Simulation de variable aléatoire

Ce travaux pratique est divisé en deux parties principales, une pour NumPy et une pour SciPy. Les deux packages contiennent des fonctions importantes pour la simulation, les distributions de probabilités et les statistiques.

1. Simualation avec le module random de Numpy

1. Simuler des variables aléatoires

1.1 Générateurs de nombres aléatoires de base

Les générateurs de nombres aléatoires NumPy sont tous stockés dans le module `numpy.random`. Pour utilser le module numpy vous devez commencer par importer le package numpy avec la commande suivante :

```
1 import numpy as np
```

L'accés aux fonctions du module `random` peut se faire via le syntaxe `np.random.nom_fonction`

`np.random.rand, np.random_sample`

`np.random.rand` et `np.random.random_sample` sont des générateurs de nombres aléatoires uniformes qui sont identiques. `random_sample` est la fonction NumPy préférée, et `np.random.rand` est une fonction pratique principalement destinée aux utilisateurs de MATLAB.

```
1 x = np.rand(3,4,5)
2 y = np.random_sample((3,4,5))
```

`np.random.randn, np.random.standard_normal`

`np.random.randn` et `np.random.standard_normal` sont des générateurs de nombres aléatoires suivant une loi normale centrée réduite. `np.random.standard_normal` est la fonction NumPy préférée, et `np.random.randn` est une fonction pratique principalement destinée aux utilisateurs de MATLAB.

```
1 np.randn(3,4,5)
2 np.standard_normal((3,4,5))
```

`np.random.randint, np.random.random_integers`

`np.randint` et `np.random_integers` sont des générateurs de nombres aléatoires entiers uniformes qui prennent 3 entrées: `low`, `high` et `size`. `np.randint` et `np.random_integers` diffèrent par le fait que `randint` génère des entiers exclus de la valeur `high` (comme la plupart des fonctions Python), tandis que `np.random_integers` inclut la valeur `high` dans sa plage.

```
>>> x = randint(0,10,(100))
>>> x.max() # Is 9 since range is [0,10)
>>> y = random_integers(0,10,(100))
>>> y.max() # Is 10 since range is [0,10]
```

1.2 Fonctions de vecteurs aléatoires

`suffle`

La fonction `shuffle` ré-ordonne de manière aléatoire les éléments d'un array.

```
>>> x = arange(10)
>>> shuffle(x)
>>> x
```

`permutation`

La fonction `permutation` retourne les éléments réorganisés de manière aléatoire d'un tableau sous forme de copie sans modifier directement l'entrée.

```
>>> x = arange(10)
>>> permutation(x)
>>> x
```

1.3 Sélectionner des générateurs de nombres aléatoires

NumPy fournit un grand choix de générateurs de nombres aléatoires pour une distribution spécifique. Tous prennent entre 0 et 2 arguments requis qui sont des paramètres de la distribution, plus un tuple contenant la taille de la sortie. Tous les générateurs de nombres aléatoires sont dans le module `numpy.random`.

`bernoulli`

Il n'y a pas de générateur de variable aléatoire de Bernoulli. Utiliser plutôt `binomial(1, p)` pour générer un seul tirage ou `binomial(1, p, (10,10))` pour générer un tableau où p est la probabilité de succès.

`beta`

`beta(a,b)` génère des réalisation d'une loi $\beta(a,b)$β(a,b). `beta(a, b,(10, 10))` génère un tableau 10×1010×10 de la même loi.

`binomial`

`binomial(n, p)` génère un tirage à partir de la distribution $B(n,p)$B(n,p). `binomial(n, p, (10,10))` génère un tableau de 10×1010×10 de suivant la même loi.

`chisquare`

`chisquare(nu)` génère une loi du Khi-deux $\chi 2 \nu$χν2, où νν est le dégré de liberté. `chisquare(nu, (10, 10))` génère un tableau de 10×1010×10 de suivant la même loi.

`exponential`

`exponential()` génère un tirage à partir de la distribution exponentielle de paramètre λ=1λ=1. `exponential(lambda, (10,10))` génère un tableau de 10×1010×10 tirages à partir de la même loi.

`f`

`f(v1, v2)` génère une loi de Fisher de paramètres ν1ν1 et ν2ν2.

`gamma`

`gamma(a)` génère une loi Γ(a,1)Γ(a,1) où aa est le paramètre de forme. `gamma(a, theta, (10, 10))` génère une loi Γ(a,θ)Γ(a,θ) où θθ est une paramètre d'échelle.

`laplace`

`laplace(loc, scale, (10, 10))` génère une loi de Laplace (double exponentielle) de paramètre de position `loc` et d'échelle `scale`.

`lognormal`

```
lognormal(mu, sigma, (10, 10))
```

`multinomial`

```
multinomial(n, p, (10, 10))
```

`multivariate_normal`

`multivariate_normal(mu, Sigma, (10,10))` attention à la dimension

`negative_binomial`

```
negative_binomial(n, p, (10, 10))
```

`normal`

```
normal(mu, sigma, (10,10))
```

`poisson`

```
poisson(lambda, (10, 10))
```

`standard_t`

```
standard_t(nu, (10, 10))
```

`uniform`

```
uniform(low, high, (10, 10))
```

2 Simulation et génération de nombres aléatoires

Les nombres aléatoires simulés par ordinateur sont généralement construits à partir de fonctions très complexes mais finalement déterministes. Parce qu'ils ne sont pas réellement aléatoires, les nombres aléatoires simulés sont généralement qualifiés de pseudo-aléatoires. Tous les nombres pseudo-aléatoires de NumPy utilisent un générateur de nombres aléatoires basé sur Mersenne Twister, un générateur capable de produire une très longue série de données pseudo-aléatoires avant de répéter (jusqu'à 219937−1219937−1 valeurs non répétitives).

`RandomState`

`RandomState` est la classe utilisée pour contrôler les générateurs de nombres aléatoires. Plusieurs générateurs peuvent être initialisés par `RandomState`.

In [ ]:

```
gen1 = np.random.RandomState()
gen2 = np.random.RandomState()
gen1.uniform() # Generate a uniform


state1 = gen1.get_state()
gen1.uniform()

gen2.uniform() # Different, since gen2 has different seed

gen2.set_state(state1)
gen2.uniform() # Same uniform as gen1 after assigning state
```

2.1 `state`

Les générateurs de nombres pseudo-aléatoires suivent un ensemble de valeurs appelé *état*. L'état est généralement un vecteur qui a la propriété que si deux instances du même générateur de nombres pseudo-aléatoires ont le même état, la séquence des nombres pseudo-aléatoires générés sera identique. L'état du générateur de nombres aléatoires par défaut peut être lu à l'aide de numpy.random.get_state et peut être restauré à l'aide de `numpy.random.set_state`.

In [ ]:

```
st = get_state()
randn(4)

set_state(st)
randn(4)
```

Les deux séquences sont identiques car leur état est le même lorsque `randn` est appelé. L'état est un tuple de 5 éléments, le deuxième élément étant un vecteur 625×1625×1 d'entiers non signés sur 32 bits. En pratique, l'état ne devrait être stocké qu'en utilisant `get_state` et restauré en utilisant `set_state`.

`get_state`

`get_state()` obtient l'état actuel du générateur de nombres aléatoires, qui est un tuple de 5 éléments. Il peut être appelé en tant que fonction, auquel cas il obtient l'état du générateur de nombres aléatoires par défaut, ou en tant que méthode sur une instance particulière de `RandomState`.

`set_state`

`set_state(state)` définit l'état du générateur de nombres aléatoires. Il peut être appelé en tant que fonction, auquel cas il définit l'état du générateur de nombres aléatoires par défaut, ou en tant que méthode sur une instance particulière de `RandomState`. set_state ne devrait généralement être appelé qu'à l'aide d'un tuple d'état renvoyé par `get_state`.

2.2 `Seed`

`numpy.random.seed` est une fonction utile pour initialiser le générateur de nombres aléatoires et peut être utilisée de deux manières. `seed()` initialisera (ou réinitialisera) le générateur de nombres aléatoires en utilisant des données aléatoires réelles fournies par le système d'exploitation. `seed(s)` prend un vecteur de valeurs (peut être scalaire) pour initialiser le générateur de nombres aléatoires à un état particulier. `seed(s)` est particulièrement utile pour produire des études de simulation reproductibles. Dans l'exemple suivant, les appels à `seed()` génèrent différents nombres aléatoires, car ceux-ci se réinitialisent à l'aide de données aléatoires de l'ordinateur, tandis que les appels à `seed(0)` produisent la même séquence de nombres aléatoires.

In [ ]:

```
# test 1
seed()
randn()

# test 2
seed()
randn()

# test avec graine fixée
seed(0)
randn()


# test avec graine fixée
seed(0)
randn()
```

NumPy appelle toujours `seed()` lorsque le premier nombre aléatoire est généré. Appeler `standard_normal()` au cours de deux «nouvelles» sessions ne produira pas le même nombre aléatoire.

`seed`

`seed(valeur)` utilise *valeur* pour amorcer le générateur de nombres aléatoires. `seed()` prend des données aléatoires réelles du système d'exploitation lors de l'initialisation du générateur de nombres aléatoires (par exemple, `/dev/random` sous Linux ou `CryptGen-Random` sous Windows).

7.2.3 Réplication de données simulées

Il est important d'avoir des résultats reproductibles lors d'une étude par simulation. Il existe deux méthodes pour y parvenir:

- Appeler `seed()` puis `state = get_state()` et enregistrer `state` dans un fichier qui pourra ensuite être chargé ultérieurement lors de l'exécution de l'étude de simulation.
- Appeler `seed(s)` au début du programme (où `s` est une constante).

L'une ou l'autre de ces méthodes permettra d'utiliser la même séquence de nombres aléatoires.

Attention :

Ne pas *sur-initialiser* les générateurs de nombres pseudo-aléatoires. Les générateurs doivent être initialisés une fois par session, puis autorisés à produire la séquence pseudo-aléatoire. Réinitialiser à plusieurs reprises les générateurs de nombres pseudo-aléatoires produira une séquence nettement moins aléatoire que celle que le générateur avait été conçu.

Simulation sur plusieurs machines en parallèle

Les études de simulation conviennent parfaitement à la parallélisation, bien que le code parallèle rend la reproductibilité plus difficile.

3 Fonctions statistiques

mean

`mean` calcule la moyenne d'un tableau. Un deuxième argument optionnel fournit l'axe à utiliser (par défaut, utiliser un tableau entier). `mean` peut être utilisée soit comme fonction, soit comme méthode sur un tableau comme suit :

```
1  import numpy as np
2  x = np.arange(10.0)
3
4  x.mean()
5
6  np.mean(x)
7
8  x=np.reshape(np.arange(20.0),(4,5))
9
10 np.mean(x,0)
11
12 x.mean(1)
```

median

`media` calcule la valeur médiane dans un tableau. Un deuxième argument optionnel fournit l'axe à utiliser (par défaut, utiliser un tableau entier).

```
x=np.random.randn(4,5)
print(x)

print(np.median(x))

print(np.median(x, 0))
```

std

`std` calcule l'écart type d'un tableau. Un deuxième argument optionnel fournit l'axe à utiliser (par défaut, utiliser un tableau entier). `std` peut être utilisée comme fonction ou comme méthode sur un tableau.

var

`var` calcule la variance d'un tableau. Un deuxième argument optionnel fournit l'axe à utiliser (par défaut, utiliser un tableau entier). `var` peut être utilisée soit comme fonction, soit comme méthode dans un tableau.

corrcoef

`corrcoef(x)` calcule la corrélation entre les lignes d'un tableau à 2 dimensions *x*x. `corrcoef(x,y)` calcule la corrélation entre deux vecteurs à une dimension. Un mot-clé facultatif, l'argument `rowvar`, peut être utilisé pour calculer la corrélation entre les colonnes de l'entrée - c'est `corrcoef(x, rowvar=False)` et `corrcoef(x.T)` sont identiques.

In [ ]:

```
np.x = np.random.randn(3,4)
np.corrcoef(x)
np.corrcoef(x[0],x[1])
np.corrcoef(x, rowvar=False)
np.corrcoef(x.T)
```

cov

`cov(x)` calcule la covariance d'un tableau *x*x. `cov(x,y)` calcule la covariance entre deux dimensions à une dimension vecteurs. Un mot-clé facultatif, l'argument `rowvar`, peut être utilisé pour calculer la covariance entre les colonnes de l'entrée. Il s'agit de `cov(x, rowvar=False)` et de `cov(x.T)` sont identiques.

histogram

`histogram` peut être utilisée pour calculer l'histogramme (fréquence empirique, en utilisant *k*k classes) d'un ensemble de données. Un second argument optionnel fournit le nombre de classes. La valeur par défaut de *k*k est 1010. L'histogramme renvoie deux sorties, la première avec un vecteur k-élément contenant le nombre d'observations dans chaque classe et la seconde avec les *k*+1k+1 extrémités des *k*k classes.

In [ ]:

```
x = np.random.randn(1000)
count, binends = np.histogram(x)
count
```

histogram2d

`histogram2d` calcule un histogramme à 2 dimensions pour les vecteurs à 1 dimension. Un argument facultatif `bins` indique le nombre de classes à utiliser. `bins` peuvent contenir soit un seul entier scalaire, soit une liste ou un tableau à 2 éléments contenant le nombre de classes à utiliser dans chaque dimension.

SciPy

`SciPy` fournit une gamme étendue de générateurs de nombres aléatoires, de distributions de probabilités et de tests statistiques.

In [14]:

```
import scipy
import scipy.stats as stats
```

4 Variables aléatoires continues

SciPy contient un grand nombre de fonctions liées aux variables aléatoires continues. Chaque fonction est issue de sa propre classe (par exemple, `norm` pour la loi normale ou `gamma` pour Gamma), et

les classes exposent des méthodes pour la génération de nombres

aléatoires, le calcul des formats PDF (densité), CDF (répartition) et quantile, l'ajustement des paramètres à l'aide de MLE et le calcul de divers moments. Les méthodes sont répertoriées ci-dessous, où `dist` est un espace générique pour le nom de la distribution dans SciPy. Alors que les fonctions disponibles pour les variables aléatoires continues varient dans leurs entrées, toutes prennent 3 arguments génériques:

- `*args` un ensemble d'arguments non mot-clé spécifiques à la distribution. Ceux-ci doivent être entrés dans l'ordre indiqué dans la classe docstring. Par exemple, lorsque vous utilisez une distribution *F*F, deux arguments sont nécessaires, l'un pour le degré de liberté du numérateur et l'autre pour le degré de liberté du dénominateur.
- `loc` un paramètre de position qui détermine le centre de la distribution.
- `scale` un paramètre d'échelle, qui détermine la mise à l'échelle de la distribution. Par exemple, si *z*z est une loi normale centrée réduite, alors *s*×*z*s×z est une loi normal mise à l'échelle.

dist.rvs

Génération de nombres pseudo-aléatoires. Généralement, on appelle `rvs` en utilisant `dist.rvs(*args, loc=0, scale=1, size=size)` où `size` est un tuple à *n*n éléments contenant la taille du tableau à générer.

dist.pdf

Évaluation de la fonction de densité de probabilité pour un tableau de données (élément par élément). Généralement, pdf est appelé à l'aide de `dist.pdf (x, * args, loc = 0, scale = 1)`, où `x` est un tableau contenant les valeurs à utiliser lors de l'évaluation de la densité.

dist.logpdf

Évaluation du log de la fonction de densité de probabilité pour un tableau de données (élément par élément). Généralement, `logpdf` est appelé à l'aide de `dist.logpdf (x, * args, loc = 0, scale = 1)` où `x` est un tableau contenant les valeurs à utiliser lors de l'évaluation du log de la densité de probabilité.

dist.cdf

Évaluation de la fonction de répartition sur un tableau de données (élément par élément). Généralement, cdf est appelé à l'aide de `dist.cdf(x, *args, loc=0, scale=1)`, où `x` est un tableau contenant les valeurs à utiliser lors de l'évaluation de CDF.

dist.ppf

Évaluation de l'inverse de la fonction de réparation (également appelée fonction quantile) sur un tableau de valeurs comprises entre 00 et 11. Généralement, `ppf` est appelée à l'aide de `dist.ppf(p, *args, loc=0, scale=1)` où `p` est un tableau avec tous les éléments compris entre 00 et 11 contenant les valeurs à utiliser lors de l'évaluation de l'inverse de la fonction de répartition.

dist.fit

Estimation des paramètres `shape`, `location` et `scale` à partir des données par maximum de vraisemblance. Généralement, `fit` est appelée à l'aide `dist.fit(data, *args, floc=0, fscale=1)` où `data` est un tableau de données utilisé pour estimer les paramètres. `floc` force le paramètre `location` à une valeur particulière (par exemple, `floc = 0`). `fscale` permet de forcer le paramètre `scale` à une valeur particulière (par exemple, `fscale=1`). Il est nécessaire d'utiliser `floc` et / ou `fscale` lors du calcul des MLE si la distribution n'a pas de position et / ou d'échelle. Par exemple, la distribution gamma est définie à l'aide de 22 paramètres, souvent appelés forme et échelle. Afin d'utiliser ML pour estimer les paramètres d'une gamma, il faut utiliser `floc=0`

dist.median

Renvoie la médiane de la distribution. Généralement, la médiane est appelée en utilisant `dist.median(*args, loc=0, scale=1)`.

dist.mean

Renvoie la moyenne de la distribution. Généralement, la moyenne est appelée avec `dist.mean(*args, loc=0, scale=1)`.

dist.moment

Évaluation du moment non-centré d'ordre *n*n de la distribution. En général, `dist.moment(r, *args, loc=0, scale=1)` où `r` est l'ordre du moment à calculer.

dist.varr

Renvoie la variance de la distribution. Généralement, `var` est appelé en utilisant `dist.var(*args, loc=0, scale=1)`.

dist.std

Renvoie l'écart type de la distribution. Généralement, `std` est appelé en utilisant `dist.std(*args, loc=0, scale=1)`.

4.1 Exemple : gamma

La distribution gamma prend 1 paramètre de forme *a*a (*a*a est le seul élément de `*args`), qui est fixé à 2 dans tous les exemples.

```
import scipy.stats as stats
gamma = stats.gamma

gamma.mean(2), gamma.median(2), gamma.std(2), gamma.var(2)

gamma.moment(2,2) - gamma.moment(1,2)**2 # Variance

gamma.cdf(5, 2), gamma.pdf(5, 2)

gamma.ppf(.95957231800548726, 2)

np.log(gamma.pdf(5, 2)) - gamma.logpdf(5, 2)

gamma.rvs(2, size=(2,2))

gamma.fit(gamma.rvs(2, size=(1000)), floc = 0) # a, 0, shape
```