Modeling Cat Behavior with Markov Decision Processes in Python: A Simulation of Rusty the Cat

Author: Erica W

Introduction

Markov Decision Processes (MDPs) provide a formal mathematical framework for modeling sequential decision-making in environments characterized by stochastic dynamics and rewards Sutton and Barto (2018). In this project, the author implements an MDP to simulate the behavior of a virtual domestic cat named Rusty (Figure 1), whose internal states (e.g. hunger, fatigue, need for attention) and spatial context (e.g. room location) influence a series of state transitions over time.

This simulation demonstrates how MDP components- states, actions, transition probabilities, and a reward function- can be created to reflect behavior in a simplified but structured domain. While MDPs are traditionally applied to fields such as robotics, control systems, and reinforcement learning, this project extends their utility to modeling natural behavior in a domestic setting.

By defining a generous state space and incorporating stochastic transitions and reward values, this experiment explores how policy selection through value iteration can yield interpretable utility-driven behavior. The project offers a compact and expressive example of MDP modeling and contributes a novel application that is both relatable and useful.

Figure 1: Rusty the Cat

MDP Definition

An MDP is formally defined as a tuple $\langle S, A, T, R, \gamma \rangle$ Silver (2015), where:

- S: a finite set of states.
- A: a finite set of actions.

- $T(s, a, s') = P(s' \mid s, a)$: the transition function.
- $R(s, a)$: the reward function.
- $\gamma \in [0, 1]$: the discount factor.

The objective is to find a policy π that maximizes expected cumulative reward (Bellman Equation):

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) \mid s_0 = s\right]$$

System Design: Modeling Rusty the Cat

0.1 States and Actions

To comply with the formal definition of an MDP, a formal set of states and a set of actions were chosen. The states were chosen based on a combination of Rusty's possible feelings (e.g. tired, rested, hungry, fed, needs bathroom, relieved, wants attention, happy) and a location (living room, bathroom, or bedroom).

```
#states
    rusty_states = [#living room states
                    "tired_living_room",
                    "rested_living_room",
                    "hungry_living_room",
                    "fed_living_room",
                    "needs_bathroom_living_room",
                    "relieved_living_room",
                    "wants_attention_living_room",
                    "happy_living_room",
                    #kitchen states
                    "tired_kitchen",
                    "rested_kitchen",
                    "hungry_kitchen",
                    "fed_kitchen",
                    "needs_bathroom_kitchen",
                    "relieved_kitchen",
                    "wants_attention_kitchen",
                    "happy_kitchen",
                    #bedroom states
                    "tired_bedroom",
                    "rested_bedroom",
                    "hungry_bedroom",
                    "fed_bedroom",
                    "needs_bathroom_bedroom",
                    "relieved_bedroom",
                    "wants_attention_bedroom",
                    "happy_bedroom",]
```

Listing 1: Set of Possible States

A set of actions was also chosen based on these states. Not all actions are effective in all states, as will be seen in the transition and reward models.

```
#actions
    rusty_actions = ["eat", "go_to_bathroom", "play", "sleep", "groom", "wait"]
```

Listing 2: Set of Possible Actions

0.2 Transition Function

A transition function $T(s, a, s')$ defines the probability of transitioning from state s to state s' after taking action a. In the Rusty MDP, transitions are deterministic for some actions and random for others, modeling realistic uncertainty in Rusty's behavior.

Each state encodes a combination of Rusty's internal need (e.g., tired, hungry) and location (e.g., kitchen, bedroom). Actions such as `sleep`, `eat`, `play`, `groom`, `go_to_bathroom`, and `wait` attempt to satisfy these needs, with varying success rates.

For example, the action `sleep` in a state like `tired_bedroom` leads to the new state `rested_bedroom` with probability 0.6, and remains in `tired_bedroom` with probability 0.4. Similarly, attempting to go to the bathroom from `needs_bathroom_bedroom` has a 0.4 chance of success (transitioning to `relieved_bedroom`) and a 0.6 chance of failure.

The transitions are implemented in Python using a dictionary keyed by `(state, action)` pairs, mapping to lists of `(next_state, probability)` tuples. Below is an excerpt of the implementation:

```
for action in actions:
                key = (state, action)
                next_states = []

                #sleep
                if action == "sleep":
                    if "tired" in state:
                        next_state = state.replace("tired", "rested")
                        next_states.append((next_state, 0.6)) #mostly successful
                        next_states.append((state, 0.4))   #sometimes stays tired
                    else:
                        next_states.append((state, 1.0))

                #eat
                elif action == "eat":
                    if "hungry" in state:
                        next_state = state.replace("hungry", "fed")
                        next_states.append((next_state, 1.0))
                    else:
                        next_states.append((state, 1.0))
```

Listing 3: Excerpt of Transition Function for the Rusty MDP

These probabilistic transitions ensure that Rusty's behavior is not completely predictable, which is characteristic of a true Markov Decision Process.

0.3 Reward Function

The reward function $R(s, a, s')$ defines the immediate gain or penalty received when the agent takes action a in state s and transitions to state s'. For Rusty, this function is designed to capture both desirable and undesirable behaviors.

Positive rewards are given for fulfilling needs. For instance:

- `eat` in a `hungry_kitchen` state gives a reward of +100.
- `sleep` in a `tired_bedroom` state gives +90.
- `go_to_bathroom` in a `needs_bathroom_kitchen` state gives +40 if successful.
- `play` when `wants_attention` transitions to `happy` and earns +80.

Penalties discourage inappropriate or unproductive behavior:

- Going to the bathroom in the bedroom (unfortunately an unwelcome action in real life as well) yields a penalty of −150 due to improper location.
- Waiting while hungry, tired, or needing the bathroom incurs smaller penalties like −16 to simulate discomfort.
- Performing the wrong action (e.g., playing when hungry) results in negative reward in the range of −15 to −25.

Finally, Rusty receives small positive rewards (+15) for idle but peaceful actions like `grooming` when no needs are present. These incentives balance utility across idle and active states and make the policy more expressive and interpretable.

Implementation

The simulation was implemented in Python using a Markov Decision Process (MDP) engine tailored to Rusty's behavioral model. The implementation includes:

- **State Space:** Created as a list of strings representing combinations of internal needs (e.g., `tired`, `hungry`, `wants_attention`, etc.) and room location (`living_room`, `kitchen`, `bedroom`).
- **Action Set:** The agent can choose from six actions: `eat`, `sleep`, `go_to_bathroom`, `play`, `groom`, and `wait`.
- **Transition Function:** Encodes probabilities of moving from state s to s' when taking action a. For example, taking `sleep` in a `tired_living_room` state transitions to `rested_living_room` with probability 0.6, and remains tired with probability 0.4.
- **Reward Function:** Implements the $R(s, a, s')$ function as described previously, guiding Rusty's behavior by rewarding fulfilled needs and penalizing mistakes or inaction.
- **Value Iteration Algorithm:** A method was used to compute the optimal utility for each state. This was followed by a policy extraction step to determine which action Rusty should take in each state.

```
def value_iteration(states, actions, transition_probs, rewards, gamma, threshold=0.01):
        utilities = {s: 0 for s in states}
        while True:
            delta = 0
            new_utilities = copy.deepcopy(utilities)
            for s in states:
                action_values = []
                for a in actions:
                    transitions = transition_probs.get((s, a), [])
                    value = 0
                    for s_prime, prob in transitions:
                        reward = rewards.get((s, a, s_prime), 0)
                        value += prob * (reward + gamma * utilities[s_prime])
                    action_values.append(value)
                if action_values:
                    new_utilities[s] = max(action_values)
                    delta = max(delta, abs(new_utilities[s] - utilities[s]))
            utilities = new_utilities
            if delta < threshold:
                break
        return utilities
```

Listing 4: Value Iteration Function

```
def extract_policy(states, actions, utilities, transition_probs, rewards, gamma):
        policy = {}
        for s in states:
            best_action = None
            best_value = float("-inf")
            for a in actions:
                value = 0
                for s_prime, prob in transition_probs.get((s, a), []):
                    reward = rewards.get((s, a, s_prime), 0)
                    value += prob * (reward + gamma * utilities[s_prime])
                if value > best_value:
                    best_value = value
                    best_action = a
            policy[s] = best_action
        return policy
```

Listing 5: Policy Extraction Function

In the `value_iteration` function, each state is initially assigned a utility value of zero. The algorithm then iteratively updates these utilities using the Bellman equation. For each state, it computes the expected utility of each possible action by considering the transition probabilities to successor states, the immediate rewards, and the discounted future utilities (using gamma). The value of a state is updated to the maximum expected utility across all actions. This process continues until the maximum utility change across all states (`delta`) falls below a specified threshold, indicating convergence.

The `extract_policy` function then uses the computed utilities to determine the optimal policy. For each state, it evaluates the expected utility of each action (using the same Bellman update logic), and selects the action with the highest value as the best decision for that state. The result

is a mapping from each state to its optimal action, forming a complete policy for the agent.

Results and Analysis

The final policy generated by value iteration defines the optimal action for Rusty in each possible state. In general, the policy shows that Rusty behaves in a goal-oriented manner- taking actions that directly address his needs with minimal delay. The chosen gamma value of 0.8 indicates that Rusty values more long-term reward (closer to 1) rather than only immediate rewards (closer to 0).

For example:

- States such as `tired_living_room`, `tired_kitchen`, and `tired_bedroom` all map to the `sleep` action, indicating a preference for rest when tired.
- Similarly, all `hungry_*` states lead to the `eat` action, showing Rusty's prioritization of food when hungry, regardless of location.
- States like `wants_attention_bedroom` and `wants_attention_kitchen` correctly map to `play`, which satisfies Rusty's social needs.
- Interestingly, in non-urgent or satisfied states such as `rested_kitchen` or `fed_bedroom`, the optimal action is `groom`, suggesting a preference for self-care when no pressing needs are present (hungry, tired, needs bathroom, etc.).

```
Policy: {'tired_living_room': 'sleep', 'rested_living_room': 'groom', 'hungry_living_room
    ': 'eat', 'fed_living_room': 'groom', 'needs_bathroom_living_room': 'go_to_bathroom',
     'relieved_living_room': 'groom', 'wants_attention_living_room': 'play', '
    happy_living_room': 'groom', 'tired_kitchen': 'sleep', 'rested_kitchen': 'groom', '
    hungry_kitchen': 'eat', 'fed_kitchen': 'groom', 'needs_bathroom_kitchen': '
    go_to_bathroom', 'relieved_kitchen': 'groom', 'wants_attention_kitchen': 'play', '
    happy_kitchen': 'groom', 'tired_bedroom': 'sleep', 'rested_bedroom': 'groom', '
    hungry_bedroom': 'eat', 'fed_bedroom': 'groom', 'needs_bathroom_bedroom': 'play', '
    relieved_bedroom': 'groom', 'wants_attention_bedroom': 'play', 'happy_bedroom': '
    groom'}
```

Listing 6: Final Policy Extracted

A somewhat unusual case occurs in `needs_bathroom_bedroom`, where the policy selects `play` instead of `go_to_bathroom`. This reflects the penalty assigned to attempting bathroom relief in the bedroom (-150) and the transition probabilities that make successful relief less likely in that room. Rusty appears to avoid this risky behavior in favor of a safer action.

To better understand the behavior of the system, utility values were visualized using a heatmap via the seaborn library (see Figure 2). States where Rusty has met his needs (e.g., `happy_kitchen`, `relieved_living_room`) show higher utilities, while high-need states or locations associated with penalties (e.g., `needs_bathroom_bedroom`) exhibit lower utility values.

Overall, the policy reflects an interpretable and reasonable behavioral model. It demonstrates how reward shaping and transition modeling can encode real-world constraints and preferences, even in a stylized simulation.

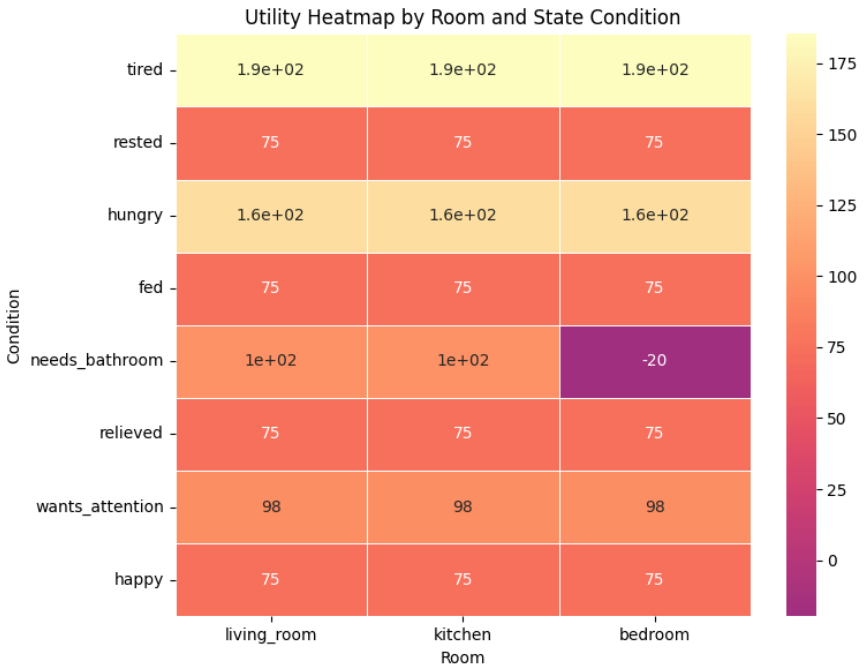

Figure 2: Utility Heatmap

Discussion

This experiment showed that Rusty was mainly utility-driven, avoiding unfavorable actions and instead being more in favor of ones that met his immediate needs. Given the probabilities and rewards as they were defined, this was not entirely surprising, even with large penalties for unfavorable actions. These results reflect what one might expect from a real cat- the behaviors were mostly intuitive.

The model strength worked well for its intended purpose, providing a simplified structure of key needs and actions in Rusty's day. A clear policy was extracted from the framework, and the probabilistic transitions added some realism to the simulation.

Some limitations of this experiment were a limited state and action space. This was mainly due to the interest of time, as a larger state space could quickly grow out of hand and may not be as easily code-able. The model is extremely simplified, showing no other external stimuli like time of day, needs decaying over time, etc. There is also no memory or learning over time from Rusty.

Possible improvements for this experiment would be to change the gamma values (to reflect the importance of immediate vs long-term rewards), change the reward/penalty values, and add more states or external stimuli or behavior, like energy decay or time of day, seeing how these affect the final utility values.

Some real-world takeaways from this experiment include the application of MDPS to robotic pets or user agents, modeling decisions in healthcare or robotics, and teaching AI through relatable systems.

References

Silver, D. (2015). Markov decision processes. https://web.stanford.edu/class/cme241/lecture_slides/david_silver_slides/MDP.pdf. Lecture slides from Reinforcement Learning course, University College London.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition.