HMM - TP noté à rendre pour le 07/04/2020 - Gaël Savouré

I. Chaînes de Markov

I.2.a Matrice de transitions

A quelles probabilités correspond la première ligne de la matrice de transition ? et celles de la dernière colonne ?
La première ligne correspond aux probabilités d'avoir chaque lettre de l'alphabet en début de mot.
La dernière colonne correspond, pour chaque lettre, aux probabilités de terminer le mot.

Pour chaque lettre de l'alphabet, indiquer la transition la plus fréquente depuis cette lettre.

Listing 1: Output - Transitions les plus fréquentes

```
Most frequent transitions:
letter '␣': --> 't' (proba = 0.16452225)
letter 'a': --> 'n' (proba = 0.22051689)
letter 'b': --> 'e' (proba = 0.28275808)
letter 'c': --> 'o' (proba = 0.16686338)
letter 'd': --> '␣' (proba = 0.59884373)
letter 'e': --> '␣' (proba = 0.36047379)
letter 'f': --> '␣' (proba = 0.39653963)
letter 'g': --> '␣' (proba = 0.31566736)
letter 'h': --> 'e' (proba = 0.46971451)
letter 'i': --> 'n' (proba = 0.24528243)
letter 'j': --> 'o' (proba = 0.46979866)
letter 'k': --> '␣' (proba = 0.37225637)
letter 'l': --> 'e' (proba = 0.17086033)
letter 'm': --> 'e' (proba = 0.26768727)
letter 'n': --> '␣' (proba = 0.29421872)
letter 'o': --> 'n' (proba = 0.16035077)
letter 'p': --> 'e' (proba = 0.19473793)
letter 'q': --> 'u' (proba = 0.96368715)
letter 'r': --> 'e' (proba = 0.24477159)
letter 's': --> '␣' (proba = 0.43030156)
letter 't': --> 'h' (proba = 0.33937505)
letter 'u': --> 'r' (proba = 0.15036937)
letter 'v': --> 'e' (proba = 0.61843409)
letter 'w': --> 'a' (proba = 0.20323878)
letter 'x': --> 't' (proba = 0.20061728)
letter 'y': --> '␣' (proba = 0.77582944)
letter 'z': --> 'e' (proba = 0.55662188)
letter '␣': --> '␣' (proba = 1.0)
```

I.2.a Générer un mot

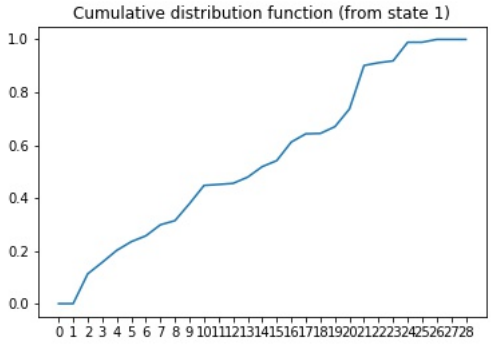
Ecrire une fonction etat_suivant qui génère un état (à t+1) à partir de l'état courant (à t) et à l'aide de la matrice de transitions et de la fonction de répartition.

Listing 2: Fonction etat_suivant

```
def nextState(transition_matrix, dic, state):
    proba_state = transition_matrix[int(state)-1]
```

```
    next_state = np.random.choice(list(dic.keys()), p=proba_state)
    return proba_state, str(next_state)
```

Afficher sur un graphique la fonction de répartition pour une ligne de la matrice de transition et expliquer son rôle pour la génération de l'état à t+1.



La fonction de répartition sert à la génération de l'état t+1 car plus un segment est pentu entre deux état, plus la probabilité de transition vers cet état est forte. On voit bien depuis cette fonction de répartition que transiter vers la lettre 't' (état 21) depuis le début d'un mot (état 1) a la probabilité la plus forte.

Utiliser cette fonction pour écrire la fonction genere_state_seq qui génère une séquence d'états jusqu'à aboutir à l'état final (28).

Listing 3: Fonction genere_state

```
def genere_state_seq(transition_matrix, dic, final_state):
    generated_states = []
    state = '1'
    while(state != final_state):
        _, state = nextState(transition_matrix, dic, state)
        generated_states.append(state)
    return generated_states
```

Ecrire une fonction display_seq qui transforme une séquence d'états en séquence de caractères, à l'aide d'un dictionnaire.

Listing 4: Fonction display_seq

```
def display_seq(dic, states):
    elements = []
    for state in states:
        elements.append(dic[state])
    return ''.join(elements)
```

Utiliser ces fonctions pour générer des mots et donner des exemples de mots générés.

Listing 5: Output - Génération de mots

```
mby
w
so
tlt
teay
```

I.2.c Générer une phrase

On veut générer une suite de mots (phrase). Créer un état final de phrase (état 29, correspondant au caractère .) dont la probabilité de transition vers cet état depuis un état final de mot est 0.1. Ecrire une fonction modifie_mat_dic qui modifie la matrice de transition et le dictionnaire en conséquence.

Listing 6: Fonction modifie_mat_dic

```
def modifyMatAndDic(transition_matrix, dic):
    dic_2 = dic.copy()

    # modify dic
    dic_2['28'] = '' # since P(28 -> 1) = 0.9
    dic_2['29'] = '.'

    # add row
    transition_matrix_2 = np.vstack((transition_matrix, np.zeros((1,28))))

    # add column
    transition_matrix_2 = np.hstack((transition_matrix_2, np.zeros((29,1))))

    # add probabilities
    transition_matrix_2[27,0]=0.9 # probability to go from final state '' to beginning of word
    transition_matrix_2[27,28]=0.1 # probability to go from end of word to dot '.'
    transition_matrix_2[27,27]=0.0
    transition_matrix_2[28,28]=1.0

    return transition_matrix_2, dic_2
```

Donner des exemples de phrases générées.

Listing 7: Output - phrases générées

```
ganoum buccovaprerined ichithe istoprgavinthe warave g re ankitig ilo er ot are pthonecad cud a
w e ppathe to spe oulee darhie cryorthag.
spre aravey bitr tisathege cher gne worone an cein achans of.
ulvedevethe morthe bley f bund he cre thigons we thes wheviftitanoretes tw orly d coper benobol
on weaigweacokeioves ten hatheroras at ined th musid of nedsucore heg.
```

I.3. Reconnaissance de la langue

Ecrire une fonction calc_vraisemblance qui calcule la vraisemblance du modèle français pour une phrase donnée en multipliant les probabilités de transition.

Listing 8: Fonction vraisemblance

```
def computeLikelihood(sentence, transition_matrix):
    new_sentence = processSentence(sentence) # transfrom sentences with + and -
```

```
    state_prev = '1'
    proba_list = []
    for letter in new_sentence[1:]:
        if letter == '-':
            state = '1'
        elif letter == '+':
            state = '28'
        else:
            state = dic_inv[letter]
        proba = transition_matrix[int(state_prev)-1, int(state)-1]
        proba_list.append(proba)
        state_prev = state
    return proba_list
```

Calculer la vraisemblance des modèles français et anglais pour la phrase « to be or not to be ».

Listing 9: Output - Vraisemblance "to be or not to be."

```
***** to be or not to be *****
likelihood (fr): 5.960208101868 64e-29
likelihood (en): 8.112892227809414e-19
```

On remarque que la vraisemblance est plus forte en utilisant le modèle anglais.

De même calculer la vraisemblance des modèles français et anglais pour la phrase « etre ou ne pas etre ».

Listing 10: Output - Vraisemblance "etre ou ne pas etre."

```
***** etre ou ne pas etre *****
likelihood (fr): 1.145706887234789e-18
likelihood (en): 4.462288711775253e-23
```

On remarque que la vraisemblance est plus forte en utilisant le modèle français.

II. HMM

II. 2. Génération de séquences d'observations

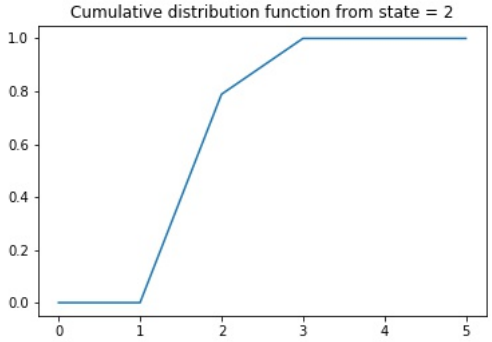
A quoi correspondent les zéros de la matrice B ? et ceux de la matrice A et du vecteur ?
La matrice B contient les probabilités d'observations depuis les différents états. Un zéro situé aux coordonnées (i,j) signifie qu'il est impossible d'obtenir l'observation (i+1) depuis l'état caché (j+1).
La matrice A contient les probabilités de transition entre les différents états cachés. Un zéro situé aux coordonnées (i,j) signifie qu'il est impossible de transiter de l'état (i+1) à (j+1).
Le vecteur *pi* contient les probabilités associées aux états initiaux. Un zéro situé à la composante i signifie qu'il est impossible de commencer par l'état (i+1).

Ecrire une fonction etat_suivant qui génère un état q_{t+1} à partir de l'état courant q_t à l'aide de la matrice de transitions et de la fonction de répartition cumsum.

Listing 11: Fonction etat_suivant

```
def nextState(transition_matrix, state):
    proba_state = transition_matrix[state-1]
    proba_cum = np.cumsum(proba_state)
    n = np.random.random()
    next_state = len(np.where(proba_cum < n)[0])+1
    return next_state, proba_cum
```

Afficher la fonction de répartition pour une ligne de la matrice de transition et expliquer son rôle pour la génération de l'état à t+1.



La fonction de répartition sert à la génération de l'état t+1 car plus un segment est pentu entre deux état, plus la probabilité de transition vers cet état est forte. On voit depuis cette fonction de répartition que transiter vers l'état 2 depuis l'état 2 est la transition avec la plus forte probabilité.

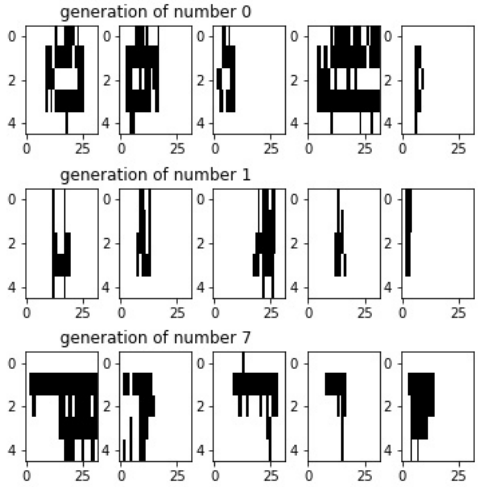
Générer une séquence d'observations suivant le modèle de Markov Caché du chiffre 0. On commencera par générer une séquence d'états suivant ce modèle à l'aide de la fonction etat_suivant. Puis on générera la séquence d'observations par le même procédé.

Listing 12: Fonctions pour générer des observations

```
def generateStates(initial_matrix, transition_matrix):
    state_prev = np.where(np.max(initial_matrix))[0][0]+1
    state_sq = []
    for i in range(v.shape[1]):
        state_sq.append(state_prev)
        state, _ = nextState(transition_matrix, state_prev)
        state_prev = state
    return state_sq

def observationsFromStates(observation_matrix, state_sq):
    observation_sq = []
    for state in state_sq:
        proba_observation = observation_matrix[:,state-1]
        proba_cum = np.cumsum(proba_observation)
        n = np.random.random()
        observation = len(np.where(proba_cum < n)[0])+1
        observation_sq.append(observation)
    return observation_sq
```

Visualiser le résultat sous forme d'image. Générer des séquences pour le chiffre 7 et le chiffre 1 (matrices B1.txt, B7.txt, etc...)



II.3. Calcul de la vraisemblance de séquences d'observations

Calculer la vraisemblance de ces séquences suivant chacun des modèles (0, 1 et 7) par l'algorithme de Viterbi (on pourra implémenter la version logarithmique de cet algorithme). Pour cela les matrices A, B et π seront converties en logarithmes (utiliser np.log).

Listing 13: Algorithme de Viterbi

```
def likelihoodViterbi(seq, A, B, pi):

    A = np.ma.log(A).filled(-500)
    B = np.ma.log(B).filled(-500)
    pi = np.ma.log(pi).filled(-500)

    vit_matrix = np.zeros((A.shape[0], len(seq)))
    # initialization
    for state in range(1,vit_matrix.shape[0]+1):
        observation = seq[0]
        proba_start = pi[state-1]
        proba_obs = B[state-1,observation-1]
        vit_matrix[state-1,0] = proba_start + proba_obs
    # fill the rest of the matrix
    state_list = np.arange(vit_matrix.shape[0])+1
    for obs_idx in range(len(seq[1:])):
        observation = seq[obs_idx]
        for state in state_list:
            proba_trajectories = []
            for state_prev in state_list:
                delta_prev = vit_matrix[state_prev-1,obs_idx]
                proba_obs = B[observation-1,state-1]
                proba_state = A[state_prev-1,state-1]
                proba_trajectories.append(delta_prev + proba_obs + proba_state)
```

```
            vit_matrix[state-1,obs_idx+1] = np.max(proba_trajectories)
    return np.max(vit_matrix[:,-1])
```

II.3.2 Donner le résultat de la classification des images de test en considérant un problème à trois classes : 0, 1 et 7.

Listing 14: Output - Résultats de classification (SeqTest0.txt)

```
Sequence 1: model prediction 0
Sequence 2: model prediction 0
Sequence 3: model prediction 0
Sequence 4: model prediction 0
Sequence 5: model prediction 0
Sequence 6: model prediction 0
Sequence 7: model prediction 0
Sequence 8: model prediction 0
Sequence 9: model prediction 0
Sequence 10: model prediction 0
```

Listing 15: Output - Résultats de classification (SeqTest1.txt)

```
Sequence 1: model prediction 1
Sequence 2: model prediction 1
Sequence 3: model prediction 1
Sequence 4: model prediction 1
Sequence 5: model prediction 1
Sequence 6: model prediction 1
Sequence 7: model prediction 1
Sequence 8: model prediction 1
Sequence 9: model prediction 1
Sequence 10: model prediction 1
```

Listing 16: Output - Résultats de classification (SeqTest7.txt)

```
Sequence 1: model prediction 7
Sequence 2: model prediction 7
Sequence 3: model prediction 1
Sequence 4: model prediction 7
Sequence 5: model prediction 7
Sequence 6: model prediction 7
Sequence 7: model prediction 7
Sequence 8: model prediction 7
Sequence 9: model prediction 7
Sequence 10: model prediction 7
```