Machine Learning

Markov Decision Processes

Mirco Mutti
Credits to Francesco Trovò

Markov Decision Process

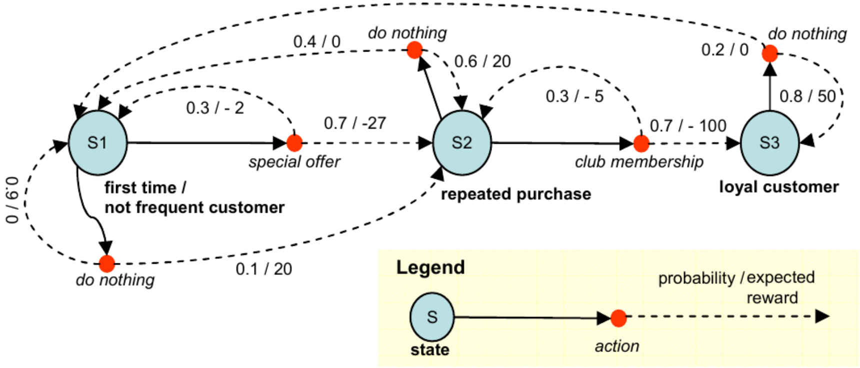
Two different problems

We would like to model the dynamics of a process and the possibility to choose among different actions in each situation

Two different problems:

- Prediction: given a specific behaviour (policy) in each situation, *estimate the expected long-term reward* starting from a specific state
- Control: learn the optimal behaviour to follow in order to *maximize the expected long-term reward* provided by the underlying process

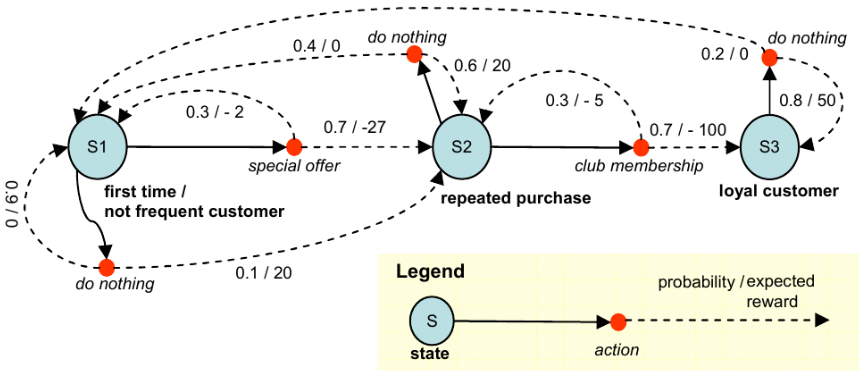
Example: Advertising Problem



- Given the actions in each state ($S1$, $S2$, $S3$) compute the value of a state
- Determine the best action in each state

Prediction

Prediction on the Advertising Problem



Given the policy $(do\ nothing, do\ nothing, do\ nothing)$, compute the value of each state

Modeling the MDP

First, we model the MDP $\mathcal{M} := (\mathcal{S}, \mathcal{A}, P, R, \mu, \gamma)$ for the given problem:

- States: $\mathcal{S} = \{\text{first time, repeated purchaser, loyal customer}\}$
- Actions: $\mathcal{A} = \{\text{do nothing, special offer, club membership}\}$
- Transition model: $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$, we need $\dim(P) = |\mathcal{S}||\mathcal{A}||\mathcal{S}|$ numbers to store it
- Reward function: $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, we need $\dim(R) = |\mathcal{S}||\mathcal{A}|$ numbers to store it
- Initial distribution $\mu \in \Delta(\mathcal{S})$
- Discount factor: $\gamma \in (0, 1]$

where $\Delta(\cdot)$ represents the simplex of a set

We assume that all the customer are first timers $\mu = (1, 0, 0)$ and use $\gamma = 0.9$

Modeling the MDP in Python

Since we know the policy π already, which is defined as

$$\pi : \mathcal{S} \to \Delta(\mathcal{A})$$

we can directly represent P^π and R^π, which are defined as:

$$P^\pi(s'|s) = \sum_a \pi(a|s) P(s'|s,a) \qquad \dim(P^\pi) = |\mathcal{S}||\mathcal{S}|$$

$$R^\pi(s) = \sum_a \pi(a|s) R(s,a), \qquad \dim(R^\pi) = |\mathcal{S}|$$

Computing the Value of the States

We have the Bellman expectation equation:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \right]$$

which we can rewrite in matrix form as:

$$V^\pi = R^\pi + \gamma P^\pi V^\pi \qquad\qquad \dim(V^\pi) = |\mathcal{S}|$$

Closed-Form Solution

Thanks to the Bellman expectation equation:

$$V^\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

Since P^π is a stochastic matrix, we have that the eigenvalues of $(I - \gamma P^\pi)$ are in $(0, 1)$ for $\gamma \in (0, 1)$ and the matrix is invertible

Recursive Solution

In the case we are not able to invert the matrix (the state space is too large) let us consider the recursive version of the Bellman expectation equation:

$$V^\pi = R^\pi + \gamma P^\pi V^\pi$$

```
V_old = np.zeros(nS)
tol = 0.0001
V = pi @ R_sa
while np.any(np.abs(V_old - V) > tol):
  V_old = V
  V = pi @ (R_sa + gamma * P_sas @ V)
```

Evaluating Different Policies

By changing the policy, which in matrix form is

$$\pi(a|s) = \Pi(s, a|s) \qquad\qquad \dim(\Pi) = |\mathcal{S}||\mathcal{S}||\mathcal{A}|$$

we are able to compute the values of the states with different strategies:

- **myopic:** we do not want to spend any money in marketing

```
pi_myo = np.array([[1., 0., 0., 0., 0.],
                   [0., 0., 1., 0., 0.],
                   [0., 0., 0., 0., 1.]])
```

- **far-sighted:** we want to spend some money in marketing for the customer in both cases if she is a new customer or if she repeatedly purchased

```
pi_far = np.array([[0., 1., 0., 0., 0.],
                   [0., 0., 0., 1., 0.],
                   [0., 0., 0., 0., 1.]])
```

Results with Different Discounts

$\gamma = 0.5$		$\gamma = 0.9$		$\gamma = 0.99$	
m	f	m	f	m	f
5.3333	-47.6202	36.3636	-9.2889	396.0396	785.3831
18.6667	-59.9347	54.5455	20.1890	415.8416	824.8548
67.5556	58.7300	166.2338	136.8857	569.3069	939.9320

- For $\gamma = 0.5$ the myopic policy evidently outperforms the far-sighted one
- For $\gamma = 0.9$ the two policies are getting close
- For $\gamma = 0.99$ the far-sighted policy becomes the most rewarding one

Control

Select the Policy

- Brute force: enumerate all the possible policies, evaluate their values and consider the one having the maximum values, generally requires $|\mathcal{S}|^{|\mathcal{A}|}$ evaluation steps
- Policy Iteration: iteratively evaluate the current policy and update it in the greedy direction
- Value Iteration: iteratively apply the Bellman optimality equation in its recursive form

In this case we do not have the option to solve the Bellman optimality equation in a closed form since the $\max$ operator is not linear

Policy Iteration

We want to solve the following problem:

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|a,s) V^*(s') \right\}$$

Decouple the process into:

- *policy evaluation*, where we compute the value V^π of the given policy
- *policy improvement*, where we change the policy from π to π' according to the newly estimated values (greedy improvement)

$$\begin{aligned} a'(s) &= \arg\max_{a \in \mathcal{A}} Q^\pi(s,a) \\ &= \arg\max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|a,s) V^\pi(s') \right\} \quad \forall s \in \mathcal{S} \end{aligned}$$

Value Iteration

Instead of iterating between policy evaluations and improvements, let us try to evaluate the optimal policy directly (i.e., to compute $V^*(s)$), by repeatedly applying the Bellman optimality equation on the current value function $V_k(s)$:

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V_k(s') \right\}$$

This procedure is guaranteed to $V^*(s)$ eventually (because the Bellman optimality equation induces a contraction)

Once we have $V^*(s)$, we can easily recover the optimal policy, i.e., the greedy one w.r.t. $V^*(s)$