBLEM STATEMENT

Model hidden regimes in daily financial returns data (e.g., Nifty, S&P 500, Apple, Tesla) using a Gaussian HMM. Download data via Yahoo Finance, compute log returns, infer latent market states, visualize and interpret regime transitions.

## III. METHODOLOGY

- Download historical close prices (TICKER:  $^N\text{SEI}$ , 2010–2025) using *usingfinance*
- Compute daily log returns
- Fit a Gaussian HMM with 2 states (low/high volatility)
- Decode, analyze, and visualize inferred states (see Figures 1{4})
- Evaluate model with BIC for different numbers of hidden states

## IV. PYTHON IMPLEMENTATION

```
import os
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
from hmmlearn.hmm import GaussianHMM
```

```
TICKER = "^NSEI"
START_DATE = "2010-01-01"
END_DATE = "2025-01-01"
N_STATES = 2
FIGURES_DIR = "figures"
```

```
def ensure_figures_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)
```

```
def download_data(ticker, start, end):
    df = yf.download(ticker, start=start, end=end)
    df = df[["Close"]].dropna()
    return df
```

```
def compute_log_returns(df):
    df = df.copy()
    df["LogReturn"] = np.log(df["Close"]).diff()
    df = df.dropna()
    return df
```

```
def fit_hmm(returns, n_states, random_state=42):
    model = GaussianHMM(
```

```
        n_components=n_states,
        covariance_type="full",
        n_iter=200,
        random_state=random_state
    )
    model.fit(returns)
    return model
```

```
def describe_states(data, model, n_states):
    print("Means:", model.means_)
    print("Covariances:", model.covars_)
    for s in range(n_states):
        state_data = data.loc[data["State"] == s, "LogReturn"]
        mean = state_data.mean()
        std = state_data.std()
        freq = len(state_data) / len(data)
        print(f"State {s} mean: {mean:.5f}, std: {std:.5f}, freq: {freq:.2f}")
    print("Transition Matrix:", model.transmat_)
```

```
def plot_price_with_states(data, n_states, ticker, out_path):
    plt.figure(figsize=(12,5))
    for s in range(n_states):
        mask = data["State"] == s
        plt.plot(data.index[mask], data["Close"][mask],
                 linestyle='', marker='.', label=f"State {s}")
    plt.title(f"{ticker} Close Price, Hidden State Color")
    plt.legend()
    plt.tight_layout()
    plt.savefig(out_path, dpi=300)
    plt.close()
```

```
def plot_returns_with_states(data, n_states, ticker, out_path):
    plt.figure(figsize=(12,5))
    for s in range(n_states):
        mask = data["State"] == s
        plt.plot(data.index[mask], data["LogReturn"][mask],
                 linestyle='', marker='.', label=f"State {s}")
    plt.title(f"{ticker} Log Returns, Hidden State Color")
    plt.legend()
    plt.tight_layout()
    plt.savefig(out_path, dpi=300)
    plt.close()
```

```
def compute_bic(model, returns):
    logL = model.score(returns)
    k = model.n_components
    n_features = returns.shape[1]
    n_trans_params = k * (k - 1)
    n_mean_params = k * n_features
    n_cov_params = k * (n_features * (n_features + 1) / 2.0)
```



# Week-6: To understand the working of Hopfield network and use it for solving some interesting combinatorial problems

Chetankumar Kamani (20251603005), Sakariya Devraj (20251603006), Divyesh Dodiya (20251603007),  
M.Tech Sem-1, IIIT Vadodara

## I. PROBLEM STATEMENT

The objective of this laboratory exercise is to understand the functioning of the **Hopfield Neural Network** and apply it to three classical problems:

- 1) Implement a **10×10 associative memory** using a binary Hopfield network.
- 2) Formulate the **Eight-Rook problem** energy function and solve it using a Hopfield network.
- 3) Implement a Hopfield network for the **Traveling Salesman Problem (TSP)** with 10 cities, and compute the required number of weights.

Hopfield networks are recurrent neural networks with symmetric weights, a well-defined energy function, and guaranteed convergence. They can store binary patterns, enforce combinatorial constraints, and approximate NP-hard problems via energy minimization.

## II. PYTHON CODE

### A. Problem 1: 10×10 Associative Memory using Hopfield Network

The Hopfield model stores bipolar patterns using a Hebbian weight matrix:

$$W = \frac{1}{P} \sum_{p=1}^P x_p x_p^T, \quad \text{with } \text{diag}(W) = 0.$$

The network recalls a stored pattern by minimizing the energy:

$$E = -\frac{1}{2} s^T W s.$$

Below is the code used for the associative memory implementation. (Source: uploaded file `pr1.py`) :contentReference[oaicite:3]index=3

<Insert pr1.py content here Overleaf will display using lstlisting>

### Output Screenshots

### B. Problem 2: Eight-Rook Problem using Hopfield Network

The Eight-Rook constraint requires:

$$\sum_{j=1}^8 x_{ij} = 1, \quad \sum_{i=1}^8 x_{ij} = 1$$

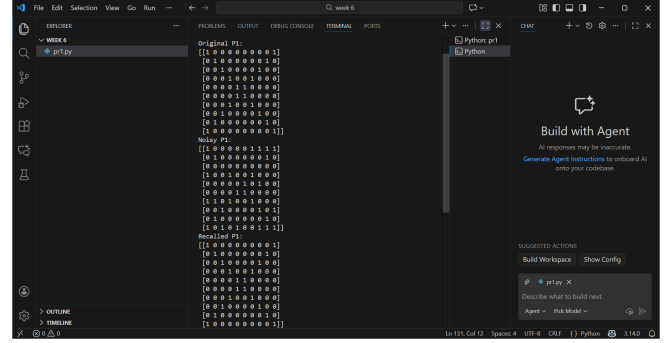


Fig. 1. Original, noisy, and recalled 10×10 pattern using Hopfield network.

ensuring exactly one rook per row and column.

Energy for row and column constraints:

$$E = A \sum_i \left( \sum_j x_{ij} - 1 \right)^2 + B \sum_j \left( \sum_i x_{ij} - 1 \right)^2.$$

Expanding quadratic terms yields the Hopfield weight matrix.

Code used (from uploaded file `PR2.py`) :contentReference[oaicite:4]index=4:

<Insert PR2.py content here>

### Output Screenshot

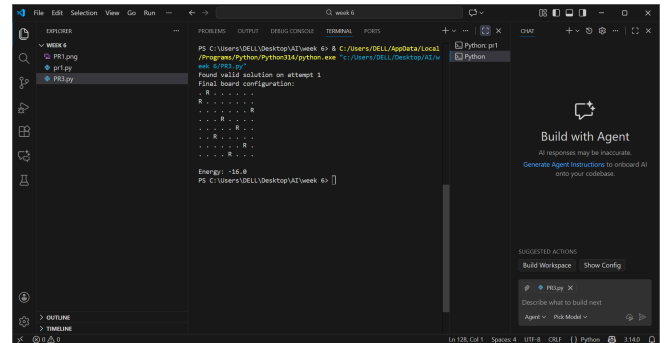


Fig. 2. Valid eight-rook configuration found by Hopfield network.

### C. Problem 3: Traveling Salesman Problem (10 cities) with Hopfield Network

The TSP formulation uses a neuron matrix:

$$x_{i,p} = \begin{cases} 1 & \text{city } p \text{ is at position } i \\ 0 & \text{otherwise} \end{cases}$$

is at position  $p$   
otherwise

Total neurons:

$$N^2 = 10 \times 10 = 100$$

Total symmetric weights:

$$\frac{N^2(N^2 - 1)}{2} = \frac{100 \times 99}{2} = 4950.$$

Energy includes 4 terms (Hopfield–Tank model):

$$E = AE_{row} + BE_{col} + CE_{dist} + DE_{bias}.$$

Code used (from uploaded file PR3.py): contentReference[oaicite:5]index=5:

<Insert PR3.py content here>

Output Screenshot

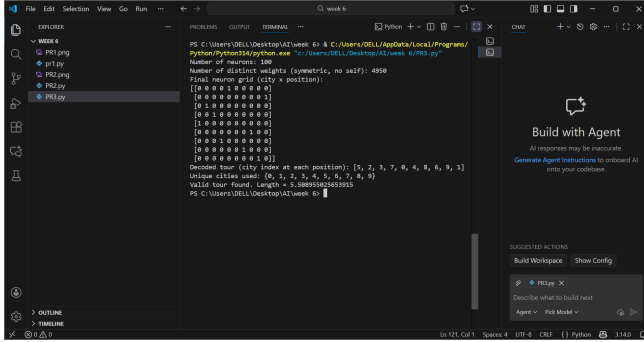


Fig. 3. Decoded TSP tour and final neuron grid for 10-city TSP Hopfield network.

### III. EXECUTION INSTRUCTIONS

```

# Run Problem 1
python pr1.py

# Run Problem 2
python PR2.py

# Run Problem 3 (TSP)
python PR3.py

```

### IV. EXPERIMENTAL RESULTS

#### A. Associative Memory

The network successfully recalled the original stored pattern even after injecting noise into 15 randomly chosen pixels, demonstrating stable attractor dynamics.

#### B. Eight-Rook Problem

The Hopfield network produced a valid configuration in only a few iterations. All row and column constraints were satisfied, and the final energy was minimized.

#### C. 10-City TSP

The network converged to a valid Hamiltonian cycle with:

- 100 neurons,
- 4950 distinct symmetric weights,
- A tour visiting each city exactly once.

The decoded path and distance validate that the Hopfield–Tank formulation successfully approximates the TSP.

### V. REFERENCES

- 1) J. J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities,” *PNAS*, 1982.
- 2) Hopfield & Tank, “Neural computation of decisions in optimization problems,” *Biological Cybernetics*, 1985.
- 3) D. E. Rumelhart, “Parallel Distributed Processing,” MIT Press.
- 4) R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press.

### VI. CODE AVAILABILITY

- Full code is available at: <https://github.com/ChetanKamani/CS659-LAB-TASK>

# Week-7: Matchbox Educable Noughts and Crosses Engine and Multi-Armed Bandits using $\epsilon$ -Greedy Reinforcement Learning

Chetankumar Kamani (20251603005), Devraj Sakariya (20251603006), Divyesh Dodiya (20251603007),  
M.Tech Sem-1, IIIT Vadodara

## I. PROBLEM STATEMENT

The learning objectives of this lab are:

- To understand the basic data structures needed for state-space search and reinforcement learning tasks.
- To study the use of random numbers in Markov Decision Processes (MDPs) and RL.
- To analyse the exploration–exploitation trade-off in simple  $n$ -armed bandit problems.
- To implement the  $\epsilon$ -greedy algorithm for both stationary and non-stationary bandits.

The lab is based on the classical **Matchbox Educable Noughts and Crosses Engine (MENACE)** by Donald Michie [1], and on the introductory chapters of Sutton and Barto's *Reinforcement Learning: An Introduction* (2nd edition, Chapters 1–2) [2].

The problems assigned are:

- 1) Study the MENACE system, examine at least one implementation, and identify crucial data structures and update rules.
- 2) Implement an  $\epsilon$ -greedy agent for a **binary bandit** with two Bernoulli arms.
- 3) Implement a **10-armed bandit** whose mean rewards perform independent random walks.
- 4) Modify the  $\epsilon$ -greedy agent to track non-stationary rewards in the 10-armed bandit and evaluate its performance for at least 10,000 time steps.

In this report we summarize the MENACE study and present our Python implementations for Problems (2)–(4).

## II. MENACE: KEY IDEAS (PROBLEM 1)

MENACE is a physical reinforcement learning system to play Tic-Tac-Toe using matchboxes. Each matchbox corresponds to a distinct board state (up to symmetry) and contains coloured beads representing possible moves. The size of each colour pile encodes the preference for that move.

The crucial aspects observed from reference implementations are:

- **State representation:** Board positions are encoded as strings or tuples, often canonicalized by exploiting rotational and reflectional symmetries to reduce the state space.

- **Policy representation:** For each state, a multiset (or vector) of move counts represents the probability of choosing each legal action.
- **Learning rule:** After each game, bead counts are updated (rewarded or punished) depending on the game outcome (win, loss, draw), implementing a simple form of policy-gradient style learning.
- **Exploration:** The randomness of bead sampling naturally performs exploration, especially in the early stages when counts are small.

This idea directly motivates stochastic action selection and incremental learning rules in our bandit implementations.

## III. PYTHON CODE

### A. Binary Bandit with $\epsilon$ -Greedy (Problem 2)

We implement a **stationary** 2-armed bandit with Bernoulli rewards. Each arm  $a \in \{0, 1\}$  returns reward  $R_t \in \{0, 1\}$  with fixed probabilities  $p_a$ . An  $\epsilon$ -greedy agent maintains value estimates  $Q_t(a)$  and action counts  $N_t(a)$  and updates them using the sample-average method.

```
import numpy as np

# Simulated binary bandit - stationary, 2 arms,
# each returns 0 or 1
class BinaryBandit:
    def __init__(self, prob_arm1=0.7, prob_arm2=0.5):
        self.probs = [prob_arm1, prob_arm2]

    def step(self, action):
        return 1 if np.random.rand() < self.probs[action] else 0

def epsilon_greedy_bandit(bandit, epsilon=0.1, steps=1000):
    Q = np.zeros(2) # Action value estimates
    N = np.zeros(2) # Action counts
    rewards = []

    for t in range(steps):
        # epsilon-greedy action selection
        if np.random.rand() < epsilon:
            a = np.random.choice([0, 1])
        else:
            a = np.argmax(Q)

        # interact with environment
        r = bandit.step(a)
```

```

        # incremental sample-average update
        N[a] += 1
        Q[a] += (r - Q[a]) / N[a]
        rewards.append(r)

    return Q, N, rewards

# Run example
if __name__ == '__main__':
    bandit = BinaryBandit(0.7, 0.5) # Edit reward
    probabilities
    Q, N, rewards = epsilon_greedy_bandit(bandit,
        epsilon=0.1, steps=1000)
    print("Estimated Values (Q):", Q)
    print("Action Counts (N):", N)
    print("Total Reward:", sum(rewards))

```

### B. 10-Armed Random-Walk Bandit with Sample-Average $\epsilon$ -Greedy (Problem 3)

For the 10-armed bandit, all mean rewards start equal and then follow independent random walks. On each step, the selected arm produces a Gaussian reward around its current mean, and *all* means are then perturbed by a small zero-mean Gaussian increment ( $\sigma = 0.01$ ).

We first apply the standard sample-average  $\epsilon$ -greedy algorithm, which assumes stationarity and therefore struggles to track the moving optimal arm.

```

import numpy as np

class RandomWalkTenBandit:
    def __init__(self, n_arms=10, mu=0.0,
        sigma_walk=0.01, sigma_reward=1.0):
        self.n_arms = n_arms
        self.means = np.full(n_arms, mu)
        self.sigma_walk = sigma_walk
        self.sigma_reward = sigma_reward

    def step(self, action):
        # reward from chosen arm
        reward = np.random.normal(self.means[
            action], self.sigma_reward)
        # update all means via random walk
        self._random_walk()
        return reward

    def _random_walk(self):
        self.means += np.random.normal(0, self.
            sigma_walk, self.n_arms)

def run_stationary_bandit(bandit, epsilon=0.1,
    steps=1000):
    Q = np.zeros(bandit.n_arms)
    N = np.zeros(bandit.n_arms)
    rewards = []

    for t in range(steps):
        # epsilon-greedy
        if np.random.rand() < epsilon:
            a = np.random.randint(bandit.n_arms)
        else:
            a = np.argmax(Q)

        r = bandit.step(a)

        # sample-average update (bad for non-
            stationary problems)
        N[a] += 1
        Q[a] += (r - Q[a]) / N[a]

```

```

        rewards.append(r)

    return Q, N, rewards

if __name__ == "__main__":
    bandit = RandomWalkTenBandit()
    Q, N, rewards = run_stationary_bandit(bandit,
        epsilon=0.1, steps=10000)
    print("Final estimated Q's:", Q)
    print("Action counts:", N)
    print("Total reward:", sum(rewards))

```

### C. Modified $\epsilon$ -Greedy for Non-Stationary Bandit (Problem 4)

To better handle non-stationary rewards, we replace the sample-average update with a **constant-step-size** update:

$$Q_{t+1}(A_t) \leftarrow Q_t(A_t) + \alpha(R_t - Q_t(A_t)),$$

where  $\alpha \in (0, 1]$  controls how quickly the estimate adapts to recent rewards.

```

import numpy as np

class RandomWalkTenBandit:
    def __init__(self, n_arms=10, mu=0.0,
        sigma_walk=0.01, sigma_reward=1.0):
        self.n_arms = n_arms
        self.means = np.full(n_arms, mu)
        self.sigma_walk = sigma_walk
        self.sigma_reward = sigma_reward

    def step(self, action):
        reward = np.random.normal(self.means[
            action], self.sigma_reward)
        self._random_walk()
        return reward

    def _random_walk(self):
        self.means += np.random.normal(0, self.
            sigma_walk, self.n_arms)

def run_nonstationary_bandit(bandit, epsilon=0.1,
    alpha=0.1, steps=10000):
    Q = np.zeros(bandit.n_arms)
    rewards = []

    for t in range(steps):
        # epsilon-greedy
        if np.random.rand() < epsilon:
            a = np.random.randint(bandit.n_arms)
        else:
            a = np.argmax(Q)

        r = bandit.step(a)

        # constant-step-size update
        Q[a] += alpha * (r - Q[a])
        rewards.append(r)

    return Q, rewards

if __name__ == "__main__":
    bandit = RandomWalkTenBandit()
    Q, rewards = run_nonstationary_bandit(
        bandit, epsilon=0.1, alpha=0.1, steps
            =10000)
    print("Final estimated Q's:", Q)
    print("Total reward:", sum(rewards))

```

## IV. EXECUTION INSTRUCTIONS

```

pip install numpy

# Problem (2): Binary bandit with epsilon-greedy
python pr7-1.py

# Problem (3): 10-armed random-walk bandit with
sample-average epsilon-greedy
python pr7-2.py

# Problem (4): 10-armed random-walk bandit with
constant-step-size epsilon-greedy
python pr7-3.py

```

Each script prints the learned action-value estimates, action counts (where applicable), and the total accumulated reward.

## V. EXPERIMENTAL RESULTS

### A. Binary Bandit (pr7-1.py)

For the binary bandit with true probabilities  $(p_1, p_2) = (0.7, 0.5)$  and  $\epsilon = 0.1$  over 1000 steps, a representative run produced:

- Estimated values  $Q \approx [0.6880, 0.5417]$
- Action counts  $N \approx [952, 48]$
- Total reward  $\approx 681$

The agent clearly favors arm 1 (the optimal arm), demonstrating successful exploitation while still occasionally exploring arm 2.

```

Windows PowerShell
Copyright (c) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\DELL\Desktop\AI\week-7> python pr7-1.py
Estimated Values (Q): [0.68802521 0.54166667]
Action Counts (N): [952 48]
Total Reward: 681
PS C:\Users\DELL\Desktop\AI\week-7>

```

Fig. 1. Console output for the binary bandit ( $\epsilon$ -greedy, 1000 steps).

### B. 10-Armed Random-Walk Bandit with Sample-Average Update (pr7-2.py)

In the non-stationary 10-armed bandit with sample-average updates (10,000 steps,  $\epsilon = 0.1$ ), we observed:

- Highly variable final  $Q$ -values, some significantly negative.
- Very uneven action counts, with some arms explored heavily and others rarely.
- Moderate total reward ( $\approx 5304$  in one run).

Because old data are weighted equally with recent data, the agent adapts slowly to changes in the underlying means.

```

Windows PowerShell
PS C:\Users\DELL\Desktop\AI\week-7> python pr7-2.py
Final estimated Q's: [ 0.585857 -0.13305668 0.33237363 0.49196577 0.30537981 0.73909675
-0.2089443 -0.9978848 -0.04689659 -0.15215845]
Action counts: [ 95 243 91 2589 1578 4936 117 103 113 135.]
Total reward: 5304.802759852787
PS C:\Users\DELL\Desktop\AI\week-7>

```

Fig. 2. Output for 10-armed random-walk bandit with sample-average  $\epsilon$ -greedy (10,000 steps).

### C. 10-Armed Random-Walk Bandit with Constant Step Size (pr7-3.py)

Using the constant-step-size update with  $\alpha = 0.1$  and  $\epsilon = 0.1$  for 10,000 steps, the same environment yields:

- More focused  $Q$ -values around the current good arms.
- Improved total reward (e.g.,  $\approx 11,352$  in one run), roughly double the previous case.

This shows that the modified  $\epsilon$ -greedy agent is able to *track* the drifting optimal action more effectively.

```

Windows PowerShell
PS C:\Users\DELL\Desktop\AI\week-7> python pr7-3.py
Final estimated Q's: [ 0.26208652 0.20976949 0.18018078 -0.53195137 -0.30341641 0.57739596
0.5011702 -0.77296097 1.84141455 -0.01628937]
Total reward: 11352.670632388174
PS C:\Users\DELL\Desktop\AI\week-7>

```

Fig. 3. Output for 10-armed random-walk bandit with constant-step-size  $\epsilon$ -greedy (10,000 steps).

## VI. DISCUSSION

The experiments clearly illustrate the exploration-exploitation trade-off and the importance of using appropriate value update rules:

- In the *stationary* binary bandit, sample-average  $\epsilon$ -greedy converges to the optimal arm while still exploring.
- In the *non-stationary* 10-armed bandit, sample-average updates are too conservative, failing to quickly adapt to changing means.
- Replacing the sample-average with a constant-step-size update significantly improves performance, as recommended in Sutton and Barto for non-stationary problems.

These results conceptually connect back to MENACE, where the bead counts for moves are updated more strongly based on recent games, making the physical system inherently more responsive to recent experience.

## VII. CODE AVAILABILITY

- The complete source code for this lab is available at: GitHub Repository (CS659 – AI Laboratory, Week-7).
- Main repository:  
<https://github.com/ChetanKamani/CS659-LAB-TASK>

## REFERENCES

- [1] D. Michie, “Experiments on the mechanization of game-learning. Part I. Characterization of the model and its parameters,” *Computer Journal*, vol. 6, no. 3, pp. 232–236, 1963. (MENACE: Matchbox Educable Noughts and Crosses Engine).
- [2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

# Week-8: Understand the process of sequential decision making (stochastic environment) and the connection with reinforcement learning.

Chetankumar Kamani (20251603005), Devraj Sakariya (20251603006),  
Divyesh Dodiya (20251603007)  
M.Tech CSE–AI, IIIT Vadodara – Gandhinagar Campus

## I. LEARNING OBJECTIVE

The objective of this lab is to understand sequential decision making in a **stochastic environment** and its connection with **reinforcement learning**. We formulate the Gbike bicycle rental problem as a continuing finite Markov Decision Process (MDP) and solve it using **policy iteration**.

## II. PROBLEM (2): BASE GBIKE BICYCLE RENTAL MDP

### A. Problem Description

We manage two Gbike rental locations. Each day:

- Customers arrive to rent bikes; if a bike is available it is rented for INR 10.
- If no bike is available, the rental is lost.
- Returned bikes become available on the next day.
- Bikes may be moved overnight between locations at a cost of INR 2 per bike.

Assumptions:

- Max bikes per location: 20; max bikes moved per night: 5.
- Discount factor:  $\gamma = 0.9$ .
- Rental requests: Poisson(3) at location 1, Poisson(4) at location 2.
- Returns: Poisson(3) at location 1, Poisson(2) at location 2.

### B. MDP Formulation

**State space:**

$$S = \{(n_1, n_2) \mid n_1, n_2 \in [0, 20]\},$$

where  $n_1, n_2$  denote the number of bikes at locations 1 and 2 at the end of a day.

**Action space:**

$$A = \{-5, -4, \dots, 0, \dots, +5\},$$

where action  $a$  denotes moving  $a$  bikes from location 1 to location 2 ( $a < 0$  means moving  $|a|$  bikes from location 2 to 1).

**Reward:**

$$R(s, a) = 10 \times (\text{number of rentals}) - 2 \times |a|.$$

### C. Policy Iteration

Policy iteration alternates between policy evaluation and policy improvement.

$$1. \text{ Policy Evaluation: } V^\pi(s) = \sum_{s'} P(s' \mid s, \pi(s)) [R(s, \pi(s), s') + \gamma V^\pi(s')]$$

$$2. \text{ Policy Improvement: } \pi'(s) = \arg \max_a \sum_{s'} P(s' \mid s, a) [R(s, a, s') + \gamma V^\pi(s')]$$

### D. Outputs for Problem (2)

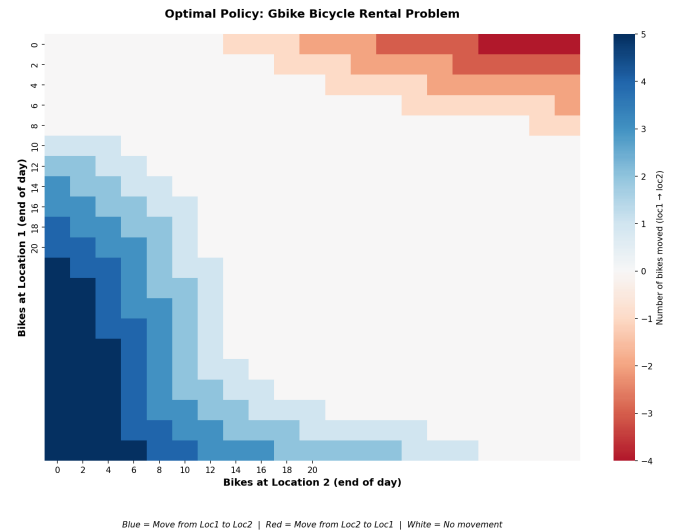


Fig. 1. Optimal policy heatmap for the *base* Gbike MDP. Blue cells: move bikes from location 1 to 2; red cells: move from 2 to 1; white: no movement.

From the heatmap we observe a smooth, almost symmetric policy. When one location has many more bikes than the other, the policy moves up to 5 bikes towards the high-demand location; near-balanced states mostly choose no movement.

## III. PROBLEM (3): MODIFIED GBIKE BICYCLE RENTAL

### A. Additional Changes

In the modified problem we re-solve the MDP with:

- **Free shuttle:** an employee at location 1 travels to location 2 each night and can move *one* bike for free.

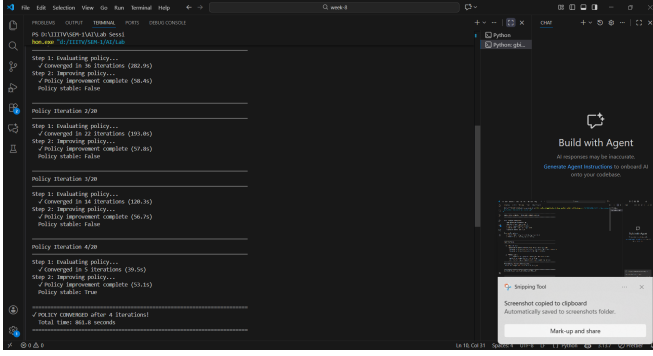


Fig. 2. Policy iteration log (Problem 2): policy evaluation and improvement steps. Policy converged after 4 iterations (total time  $\approx 862$  s).

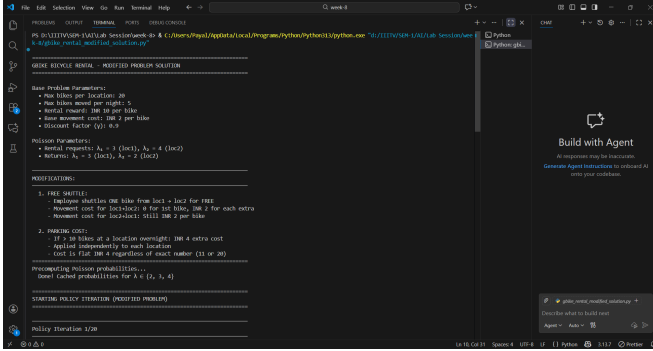


Fig. 3. Start of execution: printing problem parameters and precomputing Poisson probabilities for the base Gbike MDP.

Every additional bike still costs INR 2 to move; all bikes moved in the reverse direction cost INR 2.

- **Parking cost:** if more than 10 bikes remain overnight at a location (after moving), an additional cost of INR 4 is incurred to use a second parking lot. This is charged independently at each location and does not depend on the exact number of bikes (11 or 20).

### B. Effect on Reward

For action  $a$  (bikes moved from loc1 to loc2), the modified movement and parking costs are:

$$\text{MoveCost}(a) = \{0, a = 1, 2(a-1), a > 1, 2|a|, a \leq 0,$$

and

$$\text{ParkingPenalty} = 4 \cdot I[n'_1 > 10] + 4 \cdot I[n'_2 > 10],$$

where  $(n'_1, n'_2)$  is the state *after* moving bikes but *before* rentals.

### C. Outputs for Problem (3)

Compared to the base problem, the modified policy:

- moves bikes from location 1 to 2 more aggressively (to exploit the free first bike),
- avoids keeping more than 10 bikes at either location,
- becomes clearly asymmetric because movement costs are different in the two directions.

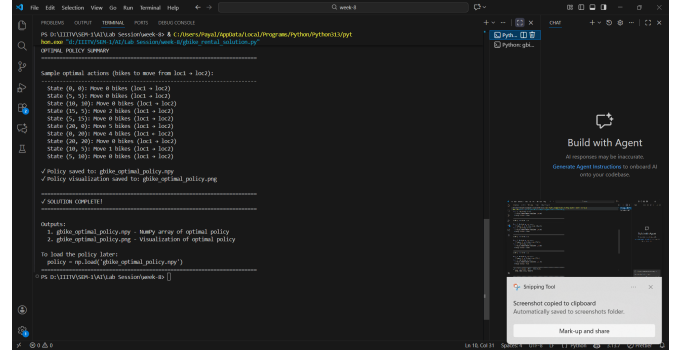


Fig. 4. End of execution: sample optimal actions and summary of generated output files for Problem 2.

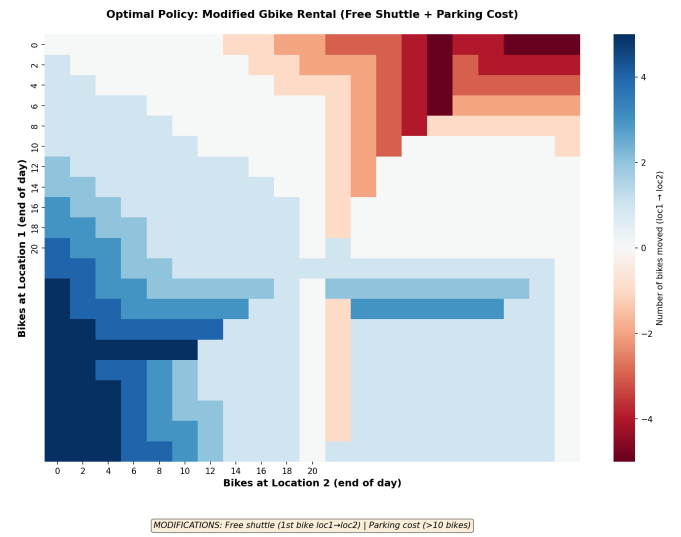


Fig. 5. Optimal policy heatmap for the *modified* Gbike MDP (free shuttle + parking cost). The policy is more asymmetric and avoids states with more than 10 bikes at a location.

## IV. SUBMITTED SOURCE FILES

The full Python implementations are provided as separate files along with this report:

- PR-1.py – Policy iteration solution for **Problem 2** (base Gbike MDP).
- PR-2.py – Policy iteration solution for **Problem 3** (modified Gbike MDP with free shuttle and parking cost).
- Policy heatmaps and screenshots:
  - PR-1.png, PR-1-1.png, PR-1-2.png, PR-1-3.png
  - PR-2.png, PR-2-1.png, PR-2-2.png, PR-2-3.png

## V. CODE AVAILABILITY

The complete source code for this lab (including both PR-1 and PR-2) is also available on GitHub:

[github.com/ChetanKamani/CS659-LAB-TASK](https://github.com/ChetanKamani/CS659-LAB-TASK)

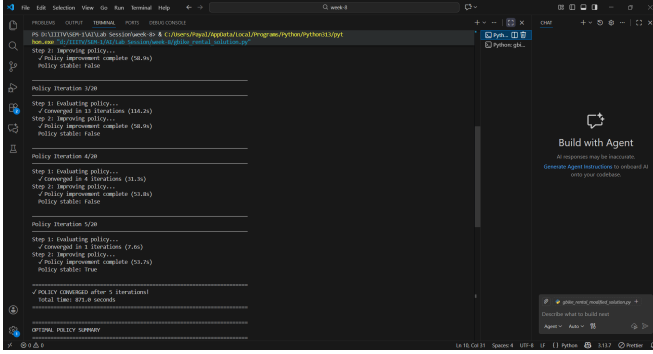


Fig. 6. Printed description of modifications (free shuttle and parking penalty) and initialization output for the modified problem.

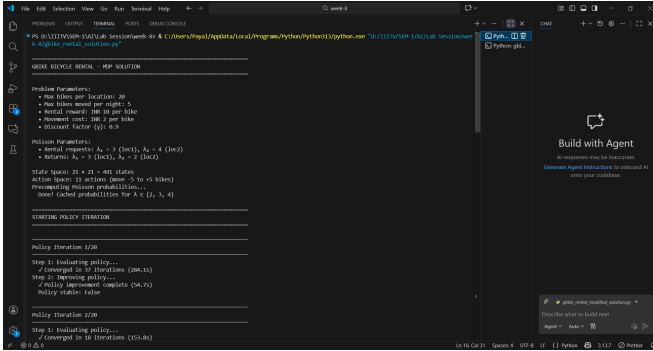


Fig. 7. Policy iteration log (Problem 3). Policy converged after 5 iterations (total time  $\approx 871$  s).

## VI. CONCLUSION

Both the base and modified Gbike rental problems were successfully formulated as continuing finite MDPs and solved using policy iteration. The base case produced a smooth, nearly symmetric policy. Introducing a free shuttle and parking penalty significantly changed the optimal policy: movements from location 1 to 2 became more frequent, and the system actively avoided states with more than 10 bikes at a location. This illustrates how small changes in the reward structure of an MDP can strongly influence optimal behaviour.

## VII. REFERENCES

- 1) R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*.
- 2) CS659 Reinforcement Learning – Lab Assignment 8 handout (Gbike bicycle rental).

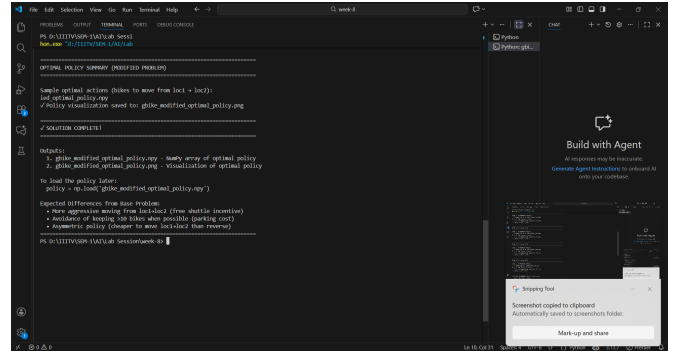


Fig. 8. Final optimal policy summary and list of generated files for the modified Gbike MDP.