SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

3I005

PROJET 3

Chaînes de Markov

Encadré par : Pierre-Henri Wuillemin

Réalisé par :

RIABI Arij (3702151)

Semaine 1:

Pour la première partie de S1 on a essayé d'implémenter des fonctionnalités généralisées pour la Classe CdM pour les utiliser avec nos différents chaines de Markov.

Question 8 : analyse du graphe de transition

1.CdM.get_communication_classes():

Retourne les classes communicantes du graphe de transition (composantes fortement connexes) en utilisant l'algorithme de Tarjan impmlimanté dans utils.py qui est une variante de l'algorithme récursif de parcours en profondeur avec utilisation d'une pile Π

2.CdM.get_absorbing_classes():

Retourne les classes absorbante en cherchant pour tout les classes communicantes s'il existe un arc sortant vers un etats qui n'appartient pas à cette composante dans ce cas cela ne représente pas une sous chaîne de markov donc c'est pas une classe absorbante

3. CdM.is_irreducible():

Teste si le nombre des classes communicantes est égale 1

4.CdM.get_periodicity():

Au début si la chaine est n'es irreducible chercher la periodicity n'as pas de sens donc on retourne 1 après à chaque fois on teste si il existe un état avec une période 1 la période de la chaîne est 1 sinon on prend n'importe quel etat de la seul composante fortement connexe on cherche tous les cycles passant par cet état et après on calcule le pgcd des longueurs de ces cycles qui représente la périodicité

5.CdM.is_aperiodic():

on teste si la période est égale à 1

Semaine 2:

Question 9 :

draw_from_distribution(distribution): on tire une valeur aléatoire P et on itère sur key,value de la distribution si on trouve p<= value on retourne le key sinon on décrémente p de value et on continue.

Question 10
CollGetDistribution:

self.epsilon=epsilon # la valeur pour décider la convergence

self.pas=pas# le pas ou on va afficher a chaque fois la distribution

self.proba_states={} #stocker les probabilités des états au fur et à mesure issus de la simulation

self.prev_dest={}#stocker la dernière distribution pour faire la différence avec la distribution actuel et comparer avec epsilon

self.error=0 #on va comparer error a chaque fois à epsilon si moins on va mettre à jour self.error sinon si on arrive a la fin de la simulation sans convergence on retourne la dernière valeur

*On met à jour a chaque itération la probabilité de chaque état après on extraire le vecteur représentant des ces probabilités et on fait la différence entre celle-la et la précédente distribution si moins que epsilon on a convergence

Question 11¶
on teste si la chaîne est irréductible et apériodique

Question 12
On fait la simulation pour 3 chaînes différentes dont une est ergodique , une irréductible et une est apériodique pour mettre à jour l'importance de l'ergodicité pour la convergence ,on remarque que même si la chaîne n'est ergodique ca peut converger vers une distribution stationnaire mais cela dépend de PI 0

Question 13
On essaye d'observer la simulation selon les 4 méthodes :

Méthode 1 : Simulation

on garde la même implémentation précédente et on passe en paramètre le nbr d'itérations et le temps maximal pour pouvoir comparer le taux d’erreur selon ces hyperparamètres

Méthode 2 :Convergence de πn

On sauvegarde le temps de début de calcul et on initialise pn_1 à Pi0 et on calcule pi_n le produit de pn_1 et la matrice de transition après on commence à boucler sur cet instruction jusqu’a la convergence différence < epsilon ou le temps de calcul dépasse le temps maximale ou on atteint le nombre maximum d'itérations

Méthode 3 :Convergence de Mn

C’est le même principe que la fonction précédente sauf qu’on calcule la convergence de la matrice de transition on faisant à chaque fois Mn *Mn_1

Méthode 4 : Point fixe:

On utilise les valeurs propres de la matrice de transition `numpy.linalg.eig` `pour recupurer la valeur propre de vecteur (1)`

Semaine 3:

Question 14 et 15 pas faits par manque de temps

Jeu de l'Oie généralisée:

Un Oie(n) n’est pas une CdM à nn états puisque dans les etats pigés de type puits on sera bloquée pour deux tours donc la chaîne de markovs va avoir n+ 2 * nombre des etats puits.

On a implemanté la classe Oie.py et CollTempsMoyen.py (il ya une rreur pas trouvé encore)

Pour la classe Oie.py :

```
def get_type(self):
        type1=["N","P"]
        type2=["g","t","p"]
        t=np.random.choice(type1, p=[0.9,0.1])
        if t != "N":
```

```
        t=np.random.choice(type2, p=[self.p,self.p,self.q])
```
Cette fonction permet d'affecter un type à un state selon les p et q donnés et la probabilité d'avoir un état piégé

```
    def get_states(self):
        states=[]
        for i in list(range(1, self.n + 1)):
            states.append(str(i))
        states.append("b1")
        states.append("b2")
       return satates
```
Puisque les listes python ne supportent pas des types des données différentes et on a ajouter deux types b1 et b2 pour les 2 tours de blocage on sauvegarde tout les etats en chaînes

```
    def get_transition_distribution(self, state):
        if state == "1": # si on est au premier état on veut toujours
```
avancer
```
          self.case=2
          return {2 : 1}
        elif state == str(self.n):  # si on est arrivé à la fin on retourne au
```
début et commencement d'une nouvelle partie
```
          return {1: 1}
        elif state == 'b1': # si on est bloqué pour un tour c'est sur qu'on
```
sera bloqué pour le 2eme
```
            return {'b2': 1}
        elif state == 'b2': # si on est bloqué pour un  2eme on rentre au
```
etat ou on etait avant blocage
```
            return {str(self.case ): 1}
        elif int(state) > self.n: # si on  a dépassé la fin on rentre a la fin
            return {str(self.n) : 1}
        else:
          s=int(state)
          t=self.get_type() # on tire un type pour l'etat
          if t=="N": # si on est a un etat normal on suit le tirage du dé
```
uniform
```
              return
{str(s+1):1/6,str(s+2):1/6,str(s+3):1/6,str(s)+4:1/6,str(s+5):1/6,str(s
+6):1/6}
          else:
              if(t=="g"): # on recule d'un
```

```
        return {str(s-1): 1}
    if(t=="t"): # on avance  d'un
        return {str(s+1): 1}
    else:
        self.case=s
        return {'b1': 1}
```