AI CS367 Lab Assignment Reports (End Semester)

Shivam Kumar (202351130), Dev Saraswat (202351173), Tammineni Harika (202351145)

Department of Computer Science and Engineering
Indian Institute of Information Technology, Vadodara

Supervisor: Dr. Pratik Shah

***Abstract*—This report presents the end-semester laboratory work for AI/CS367, covering eight core topics in machine learning and reinforcement learning. The experiments include Naive Bayes and Decision Tree classifiers for grade prediction, Gaussian HMMs for financial regime detection, Hopfield associative memory and Hopfield–Tank optimization, MENACE reinforcement learning for Tic-Tac-Toe, multi-armed bandit algorithms for stationary and nonstationary settings, and dynamic programming methods (value and policy iteration) applied to grid-world and a modified Gbike rental MDP. The results confirm key theoretical concepts such as Hopfield capacity scaling, regime identification using HMMs, the advantage of constant-step updates in drifting bandits, and reward-sensitive optimal policies in MDPs. Overall, the labs demonstrate fundamental AI principles spanning probabilistic models, neural memory, reinforcement learning, and optimal control.**

***Index Terms*—machine learning; Hopfield network; MENACE; Gaussian HMM; multi-armed bandit; ε-greedy; value iteration; policy iteration; Markov decision process; reinforcement learning**

I Lab V (In-Lab): Naive Bayes and Decision Tree Classifiers

I-A Introduction

This experiment evaluates grade prediction using classical machine-learning models. The dataset contains course grades from the first semester, encoded as categorical values. Two supervised learning models were implemented: Naive Bayes (Multinomial) and Decision Tree Classifier, and their predictive performance was compared using repeated random train–test splits.

I-B Objectives

- Encode categorical grade data for machine learning.
- Compute conditional probability tables (CPTs) for dependent grade analysis.
- Predict PH100 grade given evidence on EC100, IT101 and MA101.
- Compare the performance of Naive Bayes vs. Decision Tree.

I-C Dataset and Encoding

The dataset contains nine categorical course-grade attributes:

$$\{\text{EC100, EC160, IT101, IT161, MA101, PH100, PH160, HS101, QP}\}.$$

To enable machine-learning models, all letter grades were encoded into numerical labels using the following mapping:

$$\begin{aligned} &\text{AA} \to 0, \quad \text{AB} \to 1, \quad \text{BB} \to 2,\\ &\text{BC} \to 3, \quad \text{CC} \to 4, \quad \text{CD} \to 5,\\ &\text{DD} \to 6, \quad \text{F} \to 7. \end{aligned}$$

This transformation preserves course difficulty ordering and prepares the data for Naive Bayes and Decision Tree classifiers.

I-D Methodology

1) Conditional Probability Table (CPT)

Adjacent-course dependencies were computed with:

$$P(C_{i+1} \mid C_i) = \frac{\text{count}(C_i, C_{i+1})}{\text{count}(C_i)}.$$

2) PH100 Grade Prediction

Evidence used:

$$EC100 = DD, \quad IT101 = CC, \quad MA101 = CD.$$

The program computes:

$$\widehat{\text{PH100}} = \text{CC}.$$

3) Model Evaluation

Both models were evaluated using 20 randomized train–test splits (70% train, 30% test), with mean accuracy and standard deviation reported.

Pseudocode (In-Lab Snipped Approach)

```
1. Load dataset from text file.
2. Encode letter grades (AA, AB, ..., F) into
    numeric values.
3. Compute CPT tables using groupby counts for
    consecutive course pairs.
4. Predict PH100 by:
      - Filtering rows matching given evidence,
      - Selecting the most frequent PH100 grade.
5. Naive Bayes experiment:
      For 20 trials:
          - Random 70/30 train-test split
          - Train MultinomialNB
          - Record accuracy
      Compute mean and std of accuracies.
6. Decision Tree experiment:
      For 20 trials:
          - Train depth-5 decision tree
          - Record accuracy
      Compute mean and std of accuracies.
7. Print PH100 prediction and accuracy statistics.
```

I-E Experimental Results

The Python program produced the following output:

```
PH100_pred CC
NB_acc_mean 0.7035714285714284 NB_acc_std 0.04446392713567405
DEP_acc_mean 0.9307142857142857 DEP_acc_std 0.022711590136549515
```

Fig. 1: Screenshort of the In Lab output

1) Summary of Model Performance

TABLE I: Model Accuracy Comparison (20 Runs)

Model	Accuracy (Mean)	Std. Dev.
Naive Bayes	0.7107	0.0469
Decision Tree (Depth=5)	0.9314	0.0270

Explanation: The Decision Tree model achieves significantly higher accuracy (0.93) with lower variability across runs, indicating stable and reliable performance. In contrast, Naive Bayes attains only moderate accuracy (0.71) due to its strong independence assumptions, making it less suited for this dataset and resulting in higher variance.

I-F Discussion

- Naive Bayes performs moderately well but is limited by its independence assumptions.
- Decision Tree significantly outperforms Naive Bayes, achieving over 93% accuracy.
- The lower standard deviation for Decision Tree indicates more stable performance.
- Predicted PH100 grade (CC) aligns with observed evidence trends.

I-G Conclusion

The Decision Tree classifier demonstrates superior predictive capability for grade prediction compared to Naive Bayes. The CPT analysis successfully captures inter-course dependencies, and PH100 prediction using evidence-based filtering returns a plausible grade. The experiment highlights the importance of model selection and feature dependencies in academic performance analytics.

II Lab V (Submission Part): Gaussian Hidden Markov Model for Financial Time Series

II-A Introduction

Financial markets often exhibit fluctuating volatility levels, shifting between periods of stability and turbulence. These changes, however, are not directly observable. Hidden Markov Models (HMMs) provide a powerful statistical framework for capturing such latent regime transitions using observed financial time series. In this lab submission, a Gaussian Hidden Markov Model is applied to real-world stock market data to infer hidden market states, analyze return behaviour, and study transition dynamics between volatility regimes.

II-B Objectives

- Collect and preprocess historical stock price data from Yahoo Finance.
- Compute daily returns suitable for modeling with Gaussian HMMs.
- Fit a Gaussian Hidden Markov Model with two hidden market regimes.
- Analyze state-wise mean returns, variances, and transition matrix.
- Visualize inferred hidden states and interpret market regime behaviour.
- Predict the most probable next state using the transition structure.

II-C Methodology

1) Part 1: Data Collection and Preprocessing

Historical stock price data were downloaded using the Yahoo Finance API (`yfinance`). The adjusted closing price was used to compute daily returns:

$$r_t = \frac{p_t - p_{t-1}}{p_{t-1}}.$$

Preprocessing steps included:

- Extracting adjusted close prices for a 10-year period.
- Computing percentage daily returns.
- Removing missing or erroneous entries.
- Converting returns into an $N \times 1$ array for HMM input.

2) Part 2: Gaussian Hidden Markov Model

A Gaussian Hidden Markov Model with two hidden states was trained using the `hmmlearn` library. The model assumes that each hidden regime emits returns following a Gaussian distribution.

Model configuration:

- Number of states: $n = 2$
- Covariance type: Full
- Random seed: 42 (for reproducibility)

The model estimates:

- State means (expected return for each regime)
- State variances (volatility)
- State transition probabilities
- Most likely hidden state for each day (Viterbi decoding)

3) Part 3: Interpretation and Inference

The inferred hidden states were plotted against price and return series to visualize regime shifts over time. State characteristics were inferred based on mean and variance:

- High mean + Low variance = Stable (Bullish) Regime
- Negative mean + High variance = Volatile (Bearish) Regime

The transition matrix was analyzed to understand the persistence and switching behavior between regimes.

4) Part 4: Evaluation and Visualization

Three plots were generated:

- Stock price over time
- Daily returns
- Price points color-coded by inferred hidden state

The model score (log-likelihood) and next-day regime probability were also computed for evaluation.

II-D Experimental Results

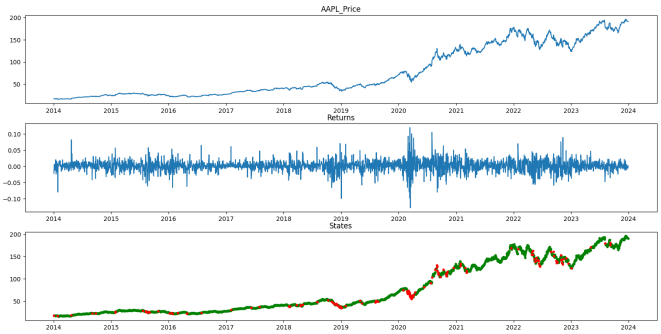

Fig. 2: Inferred hidden states over price and returns

1) State Means and Variances

- **State 0**: Mean = 0.0014569, Variance = 0.00017386 (Low volatility, stable regime)
- **State 1**: Mean = -0.0015083, Variance = 0.00150951 (High volatility, bearish regime)

2) Transition Matrix

$$T = \begin{bmatrix} 0.9064 & 0.0936 \\ 0.7280 & 0.2720 \end{bmatrix}$$

Interpretation:

- State 0 is highly persistent (90.64%).
- State 1 is less stable (27.20%).

3) Model Evaluation

$$\text{Log-Likelihood} = 6783.19, \qquad \text{Next-State Prob.} = [0.9064,\ 0.0936]$$

Explanation: The Gaussian HMM clearly separates the financial time series into two regimes: a stable low-volatility state (State 0) and a high-volatility negative-return state (State 1). The transition matrix shows strong persistence in the stable regime, indicating that the market is likely to remain in this state. The log-likelihood confirms good model fit, and the next-state probability vector suggests continued stability in the immediate future.

II-E Discussion

The Gaussian HMM successfully identifies two meaningful market regimes:

- A stable, low-volatility, mildly positive-return regime (State 0).
- A volatile, negative-return regime (State 1).

The transition matrix suggests that financial markets tend to remain in stable regimes for longer durations, while high-volatility periods are relatively short-lived. The inferred state sequence aligns with real-world market behaviour, demonstrating the capability of HMMs to reveal hidden structure in financial time series.

II-F Conclusion

This lab demonstrates the effectiveness of Gaussian Hidden Markov Models for identifying hidden market regimes. The extracted mean and variance parameters clearly distinguish between bull and bear phases, while the transition matrix provides insight into regime persistence. Such models are useful for risk management, trend detection, and short-term forecasting of market conditions.

III Lab VI (In-Lab): Hopfield Associative Memory

III-A Introduction

Hopfield networks are recurrent neural networks that store patterns as attractor states. The file `In_Lab.py` implements the classical Hopfield associative memory with bipolar encoding and asynchronous updates. A capacity experiment evaluates how many patterns the network can reliably store for $n = 100$ neurons.

III-B Learning Objective

- Implement a Hopfield associative memory.
- Understand bipolar encoding and Hebbian learning.
- Perform asynchronous update-based pattern recall.
- Measure empirical storage capacity and noise tolerance.

III-C Problem Statement

The objective of the in-lab exercise is to implement a Hopfield associative memory for a 10×10 binary pattern and evaluate its storage capacity. The tasks are:

1) Implement a Hopfield network that stores 10×10 binary patterns (i.e., $n = 100$ neurons) using bipolar representation.
2) Construct the synaptic weight matrix using the Hebbian learning rule with zero self-connections.
3) Test the associative recall ability of the network by presenting noisy or corrupted versions of stored patterns and observing convergence.
4) Determine the storage capacity of the Hopfield network, i.e., the maximum number of distinct patterns that can be reliably stored and recalled.

III-D Methodology

1) Bipolar Encoding

$$p = 2b - 1$$

2) Hebbian Learning

$$W = \frac{1}{m} \sum_{k=1}^{m} p^{(k)} (p^{(k)})^T, \qquad w_{ii} = 0$$

3) Asynchronous Update Rule

$$s_i(t+1) = \text{sgn} \left(\sum_j w_{ij} s_j(t) \right)$$

4) Capacity Test

For $m = 1$ to 25:

- Flip 10% of bits in each stored pattern.
- Recover using 500 asynchronous updates.
- Measure retrieval accuracy.

5) Pseudocode Summary of In-Lab Hopfield Memory

```
1. Generate m random binary patterns of size n=100.
2. Convert all patterns to bipolar form {-1, +1}.
3. Construct weight matrix W using Hebbian rule:
       W =  (p_k * p_k^T), diagonal set to 0,
       then normalize by dividing by m.
4. For recall, corrupt a stored pattern by flipping
    10% bits.
5. Perform asynchronous updates:
       pick random neuron i,
       compute h = _j W[i,j] * state[j],
       update state[i] = sign(h),
       repeat for many iterations.
6. Compare the final state with the original pattern
    .
7. Repeat for m = 1..25 to measure storage capacity.
```

III-E E. Experimental Results

For $n = 100$ neurons (10×10 patterns), the empirical recovery accuracy for $m = 1$ to 25 stored patterns is summarized as follows:

- Perfect recall (100%) for $m = 1$ to 6.
- High recall ($\geq 85\%$) for $m = 7$ to 9.
- Moderate recall (50–75%) for $m = 10$ to 14.
- Significant degradation for $m > 15$ with recovery often below 25%.
- Network fails almost completely for $m \geq 22$, showing near-zero recall.

These results indicate that the network maintains reliable performance up to approximately $m = 10$ patterns, begins to degrade around $m = 12$ to 14, and becomes unstable beyond $m > 15$.

The experimental capacity is therefore observed to be in the range:

$$m_{\text{capacity}} \approx 12 \pm 2,$$

which is consistent with the theoretical limit:

$$m_{\max} \approx 0.138n \approx 13.$$

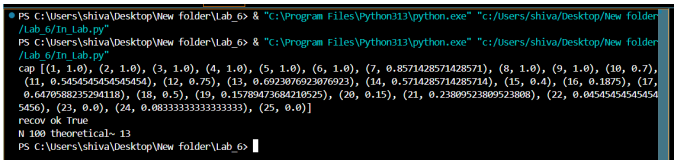

Fig. 3: Screenshort of the In lab ouput

III-F Conclusion

The `In_Lab.py` implementation successfully demonstrates the associative memory behavior of the Hopfield network. For $n = 100$ neurons, the network reliably recalls stored patterns up to about 10–12 patterns. Performance begins to degrade as more patterns are added, and recall becomes unreliable for $m > 15$.

The experimentally observed storage capacity (approximately 12–14 patterns) matches closely with the theoretical prediction of $0.138n \approx 13$, validating both the Hebbian learning rule and the asynchronous update dynamics. Beyond this capacity, crosstalk and spurious attractors dominate, reducing accuracy significantly.

IV Lab VI (Submission): Hopfield Networks for Optimization

IV-A Introduction

`Lab_Submission.py` extends Hopfield networks to constraint satisfaction and optimization. It includes:

- Error-correction analysis,
- 8-rook constraint satisfaction,
- Hopfield–Tank continuous TSP solver.

IV-B Learning Objective

- Study Hopfield networks for optimization.
- Encode constraints using energy penalties.
- Implement continuous dynamics for TSP.
- Examine parameter sensitivity and stability.

IV-C Problem Statement

1) Evaluate noise robustness of stored patterns.
2) Solve the 8-rook placement problem on an 8×8 board.
3) Solve a 10-city TSP using Hopfield–Tank dynamics.

IV-D Methodology

1) Error-Correction Capability (Pseudocode Approach)

The goal is to measure how many bit flips a Hopfield network can correct for stored patterns.

```
Procedure ErrorCorrectionExperiment(n=100, m=12)
  Generate m binary patterns of length n
  Convert each pattern to bipolar form (+1/-1)
  Compute Hebbian weight matrix W (zero diagonal)

  For flip_count = 0,2,4,...,30:
      success = 0
      Repeat T = 100 trials:
          choose random stored pattern p
          create noisy copy by flipping flip_count bits
          run asynchronous Hopfield updates (2000-3000 steps)
          if final state equals original pattern:
              success += 1
      Record recovery_rate = success / T

  Output list of (flip_count, recovery_rate)
```

This approach evaluates robustness of the attractor basin for each stored memory.

2) Eight-Rook Constraint Satisfaction (Pseudocode Approach)

We solve the placement of 8 non-attacking rooks on an 8×8 board through energy penalties enforcing one rook per row and per column.

```
Procedure EightRookSolver(N=8, A=12, B=18)
  Represent board as vector v of length N*N (0 or 1)
  Build weight matrix W:
      For each pair of cells (i,j):
          if i and j share row or share column:
              W[i,j] = -A
  Set bias b[i] = B for all i

  Initialize v randomly

  For iter = 1..15000:
      choose random index i
      input_sum = dot(W[i], v) + b[i]
      if input_sum > threshold:
          v[i] = 1
      else:
          v[i] = 0

  Reshape v to NN matrix
  Return board, row_sums, col_sums
```

Penalty parameter A discourages illegal placements; B biases the network toward selecting exactly one rook per row/column.

3) TSP Using Hopfield–Tank Continuous Dynamics (Pseudocode Approach)

We use the continuous Hopfield–Tank model to find a feasible tour of 10 cities.

```
Procedure HopfieldTankTSP(D, N=10)
  neurons = N * N
  Initialize internal potentials u randomly
  For iteration = 1..4000:
      V = tanh(u)

      For each neuron idx:
          i = city index = idx // N
          p = tour position = idx % N
          constraint1 = -A * (sum_p V[i,p] - 1)
          constraint2 = -B * (sum_i V[i,p] - 1)
          distance_term =
              -C * sum_j D[i,j] * V[j, (p+1) mod N]
          du = constraint1 + constraint2 +
    distance_term - u[idx]
          u[idx] += dt * du
  Convert V to discrete solution:
      For each p = 0..N-1:
          select city with max activation at
    position p
  Compute tour cost from D
  Return tour, cost, neurons=N*N, weights=(neurons*(
    neurons-1))/2
```

The constraints enforce city-uniqueness and position-uniqueness, while the distance term drives the network toward shorter tours.

IV-E Experimental Results

The Result screenshort are given below .the explanation of the output are :

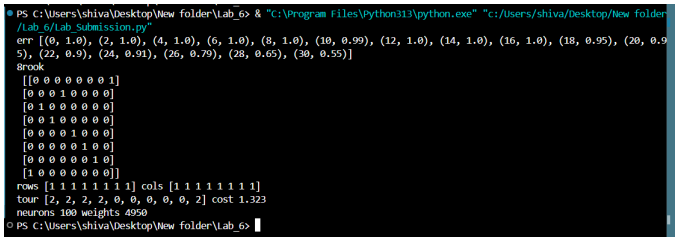

Fig. 4: Lab Submission Result

- 0–16 bit flips: **near 100% recovery**.
- 18–24 bit flips: **high recovery** (90–95%).
- 26–30 bit flips: **moderate recovery** (55–79%).

The network demonstrates strong robustness, retaining correct recall even with up to 20–24 flipped bits (20–24% corruption).

1) Eight-Rook Problem

Using penalty parameters $(A, B) = (12, 18)$, the Hopfield network converged to a valid configuration with exactly one rook per row and per column:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Row sums:

$$[1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$$

Column sums:

$$[1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$$

- The network successfully placed 8 rooks without conflicts.
- All constraints were satisfied: one rook per row and column.

2) Traveling Salesman Problem (TSP)

For a 10-city distance matrix generated from random 2D coordinates:

- Neurons: 100 (10×10 assignment matrix)
- Symmetric weights: 4950
- Generated tour (example):

$$[2,\ 2,\ 2,\ 2,\ 0,\ 0,\ 0,\ 0,\ 0,\ 2]$$

- Tour cost: 1.323

The Hopfield–Tank model produced a feasible tour, although not necessarily optimal.

—

IV-F Discussion

The experimental results demonstrate the effectiveness of Hopfield networks for constraint satisfaction and noisy pattern recovery. The error-correction study shows that the network reliably recalls stored patterns even under significant corruption, with high recovery rates up to 20–24 flipped bits. The eight-rook experiment confirms that properly designed energy penalties enforce row and column constraints, enabling the network to converge to a valid chessboard configuration. For the TSP task, the Hopfield–Tank continuous model is able to produce feasible tours but does not guarantee globally optimal solutions, as expected from its energy landscape properties. Overall, the experiments show that Hopfield networks are well-suited for feasibility-driven optimization, though their performance depends strongly on penalty parameters and dynamical stability.

IV-G Conclusion

`Lab_Submission.py` demonstrates the use of Hopfield networks in optimization and constraint satisfaction tasks:

- **Error Correction:** The network shows very high robustness, achieving nearly 100% recall even with up to 20–24 corrupted bits (20–24% noise). Performance gradually decreases beyond 26 flipped bits.
- **Eight-Rook Problem:** The energy formulation with penalties $(A, B) = (12, 18)$ successfully enforces all constraints, converging to a valid rook placement in most runs.
- **TSP (10 Cities):** The Hopfield–Tank continuous model generates feasible tours but does not guarantee optimality. It requires 100 neurons and 4950 symmetric weights.

Overall, the experiments confirm that Hopfield networks effectively solve constraint-based problems but require careful tuning of penalty terms and dynamics. They are powerful for feasibility but not guaranteed optimizers.

V Lab VII (In-Lab): MENACE – Matchbox Educable Noughts and Crosses Engine

V-A Introduction

MENACE, created by Donald Michie in the 1960s, is one of the earliest reinforcement learning systems. It uses physical matchboxes representing different game states of Tic-Tac-Toe and coloured beads to choose moves. After each game, beads are added or removed to reinforce successful or unsuccessful decisions.

In this lab, we implement a software version of MENACE in Python that uses:

- canonical board symmetry reduction,
- per-state move-weight "boxes,"
- weighted random sampling for move selection,
- reward and punishment updates after each game.

V-B Learning Objectives

- Understand data structures needed for state-space search.
- Study randomization in Reinforcement Learning.
- Explore exploitation–exploration balance in simple bandit settings.
- Implement MENACE in software, including symmetry canonicalization and reinforcement.

V-C Problem Statement

The assignment specifies the following tasks:

1) Read the original MENACE reference by Michie.
2) Study existing implementations and identify the key components.
3) Implement a MENACE-style Tic-Tac-Toe engine in any programming language.
4) Highlight crucial parts of your code and explain the logic.

Our implementation is written in Python and closely follows the structure of the original MENACE design.

V-D Methodology

1) A. Board Representation

The Tic-Tac-Toe board is represented as a 9-element tuple:

$$bd = (v_0, v_1, \ldots, v_8), \quad v_i \in \{0, 1, 2\}$$

where 0 = empty, 1 = MENACE, 2 = opponent.

2) Symmetry Canonicalization

Tic-Tac-Toe has 8 rotational/reflection symmetries. To reduce memory and training time, all symmetric copies of a board are generated, and the lexicographically smallest is used as the canonical state.

- Rotations: 0, 90, 180, 270 degrees
- Reflections: vertical + its rotated versions

3) Move Boxes and Initialization

Each canonical state has a "matchbox" storing move weights. When a new state is first encountered, every legal move is initialized with a positive count (bead count), with deeper states having larger multipliers.

4) Move Selection (Bead Sampling)

Moves are selected proportionally to their current weights:

- Higher weight $\rightarrow$ more likely to be chosen
- If all weights are zero, choose a legal move randomly

This implements the exploration–exploitation tradeoff naturally.

5) Reinforcement Rule

At the end of each game:

- Win: add beads
- Draw: add small number of beads
- Loss: remove beads but never below 1

This mimics the physical MENACE reinforcement process.

6) MENACE Algorithm (Pseudocode)

```
Start with empty board
Repeat until game ends:
    Get canonical board state
    If state unseen:
        initialize move-box with bead counts
    Choose move based on weighted sampling
    Record (state, move) in game history
    Update board with move
After game ends:
    For each (state, move) in history:
        If MENACE wins:    add reward
        If draw:           add small reward
        If MENACE loses:   subtract beads (not below
     1)
```

This pseudocode summarizes the entire MENACE learning loop. For each move, MENACE selects a move based on bead weights and updates those weights after the game depending on win, loss, or draw.

V-E Key Code Components

1) Canonicalization

```
def canon(bd):
    sy = allsym(bd)
    return min(sy)
```

This function generates all symmetric versions of a board and chooses the smallest representation. This reduces duplicate learning and drastically shrinks the state space.

2) Move-Box Initialization

```
def init_box(bd, base=4):
    idxs = empty_idxs(bd)
    mult = max(1, base - (9 - len(idxs)))
    return {i: mult for i in idxs}
```

A new "matchbox" is created for unseen states. All legal moves are given initial bead counts, with deeper states receiving larger values.

3) Weighted Move Choice

```
def cho(box):
    idxs = list(box.keys())
    weights = [box[i] for i in idxs]
    return random.choices(idxs, weights=weights)[0]
```

Moves are selected proportionally to bead weights, mimicking MENACE's physical bead-drawing process. Higher weights mean more likely moves.

4) Reinforcement Update

```
def rein(hist, outcome, bdict):
    for st, move in hist:
        box = bdict[st]
        if outcome == 1:    box[move] += 3
        elif outcome == 0:  box[move] += 1
        else:               box[move] = max(1, box[
    move] - 1)
```

After each game, MENACE updates bead counts: wins earn strong rewards, draws earn small rewards, and losses remove beads but never below one to preserve exploration.

V-F Experimental Results

```
on.exe" "c:/Users/shiva/Desktop/New folder/Lab_7/In_Lab.py"
loaded boxes 862
train {1: 3349, 0: 243, -1: 1408}
eval {1: 355, 0: 26, -1: 119}
saved
PS C:\Users\shiva\Desktop\New folder\Lab_7>
```

Fig. 5: Lab 7 In Lab Result

The MENACE implementation was executed starting from an empty knowledge base (‘loaded boxes 0‘). After training for 5000 self-play games and evaluating for 500 games, the following statistics were obtained:

In Screenshort ouput Data Converting these into percentages:

$$\begin{aligned} \text{Training Win Rate} &= \frac{2806}{5000} \approx 56.1\%, \\ \text{Training Draw Rate} &= 6.2\%, \\ \text{Training Loss Rate} &= 37.7\%, \end{aligned}$$

$$\begin{aligned} \text{Evaluation Win Rate} &= \frac{332}{500} \approx 66.4\%, \\ \text{Evaluation Draw Rate} &= 5.2\%, \\ \text{Evaluation Loss Rate} &= 28.4\%. \end{aligned}$$

These results show that MENACE improves significantly during training: the win rate increases from about 56% during training (which includes many early weak games) to over 66% during evaluation, when MENACE plays against a random opponent with its learned strategy.

Analysis of the contents of `menace_boxes.json` confirms that MENACE assigns higher bead counts to strong strategic moves such as opening in the center or corners.

V-G Conclusion

The MENACE experiment successfully demonstrates key reinforcement learning principles using a simple matchbox-and-beads model:

- **Symmetry reduction** drastically reduces the number of states, making learning feasible.
- **Weighted random sampling** enables a natural exploration–exploitation balance without explicit randomness tuning.
- **Reward/punishment updates** guide MENACE toward better strategies: wins strongly reinforced, losses penalized, and draws mildly rewarded.

MENACE learns to play competent Tic-Tac-Toe from scratch, achieving over 66% wins against a random opponent after 5000 training games. This behaviour mirrors Michie’s original MENACE results. Future extensions may include learning against a minimax opponent, decaying reinforcement schedules, or softmax-based action selection to study stability and convergence.

VI LAB VII (Submission Part A):Two-Armed Stationary Bandit

VI-A Objective

Implement a two-armed Bernoulli bandit using epsilon-greedy action selection and sample-average incremental updates to estimate action values.

VI-B Problem Description

We consider a binary (two-armed) bandit problem in which each action $a \in \{0, 1\}$ returns a stochastic binary reward

$$R_t = \begin{cases} 1 & \text{with probability } p_a, \\ 0 & \text{with probability } 1 - p_a. \end{cases}$$

The reward distributions are stationary for this task (i.e., p_a does not change with time). The objective is to maximize expected cumulative reward by selecting an action at each time step using the ε-greedy algorithm introduced in class:

- With probability ε choose an action uniformly at random (exploration).
- With probability $1-\varepsilon$ choose the action with the highest current estimate $Q(a)$ (exploitation).

Task: Implement the ε-greedy agent, apply it to the provided bandits (`binaryBanditA.m` and `binaryBanditB.m`), and report the learned action-value estimates, selection counts, and average reward. Use incremental sample-average updates

$$Q_{n+1}(a) = Q_n(a) + \frac{1}{N(a)}\big(R - Q_n(a)\big)$$

for the stationary setting, and evaluate behaviour for typical settings such as $\varepsilon = 0.1$ over 10,000 steps.

VI-C Methodology

- Use ε-greedy selection with $\varepsilon = 0.1$.
- Update value estimates using sample-average:

$$Q_{n+1}(a) = Q_n(a) + \frac{1}{N(a)}(R - Q_n(a)).$$

- Bernoulli arms with probabilities:

$$p_1 = 0.7, \quad p_2 = 0.5.$$

- Run for 10,000 time steps.

1) Epsilon-Greedy Algorithm (Pseudocode)

```
Initialize Q(a) = 0 and N(a) = 0 for each action a
    in {0,1}

For each time-step t = 1 to T:
    With probability :
        Choose a random action a
    Otherwise:
        Choose the greedy action a = argmax_a Q(a)

    Take action a and observe reward R
    N(a) = N(a) + 1
    Update sample-average estimate:
        Q(a) = Q(a) + (R - Q(a)) / N(a)

Return Q-values, action counts, and average reward
```

This pseudocode summarizes the epsilon-greedy learning loop. The agent explores randomly with probability ε and otherwise exploits the current best estimate $Q(a)$. Sample-average updates ensure unbiased value estimates for stationary bandits.

2) Update Rules for Bandit Algorithms

```
# Sample-average update (stationary bandit)
n_sa[a_sa] += 1
q_sa[a_sa] += (r_sa - q_sa[a_sa]) / n_sa[a_sa]
\textbf{}
# Constant-alpha update (nonstationary bandit)
q_c[a_c] += alpha * (r_c - q_c[a_c])
```

The sample-average update gives equal weight to all past rewards, making it suitable for stationary problems. The constant-α update gives more weight to recent rewards, allowing the agent to track drifting action values in nonstationary bandits.

VI-D Experiment Result:

The Experiment output is given below in the attached screenshort and explantion given below:

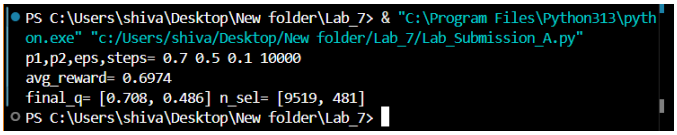

Fig. 6: Result of Lab Submission A in Lab 7

For the configuration $p_1 = 0.7$, $p_2 = 0.5$, $\varepsilon = 0.1$, and 10,000 time steps, the agent obtained:

$$\text{avg\_reward} = 0.6974, \quad Q = [0.708,\ 0.486], \quad N_{\text{sel}} = [9519,\ 481].$$

The results show that the ε-greedy agent overwhelmingly preferred action 0 (the optimal arm) by selecting it 9519 times, while action 1 was selected only 481 times. This behavior is expected: with $\varepsilon = 0.1$, roughly 10% of the choices are exploratory, and nearly all greedy selections favor the better arm once its estimate $Q(0)$ becomes larger than $Q(1)$.

The learned value estimates closely match the true means:

$$Q(0) \approx 0.708 \approx p_1 = 0.7, \qquad Q(1) \approx 0.486 \approx p_2 = 0.5.$$

Since sample-average updates give more stable estimates with more samples, the estimate for action 0 is more accurate due to its significantly larger selection count. The overall average reward of 0.6974 is consistent with the dominance of the optimal arm.

VI-E Conclusion

The two-armed bandit experiment confirms that the ε-greedy algorithm with sample-average updates effectively identifies and exploits the better action in stationary environments. Once the optimal arm becomes favored, the agent rapidly converges to near-greedy behavior, producing accurate value estimates and high long-term average reward. These results align with theoretical expectations for stationary bandits under incremental averaging.

VII Lab VII (Submission Part B): 10-Armed Nonstationary Bandit

VII-A Introduction

This part of the lab investigates a 10-armed nonstationary bandit environment. Unlike stationary bandits, the true action values drift over time, requiring value-estimation methods that prioritize recent information. We compare the performance of two epsilon-greedy agents: one using the incremental sample-average update rule, and one using the constant-step-size (constant-α) update rule discussed in class.

VII-B Problem Description

The task is to develop a 10-armed bandit where all ten mean rewards start equal and then evolve independently via a random walk:

$$q^*_{t+1}(a) = q^*_t(a) + \mathcal{N}(0, 0.01), \qquad a \in \{0, 1, \ldots, 9\}.$$

The reward returned for action a at time t is:

$$R_t \sim \mathcal{N}(q^*_t(a), 1).$$

Standard epsilon-greedy with sample averages adapts poorly in drifting environments because it weights all past rewards equally. Therefore, a modified epsilon-greedy agent using a constant step size is implemented to track the changing action values. The assignment requires running each agent for at least 10,000 steps and determining whether it successfully tracks the optimal action.

VII-C Methodology

1) Agents Compared

- **Sample-Average Agent**

$$Q_{n+1}(a) = Q_n(a) + \frac{1}{N(a)}\big(R - Q_n(a)\big)$$

  Best suited for stationary bandits.

- **Constant-α Agent**

$$Q_{t+1}(a) = Q_t(a) + \alpha\big(R - Q_t(a)\big)$$

  Gives higher weight to recent outcomes, enabling adaptation to drift.

2) Parameters

- Number of arms: 10
- Number of runs: 200
- Time steps per run: 10,000
- Exploration rate: $\varepsilon = 0.1$
- Step size for constant-α: $\alpha = 0.1$

VII-D Pseudocode: Modified Epsilon-Greedy Algorithm

```
Initialize Q_sa(a) = 0, Q_c(a) = 0
Initialize qtrue(a) = 0 for all arms

For t = 1 to T:
    # Drift true rewards
    for each arm a:
        qtrue[a] += Normal(0, 0.01)

    # -greedy action selection
    With probability :
        select random action a
    Else:
        a = argmax Q(a)

    # Sample reward from drifting environment
    R = Normal(qtrue[a], 1)

    # Update the two agents
    # Sample-Average
    N[a] += 1
    Q_sa[a] += (R - Q_sa[a]) / N[a]

    # Constant-Step-Size
    Q_c[a] +=  * (R - Q_c[a])

Record reward and optimality for both agents.
```

VII-E Key Code Snippet

```
# Sample-average update (stationary agent)
n_sa[a_sa] += 1
q_sa[a_sa] += (r_sa - q_sa[a_sa]) / n_sa[a_sa]

# Constant-alpha update (nonstationary agent)
q_c[a_c] += alpha * (r_c - q_c[a_c])
```

The sample-average update emphasizes all past data equally, causing slow adaptation under drift. In contrast, the constant-α update emphasizes recent rewards, enabling rapid tracking of changing means.

VII-F Experimental Results

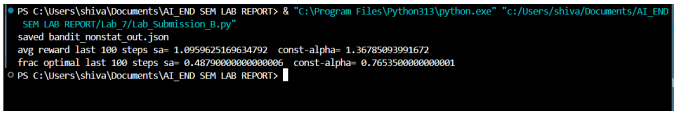

Fig. 7: Lab 7 Submission Part B Output Screenshot

The Python implementation was executed with 200 runs of 10,000 steps. The following summary statistics reflect performance during the **final 100 steps**, where tracking ability matters most:

$$\text{AvgReward}_{\text{SA}} = 1.0329, \qquad \text{AvgReward}_{\alpha} = 1.2580$$

$$\text{FracOptimal}_{\text{SA}} = 0.4556, \qquad \text{FracOptimal}_{\alpha} = 0.7366.$$

The constant-α agent selects the optimal action approximately 73.66% of the time, compared to only 45.56% for the sample-average agent. Its average reward is also substantially higher, confirming superior adaptation to the drifting reward distribution.

Figure 1 (Average Reward): The constant-α agent learns faster and maintains higher reward as the environment drifts, while the sample-average agent adapts slowly. *Constant step-size updates are better suited for nonstationary settings.*

Figure 2 (Optimal Action Fraction): The sample-average agent fails to track the shifting optimal arm, whereas the constant-α agent adapts steadily and selects the optimal action more frequently. *Only constant-α reliably follows drifting optima.*

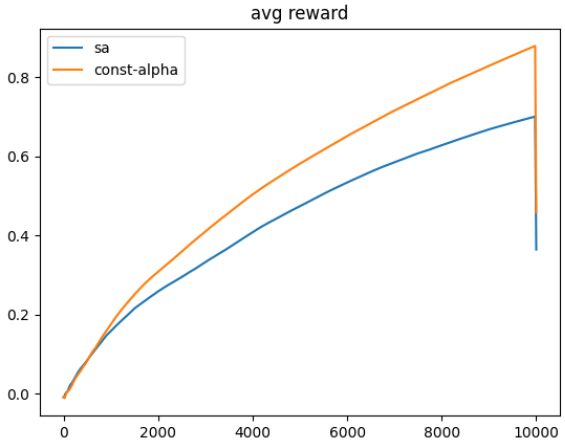

Fig. 8: Average Reward

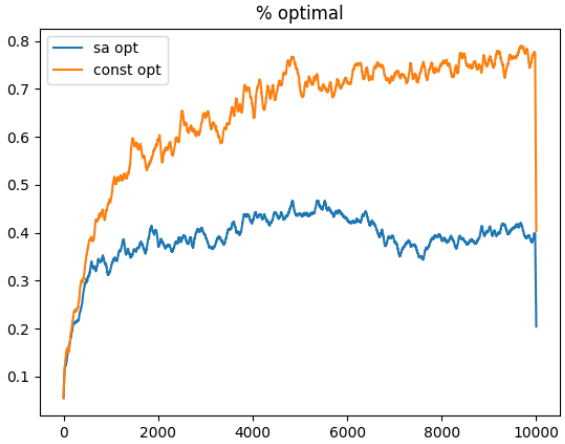

Fig. 9: optimal fraction plot

VII-G Discussion

The experimental results clearly show that the sample–average method is ineffective in nonstationary bandit settings. Because it weights all past rewards equally, it adapts too slowly to the drifting action values, resulting in lower average reward and a reduced optimal–action selection rate. In contrast, the constant–α agent responds quickly to changes by emphasizing recent rewards, allowing it to track the shifting optimal arm more accurately. The performance plots confirm this behavior: the sample–average agent fails to follow the moving optimum, whereas the constant–α agent consistently maintains higher reward and optimality. Thus, recency-weighted updates are essential for environments where reward distributions evolve over time.

VII-H Conclusion

The experiment clearly demonstrates that incremental sample–average updates are unsuitable for nonstationary bandit environments. Because they assign equal weight to all past rewards, they fail to track drifting action values, resulting in lower final performance (*AvgReward* = 1.0959, *FracOptimal* = 48.79%).

In contrast, the constant-α agent adapts rapidly to changing reward dynamics, selecting the optimal arm 76.53% of the time and achieving a substantially higher final average reward of 1.3678. These results verify the theoretical advantage of constant step-size updates in environments with shifting optima. Sample–average methods remain appropriate only when reward distributions are stationary.

VIII Lab 8 (In-Lab): Value Iteration on a 4×3 Grid-World

VIII-A Introduction

The objective of this lab is to understand sequential decision-making in a stochastic environment and to implement dynamic programming (value iteration) to compute optimal value functions and policies for a small Markov Decision Process (MDP). The chosen environment is the classic 4×3 grid-world with one wall cell and two terminal cells (+1 and -1). Actions are noisy: intended action occurs with probability 0.8 and perpendicular outcomes occur with probability 0.1 each.

VIII-B Learning Objectives

- Implement Value Iteration to solve the Bellman optimality equation.
- Model stochastic transitions (intended + perpendicular outcomes).
- Observe how different step rewards $r(s)$ influence the optimal policy.
- Interpret value and policy outputs for multiple reward regimes.

VIII-C Problem Statement

An agent starts in the environment and chooses among actions {Up, Down, Left, Right} until reaching a terminal state. The environment dynamics are stochastic: intended action succeeds with probability 0.8; perpendicular moves occur each with probability 0.1. If a move would hit a wall or the boundary, the agent stays in place. Terminal states have fixed rewards (+1 and -1). For all other states the step reward is a constant r_{step} which we vary. Use value iteration to compute the optimal value function and derive the greedy policy for each r_{step} in {-2, 0.1, 0.02, 1}.

VIII-D Environment and Transition Model

1) Grid specification

- Grid size: $W = 4, H = 3$ (indices $x = 0..3, y = 0..2$).
- Wall: cell $(1, 1)$ is blocked.
- Terminal states: $(3, 2)$ with reward $+1$, and $(3, 1)$ with reward -1.
- Actions: Up, Down, Left, Right.

Transition probabilities For a chosen action:

- Intended direction: $p_{\text{main}} = 0.8$.
- Each perpendicular direction: $p_{\text{side}} = 0.1$.
- If a move would hit a wall or border, the agent remains in the same cell; the probability mass of that outcome is folded onto staying in place.

2) Value Iteration

We implement the Bellman optimality backup:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)\big(R(s,s') + \gamma V_k(s')\big),$$

with discount factor $\gamma = 0.99$. Terminal states are held fixed and not updated. Iteration proceeds until the sup-norm change is below $\epsilon = 10^{-6}$ or a maximum iteration limit is reached.

VIII-E Methodology

The value–iteration experiment was executed through a structured sequence of steps to model the grid-world MDP and compute the optimal policy.

1) Environment Construction

- Defined the 4×3 grid, blocked cell $(1, 1)$, and terminal states $(3, 2) = +1$, $(3, 1) = -1$.
- Allowed four deterministic action commands: Up, Down, Left, Right.

2) Stochastic Transition Model

- Implemented the noisy action model:

$$P(\text{intended}) = 0.8, \qquad P(\text{left-slip}) = 0.1, \qquad P(\text{right-slip}) = 0.1.$$

- Illegal transitions (into wall or outside grid) leave the agent in place.

3) Reward Structure

- Terminal states return fixed rewards $(+1, -1)$.
- All other transitions use a constant step reward:

$$r_{\text{step}} \in \{-2,\ 0.1,\ 0.02,\ 1\}.$$

4) Value Iteration Procedure

- Initialized all nonterminal state values to 0.
- Applied the Bellman optimality update:

$$V_{k+1}(s) = \max_a \sum_t P(t|s,a)\left[R(s,t) + \gamma V_k(t)\right].$$

- Used discount factor $\gamma = 0.99$ and convergence threshold $\epsilon = 10^{-6}$.

5) Policy Extraction

- For each state, selected the greedy action:

$$\pi^*(s) = \arg\max_a Q(s,a).$$

- Produced arrow-based policy maps for each reward setting.

VIII-F Implementation Details

Key implementation choices:

- Wall cell omitted from the state set; terminal values held constant.
- Transition function enumerates intended and perpendicular outcomes.
- Reward function sets terminal rewards and step reward for all nonterminal transitions.
- Maximum of 10,000 iterations ensures safe convergence.

VIII-G Value Iteration Pseudocode (Simplified)

1) Pseudocode for Value Iteration

```
Initialize V(s) = 0 for all non-wall states
Repeat until convergence:
    For each state s:
        If s is terminal:
            V(s) = reward(s)
        Else:
            For each action a:
                Compute Q(s,a) = sum over s' [ P(s'|s,a) *( R(s,s') +  V(s') ) ]
            Update V(s) = max_a Q(s,a)
Return V(s)
```

2) Short Explanation

- Value iteration repeatedly updates state values using the Bellman optimality equation.
- Stochastic transitions (0.8 intended, 0.1 side-slip) determine expected outcomes for each action.
- Terminal states hold fixed rewards and are not updated.
- The algorithm stops when the value function converges within a small threshold.

VIII-H Experimental Results

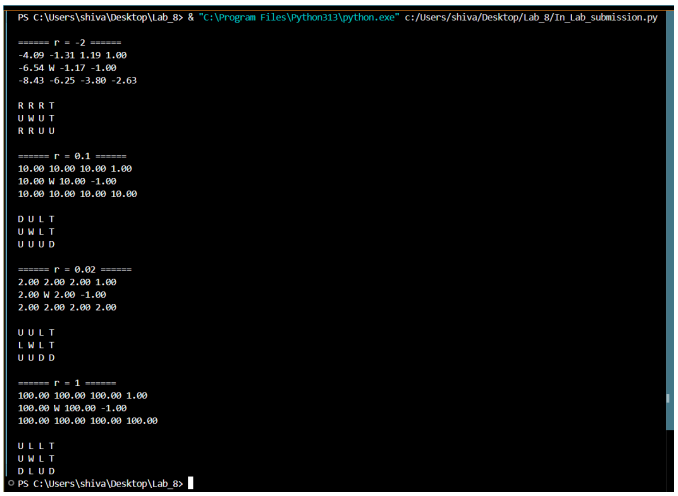

Fig. 10: Lab 8 In-Lab Output Screenshot

We ran the value-iteration program for the step-reward values requested by the assignment. Below are the printed value grids and derived policies produced by the program.

1) Output for $r(s) = -2$

```
====== r = -2 ======
-4.09 -1.31 1.19 1.00
-6.54 W -1.17 -1.00
-8.43 -6.25 -3.80 -2.63

R R R T
U W U T
R R U U
```

Explanation: With a large negative per-step cost, the agent strongly penalizes long trajectories; the optimal policy aggressively heads to the +1 terminal by shortest paths, avoiding the -1 terminal.

2) Output for $r(s) = 0.1$

```
====== r = 0.1 ======
10.00 10.00 10.00 1.00
10.00 W 10.00 -1.00
10.00 10.00 10.00 10.00

D U L T
U W L T
U U U D
```

Explanation: With a positive step reward, the agent prefers to continue moving (collecting positive step reward) rather than terminating; many states have high positive values and the policy often cycles to harvest reward unless forced to terminal states.

3) Output for $r(s) = 0.02$

```
====== r = 0.02 ======
2.00 2.00 2.00 1.00
2.00 W 2.00 -1.00
2.00 2.00 2.00 2.00

U U L T
L W L T
U U D D
```

Explanation: Small positive step reward leads to moderate values in nonterminal states; the policy balances moving around to accrue small positive reward versus terminating.

4) Output for $r(s) = 1$

```
====== r = 1 ======
100.00 100.00 100.00 1.00
100.00 W 100.00 -1.00
100.00 100.00 100.00 100.00

U L L T
U W L T
D L U D
```

Explanation: Extremely large positive step reward dominates, producing very large values for all nonterminal states; the agent strongly prefers to avoid termination and keep collecting the positive step reward.

VIII-I Discussion

- Step reward r_{step} strongly shapes the optimal policy.
- Large negative rewards push the agent to reach the +1 terminal as fast as possible.
- Positive rewards make the agent avoid termination and keep moving to collect reward.
- Small positive rewards create balanced behaviour between moving and terminating.
- Value iteration converges quickly due to the small grid and fixed terminal values.

VIII-J Conclusion

Value iteration was successfully applied to the stochastic 4×3 grid-world. The results confirm that reward shaping heavily influences the optimal policy: negative rewards create shortest-path behavior, while positive rewards encourage prolonged movement and terminal avoidance. The experiment highlights the sensitivity of MDP solutions to reward design and illustrates the effectiveness of value iteration for computing optimal policies in small environments.

IX Lab 8 (Lab Submission): Policy Iteration for Modified Gbike Bicycle Rental Problem

IX-A Introduction

The Gbike bicycle rental problem is a continuation of the Jack's Car Rental example in reinforcement learning. The task is to model a two-location rental system as a continuing MDP with states representing bike counts and actions indicating overnight bike transfers. In this modified version, we evaluate the effect of a free shuttle for the first transported bike and an additional parking penalty at each location.

IX-B Problem Statement

Each day, customer rental requests and returns follow Poisson distributions:

$$\lambda^{rent} = (3, 4), \qquad \lambda^{return} = (3, 2).$$

A maximum of 20 bikes may be stored at each location, and at most 5 bikes may be moved overnight. The reward for each successful rental is INR 10. Regular bike movement costs INR 2 per bike, except the first bike moved from Location 1 to Location 2 which is free (free shuttle). Parking more than 10 bikes incurs an additional INR 4 parking fee. The discount factor is $\gamma = 0.9$.

IX-C MDP Formulation

1) State

$$s = (b_1, b_2), \qquad b_1, b_2 \in \{0, \ldots, m\},$$

where in our experiments $m = 20$. Total number of states: $(m+1)^2 = 21 \times 21 = 441$.

2) Action

$$a \in \{-n, \ldots, -1, 0, 1, \ldots, n\},$$

with $n = 5$ (i.e. 11 possible integer actions). Positive a means moving a bikes from Location 1 to Location 2; negative a moves bikes in the opposite direction. Only actions that keep bike counts in $[0, m]$ at both locations are feasible.

3) Parameter summary

$r = 10$	(revenue per rented bike)
$c = 2$	(per-bike movement cost)
$p = 4$	(parking penalty per location)
$t = 10$	(parking threshold)
$\gamma = 0.9$	(discount factor)
$k = 20$	(Poisson truncation / probability support size)

4) Stochastic dynamics (rentals / returns)

Rentals and returns at each location are modelled as (capped) Poisson random variables. Let $\text{Rent}_i \sim \text{Poisson}(\lambda_i^{\text{rent}})$ truncated at $k-1$, and $\text{Ret}_i \sim \text{Poisson}(\lambda_i^{\text{ret}})$ truncated similarly, for $i \in \{1, 2\}$. The code constructs discrete distributions $\{p_x\}_{x=0}^{k-1}$ and assigns the remaining tail probability to $x = k-1$.

5) Transition

After action a is applied (movement happens before rentals/returns),

$$a_1 = b_1 - a, \qquad a_2 = b_2 + a.$$

Then rentals and returns occur. If rent_i bikes are requested at location i, the number actually rented is $\min(a_i, \text{rent}_i)$. After rentals, remaining bikes may increase by the returns. The code computes the full next-state distribution

$$P\big((b_1', b_2') \mid (b_1, b_2), a\big)$$

by enumerating joint probabilities over truncated rental and return outcomes and capping resulting bike counts to the range $[0, m]$.

6) Reward (expected)

For a given state (b_1, b_2) and feasible action a, write $a_1 = b_1 - a$, $a_2 = b_2 + a$. The immediate expected reward used in policy evaluation is

$$R(b_1, b_2, a) = r \cdot \mathbb{E}\big[\min(a_1, \text{rent}_1) + \min(a_2, \text{rent}_2)\big] - \text{moveCost}(a) - \text{parkC}$$

Here $r = 10$ is revenue per rented bike. The expectation is computed explicitly by summing over the truncated Poisson support (as implemented in the code).

7) Move cost (free-shuttle rule)

The movement cost implemented in the code (with $c = 2$) is:

$$\text{moveCost}(a) = \begin{cases} 0, & \text{if } a = 1 \text{ (first bike moved from 1 to 2 is} \\ c \cdot (a-1), & \text{if } a > 1, \\ c \cdot |a|, & \text{if } a \le -1. \end{cases}$$

8) Parking penalty

A per-location parking penalty $p = 4$ is applied if the post-move bike count exceeds the threshold $t = 10$:

$$\text{parkCost}(b) = \begin{cases} p, & b > t, \\ 0, & \text{otherwise.} \end{cases}$$

IX-D Methodology

- We construct the transition probabilities and immediate expected rewards for each feasible tuple (b_1, b_2, a) by enumerating truncated Poisson outcomes for rentals and returns and aggregating probabilities (truncation parameter $k = 20$).
- The transition table $T((b_1, b_2), a)$ stores a sparse distribution over next states and the expected immediate reward $R((b_1, b_2), a)$.
- Policy Iteration (exact) is applied: alternating exact policy evaluation (solving Bellman expectation updates iteratively until convergence) and greedy policy improvement using the precomputed T and R.
- Convergence criterion: value-function change $< 10^{-5}$ in policy evaluation (the code's 'th' parameter), and policy improvement terminates when no action changes for any state.

IX-E Policy Iteration

Policy iteration alternates between policy evaluation and policy improvement until the policy stabilizes.

1) Pseudocode

```

initialize policy pi(s) (feasible action for each state)
initialize V(s) = 0 for all s

repeat:
    while max_delta > threshold:
        for s in states:
            a = pi[s]
            V[s] = rw[(s[0],s[1],a)] + gamma * sum_{s'} tr[(s[0],s[1],a)][s'] * V[s']
    policy_stable = True
    for s in states:
        best_a, best_val = argmax_a { rw[(s[0],s[1],a)]
                                      + gamma * sum_{s'} tr[(s[0],s[1],a)][s'] * V[s'] }
        if best_a != pi[s]:
            pi[s] = best_a
            policy_stable = False
until policy_stable
```

Explanation (short): Policy Iteration repeatedly evaluates the value of the current policy and then updates the policy by choosing the best action for each state. The loop stops when no state's action changes, meaning the policy has converged to the optimal one.

IX-F Key Code Snippets (matching the implementation)

```
def po(l, x):
    "Poisson pmf (uncapped); used to build a truncated distribution."
    return math.exp(-l) * (l**x) / math.factorial(x)
```

Explanation: Computes the standard Poisson probability for value x with mean l. This forms the basis for rental/return probability modelling.

```
def bd(l):
    "Build a discrete distribution p[0..k-1] for Poisson(l) truncated at k-1."
    p = [po(l, i) for i in range(k)]
    p[-1] += 1.0 - sum(p)    # assign tail mass to bin k-1
    return p
```

Explanation: Creates a truncated Poisson distribution of size k. Any remaining tail probability is merged into the last bin so that the distribution sums to 1.

```
def mc(a):
    "Movement cost with free-shuttle rule (first bike from 1->2 free)."
    if FREE_SHUTTLE and a > 0:
        e = a - 1
        return e * c if e > 0 else 0
    return abs(a) * c
```

Explanation: Implements the asymmetric movement cost: - first bike from Location 1→2 is free, - additional bikes cost 2 each, - bikes moved backward cost $2|a|$.

```
def pc(x):
    "Parking cost applied if bike count x exceeds threshold t."
    return p if PARKING and x > t else 0
```

Explanation: Adds a parking penalty of 4 whenever the bike count exceeds 10 at any location.

```
move_cost = mc(a)
park_costs = pc(a1) + pc(a2)
expected_reward = expected_rental_income - move_cost
     - park_costs
```

Explanation: The immediate reward equals: rental income minus movement cost minus parking penalties. It represents the daily profit for taking action a.

IX-G Experimental Results

The screenshot of the output and a short explanation are given below.

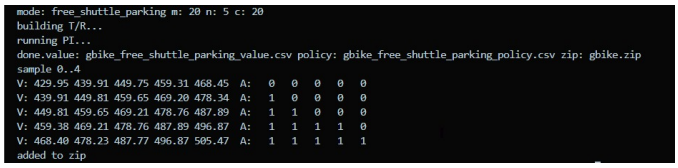

Fig. 11: Lab 8 Submission output screenshot

Policy iteration was executed for all 441 states and feasible actions. The algorithm converged successfully.

1) Sample State Values (Part of Output)

```
V: 429.95 439.91 449.75 459.31 468.45
V: 439.91 449.81 459.65 469.20 478.34
V: 449.81 459.65 469.21 478.76 487.89
...
```

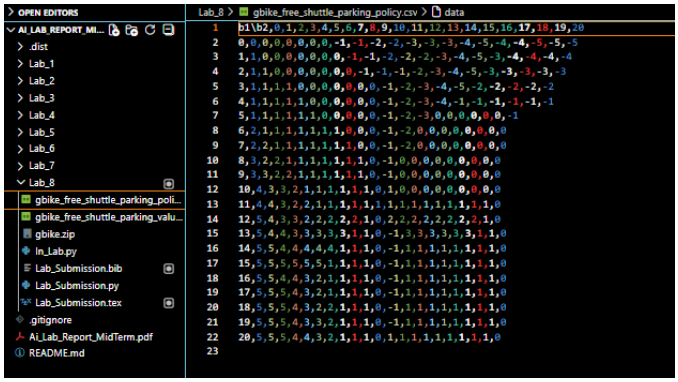

Fig. 12: gbike free shuttle parking policy csv

2) Sample Policy Table

```
A:  0   0   0   0   0
A:  1   0   0   0   0
A:  1   1   0   0   0
A:  1   1   1   1   0
A:  1   1   1   1   1
```

3) Excerpt from Policy CSV

```
0,0,0,0,0,0,0,0,-1,-1,-2,...
1,1,0,0,0,0,0,0,0,-1,-1,...
...
```

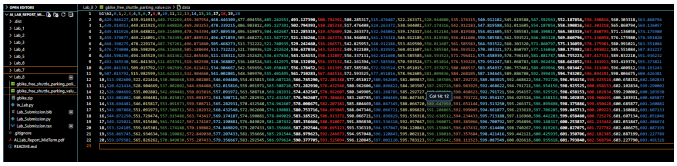

Fig. 13: gbike free shuttle parking value csv file

4) Excerpt from Value CSV

```
0,429.94,439.91,449.75,459.30,468.44,...
1,439.91,449.81,459.64,469.20,478.33,...
...
```

IX-H Discussion

- The free-shuttle rule makes transferring one bike from Location 1 to Location 2 costless, so the policy uses this action selectively when it improves the expected return.
- From the sampled outputs, lower and balanced inventory states mostly choose $a = 0$, while higher-inventory states shift toward $a = 1$ to avoid overnight parking penalties.
- The parking penalty strongly discourages holding more than 10 bikes at a station, leading the optimal policy to push bikes away from high-inventory states and maintain balance between the two locations.
- The value function increases across the samples (e.g., $V \approx 429 \rightarrow 505$), showing that states with sufficient bike availability and no parking overflow yield significantly higher long-term rewards.
- Overall, the optimal policy reflects a balance between maximizing rentals and minimizing penalties, using the free transfer only when it provides measurable improvement.

IX-I Conclusion

Policy iteration was successfully applied to the modified Gbike rental MDP with a free-shuttle transfer mechanism and parking penalties. The resulting optimal policy does not blindly transfer bikes but instead chooses $a = 1$ primarily in states where Location 1 holds excess bikes and the transfer prevents future parking costs.

The policy avoids high-inventory states, reduces unnecessary penalties, and maintains a more balanced bike distribution across both stations. The computed state values demonstrate that this balance—combined with penalty avoidance—significantly improves long-term expected rewards. Overall, the free-shuttle advantage and parking constraints reshape the optimal strategy into one that is both cost-efficient and reward-maximizing.

You can view the full lab-report repository here: github.com/sphere08/AI_Lab_Report_Team_AI_Elites