ARTIFICIAL INTELLIGENCE LAB REPORT

Group Name: Acode

Tejas Pakhale-202211061, Rajat Kumar Thakur-202211070, Tanay Patel-202211094, Abhi Tundiya-202211095

***Abstract*—The report covers the implementation of Bayesian Networks to model dependencies between student grades and predict internship qualifications. The study uses naive Bayes classification to analyze the data, testing classifier accuracy under different assumptions about feature dependencies. The report contrasts scenarios where features are independent versus dependent, demonstrating the utility of probabilistic models in handling real-world data. Furthermore, the report delves into Gaussian Hidden Markov Models (HMMs) for financial market prediction. By analyzing stock returns, HMMs are used to identify hidden market regimes (e.g., high and low volatility), offering insights into market dynamics and risk levels. Through these diverse AI methods, the report demonstrates the practical application of reinforcement learning, probabilistic models, and dynamic systems in solving real-world decision-making challenges.**

the report examines the use of Hopfield Networks for solving combinatorial problems, including implementing a 10x10 associative memory and analyzing the network's capacity for storing distinct patterns. The error-correcting capability of the Hopfield network is also explored, along with its application to solve the Eight-rook problem and the Traveling Salesman Problem (TSP), demonstrating its potential in optimization tasks. This report applies the epsilon-greedy algorithm to multi-armed bandit problems. It first explores a binary bandit with stochastic rewards, then develops a 10-armed bandit with non-stationary rewards. A modified epsilon-greedy agent is used to address the changing rewards, and its performance is evaluated over 10,000 steps. The report highlights the challenges of using epsilon-greedy in dynamic environments.

I. Introduction

This report explores various AI techniques, including Bayesian Networks for modeling dependencies in student grades and predicting internship qualifications, Gaussian Hidden Markov Models (HMMs) for analyzing financial market volatility, Hopfield Networks for solving combinatorial optimization problems like the Traveling Salesman Problem (TSP), and the epsilon-greedy algorithm for addressing challenges in multi-armed bandit problems with non-stationary rewards. Additionally, it applies Markov Decision Processes (MDPs) and dynamic programming for optimal sequential decision-making in stochastic environments, and models the Gbike bicycle rental problem as a finite MDP to optimize bike movement strategies under practical constraints.

Week 5: Graphical Models and Bayesian Networks Inference and Classification in R
Understand the graphical models for inference under uncertainty, build Bayesian Network in R, Learn the structure and CPTs from Data, naive Bayes classification with dependency between features.

Challenge Problem : Extend the Gaussian HMM to include more than two hidden states to capture more complex market behavior, such as extreme volatility periods or stable growth phases. Compare the results of the HMM model on different financial instruments (e.g., comparing stocks of different companies or indices) to understand how market dynamics vary across assets.

Week 6: Hopefield Network To understand the working of Hopfield network and use it for solving some interesting combinatorial problems.

Week 7: N arm Bandit
Basics of data structure needed for state-space search tasks and use of random numbers required for MDP and RL, Understanding Exploitation - Exploration in simple n-arm bandit reinforcement learning task, epsilon-greedy algorithm

Week 8: Markov Decision Process and Dynamic Programming
Understand the process of sequential decision making (stochastic environment) and the connection with reinforcement learning.

II. Week 5 :Introduction

In this lab, we use graphical models and Bayesian inference to handle uncertainty in educational data. We build Bayesian Networks in R to model relationships between student grades and internship qualification. The lab focuses on using the naive Bayes classifier to predict outcomes, both assuming grade independence and considering dependencies. The goal is to train and evaluate the classifier, introducing probabilistic models for data analysis and classification.

III. Fundamentals

A. *Bayesian Inference and Graphical Models*

Graphical models use diagrams to represent relationships between variables, helping us understand uncertainty. Bayesian inference allows us to update our beliefs about unknowns as new information becomes available. Together, these methods aid in making informed decisions from complex data. Before diving into practical work, it's essential to grasp concepts like conditional probability, Bayes' theorem, and how graphical models depict variable relationships, which are crucial for managing uncertainty.

B. *Constructing Bayesian Networks in R*

Bayesian Networks are graphical models that represent relationships between variables using a directed acyclic graph (DAG). In R, the bnlearn package simplifies the process of building, visualizing, and analyzing these networks.

Learners will understand how to create the structure of Bayesian Networks, set up Conditional Probability Tables (CPTs), and carry out tasks like asking probabilistic questions and learning from data.

C. *Learning Dependencies from Data*

Bayesian Networks can be constructed using expert knowledge or learned from data. Structure and parameter learning techniques, such as constraint-based, score-based, and hybrid methods, help identify relationships between variables and build Conditional Probability Tables (CPTs) from observed data.

D. *Naive Bayes Classification*

Naive Bayes is a simple probabilistic model that uses Bayes' theorem and assumes feature independence. It's widely used in tasks like document classification and spam filtering.

Understanding: Students will learn how Naive Bayes calculates class probabilities using Bayes' theorem and assumes feature independence.

Bayes Theorem:

$$P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$

Naive Bayes Classification Formula:

$$P(C|X) \propto P(C) \cdot P(x_1|C) \cdot P(x_2|C) \cdot \ldots \cdot P(x_n|C)$$

Where:

- $P(C|X)$ is the **posterior probability**: the probability of class C given the features X.
- $P(X|C)$ is the **likelihood**: the probability of the features X given the class C.
- $P(C)$ is the **prior probability** of the class C.
- $P(X)$ is the **marginal likelihood** or evidence: the total probability of the features X.

E. *Classifier Implementation and Evaluation*

Setting up a Naive Bayes classifier involves training on labeled data to calculate class priors and feature probabilities. The model predicts outcomes for new data, with performance evaluated using accuracy, precision, recall, and F1 score. Learners will gain practical experience in R for data preparation, model training, prediction, and evaluation in educational data analysis.

IV. Problem Statement

A. *Problem I*

V. Problem Statement

The dataset *2020_bn_nb_data.txt* includes student grades from various courses. The objective is to model the relationships between these courses and learn the Conditional Probability Tables (CPTs). Furthermore, the aim is to predict a student's grade in PH100 based on their grades in other courses.

We'll train a Naive Bayes classifier using 70% of the data to predict internship qualification based on grades, assuming course independence. We'll evaluate accuracy on the remaining 30% over 20 runs, then repeat the experiment considering relationships between course grades.

VI. Implementation Method

1) **Data Preprocessing**:
   - Load the dataset (*2020_bn_nb_data.txt*) into R.
   - Analyze the data to gain insights into its structure and content.
   - Clean the data as needed by addressing any missing values or outliers.
2) **Bayesian Network Construction**:
   - Use the `bnlearn` package to construct a Bayesian Network.
   - Establish the network structure using domain knowledge or apply structure learning algorithms to derive it from the data.
   - Estimate Conditional Probability Tables (CPTs) for each node in the network.
3) **Grade Prediction in PH100**:
   - Given a student's grades in other courses, use the Bayesian Network to predict the grade in PH100.
   - Use the known grades to query the network and determine the most probable grade in PH100.
4) **Naive Bayes Classifier**:
   - Split the dataset into training and testing sets (70% training, 30% testing).
   - Train a naive Bayes classifier using the training data, assuming independence between course grades.
   - Evaluate the classifier's performance on the testing data using accuracy and other relevant metrics.
5) **Assessing Classifier Accuracy**:
   - Repeat the training and testing of the Naive Bayes classifier 20 times using randomly selected data.
   - Document the accuracy of the classifier during each iteration.
6) **Considering Dependencies**:
   - Adjust the Naive Bayes classifier to account for possible relationships between course grades.
   - Redo the training and testing process, evaluating the accuracy of the classifier based on this revised approach.

VII. Solution

VIII. Conclusion

In this lab, we used Bayesian Networks to model student grade relationships and created Conditional Probability Tables. We predicted internship eligibility with a Naive Bayes classifier, assuming course independence. Considering dependencies improved accuracy, emphasizing the importance of understanding relationships for better predictions in education.

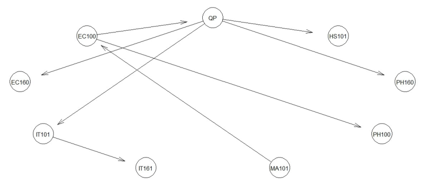
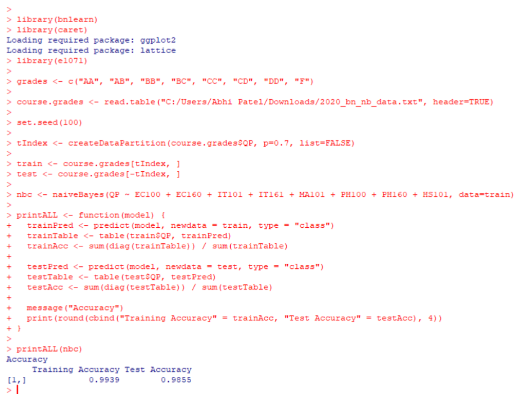


Figure 1: Dependencies



```
>
> library(bnlearn)
> library(caret)
Loading required package: ggplot2
Loading required package: lattice
> library(e1071)
>
> grades <- c("AA", "AB", "BB", "BC", "CC", "CD", "DD", "F")
>
> course.grades <- read.table("C:/Users/Abhi Patel/Downloads/2020_bn_nb_data.txt", header=TRUE)
>
> set.seed(100)
>
> tIndex <- createDataPartition(course.grades$QP, p=0.7, list=FALSE)
>
> train <- course.grades[tIndex, ]
> test <- course.grades[-tIndex, ]
>
> nbc <- naiveBayes(QP ~ EC100 + EC160 + IT101 + IT161 + MA101 + PH100 + PH160 + HS101, data=train)
>
> printALL <- function(model) {
+   trainPred <- predict(model, newdata = train, type = "class")
+   trainTable <- table(train$QP, trainPred)
+   trainAcc <- sum(diag(trainTable)) / sum(trainTable)
+
+   testPred <- predict(model, newdata = test, type = "class")
+   testTable <- table(test$QP, testPred)
+   testAcc <- sum(diag(testTable)) / sum(testTable)
+
+   message("Accuracy")
+   print(round(cbind("Training Accuracy" = trainAcc, "Test Accuracy" = testAcc), 4))
+ }
>
> printALL(nbc)
Accuracy
     Training Accuracy Test Accuracy
[1,]            0.9939        0.9855
> |
```

Figure 2: Final Solution

IX. Problem Statement for Submission

The goal of this assignment is to use a Gaussian Hidden Markov Model (HMM) to identify hidden patterns or "regimes" in a financial time series, like stock returns. Financial markets often go through phases with different levels of risk or volatility, such as bull (rising) and bear (falling) markets. These phases are not directly visible, but the HMM can help uncover them by analyzing observable data like stock price changes.

A. *Data collection*

Historical stock data for Microsoft (MSFT) was downloaded from Yahoo Finance, including columns like Date, Open, High, Low, Close, Adj Close, and Volume.

B. *Data Cleaning*

- Missing rows were removed to ensure a complete dataset.
- Dates were standardized to YYYY-MM-DD format.
- Columns were renamed for clarity.

C. *Feature Engineering*

- Daily Returns: A new column was created to calculate percentage changes in adjusted closing prices between consecutive days:

$$\text{Daily Return} = \frac{\text{Adj Close (Day t)} - \text{Adj Close (Day t-1)}}{\text{Adj Close (Day t-1)}}$$

D. *Data Transformation*

- Daily return was reshaped into a 2D array to fit the HMM model.

E. *Data Transformation*

- Saving Data The cleaned data was saved as CSV for easy reuse in future steps.

F. *Analysis*

In this model, We are capturing 4 hidden states from 0 to 3.

When training the Hidden Markov Model (HMM), the algorithm:

Estimates the mean (μ) and covariance (σ^2) of the Gaussian distribution for each state.

$$\mu_i = \frac{1}{N_i}\sum_{t=1}^{N_i} x_t \quad \text{and} \quad \sigma_i^2 = \frac{1}{N_i}\sum_{t=1}^{N_i}(x_t - \mu_i)^2$$

where μ_i and σ_i^2 represent the mean and variance of the returns for state i, and N_i is the number of data points in state i.

Optimizes the transition probabilities between states to best explain the observed sequence of returns. This is done by estimating the transition matrix A, where the element A_{ij} represents the probability of transitioning from state i to state j:

$$A_{ij} = P(\text{state at time } t+1 = j \mid \text{state at time } t = i)$$

The goal is to find the transition probabilities that maximize the likelihood of observing the given sequence of returns.

Determining State Labels:

The states are identified and labeled after the model is trained based on:

- **Volatility (Standard Deviation of Returns)**: States with low standard deviation (σ^2) are labeled as “Low Volatility.” States with high standard deviation are labeled as “High Volatility.”
- **Mean Return** (μ): States with positive mean return are labeled as “Bullish.” States with negative mean return are labeled as “Bearish.”

G. *Analysis of Microsoft Stocks*

here , we are taking historical data of MICROSOFT(MSFT) of Past 10 years, from date 12/04/2014 to 12/04/2024 using yahoo finance APIs.

Market Breakdown:

Plot of all 4 States for Microsoft:

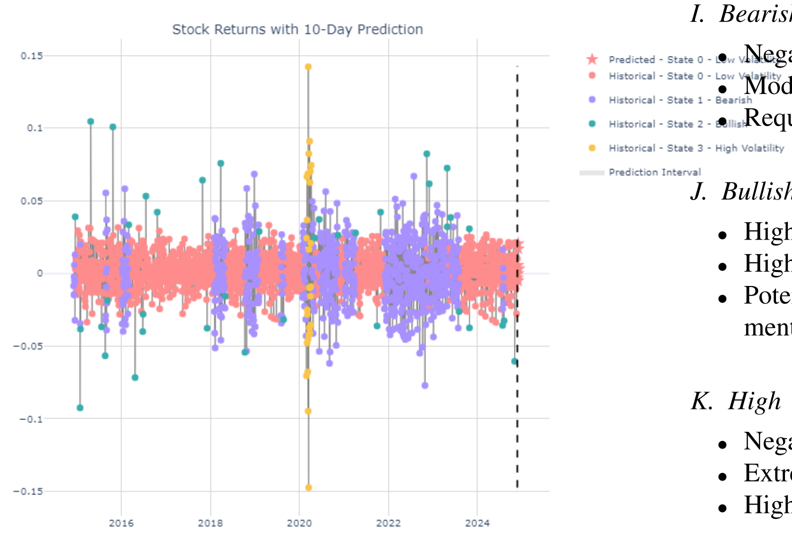


Figure 3: States for 10 years

Classification of States:

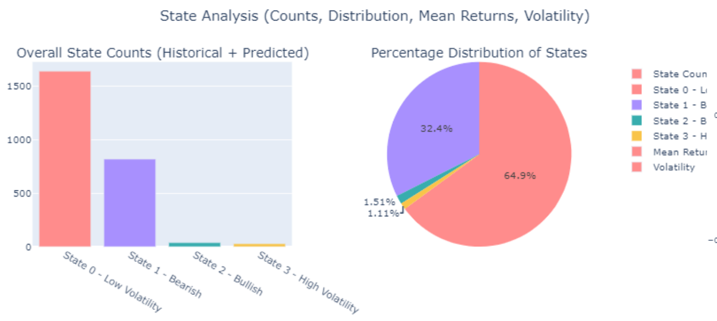


Figure 4: classification

- Low Volatility (State 0): Most common state, 1636 occurrences.
- Bearish Market (State 1): 817 occurrences.
- Bullish Market (State 2): Rare, only 38 occurrences.
- High Volatility (State 3): Least frequent, 28 occurrences

Performance Characteristics:

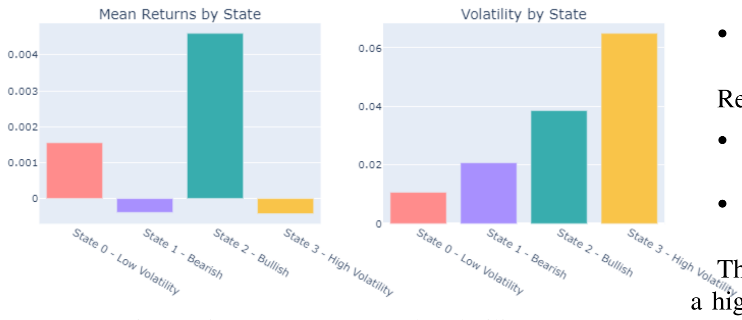


Figure 5: Mean returns and Volatility

H. *Low Volatility State (State 0)*

- Stable environment
- Modest positive return (0.0016)
- Lowest volatility (0.0107)
- Ideal for conservative investment strategies

I. *Bearish Market (State 1)*

- Negative mean return (-0.0004)
- Moderate volatility (0.0208)
- Requires defensive investment approaches

J. *Bullish Market (State 2)*

- Highest mean return (0.0046)
- Highest volatility (0.0386)
- Potential for significant gains, but requires risk management

K. *High Volatility State (State 3)*

- Negative mean return (-0.000421)
- Extreme volatility (0.0649)
- High-risk environment

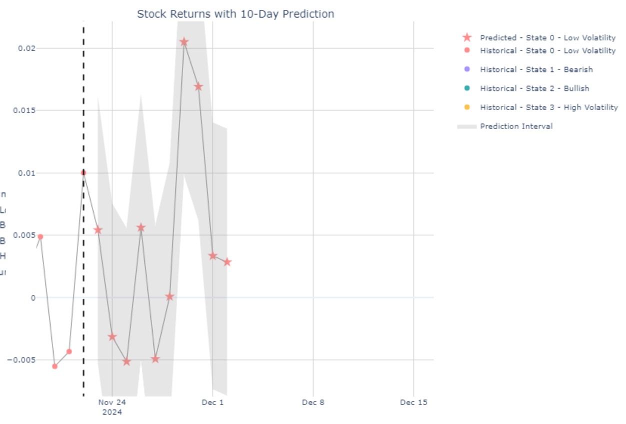


Figure 6: 10 Days prediction

10-Day Stock Returns Predictions

above image shows the predicted 10-day stock returns for different market states. The key points are as follows:

- The predicted state is State 0 - Low Volatility, indicated by the red star.
- The historical data for the different states is shown in the legend.
- The prediction interval is displayed, indicating the expected range of returns.

Regarding the 10-day stock return predictions:

- The predictions show the expected daily returns for the next 10 days, all within the Low Volatility state.
- The returns range from approximately -0.5% to +2%, suggesting a relatively stable market environment.

The results align with the transition matrix, which shows a high probability (0.969) of remaining in the Low Volatility state.

Date	Predicted Return	Market State
2024-11-23	0.0054	State 0 - Low Volatility
2024-11-24	-0.0031	State 0 - Low Volatility
2024-11-25	-0.0051	State 0 - Low Volatility
2024-11-26	0.0056	State 0 - Low Volatility
2024-11-27	-0.0049	State 0 - Low Volatility
2024-11-28	0.0001	State 0 - Low Volatility
2024-11-29	0.0205	State 0 - Low Volatility
2024-11-30	0.0169	State 0 - Low Volatility
2024-12-01	0.0034	State 0 - Low Volatility
2024-12-02	0.0028	State 0 - Low Volatility

How it is being calculated:

1. Starting State

The prediction begins with the most recent hidden state, which is derived from the trained Hidden Markov Model (HMM).

2. Transitioning Between States

Using the transition matrix T, the model simulates the future states for the next 10 days. At each step, the current state is used to select the next state based on the probabilities from the transition matrix:

$$P(\text{next state} \mid \text{current state}) = T_{\text{current state,next state}}$$

The next state is chosen by sampling from this probability distribution.

3. Predicted Returns

For each predicted state, the future returns are drawn from the Gaussian distribution associated with that state:

$$r_t \sim \mathcal{N}(\mu_s, \sigma_s^2)$$

where μ_s and σ_s^2 are the mean and variance of the returns in state s, which were learned during the training of the HMM.

4. Future Predictions

This process is repeated for 10 steps (days) to generate a sequence of predicted returns and states.

5. Confidence Intervals

The confidence intervals for the predicted returns are calculated using the standard deviation σ_s of the returns for each predicted state:

$$\text{Upper Bound} = r_t + \sigma_s$$

$$\text{Lower Bound} = r_t - \sigma_s$$

These intervals provide an estimate of the range within which the future returns are expected to lie, with a certain level of confidence.

State Transition Probabilities Matrix

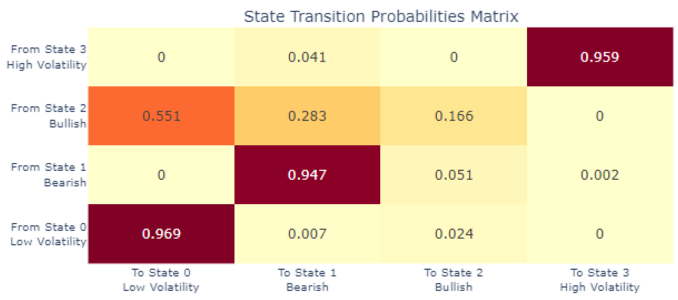


Figure 7: Transition Matrix for States

The above image displays the State Transition Probabilities Matrix. This matrix represents the likelihood of transitioning between different market states. The key features are:

- The rows represent the current state, while the columns represent the next state.
- The values in the cells indicate the probability of transitioning from the row state to the column state.
- For example, the probability of transitioning from State 2 (Bullish) to State 2(bearish) is 0.283.

This transition matrix provides insights into the expected market dynamics and the likelihood of moving between different volatility regimes.

Conclusion

In summary, the data and visualizations suggest a market environment that is expected to remain in a low volatility regime over the next 10 days, with modest positive returns. The transition matrix further reinforces the stability of this state and the low likelihood of transitioning to more volatile market conditions during this period.

X. Bonus Problem:

in 1st Section of Bonus , We already discussed 4 States in above analysis, for 2nd Section We will Analyze Historical price of Cisco.

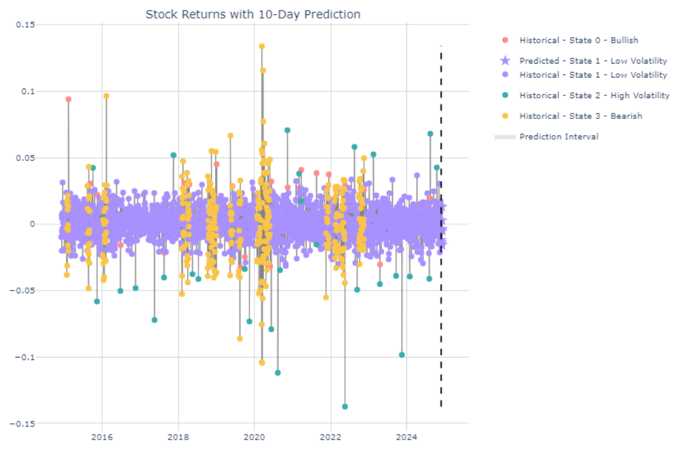


Figure 8: States for 10 years

Perform Analysis:

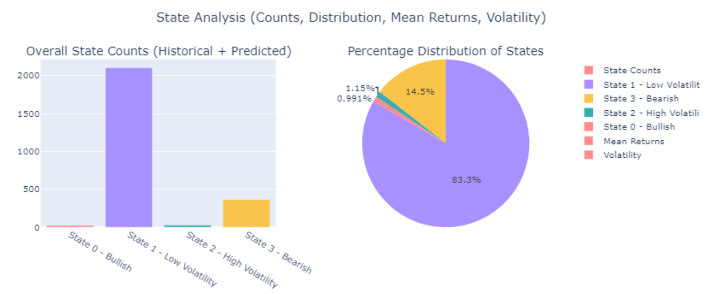


Figure 9: classification

State Breakdown:

- **Bullish (State 0)**: 25 occurrences
- **Low Volatility (State 1)**: 2102 occurrences
- **High Volatility (State 2)**: 29 occurrences
- **Bearish (State 3)**: 366 occurrences

Performance Characteristics:

Bullish (State 0)::

- **Mean Return**: 0.0086
- **Volatility**: 0.0301

Low Volatility (State 1)::

- **Mean Return**: 0.0010
- **Volatility**: 0.0102

High Volatility (State 2)::

- **Mean Return**: -0.0095
- **Volatility**: 0.04677

Bearish (State 3)::

- **Mean Return**: -0.00121
- **Volatility**: 0.02826

Key Insights:

- The market is primarily in the **Low Volatility** state (2102 occurrences).
- **Bullish** and **High Volatility** states are relatively rare.
- The **Bearish** state is more common than both the **Bullish** and **High Volatility** states.
- The **Low Volatility** state offers modest positive returns with low risk.
- The **Bullish** state provides the highest returns but also carries higher risk.
- **High Volatility** and **Bearish** states are associated with negative returns.

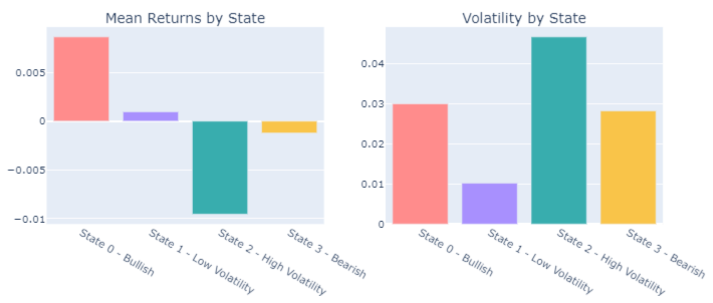


Figure 10: Transition Matrix

Transition Matrix for Cisco

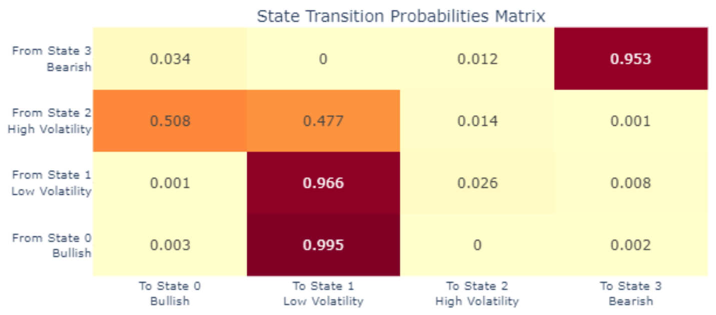


Figure 11: Transition Matrix

Head to Head Comparison:

For Conservative Investors:

- **Cisco** is the safer option, with more time spent in the **Low Volatility** state (2102 occurrences vs. 1636 for Microsoft).
- Cisco's **Low Volatility** state has a higher mean return (0.0010 vs. 0.0016 for Microsoft) and lower volatility (0.0102 vs. 0.0107).

For Long-Term Investors:

- **Microsoft's Bearish** state is more common (817 occurrences vs. 366 for Cisco), which could pose more challenges for long-term investors.
- However, **Microsoft's Low Volatility** state still offers modest positive returns (mean return of 0.0016) with relatively low risk (volatility of 0.0107).

For Aggressive, Short-Term Investors:

- **Cisco's Bullish** state provides a higher mean return (0.0086 vs. 0.0046 for Microsoft), but also higher volatility (0.0301 vs. 0.0386).
- Opportunities in **Cisco's Bullish** state may be more lucrative but come with increased risk.

Conclusion:

Overall, **Cisco** appears the safer, more stable option, especially for conservative and long-term investors. **Microsoft** has more pronounced **Bearish** periods but also higher potential returns in the **Bullish** state for aggressive investors willing to take on more risk.

XI. Week 6: Hopfield Theory and Network

THe Hopfield network, introduced by John Hopfield in 1982, is a type of recurrent artificial neural network designed for associative memory and optimization tasks. It operates by storing patterns in a high-dimensional space and recalling them through an energy minimization process. The network consists of binary neurons that interact with each other through symmetric weights.

Hopfield networks are particularly useful in solving problems where an optimal solution needs to be found among a set of possible solutions, such as combinatorial optimization problems, pattern recognition, and constraint satisfaction. The network converges to a stable state, representing the solution, by minimizing an energy function. This makes it applicable to various NP-hard problems like the traveling salesman problem (TSP), N-Queens, and more.

A. *Problem Definitions*

1) **10x10 Binary Associative Memory:**
    - Implement a 10x10 binary associative memory using the Hopfield network.
    - Evaluate the network's capacity to store and retrieve distinct patterns, and its error-correcting ability.
2) **Error-Correcting Capability:**
    - Analyze the network's ability to correct corrupted or noisy patterns.
3) **Eight-Rook Problem:**
    - Solve the Eight-Rook problem on an 8x8 chessboard using the Hopfield network.
    - Define an energy function and discuss the choice of weights.
4) **Traveling Salesman Problem (TSP):**
    - Solve the TSP for 10 cities using the Hopfield network.
    - Derive the energy function and calculate the number of weights required.

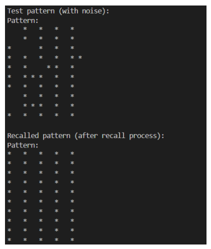
Understanding Terminologies

Hopfield Architecture

- **Neurons:** Binary units representing network nodes, with states of 1 (active) or 0 (inactive). Neurons update based on inputs from other neurons.
- **Weights:** Symmetric connections ($w_{ij} = w_{ji}$) that determine interaction strength between neurons. Proper weights guide the network to stable states.
- **Capacity:** The network can store approximately $0.15 \times N$ distinct patterns, where N is the number of neurons.

Energy Function and Landscape

- **Energy Function:** A scalar value associated with the network's state, minimized during updates. Lower energy corresponds to more stable states.
- **Energy Landscape:** A representation of all possible states and their energy levels. Stable states are local minima, with the global minimum representing the optimal solution.



```
Test pattern (with noise):
Pattern:
  * * * *
  * * * *
*   * * *
* * * * **
* *  ** *
* *** * *
* * * * *
  * * * *
  *** * *
* * * * *

Recalled pattern (after recall process):
Pattern:
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Figure 12: Working

Related:

- **Convergence:** The network evolves to minimize energy, ensuring convergence to a stable state.
- **Associative Memory:** Enables pattern storage and recall, even with noisy or partial inputs.
- **Error Correction:** Corrects small errors in corrupted inputs by moving toward the closest stored pattern.

Energy Function in Optimization Problems

- For problems like the TSP and Eight-Rook Problem, the energy function incorporates constraints such as visiting each city once or avoiding conflicts, while optimizing the solution.

XII. Implementing Addressable Memory

Algorithm 1 Training Algorithm for Hopfield Network

```
Require: Patterns $\{p_k\}$, number of neurons $N$
Ensure: Weight matrix $W$
1: Initialize $W \leftarrow \mathbf{0}_{N\times N}$
2: for each pattern $p_k$ do
3:     $W \leftarrow W + p_k \otimes p_k$          ▷ Outer product update
4: end for
5: for $i = 1$ to $N$ do
6:     $W_{ii} \leftarrow 0$          ▷ No self-connections
7: end for
8: return $W$
```

Algorithm 2 Recall Algorithm for Hopfield Network

```
Require: Initial pattern $p$, weight matrix $W$, number of steps $T$
Ensure: Final recalled pattern $p$
1: for $t = 1$ to $T$ do          ▷ Iterate for $T$ steps
2:     for $i = 1$ to $N$ do
3:         activation$_i \leftarrow \sum_{j=1}^{N} W_{ij} \cdot p_j$          ▷ Compute activation
4:         $p_i \leftarrow \begin{cases} +1 & \text{if activation}_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$
5:     end for
6: end for
7: return $p$
```

Statistical Physics Approach (Storkey Heuristic):

- Capacity is approximately $0.138 \times N$

- For $N = 100$ neurons, estimated capacity is approximately 14 distinct patterns

Classic Gardner Bound:

- Capacity is approximately $0.15 \times N$
- For $N = 100$ neurons, capacity is approximately 15 distinct patterns

Computational Learning Theory Estimate:

- Capacity is roughly between $0.12 \times N$ and $0.14 \times N$
- For $N = 100$ neurons, capacity is around 12-14 patterns

The error-correcting capability of the network depends on the Hamming distance between stored patterns, d_H, and can be expressed as:

$$d_H < \frac{N}{2} \tag{1}$$

where d_H is the maximum Hamming distance the network can tolerate for error correction. For $N = 100$, the maximum Hamming distance is:

$$d_H < \frac{100}{2} = 50$$

(2)

it can correct up to 49 errors (fliped bits)

8 Rook Problem

The Eight-Rook problem involves placing eight rooks on an 8x8 chessboard such that no two rooks threaten each other. A rook threatens another if it is in the same row or column. This is a classic combinatorial problem that can be solved by encoding the constraints into the network's energy function.

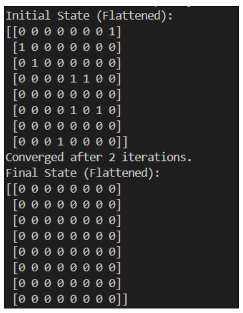
Energy Function

The energy function used in the Hopfield network ensures that the system converges to a valid configuration where the constraints of the Eight-Rook problem are satisfied. The energy function is defined as:

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

where:

- $\mathbf{s}$ is the state vector representing the chessboard (flattened into a single array).
- W is the weight matrix encoding the constraints that penalize two rooks being in the same row or column.



```
Initial State (Flattened):
[[0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 0 0 0 1 1 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 1 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]]
Converged after 2 iterations.
Final State (Flattened):
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
```

Figure 13: 8 Rook Problem

Algorithm 3 Hopfield Network for Solving the 8 Rooks Problem

```
Require: Chessboard size $N$, number of iterations $num\_iterations$
Ensure: Final configuration of rooks on the chessboard
 1: Initialize the board with random rook placements: $p_0 \in \{0,1\}^{N^2}$    ▷ Random initial state
 2: Initialize weight matrix $W$ based on row and column constraints    ▷ No two rooks in the same row/column
 3: for $t = 1$ to $num\_iterations$ do    ▷ Iterate for $num\_iterations$ steps
 4:     for $i = 1$ to $N^2$ do    ▷ For each neuron (position) on the chessboard
 5:         activation$_i \leftarrow \sum_{j=1}^{N^2} W_{ij} \cdot p_j$    ▷ Compute the activation for the $i$-th position
 6:         $p_i \leftarrow \begin{cases} +1 & \text{if activation}_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$    ▷ Update state based on activation
 7:     end for
 8:     if $p = p_{\text{new}}$ then    ▷ Check for convergence
 9:         Return $p$    ▷ Solution found, return the final configuration
10:         Break    ▷ Exit the loop if the network has converged
11:     end if
12: end for
13: return $p$    ▷ Return the final state after all iterations
```

Reasoning for Weight Initialization in the 8 Rooks Problem

The weight matrix for the Hopfield network is initialized to encode the constraints of the 8 Rooks problem, where no two rooks can share the same row or column.

Weight Initialization

The weight matrix W is set as follows:

$$W_{ij,kl} = \begin{cases} -1 & \text{if } i = k \text{ and } j \neq l \text{ (same row, diff. column)} \\ -1 & \text{if } j = l \text{ and } i \neq k \text{ (same column, diff. row)} \\ 0 & \text{otherwise} \end{cases}$$

A. Reasoning Behind Negative Weights

The negative weights enforce the constraint that no two rooks can occupy the same row or column. When two neurons (representing positions on the chessboard) are in the same row or column, their negative weights cause them to "repel" each other, discouraging configurations where multiple rooks are placed in the same row or column.

Zero Weights on Diagonal

Diagonal weights are set to zero, as there is no need for a neuron to interact with itself. This ensures that the network's dynamics are only influenced by interactions between different positions.

Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a classic optimization problem where a salesman must visit a set of cities exactly once, starting and ending at the same city, and minimize the total travel distance. Given its NP-hard nature, the TSP has important applications in logistics, route planning, and many other optimization problems.

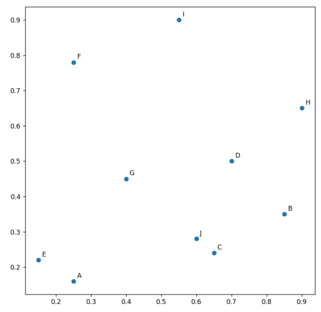


Figure 14: 10 cities

B. Energy Function for TSP in Hopfield Network

The energy function used in the provided code for solving the TSP within a Hopfield network is defined as:

$$E = \frac{A}{2}\sum_x \sum_{i\neq j} \mathbf{y}_{x,i}\cdot \mathbf{y}_{x,j} + \frac{B}{2}\sum_i \sum_{x\neq y} \mathbf{y}_{x,i}\cdot \mathbf{y}_{y,i}$$

$$+\frac{C}{2}\left(\sum_x \sum_i \mathbf{y}_{x,i} - (\text{number\_of\_objects} + \sigma)\right)^2$$

$$+\frac{D}{2}\sum_{x\neq y}\sum_i \mathbf{y}_{x,i}\cdot \mathbf{y}_{y,i+1}$$

Where:

- $\mathbf{y}_{x,i}$ represents the binary state of the system indicating whether city i is visited at position x.
- A, B, C, D are constants controlling the weights for row/-column inhibition and path constraints.
- σ is a constant related to the number of objects.

This energy function models the objective of minimizing the total distance traveled while adhering to the constraints of visiting each city exactly once and returning to the origin.

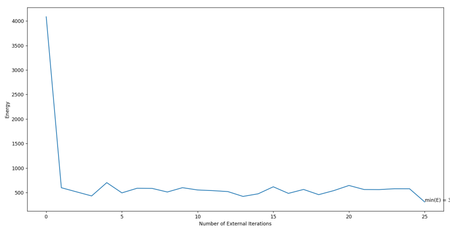


Figure 15: Energy-Iteration Plot

Algorithm for Solving the TSP

Algorithm 4 TSP Solution using Modified Hopfield Network

```
Require: Cities' coordinates, distance matrix $D$, constants $A$, $B$, $C$, $D$, $\sigma$, number of iterations
Ensure: Optimal path and total distance
 1: Initialize Hopfield network with $number\_of\_cities$, $A$, $B$, $C$, $D$, $\sigma$
 2: Initialize distances matrix $D$ between cities
 3: Define energy function incorporating path constraints and city locations
 4: for each iteration do
 5:     Compute input potentials for each city and update output potentials
 6:     Update output potentials based on the energy function
 7:     Compute energy and check for convergence
 8:     if Energy has stabilized or reached a minimum then
 9:         Stop training
10:     end if
11: end for
12: Extract the optimal path from the output potentials
13: Calculate the total distance based on the optimal path
14: return Optimal path and total distance
```

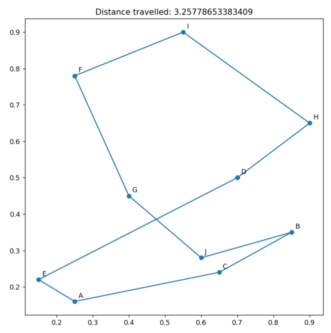


Figure 16: Enter Caption

XIII. Calculation of Weights in Hopfield Network for TSP

The total number of weights in the Hopfield network is calculated as follows:

$$\text{Inhibition Weights in the Same Row} = \frac{n^2(n-1)}{2} = 900$$

$$\text{Inhibition Weights in the Same Column} = \frac{n^2(n-1)}{2} = 900$$

$$\text{Path Constraint Weights} = n\times(2n-2) = 1620$$

$$\text{Total Weights} = \frac{n^2(n-1)}{2} + \frac{n^2(n-1)}{2} + n\times(2n-2) = 3420$$

WEEK 7 : IMPLEMENTING MENACE AND EPSILON-GREEDY ALGORITHMS FOR STATIONARY AND NON-STATIONARY BANDIT PROBLEMS

I. IMPLEMENTING MENACE FOR TIC-TAC-TOE

1. State Space (S)

The state space represents all possible configurations of the Tic-Tac-Toe board. A board state is represented as a string of nine characters, where each character can be:

- X: Represents a move made by the AI (MENACE).
- O: Represents a move made by the human player.
- : Represents an empty space on the board.

Let $s \in S$ be a state defined as:

$$s = [s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9]$$

where each $s_i \in \{X, O, \}$ for $i = 1, 2, \ldots, 9$.

The total number of unique board configurations can be computed as:

$$|S| \leq 3^9 = 19,683$$

2. Action Space (A)

The action space defines the possible moves that can be made from a given state. Each action corresponds to placing a marker in one of the empty spaces on the board. Let $a \in A$ be an action defined as:

$$a = \{i | s_i = \text{ for } i = 1, 2, \ldots, 9\}$$

where a is the set of indices representing valid moves (the empty spaces).

The size of the action space $A(s)$ from state s can be defined as:

$$|A(s)| = \sum_{i=1}^{9} I(s_i =)$$

where I is an indicator function that returns 1 if s_i is empty and 0 otherwise.

3. Transition Function (T)

The transition function defines the result of taking an action a in state s, leading to a new state s':

$$s' = T(s, a)$$

The new state s' is formed by replacing the empty space at index a in s with the current player's marker.

4. Reward Function (R)

The reward function assigns a numerical value based on the outcome of the game:

- $R(s, a) = +1$ if the player wins after the move.
- $R(s, a) = 0$ if the game is drawn.
- $R(s, a) = -1$ if the player loses after the move.

II. BINARY BANDIT PROBLEM

PROBLEM 2: BINARY BANDIT PROBLEM WITH EPSILON-GREEDY ALGORITHM

State Space

The state space for the binary bandit problem can be defined as follows:

$$S = \{s_1, s_2\}$$

where:

- s_1: represents the state when the first action is selected.
- s_2: represents the state when the second action is selected.

Action Space

The action space consists of two actions:

$$A = \{a_1, a_2\}$$

where:

- a_1: represents selecting the first action.
- a_2: represents selecting the second action.

Epsilon-Greedy Algorithm

The **epsilon-greedy algorithm** is a strategy used in reinforcement learning to balance exploration and exploitation:

- With probability $1 - \epsilon$ (in this case, 0.9), the algorithm chooses the action with the highest estimated reward:

$$A = \arg\max_{a \in A} Q(a)$$

- With probability ϵ (in this case, 0.1), the algorithm randomly selects one of the available actions:

$$A = \text{random selection from } A$$

- The algorithm updates the Q-values based on the received reward R:

$$Q(A) = Q(A) + \frac{R - Q(A)}{N(A)}$$

where $N(A)$ is the count of times action A has been selected.

Problem 3: Non-Stationary Bandit Problem with Epsilon-Greedy Algorithm

State Space

The state space for the non-stationary bandit problem can also be defined similarly, considering there are 10 actions (one for each bandit):

$$S = \{s_1, s_2, \ldots, s_{10}\}$$

where:

- Each state s_i corresponds to the selection of action a_i.

Action Space

The action space consists of 10 actions:

$$A = \{a_1, a_2, \ldots, a_{10}\}$$

Non-Stationary Epsilon-Greedy Algorithm

The non-stationary epsilon-greedy algorithm adapts to the changing reward distributions. The main differences are:

- After each action selection, the rewards are updated through an independent random walk:

$$R(a) \leftarrow R(a) + \text{random noise}$$

Problem 3 (Updated): Non-Stationary Bandit Problem with Epsilon-Greedy and Learning Rate

Learning Rate (α)

The new addition in this version of the code is the introduction of a learning rate α, which is used to control how much the Q-value is updated at each step. The learning rate is set to:

$$\alpha = 0.7$$

Modified Q-Value Update Rule

Instead of using the standard incremental update rule based on the number of times an action has been selected, the Q-values are now updated using a constant learning rate:

$$Q(A) = Q(A) + \alpha \cdot (R - Q(A))$$

This allows the agent to update the Q-values more quickly, giving more weight to recent rewards, which is especially useful in a non-stationary environment where the reward distribution may change over time.

Bandit Environment

The bandit environment remains non-stationary, with the reward probabilities changing at each step. The reward update function `bandit_nonstat` is unchanged from the previous version, except now the Q-values update with the learning rate α.

Performance Evaluation

The running average of the rewards is calculated as before, and the plot visualizes the average reward over 10,000 steps:

- The plot illustrates the performance of the agent using epsilon-greedy action selection and learning rate-based Q-value updates.
- The average reward over time is depicted using the following formula for updating the running average:

$$R(i) = \frac{(i-1) \cdot R(i-1) + RR}{i}$$

Results

MENACE (Tic-Tac-Toe)

MENACE successfully learns to play Tic-Tac-Toe by reinforcing successful moves and punishing losing ones. Over time, its strategy improves, eventually leading to better performance against random or human opponents.

Binary Bandit Problem (Stationary)

The epsilon-greedy algorithm for the binary bandit problem converges to the action with the higher expected reward. As the agent explores and exploits, the average reward stabilizes, favoring the optimal action.

Non-Stationary Bandit Problem

In the non-stationary bandit problem, the epsilon-greedy algorithm with a learning rate adapts to changing reward distributions. This allows the agent to track the best action even as rewards shift, with the average reward gradually improving through exploration and exploitation.

Week 8 :Optimal Decision-Making in Stochastic Environments: Applications of MDPs in Navigation and Bicycle Rental Problems

This report explores sequential decision-making in stochastic environments using Markov Decision Processes (MDPs) and Dynamic Programming (DP). The objective is to compute optimal policies for decision-making problems where outcomes are uncertain. The report discusses two main case studies:

1) **Stochastic Navigation Problem:** An agent navigating a grid world with uncertain movements and rewards.
2) **Bicycle Rental Problem:** Optimizing the movement of bicycles between two locations to maximize profits under uncertain demands and returns.

These problems demonstrate how MDPs provide solutions to real-world decision-making scenarios where uncertainty and randomness play a significant role. This report employs the Value Iteration and Policy Iteration algorithms to derive the optimal solutions.

III. Problem for In-Lab Discussion

A. Stochastic Navigation Problem

1) Problem Statement: In this problem, an agent navigates a 4x3 grid environment with stochastic transitions. The agent must reach one of two terminal states with rewards of +1 and -1, while avoiding penalties in non-terminal states. The agent chooses from four possible actions: Up, Down, Left, and Right. However, each action is stochastic, with an 80% chance of success in the intended direction and a 10% chance for each perpendicular direction.

The objective is to find the value function corresponding to the optimal policy using the Value Iteration algorithm.

2) Concepts of MDP and Value Iteration: Markov Decision Processes (MDPs) provide a formal framework to model decision-making in environments with uncertainty. The core components of an MDP are:

- **States** (S): The possible positions of the agent within the grid.
- **Actions** (A): The possible movements of the agent, which include up, down, left, and right.
- **Transition Model** ($P(s'|s,a)$): A probability distribution that describes the likelihood of transitioning to a new state s' when action a is taken from state s.
- **Rewards** ($R(s)$): The rewards associated with each state.
- **Discount Factor** (γ): A factor used to discount future rewards. A higher value of γ places more emphasis on future rewards.

In this problem, the agent faces uncertainty in its movements, making the environment stochastic. The goal is to determine the optimal policy, which is a mapping from states to actions that maximizes the expected long-term reward.

Value Iteration is an algorithm used to compute the optimal policy in an MDP by iteratively improving the value function $V(s)$. The Bellman equation is central to this algorithm, as it defines the relationship between the value of a state and the expected rewards from taking an action:

$$V(s) = \max_a \sum_{s'} P(s'|s,a) \cdot [R(s') + \gamma \cdot V(s')] \tag{1}$$

Here:

- $V(s)$ is the value of state s, representing the maximum expected reward starting from state s and following the optimal policy.
- a is an action taken from state s.
- $P(s'|s,a)$ is the probability of transitioning from state s to state s' when action a is taken.
- $R(s')$ is the reward for reaching state s'.
- γ is the discount factor, controlling the importance of future rewards.

The algorithm proceeds by iteratively updating the value function $V(s)$ until convergence, i.e., the values no longer change significantly. The iteration stops when the difference between the new and old value function is less than a predefined threshold θ.

3) Algorithm: Value Iteration: The pseudocode for the Value Iteration algorithm is as follows:

Algorithm 1 Value Iteration

```
1: Initialize $V(s) = 0$ for all states $s$
2: Set convergence threshold $\theta$
3: Set discount factor $\gamma$
4: repeat
5:     $\Delta = 0$
6:     for each state $s$ in the grid do
7:         $v = V(s)$
8:         for each action $a$ in actions do
9:             Calculate the expected value for action $a$ based on Bellman equation
10:            Update $V(s)$ to the maximum value among all actions
11:        end for
12:        $\Delta = \max(\Delta, |v - V(s)|)$
13:    end for
14: until $\Delta < \theta$
15: Return $V(s)$
```

4) Results and Analysis: After applying Value Iteration, the agent's optimal policy was determined. The optimal policy directs the agent towards the positive terminal state and avoids the negative one, accounting for stochastic movements.

B. Bicycle Rental Problem

1) Problem Statement: The Bicycle Rental Problem involves two locations where the number of bikes requested and returned follows a Poisson distribution. The task is to determine the optimal number of bikes to move between the two locations each day to maximize profits, while considering operational costs and parking constraints. The goal is to compute the optimal policy for transferring bikes between the locations.

2) Concept of Policy Iteration: Policy Iteration is an algorithm used to find the optimal policy for a given MDP. It alternates between two steps:

- **Policy Evaluation:** For a given policy, compute the value function $V(s)$ by solving the Bellman equation for the current policy.
- **Policy Improvement:** Given the value function $V(s)$, update the policy $\pi(s)$ by selecting the action that maximizes the expected return for each state.

The Bellman equation for Policy Evaluation is:

$$V(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) \cdot V(s') \tag{2}$$

The policy improvement step selects the action that maximizes the expected value:

$$\pi'(s) = \arg\max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) \cdot V(s') \right] \tag{3}$$

This process continues until the policy converges, meaning no further changes are possible.

Algorithm 2 Standard Policy Iteration

```
1:  Initialize $\pi(s)$ for all states $s$
2:  Initialize $V(s) = 0$ for all states
3:  repeat
4:      Policy Evaluation:
5:      repeat
6:          for each state $s$ do
7:              Calculate $V(s)$ using the current policy $\pi(s)$
8:          end for
9:      until Convergence of $V(s)$
10:     Policy Improvement:
11:     for each state $s$ do
12:         Update $\pi(s)$ by selecting the action $a$ that maximizes the expected value
13:     end for
14: until No change in $\pi(s)$
15: Return the optimal policy $\pi(s)$
```

3) Algorithm: Standard Policy Iteration: The pseudocode for the Policy Iteration algorithm is as follows:

4) Results and Analysis: In this problem, the Policy Iteration algorithm derived the optimal policy for bike transfers between two locations. The optimal policy balanced the customer demand with bike availability, minimizing transfer and parking costs.

5) Modifications for Extended Problem: In the extended version of the problem, we introduced additional constraints:

- An employee at the first location can shuttle one bike to the second location for free, but each additional bike costs INR 2.
- If more than 10 bikes are parked at a location, an additional INR 4 is charged for parking.

These modifications impacted the transfer policy, encouraging the movement of bikes between locations without exceeding parking limits, which reduces costs.

6) Algorithm: Extended Policy Iteration with Additional Constraints: Here is the modified pseudocode that accounts for the additional constraints:

IV. Conclusion

Through these lab assignments, we explored various concepts in probabilistic modeling, decision-making, and reinforcement learning in a hands-on way. Working with Bayesian Networks, we learned how to model dependencies between variables and build classifiers to predict outcomes. Using Gaussian Hidden Markov Models, we analyzed financial data to uncover hidden patterns like periods of high or low market volatility, which gave us a deeper understanding of market behavior. Hopfield Networks helped us tackle problems like error correction and optimization, including solving the Traveling Salesman Problem. Reinforcement learning introduced us to strategies like exploration versus exploitation and adapting algorithms to dynamic environments. With Markov Decision Processes, we applied value and policy iteration to practical problems, like managing Gbike rentals efficiently. These labs combined theoretical concepts with real-world scenarios, helping us strengthen our problem-solving skills and gain valuable insights into these fields.n optimal policies from their environment, making them adaptable and more efficient.

Algorithm 3 Extended Policy Iteration with Constraints

```
1:  Initialize $\pi(s)$ for all states $s$
2:  Initialize $V(s) = 0$ for all states
3:  repeat
4:      Policy Evaluation:
5:      repeat
6:          for each state $s$ do
7:              Calculate $V(s)$ considering employee bike shuttle and parking fees
8:          end for
9:      until Convergence of $V(s)$
10:     Policy Improvement:
11:     for each state $s$ do
12:         Update $\pi(s)$ by selecting the action $a$ that minimizes transfer and parking costs
13:     end for
14: until No change in $\pi(s)$
15: Return the optimal policy $\pi(s)$
```

Repository Link

x The GitHub repository link can be found here.

References

[1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction.* Cambridge: MIT Press.

[2] Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

[3] MacKay, D. J. C. (2003). *Information Theory, Inference and Learning Algorithms.* Cambridge University Press.

[4] Mandziuk, J. (1999). Solving the Travelling Salesman Problem with a Hopfield-Type Neural Network. *Biological Cybernetics*, 81(3), 199-206.

[5] Michie, D. (1968). *The Games Machines.* London: J. Murray.