Pan Meng
School of Computer Science and Engineering
Sun-Yat Sen University

Dec 30, 2021

HOME WORK 9 — Documentation

Problem 1. A Manufacturer at each time period receives an order for her product with probability p and receives no order with probability $1-p$. At any period, she has a choice of processing all the unfilled orders in a batch, or process no order at all. The maximum number of orders that can remain unfilled is n. The cost per unfilled order at each time period is $c > 0$, the setup cost to process the unfilled orders is $K > 0$. The manufacturer wants to find a processing policy that minimizes the total expected cost with discount factor $\alpha < 1$

1. model formulation

- State $i \in \{0, 1, \cdots, n\}$: number of unfilled orders
- Action $u \in \{0, 1\}$: process (1) or not (0)

$$u \in \{0,1\}, \text{if } i < n; \quad u = 1, \text{if } i = n$$

- State Transition $p_{ij}(u)$:

$$p_{i1}(1) = p_{i(i+1)}(0) = p, \quad p_{i0}(1) = p_{ii}(0) = 1-p, \quad i < n$$

$$p_{n1}(1) = p, \quad p_{n0}(1) = 1-p$$

- Per-stage cost

$$g(i,1) = K, \quad g(i,0) = ci$$

Figure 1: The formulation of Batch Manufacturing problem

2. pseudocode

Algorithm 1: Value Iteration

Input: $\boldsymbol{c, K, n, p, u, \alpha}$
Output: policy convergeed $\boldsymbol{J^*}$

Initialize $\boldsymbol{\theta = 0.001, \delta = \inf}$;
while $\boldsymbol{\delta > \theta}$ **do**
  for $\boldsymbol{i = 0}$ **to** $\boldsymbol{n}$ **do**

$$\boldsymbol{J_{K+1}(i) = min(k + \alpha(1-p)J_K(0) + \alpha p J_K(1), ci + \alpha(1-p)J_k(i) + \alpha p J_k(i+1)}$$

$$\boldsymbol{= |J_{K+1}(i) - J_K(i)|}$$

for $\boldsymbol{i = 0}$ **to** $\boldsymbol{n}$ **do**

$$\boldsymbol{policy_i = argmin_{u \in U}(i + \alpha(1-p)J_K(i) + \alpha p J_k(i+1))}$$

Algorithm 2: Policy Iteration

Input: c, K, n, p, u, α, s
Output: policy converged J^*

Initialize $\theta = 0.001, \delta = \inf, s = 0$;
while $\delta > \theta$ **do**
    for $i = 0$ **to** n **do**

$$value_{expected} = 0$$

$$value_{experted} = \sum_{u \in U} p_u (K + \alpha \sum_{s \in S} J_k(i)$$

for $i = 0$ **to** n **do**

$$policy_i = argmin_{uU}(i + \alpha(1-p)J_k(i) + \alpha p J_k(i+1))$$

3. code

```
class DiscountProblem():

  def __init__(self, c, K, n, p, a):
      self.c = c
      self.K = K
      self.n = n
      self.p = p
      self.a = a
      self.action_prob = {0: 0.5, 1: 0.5}
      self.transition = self.__init_transition()
      self.V = [0 for _ in range(n + 1)]

  def __init_transition(self):
      state_action = {}
      for i in range(self.n + 1):
          if (i == self.n):
              state_action[self.n] = {1: {1: self.p, 0: 1 - self.p}}
              continue
          state_action[i] = {1: {0: 1 - self.p, 1: self.p}, 0: {i + 1: self.p, i: 1 - self.p}}
      return state_action

  def next_best_action(self, s, V):
      action_values = np.zeros(2)
      for a in self.transition[s]:
          for state in self.transition[s][a]:
              cost = self.K if a == 1 else self.c * s
              action_values[a] += self.transition[s][a][state] * (cost + self.a * V[state])
      return np.argmin(action_values), np.min(action_values)

  def value_Iteration(self):
      THETA = 0.0001
      delta = float("inf")
      round_num = 0

      while delta > THETA:
          print(delta)
          delta = 0
          print("\nValue Iteration: Round " + str(round_num))
          for s in range(self.n + 1):
              best_action, best_action_value = self.next_best_action(s, self.V)
              delta = max(delta, np.abs(best_action_value - self.V[s]))
              self.V[s] = best_action_value

          print(delta)
          round_num += 1
```

```python
        policy=[]
        for s in range(self.n + 1):
            best_action, best_action_value = self.next_best_action(s, self.V)
            policy.append(best_action)
        return policy

    def __policy_evaluation(self):
        V = np.zeros(self.n+1)
        THETA = 0.0001
        delta = float("inf")

        while delta > THETA:
            delta = 0
            for s in range(self.n+1):
                expected_value = 0
                for a in self.transition[s]:
                    for state in self.transition[s][a]:
                        cost = self.K if a == 1 else self.c * s
                        expected_value += 0.5*self.transition[s][a][state] * (cost + self.a *
                            V[state])
                delta = max(delta, np.abs(V[s] - expected_value))
                V[s] = expected_value
        return V

    def policy_iteration(self):
        policy = np.tile(np.eye(2)[1], (self.n+1, 1))

        is_stable = False

        round_num = 0

        while not is_stable:
            is_stable = True

            print("\nRound Number:" + str(round_num))
            round_num += 1

            print("Current Policy")

            V = self.__policy_evaluation()

            for s in range(self.n+1):
                action_by_policy = np.argmax(policy[s])
                best_action, best_action_value = self.next_best_action(s, V)
                # print("\nstate=" + str(s) + " action=" + str(best_action))
                policy[s] = np.eye(2)[best_action]
                if action_by_policy != best_action:
                    is_stable = False

        policy = [np.argmax(policy[s]) for s in range(self.n+1)]
        return policy
```

4. simulation experiment

Discounted Problem

input: 7 2 8 0.57 0.49

value iteration: 5

strategy: [0, 1, 1, 1, 1, 1, 1, 0]

policy iteration:38

strategy: [0, 1, 1, 1, 1, 1, 1, 0]

⟶ *Answer*

Submitted by Pan Meng on Dec 30, 2021.