HW2

Bucky Park

1 Introduction

I recently have been interested in finance and realized that the reinforcement learning can be applied to finance. While studying the textbook *Foundation of Reinforcement Learning with Applications in Finance* by Ashwin Rao and Tikhon Jelvis, I would like to summarize the methods for predicting and analyzing stock prices using RL from the first few chapters, mainly chapter 3 and 4. Codes are available at `https://github.com/TikhonJelvis/RL-book`. This textbook introduces some basic concepts in RL such as Markov processes, Markov reward processes and Markov decision processes with simple inventory examples. We first explore the setup of this example.

2 Setup

Let us focus on the inventory of a particular type of bicycle. Each day, there is random demand which is a non-negative integer. The probability of demand follows a Poisson distribution with parameter λ. In other words, for each $i = 0, 1, 2, \ldots$, the probability of a demand i is

$$f(i) = \frac{e^{-\lambda}\lambda^i}{i!}.$$

Let C denote the maximum storage capacity. At the end of each day, you have the option to order a certain number of bicycles from your supplier. The ordered bicycles will arrive during the store's open hours in 2 days. Let the store's state at the end of each day be (α, β), where α is the current inventory in the store and β is the inventory on a truck from the supplier, which was ordered the previous day and will arrive the next morning. Given the storage capacity constraint of at most C, your ordering policy for today is to order $C - (\alpha + \beta)$ if $\alpha + \beta < C$; and to not order otherwise.

3 Markov Processes with Inventory Example

In the setting above, we consider the set of states

$$S = \{(\alpha, \beta) \in \mathbb{Z}^2_{\geq 0} \mid 0 \leq \alpha + \beta \leq C\}.$$

Let $S_t = (\alpha, \beta)$ be the current state, then there are only $\alpha + \beta + 1$ possible next states S_{t+1} for $i = 0, 1, \ldots \alpha + \beta$ as follows:

$$(\alpha + \beta - i, C - (\alpha + \beta))$$

with transition probabilities by the Poisson distribution of demand as

$$\begin{cases} P((\alpha, \beta), (\alpha + \beta - i, C - (\alpha + \beta))) = f(i), & \text{for } 0 \leq i \leq \alpha + \beta - 1 \\ ((\alpha, \beta), (0, C - (\alpha + \beta))) = \sum_{j=\alpha+\beta}^{\infty} f(j), & \text{otherwise .} \end{cases}$$

We write code for this simple inventory example by defining transition map of the Markov processes above. Here, we cannot implement j going to infinity, in the second case where $i > \alpha + \beta$, we will store it as a single state.

```python
from dataclasses import dataclass
from typing import Mapping, Dict
from rl.distribution import Categorical, FiniteDistribution
from rl.markov_process import FiniteMarkovProcess
from scipy.stats import poisson

@dataclass(frozen=True)
class InventoryState: # class for the inventory state
    on_hand: int # represent alpha
    on_order: int # represent beta

    def inventory_position(self) -> int:
        return self.on_hand + self.on_order # return alpha + beta

class SimpleInventoryMPFinite(FiniteMarkovProcess[InventoryState]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float
    ):
        # initialization for the class from the given paramters
        self.capacity: int = capacity
        self.poisson_lambda: float = poisson_lambda

        self.poisson_distr = poisson(poisson_lambda) # poisson distribution
        super().__init__(self.get_transition_map()) # get_transition_map() function inherited from the superclass FinitMakovProcess

    def get_transition_map(self) -> \
            Mapping[InventoryState, FiniteDistribution[InventoryState]]: # override get_transition_map method
        d: Dict[InventoryState, Categorical[InventoryState]] = {} # dictionary representing the transition probabilities
        for alpha in range(self.capacity + 1): # iterate all the states, i.e. all possible combinations of (alpha, beta)
            for beta in range(self.capacity + 1 - alpha):
                state = InventoryState(alpha, beta) # extract the current state
                ip = state.inventory_position() # extract alpha + beta from the current state
                beta1 = self.capacity - ip # obtain the amount of order for today, will be beta for the next step
                state_probs_map: Mapping[InventoryState, float] = {
                    InventoryState(ip - i, beta1): # for each i, create the next possible state
                    (self.poisson_distr.pmf(i) if i < ip else # with probability f(i) if i < ip
                     1 - self.poisson_distr.cdf(ip - 1)) # with 1 - cdf otherwise
                    for i in range(ip + 1) # we do not iterate i to infinity as above.
                                       # Just iterate i until alpha + beta and gather all probability mass for i > alpha + beta to i = alpha + beta
                } # calculate transition probabilities of the current state
                d[InventoryState(alpha, beta)] = Categorical(state_probs_map) # put transition probabilities of the current state into the dictionary
        return d # return obtained transition probabilities

if __name__ == '__main__':
    user_capacity = 2 # set the inital number of bicycle
    user_poisson_lambda = 1.0 # set the Poisson parameter

    si_mp = SimpleInventoryMPFinite(
        capacity=user_capacity,
        poisson_lambda=user_poisson_lambda
    ) # define Markov processes based on the initial paramters

    print("Transition Map")
    print("--------------")
    print(si_mp)
```

```
    print("Stationary Distribution")
    print("-----------------------")
    si_mp.display_stationary_distribution()
```

Code 1: rl/chapter2/simple_inventory_mp.py

4 Markov Reward Processes with Inventory Example

Let us assume that there are two types of costs:

- A holding cost of h for each bicycle that remains in your store overnight.
- A stockout cost of p for each unit of unmet demand.

After closing the store each day, the reward R_{t+1} is calculated as the negative sum of the overnight holding cost and the day's stockout cost. Since customer demand can take an infinite number of values, this results in an infinite set of possible next states and rewards from the current state. For the current state $S_t = (\alpha, \beta)$, and customer demand i, S_{t+1} can be represented as

$$(\max(\alpha + \beta - i, 0), \max(C - (\alpha + \beta), 0))$$

and the reward R_{t+1} is

$$-h \cdot \alpha - p \cdot \max(i - (\alpha + \beta), 0)$$

Therefore, the transition probability for each $i = 0, 1, 2, ...$ can be represented as

$$P((\alpha, \beta), -h \cdot \alpha - p \cdot \max(i - (\alpha + \beta), 0), (\max(\alpha + \beta - i, 0), \max(C - (\alpha + \beta), 0))) = \frac{e^{-\lambda}\lambda^i}{i!}$$

We now work out the calculation of the reward function R_t. When α of the next state S_{t+1} is positive, this correspond to no stockout cost and only an overnight holding cost $h\alpha$. Therefore, for all $0 \leq \theta \leq C - (\alpha + \beta)$,

$$R_t((\alpha, \beta), \theta, (\alpha + \beta - i, \theta)) = -h\alpha$$

for $0 \leq i \leq \alpha + \beta - 1$. On the other hand, if α of S_{t+1} is zero, the demand for the day was at least $\alpha + \beta$ meaning there is some stockout cost and overnight holding cost. In this case, the exact stockout cost can be calculated by the expectation of the number of missed demand conditioned on the corresponding Poisson probabilities of demand exceeding $\alpha + \beta$. Therefore,

$$R_t((\alpha, \beta), \theta, (0, \theta)) = -h\alpha - p\frac{\sum_{j=\alpha+\beta+1}^{\infty} f(j) \cdot (j - (\alpha + \beta))}{\sum_{j=\alpha+\beta}^{\infty} f(j)} = -h\alpha - p\left(\lambda - (\alpha + \beta)(1 - \frac{\alpha + \beta}{1 - F(\alpha + \beta - 1)})\right).$$

We write code for this scenario by defining transition reward map of the Markov processes above. As in the case of Markov Processes, since we cannot implement i going to infinity, we will handle it as a single state.

```python
from dataclasses import dataclass
from typing import Tuple, Dict, Mapping
from rl.markov_process import MarkovRewardProcess
from rl.markov_process import FiniteMarkovRewardProcess
from rl.markov_process import State, NonTerminal
from scipy.stats import poisson
from rl.distribution import SampledDistribution, Categorical, \
    FiniteDistribution
import numpy as np


@dataclass(frozen=True)
class InventoryState: # same as the above
    on_hand: int
```

```python
    on_order: int

    def inventory_position(self) -> int:
        return self.on_hand + self.on_order


class SimpleInventoryMRP(MarkovRewardProcess[InventoryState]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float,
        holding_cost: float,
        stockout_cost: float
    ):
    # initialize the class from the given parameters
        self.capacity = capacity
        self.poisson_lambda: float = poisson_lambda
        self.holding_cost: float = holding_cost
        self.stockout_cost: float = stockout_cost

    def transition_reward(
        self,
        state: NonTerminal[InventoryState]
    ) -> SampledDistribution[Tuple[State[InventoryState], float]]:
    # implement the abstract method transition_reward

        def sample_next_state_reward(state=state) ->\
                Tuple[State[InventoryState], float]:
            # sample pair of next state and reward, return an instance of SampledDistribution

            demand_sample: int = np.random.poisson(self.poisson_lambda) # sample the demand from Poisson distribution
            ip: int = state.state.inventory_position() # get alpha + beta
            next_state: InventoryState = InventoryState(
                max(ip - demand_sample, 0),
                max(self.capacity - ip, 0)
            ) # next state based on (alpha + beta) and the demand
            reward: float = - self.holding_cost * state.state.on_hand\
                - self.stockout_cost * max(demand_sample - ip, 0) # calculate the reward
            return NonTerminal(next_state), reward # return next state and reward

        return SampledDistribution(sample_next_state_reward) # return an instance of SampledDistribution with
             sample_next_state_reward


class SimpleInventoryMRPFinite(FiniteMarkovRewardProcess[InventoryState]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float,
        holding_cost: float,
        stockout_cost: float
    ):
    # initialize the class from the given parameters
        self.capacity: int = capacity
        self.poisson_lambda: float = poisson_lambda
        self.holding_cost: float = holding_cost
        self.stockout_cost: float = stockout_cost

        self.poisson_distr = poisson(poisson_lambda)
        super().__init__(self.get_transition_reward_map())
        # get_transition_reward_map function inherited from the superclass FinitMakovRewardProcess

    def get_transition_reward_map(self) -> \
```

```python
            Mapping[
                InventoryState,
                FiniteDistribution[Tuple[InventoryState, float]]
            ]: # override get_transition_reward_map method
        d: Dict[InventoryState, Categorical[Tuple[InventoryState, float]]] = {} # dictionary representing the transition reward map
        for alpha in range(self.capacity + 1):
            for beta in range(self.capacity + 1 - alpha): # iterate all the possible states
                state = InventoryState(alpha, beta) # extract the current state
                ip = state.inventory_position() # extract alpha + beta from the current state
                beta1 = self.capacity - ip # obtain the amount of order for today, will be candidate of beta for the next step
                base_reward = - self.holding_cost * state.on_hand # precalculate h * alpha, will store the reward for the given i

                sr_probs_map: Dict[Tuple[InventoryState, float], float] =\
                    {(InventoryState(ip - i, beta1), base_reward):
                     self.poisson_distr.pmf(i) for i in range(ip)}
                # construct transition reward map when (alpha + beta) > i, in this case the reward = - h * alpha

                # construct transition reward map when (alpha + beta) < i, but i can be infinite so gather all probability mass to one
                probability = 1 - self.poisson_distr.cdf(ip - 1)
                # in this case, calculate the expected reward assuming i >= alpha + beta
                reward = base_reward - self.stockout_cost * \
                    (self.poisson_lambda - ip *
                     (1 - self.poisson_distr.pmf(ip) / probability))

                # put this (state, reward) and probability into the transition map
                sr_probs_map[(InventoryState(0, beta1), reward)] = probability
                d[state] = Categorical(sr_probs_map) # put transition map of the current state into the dictionary
        return d


if __name__ == '__main__':
    # set the initial parameters
    user_capacity = 2 # represents C
    user_poisson_lambda = 1.0 # represents lambda
    user_holding_cost = 1.0 # represents h
    user_stockout_cost = 10.0 # represents p

    user_gamma = 0.9 # represents discount factor

    si_mrp = SimpleInventoryMRPFinite(
        capacity=user_capacity,
        poisson_lambda=user_poisson_lambda,
        holding_cost=user_holding_cost,
        stockout_cost=user_stockout_cost
    ) # define Makov Reward Processes based on the parameters

    from rl.markov_process import FiniteMarkovProcess
    print("Transition Map")
    print("--------------")
    print(FiniteMarkovProcess(
        {s.state: Categorical({s1.state: p for s1, p in v.table().items()})
         for s, v in si_mrp.transition_map.items()}
    ))

    print("Transition Reward Map")
    print("---------------------")
    print(si_mrp)

    print("Stationary Distribution")
    print("-----------------------")
    si_mrp.display_stationary_distribution()
```

```
    print()

    print("Reward Function")
    print("---------------")
    si_mrp.display_reward_function()
    print()

    print("Value Function")
    print("--------------")
    si_mrp.display_value_function(gamma=user_gamma)
    print()
```

Code 2: rl/chapter2/simple_inventory_mrp.py

5 Markov Decision Processes with Inventory Example

We have assumed that the policy is fixed to order $C - (\alpha + \beta)$ if $\alpha + \beta < C$, and not to order otherwise. Now, we find the optimal value function and the optimal policy using dynamic programming With the Markov reward process we have constructed above. First, we create a class inherited from `FiniteMarkoveDecisionProcess` and use some classes methods to RL in where `rl.policy` and `rl.dynamic_programming`.

```
from dataclasses import dataclass
from typing import Tuple, Dict, Mapping
from rl.markov_decision_process import FiniteMarkovDecisionProcess
from rl.policy import FiniteDeterministicPolicy
from rl.markov_process import FiniteMarkovProcess, FiniteMarkovRewardProcess
from rl.distribution import Categorical
from scipy.stats import poisson


@dataclass(frozen=True)
class InventoryState: # same as the above
    on_hand: int
    on_order: int

    def inventory_position(self) -> int:
        return self.on_hand + self.on_order


InvOrderMapping = Mapping[
    InventoryState,
    Mapping[int, Categorical[Tuple[InventoryState, float]]]
]


class SimpleInventoryMDPCap(FiniteMarkovDecisionProcess[InventoryState, int]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float,
        holding_cost: float,
        stockout_cost: float
    ):
        self.capacity: int = capacity
        self.poisson_lambda: float = poisson_lambda
        self.holding_cost: float = holding_cost
        self.stockout_cost: float = stockout_cost

        self.poisson_distr = poisson(poisson_lambda)
        super().__init__(self.get_action_transition_reward_map())

```

```python
    # implement get_action_transition_reward_map method for MDP
    def get_action_transition_reward_map(self) -> InvOrderMapping:
        d: Dict[InventoryState, Dict[int, Categorical[Tuple[InventoryState,
                                                            float]]]] = {}

        for alpha in range(self.capacity + 1):
            for beta in range(self.capacity + 1 - alpha): # iterate all the possible states
                state: InventoryState = InventoryState(alpha, beta) # extract the current state
                ip: int = state.inventory_position() # extract alpha + beta from the current state
                base_reward: float = - self.holding_cost * alpha # precalculate h * halpha, will store the reward for the given i
                d1: Dict[int, Categorical[Tuple[InventoryState, float]]] = {} # dictionary for transition maps for different order policies

                for order in range(self.capacity - ip + 1): # iterate all the possible order from 0 to C - alpha + beta

                    # this part is the same with above code
                    sr_probs_dict: Dict[Tuple[InventoryState, float], float] =\
                        {(InventoryState(ip - i, order), base_reward):
                         self.poisson_distr.pmf(i) for i in range(ip)}

                    probability: float = 1 - self.poisson_distr.cdf(ip - 1)
                    reward: float = base_reward - self.stockout_cost * \
                        (self.poisson_lambda - ip *
                        (1 - self.poisson_distr.pmf(ip) / probability))
                    sr_probs_dict[(InventoryState(0, order), reward)] = \
                        probability
                    d1[order] = Categorical(sr_probs_dict)

                # store the obtained dictionary of transition reward maps into the dictionaries
                d[state] = d1
        return d


if __name__ == '__main__':
    from pprint import pprint
    #set the inital parameter
    user_capacity = 2
    user_poisson_lambda = 1.0
    user_holding_cost = 1.0
    user_stockout_cost = 10.0

    user_gamma = 0.9

    si_mdp: FiniteMarkovDecisionProcess[InventoryState, int] =\
        SimpleInventoryMDPCap(
            capacity=user_capacity,
            poisson_lambda=user_poisson_lambda,
            holding_cost=user_holding_cost,
            stockout_cost=user_stockout_cost
        ) # define MDP based on the paramters

    print("MDP Transition Map")
    print("------------------")
    print(si_mdp)

    fdp: FiniteDeterministicPolicy[InventoryState, int] = \
        FiniteDeterministicPolicy(
            {InventoryState(alpha, beta): user_capacity - (alpha + beta)
             for alpha in range(user_capacity + 1)
             for beta in range(user_capacity + 1 - alpha)}
    ) # Define policies for all the possible (alpha, beta)

    print("Deterministic Policy Map")
    print("------------------------")
```

```
    print(fdp)

    implied_mrp: FiniteMarkovRewardProcess[InventoryState] =\
        si_mdp.apply_finite_policy(fdp) # apply policies to mdp, it will return MRP based on the given policies
    print("Implied MP Transition Map")
    print("--------------")
    print(FiniteMarkovProcess(
        {s.state: Categorical({s1.state: p for s1, p in v.table().items()})
         for s, v in implied_mrp.transition_map.items()}
    )) # print transition map from MRP

    print("Implied MRP Transition Reward Map")
    print("---------------------")
    print(implied_mrp) # print transition reward map from MRP

    print("Implied MP Stationary Distribution")
    print("-----------------------")
    implied_mrp.display_stationary_distribution() # print stationary distribution from MRP
    print()

    print("Implied MRP Reward Function")
    print("---------------")
    implied_mrp.display_reward_function() # print reward function from MRP
    print()

    print("Implied MRP Value Function")
    print("--------------")
    implied_mrp.display_value_function(gamma=user_gamma) # print value function from MRP
    print()

    from rl.dynamic_programming import evaluate_mrp_result
    from rl.dynamic_programming import policy_iteration_result
    from rl.dynamic_programming import value_iteration_result

    print("Implied MRP Policy Evaluation Value Function")
    print("--------------")
    pprint(evaluate_mrp_result(implied_mrp, gamma=user_gamma)) # print policy evaluation value function
    print()

    print("MDP Policy Iteration Optimal Value Function and Optimal Policy")
    print("--------------")
    opt_vf_pi, opt_policy_pi = policy_iteration_result(
        si_mdp,
        gamma=user_gamma
    )
    pprint(opt_vf_pi)
    print(opt_policy_pi) # print optimal value function and optimal policy from policy iteration
    print()

    print("MDP Value Iteration Optimal Value Function and Optimal Policy")
    print("--------------")
    opt_vf_vi, opt_policy_vi = value_iteration_result(si_mdp, gamma=user_gamma)
    pprint(opt_vf_vi)
    print(opt_policy_vi) # print optimal value function and optimal policy from value iteration
    print()
```

Code 3: rl/chapter3/simple_inventory_mdp_cap.py

Figure 1 shows the value functions and policies from policy iteration and value iteration, respectively.

```
MDP Policy Iteration Optimal Value Function and Optimal Policy
--------------
{NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -43.59563313047815,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -37.97111179441265,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -37.3284904356655,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -38.97111179441265,
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -38.3284904356655,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -39.3284904356655}
For State InventoryState(on_hand=0, on_order=0): Do Action 2
For State InventoryState(on_hand=0, on_order=1): Do Action 1
For State InventoryState(on_hand=0, on_order=2): Do Action 0
For State InventoryState(on_hand=1, on_order=0): Do Action 1
For State InventoryState(on_hand=1, on_order=1): Do Action 0
For State InventoryState(on_hand=2, on_order=0): Do Action 0


MDP Value Iteration Optimal Value Function and Optimal Policy
--------------
{NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -37.97111179441265,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -37.3284904356655,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -38.97111179441265,
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -38.3284904356655,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -39.3284904356655,
 NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -43.59563313047815}
For State InventoryState(on_hand=0, on_order=0): Do Action 2
For State InventoryState(on_hand=0, on_order=1): Do Action 1
For State InventoryState(on_hand=0, on_order=2): Do Action 0
For State InventoryState(on_hand=1, on_order=0): Do Action 1
For State InventoryState(on_hand=1, on_order=1): Do Action 0
For State InventoryState(on_hand=2, on_order=0): Do Action 0
```

Figure 1: Optimal value function and policy from policy iteration and value iteration