

SY09 Printemps 2020

TP 08 — Éléments de théorie bayésienne de la décision

Dans ce TP, on souhaite étudier les stratégies de Neyman-Pearson et de Bayes pour résoudre un problème de décision, dans le cas où les distributions conditionnelles voire les probabilités a priori sont connues.

Problème et modélisation

On considérera un problème (simplifié) d'identification de produits chimiques à partir de leur temps de dégradation. Plus particulièrement, on supposera être en présence de $g = 2$ produits, présents en proportions initiales π_1 (classe ω_1), et π_2 (classe ω_2).

On suppose pouvoir tester le temps de dégradation des produits selon deux protocoles; on notera X^1 et X^2 les temps de dégradation selon le premier et le second protocole, respectivement. Pour un même produit, on supposera indépendants ces temps X^1 et X^2 mesurés par chacun des deux protocoles. On suppose en outre que les distributions de temps de dégradation sont modélisés par des lois exponentielles

$$\begin{aligned} X^1 &\underset{\omega_1}{\sim} \mathcal{E}(\lambda_1), & X^2 &\underset{\omega_1}{\sim} \mathcal{E}(\lambda_2); \\ X^1 &\underset{\omega_2}{\sim} \mathcal{E}(\theta_1), & X^2 &\underset{\omega_2}{\sim} \mathcal{E}(\theta_2). \end{aligned}$$

Questions préliminaires

1. Quelle est la densité jointe du vecteur aléatoire $\mathbf{X} = (X^1, X^2)^T$ dans chacune des classes ?

En utilisant l'indépendance conditionnelle, et en notant f_{kj} la densité conditionnelle de X^j dans la classe ω_k et f_k la densité jointe de \mathbf{X} dans la classe ω_k , on a

$$f_1(\mathbf{x}) = f_{k1}(x_1)f_{k2}(x_2) = \lambda_1\lambda_2 \exp(-\lambda_1x_1 - \lambda_2x_2) = \lambda_1\lambda_2 \exp(-\lambda^T \mathbf{x}),$$

avec $\lambda = (\lambda_1, \lambda_2)^T$. De même, on a

$$f_2(\mathbf{x}) = \theta_1\theta_2 \exp(-\theta^T \mathbf{x}),$$

avec $\theta = (\theta_1, \theta_2)^T$.

2. Montrer que la frontière de décision obtenue en appliquant la stratégie de Neyman-Pearson est une droite dont on précisera les paramètres.

On a

$$\delta(\mathbf{x}) = \omega_1 \Leftrightarrow \frac{f_1(\mathbf{x})}{f_2(\mathbf{x})} \geq s,$$

avec

$$\Pr \left(\frac{f_1(\mathbf{X})}{f_2(\mathbf{X})} < s \mid Z = \omega_1 \right) = \alpha^*.$$

La frontière de décision correspond donc bien à l'équation d'une droite :

$$\frac{f_1(\mathbf{x})}{f_2(\mathbf{x})} < s \Leftrightarrow (\theta - \lambda)^T \mathbf{x} - \ln \left(\frac{\theta_1\theta_2}{\lambda_1\lambda_2} s \right) < 0.$$

3. Quelle frontière de décision obtient-on si l'on applique la stratégie de Bayes ?

On a

$$\delta(\mathbf{x}) = \omega_1 \Leftrightarrow \frac{f_1(\mathbf{x})}{f_2(\mathbf{x})} \geq \frac{\pi_2}{\pi_1} \Leftrightarrow (\theta - \lambda)^T \mathbf{x} - \ln \left(\frac{\theta_1 \theta_2 \pi_2}{\lambda_1 \lambda_2 \pi_1} \right) \geq 0.$$

Simulation

On cherche à générer un échantillon de taille n suivant le modèle génératif décrit ci-dessus. Intuitivement, pour chaque nouvel individu, il faudrait (1) déterminer sa classe puis (2) générer un vecteur aléatoire (composante par composante, les variables X^1 et X^2 étant indépendantes conditionnellement à la classe) suivant les paramètres correspondants.

On propose donc le protocole de simulation suivant :

1. calculer le nombre de points n_1 présents dans la classe ω_1 : on verra n_1 comme la réalisation d'une v.a. $N_1 \sim \mathcal{B}(n, \pi_1)$;
2. générer n_1 vecteurs $\mathbf{x}_1, \dots, \mathbf{x}_{n_1}$ en concaténant n_1 réalisations de variables aléatoires $X_1^1 \sim \mathcal{E}(\lambda_1)$ et $X_1^2 \sim \mathcal{E}(\lambda_2)$;
3. générer $n_2 = n - n_1$ vecteurs $\mathbf{x}_{n_1+1}, \dots, \mathbf{x}_n$ en concaténant n_2 réalisations de variables aléatoires $X_2^1 \sim \mathcal{E}(\theta_1)$ et $X_2^2 \sim \mathcal{E}(\theta_2)$.

Questions.

1. Implémenter ce protocole de simulation, et générer un échantillon et ses étiquettes de taille $n = 1000$, en utilisant $\pi_1 = 0.6$, $\lambda_1 = 1$, $\lambda_2 = 2$, $\theta_1 = 2$, $\theta_2 = 1$.

```
In [1]: pi1 = 0.6
        pi2 = 1 - pi1
        lambda1 = 1
        lambda2 = 2
        theta1 = 2
        theta2 = 1

        n = 1000
        n1 = np.random.binomial(n, pi1, 1)[0]
        n2 = n - n1

        X1 = np.column_stack(
            (
                np.random.exponential(scale=1/lambda1, size=n1),
                np.random.exponential(scale=1/lambda2, size=n1),
            )
        )
        X2 = np.column_stack(
            (
                np.random.exponential(scale=1/theta1, size=n2),
                np.random.exponential(scale=1/theta2, size=n2),
            )
        )
        X = np.concatenate((X1, X2), axis=0)

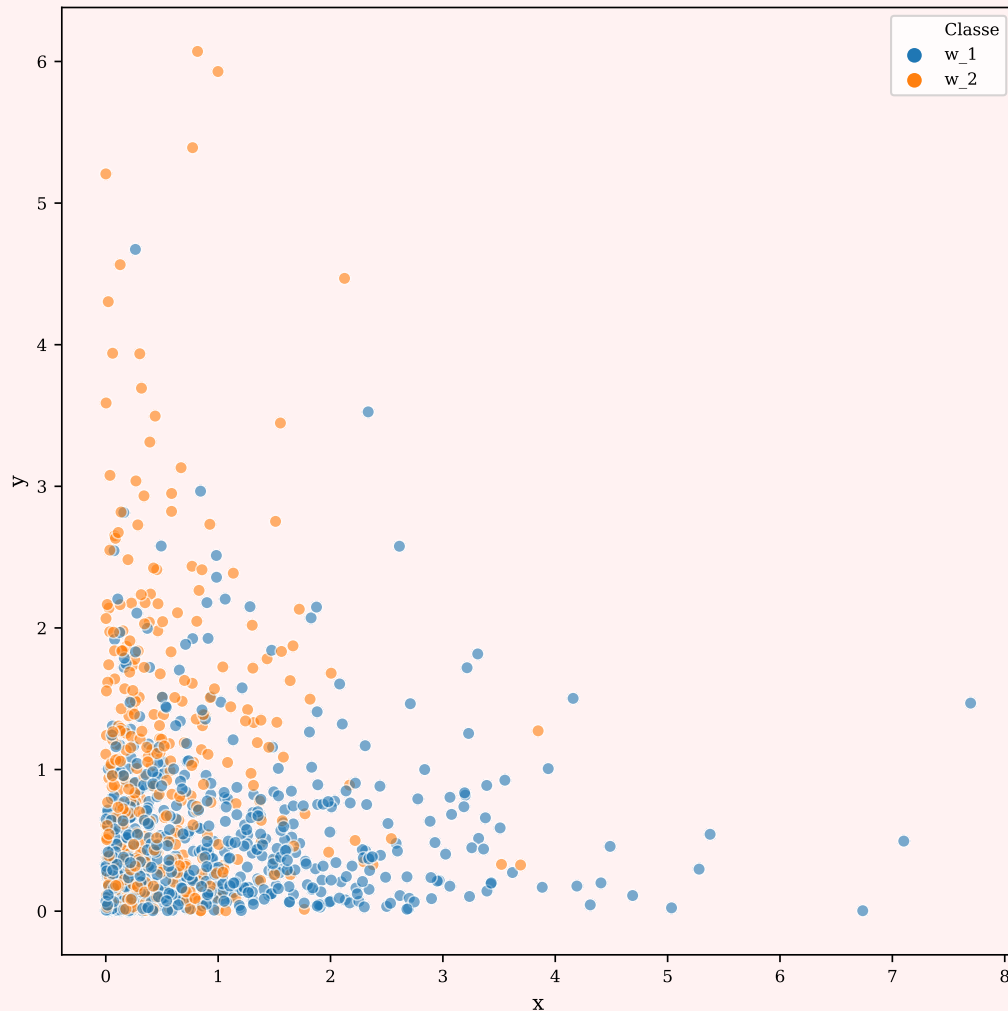
        z = np.repeat([0, 1], [n1, n2])

        from sklearn.utils import shuffle
        X, z = shuffle(X, z)
```

2. Afficher le jeu de données avec l'information de classe. Quel serait un bon séparateur linéaire ?

```
In [2]: classes = np.array(["w_1", "w_2"])[z]
        data = pd.DataFrame(dict(x=X[:, 0], y=X[:, 1], Classe=classes))

        sns.scatterplot(x="x", y="y", hue="Classe", data=data, alpha=.6)
        plt.show()
```



La droite $y = x$ semble être un bon séparateur linéaire.

3. Estimer le taux d'erreur de Bayes au moyen de cet échantillon. On pourra utiliser la fonction `np.apply_along_axis`.

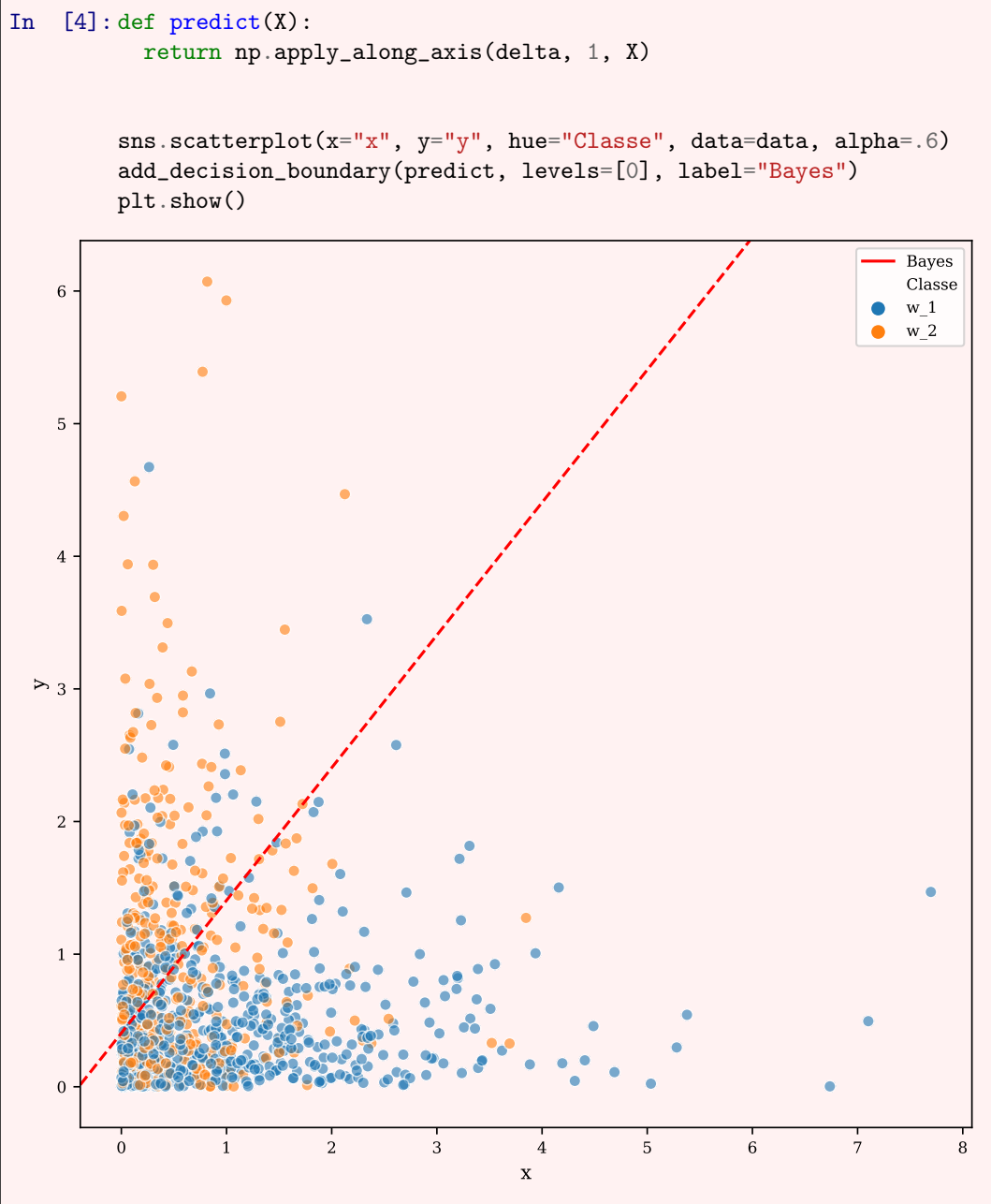
```
In [3]: def delta(row):
        x, y = row
        cste = np.log(theta1 * theta2 * pi2 / lambda1 / lambda2 / pi1)
        d = (theta1 - lambda1) * x + (theta2 - lambda2) * y - cste
        return d

        pred = np.apply_along_axis(delta, 1, X)
        pred = (pred < 0).astype(int)

        taux_Bayes = np.mean(pred != z)
        taux_Bayes

Out [3]: 0.316
```

4. Tracer la frontière de Bayes en utilisant la fonction fournie `add_decision_boundary`.



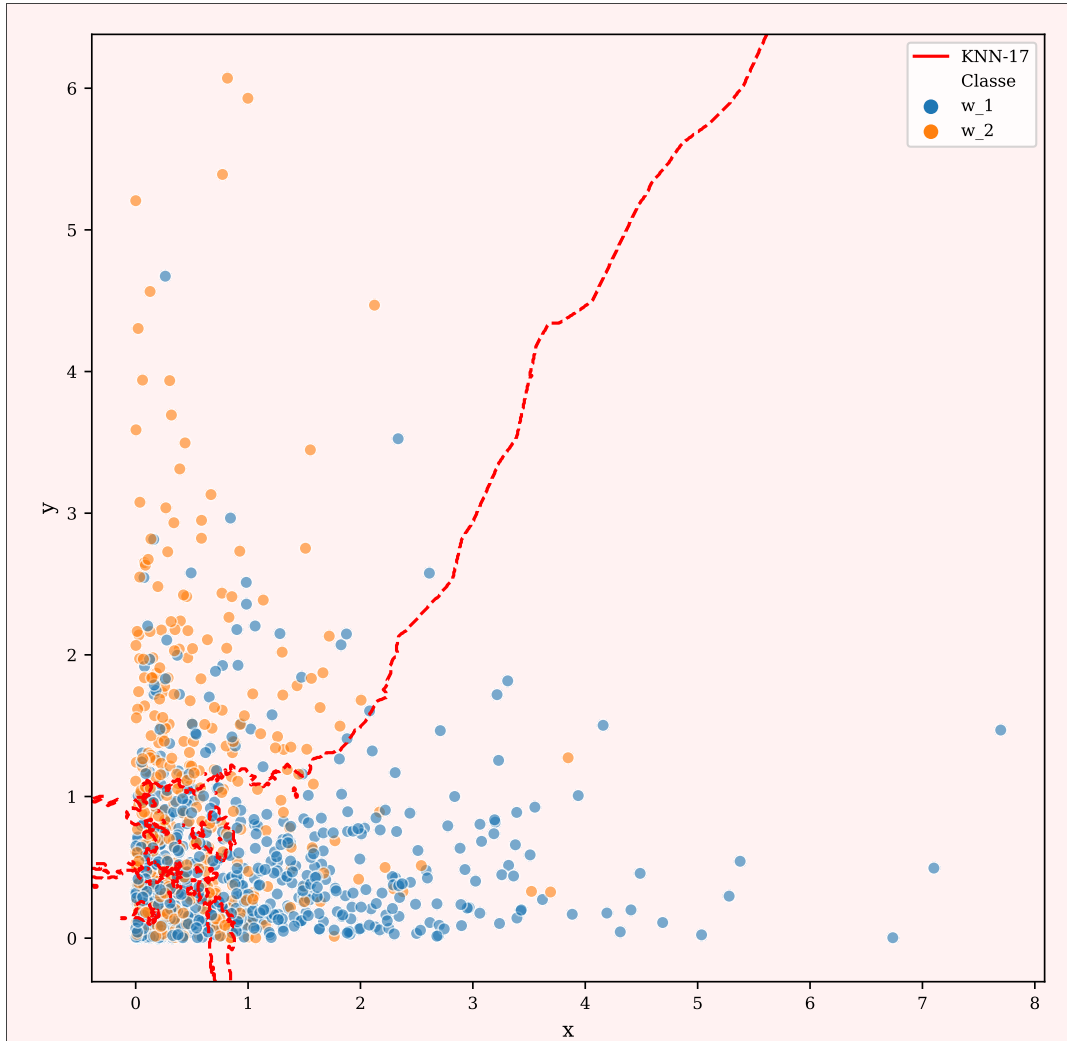
5. En utilisant le protocole d'évaluation des performances vu précédemment, appliquer la stratégie des K plus proches voisins à ce problème de décision. Comparer les performances obtenues au taux d'erreur de Bayes.

```
In [5]: from sklearn.neighbors import KNeighborsClassifier
        from sklearn.model_selection import train_test_split
        from sklearn.model_selection import GridSearchCV
        from sklearn.metrics import accuracy_score

        X_train, X_test, z_train, z_test = train_test_split(X, z,
        → test_size=0.33)

        n_neighbors_list = np.unique(np.round(np.geomspace(1, 500,
        → 100)).astype(int))
        param_grid = {"n_neighbors": n_neighbors_list}

        cls = KNeighborsClassifier()
        search = GridSearchCV(cls, param_grid, scoring="accuracy", cv=10)
        search.fit(X_train, z_train)
        search.best_params_
Out [5]: {'n_neighbors': 17}
In [6]: z_pred = search.predict(X_test)
        miscls_KNN = 1 - accuracy_score(z_pred, z_test)
        miscls_KNN
Out [6]: 0.3727272727272727
In [7]: z_pred = search.predict(X_train)
        miscls_KNN_train = 1 - accuracy_score(z_train, z_pred)
        miscls_KNN_train
Out [7]: 0.26417910447761195
In [8]: sns.scatterplot(x="x", y="y", hue="Classe", data=data, alpha=.6)
        add_decision_boundary(search,
        → label=f"KNN-{search.best_params_['n_neighbors']}")
        plt.show()
```



On constate que le taux d'erreur obtenu avec les K plus proches voisins est proche du taux d'erreur de Bayes estimé précédemment. En revanche, l'erreur d'apprentissage est plus faible mais c'est une estimation biaisée de l'erreur de test.

6. (Subsidiaire) Justifier rigoureusement l'estimation de la probabilité d'erreur de Bayes ϵ^* par la moyenne empirique des erreurs commises par la règle de Bayes δ^* sur un échantillon donné.

On commencera par rappeler la définition du taux d'erreur de Bayes donnée en cours :

$$\epsilon^* = \epsilon(\delta^*) = \int_{\mathbf{x}} \epsilon(\delta^*|\mathbf{x}) f(\mathbf{x}) d\mathbf{x} = \mathbb{E}_{\mathbf{X}} [\epsilon(\delta^*|\mathbf{x})],$$

où l'erreur conditionnelle $\epsilon(\delta^*|\mathbf{x})$ est elle-même définie comme l'espérance d'erreur (espérance calculée par rapport à la distribution de la variable de classe Z) :

$$\epsilon(\delta^*|\mathbf{x}) = \Pr(\delta^*(\mathbf{X}) \neq Z | \mathbf{X} = \mathbf{x}) = \mathbb{E}_Z [\mathbf{1}_{[\delta^*(\mathbf{x}) \neq Z]}],$$

avec $\mathbf{1}_{[\delta^*(\mathbf{x}) \neq Z]}$ la variable indicatrice d'une erreur conditionnelle à l'événement $\mathbf{X} = \mathbf{x}$. Le taux d'erreur de Bayes est donc l'espérance, calculée par rapport à la distribution jointe du couple (\mathbf{X}, Z) , de l'indicatrice d'erreur :

$$\epsilon(\delta^*) = \mathbb{E}_{\mathbf{X}, Z} [\mathbf{1}_{[\delta^*(\mathbf{X}) \neq Z]}].$$

Soulignons que cette dernière formule peut également être retrouvée à partir de la notion de risque d'une règle de décision, définie en cours comme l'espérance du coût des décisions prises par cette règle :

$$r(\delta^*) = \mathbb{E}_{\mathbf{X}, Z} [c(\delta^*(\mathbf{X})|Z)];$$

on retombe bien sur l'expression précédente en considérant des coûts 0/1. Pour estimer cette espérance, on peut donc utiliser un échantillon généré suivant la distribution jointe $\Pr(\mathbf{X}, Z)$, sur lequel on calculera la moyenne des erreurs de classement par la règle de Bayes δ^* . La propriété de convergence de la moyenne empirique (loi faible des grands nombres) permet d'obtenir, pour un échantillon suffisamment grand, une estimation raisonnablement proche du taux (théorique) d'erreur de Bayes $\varepsilon(\delta^*)$.

Calcul numérique de l'erreur de Bayes.

- ① Montrer que lorsqu'il n'y a que deux classes, l'erreur de Bayes peut s'écrire

$$\varepsilon(\delta^*) = \int \min(\pi_1 f_1(\mathbf{x}), \pi_2 f_2(\mathbf{x})) d\mathbf{x},$$

avec f_1 et f_2 les densités conditionnelles et π_1 et π_2 les proportions des classes.

On a successivement

$$\begin{aligned} \varepsilon(\delta^*) &= \mathbb{E}_{X,Z} [\mathbb{1}_{\delta^*(X) \neq Z}] \\ &= \mathbb{E}_X [\mathbb{1}_{\delta^*(X|Z=0) \neq 0}] + \mathbb{E}_X [\mathbb{1}_{\delta^*(X|Z=1) \neq 1}] \\ &= \int \mathbb{1}_{\delta^*(\mathbf{x}) \neq 0} f(\mathbf{x} | z=0) d\mathbf{x} + \int \mathbb{1}_{\delta^*(\mathbf{x}) \neq 1} f(\mathbf{x} | z=1) d\mathbf{x} \\ &= \int_{\delta^*(\mathbf{x}) \neq 0} f(\mathbf{x} | z=0) d\mathbf{x} + \int_{\delta^*(\mathbf{x}) \neq 1} f(\mathbf{x} | z=1) d\mathbf{x} \\ &= \int_{\delta^*(\mathbf{x}) \neq 0} \pi_1 f_1(\mathbf{x}) d\mathbf{x} + \int_{\delta^*(\mathbf{x}) \neq 1} \pi_2 f_2(\mathbf{x}) d\mathbf{x} \\ &= \int_{\pi_1 f_1(\mathbf{x}) < \pi_2 f_2(\mathbf{x})} \pi_1 f_1(\mathbf{x}) d\mathbf{x} + \int_{\pi_1 f_1(\mathbf{x}) \geq \pi_2 f_2(\mathbf{x})} \pi_2 f_2(\mathbf{x}) d\mathbf{x} \\ &= \int \min(\pi_1 f_1(\mathbf{x}), \pi_2 f_2(\mathbf{x})) d\mathbf{x} \end{aligned}$$

- ② Créer une fonction en Python prenant en argument une abscisse et une ordonnée et renvoyant la fonction sous le signe intégrale.

On pourra utiliser la fonction `expon` définie dans le module `scipy.stats` pour calculer la densité d'une loi exponentielle.

```
In [9]: from scipy.stats import expon

pi1 = 0.6
pi2 = 1 - pi1
lambda1 = 1
lambda2 = 2
theta1 = 2
theta2 = 1

def err_function(x, y):
    f1x = expon(scale=1/lambda1).pdf(x)
    f1y = expon(scale=1/lambda2).pdf(y)
    f2x = expon(scale=1/theta1).pdf(x)
    f2y = expon(scale=1/theta2).pdf(y)

    return min(pi1 * f1x * f1y, pi2 * f2x * f2y)
```

- ③ En utilisant la fonction `dblquad` d'intégration numérique d'une intégrale double, donner un encadrement de l'erreur de Bayes à 10^{-3} près.

```
from scipy.integrate import dblquad
```

```
In [10]: from scipy.integrate import dblquad
         dblquad(err_function, 0, np.inf, 0, np.inf, epsabs=1e-3)
Out [10]: (0.31110590440142405, 0.0009941577509708348)
```



Calcul numérique de l'erreur d'un classifieur

Dans cette partie, on se propose de calculer numériquement l'erreur théorique d'un classifieur δ quelconque. D'après la question 1, on sait que l'erreur de δ peut s'écrire

$$\varepsilon(\delta) = \int_{\delta(\mathbf{x})=1} \pi_1 f_1(\mathbf{x}) d\mathbf{x} + \int_{\delta(\mathbf{x})=0} \pi_2 f_2(\mathbf{x}) d\mathbf{x}.$$

En d'autres termes, il suffit d'identifier les deux régions de décision complémentaires de δ , et d'intégrer la bonne densité conditionnelle sur chacune des régions pour calculer l'erreur théorique de ce classifieur.

Nous allons utiliser la méthode de Simpson pour calculer numériquement ces intégrales. Pour cela, il faut commencer par discrétiser l'espace sur lequel on va intégrer. Une discrétisation uniforme ne présentant guère de sens pour notre problème, on choisit une discrétisation basée sur des fractiles exponentiels.

- ④ À l'aide de `np.linspace`, générer une discrétisation uniforme des ordres de fractiles entre 0 et 1 exclu.

```
In [11]: alpha = np.linspace(0, 1, 1000, endpoint=False)
```

- ⑤ À l'aide de la méthode `ppf` calculant les fractiles, en déduire une discrétisation exponentielle des abscisses et des ordonnées.

```
In [12]: rv = expon(scale=1/lambda1)
         x = rv.ppf(alpha)
         y = x
```

Il faut à présent calculer en chacun des points repérés par leur abscisse et leur ordonnée les deux densités possibles $f_1(\mathbf{x})$ et $f_2(\mathbf{x})$.

- ⑥ Créer un tableau bidimensionnel des densités pour f_1 . Faire de même pour f_2 .

```
In [13]: f1x = expon(scale=1/lambda1)
         f1y = expon(scale=1/lambda2)
         Y1, X1 = np.meshgrid(f1y.pdf(y), f1x.pdf(x))
         f1 = pi1 * Y1 * X1

         f2x = expon(scale=1/theta1)
         f2y = expon(scale=1/theta2)
         Y2, X2 = np.meshgrid(f2y.pdf(y), f2x.pdf(x))
         f2 = pi2 * Y2 * X2
```

Il faut maintenant identifier les deux régions de décision et prédisant la classe pour chaque point de la discrétisation bidimensionnelle.

- ⑦ Créer un tableau bidimensionnel donnant la classe pour chaque point de la discrétisation bidimensionnelle.


```
In [14]: Y, X = np.meshgrid(y, x)
         xy = np.column_stack((X.ravel(), Y.ravel()))
         Z = search.predict(xy).reshape(X.shape)
```

⑧ Créer un tableau bidimensionnel calculant la valeur de la fonction à intégrer en utilisant la prédiction de la classe.

On pourra utiliser la fonction `np.where`.

```
In [15]: err_KNN = np.where(Z == 0, f2, f1)
         err_bayes = np.minimum(f1, f2)
```

⑨ Intégrer numériquement la fonction obtenue. On utilisera deux intégrations successives avec la méthode de Simpson.

```
from scipy.integrate import simps
simps(simps(<tableau bidim à intégrer>, <discrétisation des y>),
      ↪ <discrétisation des x>)
```

```
In [16]: from scipy.integrate import simps
         simps(simps(err_bayes, y), x)
Out [16]: 0.31111018030312493
In [17]: simps(simps(err_KNN, y), x)
Out [17]: 0.34551556445092557
```