Chaînes de Markov

November 27, 2019

1 Algorithmique et Pensée Computationnelle

2 Chaînes de Markov

2.1 Exercice

Pour cet exercice, nous avons analysé le texte intégral de **Candide, ou l'Optimisme** de *Voltaire* afin de créer un dictionnaire qui se comporte comme une chaîne de Markov, où chaque mot peut être suivi d'un autre mot avec une probabilité `p`. Pour savoir quels mots suivent quel autre mot avec une certaine probabilité, vous pouvez utiliser `data[mot]` et voir quels mots le suivent.

Par exemple `print(data["leibnitz"])` affiche `{'navait': 0.5, 'ne': 0.5}`

En partant du mot ***leibnitz****, dessinez une chaîne de Markov qui affiche les probabilités d'avoir un mot à la suite d'un autre.*

[ ]:
```
from functools import reduce

class Candide:
    _data = None

    @staticmethod
    def data():
        if Candide._data is None:
            Candide.load_text()
        return Candide._data

    @staticmethod
    def clean_word(word):
        return ''.join(l for l in word.lower().strip() if l.isalnum())

    @staticmethod
    def load_text():
        with open("candide.txt", "r") as file:
            candide = file.read().split()
            data = {}
            for i in range(len(candide) - 1):
                word = Candide.clean_word(candide[i])
```

```
                next_word = Candide.clean_word(candide[i+1])
                if word in data:
                    if next_word in data[word]:
                        data[word][next_word] += 1
                    else:
                        data[word][next_word] = 1
                else:
                    data[word] = {next_word: 1}
            Candide._data = Candide.compute_stats(data)

    @staticmethod
    def compute_stats(_data):
        for i in _data.keys():
            _sum = reduce(lambda accumulator, j: accumulator + _data[i][j],␣
 ↪_data[i].keys(), 0)
            for j in _data[i]:
                _data[i][j] /= _sum
        return _data
```

[ ]:
```
data = Candide.data()

def most_likely(word_data):
    _max = 0
    best = None
    for next_word in word_data.keys():
        if word_data[next_word] > _max:
            best = next_word
            _max = word_data[next_word]
    return best

def create_sentence(first_word, data, depth):
    current_word = first_word
    sentence = first_word
    for i in range(depth):
        current_word = most_likely(data[current_word]) # On appelle␣
 ↪most_likely() ici
        sentence = sentence + " " + current_word
    return sentence

create_sentence("leibnitz", data, 105)
```

2.2 Exercice

Comme vous le voyez, votre phrase contient beaucoup de répétitions, car certains mots sont très fréquents (comme le mot `et`) et ils créent donc une boucle infinie.

Pour empêcher ceci, faites en sorte que les mots soient choisis aléatoirement tout en respectant la

probabilité qu'ils soit choisis. Vous pouvez utiliser une `table de probabilités`

[ ]:
```
from random import random

data = Candide.data()

def proba_table(word_data, sentence):
    cumulative = 0
    r = random()
    for word in word_data.keys():
        proba = word_data[word]
        if proba + cumulative > r:
            return word
        cumulative += proba

def create_sentence(first_word, data, depth):
    current_word = first_word
    sentence = first_word
    for i in range(depth):
        current_word = proba_table(data[current_word], sentence) # On appelle␣
↪probal_table() ici
        sentence = sentence + " " + current_word
    return sentence

create_sentence("leibnitz", data, 100)
```

2.3 Exercice

Votre bon ami `Vladimir` vous propose d'investir `$10000` dans son business de "plantes médicinales".

`Vladimir` est un revendeur très réputé dans votre quartier, cependant vous savez qu'il peut arriver que ses plantes ne soient pas toujours légales et du coup que vous risquez de vous faire arrêter en vous associant à lui.

Vous estimez néanmoins qu'il peut être intéressant d'investir dans ce projet, cependant vous ne voulez pas prendre trop de risques. C'est pourquoi, vous investissez la somme `i`, tel que:

$$i < 10000$$

Soit , un nombre qui correspond au pourcentage de votre investissement par rapport à `10000`, tel que:

$$\pi = \frac{\mathrm{i}}{10000}$$

Vous savez que tous les jours, votre investissement vous rapportera

$$r(\pi) = \pi^2$$

Néanmoins, plus votre investissement est élevé, plus vos collègues vont se méfier de vos rendements, c'est pourquoi, tous les jours vous avez la probabilité ( ) de vous faire arrêter:

$$\sigma(\pi) = \frac{1}{1 + e^{\frac{-\pi}{5}}} - \frac{1}{2}$$

2.4 Créez un chaîne de Markov Monté Carlo qui détermine combien d'argent vous aurez après T périodes et quelle est la probabilité que vous vous fassiez arrêter durant ces T périodes.

[ ]:
```python
from math import exp
from random import random

def probability_prison(pi):
    return 1/(1+exp(-pi/5)) - 1/2

def run_simulation(pi, proba_prison):
    bank = 0
    for i in range(T):
        r = random()
        if r < proba_prison:
            return -1
        bank += r**2
    return bank

def markov_chain(i, pi, T):
    _simulations = 1000
    proba_prison = probability_prison(pi)
    prisons = 0
    bank = 0
    for i in range(_simulations): # run 100 simulations
        outcome = run_simulation(pi, proba_prison)
        if outcome == -1:
            prisons += 1
        else:
            bank = outcome
    return (prisons/_simulations, bank)

i = 1000
T = 30
pi = i / 10000

markov_chain(i, pi, T)
```

[ ]:
```
```

Correction Pierre - Feuille - Ciseaux | Chifoumi

November 27, 2019

1 Algorithmique et Pensée Computationnelle

2 Pierre - Feuille - Ciseaux | Chifoumi

Dans ce notebook, on vous demande de créer un jeu de Pierre - Feuille - Ciseaux ou Chifoumi où l'utilisateur pourra jouer face à l'ordinateur.

Les règles : * Pierre vs papier -> papier gagne * Pierre vs ciseaux -> pierre gagne * Papier vs ciseaux -> ciseaux gagne

Dans une boucle, générez aléatoirement une action pour l'ordinateur, demandez à l'utilisateur son choix et comparez pour déterminer le vainqueur. Entre chaque manche, affichez le gagnant et le score.

[ ]:
```
from random import choice

#create a list of play options
#init scoreboard
computer_score = 0
player_score = 0

move_table = {
    "Rock": ("Paper", "Scissors"),
    "Paper": ("Scissors", "Rock"),
    "Scissors": ("Rock", "Paper")
}

options = tuple(move_table.keys())

while True:
    print("YOU:", player_score, ": COMPUTER:", computer_score)
    player = input("Rock, Paper, Scissors? Enter Exit to end the game\n")
    if player == "Exit":
        break
    if player not in move_table: # O(1) vs `in options` O(n)
        print("INVALID PLAY")
        continue
```

```
    #assign a random play to the computer
    computer = choice(options)
    print("\nCOMPUTER PLAYS:", computer, "\n")
    if move_table[player][0] == computer:
        print("COMPUTER WINS\n")
        computer_score += 1
    elif move_table[player][1] == computer:
        print("YOU WIN\n")
        player_score += 1
    else:
        print("EQUALITY\n")
```

[ ]:

Fingerprinting_solutions

November 27, 2019

1 Algorithmique et Pensée Computationnelle

1.0.1 Fingerprinting

L'algorithme Fingerprinting est une procédure qui met en correspondance un élément de données arbitrairement volumineux avec une chaîne de bits beaucoup plus courte. Tout comme une empreinte digitale, il identifie de façon unique les données originales.

https://en.wikipedia.org/wiki/Fingerprint_(computing)

Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs. Ces deux diviseurs sont 1 et le nombre considéré, puisque tout nombre a pour diviseurs 1 et lui-même, les nombres premiers étant ceux qui n'en possèdent aucun autre.

Ecrire une fonction qui vérifie si un nombre est premier ou non.

[ ]:
```
def is_a_prime_number(num):
    # votre code ici
    if num <= 1:
        return False
    for i in range(2, int(num**.5)):
        if num % i == 0:
            return False
    return True
```

L'algorithme Fingerprinting est le suivant:

a. Si **p** est un nombre premier, calculez la valeur de hachage de la chaîne à l'aide de la fonction **hash(...)**, puis calculez le modulo du résultat du hachage.

b. Sinon, imprimez un message qui dit que le nombre n'est pas un nombre premier.

[ ]:
```
def fingerprinting(p, message):
    # votre code ici
    if is_a_prime_number(p):
        result = hash(message) % p
        return result
    print(str(p) + " is not a prime number!")
```

[ ]:
```
message_to_hash = "Hello, world!"
p = 11

print("Hash result is " + str(fingerprinting(p, message_to_hash)))
```

[ ]:
```
your_details = [19, hash("mypassword")]
```

L'algorithme Fingerprinting est une bonne pratique pour stocker les mots de passe dans une base de données. Chaque fois qu'un utilisateur veut se connecter, l'algorithme effectue les calculs et compare les résultats.

Par exemple, si votre mot de passe est **mypassword**, alors votre mot de passe sera enregistré comme **295848469771299555281**.

Ecrire une fonction, **login**, qui retourne vrai (true) si l'utilisateur saisit son mot de passe avec succès.

[ ]:
```
def login(password, your_details):
    # votre code ici
    return your_details[1] % your_details[0] == fingerprinting(your_details[0],␣
 ↪password)
```

Ecrire un bout de code qui prend un mot de passe de l'utilisateur et l'imprime si la connexion est réussie.

[ ]:
```
# votre code ici
password = input("Enter your password")
print("Login successful or not? " + str(login(password, your_details)))
```

[ ]:
```

```

Monte Carlo

November 27, 2019

1 Algorithmique et Pensée Computationnelle

2 Monte Carlo

3 PI

Pour approximer , il suffit de résoudre l'équation de l'aire d'un cercle:

$$A = \pi * r^2$$
$$\pi = \frac{A}{r^2}$$

Maintenant, prenons un cercle de rayon `1`:

$$\pi = A$$

Il ne nous manque que l'aire pour trouver . Or, nous savons calculer une aire en utilisant Monte Carlo. Il nous suffit donc de trouver l'aire de ce cercle imaginaire de `1`de rayon.

Pour ce faire, nous savons que la distance entre deux points vaut:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Nous cherchons la distance avec le point `(0, 0)`:

$$\sqrt{(x - 0)^2 + (y - 0)^2}$$

Pour qu'un point soit hors de notre cercle de rayon `1`, il faut que:

$$\sqrt{x^2 + y^2} > 1$$

$$x^2 + y^2 > 1$$

Pour nous simplifier la tache, nous n'allons calculer que les points pour

$$x, y \in \{0, 1\}$$

Ce qui veut dire que nous allons calculer

$$\frac{\pi}{4}$$

Pour ce faire, nous créons des paires aléatoires de nombres `(x, y)` compris entre `0` et `1`.

Si:

$$x^2 + y^2 \leq 1,$$

on incrémente une variable `inside`. Le nombre de paires dans notre cercle divisé par le nombre total de points évalués nous donne un approximation de `PI/4`.

[ ]:
```
from random import uniform

def evaluate_pi(iterations):
    inside = 0
    for i in range(iterations):
        x, y = (uniform(0, 1), uniform(0, 1))
        if x**2 + y** 2 <= 1:
            inside += 1
    return (inside/iterations)*4

evaluate_pi(10000000)
```

3.1 Exercice

3.1.1 Soient les fonctions:

$$f(x) = x^2$$
$$g(x) = e^x$$
$$h(x) = \tanh(x) \ln(x)$$
$$z(x) = \frac{h(f(g(x)))}{e^{g(x)}}$$

Utilisez Monte Carlo pour trouver les intégrales des fonctions `f`, `g`, `h` et `z` dans l'interval

$$X = [0, 1]$$

[ ]:
```
from numpy import cos, exp, log
from random import uniform
from math import inf

def f(x):
```

```
    return x**2

def g(x):
    return exp(x)

def h(x):
    return cos(x)*log(x)

def z(x):
    return h(f(g(x)))/exp(g(x))

def integrale(function, _min = 0, _max = 1, iterations = 100000):
    total = 0
    for i in range(iterations):
        rand_x = uniform(_min, _max)
        total += function(rand_x)
    return total/iterations

print("F(x):", integrale(f))
print("G(x):", integrale(g))
print("H(x):", integrale(h))
print("Z(x):", integrale(z))
```

[ ]:

Algorithmique et Pensée Computationnelle

Quick sort

Quick Sot est une autre manière de trier une liste en utilisant un pivot selectionné de manière aléatoire. C'est aussi un algorithme utilisant le paradigme diviser-pour-régner comme le tri par fusion. L'avantage de quick sort est qu'il existe une version de l'algorithme probabilistique. Voici deux liens utiles pour comprendre son fonctionnement: https://fr.wikipedia.org/wiki/Tri_rapide (https://fr.wikipedia.org/wiki/Tri_rapide), https://www.youtube.com/watch?v=xcyDSLSkb0k (https://www.youtube.com/watch?v=xcyDSLSkb0k)

"La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié. "

1) Ecrire la fonction partition qui prend en entrée la list et les bornes de la fénêtre. Partition retourne l'index de partition. Le pseudocode est disponible sur la page wiki si besoin.

In [ ]:

```
def partition(L, low, high):
    i = low - 1     # index of smaller element
    pivot = L[high]  # pivot
    # pivot=randint(start,end)

    for j in range(low, high):
        # If current element is smaller than or
        # equal to pivot
        if L[j] <= pivot:

            # increment index of smaller element
            i = i+1
            L[i],L[j] = L[j],L[i]

    L[i+1], L[high] = L[high], L[i+1]
    return i+1
```

2) Ecrire la fonction quick sort à l'aide de la fonction partition.

In [ ]:

```
def quick_sort(L, low, high):
    if low < high:
        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quick_sort(arr, low, pi-1)
        quick_sort(arr, pi+1, high)
    return arr
```

3) Nous allons maintenant comparer la complexié de quick sort par rapport au tri par fusion. Est ce que le randomized Quick sort est plus efficace ? Pour répondre à cette question etudier les cas moyens de quick sort et le pire cas pour Merge Sort. Voici le code de merge sort lancer le avec differents tableaux (tié, non-trié etc...). Que constatez-vous par rapport à quick sort ? (l'utilisation au début de la cellule de **%time** sera utile ici)

In [ ]:

```
def merge_sort(arr):
    if len(arr) >1:
        mid = len(arr)//2 #Finding the mid of the array
        L = arr[:mid] # Dividing the array elements
        R = arr[mid:] # into 2 halves

        merge_sort(L) # Sorting the first half
        merge_sort(R) # Sorting the second half

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i+=1
            k+=1

        while j < len(R):
            arr[k] = R[j]
            j+=1
            k+=1
    return arr
```

In [ ]:

```
%time
arr = [5, 8, 6, 18, 97, 56, 899, 17, 6, 18, 78, 28, 18, 27, 18]
print("Pour un tableau non trié merge sort: ", merge_sort(arr))
%time
arr = [5,8, 10, 13, 18, 29, 39, 56, 78, 78, 89, 90, 99, 120, 149]
print("Pour un tableau trié merge sort: ", merge_sort(arr))
```

In [ ]:

```
%time
arr = [5, 8, 6, 18, 97, 56, 899, 17, 6, 18, 78, 28, 18, 27, 18]
print("Pour un tableau non trié quick sort: ", quick_sort(arr, low=0, high=len(a
rr)-1))
%time
arr = [5,8, 10, 13, 18, 29, 39, 56, 78, 78, 89, 90, 99, 120, 149]
print("Pour un tableau trié quick sort: ", quick_sort(arr, low=0, high=len(arr)-
1))
```

4) Calculer la complexité de Quick sort, dans le meilleure des cas, le pire des cas et le cas moyen.

Indication: pour calculer le meilleure et le pire des cas, réfléchisser au cas particuliers quand cela arrive. Pour le cas moyen, il y a différents manières de le résoudre soit utiliser une équation de récurrence (idéal pour les appels récursifs) soit exprimer une variable aléatoire de probabilité et calculer son espérance de la forme: $E[X] = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} c_{i,j}$ *où* $c_{i,j}$ *est une variable aléatoire binaire (prenant la valeur 0 si l'élément à la position i est comparé avec* x_j*)*

Meilleur cas: le meilleur cas arrive lorsque le pivot est choisi tel que la liste est toujours divisée par deux en proportion égale. Cela signifie que chaque appel récursif traite une liste dont la taille est réduite de moitié. Par conséquent, nous ne pouvons faire que des appels imbriqués $log_2(n)$ avant d'atteindre une liste de taille 1. Cela signifie que la profondeur de l'arborescence des appels est $log_2(n)$. Mais aucun appel au même niveau de l'arborescence des appels ne traite la même partie de la liste d'origine; ainsi, chaque niveau d'appels n'a besoin que de $O(n)$ fois dans son ensemble (chaque appel a un surcoût constant, mais puisqu'il n'y a que des appels $O(n)$ à chaque niveau, il est inclus dans le facteur $O(n)$). Le résultat est que l'algorithme utilise uniquement le temps $O(nlog(n))$.
Pire des cas: La partition la plus déséquilibrée se produit lorsque l'une des sous-listes renvoyées par la routine de partitionnement est de taille n-1. Cela peut se produire si le pivot est l'élément le plus petit ou le plus grand de la liste. Si cela se produit de manière répétée dans chaque partition, chaque appel récursif traite une liste de taille inférieure à la liste précédente. Par conséquent, nous pouvons faire $n - 1$ appels imbriqués avant d'atteindre une liste de taille 1. Cela signifie que l'arborescence des appels est une chaîne linéaire de $n - 1$ appels imbriqués. Le $i^{ème}$ appel fonctionne $O(n - i)$ pour créer la partition et $\sum_{i=0}^{n}(n - i) = O(n^2)$, donc dans ce cas Quicksort prend $O(n^2)$. **Cas moyen:** Utiliser l'équation $T(n) = 0(1) + O(n) + 2T(n/2)$ ce qui conduit à une division par deux de l'espace dans la fonction quick*sort avec les appels récursifs et* $0(n)$ *pour la fonction partition ou $E[X] = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n} c_{i,j} = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j+1} = O(\sum_{i=0}^{n-1}log(i)) = O(nlog(n))$*

In [ ]:

```

```

Randomized_algos_solutions

November 27, 2019

1 Algorithmique et Pensée Computationnelle

1.1 Randomized algorithms

Un algorithme randomisé prend par exemple des valeurs aléatoires comme inputs dans le but d'améliorer sa performance.

Il existe deux principaux types d'algorithmes randomisés: Las Vegas & Monte Carlo.

1.1.1 Las Vegas Algorithm

Il s'agit d'un algorithme utilisant une valeur aléatoire comme input pour toujours trouver une réponse **correcte**. Le temps d'exécution est cependant **très variable car non-déterministique**. Le nombre d'itérations varie.

1.1.2 Monte Carlo Algorithm

Il s'agit d'un algorithme ayant une chance de produire une réponse **incorrecte**. Le temps d'exécution est cependant **fixe**.

Exercise Etant donné une liste `L` de n éléments contenant la valeur `z`, trouvez l'index de `z`. Utilisez l'algorithme Las Vegas.

Autrement dit, à la place de faire une recherche linéaire ou binaire, faites en sorte que l'algorithme cherche `z` de façon aléatoire.

Le résultat doit retourner dans un tuple l'index de `z` et le nombre d'itérations effectuées. Example: (5,4)

[ ]:
```
from random import randrange
def lasvegas_search(L,n):
    # votre code ici
    i= 0
    while True:
        i += 1
        index = randrange(n)
        if L[index] == "z":
```

```
            return index,i
            break


L = ["a", "g", "t", "r", "o", "z", "u", "p"]
n = len (L)
lasvegas_search(L,n)
```

Exercise Etant donné une liste `L` de n éléments contenant la valeur `z`, trouvez l'index de `z`. Utilisez l'algorithme Monte Carlo.

Le résultat doit retourner dans un tuple la valeur de `tag`, l'index de `z` et le nombre d'itérations effectuées. Example: (True, 2, 6)

[ ]:
```
from random import randrange
def montecarlo_search(L,n,x):
    i = 0
    tag = False
    # votre code ici
    while i <= x:
        index = randrange(n)
        i += 1
        if L[index] == "z":
            elem = index
            tag = True
    return tag, index, i

L = ["a", "g", "t", "r", "o", "z", "u", "p"]
n = len (L)
x = 5 #number of maximum iterations
montecarlo_search(L,n,x)
```

Source: https://www.youtube.com/watch?v=Nn_Eq7–Rk4