Université de Bordeaux
L2 Informatique/Maths-Informatique

Probabilités, Statistiques, Combinatoire : TM4

Simulations

L'objectif de cette feuille est de *simuler* des expériences aléatoires d'une part, et d'*observer* le résultat de ces expériences, en les répétant un grand nombre de fois, d'autre part.

On veut faire apparaître ce qui est la définition empirique intuitive de la probabilité : Par exemple, on dit qu'on a une chance sur six d'obtenir un six lors d'un lancer d'un dé (équilibré). Comment observer que c'est vraiment le cas ? Si on lance le dé un très grand nombre de fois, on s'attend à ce que la proportion des lancers donnant le six parmi tous les lancers soit proche de 1/6. Rien ne dit pourtant qu'en lançant le dé six cent fois, on doit obtenir exactement cent fois chacune des valeurs possibles de 1 à 6.

On appellera *simulateur* d'une expérience, une **fonction** (sans arguments) qui simule un tirage selon l'expérience demandée, et retourne le résultat. Des appels successifs à la fonction retourneront des résultats indépendants. L'un des objectifs de cette feuille est d'écrire des simulateurs pour diverses expériences ; autant que possible, on cherchera à réutiliser ces simulateurs pour en construire d'autres.

Par exemple, `simuleDe6` est un simulateur d'un lancer d'un dé équilibré à six faces :

```
simuleDe6 = lambda : random.randint(1,6)
```

(qu'on aurait pu aussi définir par

```
def simuleDe6():
    return random.randint(1,6)
```

)

Afin de **tester** empiriquement les simulateurs, on procédera de la manière suivante : on appellera le simulateur un grand nombre n de fois ($n = 100$, 1000, un million : c'est la machine qui fait le travail), et on résumera les résultats en retournant une structure de données indiquant, pour chaque "valeur", le *nombre de fois* qu'elle a été obtenue. En divisant ces nombres par n, on obtiendra des *fréquences empiriques*, dont on espère qu'elles seront proches des probabilités qui définissent le modèle.

Dans cette séance, vous serez amené à écrire ces deux types de fonctions. Les fonctions dont le nom commence par `simule` sont supposées être des simulateurs ; les fonctions dont le nom commence par `teste`, sont des testeurs.

Les dictionnaires de `Python`

Pour manipuler les fréquences empiriques d'apparition d'événements, nous allons utiliser une structure de données de *dictionnaire*, fournie par le langage `python`.

Le dictionnaire est un conteneur – il n'y a pas d'ordre sur les données. Celles-ci sont organisées en couples de forme `clé : valeur`, par exemple `{ (1,4): 3, 6: 'badPassword'}` est un dictionnaire avec deux clés, le tuple `(1,4)` (auquel est associée la valeur 3) et l'entier `6` (auquel est associée comme valeur, une chaîne de caractères). Les clés peuvent être des entiers (comme dans un tableau), mais aussi des tuples, ou des chaînes de caractères (on ne peut pas utiliser de liste, ni aucun objet modifiable, comme clé d'un dictionnaire).

Pour initialiser un dictionnaire vide, on peut utiliser `monDict = {}`.

Pour ajouter un couple clé-valeur à un dictionnaire, on utilise la clé comme indice :
`monDict[(1,2)] = 4`

Si la clé n'existe pas encore dans le dictionnaire, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe `=` ; si la clé est déjà présente, l'ancienne valeur est remplacée par la nouvelle.

Pour accéder à une valeur précise, on utilise la même syntaxe qu'avec les listes :
`a = monDict[c]`
affectera la valeur associée à la clé `c` à la variable `a` – ou provoquera une erreur, si la clé n'existe pas dans le dictionnaire.

Pour tester si une clé appartient au dictionnaire, et aussi pour parcourir toutes les clés du dictionnaire, on utilise le mot-clé `in` comme pour les listes :
`(1,3) in monDict` vaut `False`, `(1,2) in monDict` vaut `True`.

```
for c in monDict:
   print(monDict[c])
```

affiche chaque clé présente dans le dictionnaire (dans un ordre qui n'est pas forcément celui qu'on souhaiterait ; la seule chose garantie est que chaque clé sera affichée une fois).

4.1 Simulation de lois de probabilités

Voici la fonction `testeDe6(n)`.

```
def testDe6(n):
   freq={}
   for i in range(n):
      resultat = simuleDe6()
      if resultat in freq :
         freq[resultat]+=1
      else:
         freq[restulat]=1
   for valeur in freq:
      freq[valeur] = freq[valeur]/n
   return freq
```

Cette fonction retourne un dictionnaire contenant les *fréquences empiriques* d'apparition des valeurs de 1 à 6 après n lancers d'un dé, calculées avec l'algorithme suivant :

— Initialiser un dictionnaire vide.

— Effectuer n lancers ; pour chaque lancer, si le résultat est déjà une clé présente dans le dictionnaire, incrémenter de 1 la valeur associée ; sinon, ajouter la clé et lui associer la valeur 1.

— Avant de retourner, normaliser, c'est-à-dire diviser toutes les valeurs (nombres d'occurrences des différentes clés) par n pour obtenir des valeurs dont la somme soit 1.

La valeur retournée de `testeDe6(5)` pourrait donc ressembler à { `1 : 0.2, 2 : 0.4, 3 : 0.2, 6 : 0.2`} (ici les clés sont les résultats possibles de l'expérience, et les valeurs, les fréquences observées, sous formes de flottants)

0. Tester la fonction `testeDe6(n)` avec des valeurs de n comme $10, 100, 1000, 1000000$, chacune à plusieurs reprises. Commenter les résultats obtenus, notamment en ayant en tête la vision "fréquentiste" des probabilités.

1. Écrire une fonction `simulePremierSix()` qui simule le lancer répété d'un dé et retourne le nombre total de lancers nécessaires pour obtenir le résultat 6 pour la première fois. Quelle est la loi de probabilité théorique associée à cette expérience ?
2. Écrire une fonction `testePremierSix(n)` comparable à la fonction `testeDe6()`, mais pour l'expérience consistant à attendre le premier six, à répéter `n` fois ; comme pour la fonction `testeDe6()`, expérimenter avec des valeurs de `n` de 10, 100, 1000 et 1000000 et commenter les résultats obtenus.
3. Faire la même chose avec l'expérience correspondant à lancer deux dés, et à prendre la somme des deux dés : écrire une fonction `simuleDeuxDes()` qui simule cette opération, et une fonction `testeDeuxDes(n)` qui retourne un dictionnaire contenant les fréquences observées des différents résultats possibles pour n tirages. Quelles sont les fréquences que vous vous attendez à observer ?

Il est évident maintenant que les définitions de toutes les fonctions de test se ressemblent beaucoup. La seule différence est le simulateur utilisé pour obtenir les résultats.

Il va de soi qu'il vaut mieux définir une fonction de test générique, à laquelle on peut passer le simulateur en paramètre.

4. Écrire une fonction `frequencesEmpiriques(sim,n)` qui, étant donné un simulateur `sim` et un entier n, retourne le dictionnaire contenant les fréquences empiriques des résultats observé à l'issue de n appels à `sim`. Tester votre fonction, avec comme simulateur `simuleDe6`, `simulePremierSix` et `simuleDeuxDes`.

4.2 Simulateurs génériques

Nous allons définir des fonctions permettant de générer des simulateurs des variables aléatoires suivant des lois usuelles. Une telle fonction va donc retourner une fonction :

```
def genereBernoulli(p):
   def sim():
      if random.random()<p:
         return 1
      else:
         return 0
   return sim
```

qui peut par la suite être stockée dans une variable, et utilisée lors des testes comme ci :

```
monSimulateur = genereBernoulli(0.1)
observations = frequencesEmpiriques(monSimulateur,1000000)
```

Ici on profite du fait que `random.random()` retourne un réel compris entre 0 et 1 selon une loi uniforme – il a probabilité p d'être inférieur à p.

0. Observez les fréquences empiriques des variables de Bernoulli pour diverses valeurs de p, comme 0.001, 0.01, 0.1, 0.5, 0.9, 0.99, 0.999.
1. Écrire un générateur `genereBinomiale(k,p)`, qui retourne un simulateur de la loi binomiale de paramètres (k, p). (Indication : pour simuler une variable binomiale de paramètres (k, p), il suffit de faire la somme de k variables de Bernoulli de paramètre p.)

   Observez les fréquences empiriques pour diverses combinaisons de k, p, et n.

2. Écrire un générateur `genereGeometrique(p)`, qui retourne un simulateur de la loi géométrique de paramètre p. (Indication : pour simuler une variable géométrique, il suffit de simuler une suite de variables Bernoulli de paramètre p jusqu'à la première apparition d'un 1, et retourner le nombre total de répétitions.)

   Observez les fréquences empiriques pour diverses combinaisons de p et n.

4.2.1 Poisson

Les constructions précédentes ne permettent pas d'obtenir simplement des simulateurs pour des variables aléatoires qui n'ont pas de représentation simple en termes d'expériences de variables aléatoires indépendantes "plus simples", comme la loi de Poisson (une telle représentation existe, mais c'est un peu compliqué à voir).

1. Vous allez donc écrire une fonction qui retourne un *simulateur* à partir d'un dictionnaire contenant une loi, `genereSimulateur(loi)`. Comme pour les dictionnaires de fréquences que vous avez produits jusqu'ici, les clés du dictionnaire de loi seront les valeurs prises par la variable aléatoire simulée, et les valeurs associées seront les probabilités respectives des valeurs.

   L'algorithme de simulation est le suivant : on fait un (et un seul !) tirage d'un réel x entre 0 et 1, avec la fonction `random.random()` ; puis, on passe en revue (dans un ordre quelconque – on pourrait chercher à optimiser l'ordre, mais c'est une autre question) les clés :
   - si x est inférieur à la probabilité de la première clé, on retourne la première clé ;
   - si x est compris entre la probabilité de la première clé, et la somme des probabilités des deux premières clés, on retourne la deuxième clé ;
   - si x est compris entre la somme des probabilités des deux premières clés, et la somme des probabilités des trois premières, on retourne la troisième clé ;
   - plus généralement, si x est compris entre la somme des probabilités des k premières clés, et la somme des probabilités des $k+1$ premières, on retournera la $k+1$-ème clé.

   Par exemple, `genereSimulateur({'a':1/2,'b':1/3,'c':1/6})` devrait retourner un simulateur, qui lui retournerait la lettre `'a'` avec probabilité $1/2$, la lettre `'b'` avec probabilité $1/3$ et la lettre `'c'` avec probabilité $1/6$. Pour le faire, le simulateur tirera un réel x entre 0 et 1. Si $0 \leq x < 1/2$, il retournera `'a'` ; sinon, si $x < 1/2 + 1/3$, il retournera `'b'` ; sinon, si $x < 1/2 + 1/3 + 1/6$, il retournera `'c'`.

2. Écrire une fonction `loiBinomiale(k,p)` qui retourne la loi binomiale avec paramètres (k, p) sous forme d'un dictionnaire. Vous pouvez profiter de la relation

$$\mathbb{P}(X = i+1) = \frac{p}{1-p} \cdot \frac{k-i}{i+1} \cdot \mathbb{P}(X = i)$$

   pour tout $i = 0, \ldots, k-1$, où X est une variable aléatoire binomiale avec paramètres (k, p).

   Au final, `genereSimulateur(loiBinomiale(k,p))` devrait retourner un simulateur équivalent à celui retourné par `genereBinomiale(k,p)`.

   Observez s'il y a une différence entre
   `frequencesEmpiriques(genereSimulateur(loiBinomiale(k,p)),n)`
   et

`frequencesEmpiriques(genereBinomiale(k,p),n)`
pour diverses valeurs de k, p, et n.

3. Pour simuler les lois de Poisson, il reste une difficulté : ces lois prennent n'importe quelle valeur entière, il faudrait donc préparer un dictionnaire avec une infinité de clés... Vous allez contourner le problème en préparant un dictionnaire approché, en utilisant la propriété suivante (vue en TD) : pour une variable aléatoire X suivant une loi de Poisson de paramètre x, la suite des probabilités $(\mathbb{P}(X = k))_{k\geq 0}$ est croissante jusqu'à $k = \lfloor x \rfloor$, et décroissante au-delà de $k = \lceil x \rceil$. Vous calculerez donc un dictionnaire pour la "loi de Poisson tronquée" : toutes les valeurs au-delà d'un certain k_{max} seront remplacées par k_{max}, avec la probabilité correspondante. L'indice k_{max} pourra être pris comme le plus petit entier $k > x$ tel que la probabilité $\mathbb{P}(X \geq k)$ soit inférieure à une limite suffisamment petite, par exemple 10^{-8}.

   Vous écrirez donc une fonction `genereePoisson(x)`, qui retourne un simulateur (approché) pour la loi de Poisson de paramètre x. **Indication :** la fonction exponentielle est `math.exp` ; vous aurez besoin d'un `import math` au début de votre fichier. Pour calculer les probabilités, mieux vaut procéder de proche en proche : si on note p_k la probabilité que la variable de Poisson prenne la valeur k, on a $p_0 = e^{-x}$, et une relation simple (à trouver !) permet de calculer p_k à partir de k et de p_{k-1}, sans autre appel à la fonction exponentielle ni calcul direct de factorielle.

4.3 Expériences autour de la loi de Poisson

Une des raisons pour lesquelles la loi de Poisson est très utilisée en modélisation, est la suivante (qu'il n'est pas question de démontrer dans ce cours) : si on a un grand nombre de variables aléatoires de Bernoulli indépendantes, toutes de petits paramètres (mais la somme des paramètres, elle, peut ne pas être petite), alors leur somme "ressemble beaucoup" à une variable de Poisson dont le paramètre serait la somme des paramètres des Bernoulli.

Vous pouvez en faire l'expérience avec des variables binomiales (équivalentes à des sommes de Bernoulli) : faites simuler, par exemple, des binomiales de paramètres $(1000, 0.01)$ et des Poisson de paramètre 10, et comparez les dictionnaires ; ou des binomiales de paramètres $(1000, 0, 005)$ et des Poisson de paramètre 5...

Une autre expérience facile à réaliser est la suivante : une propriété des lois de Poisson (vue en TD) est que la somme de deux variables de Poisson indépendantes suit également une loi de Poisson (dont le paramètre est la somme des paramètres). Vous pouvez facilement réaliser des dictionnaires de fréquences pour la somme de deux Poisson et les comparer à une Poisson du paramètre correspondant.