Feuille de travaux pratiques - Python #1

1 Elements de base en Python

1.1 Array : vecteurs, matrices

Un objet de type `array` est un tableau dont tous les éléments sont du même type. La différence avec les `list` : sa taille n'est pas modifiable une fois créé. Il servira à représenter les vecteurs et les matrices. Il est dans le module `numpy`. Dans la suite, je noterai M pour une matrice, V pour un vecteur, A pour un array quelconque.

```
M=np.array([[0,1],[2,3]]) # matrice
V=np.array([0,1,4,9])# vecteur ligne
```

Dans le module `scipy`, vous trouverez le sous-module `linalg` permettant de faire du calcul matriciel. Voici quelques commandes utiles pour manipuler les `array` :

Pour créer un `array`, connaître sa longueur et accéder à ses éléments :

`np.eye(n)`
`np.ones((n,m))`
`np.zeros((n,m))`
`np.empty((n,m))`
	matrice identité de taille n
	matrice 1 de taille $n \times m$
	matrice nulle de taille $n \times m$
	matrice vide de taille $n \times m$

`np.arange(debut, fin(non inclus), pas)`	le tableau des entiers de debut à fin avec un pas
`np.linspace(min, max, nb de points)`	points réguliers $[min, ..., max]$
`np.diag(V)`	matrice avec en diagonale V
`V[-1]`	dernier élément du vecteur V
`V[start:end:step]`	pour extraire une sous-liste
`M[i,j]`	élément à l'indice $(i_{ligne}, j_{colonne})$
`A.shape`	renvoie (nb de lignes, nb de colonnes)
`len(V)`	renvoie la longueur du vecteur
`A.reshape([n,m])`	redimensionne le tableau A en n lignes et m colonnes. Si l'une des valeurs vaut -1, `numpy` la trouve tout seul.
`V.reshape([nbLign,1])` ou `V[:,np.newaxis]`	transforme un vecteur ligne en colonne

Modification et opérations sur les array :

Faire des opérations sur des `array` de tailles différentes, s'appelle du broadcasting. Cela fait les opérations terme à terme :

`A+10`	ajoute terme à terme
`M+V`	ajoute v à chaque ligne de M
`A*A`	multiplication terme à terme
`V*M`	répète v selon l'autre dimension pour avoir la même taille que M avant de multiplier terme à terme
`1/A`	inverse terme à terme

N.B. *De manière générale, quand vous n'êtes pas sûr du résultat que donne un code (ex : opération terme à terme ou non), testez sur un exemple.*

Les opérations usuelles sont les suivantes :

`A.dot(B)`	multiplication matricielle : matrice/matrice, matrice/vecteur, vecteur/vecteur (produit scalaire)
`np.linalg.inv(M)`	inverse matriciel
`M.T`	transposée
`np.linalg.det(M)`	déterminant
`V[0:3]=5`	remplace les 3 premiers éléments de V par des 5.
`A>0`	renvoie un `array` de même taille que A avec des booléens.
`A[A>0]`	renvoie un `array` avec les éléments de A qui vérifient la condition Ici ça fait une copie et non une référence. Pour les conditions, on utilise `&` pour "et" et `|` pour "ou"

`np.sum(A)`
`np.sum(A,axis=0)`
`np.sum(A,axis=1)`
	somme des éléments de A
	pour sommer selon les colonnes,
	selon les lignes

`np.cumsum(A)`	somme cumulative des éléments de A, on peut encore utiliser `axis`
`np.mean(A)`	moyenne des éléments de A, on peut encore utiliser `axis`
`np.std(A)`	écart-type des éléments de A, on peut encore utiliser `axis`

`eigVal,eigVec=np.linalg.eig(M)`
 	
`eigVal,eigVec=np.linalg.eigh(M)`
	valeurs propres (avec multiplicité) et vecteurs propres à droite, en colonne, normalisés et dans le même ordre.
	pour les matrices symétriques, hermitiennes

1.2 Tracer des graphes

Pour tracer des graphes, nous avons besoin du sous module `pyplot` de `matplotlib`. Ensuite, on définit la figure, qu'on appelle en général `fig`. Puis dans la figure, on définit les sous-figures, selon les axes. On va utiliser la fonction `plot` qui prend en arguments des `array`.

```
fig, axs=plt.subplots(n,m) # on aura des sous-figures sur n lignes et m colonnes
#sans argument quand on n'en veut qu'un.
axs[i,j].plot(x,y,"r", label="nom fonction") # trace dans le graphique (i,j),
#en couleur rouge, et associe une legende
fig.suptitle("titre principal")
```

```
axs[0,0].set_title("titre graphe 1")
axs[0,1].set_title(r"$math latex$") # le 'r' dit à Python de ne pas interpreter
#les caracteres speciaux
#Ne pas oublier:
axs.legend() # affiche les légendes
plt.show() # pour ouvrir la fenetre graphique
plt.clf() #efface la fenêtre graphique
```

N.B. *Le troisième argument de* `plot` *est le style du trait. On mettra "o" pour avoir des points non reliés.*

D'autres commandes :

`axs[..].set_ylabel("titre ordonnees")`	donne un titre aux ordonnées
`axs[..].set_ylim([a,b])`	limite le graphe aux ordonnées $[a, b]$
`ax[..].set_xticks(rray)`	graduation axe des abscisses
`ax[..].set_sticklabels(["0", "1",...])`	noms des graduations de l'axe des abscisses

Diagramme en bâtons : histogrammes de lois discrêtes

`ax.bar(x,y,width=...)` à la place de `plot`.

Histogramme

`plt.hist(X,bins=n)` répartit les données de X dans n sous intervalles de $[min(X), max(X)]$. `hist` prend la place de `plot`, on garde la même syntaxe que précédemment pour faire différents graphes par exemple. Mais pour mettre plusieurs histogrammes sur le même on fait `plt.hist([X1,X2,X3], bins=n, label=...)`.

N.B. *Avec* `a=plt.hist(X,bins=n)`*, on récupère en premier la hauteur des bâtons et en deuxième le découpage des sous-intervalles fait par Python.*

N.B. *On peut donner à* `bins` *ses propres sous-intervalles dans un* `array`*.*

Pour superposer avec une densité, il faut demander à Python de normaliser les histogrammes, ça se fait avec l'option de `hist` : `density=True`. Puis on plote la densité.

1.3 Aléatoire

`Scipy` contient un sous-module `stats`.
On conseille de l'importer sous le format `import scipy.stats as stats`
À chaque loi de probabilité sont associées des commandes :

- `rvs` : simulation
- `pdf(x, parametre)` : probability density function i.e. densité en x
- `pmf(k, parametre)` : probability mass function i.e. densité discrête en k
- `cdf(x,parametre)` : cumulative density function i.e. fonction de répartition en x
- `ppf(q, parametre)` : percent point function i.e. q-ième quantile.

On les utilisera comme ceci : `stats.norm.rvs(parametres)`. Les paramètres sont (paramètres de forme, loc=μ, scale=σ, size=taille échantillon). Pour créer un vecteur aléatoire (`array`) de taille $n \times p$, on donnera l'argument `size=(n,p)`. Les paramètres μ et σ sont tels que si l'appel de `stats.xxx.rvs()` renvoie une v.a X alors `stats.xxx.rvs(loc=mu,scale=sigma)` renvoie une v.a ayant la même loi que $\sigma X + \mu$. En tapant `stats.expon?` ou `help(stats.expon)`, vous aurez la normalisation choisie pour la loi exponentielle par exemple. Seuls les paramètres de forme sont obligatoires.

loi(paramètres de forme)	Lois
`norm()`	Loi normale $\mathcal{N}(0,1)$.
`gamma(a=)`	Loi gamma $\gamma(a,1)$.
`beta(a=, b=)`	Loi béta $\beta(a,b)$.
`t(df=n)`	Loi Student de n degrés de liberté.
`cauchy()`	Loi de Cauchy.
`expon(scale=1/lambda)`	Loi exponentielle $\mathcal{E}(\lambda)$.
`uniform(loc=, scale=)`	Loi uniforme $\mathcal{U}(]loc, loc+scale[)$.
`randint(low, high)`	Loi uniforme $\mathcal{U}(\{low, low+1, \cdots, high-1\})$.
`binom(n=, p=)`	Loi binomiale $\mathcal{B}(n,p)$.
`geom(p=)`	Loi géométrique $\mathcal{G}(p)$.
`poisson(mu=)`	Loi de Poisson $\mathcal{P}(\mu)$.

2 Échauffement

On commence par s'entraîner à faire quelques manipulations classiques sur les variables Gaussiennes.

1. Générer un vecteur X comprenant 1000 réalisations i.i.d. d'une loi $\mathcal{N}(-1,4)$.
2. Calculer la moyenne et l'écart type des valeurs de X et vérifier que le résultat obtenu est cohérent par rapport aux paramètres de la loi normale.
3. Tracer un histogramme des valeurs de X et, sur le même graphe, tracer la densité de la loi $\mathcal{N}(-1,4)$.

3 Exemple d'estimation de la loi d'un jeu de données

On considère un échantillon $X_1, \cdots, X_n$ d'une loi inconnu que l'on cherche à déterminer. On trouvera une réalisation d'un tel échantillon ici.

1. Copier les données dans un fichier interne et créer un vecteur X contenant cet échantillon à l'aide de la fonction *loadtxt* de *numpy*, créer un vecteur X contenant cet échantillon. Quelle est la taille n de cet échantillon ?
2. Tracer l'histogramme de X. A votre avis, à quelle famille de loi appartient X ?

On rappelle que la fonction de répartition empirique F_n est la fonction constante par morceaux définie par

$$F_n(t) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{X_i \leq t\}$$

3. A l'aide de la fonction *step* de *matplotlib.pyplot*, tracer la fonction de répartition empirique associée à X.
4. Sur le même graphe, tracer la fonction de répartition de la loi que vous pensez correspondre aux données en choisissant correctement son/ses paramètres(s). Qu'en pensez-vous ?

4 Estimation de la moyenne d'une Gaussienne

Par la moyenne empirique

On considère un échantillon $X_1, \cdots, X_n$ d'une loi $\mathcal{N}(m, 4)$ de paramètre inconnu $m \in \mathbb{R}$ à estimer. On trouvera une réalisation d'un tel échantillon ici.

1. A l'aide de la fonction *loadtxt* de *numpy*, créer un vecteur X contenant cet échantillon. Quelle est la taille n de cet échantillon ?
2. On cherche à estimer m par l'estimateur de la moyenne empirique $\hat{m}_1 = \overline{X}_n = \frac{1}{n}\sum_{1\leq i\leq n} X_i$. Calculer cette quantité sur les données de la question précédente.
3. Quelle est l'erreur quadratique moyenne $\mathbb{E}(\hat{m}_1 - m)^2]$ de l'estimateur $\hat{m}_1$.

Par la médiane

Soit $v = (v_1, \cdots, v_n)$, on note $(v_{(1)}, \cdots, v_{(n)})$ le n-uplet réordonné par ordre croissant et on rappelle que la médiane de v est définie par

$$\text{med}(v) = \begin{cases} v_{((n+1)/2)} & \text{si n impair} \\ \frac{1}{2}(v_{(n/2)} + v_{(n/2+1)}) & \text{si n pair} \end{cases}$$

On cherche alors à estimer m par la médiane empirique $\hat{m}_2 = \text{med}(X)$.

1. Créer une fonction **mediane**(v) qui renvoie la médiane d'un n-uplet v et calculer $\hat{m}_2$ pour le jeu de données étudié.

On ne sait pas calculer l'erreur quadratique moyenne exacte pour la médiane empirique. On va alors estimer numériquement cette quantité.

2. Générer 1000 simulations d'échantillon de taille n d'une loi $\mathcal{N}(0, 4)$ et créer un vecteur $S = (S_1, \cdots, S_{1000})$ correspondant aux estimateurs de la médiane de chaque échantillon.
3. On estime alors l'erreur quadratique moyenne de $\hat{m}_2$ par $\frac{1}{1000}\sum_{1\leq i\leq 1000} S_i$. Comparer avec l'erreur quadratique moyenne de $\hat{m}_1$.

Par une méthode tordue

1. (optionnel) Démontrer que si X est une v.a. de loi $\mathcal{N}(m, 4)$ alors

$$m = \log\left(\mathbb{E}\left[e^X\right]\right) - 2.$$

On va donc estimer m par

$$\hat{m}_3 = \log\left(\frac{1}{n}\sum_{i=1}^{n} e^{X_i}\right) - 2$$

2. Calculer $\hat{m}_3$ pour le jeu de données étudié.
3. En s'inspirant de ce qui a été fait précédemment, estimer la variance empirique de $\hat{m}_3$. Comparer.