Improving Healthcare Policies Using Reinforcement Learning on Patterns of Service Utilization

by

Nadia Enhaili

B.Sc., Université du Québec à Montréal, 2022

Project Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Statistics and Actuarial Science
Faculty of Science

SIMON FRASER UNIVERSITY
Summer 2024

Declaration of Committee

Name:	**Nadia Enhaili**
Degree:	**Master of Science**
Thesis title:	**Improving Healthcare Policies Using Reinforcement Learning on Patterns of Service Utilization**
Committee:	**Chair:** Rachel Altman Professor, Statistics and Actuarial Science
	Lloyd T. Elliott Supervisor Associate Professor, Statistics and Actuarial Science
	Derek Bingham Committee Member Professor, Statistics and Actuarial Science
	Joan Hu Examiner Professor, Statistics and Actuarial Science

Abstract

Reinforcement Learning (RL) is an important class of methods in Artificial Intelligence (AI), particularly for optimization problems and decision-making under uncertainty. However, practical and ethical concerns in healthcare settings can limit the application of traditional RL methods, requiring innovative approaches. This thesis explores the application of RL methods in healthcare to evaluate treatment strategies.

We begin with an overview of RL, followed by an introduction to Q-Learning and Dyna-Q, two fundamental RL algorithms. We demonstrate the application of these algorithms using a simulated robot, AdventureBot, navigating a grid world. We then introduce Hidden Markov Mixture Models (HMMMs) as a method for extracting patient subgroups with distinct patterns from longitudinal data, which we apply to a simulated dataset. Finally, we describe our proposed pipeline for integrating HMMMs with CFRL to evaluate healthcare policies in an offline setting.

Keywords: counterfactual reinforcement learning, offline learning, dependent mixture models, patient service utilization

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Lloyd T. Elliott, for his guidance and endless support throughout this research.

I am also thankful to Dr. Derek Bingham for his valuable feedback and encouragement, and to Dr. Joan Hu and Dr. Rachel Altman for being part of my defence committee.

I would like to acknowledge the Department of Statistics and Actuarial Science at Simon Fraser University for providing a resourceful and stimulating research environment.

Finally, I am grateful to my family and friends for their love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reinforcement Learning (RL) is a pivotal technology in Artificial Intelligence (AI), driving substantial progress across many domains. One such example is Boston Dynamics' Spot robot, which is capable of autonomously navigating hazardous environments using unsupervised RL techniques. This robot was deployed on Japan's Fukushima nuclear power plant in 2022 to navigate difficult terrain, carry substantial weight, open and close doors, and collect analysis samples (Wessling, 2023). Another example is Google's AlphaGo, which defeated the world champion Lee Sedol in the game of Go in 2016 (Moyer, 2016). At the time, the consensus among artificial intelligence and game theory experts was that such a feat was at least a decade away, given that the best Go-playing programs were still only at the level of an advanced amateur (Brown, 2016; Wedd, 2018). These examples highlight the capabilities of RL in solving problems when the environment is uncertain and complex.

1.1 Reinforcement Learning in Healthcare

In the field of healthcare, integrating RL methods has the potential to enable more precise and personalized medical interventions. This could result in enhanced patient care outcomes, increased diagnostic precision, or sophisticated predictive models for various diseases. However, healthcare applications of RL must fundamentally differ from their typical use in other domains. Healthcare providers must abide by ethical and regulatory limitations, obtain informed consent from patients, adhere to written protocols, and ensure robust oversight when carrying out actions. Additionally, they must safeguard patient confidentiality to comply with regulations such as PIPEDA (Personal Information Protection and Electronic Documents Act) in Canada and HIPAA (Health Insurance Portability and Accountability Act) in the United States.

Navigating these complexities, health authority regions such as the Vancouver Island Health Authority (VIHA) engage in various research and development (R&D) projects to advance medical knowledge and improve patient outcomes (Vancouver Island Health Authority, 2023b). VIHA is one of five geographically-based health authorities in British

Columbia (Government of British Columbia, 2023), providing health care services to 885,000 individuals (Vancouver Island Health Authority, 2023a). As part of its R&D initiatives, VIHA collects and analyzes patient service utilization (PSU) data in an effort to optimize patient care and resource allocation. PSU data includes records of patient interactions with healthcare services, such as hospital visits, consultations, and treatments.

In this thesis, we explore the application of RL methods in the healthcare domain, focusing on the use of reinforcement learning to improve patient treatment strategies. We begin by providing an overview of RL and its core concepts. Specifically, we introduce Q-Learning as a fundamental RL algorithm to learn optimal behaviour policies in environments that provide rewards based on actions. We then discuss Dyna-Q, an extension of Q-Learning that incorporates an offline component to accelerate learning without requiring additional real-world interactions. We demonstrate the application of Q-Learning and Dyna-Q using an unsupervised simulated robot, AdventureBot. The robot navigates a grid world to find a treasure, while avoiding obstacles.

We then introduce Hidden Markov Mixture Models (HMMMs) as a method for analyzing PSU data to identify patient subgroups with distinct service utilization patterns. We apply this method to simulated PSU data to demonstrate its utility in identifying patient subgroups that display similar service utilization patterns. Finally, we introduce Counterfactual Reinforcement Learning (CFRL) as an advanced technique for evaluating policies in an offline setting. We propose a pipeline for integrating HMMs with CFRL to evaluate treatment strategies in an offline setting. The proposed steps of the pipeline are as follows:

1. Preprocessing PSU data to extract a patient service utilization matrix for each patient. The matrix dimensions are $T \times M$, where T is the number of time periods where service utilization is recorded, and M is the number of distinct services available to patients.

2. Utilizing the `depmixS4` R package to extract patient subgroups (latent states) from the PSU matrix.

3. Use the extracted latent states to generate the input for the CFRL algorithm to evaluate offline treatment strategies. Specifically, the latent states are used to define a low-dimensional representation of a patient's time series, which is effectively a summary of the patient's service utilization over time.

For step 1, we define a preprocessing algorithm in Section 4.4. For step 2, we provide an experiment on simulated data in Sections 4.1 to 4.3. Our results show that the DepmixS4 model performs well in extracting patient subgroups from the simulated PSU data when the dataset presents strong patterns. Step 3 is further described in Section 4.5. However, we do not implement step 3 in this thesis, as it requires extensive additional work, including the collaboration of domain experts to define essential elements such as 'evaluation policies' and

'rewards.' In addition, we provide a demonstration of some concepts in offline RL (Dyna-Q) in Section 2.3.1.

1.2 Related Work

Dynamic treatment regimes (DTRs) offer a structured approach to personalized medicine by tailoring treatment decisions to an individual's changing status over time. Unlike traditional treatment methods, which apply a uniform approach to all patients, DTRs adjust the level and type of treatment based on the evolving needs of each patient (Murphy, 2003). DTRs are particularly relevant in chronic diseases such as diabetes, cancer, and mental health disorders, where treatment effectiveness can vary significantly across patients. By incorporating patient-specific information, DTRs can optimize treatment strategies to improve patient outcomes and reduce costs.

In healthcare applications, both DTRs and RL aim to optimize a sequence of actions to achieve a desired outcome. In subsequent chapters, we describe counterfactual reinforcement learning (CFRL) as a method for evaluating what-if outcomes that are never observed in the real world. In contrast, DTRs typically involve using longitudinal data to construct estimators of optimal decision rules based on the outcomes that are directly measurable and recorded during the treatment process.

Chapter 2

Reinforcement Learning

2.1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) is a critical area of study in machine learning and AI, characterized by agents learning to make decisions through interaction with an environment. This decision-making process hinges on the concept of learning from experience, with the objective of maximizing a cumulative reward over time. Central to RL are three core components: states (s), actions (a), and rewards (r). The RL agent's task is to establish a behaviour policy $\pi(a|s)$, i.e. a mapping from states to actions, that will maximize expected rewards across a time horizon. This time horizon can be finite in the case of episodic tasks or infinite for continuous tasks, the latter of which could require discounting future rewards to ensure convergence (Sutton & Barto, 2018).

Learning an optimal policy often requires the agent to balance the exploration of unfamiliar states and actions with the exploitation of existing knowledge to optimize long-term outcomes. The theoretical framework of RL is closely linked to dynamic programming, Markov decision processes (MDPs), dynamic treatment regimes, and optimal control theory, all of which emphasize probabilistic decision making under uncertainty (Murphy, 2003; Sutton & Barto, 2018). For a comprehensive understanding of these principles and their applications in various fields, Sutton and Barto's text *Reinforcement Learning: An Introduction* provides an in-depth exploration of the subject matter, detailing both foundational theories and advanced methodologies in RL (Sutton & Barto, 2018).

2.1.1 Offline vs. Online Reinforcement Learning

RL methods can be broadly divided into offline and online paradigms. Online learning involves the agent continuously learning and adapting its strategy based on real-time interactions with the environment. This approach is dynamic and allows the agent to immediately incorporate new experiences into its learning process.

On the other hand, offline learning, also known as batch learning, involves the agent learning from a fixed dataset, without further interaction with the environment. It is par-

ticularly useful in scenarios for which interacting with the environment is costly, risky, or not feasible (Sutton & Barto, 2018).

2.1.2 Model-based vs. Model-free Reinforcement Learning

Within the offline and online paradigms, RL methods can be further categorized into model-based and model-free approaches. These categories refer to the underlying mechanisms through which the agent makes decisions.

Model-based RL involves the agent developing an internal model of the environment as it learns it, which the agent uses to simulate and predict future states and rewards. This approach allows for planning and decision-making based on the model's predictions. This is particularly useful for scenarios in which the environment's dynamics are broadly known and can be modeled. For example, in a game of chess, a model-based agent can simulate future moves and outcomes because the rules of the game are well-defined and deterministic.

In contrast, model-free RL methods do not rely on an internal model of the environment. Instead, they focus on learning optimal policies or value functions directly from interactions with the environment. This approach is more flexible and can be applied to environments where the dynamics are unknown or difficult to model. For example, in autonomous driving, the dynamics of the environment are uncertain and may change rapidly, making it challenging to develop an accurate model of the environment. A model-free agent would learn optimal policies through trial and error, without explicitly modeling the environment.

Within the domain of model-free reinforcement learning, Q-Learning is a foundational algorithm. It enables agents to learn optimal policies by estimating the expected rewards associated with taking specific actions in given states. We introduce Q-Learning in the following section.

2.2 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm that can be used to learn optimal policies in Markov decision processes (MDPs). The algorithm works by learning an action-value function, $Q(a|s)$, that estimates the expected reward from taking action a in state s. An optimal behaviour policy $\pi(a|s)$ can then obtained by selecting the action with the highest value Q-value for each state (Sutton & Barto, 2018; Watkins & Dayan, 1992).

The Q-Learning algorithm runs for a fixed number of episodes, with each episode consisting of a sequence of steps. At each step, the agent selects an action based on the ε-greedy policy, i.e. with probability ε, the agent selects a random action among all available actions, and with probability $1 - \varepsilon$, the agent selects the action with the highest Q-value for the current state. The agent then takes the selected action, observes the reward and the next state, and updates the Q-value for the previous state-action pair using the Q-Learning for-

mula (Sutton & Barto, 2018). The episode ends when a terminal state is reached, i.e. the agent reaches a goal or fails to achieve its objective, and a new episode begins.

We formalize the algorithm and give an example application in the next section.

2.2.1 Algorithm and Example

Data: Learning rate (α), discount factor (γ)
Result: Estimate of optimal policy $\pi(a|s)$

Initialize $Q(a|s)$ arbitrarily for all state-action pairs;

for *each episode* **do**
  Initialize state s;
  while *state s is not terminal* **do**
    Choose action a using policy derived from $Q(a|s)$ (e.g., ε-greedy);
    Take action a, observe reward r and next state s';
    $Q(a|s) \leftarrow Q(a|s) + \alpha[r + \gamma \max_{a'} Q(a'|s') - Q(a|s)]$;
    $s \leftarrow s'$;
  end
end

Algorithm 1: The Q-Learning algorithm. The algorithm runs for a set number of episodes. Each episode consists of a sequence of steps where the agent interacts with the environment, updating the Q-values based on the observed rewards. (Sutton & Barto, 2018)

To illustrate reinforcement learning and to offer a new framework for evaluating RL algorithms, we construct an example called AdventureBot, a simulated robot navigating a $D \times D$ grid world. AdventureBot's mission is to find a treasure located in one of the grid cells. The grid also contains one obstacle that the robot must avoid. Each cell coordinate in the grid represents a state, and AdventureBot can move up, down, left, or right (actions). The Q-Learning algorithm works as follows in this scenario:

1. **Initial Setup**:

   - The grid is a $D \times D$ matrix with an obstacle and a treasure that are randomly placed.

   - The Q-table is initialized with all zeros. It has a row for every state (grid cell) and a column for each possible action, so it's a $(D \times D) \times 4$ matrix.

- The penalty for hitting an obstacle is -1, and the reward for finding the treasure is $+1$. All other rewards (i.e. for moving to an empty cell) are 0. We set a maximum number of steps per episode, after which an episode must end.

2. **For each episode**:

   (a) AdventureBot starts at a random position. It picks an action to take based on the ε-greedy policy, i.e. with probability ε, it chooses a random action, and with probability $1 - \varepsilon$, it chooses the action with the highest Q-value for the current state.

   (b) AdventureBot moves to the next state and receives a reward based on the action taken. The Q-value for the previous state-action pair is then updated using the Q-Learning formula defined in Algorithm 1.

   (c) AdventureBot will repeat steps (a) and (b) until a terminal state is reached, i.e. AdventureBot finds the treasure, hits the obstacle, or reaches the maximum number of steps per episode.

3. **Convergence**:

   - After many episodes, the Q-values begin to converge, which means AdventureBot has learned an effective strategy for navigating the grid. With every new episode, it will require fewer and fewer steps to reach the treasure, and the Q-values will stabilize.

   - After convergence, the result will be a Q-table that approximates the optimal Q-values for each state-action pair, which can be used to estimate the optimal policy $\pi(a|s)$ for AdventureBot.

In this example, AdventureBot starts with no knowledge of the environment. Through trial and error, AdventureBot learns an optimal path to the treasure. The Q-Learning algorithm allows it to balance exploration (trying out uncertain paths) with exploitation (using known good paths), leading to a gradual improvement in its strategy. Over time, it becomes proficient in navigating the grid, demonstrating the power of Q-Learning in developing effective behaviors through interaction with an environment.

Figure 2.1 illustrates the number of steps taken by AdventureBot at each episode. As expected, the number of steps per episode decreases over time, indicating that AdventureBot can learn an optimal policy for navigating the grid. Code for AdventureBot's Q-Learning algorithm is included in Appendix A.1.

In RL, test environments provide a controlled setup for comparing algorithms. One such environment is *Gymnasium*'s Frozen Lake. *Gymnasium* (formerly *OpenAI Gym*) is

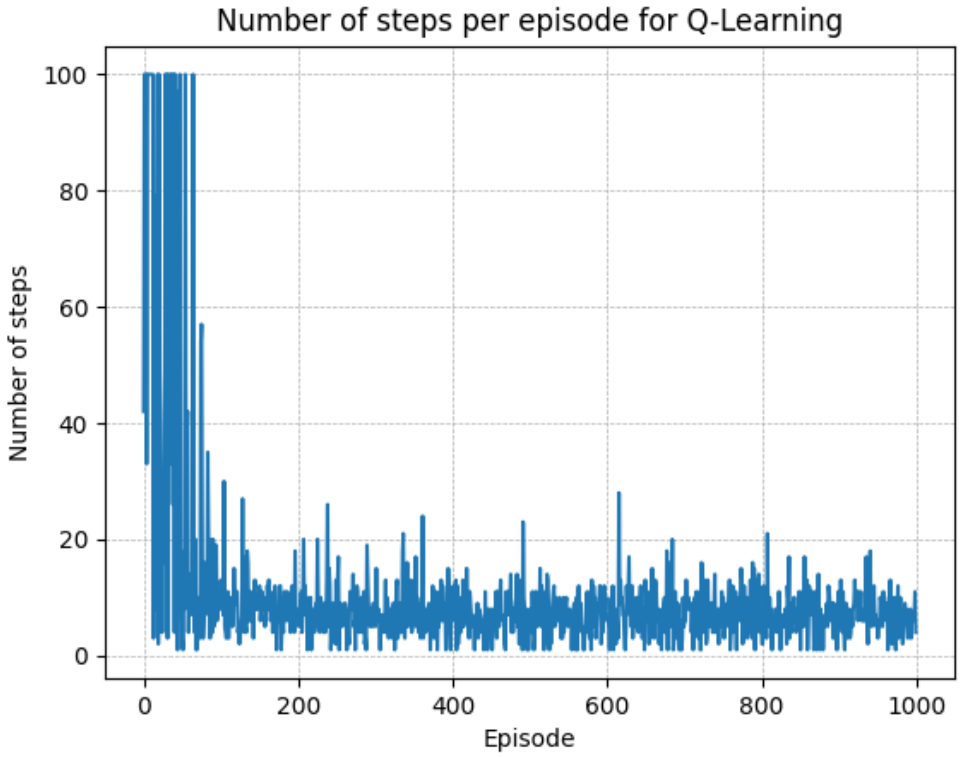

Figure 2.1: **AdventureBot Q-Learning Efficiency.** The plot illustrates a decrease in the number of steps leading to a terminal state at each episode, suggesting AdventureBot's increasing proficiency in navigating a 7×7 grid. Parameters: learning rate ($\alpha = 0.2$), discount factor ($\gamma = 0.9$), exploration rate ($\varepsilon = 0.1$), episodes (1000), and maximum steps per episode (100).

a popular open-source library that provides a wide range of environments for testing and benchmarking RL algorithms. Frozen Lake is a grid world similar to AdventureBot, in which the agent must navigate a frozen lake to reach a goal without falling into holes (*Frozen Lake - Gymnasium Documentation*, 2023). While AdventureBot and Frozen Lake are similar Q-Learning environments, the latter is limited to a single reward structure, two grid size options, and other preset environment configurations. AdventureBot, on the other hand, offers granular control on the environment setup, which makes it a great complement to the *Gymnasium* environments for testing and evaluating RL algorithms.

Building on the foundation of Q-Learning, Dyna-Q introduces an innovative extension that incorporates both direct and simulated experiences, significantly enhancing the agent's ability to learn efficiently.

2.3 Dyna-Q, an extension of Q-Learning

In healthcare applications, regulatory constraints make real-world experimentation challenging. In light of these challenges, we explore Dyna-Q, an extension of Q-Learning that incorporates model-based planning in order to accelerate the learning process. Dyna-Q offers a unique advantage by enabling offline learning, in which the agent learns from historical data without interacting with the environment (Sutton, 1990). While still reliant on the learned policy for selecting actions, Dyna-Q's algorithm is a step toward a reinforcement learning approach that relies on historical data for learning.

Dyna-Q can be conceptualized as an enhanced variant of the Q-Learning algorithm, wherein after each real interaction step, a series of additional steps are taken based on simulated experiences derived from an internal model of the environment (Sutton, 1990). Essentially, Dyna-Q retains the core mechanism of Q-Learning—learning an action-value function, $Q(a|s)$—and augments it by incorporating a model-based component. The model of the environment, constructed through observed interactions, is used to generate simulated experiences. These simulated experiences, in turn, provide supplementary data for updating the action-value function without having to take any additional actions in the environment (Sutton & Barto, 2018).

In each iteration, Dyna-Q proceeds as follows: after the standard Q-Learning update using a real interaction, the algorithm performs a predefined number of updates using experiences drawn from the internal model. This effectively means that for each actual step in the environment, Dyna-Q performs multiple simulated 'planning' steps, updating its $Q(a|s)$ values as if these simulated steps were real. This additional phase of model-based updates serves to accelerate the learning process.

In the next section, we formalize the Dyna-Q algorithm and describe it in the context of AdventureBot.

2.3.1 Algorithm and Example

Data: Learning rate (α), discount factor (γ), number of planning steps (n)
Result: Estimate of optimal policy $\pi(a|s)$

Initialize $Q(a|s)$ arbitrarily for all state-action pairs;
Initialize the internal model of the environment $M(a|s)$ as empty;

for *each episode* **do**
  Initialize state s;
  while *state s is not terminal* **do**
    Choose action a from s using policy derived from Q (e.g., ε-greedy);
    Take action a, observe reward r and next state s';
    $Q(a|s) \leftarrow Q(a|s) + \alpha[r + \gamma \max_{a'} Q(a'|s') - Q(a|s)]$;
    Update the internal model, $M(a|s) \leftarrow (a, r, s')$;
    for $i = 1$ *to* n **do**
      Select a random previously observed state s_i and action a_i;
      Retrieve (r_i, s_i') corresponding to (a_i, s_i) using $M(a_i|s_i)$;
      $Q(a = a_i|s = s_i) \leftarrow Q(a_i|s_i) + \alpha[r_i + \gamma \max_{a'} Q(a'|s_i') - Q(a_i|s_i)]$;
    end
    $s \leftarrow s'$;
  end
end

Algorithm 2: Dyna-Q Algorithm with n Planning Steps. The algorithm combines Q-Learning with model-based planning to accelerate the learning process.

Consider the same AdventureBot described in Section 2.2.1, navigating a $D \times D$ grid world. In Dyna-Q, AdventureBot learns not only from direct interactions with the grid, but also from simulated scenarios generated by its internal model. The algorithm steps for AdventureBot can be described as follows:

1. **Initial Setup**: Similar to the Q-Learning setup with the addition of the model M, which is initialized alongside the table of Q-values. M can be represented as a $(D \times D) \times 4$ matrix, with each cell containing a tuple of the form (r, s'), where r is the reward for taking action a in state s and s' is the resulting state. Another way to think of M is as a lookup table for the past outcomes (one reward and one new state) that resulted from taking an action in a given state.

2. **For each episode**:

   (a) AdventureBot follows the same real-world interaction steps as in Q-Learning.

(b) After updating the Q-value with real-world data, AdventureBot then conducts n simulated planning steps. In each planning step, it randomly selects a state-action pair it has experienced before, retrieves the previously observed outcome (r and s') of that action from its model M, and updates the Q-table as if the simulated experience were real.

3. **Convergence**:

   - The added simulated experiences help AdventureBot to learn faster, as it can effectively experience many more scenarios within each actual episode. The Q-values converge to the optimal policy more quickly compared to standard Q-Learning.

Figure 2.2 compares the number of steps per episode for Q-Learning and Dyna-Q. As expected, Dyna-Q converges to the optimal policy more quickly, demonstrating the benefits of incorporating simulated experiences into the learning process. It is important to note that Dyna-Q requires a deterministic environment, as the model M must accurately predict the outcomes of actions in order to generate meaningful simulated experiences. Code for the Dyna-Q algorithm is included in Appendix A.2.

While Dyna-Q offers significant advantages in accelerating learning and improving decision-making, it is not a complete solution for offline learning. The agent still needs to interact with the environment to update its model and generate simulated experiences. In the next section, we introduce Counterfactual Reinforcement Learning (CFRL) as a method for evaluating policies in an offline setting, enabling robust policy evaluation without real-world interactions.

2.4 Counterfactual Reinforcement Learning

Counterfactual Reinforcement Learning (CFRL) is a reinforcement learning technique that seeks to construct and evaluate counterfactual scenarios, i.e. alternative scenarios that did not happen. CFRL extends the traditional RL paradigm by incorporating counterfactual state-action pairs, (s', a'), and their corresponding reward, r' (Gajcin & Ivana, 2022). CFRL offers significant advantages in environments for which real-world experimentation is costly or infeasible. By analyzing counterfactual scenarios, CFRL enables more robust policy evaluation and development without additional real-world interactions.

Moreover, CFRL provides a framework to quantify the impact of past decisions, which is crucial in retrospective analysis and policy refinement. This approach goes beyond methods like Dyna-Q by entirely eliminating the need for continuous real interactions. Instead, CFRL relies solely on historical data to simulate and evaluate different strategies, making it a powerful tool for offline reinforcement learning. This is particularly relevant in healthcare applications, where real-world experimentation is not always feasible (Gajcin & Ivana, 2022).

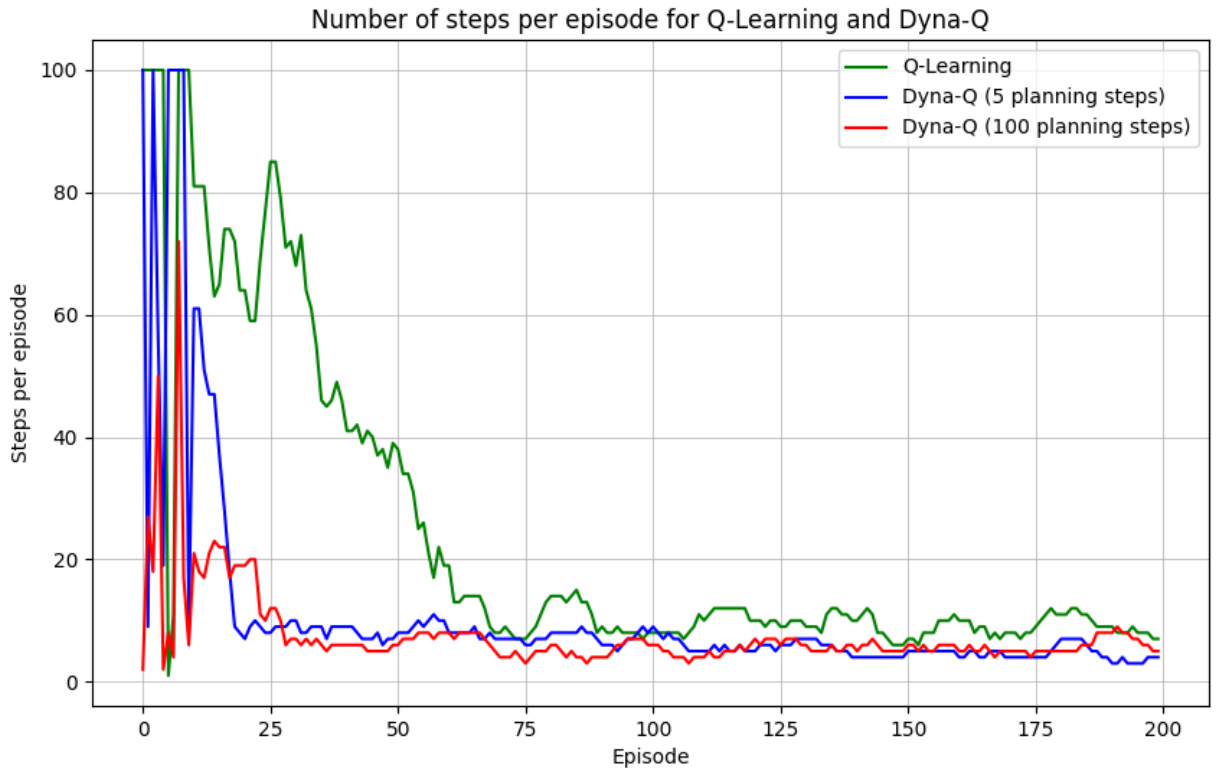

Figure 2.2: **Comparative Analysis of Learning Efficiency for Q-Learning and Dyna-Q.** This figure illustrates the performance of Q-Learning (green line) against Dyna-Q with 5 planning steps (blue line) and 100 planning steps (red line) in a 7×7 grid environment. Dyna-Q performances stabilizes after episode 25, whereas Q-Learning requires three times as many episodes to reach the same performance. Notably, the line for Dyna-Q with 100 planning steps demonstrates a rapid and substantial drop in steps per episode, indicating a swifter learning process compared to Dyna-Q with 5 planning steps and Q-Learning. This trend highlights the accelerated learning and problem-solving capabilities endowed by a higher number of planning steps in the Dyna-Q algorithm. Parameters: learning rate ($\alpha = 0.2$), discount factor ($\gamma = 0.9$), exploration rate ($\varepsilon = 0.1$), episodes (200), and maximum steps per episode (100).

These advantages highlight CFRL as a valuable technique for evaluating healthcare policies using historical patient service utilization data. CFRL can enable healthcare providers to optimize treatment strategies, improve patient outcomes, and enhance resource allocation without the need for real-world experimentation. In the next sections, we explore the foundational principles of CFRL and its application in healthcare policy evaluation.

2.4.1 Foundational Knowledge

Formally, CFRL seeks to estimate the expected reward of a target policy π using observed data that was generated under a different, fixed policy π_0 (Joachims & Swaminathan, 2016). Since CFRL is a reinforcement learning algorithm, the notation is consistent with the standard RL framework, with states denoted as s, actions as a, and rewards as r. Our formulation of CFRL assumes discrete states and actions, as is common in many RL applications. Since we observe the data generated by π_0, and we know π_0, we can estimate the expected reward $\hat{R}$ of π_0 accurately as:

$$\hat{R}(\pi_0) = \frac{1}{N} \sum_{i=1}^{N} r_i \tag{2.1}$$

The key challenge in CFRL is to accurately estimate $\hat{R}(\pi)$, the expected reward under the target policy π, based on the data observed under the behavior policy π_0 without executing the target policy in the environment. One approach to this problem is to try to model the discrepancy between the current policy and the target policy, and correct for it. We define the propensity p_i as the probability of taking action a_i under the current policy π_0 given an observed state s_i. Mathematically, the propensity is written as:

$$p_i = \pi_0(A_i = a_i | s_i) \tag{2.2}$$

If we can compute these propensities for each action a and each state s, we can compute an unbiased estimator of the reward of each action using the Inverse Propensity Scoring (IPS) estimator (Joachims & Swaminathan, 2016). Note that while we use the notation $\hat{R}$ for the IPS estimator, Joachims and Swaminathan (2016) use $\hat{U}$.

The IPS estimator of the reward of taking action a is defined as:

$$\hat{R}_{\text{IPS}}(a) = \frac{1}{N} \sum_{i=1}^{N} \frac{\mathbf{I}(a_i = a)}{p_i} r(a_i | s_i), \tag{2.3}$$

where $\mathbf{I}(a_i = a)$ is an indicator function that is 1 if the action a_i is equal to a and 0 otherwise, and $r(a_i|s_i)$ is the reward observed after taking action a_i in state s_i. This equation can be conceptually understood as weighting the observed rewards by the inverse of the propensity of taking the action that led to the reward. For example, if an action was taken with a high propensity, the observed reward will be downweighted, and vice versa.

Given $\{(s_1, a_1, r_1), ..., (s_N, a_N, r_N)\}$ collected under π_0, the IPS estimator can be used to obtain an unbiased estimate of the reward for a target policy π as follows:

$$\hat{R}_{\text{IPS}}(\pi) = \frac{1}{N} \sum_{i=1}^{N} \frac{\pi(a_i | s_i)}{p_i} r(a_i | s_i), \tag{2.4}$$

where $p_i = \pi_0(a_i|s_i)$.

Joachims and Swaminathan (2016) demonstrate that the IPS method performs almost as well as if we had actually observed the data under the target policy π, and significantly outperforms approaches that try to model the reward directly. One such approach would be to fit a regression model on the reward observed under the behavior policy π_0 and then use this model to predict the reward under the target policy π.

Note that CFRL cannot be directly applied to VIHA's PSU data as it requires the states to be discrete and time-invariant. In the next sections, we introduce Hidden Markov Mixture Models (HMMMs) as a method for extracting time-invariant latent states from PSU data to enable the application of CFRL.

Chapter 3

Hidden Markov Mixture Models

VIHA datasets comprise records of patient interactions across more than 200 health services. The frequencies of these service accesses vary across patients and change over time. To capture these patterns and assist in dimensionality reduction, we consider Hidden Markov Models (HMMs) as a method for clustering patients based on their service utilization patterns. HMMs are well-suited for modeling sequential data and can capture underlying structures in the data that may not be immediately apparent. In the next section, we provide an overview of HMMs and define the components of HMMs in the context of this thesis.

3.1 Overview of Hidden Markov Models

Hidden Markov Models (HMMs) are a class of probabilistic models that are widely used for modeling time series data (Rabiner & Juang, 1986). HMMs assume that the system being modeled is a Markov process with unobserved (hidden) states. The system emits observed 'symbols' at each time step, and the sequence of these symbols is used to infer the sequence of hidden states. Formally, a HMM is defined by the following components:

- **Observations** O: A sequence of observed symbols, $O = (O_1, O_2, \ldots, O_T)$, where O_t is the observation at time t. In our work, we assume the observations are discrete values.

- **Hidden States** S: A sequence of hidden states, $S = (S_1, S_2, \ldots, S_T)$, where S_t is the hidden state at time t. We assume the state space is also a vector of discrete values.

- **Initial State Probabilities** p_0: A probability distribution over the initial hidden states, $\Pr(S_1 = i)$. These probabilities range from 0 to 1 and must sum to 1: $\sum_i \Pr(S_1 = i) = 1$.

- **State Transition Probabilities** p: A matrix of transition probabilities, $\Pr(S_{t+1} = j|S_t = i)$, where S_t is the latent state at time t. These probabilities also range from 0 to 1 and must sum to 1: $\sum_j \Pr(S_{t+1} = j|S_t = i) = \sum_i \Pr(S_{t+1} = j|S_t = i) = 1$.

- **Emission Probabilities** b: A matrix of probabilities of observing each symbol given the hidden state, $\Pr(O_t = o|S_t = s)$. These probabilities also range from 0 to 1. In our work, the observed symbols are the patients' service accesses. Since a patient can access any combination of services in a time period, the emission probabilities b do not need to sum to 1.

By modeling patient data using HMMs, we can capture hidden signals and patterns in the data. Additionally, the hidden states inferred by the HMM can be used to create an alternative representation of the data, reducing the dimensionality of the data and simplifying subsequent analyses. Dimensionality reduction is particularly significant in our work given that the vector of service utilization for each patient has 2^M possible states (1 if a service was used, 0 otherwise), where M is the number of healthcare services available.

Hidden Markov Models lay the foundation for more advanced models, such as Dependent Mixture Models, which can capture more complex relationships and patterns in the data by allowing the mixture components to depend on observations or latent variables that change over time. In the next sections, we introduce Mixture Models and Dependent Mixture Models.

3.2 Mixture Models

Mixture models are a class of probabilistic models that represent data as a mixture of underlying distributions. These underlying distributions are combined to form a composite distribution that can capture complex patterns in the data (McLachlan & Peel, 2000). Mixture models are particularly useful when the population of the data contains multiple subpopulations with distinct characteristics. For example, Island Health's PSU data may contain a patient group that accesses consultation services sporadically, while another group accesses emergency services very frequently.

Mixture models can help model the overall population's service utilization more accurately, which in turn can help healthcare providers identify patient subgroups that would benefit from tailored treatment strategies. Another advantage of mixture models is that they can handle overlapping distributions, where some data points may belong to multiple subpopulations simultaneously. This flexibility makes mixture models a powerful tool for analyzing complex datasets with multiple underlying patterns, as can be the case in healthcare data. Figure 3.1 shows an illustrative dataset that can be modeled more accurately using a Mixture Model with two Gaussian components, rather than a single Gaussian distribution.

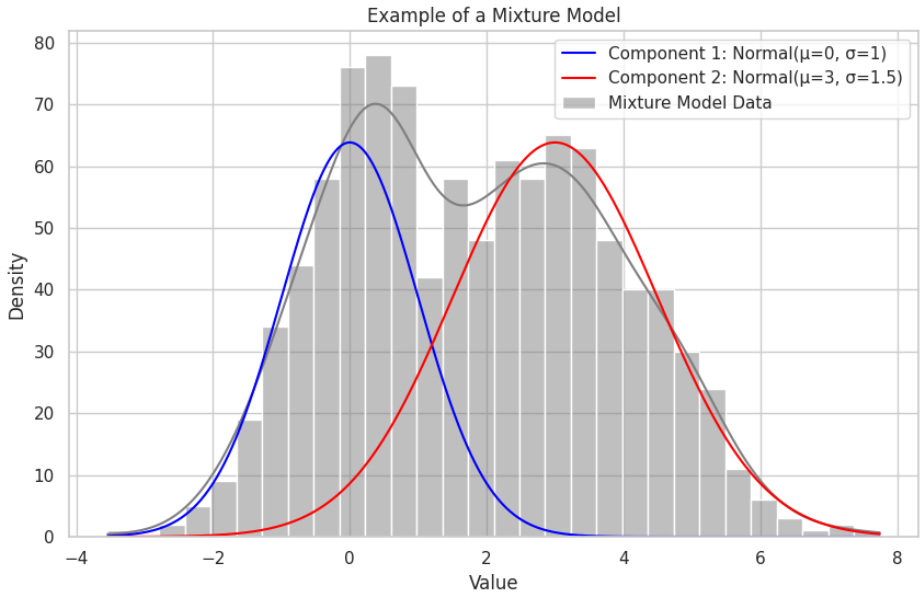

Figure 3.1: **Illustration of a Mixture Model with Two Gaussian Components.** The blue and red curves represent the individual components with parameters $\mu = 0$, $\sigma = 1$ and $\mu = 3$, $\sigma = 1.5$, respectively. The gray area and line indicate the combined dataset and the overall density estimate of the mixture model.

3.3 Dependent Mixture Models

Dependent mixture models are statistical models that represent data using a mixture of underlying distributions, in which the mixture components depend on unobserved variables. This dependency structure allows the model to capture more complex relationships and patterns in the data, compared to traditional mixture models where components are independent of factors such as time. We will follow the description of a dependent mixture model from (Lijoi et al., 2014).

A dependent mixture model's emission probability distribution, b, is a mixture model. In the context of patient service utilization (PSU), that mixture model is time-dependent. At each time point, patients are assigned to mixture components. These assignments can change over time. This dependency structure enables the model to capture the joint distribution of the observed data and the latent variables, providing a more nuanced representation of the underlying data generating process (Lijoi et al., 2014).

Dependent mixture models can capture the influence of multiple latent factors (for example, health status) on the observed service utilization patterns of patients. For a single patient, the joint distribution of the observed data $\mathbf{O}$ and the latent variables $\mathbf{S}$ in a dependent mixture model is expressed in Lijoi et al. (2014) as:

$$\Pr(\mathbf{O}_{1:T}, \mathbf{S}_{1:T}|\theta) = p_0(S_1)\mathbf{b}(\mathbf{O}_1|S_1) \prod_{t=1}^{T-1} p(S_{t+1}|S_t)\mathbf{b}(\mathbf{O}_{t+1}|S_{t+1}). \tag{3.1}$$

The elements of this equation are as follows:

- $p_0(S_1)$ is the vector of initial latent state probabilities.
- $p(S_{t+1}|S_t)$ is the transition probability from latent state S_t to latent state S_{t+1}.
- $\mathbf{b}(\mathbf{O}_t|S_t)$ is the emission probability distribution of the observed data at time t given the latent state at time t, i.e. $\mathbf{b}(\mathbf{O}_t|S_t) = \Pr(O_t|S_t)$.
- θ is the vector of model parameters, including the initial state probabilities p_0, transition probabilities p, and emission probabilities b.

If we consider the observed and latent data to be independently distributed across all N patients, the joint distribution across all patients and all time points can be expressed as:

$$P(\mathbf{O}_{1:T}^{(1:N)}, \mathbf{S}_{1:T}^{(1:N)}|\theta) = \prod_{n=1}^{N} P(\mathbf{O}_{1:T}^{(n)}, \mathbf{S}_{1:T}^{(n)}|\theta), \tag{3.2}$$

where $\mathbf{O}^{(n)}$ and $\mathbf{S}^{(n)}$ represent the observed and latent data for patient n, respectively.

The model parameters θ for a Dependent Mixture Model can be learned from the observed data using probabilistic parameter estimation methods. In the next section, we describe two parameter estimation methods that are used in our work: the Expectation-Maximization (EM) algorithm and Maximum *a Posteriori* (MAP) estimation.

3.4 Probabilistic Parameter Estimation

Probabilistic parameter estimation is a fundamental task in statistical modeling, where the goal is to infer the parameters of a probabilistic model from observed data. This process involves finding the values of the model parameters that best explain the observed data, typically by maximizing the likelihood of the data given the model. In the context of Hidden Markov Models and Mixture Models, probabilistic parameter estimation is crucial for learning the initial state probabilities, transition probabilities, and emission probabilities that best describe the data.

3.4.1 Expectation-Maximization Algorithm

The Expectation-Maximization (EM) algorithm is a statistical method used for finding maximum likelihood estimates of parameters in probabilistic models, particularly those with latent variables. The EM algorithm iterates between two steps:

1. the Expectation step, where it calculates the expected value of the log-likelihood concerning the latent variables S, and

2. the Maximization step, where it maximizes this expected log-likelihood to update the model parameters $p_0, p,$ and b.

This process is repeated until convergence, leading to a local maximum of the likelihood function. The EM algorithm is particularly useful in the context of mixture models, where it efficiently handles the complexities of estimating multiple overlapping distributions. The seminal work on the EM algorithm in Dempster et al. (1977) provides a comprehensive foundation for understanding and applying this method.

3.4.2 Maximum *a Posteriori* Estimation

Maximum *a Posteriori* (MAP) estimation is a Bayesian approach to parameter estimation that incorporates prior knowledge about the parameters into the estimation process. In MAP estimation, the goal is to find the parameter values that maximize the posterior probability of the parameters given the observed data. This is achieved by combining the likelihood of the data with the prior distribution of the parameters, resulting in a more robust estimation procedure. The choice of prior distribution can significantly impact the final estimates, particularly in cases where the data is limited or noisy.

Formally, a MAP estimate is defined as:

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} \Pr(\theta|O) = \arg\max_{\theta} \Pr(O|\theta)\Pr(\theta). \tag{3.3}$$

Here, θ represents the parameters of the model, O is the observed data, $\Pr(O|\theta)$ is the likelihood of the data given the parameters, and $\Pr(\theta)$ is the prior distribution over the parameters.

MAP estimation is useful for incorporating domain knowledge or constraints into the parameter estimation process, leading to more interpretable and robust models. The MAP estimates of the parameters in Hidden Markov Models can be computed using the forward-backward algorithm, which provides a framework for computing the posterior distribution over the hidden states of an HMM (Rabiner & Juang, 1986).

Rabiner and Juang (1986) defines the following elements:

1. A forward variable $\alpha_T(i) = \Pr(O_1, O_2, \ldots, O_T, S_T = i|\theta)$.

2. A backward variable $\beta_T(i) = \Pr(O_{T+1}, O_{T+2}, \ldots, O_T|S_T = i, \theta)$.

Using the forward and backward variables, the posterior distribution over the hidden states can then be computed as:

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} \sum_{i=1}^{N} \alpha_T(i)\beta_T(i). \tag{3.4}$$

The forward-backward algorithm provides the solution to the MAP estimates of the parameters in Hidden Markov Models, enabling accurate and efficient parameter estimation in complex probabilistic models.

In the next section, we introduce `depmixS4`, an R package that implements Dependent Mixture Models and relies on the EM algorithm to estimate the model parameters (Visser & Speekenbrink, 2021). We also describe how we leverage MAP estimation using the output of the DepmixS4 model.

Chapter 4

Methods & Results

To evaluate the performance of HMMM's in predicting PSU patterns, we simulate time series data for N patients, which we then divide into training and testing subsets. The last time point of each patient's time series data is used for testing, while the preceding points are used for training.

We compare two prediction approaches using our simulated datasets:

1. **DepmixS4 Model**: A HMMM is fitted to the training data using the `depmixS4` package. The trained model's estimated parameters are then used to predict the service utilization for the testing data using MAP estimation.

2. **Baseline Approach**: A baseline (or naïve) approach that does not account for latent states is used to predict the service utilization for the testing data. The predicted service utilization for a patient is the most frequently observed utilization (0 or 1) for each service class across all time points in the training data.

Each approach is evaluated based on the accuracy of its predictions over all trials, measured as the proportion of correctly predicted service utilizations in the testing data. The approaches are then compared using pairwise t-tests and the results are summarized in Section 4.3.

4.1 Simulated Dataset

We simulate a dataset of $T = 12$ time points for $N = 100$ patients. We consider a scenario with $M = 2$ service classes and $K = 2$ latent states. The dataset consists of the service utilization records for each patient at each time point, where each service class can be either utilized (1) or not utilized (0). The service utilization vectors $\mathbf{U} = (U_1, U_2)$ for each patient are generated based on the following process:

1. In the first half of time points, sample $U_1 \sim \text{Bern}(\alpha)$ and $U_2 \sim \text{Bern}(1 - \alpha)$.

2. In the second half of time points, sample $U_1 \sim \text{Bern}(1 - \alpha)$ and $U_2 \sim \text{Bern}(\alpha)$.

Here, α is a parameter we increase progressively at each simulation, from 0.1 to 0.5. This process generates service utilization patterns that switch between the two service classes at the halfway time point. Code for the dataset generation process is included in Appendix A.4. To illustrate, a sample from the simulated dataset is illustrated in Table 4.1.

The dataset is further divided into training and testing subsets, where we use the first 11 time points of each patient's data for training (U_{train} matrix) and the last time point for testing (U_{test} matrix). The training data is used to fit the prediction models, and the testing data is used to evaluate the models' performance in predicting the next time point's service utilization.

Patient ID	**Time Point**	**Used Service 1?**	**Used Service 2?**
1	1	0	1
1	2	0	1
1	3	0	1
1	4	1	1
1	5	0	1
1	6	0	1
1	7	1	0
1	8	1	0
1	9	1	0
1	10	1	0
1	11	1	0
1	12	1	0

Table 4.1: **Sample of Simulated Multivariate Time Series Data for One Patient.** The patient's data has $T = 12$ time points, $M = 2$ service classes, and $\alpha = 0.1$. Each row represents the service utilization at a specific time point for a patient. The full dataset comprises $N = 100$ patients and is a 1200×4 matrix. In this sample, the U_{train} matrix would consist of the first 11 time points, and the U_{test} matrix would consist of the 12th time point. The perfect prediction for this patient at time point 12 would be to predict $U_1 = 1$ and $U_2 = 0$. The baseline approach would predict $U_1 = 1$ and $U_2 = 1$ for this patient, as service 2 was utilized more frequently in the training data.

Note that as α approaches 0.5, the distributions of U_1 and U_2 become closer together. At $\alpha = 0.5$, the distributions are equal, and equal to Bern(0.5). For $\alpha = 0.5$, we expect both prediction methods to exhibit 50% accuracy, regardless of their sophistication.

4.2 DepmixS4: Dependent Mixture Models in R

The `depmixS4` package in R provides a framework for fitting Dependent Mixture Models to time series data (Visser & Speekenbrink, 2010). The package facilitates the estimation of initial state probabilities $\mathbf{p_0}$, transition probabilities $\mathbf{p}$, and emission probabilities $\mathbf{b}$ using the EM algorithm (Visser & Speekenbrink, 2011). The resulting models can then be used to make inferences about the hidden states of the system and to predict future observations.

Further details about the `depmixS4` package can be found in the official documentation (Visser & Speekenbrink, 2021).

We use the DepmixS4 model to fit a Dependent Mixture Model to our dataset and predict what the service utilization will be for the last time point of each patient. The parameters used to fit the model are as follows:

```
model <- depmix(formula = cbind(X1, X2, X3, ... , Xm) ~ 1,
                data = U_train,
                nstates = K,
                family = multinomial("identity"),
                ntimes = Ts_train)
```

where `X1, X2, ..., Xm` are the M binary service class utilization variables, `U_train` is the training dataset, `K` is the number of latent states, and `Ts_train` is the number of time points per patient in the training data (in our case, $Ts_{train} = 11$ for each patient). The model is then fitted to the training data using `fm <- fit(model)`.

The resulting estimated parameters can be extracted from the fitted model `fm` using the `getpars()` function. The last state of each patient's time series data can then be predicted using the $\mathbf{b_{est}}$ and $\mathbf{p_{est}}$ that were estimated by the model. First, the predicted latent state in the patient's last time point T_n is assigned as the state with the highest transition probability $\mathbf{p_{est}}$ at the patient's penultimate time point $T_n - 1$. Then, using the formula described in Section 3.4.2, the predicted service utilization is assigned as the MAP estimate of the emission probabilities $\mathbf{b_{est}}$ for the predicted latent state.

Code for the model fitting and prediction process is included in Appendix A.4.

4.3 DepmixS4 Model Evaluation

We measure the prediction accuracy of the DepmixS4 model and the baseline approach for our simulated dataset. The prediction accuracy is calculated as the proportion of correctly predicted service utilizations in the testing data. It can be written as:

$$\text{Accuracy} = \sum_{m=1}^{M} \frac{\mathbf{1}(\hat{U}_m = U_m)}{M}, \tag{4.1}$$

where $\hat{U}_m$ is the predicted service utilization for service class m, U_m is the true service utilization for service class m, and M is the total number of service classes.

The results summarized in Figure 4.1 indicate that the DepmixS4 model outperforms the baseline approach significantly for $\alpha = 0.1, 0.2, 0.3$, which suggests that the model is effective in predicting service utilization patterns when there is a clear switch in utilization patterns at the halfway point. We conduct pairwise t-tests to compare the prediction accuracy of the DepmixS4 model and the baseline approach for each value of α. The results of the t-tests

are summarized in Table 4.2. The t-tests indicate that the DepmixS4 model significantly outperforms the baseline approach for $\alpha = 0.1, 0.2, 0.3$ ($p < 0.05$ after Bonferroni correction for five tests). However, the difference in performance is not significant for $\alpha = 0.4$ and 0.5 (respective raw p-values are $p = 0.9004$ and $p = 0.3293$).

α	**t-value**	**df**	**p-value**	**Confidence Interval**	**Mean Difference**
0.1	-85.967	99	8.04e-95	[-0.8198, -0.7828]	-0.8013
0.2	-34.947	99	1.70e-57	[-0.5997, -0.5353]	-0.5675
0.3	-12.046	99	4.10e-21	[-0.2821, -0.2023]	-0.2422
0.4	0.12553	99	9.00e-1	[-0.01481, 0.01681]	0.001
0.5	-0.98039	99	3.29e-1	[-0.00907, 0.00307]	-0.003

Table 4.2: **Summary of Paired t-test Results for Varying Values of α.** The table summarizes the results of the paired t-tests comparing the prediction accuracy of the DepmixS4 model and the baseline approach for each value of α. The t-tests indicate that the DepmixS4 model significantly outperforms the baseline approach for $\alpha = 0.1, 0.2, 0.3$ (Bonferroni corrected p-values < 0.05, raw p-values are 8.04e-95, 1.70e-57, and 4.10e-21 respectively.)

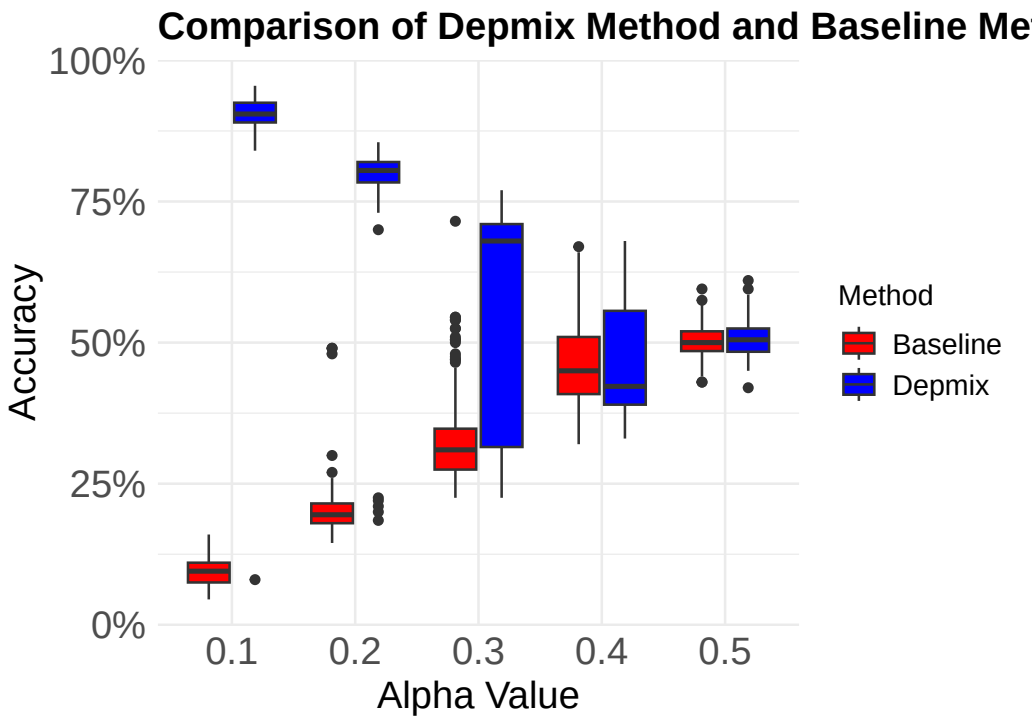

Figure 4.1: **DepmixS4 Model and Baseline Approach Prediction Accuracy.** The plot illustrates the prediction accuracy of the DepmixS4 model and the baseline approach. The DepmixS4 model outperforms the baseline approach by a significant margin for the smaller values of α.

The DepmixS4 model's ability to capture the underlying structure of the data allows it to make more accurate predictions compared to a naive approach that does not account

for latent states. However, the model's performance decreases as α approaches 0.5, due to the increased randomness in the data. The baseline approach performs similarly to the DepmixS4 model for $\alpha = 0.5$, as the data becomes more unpredictable and the model's latent state representation becomes less informative (for $\alpha = 0.5$, the emissions are independent draws from $\text{Ber}(p = 0.5)$). Code for the prediction evaluation process and t-tests is included in Appendix A.4.

In the next section, we describe how VIHA's PSU data can be preprocessed in order to fit a DepmixS4 model.

4.4 Preprocessing Healthcare Data

The VIHA dataset consists of records of interactions between patients and healthcare services. Table 4.3 illustrates a fictional sample of the raw dataset, where each row represents the history of service utilizations for a patient. The service numbers correspond to specific healthcare services provided by VIHA, such as mental health counseling, addiction treatment, or crisis intervention. We refer to this raw healthcare data as D, a sequence of N sets of triplets, where the triplet $(\text{service number}, \text{start time}, \text{end time})$ represents a service access for a patient. The n-th element of this sequence corresponds to the n-th patient.

We preprocess the dataset as follows:

Data: Raw patient data D, list of service classes of interest *target_classes*, and time unit of interest (days, weeks, months, or years) *time_unit*.
Result: Matrix $U_{T \times N}$ of service utilization per time unit for each patient.

for *each patient i in D* **do**
  Get patient's first service access time t_{start} and last service access time t_{end};
  Divide the time range $[t_{\text{start}}, t_{\text{end}}]$ into sequential periods of length *time_unit*;
  Initialize matrix $U_{T \times N}$ for patient i with all entries set to zero;
  for *each time bin in the period* **do**
    for *each service accessed by the patient in the time bin* **do**
      Find the column index for the service class in U;
      Increment the count of accesses in the corresponding cell of matrix U;
    end
  end
end

Algorithm 3: Extracting the Service Utilization Matrix from Raw Patient Data. This algorithm computes a structured matrix of service utilization counts aligned to predefined time units.

Note that in Algorithm 3, each element of the matrix U is initialized to zero. This initialization acts as a placeholder for service classes for which no data is available during the specified time units. As we process each entry corresponding to service utilizations, the respective cell in the matrix is incremented based on the number of times a service class is accessed within the time frame. If a particular service class does not have any recorded utilization for a given day, the cell retains its initial value of zero. This approach effectively pads the matrix on the right with zeros, ensuring that each row (patient) and column (service class) pair maintains a uniform format regardless of variations between patients.

Moreover, the alignment of time data ensures that each count within the matrix U corresponds precisely to the defined temporal boundaries of a day. Specifically, for each patient and each service class, the start of the period is aligned to 00:00 of the day, and the end of the period is aligned to 23:59. This consistent boundary definition allows for accurate and comparable aggregations of service utilizations across different days.

The resulting matrix U is a matrix of counts of service utilizations for each patient at each time unit, where each row represents a patient and each column represents a service class. Table 4.4 shows the U matrix for Patient C when the *time_unit* parameter is set to *days* and the *target_classes* parameter is $[23, 35]$. Code for the preprocessing algorithm is included in Appendix A.3.

patient id	**age**	**sex**	**service**	**start**	**end**
Patient A	64	Female	9	2021-01-03 12:00:00	2021-01-03 15:00:00
Patient A	64	Female	57	2021-01-03 15:00:00	2021-01-03 16:00:00
Patient B	22	Female	31	2021-01-03 15:00:00	2021-01-03 18:00:00
Patient C	50	Male	23	2021-01-02 12:00:00	2021-01-02 15:00:00
Patient C	50	Male	35	2021-01-05 08:00:00	2021-01-05 10:00:00
Patient C	50	Male	23	2021-01-07 07:00:00	2021-01-07 09:00:00

Table 4.3: **Fictional sample of raw patient service utilization data from the VIHA dataset.** Each row represents the service utilization history for a patient. The services column lists the services accessed by the patient, and the start and end columns indicate the time intervals during which the services were accessed. In this example, Patient A enters service 9 at noon on January 3rd, and exits service 9 at 3PM on the same day. Then, Patient A enters service 57 at 3PM and exits at 4PM. The element of D corresponding to Patient A is thus: { (9, 2021-01-03 12PM, 2021-01-03 3PM), (57, 2021-01-03 3PM, 2021-01-03 4PM) }.

The resulting U matrix can be used as input to the `depmixS4` package to fit a Dependent Mixture Model to the patient service utilization data. The latent states inferred by the model can then be used to predict future service utilizations and evaluate different intervention strategies, as described in the following sections.

time bin	service 23	service 35
2021-01-02	1	0
2021-01-03	0	0
2021-01-04	0	0
2021-01-05	0	1
2021-01-06	0	0
2021-01-07	1	0

Table 4.4: U **matrix for Patient C with** $time\_unit = days$ **and** $target\_classes = [23, 35]$**.** Each row represents the service utilization for a specific time bin (day) for the patient, and each column represents the number of times the corresponding service was accessed.

4.5 Counterfactual Reinforcement Learning

In this section, we describe how the latent states extracted from the DepmixS4 model can be used in a Counterfactual Reinforcement Learning (CFRL) framework to evaluate different intervention strategies, after training the DepmixS4 model on the preprocessed U matrix. We define the key components of the CFRL framework as follows:

- **States (S^*)**: A low-dimensional representation of a patient's time series, using the latent states estimated by the DepmixS4 model, which summarizes the patient's service utilization patterns over time.
  - For example, if there are $K = 2$ latent states, which could be respectively interpreted as 'healthy' and 'unhealthy' states, a patient's state S^* could be represented as a vector of the proportion of time spent in each state.
  - Note the states must be time-invariant and summarize the entire time series if they are to be used in the CFRL framework.
- **Actions (a)**: Binary variables representing interventions.
  - For example, $a = 1$ indicates that a specific intervention was made (e.g., a crisis intervention team is deployed), and $a = 0$ indicates that no intervention was made.
  - Another definition for actions could be $a = 1$ if any intervention was made at any time point within the period, and $a = 0$ otherwise.
- **Rewards (r)**: The rewards associated with each time step, which can be defined based on specific criteria.
  - For example $r = 1$ if there was no mortality within the entire period for the patient, and $r = 0$ if there was at least one death within the period.

  - Another example could be $r = 1$ if the patient recovers, defined by a specific criteria for recovery, and $r = 0$ if the patient does not recover.

- **Policy (π_0)**: The current policy used by the Vancouver Island Health Authority (VIHA) maps a state S^* to an action a, defining the intervention strategy based on the patient's state feature vector. Since this policy is only formally defined in terms of observed symbols and not in terms of latent states or state feature vectors, we can approximate it using classification methods as an additional preprocessing step. For example, we can train a logistic regression model to predict the action based on the latent states, which is effectively an estimate of $\pi_0(a|s^*)$.

In the next section, we describe an algorithm for implementing Counterfactual Reinforcement Learning using the latent states extracted from the DepmixS4 model.

4.5.1 Pipeline Algorithm

Data: Raw PSU data D, number of latent states K
Result: Counterfactual estimates $\hat{R}_{\text{IPS}}(a)$ of the reward of an offline policy $\pi(a|s^*)$

Define actions a (e.g., intervention or no intervention);
Define rewards r (e.g., no overdose/mortality = 1, at least one overdose/mortality = 0);

Step 1: Preprocessing
Generate the U matrix from D using Algorithm 3 and divide into training and testing datasets.;

Step 2: Extracting Latent States
Fit the DepmixS4 model to training data and extract latent states S for each patient;
Define S^*, a low-dimensional representation of the latent states S that summarizes a patient's service utilization patterns over time;

Step 3: Compute Counterfactual Outcomes
Train a classification model (e.g., logistic regression) on U_{train} to estimate $\pi_0(a|s^*)$, the current policy for enacting an intervention given the state feature vector;
for *each patient* i **do**
    Compute the counterfactual outcome using the Inverse Propensity Scoring (IPS) estimator: $\hat{R}_{\text{IPS}}(a) = \frac{1}{N}\sum_{i=1}^{N} \frac{\pi(a_i|s_i^*)}{\pi_0(a_i|s_i^*)} \cdot r_i$;
end

Algorithm 4: Counterfactual Reinforcement Learning with DepmixS4 Model Latent States. The algorithm outlines our proposed pipeline for integrating the DepmixS4 model's latent states into a Counterfactual Reinforcement Learning framework. The algorithm computes the estimated reward of a policy $\pi(a|s^*)$ using Inverse Propensity Scoring.

The algorithm's final output, $\hat{R}_{IPS}(a)$, provides an estimate of the reward of an offline policy $\pi(a|s^*)$ based on the latent states extracted from the DepmixS4 model. This estimate can be used to evaluate different intervention strategies and inform decision-making processes at VIHA.

Some candidate features for the state S^* that could be used in this framework include:

- The last element of the DepmixS4 timeseries latent state, S_T, which represents the most likely state for a patient at the last time point T.

- The most common state in the last month of the DepmixS4 timeseries latent states.
- Number of days elapsed since last terminal state (e.g., opioid overdose).
- Proportion of days spent in a latent state of interest (e.g., 'healthy') over the last year.
- Number of transitions between latent states in the last year.

For example, the state S^* could be represented as a vector of the proportion of time spent in each latent state over the last year, or as a vector of the most common latent state in each of the last 6 months.

Chapter 5

Conclusion

In this thesis, we explored Q-Learning as a reinforcement learning algorithm to optimize path planning for a robot in a grid world environment. We implemented the Q-Learning algorithm in Python and demonstrated its effectiveness in learning an optimal policy for the robot to reach a goal while avoiding obstacles. To partially overcome the limitations of Q-Learning stemming from the need for online interaction, we explored the Dyna-Q algorithm, which incorporates an offline model of the environment to accelerate learning without requiring additional interactions with the environment. We implemented the Dyna-Q algorithm in Python and showed that it learns an optimal policy at a significantly faster rate than Q-Learning. In our experiment, Q-Learning required approximately 75 episodes to achieve the performance that Dyna-Q achieved by episode 25. Then, we introduced a third reinforcement learning technique, Counterfactual Reinforcement Learning, as a fully offline approach to evaluating policies using longitudinal data.

We discussed the application of Hidden Markov Models and Mixture Models to uncover hidden patterns in longitudinal patient data, and we demonstrated the effectiveness of the `depmixS4` package in extracting latent states from patient service utilization data when the data presented strong signals. We showed that the DepmixS4 model outperforms the baseline model in predicting future service utilization when the data exhibits clear patterns (for $\alpha = 0.1, 0.2, 0.3$, Bonferroni corrected p-values $p < 0.05$, where the emissions in the simulation are sampled from $\text{Ber}(\alpha)$.) However, both the DepmixS4 model and the baseline approach performed similarly when the data was less predictable ($\alpha = 0.4, 0.5$, raw p-values $p = 0.9004, 0.3293$ respectively).

Finally, we proposed a Counterfactual Reinforcement Learning pipeline that leverages the latent states extracted from the DepmixS4 model to evaluate different intervention strategies without requiring interaction with the environment. We outlined an algorithm for implementing CFRL using Inverse Propensity Scoring to estimate the reward of an offline policy using domain knowledge and historical data. The methods and results presented in this thesis provide a foundation for pipeline development with counterfactual reinforcement learning applied to patient time series data.

References

Brown, M. (2016). *Elon Musk Says Google DeepMind's Go Victory Is a '10-Year' Jump for A.I.* Retrieved from https://www.inverse.com/article/12620-elon-musk-says-google-deepmind-s-go-victory-is-a-10-year-jump-for-a-i (Accessed: 2024-02-19)

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, *39*(1).

Frozen Lake - Gymnasium Documentation. (2023). https://gymnasium.farama.org/environments/toy_text/frozen_lake/. Farama Foundation. (Accessed: 2024-05-16)

Gajcin, J., & Ivana, D. (2022). Counterfactual Explanations for Reinforcement Learning. *arXiv preprint arXiv:2210.11846*.

Government of British Columbia. (2023). *Regional health authorities.* https://www2.gov.bc.ca/gov/content/health/about-bc-s-health-care-system/partners/health-authorities/regional-health-authorities. (Accessed on: December 13th, 2023)

Joachims, T., & Swaminathan, A. (2016). Counterfactual Evaluation and Learning for Search, Recommendation and Ad Placement. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval* (p. 1199–1201). New York, NY, USA: Association for Computing Machinery.

Lijoi, A., Nipoti, B., & Prünster, I. (2014). Dependent mixture models: Clustering and borrowing information. *Computational Statistics and Data Analysis*, *71*, 417-433. (Accessed: 2024-03-21)

McLachlan, G., & Peel, D. (2000). *Finite Mixture Models.* Wiley.

Moyer, C. (2016). How Google's AlphaGo Beat a Go World Champion. *The Atlantic.* Retrieved from https://www.theatlantic.com/technology/archive/2016/03/the-invisible-opponent/475611/ (Accessed: December 13th, 2023)

Murphy, S. A. (2003, 04). Optimal Dynamic Treatment Regimes. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, *65*(2), 331-355.

Rabiner, L., & Juang, B. (1986). An introduction to Hidden Markov Models. *IEEE ASSP Magazine*, *3*(1), 4-16.

Sutton, R. S. (1990). Integrated Architectures for Learning, Planning, and Reacting based on Approximating Dynamic Programming. In *Proceedings of the Seventh International Conference on Machine Learning* (pp. 216–224).

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

Vancouver Island Health Authority. (2023a). *About Us.* https://www.islandhealth.ca/about-us. (Accessed on: December 13th, 2023)

Vancouver Island Health Authority. (2023b). *Research & Capacity Building.* https://www.islandhealth.ca/research-capacity-building. (Accessed: December 13th, 2023)

Visser, I., & Speekenbrink, M. (2010). depmixS4: An R Package for Hidden Markov Models. *Journal of Statistical Software*, *36*(7), 1–21.

Visser, I., & Speekenbrink, M. (2011). depmixS4: An R Package for Hidden Markov Models [Computer software manual]. (R package version 1.5-0)

Visser, I., & Speekenbrink, M. (2021). Dependent Mixture Models - Hidden Markov Models of GLMs and Other Distributions in S4 [Computer software manual]. Retrieved from https://cran.r-project.org/web/packages/depmixS4/depmixS4.pdf (Accessed: 2024-02-19)

Watkins, C. J., & Dayan, P. (1992). Q-Learning. *Machine Learning*, *8*(3-4), 279–292.

Wedd, N. (2018). *A Brief History of Computer Go.* Retrieved from https://www.computer-go.info/h-c/index.html (Accessed: 2024-02-19)

Wessling, B. (2023). Spot goes to work on the decommissioning team at Fukushima plant. *The Robot Report*. Retrieved from https://www.therobotreport.com/spot-goes-to-work-on-the-decommissioning-team-at-fukushima-plant/ (Accessed: December 13th, 2023)

Appendix A

Code

A.1 AdventureBot Q-Learning Algorithm

```
import numpy as np
import matplotlib.pyplot as plt

# Set seed
np.random.seed(0)

# Constants
EMOJI_TREASURE = "T"
EMOJI_OBSTACLE = "X"
EMOJI_ADVENTUREBOT = "B"
EMOJI_EMPTY = "_"
ACTIONS = ["up", "down", "left", "right"]


class QLearning:
    def __init__(
        self, grid_size, max_steps, alpha, gamma, epsilon, episodes, debug=
    False
    ):
        """
        Initialize the Q-Learning agent.

        Parameters:
        - grid_size: The size of the grid world.
        - max_steps: The maximum number of steps per episode.
        - alpha: The learning rate.
        - gamma: The discount factor.
        - epsilon: The exploration rate.
        - episodes: The number of episodes to run.
        - debug: Enable debugging mode (True/False).
        """
        self.grid_size = grid_size
        self.max_steps = max_steps
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.episodes = episodes
```

```
        self.debug = debug

        # Q-table
        self.q_table = np.random.uniform(
            low=-1, high=1, size=(grid_size, grid_size, len(ACTIONS))
        )

        # Grid world
        self.state = np.full((grid_size, grid_size), EMOJI_EMPTY)
        self.adventurebot_position = self.randomize_position(
            EMOJI_ADVENTUREBOT, init=True
        )
        self.treasure_position = self.randomize_position(EMOJI_TREASURE, init=True)
        self.obstacle_position = self.randomize_position(EMOJI_OBSTACLE, init=True)

        # Counters
        self.step_count = 0
        self.reached_treasure = 0
        self.reached_obstacle = 0
        self.reached_max_steps = 0
        self.steps_per_episode = []

    def randomize_position(self, emoji, init=False):
        """
        Randomly place an emoji on an empty position in the grid.

        Parameters:
        - emoji: The emoji to place.
        - init: Whether it's the initial placement (True/False).

        Returns:
        - position: The position where the emoji is placed.
        """
        if init:
            while True:
                position = np.random.randint(0, self.grid_size, size=2)
                if self.state[tuple(position)] == EMOJI_EMPTY:
                    self.state[tuple(position)] = emoji
                    print(
                        f"Placed {emoji} for the first time at position {position.tolist()}"
                    ) if self.debug else None
                    return position.tolist()

        if emoji == EMOJI_ADVENTUREBOT:
            self.state[tuple(self.adventurebot_position)] = EMOJI_EMPTY
            self.state[tuple(self.treasure_position)] = EMOJI_TREASURE
            self.state[tuple(self.obstacle_position)] = EMOJI_OBSTACLE

            while True:
                position = np.random.randint(0, self.grid_size, size=2)
                if self.state[tuple(position)] == EMOJI_EMPTY:
                    self.state[tuple(position)] = emoji
                    print(
                        f"Placed {emoji} back at position {position.tolist()}"
```

```
                    ) if self.debug else None
                    return position.tolist()

    def print_grid_with_bot_position(self):
        """
        Print the grid world with the AdventureBot's position.
        """
        self.state = np.full((self.grid_size, self.grid_size), EMOJI_EMPTY)
        self.state[tuple(self.treasure_position)] = EMOJI_TREASURE
        self.state[tuple(self.obstacle_position)] = EMOJI_OBSTACLE
        self.state[tuple(self.adventurebot_position)] = EMOJI_ADVENTUREBOT

        for row in self.state:
            print(" ".join(row))

    def get_valid_actions(self):
        """
        Get valid actions for the AdventureBot at a given position.

        Returns:
        - valid_actions: A list of valid actions.
        """
        y, x = self.adventurebot_position
        valid_actions = []

        if y > 0:
            valid_actions.append("up")
        if y < self.grid_size - 1:
            valid_actions.append("down")
        if x > 0:
            valid_actions.append("left")
        if x < self.grid_size - 1:
            valid_actions.append("right")

        return valid_actions

    def choose_action(self):
        """
        Choose an action for the AdventureBot.

        Returns:
        - chosen_action: The chosen action.
        """
        valid_actions = self.get_valid_actions()
        if np.random.uniform(0, 1) < self.epsilon:
            return np.random.choice(valid_actions)
        else:
            q_values = self.q_table[
                self.adventurebot_position[0], self.adventurebot_position
[1], :
            ]
            max_q_value = q_values.max()
            max_actions = [
                ACTIONS[i] for i, q_val in enumerate(q_values) if q_val ==
max_q_value
            ]
            return np.random.choice(max_actions)

```

```
    def take_action(self, action):
        """
        Take an action and update the AdventureBot's position.

        Parameters:
        - action: The chosen action.

        Returns:
        - new_position: The new position after taking the action.
        - reward: The received reward after taking the action.
        """
        new_position = self.adventurebot_position.copy()
        y, x = new_position

        if action == "up" and y > 0:
            new_position = [y - 1, x]
        elif action == "down" and y < self.grid_size - 1:
            new_position = [y + 1, x]
        elif action == "left" and x > 0:
            new_position = [y, x - 1]
        elif action == "right" and x < self.grid_size - 1:
            new_position = [y, x + 1]

        self.state[
            self.adventurebot_position[0], self.adventurebot_position[1]
        ] = EMOJI_EMPTY
        self.state[new_position[0], new_position[1]] = EMOJI_ADVENTUREBOT

        reward = 0
        if new_position == self.treasure_position:
            reward = 1
        elif new_position == self.obstacle_position:
            reward = -1

        return new_position, reward

    def update_q_value(self, current_state, action, reward, new_state):
        """
        Update the Q-value based on the Q-Learning update rule.

        Parameters:
        - current_state: The previous state.
        - action: The chosen action.
        - reward: The received reward after taking the action.
        - new_state: The new state after taking the action.
        """
        prev_q_value = self.q_table[
            current_state[0], current_state[1], ACTIONS.index(action)
        ]
        future_reward = self.gamma * self.q_table[new_state[0], new_state
    [1], :].max()
        self.q_table[
            current_state[0], current_state[1], ACTIONS.index(action)
        ] += self.alpha * (reward + future_reward - prev_q_value)

    def run_episode(self):
        """
        Run a single episode of the Q-Learning agent.
```

```
204         """
205         self.step_count = 0
206         self.adventurebot_position = self.randomize_position(
    EMOJI_ADVENTUREBOT)
207         self.print_grid_with_bot_position() if self.debug else None
208 
209         while (
210             self.step_count < self.max_steps
211             and not self.adventurebot_position == self.treasure_position
212             and not self.adventurebot_position == self.obstacle_position
213         ):
214             self.perform_step()
215 
216         # Record the reason for ending the episode
217         if self.step_count == self.max_steps - 1:
218             self.reached_max_steps += 1
219         elif self.adventurebot_position == self.treasure_position:
220             self.reached_treasure += 1
221             print(f"Reached treasure! End of episode.") if self.debug else
    None
222         elif self.adventurebot_position == self.obstacle_position:
223             self.reached_obstacle += 1
224             print(
225                 f"Encountered obstacle! End of episode.") if self.debug else
     None
226 
227         self.steps_per_episode.append(self.step_count)
228 
229     def perform_step(self):
230         """
231         Choose an action and take it (one step).
232         """
233         self.step_count += 1
234         print(f"=====> Step {self.step_count}") if self.debug else None
235 
236         # Choose an action based on the current state
237         current_state = self.adventurebot_position
238         self.chosen_action = self.choose_action()
239 
240         # Take the chosen action and update the Q-value
241         new_state, reward = self.take_action(self.chosen_action)
242         self.adventurebot_position = new_state
243         self.update_q_value(current_state, self.chosen_action, reward,
    new_state)
244 
245         if self.debug:
246             print(
247                 f"Went **{self.chosen_action}** at state {current_state},
    grid is now:"
248             )
249             self.print_grid_with_bot_position()
250             print("\n")
251 
252     def train(self):
253         """
254         Train the Q-Learning agent over a specified number of episodes.
255         """
256         for episode in range(self.episodes):
```

```
            print(f"=========> Episode {episode}") if self.debug else None
            self.run_episode()

if __name__ == "__main__":
    # Initialize the Q-Learning agent
    q_learning = QLearning(
        grid_size=7,
        max_steps=100,
        alpha=0.2,
        gamma=0.9,
        epsilon=0.1,
        episodes=1000,
        # debug=True, # Uncomment to enable print statements
    )

    # Train the Q-Learning agent
    q_learning.train()

    # Print the Q-table
    print("\n")
    print("Final Q-table:")
    print(q_learning.q_table)

    # Print some statistics
    print("\n")
    print(f"Reached treasure {q_learning.reached_treasure} times.")
    print(f"Reached obstacle {q_learning.reached_obstacle} times.")
    print(f"Reached max steps {q_learning.reached_max_steps} times.")
    print(
        f"Average number of steps per episode: {np.mean(q_learning.
    steps_per_episode)}"
    )

    # Save a plot of the number of steps per episode
    plt.plot(q_learning.steps_per_episode)
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.xlabel("Episode")
    plt.ylabel("Number of steps")
    plt.title("Number of steps per episode for Q-Learning")
    plt.savefig("plots/q_learning_steps_per_episode.png")
```

A.2 AdventureBot Dyna-Q Algorithm

```
import random
import matplotlib.pyplot as plt
import numpy as np
from playground.q_learning import QLearning, EMOJI_EMPTY

# Set seed
np.random.seed(0)

EMOJI_GHOST = "P"


class DynaQ(QLearning):
    def __init__(
        self,
```

```
        grid_size,
        max_steps,
        alpha,
        gamma,
        epsilon,
        episodes,
        planning_steps=5,
        debug=False,
    ):
        """
        Initialize the DynaQ agent.

        Parameters:
        - grid_size: The size of the grid world.
        - max_steps: The maximum number of steps per episode.
        - alpha: The learning rate.
        - gamma: The discount factor.
        - epsilon: The exploration rate.
        - episodes: The number of episodes to run.
        - planning_steps: The number of planning steps to perform.
        - debug: Enable debugging mode (True/False).
        """
        super().__init__(grid_size, max_steps, alpha, gamma, epsilon,
    episodes, debug)
        self.planning_steps = planning_steps
        # Model as a dictionary
        self.model = {}

    def update_model(self, current_state, action, next_state, reward):
        """
        Update the model with the observed transition.

        Parameters:
        - current_state: The current state.
        - action: The chosen action.
        - next_state: The next state.
        - reward: The received reward.
        """
        self.model[(tuple(current_state), action)] = (next_state, reward)

    def print_grid_with_ghost(self, ghost_position):
        """
        Generate a grid with a ghost to visualize the Dyna-Q planning steps.

        Parameters:
        - ghost_position: The position of the ghost (Dyna-Q agent).

        Returns:
        - result: A string representation of the grid.
        """
        temp_state = np.copy(self.state)
        temp_state[tuple(self.adventurebot_position)] = EMOJI_EMPTY
        temp_state[tuple(ghost_position)] = EMOJI_GHOST

        result = ""
        for row in temp_state:
            result += " ".join(row) + "\n"

```

```
        return result

    def planning(self):
        """
        Perform planning using the Dyna-Q algorithm.

        During planning, the agent replays and learns from past experiences stored in a temporary model M.

        The planning steps are as follows:
        1. Randomly choose a state-action pair from the model.
        2. Retrieve the next state and reward from the model.
        3. Update the Q-value based on the model information.
        """
        if self.debug:
            print("Starting planning steps...")

        for i in range(self.planning_steps):
            print(f"Planning step {i+1}/{self.planning_steps}") if self.debug else None

            if self.model:
                # Randomly choose a state-action pair from the Dyna-Q planning model
                state, action = random.choice(list(self.model.keys()))

                # Retrieve the next state and reward from the model
                next_state, reward = self.model[(state, action)]

                # Update the Q-value based on the model information
                self.update_q_value(state, action, reward, next_state)

                print(self.print_grid_with_ghost(next_state)) if self.debug else None

    def perform_step(self):
        """
        Execute a step using the Dyna-Q agent's logic.

        Parameters:
        - step: The current step.

        Returns:
        - step_result: The result of the step execution.
        """
        # Execute the step using the base QLearning logic
        step_result = super().perform_step()

        # Record the state and action before the base step logic execution
        prev_state = self.adventurebot_position.copy()
        action = self.chosen_action

        # Update the Dyna-Q planning model and perform planning
        new_state, reward = self.take_action(action)
        self.update_model(prev_state, action, new_state, reward)
        self.planning()

        return step_result
```

```

if __name__ == "__main__":
    # Initialize the Dyna-Q agent
    dyna_q = DynaQ(
        grid_size=7,
        max_steps=100,
        alpha=0.2,
        gamma=0.9,
        epsilon=0.1,
        episodes=1000,
        planning_steps=5,
        # debug=True, # Uncomment to enable debugging
    )

    # Train the Dyna-Q agent
    dyna_q.train()

    # Print the Q-table
    print("\n")
    print("Final Q-table:")
    print(dyna_q.q_table)

    # Print some statistics
    print("\n")
    print(f"Reached treasure {dyna_q.reached_treasure} times.")
    print(f"Reached obstacle {dyna_q.reached_obstacle} times.")
    print(f"Reached max steps {dyna_q.reached_max_steps} times.")
    print(f"Average number of steps per episode: {np.mean(dyna_q.steps_per_episode)}")

    # Save a plot of the number of steps per episode
    plt.plot(dyna_q.steps_per_episode)
    plt.grid(True, which="both", linestyle="--", linewidth=0.5)
    plt.xlabel("Episode")
    plt.ylabel("Number of steps")
    plt.title("Number of steps per episode for Dyna-Q")
    plt.savefig("plots/dyna_steps_per_episode.png")
```

A.3 Preprocessing VIHA Data

```
from datetime import datetime, timedelta
from dateutil.parser import parse
import numpy as np
import pandas as pd


def get_start_end_of_period(timestamp, time_unit):
    """
    Calculate the start and end datetime objects for the specified time time_unit relative to the provided timestamp.

    Parameters:
    - timestamp (datetime.datetime): The reference point to calculate the time_unit from.
    - time_unit (str): The time time_unit to calculate ('day', 'week', 'month', 'year').
```

```
14
15      Returns:
16      - tuple: A pair of datetime.datetime objects representing the start and
        end of the time_unit.
17
18      Raises:
19      - ValueError: If the provided time_unit is not one of the valid options
        ('day', 'week', 'month', 'year').
20
21      Note:
22      - 'Day' is defined from 00:00 to 23:59 of the same calendar day.
23      - 'Week' is defined from Monday 00:00 to Sunday 23:59 of the same week.
24      - 'Month' is from the first day of the month 00:00 to the last day of
        the month 23:59.
25      - 'Year' spans from January 1st 00:00 to December 31st 23:59 of the same
         year.
26      """
27      if time_unit == "day":
28          start = timestamp.replace(hour=0, minute=0)
29          end = timestamp.replace(hour=23, minute=59)
30      elif time_unit == "week":
31          start = (timestamp - timedelta(days=timestamp.weekday())).replace(
32              hour=0, minute=0
33          )
34          end = start + timedelta(days=6, hours=23, minutes=59)
35      elif time_unit == "month":
36          start = timestamp.replace(day=1, hour=0, minute=0)
37          end = start.replace(month=start.month % 12 + 1, day=1) - timedelta(
        minutes=1)
38      elif time_unit == "year":
39          start = timestamp.replace(month=1, day=1, hour=0, minute=0)
40          end = start.replace(year=start.year + 1) - timedelta(minutes=1)
41      else:
42          raise ValueError("Invalid time time_unit")
43      return start, end
44
45
46  def parse_int_list(input_list):
47      """
48      Convert various representations of a collection of integers (strings,
        lists, single integers) into a uniform list of integers.
49
50      Parameters:
51      - input_list (str or list or int): Representation of integers, e.g.,
        "[1, 2, 3]", [1, 2, 3], or 1.
52
53      Returns:
54      - list: List of integers.
55
56      Examples:
57      - Input: "[1, 2, 3]"
58      - Output: [1, 2, 3]
59
60      - Input: [1, 2, 3]
61      - Output: [1, 2, 3]
62
63      - Input: 1
64      - Output: [1]
```

```
    """
    if isinstance(input_list, int):
        # Single integer input
        return [input_list]
    elif isinstance(input_list, list):
        # List input
        return [int(c) for c in input_list]
    elif isinstance(input_list, str):
        # String input
        try:
            return [int(c) for c in input_list.strip("[]").split(" ")]
        except ValueError:
            raise ValueError("String input must be a valid list of integers.")
    else:
        raise TypeError("Input must be a string, list, or integer.")


def parse_datetime_list(input_list):
    """
    Convert various representations of a collection of datetime objects (strings, lists, single datetimes) into a uniform list of datetime objects.

    Parameters:
    - input_list (str or list or datetime): Representation of datetime(s), e.g., "['2023-01-01 00:00:00', '2023-01-02 00:00:00']", [datetime(2023, 1, 1), datetime(2023, 1, 2)], or datetime(2023, 1, 1).

    Returns:
    - list: List of datetime.datetime objects.

    Examples:
    - Input: "['2023-01-01 00:00:00', '2023-01-02 00:00:00']"
    - Output: [datetime.datetime(2023, 1, 1, 0, 0), datetime.datetime(2023, 1, 2, 0, 0)]

    - Input: [datetime(2023, 1, 1), datetime(2023, 1, 2)]
    - Output: [datetime.datetime(2023, 1, 1, 0, 0), datetime.datetime(2023, 1, 2, 0, 0)]

    - Input: datetime(2023, 1, 1)
    - Output: [datetime.datetime(2023, 1, 1, 0, 0)]
    """
    if isinstance(input_list, datetime):
        # Single datetime input
        return [input_list]
    elif isinstance(input_list, list):
        # List input
        return [parse(str(dt)) for dt in input_list]
    elif isinstance(input_list, str):
        # String input
        try:
            return [parse(dt) for dt in input_list.strip("[]").split(" ")]
        except ValueError:
            raise ValueError("String input must be a valid list of datetimes.")
    else:
```

```
        raise TypeError("Input must be a string, list, or datetime.")


def get_patient_data(patient_id, df):
    """
    Retrieve a specific patient's data row from a DataFrame based on their ID.

    Parameters:
    - patient_id (str/int): Unique identifier of the patient.
    - df (pandas.DataFrame): DataFrame containing patient data with a 'patient_id' column.

    Returns:
    - pandas.Series: A single row from the DataFrame corresponding to the patient_id.

    Raises:
    - IndexError: If no data is found for the given patient_id.

    Example:
    - Input: patient_id = 123, df = [DataFrame with patient data]
    - Output: Series object containing the patient's data.
    """
    return df[df["patient_id"] == patient_id].iloc[0]


def create_access_vector(
    patient_id, timestamp, time_unit, df, target_classes, start_only=False
):
    """
    Computes a frequency vector representing a patient's usage of various service classes within a specific time time_unit.

    Parameters:
    - patient_id (str/int): Identifier of the patient whose service usage is being analyzed.
    - timestamp (datetime.datetime): The point in time from which the time_unit is calculated.
    - time_unit (str): The time time_unit for analysis ('day', 'week', 'month', 'year').
    - df (pandas.DataFrame): DataFrame containing service usage data, including the classes of services accessed.
    - target_classes (list): A list of the service classes to be analyzed.

    Returns:
    - list: A vector representing the frequency of each service class used by the patient within the specified time_unit.

    Raises:
    - ValueError: If an invalid time time_unit is specified or if the input types are incorrect.
    - IndexError: If the patient data is not found in the DataFrame.
    - TypeError: If the inputs are not of expected type.

    Example:
    - TBA
    """
```

```
    # Validate inputs
    if not isinstance(patient_id, (str, int)):
        raise TypeError("Patient ID must be a string or integer.")
    if not isinstance(timestamp, datetime):
        raise TypeError("Timestamp must be a datetime object.")
    if time_unit not in ["day", "week", "month", "year"]:
        raise ValueError(
            f"Invalid time_unit '{time_unit}'. Expected one of 'day', 'week', 'month', 'year'."
        )
    if not isinstance(target_classes, list):
        raise TypeError("Unique classes must be a list.")

    try:
        patient_data = get_patient_data(patient_id, df)
        patient_classes = parse_int_list(patient_data["srv_classes"])
        start_times = parse_datetime_list(patient_data["start_datetimes"])
        end_times = (
            parse_datetime_list(patient_data["end_datetimes"])
            if not start_only
            else start_times
        )

        start_bound, end_bound = get_start_end_of_period(timestamp, time_unit)
        results = [0] * len(target_classes)

        for service_class, start, end in zip(patient_classes, start_times, end_times):
            if (
                start <= end_bound
                and end >= start_bound
                and service_class in target_classes
            ):
                try:
                    index = target_classes.index(service_class)
                    results[index] += 1
                except ValueError:
                    raise ValueError(
                        f"Service class {service_class} not found in unique_classes."
                    )
        return results
    except IndexError:
        raise IndexError(
            "Patient data not found. Ensure the patient ID is correct and present in the DataFrame."
        )


def create_access_matrix(patient_id, time_unit, df, target_classes, start_only=False):
    """
    Generates a matrix representing a patient's usage of various service classes over a period of time. Each row corresponds to a service class and each column to one time unit (in chronological order).

    Parameters:
```

```
    - patient_id (str/int): The identifier for the patient.
    - time_unit (str): The time segment for analyzing service usage ('day', 'week', 'month', 'year').
    - df (pandas.DataFrame): DataFrame containing patient service usage data.
    - target_classes (list): A list of the service classes to be analyzed.

    Returns:
    - numpy.ndarray: A 2D array where each row corresponds to a service class and each column to a time time_unit.

    Raises:
    - ValueError: If an invalid time time_unit is specified or if the input types are incorrect.
    - IndexError: If the patient data is not found in the DataFrame.
    - TypeError: If the inputs are not of expected type.

    Example:
    - TBA
    """
    # Validate inputs
    if not isinstance(patient_id, (str, int)):
        raise TypeError("Patient ID must be a string or integer.")
    if time_unit not in ["day", "week", "month", "year"]:
        raise ValueError(
            f"Invalid time_unit '{time_unit}'. Expected one of 'day', 'week', 'month', 'year'."
        )
    if not isinstance(target_classes, list):
        raise TypeError("Unique classes must be a list.")

    try:
        patient_data = get_patient_data(patient_id, df)
        start_times = parse_datetime_list(patient_data["start_datetimes"])
        end_times = (
            parse_datetime_list(patient_data["end_datetimes"])
            if not start_only
            else start_times
        )

        earliest_start = min(start_times)
        latest_end = max(end_times)

        start_bound, _ = get_start_end_of_period(earliest_start, time_unit)
        _, end_bound = get_start_end_of_period(latest_end, time_unit)

        period_deltas = {
            "day": timedelta(days=1),
            "week": timedelta(weeks=1),
            "month": timedelta(days=31),
            "year": timedelta(days=366),
        }

        period_timestamps = []
        while start_bound < end_bound:
            period_timestamps.append(start_bound)
            start_bound += period_deltas[time_unit]

```

```
        service_usage_matrix = [
            create_access_vector(
                patient_id, timestamp, time_unit, df, target_classes,
    start_only
            )
            for timestamp in period_timestamps
        ]

        return np.array(service_usage_matrix).T
    except IndexError:
        raise IndexError(
            "Patient data not found. Ensure the patient ID is correct and
    present in the DataFrame."
        )
```

A.4 DepmixS4 Experiment

```
library(depmixS4)
library(tidyverse)
library(readr)

set.seed(123)

trials <- 100
N <- 100 # number of patients
M <- 2 # number of service classes
K <- 2 # number of latent states
alphas <- c(0.1, 0.2, 0.3, 0.4, 0.5)

simulate_data <- function(N, M, time_bins, alpha) {
  U <- matrix(NA, 0, M)
  Ts <- c()

  offset <- 1
  half_times <- c(floor(time_bins / 2), ceiling(time_bins / 2))
  for (n in 1:N) {
    X1 <- c(rbinom(half_times[1], 1, alpha), rbinom(half_times[2], 1, 1 -
    alpha))
    X2 <- c(rbinom(half_times[1], 1, 1 - alpha), rbinom(half_times[2], 1,
    alpha))

    U <- rbind(U, cbind(X1, X2))
    Ts <- c(Ts, time_bins)
  }

  list(U = U, Ts = Ts)
}

# Train/test split
split_data <- function(U, Ts, N, M) {
  U_train = matrix(NA, 0, M)
  U_test = matrix(0, N, M)

  offset = 1
  for (n in 1:N) {
    U_train_temp <- U[offset:(offset + Ts[n] - 2), ]
    U_train = rbind(U_train, U_train_temp)
```

```
39
40      U_test[n, ] <- U[offset + Ts[n] - 1, ]
41
42      offset = offset + Ts[n]
43    }
44    list(U_train = U_train, U_test = U_test, Ts_train = Ts[1:N] - 1)
45  }
46
47  # Train depmix model and return estimated model parameters
48  train_model <- function(U_train, M, K, Ts_train) {
49
50    data <- data.frame(U_train)
51    formula <- as.formula(paste("cbind(", paste0("X", 1:M, collapse = ","), ")
        ~ 1"))
52
53    model <- depmix(formula,
54                    data = data,
55                    nstates = K,
56                    family = multinomial("identity"),
57                    ntimes = Ts_train
58                    )
59    fm <- fit(model)
60    fb <- forwardbackward(fm)
61    viter <- viterbi(fm)
62    states <- viter$state
63
64    # Get estimated parameters
65    pars <- c(unlist(getpars(fm)))
66    p0_est <- pars[1:K]
67    p_est <- t(matrix(pars[(K + 1):(K + K*K)], K, K))
68    b_est <- t(matrix(pars[(K + K*K + 1):length(pars)], M, K))
69
70    return(list(p0_est = p0_est, p_est = p_est, b_est = b_est, fm = fm, fb =
        fb, states = states))
71  }
72
73  # Predict service class usage using trained depmix model
74  model_prediction <- function(model, last_state) {
75    p_est <- model$p_est
76    b_est <- model$b_est
77    predicted_next_state <- which.max(p_est[last_state, ])
78
79    # Return the MAP estimate
80    predicted_usage <- as.numeric(b_est[predicted_next_state,] > 0.5)
81    return(predicted_usage)
82  }
83
84  # Predict service class usage based on most frequent usage in training set
85  baseline_prediction <- function(U_train) {
86    predicted_usage <- numeric(M)
87    for (m in 1:M) {
88      predicted_usage[m] <- as.numeric(names(which.max(table(U_train[, m]))))
89    }
90    return(predicted_usage)
91  }
92
93  # Get last predicted latent state for each patient in training set
94  collect_last_known_states <- function(N, Ts_train, fb) {
```

```
  last_states <- numeric(N)
  for (n in 1:N) {
    index <- sum(Ts_train[1:n])
    last_states[n] <- which.max(fb$alpha[index,] * fb$beta[index,])
  }
  return(last_states)
}

depmix_accuracy <- data.frame(matrix(NA, trials, length(alphas)))
colnames(depmix_accuracy) <- paste0("alpha", alphas*10)
baseline_accuracy <- data.frame(matrix(NA, trials, length(alphas)))
colnames(baseline_accuracy) <- paste0("alpha", alphas*10)

for (trial in 1:trials) {
  for (alpha in alphas) {
    print(paste("==> Trial number ", trial, "of", trials, "for alpha =",
    alpha))

    # Get data and split into train and test sets
    sim_data <- simulate_data(N, M, time_bins = 12, alpha = alpha)
    data <- split_data(sim_data$U, sim_data$Ts, N, M)

    # Train depmix model
    trained_model <- train_model(data$U_train, M, K, data$Ts_train)

    # Get last known latent state for each patient
    last_known_states <- collect_last_known_states(N, data$Ts_train, trained
    _model$fb)

    # Predict next service class usage for each patient
    depmix_predictions <- matrix(NA, N, M)
    baseline_predictions <- matrix(NA, N, M)
    for (n in 1:N) {
      # Get last known latent state for the patient
      last_known_state <- last_known_states[n]

      # Predict service usage using depmix model
      depmix_predictions[n, ] <- model_prediction(trained_model, last_known_
    state)

      # Predict service usage using baseline
      baseline_predictions[n, ] <- baseline_prediction(data$U_train)
    }

    # Calculate accuracy
    depmix_accuracy[trial, which(alphas == alpha)] <- mean(depmix_
    predictions == data$U_test)
    baseline_accuracy[trial, which(alphas == alpha)] <- mean(baseline_
    predictions == data$U_test)
  }
}

# Export to CSV
write.csv(depmix_accuracy, "depmix_v2_depAccuracy.csv")
write.csv(baseline_accuracy, "depmix_v2_baseAccuracy.csv")

# Printing the Mean Accuracies for Each Method
print(colMeans(depmix_accuracy))
```

```r
print(colMeans(baseline_accuracy))

# Plot the results
depmix_long <- depmix_accuracy %>%
  pivot_longer(cols = starts_with("alpha"), names_to = "alpha", values_to = "value") %>%
  mutate(group = "Depmix")

baseline_long <- baseline_accuracy %>%
  pivot_longer(cols = starts_with("alpha"), names_to = "alpha", values_to = "value") %>%
  mutate(group = "Baseline")

combined_data <- bind_rows(depmix_long, baseline_long)

# Map alpha names to decimal values
alpha_labels <- setNames(seq(0.1, 0.5, by = 0.1), paste0("alpha", 1:5))
combined_data$alpha <- factor(combined_data$alpha, levels = names(alpha_labels), labels = alpha_labels)

# Plot the data
ggplot(combined_data, aes(x = alpha, y = value, fill = group)) +
  geom_boxplot() +
  labs(title = "Comparison of Depmix Method and Baseline Method Accuracies", x = "Alpha Value", y = "Accuracy", fill = "Method") +
  theme_minimal() +
  scale_fill_manual(values = c("Depmix" = "blue", "Baseline" = "red")) +
  scale_y_continuous(labels = function(x) paste0(x*100, "%")) +
  theme(text = element_text(size = 12),
        plot.title = element_text(size = 16, face = "bold"),
        axis.title = element_text(size = 16),
        axis.text = element_text(size = 16),
        legend.title = element_text(size = 12),
        legend.text = element_text(size = 12))

# Run t-tests
baseline <- read_csv("depmix_v2_baseAccuracy.csv")
depmix <- read_csv("depmix_v2_depAccuracy.csv")

results <- list()

for (i in 1:5) {
  column_name <- paste("alpha", i, sep="")
  test_result <- t.test(baseline[[column_name]], depmix[[column_name]], paired = TRUE)
  results[[column_name]] <- test_result
}

results
```