IFT 615 : Devoir 3
Travail individuel

Remise : 06 juillet 2015, 9h00 (**au plus tard**)

Ce devoir comporte 2 questions de programmation. Vous trouverez tous les fichiers nécessaires pour ce devoir ici : `http://marccote.github.io/courses/ift615/devoir_3/devoir_3.zip`.

Veuillez soumettre vos solutions à l'aide de l'outil **turnin** :

```
turnin -c ift615 -p devoir_3 solution_correcteur.py solution_serpents_echelles.py
```

1. [**5 points**] Programmez l'inférence de l'explication la plus plausible dans un modèle de Markov caché (HMM), afin de corriger les erreurs de frappe dans un mot. Un mot de T lettres correspond donc à une observation $s_{1:T}$. On suppose donc que chaque lettre entrée s_t peut être le résultat d'une erreur de frappe. On suppose également que les mots ne contiennent que des lettres minuscules sans accent (les lettres de "a" à "z"). Les distributions initiales, de transition et d'observation vous sont fournies.

   Le programme doit être écrit dans le langage Python. Plus spécifiquement, vous devez remettre un fichier `solution_correcteur.py` contenant une classe nommée `Correcteur` avec un constructeur prenant 5 arguments (dans l'ordre) :

   - `p_init` : La distribution initiale (tableau Numpy 1D). Par exemple, la probabilité $P(H_1 =$ "m") est donnée par `p_init[letters2int['m']]` et correspond à la probabilité que l'usager ait voulu écrire "m" comme première lettre.
   - `p_transition` : Le modèle de transition (tableau Numpy 2D). Par exemple, la probabilité $P(H_{t+1} =$ "t"$|H_t =$ "e") est donnée par `p_transition[letters2int['t'],letters2int['e']]` et correspond à la probabilité que l'usager ait voulu écrire la lettre "t" sachant qu'il ait voulu entrer la lettre "e" [1]. Ce sont donc les colonnes de `p_transition` qui somment à 1.
   - `p_observation` : Le modèle d'observation (tableau Numpy 2D). Par exemple, la probabilité $P(S_t =$ "y"$|H_t =$ "t") est donnée par `p_observation[letters2int['y'],letters2int['t']]` et correspond à la probabilité que l'usager ait entré la lettre "y" alors qu'il aurait voulu entrer la lettre "t" [2]. Ce sont donc les colonnes de `p_observation` qui somment à 1.
   - `int2letters` : La liste des 26 lettres. Elle permet d'associer un entier `i` (un indice de 0 à 25) à une des 26 letters (l'élément `int2letters[i]`).
   - `letters2int` : Un dictionnaire qui associe chaque lettre à son indice dans la liste `int2letters`. Par exemple, `letters2int['e']` est 4 et `int2letters[4]` retourne `'e'`.

   Cette classe contient également une fonction nommée `corrige` prenant en argument :

   - `mot` : Le mot à corriger (chaîne de caractère). Chaque caractère `mot[t]` à la position `t` correspond aux observations S_t du HMM.

   Votre fonction `corrige` doit retourner une paire (tuple) contenant le mot corrigé (c'est-à-dire l'explication la plus plausible $h^*_{1:T}$) ainsi que la probabilité dans le HMM du mot observé et du mot corrigé (c'est-à-dire $P(S_{1:T} = s_{1:T}, H_{1:T} = h_{1:T})$).

   Un fichier initial `solution_correcteur.py` vous est fourni, contenant la signature de la fonction `corrige` à écrire.

1. Ces probabilités ont été dérivées à partir de fréquences extraites de plus de 2 millions de mots du Wiktionnaire (`http://fr.wiktionary.org`).
2. Ces probabilités ont été dérivées en observant la disposition des touches sur un clavier (mon ordinateur portable Mac) et en supposant que l'usager frappe la bonne touche avec 90% de chance et avec 10% une des touches voisines (le 10% est distribué uniformément sur chaque touche voisine).

Le script Python `correcteur.py` importera la classe `Correcteur` contenue dans `solution_correcteur.py` (spécifié par l'option `-correcteur`) et utilisera la fonction membre `corrige` pour corriger les erreurs de frappe de mots.

Voici comment utiliser ce script :

```
Usage: python correcteur.py [-correcteur FICHIER] [-params FICHIER] [-valider FICHIER]
                            [mot [mot ...]]

où  -correcteur : le fichier contenant votre solution.
                  Défaut = "solution_correcteur.py"
    -params     : un fichier pickle contenant les probabilités initiales
                  de l'état caché et le modèle de transitions.
                  Défaut = "params.pkl"

    -valider    : un fichier permettant de valider votre correcteur.
                  Défaut = "correcteur_validation.pkl"

    mot         : mot à corriger. Si aucun mot n'est spécifié, le correcteur
                  sera utilisé sur une liste prédéfinie de mots et sera
                  comparé avec la réponse attendue. Une comparaison sera
                  également faite avec la solution de l'exemple sur la
                  séquence de bits '0001' vu dans le cours.
```

Par défaut, le script compare le mot corrigé ainsi que la probabilité retournée avec la solution attendue. Le script utilise également l'exemple du message binaire '0100' et du canal bruité vu dans le cours (dans ce cas, la liste des letters `int2letters` est simplement `['0','1']`). Votre code devrait pouvoir fonctionner dans ce cas également, qui peut être plus utile pour le déboggage.

2. **[5 points]** Programmez l'algorithme d'itération par politiques afin de trouver la politique de jeu optimale pour un jeu de serpents et échelles modifié. Contrairement à la version originale du jeu, le joueur a un choix entre 3 actions à chaque tour : il peut soit rouler un dé et avancer du nombre de cases indiqué (action `'D'`, comme dans le jeu original), rouler deux dés et avancer d'un nombre de cases correspondant à la somme des deux dés (action `'DD'`), ou avancer d'une seule case (action `'1'`). Le plateau de jeu est constitué de 20 cases. Les transitions serpent/échelle entre les cases sont indiquées par des flèches (`=>`). Finalement, si l'action du joueur l'amène à dépasser la case 19, il ne bouge pas et reste sur sa case. Voici le plateau du jeu :

```
-----------------------------------
|0     |1=>4  |2     |3     |4     |
|      |      |      |      |      |
|______|______|______|______|______|
|5     |6     |7=>3  |8     |9     |
|      |      |      |      |      |
|______|______|______|______|______|
|10    |11=>13|12    |13=>18|14=>11|
|      |      |      |      |      |
|______|______|______|______|______|
|15    |16    |17    |18    |19    |
|      |      |      |      |      |
|______|______|______|______|______|
```

À noter que les règles du jeu sont déjà implémentées et représentées sous la forme d'un processus de décision markovien (MDP) qui vous est fourni. La classe `MDP`, définie dans `serpents_echelles.py`, permet de représenter un MPD. Les objets de cette classe contiennent les champs suivants :

- `etats` : La liste des états S du MPD. Pour le jeu serpents et échelles, il correspond à une liste des entiers de 0 à 19.
- `actions` : Dictionnaire dont les clés sont les états et les valeurs sont les listes d'actions possibles pour chaque état. Pour le jeu serpents et échelles, tous les états ont comme liste d'action `['1','D','DD']`.
- `modele_transition` : Le modèle de transition entre les états du MDP. C'est un dictionnaire dont les valeurs sont des paires état/action (s, a). La valeur associée à une clé est une liste de paires, où le premier élément est un état suivant possible et le deuxième est sa probabilité. Par exemple, pour le jeu serpents et échelles, pour l'état $s = 6$ et l'action $a = \text{D}$, la liste des états successeurs possibles et leurs probabilités est donnée par `modele_transition[(6,'D')]`, qui vaut :

  ```
  [(3, 0.166), (8, 0.166), (9, 0.166), (10, 0.166), (12, 0.166), (18, 0.166)]
  ```

  À remarquer qu'il y a une probabilité 1/6 associée à l'état suivant $s' = 3$ à cause du serpent à la case 7, et une probabilité 1/6 à la case 18 à cause des échelles aux cases 11 et 13.
- `recompenses` : Un tableau Numpy 1D donnant les récompenses. La récompense de l'état `etats[i]` est donc donnée par `recompense[i]`.
- `escompte` : Le facteur d'escompte γ du MDP.

Votre programme doit être dans le langage Python. Plus spécifiquement, vous devez remettre un fichier nommé `solution_serpents_echelles.py` contenant les fonctions suivantes :

- `calcul_valeur(mdp, plan)` : Fonction qui retourne le tableau de valeurs d'un plan (politique). `mdp` est un objet héritant de la classe MPD et `plan` est un plan (politique) et correspond à un dictionnaire qui associe à chaque état possible une action (chaîne de caractère). Le tableau de valeurs à retourner est un tableau Numpy 1D de taille `len(mdp.etats)`, dont l'élément à la position `i` donne la valeur du plan pour l'état `mdp.etats[i]`.
- `calcul_plan(mdp, valeur)` : Fonction qui retourne un plan à partir du tableau de valeurs `valeur` donné. Le plan retourné est celui qui maximise la valeur future espérée calculée à partir du tableau de valeurs donné.

— `iteration_politiques(mdp,plan_initial)` : Fonction exécutant l'algorithme d'itération par politiques, qui retourne le plan optimal et sa valeur pour le MPD spécifié par `mdp`. La fonction doit utiliser le plan `plan_initial` pour son initialisation. La fonction doit retourner une paire dont le premier élément est le plan optimal et le deuxième élément est le tableau de valeurs.

Le script Python `serpents_echelles.py` importera ces trois fonctions à partir de votre fichier `solution_serpents_echelles.py` (qui doit être dans le même répertoire). Ces trois fonctions seront évaluées séparément lors de la correction, alors assurez-vous de bien les implémenter toutes les trois. Il serait donc sage que votre fonction `iteration_politiques(mdp,plan_initial)` utilise `calcul_valeur(mdp, plan)` et `calcul_plan(mdp, valeur)` comme sous-routine.

Veuillez appeler le script `serpents_echelles.py` sans argument. Celui-ci testera vos trois fonctions et comparera les résultats obtenus avec les solutions attendues.