Alexis PLESSIER

N3 Informatique

Thomas THIBOUT-MORAGLIA

15 Janvier 2021

Rapport

Probabilités et Statistiques

I. 1ère Partie : Régression Linéaire

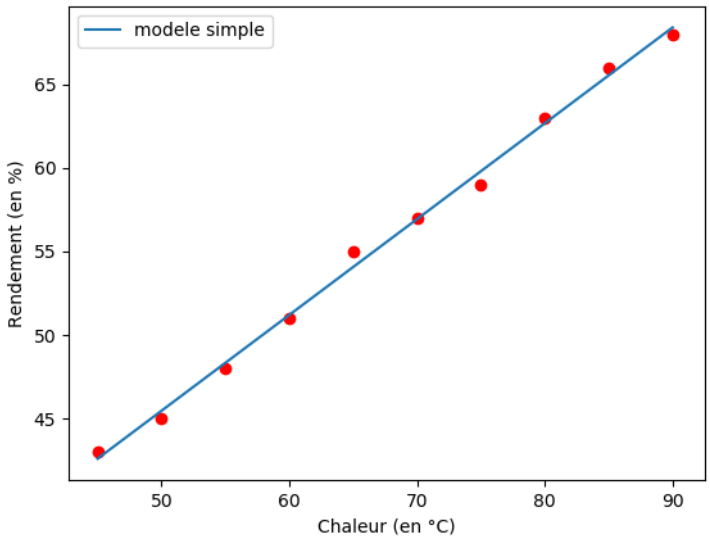
1. Démarche et résultats

a) Modèle simple

Pour calculer les coefficients de la régression Linéaire du jeu de données présent, nous avons implémenté la formule suivante :

$$\beta_1 = \frac{cov(X,Y)}{var(X)} \text{ et } \beta_0 = \bar{y} - \beta_1 * \bar{x}$$

Pour l'équation de droite $y = ax + b$ le résultat des coefficients est $a = 0.5733$ et $b = 16.7999$ et nous avons la droite suivante :



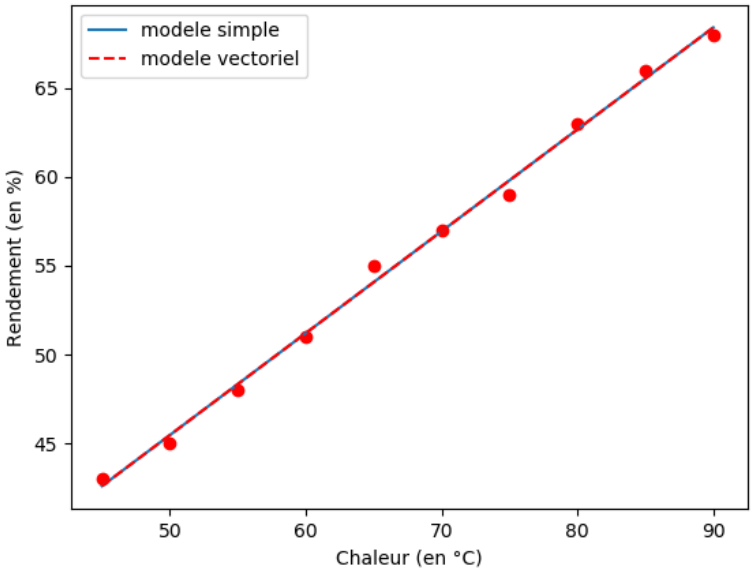
En comparaison avec la fonction *polyfit* de Python, la fonction trouve les valeurs suivantes : $a = 0.5733$ et $b = 16.8000$

b) Modèle vectorielle

Pour calculer les coefficients de régression linéaire par la formule vectorielle avec le jeu de données à disposition, nous sommes tout simplement parti de la matrice suivante :

$$A = \begin{pmatrix} 1 & 45 \\ 1 & 50 \\ 1 & 55 \\ 1 & 60 \\ 1 & 65 \\ 1 & 70 \\ 1 & 75 \\ 1 & 80 \\ 1 & 85 \\ 1 & 90 \end{pmatrix}$$

La formule $(A^T A)^{-1} A^T y$, va finalement nous donner un couple de valeur qui seront nos coefficients. Nous trouvons les valeurs $\boxed{a = 0.5733 \text{ et } b = 16.7999}$ et nous avons la droite suivante (*pointillé rouge*) :



En comparaison avec la fonction *polyfit* de Python, la fonction trouve les valeurs suivantes : $\boxed{a = 0.5733 \text{ et } b = 16.800}$. Comparer à la première méthode, les résultats sont similaires et cela se voit notamment à l'allure des deux courbes.

c) Régression linéaire et descente de gradient

Avec toutes les informations de l'énoncé, nous devons commencer par implémenter les dérivées partielles de β_0 et β_1.

Dérivée partielle en fonction de β_0 : $\frac{1}{n}\sum_{i=1}^{n} -2\,x_i(y_i - ax_i + b)$

Dérivée partielle en fonction de β_1 : $\frac{1}{n}\sum_{i=1}^{n} -2\,(y_i - ax_i + b)$

Pour commencer notre programme, on pourrait partir d'une droite de paramètre $a = 0$ et $b = 0$ mais ce ne serait pas optimisé, c'est pourquoi nous pouvons partir d'une droite qui relit les deux extrémités de nos points. Donc nous calculons le coefficient directeur de cette droite :

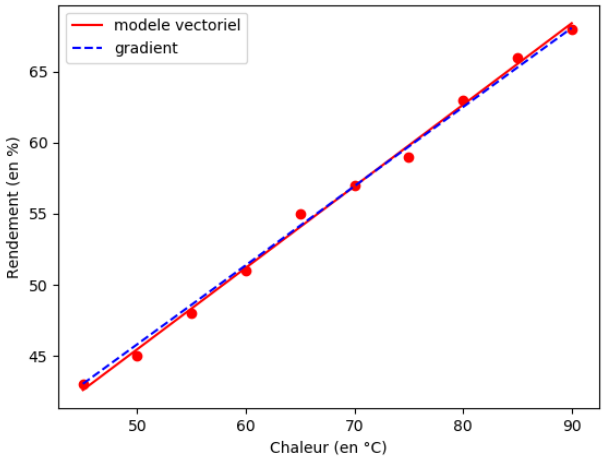
$$a = \frac{y_{10} - y_1}{x_{10} - x_1}$$

$$b = y_1 - ax_1$$

Maintenant nous pouvons créer une boucle qui testeras à chaque fois si : $\|a^t - a^{t-1}\| > e$, où e est l'erreur tolérée et a^t prend à chaque itération la valeur $a^{t-1} - \lambda\frac{\partial f(\beta)}{\partial \beta}$.

Pour une erreur à 10^{-4} et un pas λ à 10^{-3}, nous trouvons les coefficients :

$\boxed{a = 0.5563 \text{ et } b = 17.9999}$. Nous remarquons que les résultats et notamment celui de b, sont assez largement différent des deux premiers. On peut en déduire que cette méthode est légèrement moins précise que les deux premières. Cette différence peut se voir à l'œil nu sur la représentation graphique suivante :



2. Programmation

Pour programmer la totalité des TPs, nous avons utilisé les librairies python numpy, matplotlib et scipy.

Numpy : np.transpose(X) : Transpose une matrice X.

np.array(X) : Transforme une matrice en tableau.

np.dot(X, Y) : Multiplie deux matrices X et Y.

np.polyfit(X, Y, deg) : Renvoie les coefficients de régression linéaire des points x_i de X et y_i de Y, de degrés deg.

Matplotlib : plt.scatter(X, Y) : place les points x_i de X et y_i de Y sur un plan.

plt.plot(X, Y) : Trace une droite entre les points x_i de X et y_i de Y. Exemple : entre les points (X[0],Y[0]) et (X[1],Y[1]).

plt.x(y)label ($string$) : Donne un nom à l'axe x (ou y).

plt.legend() : Affiche les noms sur les axes.

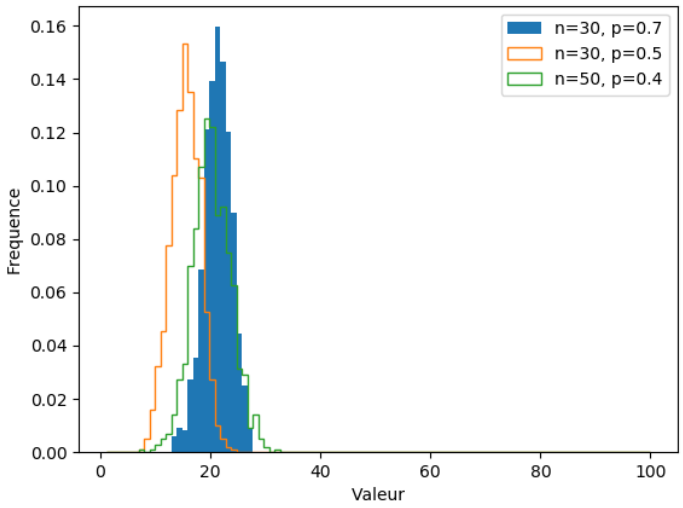
plt.show() : Affiche le graphique.

II. 2ème Partie : Etudes et manipulations de lois de probabilités

1. Démarche et résultats

a) Loi Binomiale

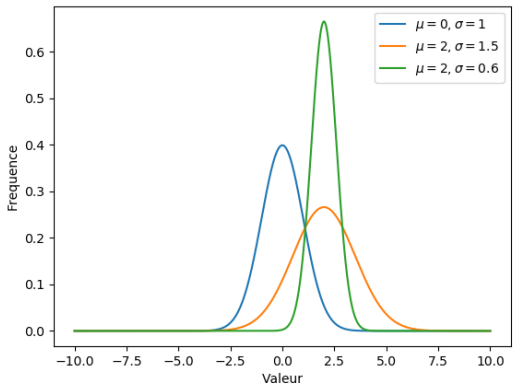
Pour la loi Binomiale, nous avons généré 1000 fois 10 essaies aléatoires pour chaque condition. Nous avons obtenu le graphique suivant :



Bien sûr, ce graphique ne représente qu'une représentation aléatoire, comme il peut en exister une infinité différente.

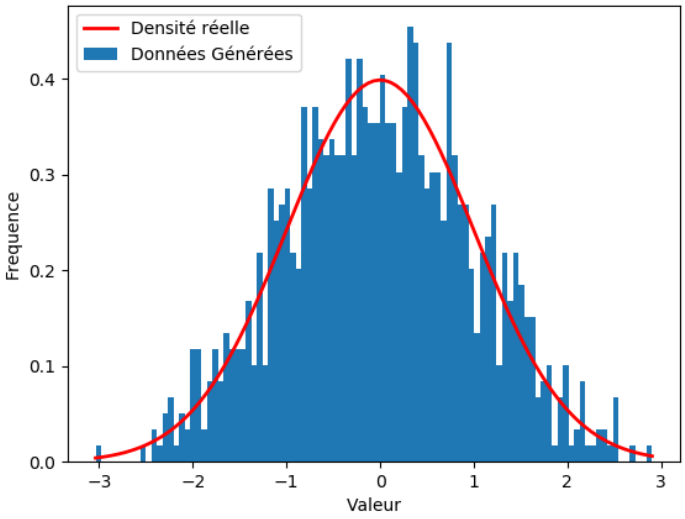
b) Loi Normale univariée

Pour la loi Normale univariée, nous avons généré 1000 valeurs aléatoires entre -10 et 10 pour chaque condition. Nous avons obtenu le graphique suivant :



c) Simulation de données à partir d'une loi : la loi Normale

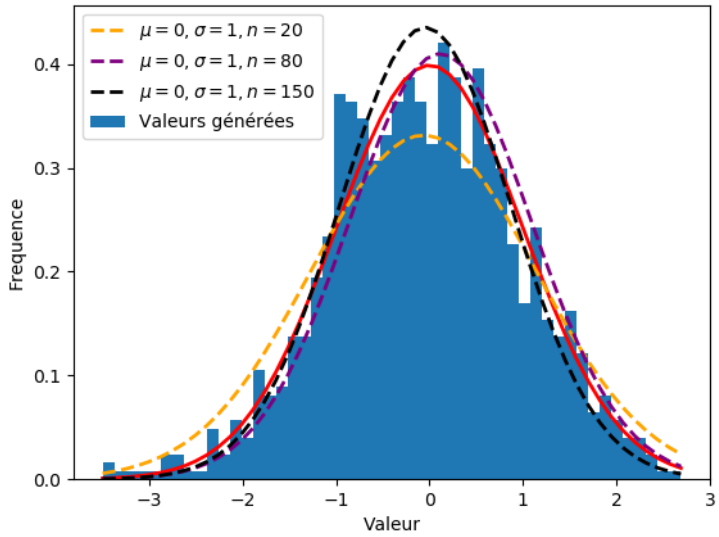
En générant 1000 valeurs suivant une loi Normale centrée et réduite (μ = 0 et σ = 1) puis en affichant sa courbe de vraie densité j'obtiens le résultat suivant :



On peut remarquer que la génération aléatoire est assez fidèle à sa représentation théorique bien qu'elle ne soit pas parfaite.

d) Estimation de densité : la loi Normale

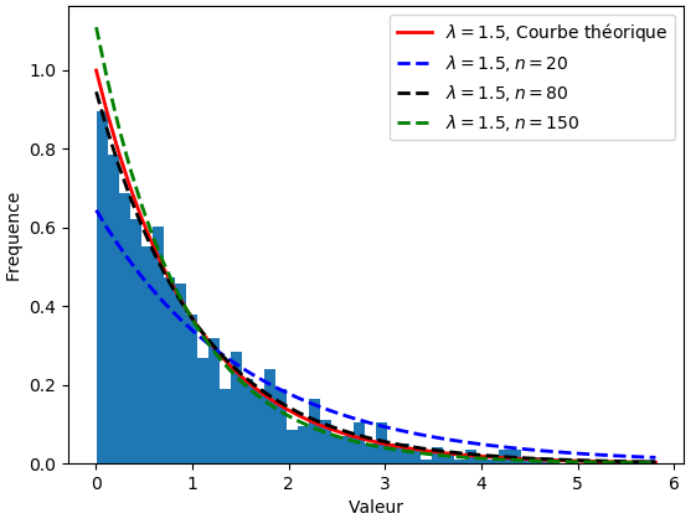
En générant 1000 valeurs aléatoires de la loi normale centrée et réduite (μ = 0 et σ = 1) et en affichant la droite de densité de lois centrée et réduite pour des échantillons n = 20, n = 80 et n = 150, j'obtiens le résultat suivant :



Nous remarquons que la génération des 1000 valeurs s'approche au mieux de l'axe rouge (courbe de densité standard de la loi normale centrée et réduite). Grâce à ce graphique, nous remarquons également que la taille de l'échantillon n'est aucunement synonyme de précision.

e) Estimation de densité : la loi exponentielle

Comme précédemment, nous avons généré 1000 valeurs aléatoires suivant la loi exponentielle de paramètre $\lambda = 1.5$ ainsi que sa courbe théorique puis la même chose avec le même paramètre λ avec des échantillons plus petits qui sont $n = 20$, $n = 80$ et $n = 150$ pour enfin afficher leur courbe et nous avons obtenu le résultat suivant :



Nous remarquons encore une fois que notre génération de 1000 valeurs suit plutôt bien la courbe théorique mais que le nombre de générations peut avoir un lien avec la précision de l'approximation de la courbe théorique.

En faisant varier λ, nous nous rendons compte que les courbes suivent bien la fonction théorique et s'agence comme pour $\lambda = 1.5$.

2. Programmation

Numpy : np.random.binomial(*n,p,n*) : Génère un nombre n de valeurs d'une distribution binomial de paramètre n et p.

np.linespace(*x,y,n*) : Renvoie n valeurs équidistante entre x et y.

np.random.normal(*mu,sigma,n*) : Génère un nombre n de valeurs d'une distribution normale de paramètre mu et sigma.

np.sqrt(*x*) : Renvoie la racine carrée de x (NaN si x est négatif).

np.pi : Renvoie le nombre PI.

np.exp(*x*) : Applique la fonction exponentielle à x.

np.std(*x*) : Calcule la déviation standard d'une distribution x.

np.random.exponential(*l,n*) : Génère un nombre n de valeurs d'une distribution exponentielle de paramètre $\lambda = l$.

Scipy.stats : norm.pdf(x,mu,sigma) : Calcule la fonction de densité d'une loi normale ayant les valeurs x et les paramètres mu et sigma.

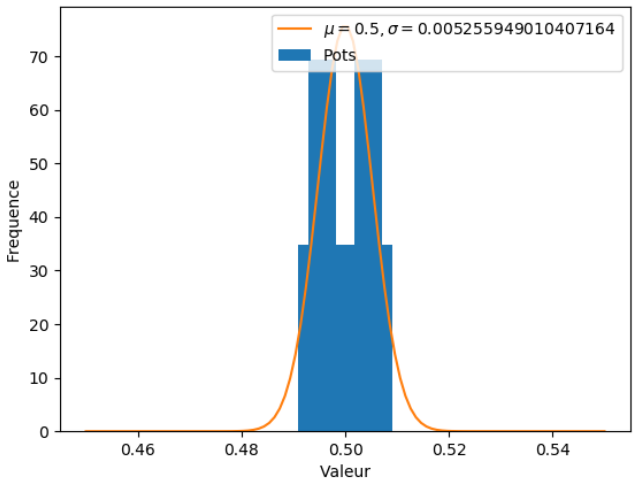
III. 3ème Partie : Intervalles de confiance

1. Démarche et résultat

a) Problème 1

Avec le jeu de données des poids des pots de confiture nous pouvons calculer la moyenne empirique qui est la somme des poids des pots divisé par le nombre totale de pots, nous avons donc : $\boxed{\bar{X} = 0.5 \text{ kg}}$.

Nous avons donc le graphique suivant :



Nous avons un petit échantillon (n=16), ce qui nous donne cet histogramme difforme mais nous pouvons voir qu'il respecte la courbe d'une loi normale de paramètres $\mu = \bar{X}$ et $\sigma =$ Ecart-type.

Grâce à la formule suivante, nous pouvons calculer l'intervalle de confiance du jeu de donnée à plusieurs niveaux de confiance :

$$\left[\bar{X}_n - u\frac{\sigma}{\sqrt{n}}\ , \bar{X}_n + u\frac{\sigma}{\sqrt{n}}\ \right]$$

Avec : $\bar{X}_n$: Moyenne Empirique

u : Le fractile d'ordre $1 - \frac{\alpha}{2}$ de la loi N(0,1)

σ : L'écart type

n : La taille de l'échantillon

Avec tous ceci nous avons :

Confiance	**Intervalle de confiance**
95%	[0.49783868330178993 , 0.5021613166982101]
99%	[0.4969432085483931 , 0.503056791451607]

Comme prévu, l'intervalle de confiance le plus élevé est également le plus large car il regroupe le plus de « chance » de réunir les possibles valeurs.

De la même manière, la moyenne empirique du poids des avocats est $\boxed{\bar{X} = 87.249375}$ et son écart-type est $\boxed{\sigma = 2.377350070850947}$.

Nous avons donc :

Confiance	**Intervalle de confiance**
95%	[86.27177677835687, 88.22697322164314]

b) Problème 2

Avec les données du problème nous pouvons déjà affirmer une chose, c'est que nous allons utiliser une loi Binomiale, mais de quel paramètre. La fréquence est de 95/500, ce sera donc une loi Binomiale B(95/500). Grâce à la formule suivante, nous pouvons calculer l'intervalle de confiance du jeu de donnée à plusieurs niveaux de confiance :

$$\left[f - u\frac{\sqrt{f(1-f)}}{\sqrt{n}} \, , f + u\frac{\sqrt{f(1-f)}}{\sqrt{n}} \right]$$

Avec : f : Fréquence

u : Le fractile d'ordre $1 - \frac{\alpha}{2}$ de la loi N(0,1)

n : La taille de l'échantillon

Les bornes de cet intervalle nous renvoient une probabilité (entre 0 et 1), il suffit de la multiplier par la taille de notre échantillon pour obtenir des bornes plus représentatives, ce qui nous donne le résultat :

Confiance	**Intervalle de confiance**
99%	[74.59300912751635, 115.40699087248366]

c) Problème 3

Nous avons dans ce problème une loi de Bernoulli de paramètre $p = \frac{1}{2}$. Pour montrer l'impact de la taille de l'échantillon, nous avons décidé générer indépendamment 100, 1 000, 10 000, 100 000 et 1 000 000 de valeurs avec les paramètres du problème et nous avons obtenu :

Confiance	**Nombre de valeurs**	**Intervalle**
95 %	100	[0.45802091766814534 , 0.6219790823318547]
95 %	1 000	[0.4749926326211129, 0.5270073673788871]
95 %	10 000	[0.49277574831379534, 0.5092242516862047]
95 %	100 000	[0.4969892589349918, 0.5021907410650082]
95 %	1 000 000	[0.4993605732416088, 0.5010054267583913]

Nous remarquons bien que, plus la distribution est générée, plus son intervalle de confiance se rapproche de sa valeur théorique.

2. Programmation

Scipy : norm.ppf(q,loc,scale) : Retourne le fractile d'ordre $1 - \frac{\alpha}{2}$ avec $\frac{q}{100}$ le degré de confiance souhaité (Ex : 0.99 pour 99%) de la loi Normale de paramètre μ = loc et σ = scale. Ici, nous l'utilisons dans le cas de la loi normale centrée et réduite donc $\mu = 0$ et $\sigma = 1$.