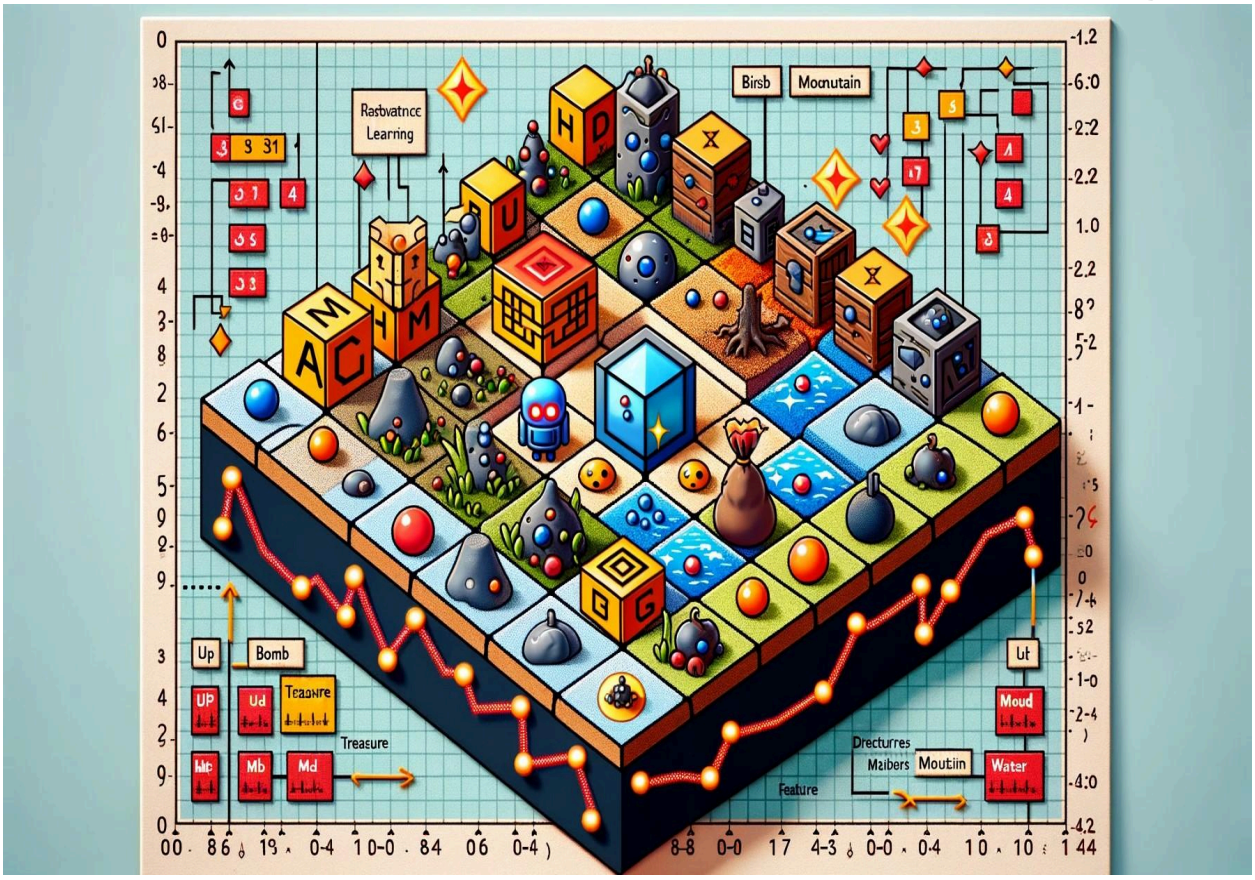
Université Paris Dauphine - PSL
Master 1 Informatique, Données, Décision
Année universitaire : 2023/2024

Bayesian Inverse Reinforcement Learning



Projet Decision Under Uncertainty

Nil-Dany Mostefai
Yani LHADJ

Enseignant : Hugo GILBERT

I. Objectif

L'objectif de ce projet est d'implémenter un mécanisme d'apprentissage par renforcement inverse pour un agent, appelé tuteur, qui opère dans un environnement, sous forme de grille, modélisé sous la forme d'un processus de décision markovien.

L'agent devra apprendre une fonction de récompense, étant donné des observations d'états-actions.

II. Mise en place de l'environnement

Le premier point que nous avons abordé est la création de l'environnement dont la mise en place s'est déroulée de manière assez fluide. Nous avons opté pour une modélisation à travers des classes car il s'agissait pour nous de la façon la plus intuitive d'implémenter le projet.

Nous avons donc créé une classe MDP, qui correspond à notre monde, composé de:

- L'environnement : représenté sous la forme particulière d'un tenseur à 3 dimensions. En effet, le tenseur peut être vu tout d'abord comme un tableau à deux dimensions, dans lequel chaque case correspond à un état de notre environnement (une case/position). Et une case correspond à une liste, qui indique les features (diamant, boue, eau, etc) qui se trouve dans cette case. D'où l'usage de ce tenseur de taille 9x9x5 qui nous simplifiera grandement lors de calculs de rewards.
- Les actions : Haut (U), Bas (D), Gauche (L), Droite (R).
- Le discount factor : 0.9 comme valeur de base.
- Les observations faites par l'agent.
- Les récompenses, notamment des constantes correspondant aux valeurs minimales et maximales liées aux récompenses. La fonction de récompense n'est pas initialisée, et est représentée dans la suite du programme (quand elle est connue) comme un tableau à 1 dimension dont chaque case correspond à la récompense d'être sur la cellule liée.
- L'état actuel du tuteur.
- La fonction de transition qui correspond, là aussi, à un tenseur à 3 dimensions, dont une case, de la forme s, a, s', correspond à la probabilité pour l'agent d'aller de l'état s vers l'état s' en appliquant l'action a. La construction de cette fonction se fait de la manière suivante : nous parcourons chaque état de l'environnement, et pour chaque état, nous regardons chaque action possible. On calcule vers quel état l'action considérée est censée nous emmener, et on attribue la probabilité 0.8 à cet état. Puis on explore deux possibilités, soit l'action est Haut ou Bas, soit elle est Gauche ou Droite, et on met les probabilités à 0.1 pour les tuples état-action-action liés.

A noter que nous avons fait en sorte de rendre cette classe la plus générale, et on peut tout à fait créer des environnement de tailles différentes, où avec des probabilités de mouvement différentes.

Avant de nous pencher dans le coeur de notre projet, nous allons présenter les quelques fonctions utilitaires que nous avons créé:

- Format_cell : elle formate les coordonnées d'une cellule donnée sous la forme décrite dans l'énoncé du projet (par exemple "A6" ou "B3") en indices de tableau.
- Format_index : qui effectue l'action inverse.
- Move_towards: qui fait bouger l'agent se situant à l'état "state", et qui effectue l'action "move" de manière déterministe.
- Agent : qui fait effectuer à l'agent l'action passée en paramètre en respectant la fonction de transition.
- Add_features: qui ajoute la feature "indice" à des cellules.
- Create_rewards : qui crée un reward simple.
- Affichage : nous nous sommes qu'il serait bien plus ludique de pouvoir directement testé notre monde et notre agent. C'est pourquoi nous avons implémenté un jeu grâce à pygame.

III. Méthodes

Randomized (question 3)

Créer un environnement MDP de manière aléatoire. La position de l'agent est initialisée aléatoirement, et un vecteur correspondant au nombre de chaque feature est passé en paramètre. Ainsi pour la feature i, on aura vector[i] features qui seront réparties sur la carte.

Policy iteration (question 4)

C'est cette fonction qui nous a incité a implémenté notre tenseur et notre reward sous les formes respectives d'un tenseur et d'un tableau.

La fonction reçoit en paramètre le reward qui va lui permettre de trouver la politique la plus adaptée, ainsi que le critère d'arrêt eps. Elle commence par initialiser une politique à 0, qui correspond à un tableau à 81 cases (soit nombre de states au carré), où chaque valeur est à 0 (la première action dans notre tableau d'action).
Tant que le critère d'arrêt n'est pas rempli, on résout le système d'équations de la forme $Av = B$ où B est le vecteur des rewards associés à chaque state, v est le vecteur qui à chaque

state associe l'espérance de gain associée à la politique courante si on exécutait celle-ci à l'infini.

Enfin le vecteur A correspondra à nos coefficients. La valeur de v pour un état s s'exprime sous la forme : $$v_\pi(s) = R(s) + \gamma \times \sum_{s' \in S} v_\pi(s') * T(s, \pi(s), s')$$

Il nous a donc suffit de modifier légèrement l'équation, mettre les variables inconnues d'un côté, et les valeur connues (les rewards) de l'autre côté, tout cela sous une forme vectorisée numpy qui a permis d'améliorer grandement les performances de la fonction.

```
-Naive computation-
3.9699581250006304  ms
-Good computation-
0.4639768939996429  ms
| 0.509 | 0.650 | 0.795 | 1.000 |
| 0.396 | ---- | 0.486 | -1.000 |
| 0.291 | 0.252 | 0.344 | 0.128 |
```

Voici un exemple de la différence de complexité entre une implémentation naïve et notre implémentation d'une fonction value iteration (fonctions vues en TP).

State actions (question 5)

Cette fonction prend une fonction reward R, un nombre de steps M, et un état initial s0. Elle renvoie une liste composée de paires état-action qui spécifie à chaque timestep (correspondant à l'indice de la liste) l'état visité par l'agent (son état actuel à ce timestep), et l'action qu'il effectue (calculée grâce à policy iteration).

Bayesian framework

Nous supposons que le tuteur applique une politique quasi-optimale, et par conséquent la valeurs Q(s, a) lorsque $\pi(s) = a$ est la plus grande parmi ces valeurs lorsque nous faisons varier la valeur de l'action a. Cela nous permet d'exprimer la probabilité d'avoir nos observations sachant une fonction de reward en supposant que chaque observation est indépendante.

En utilisant la règle de Bayes, on peut exprimer la probabilité d'avoir une certaine valeur pour le reward sachant des observations, et notre objectif c'est de faire apprendre au tuteur cette fonction de reward.

Likelihood et priors (question 6)

Nous avons mis en place deux fonctions qui calculent les probabilités à priori des rewards. La première considère que les rewards sont distribués de manière uniforme (prior_uniform), et la seconde de manière gaussienne (prior_gaussian).

On a ensuite créer une fonction likelihood qui calcule notre vraisemblance, c'est à dire la probabilité d'avoir nos observations sachant une certaine valeur de la fonction de reward. Pour ce faire, on applique policy iteration avec la valeur de reward R passée en paramètre. Puis on calcule cette probabilité comme donné dans l'énoncé du sujet.

Enfin la fonction bayesian_framework calcule les ratios.

Policy walk (question 7)

L'objectif de cet algorithme est d'estimer la probabilité d'avoir le reward R sachant des observations O.

La fonction one_neighbor nous permet d'envelopper cette idée de discrétisation de l'espace des récompenses. Elle renvoie un voisin aléatoire du reward R.

Enfin, policy_walk est la fonction qui permet de déterminer le reward qui maximise $\mathcal{P}(R|O)$.

On commence par choisir un reward au hasard dans l'espace des rewards, et on applique policy iteration pour avoir la politique optimale avec ce reward.

Dans notre boucle, on choisit un voisin R' au hasard grâce à notre fonction one_neighbor, et on calcule la fonction $Q^{\pi}(s, a, R')$.
On calcule également le ratio des vraisemblances entre R_neighbor et R.
On compare la vraisemblance la plus grande entre les deux avec la vraisemblance que nous avons enregistrée comme étant la plus grande, et si la vraisemblance calculée est plus grande, on affecte le nouveau reward.

A noter que l'on exécute policy walk avec un prior uniform. Pour changer de prior, il faut modifier le prior dans chaque appel de fonction qui utlise le prior.

Expérimentations (question 9)

Pour évaluer la performance de l'algorithme PolicyWalk sur des MDPs (Markov Decision Processes), nous avons mené une série d'expériences en variant les paramètres clés. Nous avons testé des grilles de trois tailles différentes (5x5, 10x10, 15x15), avec trois nombres de caractéristiques différents par cellule (3, 5, 7), et trois nombres d'itérations pour l'algorithme PolicyWalk (25, 50, 100). La taille de l'étape pour la modification des récompenses a été fixée à 0.1.

Pour chaque combinaison de paramètres, nous avons exécuté l'algorithme PolicyWalk dix fois afin de lisser les résultats. Pour chaque exécution, nous avons initialisé l'état de départ de l'agent de manière aléatoire et calculé la somme des récompenses obtenues. La récompense moyenne pour chaque configuration a ensuite été stockée et utilisée pour analyser les performances.

Enfin, les résultats obtenus ont été convertis en un format compatible avec les outils de visualisation et tracés sous forme de graphiques montrant la performance des algorithmes en fonction des paramètres testés. Les graphiques résultants illustrent l'impact de la taille de la grille, du nombre de caractéristiques et du nombre d'itérations sur les récompenses moyennes obtenues par l'algorithme PolicyWalk.