SY09 Printemps 2020
TP 07 — Introduction à l'apprentissage supervisé, méthode des K plus proches voisins

On souhaite utiliser l'algorithme des K plus proches voisins sur différents jeux de données, à des fins de discrimination. On complétera tout d'abord les fonctions fournies, puis on les testera sur des données synthétiques (générées selon une distribution prédéfinie) puis réelles.

1 Méthode des K plus proches voisins

On rappelle que la méthode des K plus proches voisins ne nécessite pas de phase d'apprentissage à proprement parler.

Apprentissage

La méthode des K plus proches voisins est implémentée dans la classe `KNeighborsClassifier` qu'on charge au moyen de l'instruction suivante

```
from sklearn.neighbors import KNeighborsClassifier
```

Lors de l'instanciation de la classe `KNeighborsClassifier`, l'argument le plus important est `n_neighbors` qui détermine le nombre de voisins utilisés dans la règle de décision. On peut par exemple définir

```
cls = KNeighborsClassifier(n_neighbors=3)
```

Il faut ensuite apprendre le modèle avec la méthode `fit`

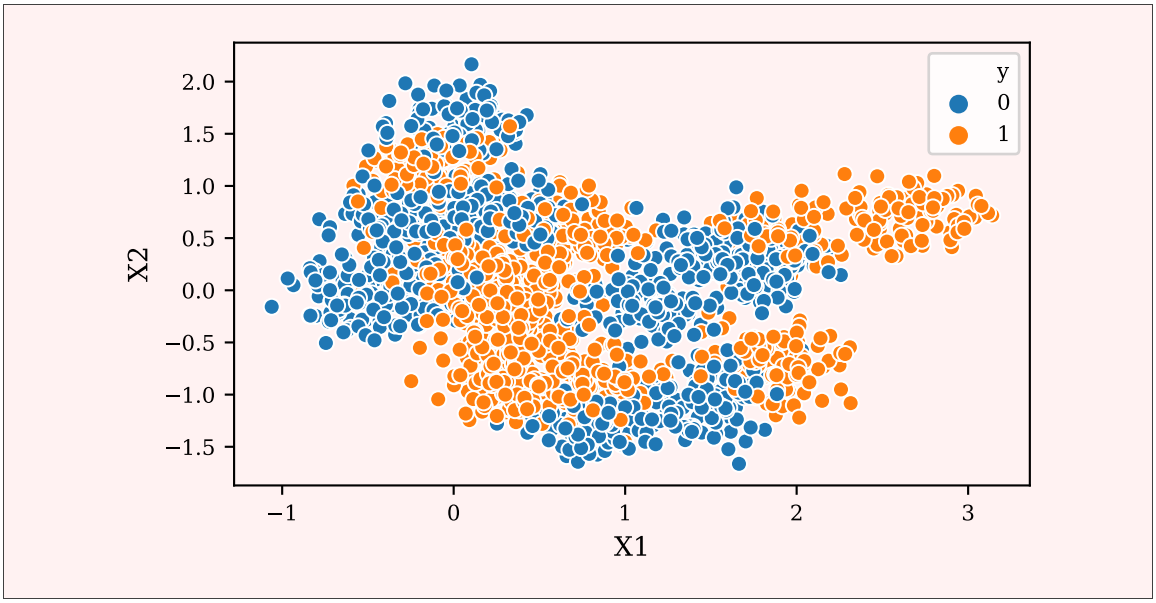
```
cls.fit(X, y)
```

avec `X` le jeu de données et `y` les étiquettes correspondantes. On peut alors prédire les étiquettes d'un autre jeu de données `Y` avec l'instruction

```
labels = cls.predict(Y)
```

(1) Charger et visualiser le jeu de données `Synth1-2000.csv` avec la fonction `plot_clustering` utilisée lors du TP05.

```
In [1]: Xy = pd.read_csv("data/Synth1-2000.csv")
        X = Xy.iloc[:, :-1]
        y = Xy.iloc[:, -1]

        plot_clustering(X, y)
        plt.show()
```



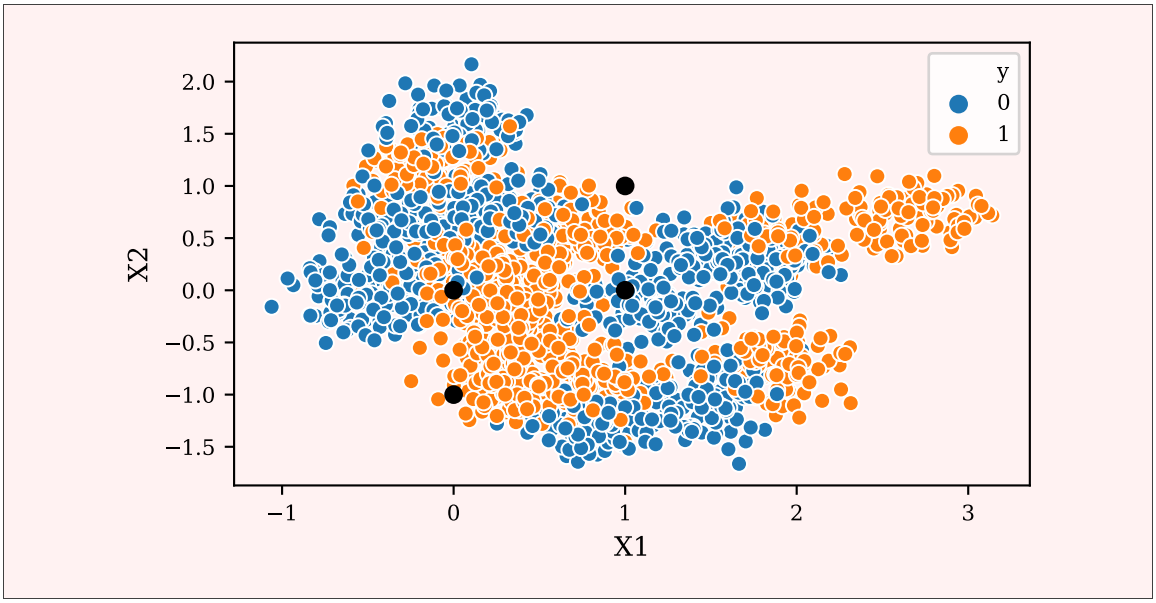
(2) Utiliser la méthode des cinq plus proches voisins sur le jeu de données précédent pour prédire la classe des points suivants :

$$\mathbf{p}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \mathbf{p}_2 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \quad \mathbf{p}_3 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{p}_4 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

On pourra utiliser le code suivant pour placer ces points

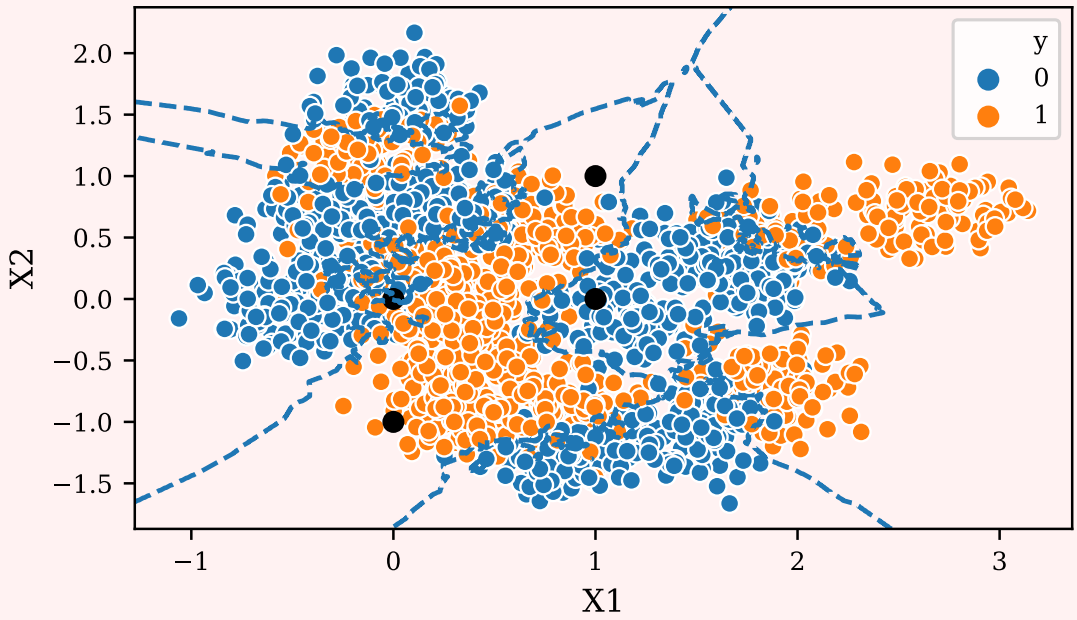
```
Y = np.array([[0, 0], [0, -1], [1, 0], [1, 1]])
plt.scatter(*Y.T, color="k")
```

```
In  [2]: from sklearn.neighbors import KNeighborsClassifier

         cls = KNeighborsClassifier(n_neighbors=5)
         cls.fit(X, y)
         Y = np.array([[0, 0], [0, -1], [1, 0], [1, 1]])
         labels = cls.predict(Y)
         labels
Out [2]: array([0, 1, 0, 1])
In  [3]: plot_clustering(X, y)
         plt.scatter(*Y.T, color="k")
         plt.show()
```



③ Utiliser la fonction `add_decision_boundary` pour visualiser la frontière de décision.

```
In  [4]: ax, _ = plot_clustering(X, y)
         ax.scatter(*Y.T, color="k")
Out [4]: <matplotlib.collections.PathCollection object at 0x1a2472ee90>
In  [5]: add_decision_boundary(cls)
         plt.show()
```



1.1 Sélection de modèle

L'hyperparamètre K du nombre de voisins a jusqu'alors été choisi arbitrairement. Pour déterminer le nombre « optimal » de voisins K_{opt}, on adopte la stratégie dite de validation simple suivante. On sépare aléatoirement l'ensemble des données disponibles de manière à former un ensemble d'apprentissage et un ensemble de validation. L'ensemble d'apprentissage est réservé à l'apprentissage du modèle uniquement. L'ensemble de validation sert à sélectionner le meilleur modèle.

(4) Séparer le jeu de données en un ensemble d'apprentissage et un ensemble de validation avec deux fois plus d'exemples d'apprentissage que de validation. On pourra utiliser la fonction `train_test_split` rendue disponible par l'instruction

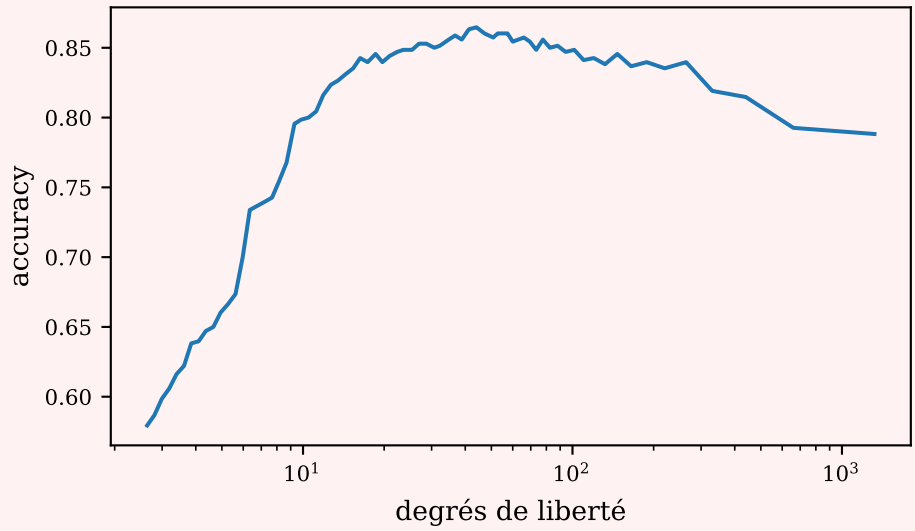
```
from sklearn.model_selection import train_test_split
```

```
In [6]: from sklearn.model_selection import train_test_split
        X_train, X_val, y_train, y_val = train_test_split(X, y, train_size=0.66)
```

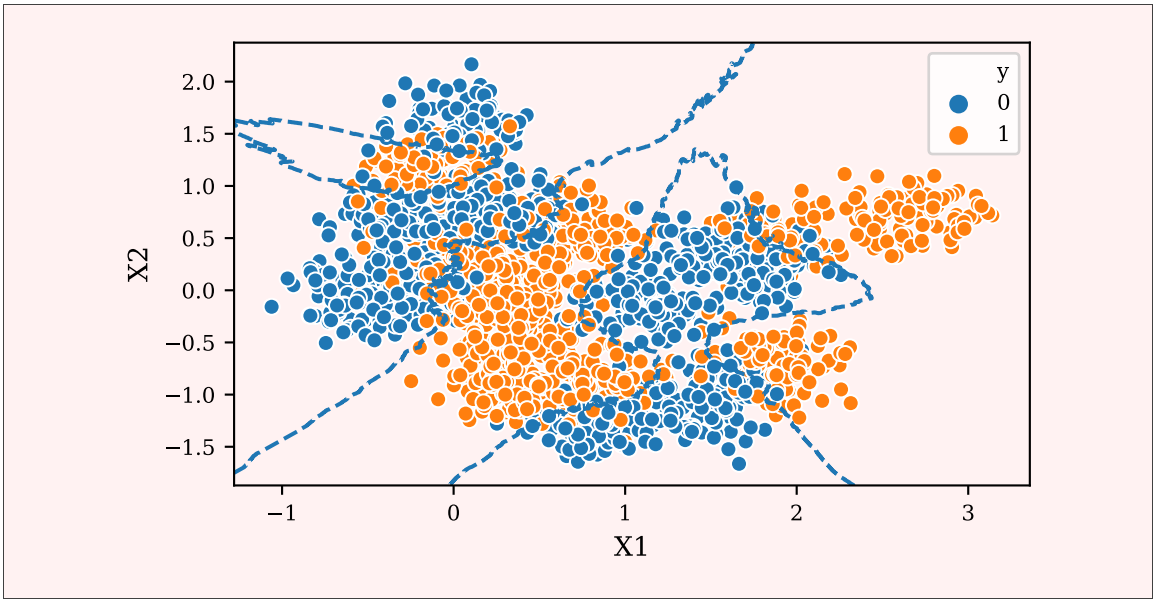
(5) Compléter les fonctions `accuracy` et `knn_simple_validation` afin de déterminer le nombre « optimal » de voisins K_{opt} (choisis parmi une liste `n_neighbors_list` de valeurs possibles), c'est-à-dire donnant les meilleurs performances sur un ensemble de validation.

On pourra visualiser les résultats avec `seaborn` en utilisant `sns.lineplot` avec les arguments `err_style` et `ci` et sélectionner le meilleur nombre de voisins avec `idxmax`.

```
In [7]: # De 1 à 500 voisins (exclu), échelle logarithmique
        n_neighbors_list = np.unique(np.round(np.geomspace(1, 500,
        ↪ 100)).astype(int))

        gen = knn_simple_validation(X_train, y_train, X_val, y_val,
        ↪ n_neighbors_list)
        df = pd.DataFrame(gen, columns=["# neighbors", "accuracy", "degrés de
        ↪ liberté"])

        sp = sns.lineplot(x="degrés de liberté", y="accuracy", data=df)
        sp.set(xscale="log")
Out [7]: [None]
In [8]: plt.show()
```




```
In [9]: Kopt = df.loc[df.accuracy.idxmax(), "# neighbors"]
        Kopt
Out [9]: 30
In [10]: cls = KNeighborsClassifier(n_neighbors=Kopt)
         cls.fit(X, y)
         plot_clustering(X, y)
         add_decision_boundary(cls)
         plt.show()
```

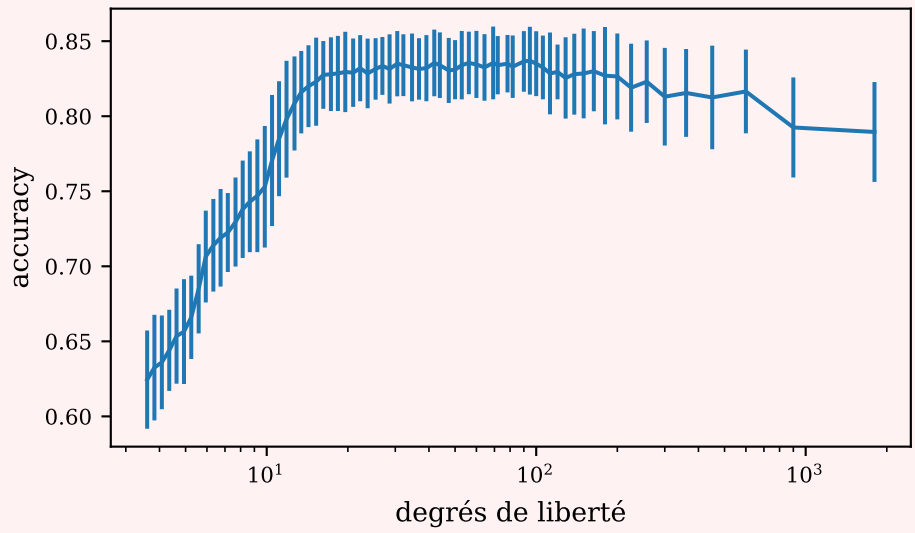


Lorsque l'espace des hyperparamètres est trop grand ou le nombre de données insuffisantes, la validation simple peut sélectionner le mauvais modèle. En effet, il se peut que le modèle sélectionné soit uniquement celui qui présente de bonnes performances sur l'ensemble de validation. Pour y pallier, on peut utiliser la validation multiple. Il s'agit de réitérer plusieurs fois la validation simple en changeant à chaque fois l'ensemble d'apprentissage et l'ensemble de validation.

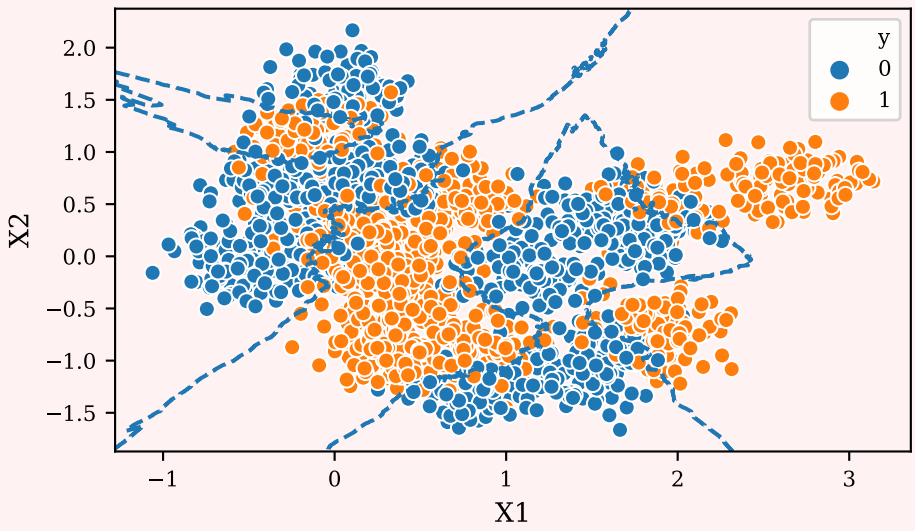
(6) Compléter la fonction `knn_multiple_validation` qui renvoie un générateur produisant les erreurs de validation.

On pourra visualiser les résultats avec `seaborn` en utilisant `sns.lineplot` et sélectionner le meilleur nombre de voisins avec `idxmax`.

```
In  [11]: train_size = 0.90
          n_splits = 10
          gen = knn_multiple_validation(X, y, n_splits, train_size,
          ↪  n_neighbors_list)

          df = pd.DataFrame(gen, columns=["# neighbors", "accuracy", "degrés de
          ↪  liberté"])
          Kopt = df.loc[df.accuracy.idxmax(), "# neighbors"]
          Kopt
Out [11]: 26
In  [12]: sp = sns.lineplot(x="degrés de liberté", y="accuracy",
          ↪  err_style="bars", ci="sd", data=df)
          sp.set(xscale="log")
Out [12]: [None]
In  [13]: plt.show()
```



In [14]:
```
cls = KNeighborsClassifier(n_neighbors=Kopt)
cls.fit(X, y)
plot_clustering(X, y)
add_decision_boundary(cls)
plt.show()
```

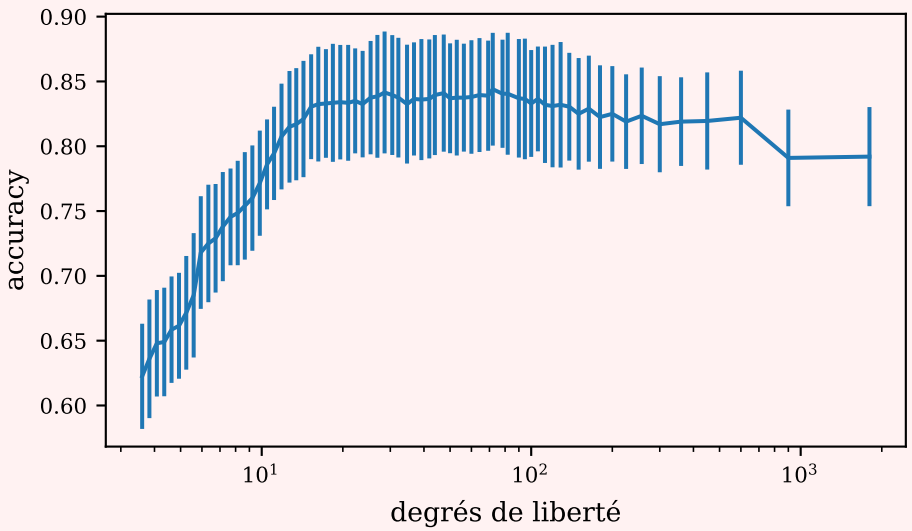


La validation simple répétée présente l'inconvénient statistique de sous ou sur-représenter certains exemples dans les jeux de données d'apprentissage ou de validation. Il faut alors répéter la validation simple un grand nombre de fois pour se débarrasser de ce biais ce qui peut être problématique pour des jeux de données conséquents.

La validation croisée réalise un compromis. Tous les exemples ont le même statut et le nombre d'apprentissage de modèles à effectuer est limité.

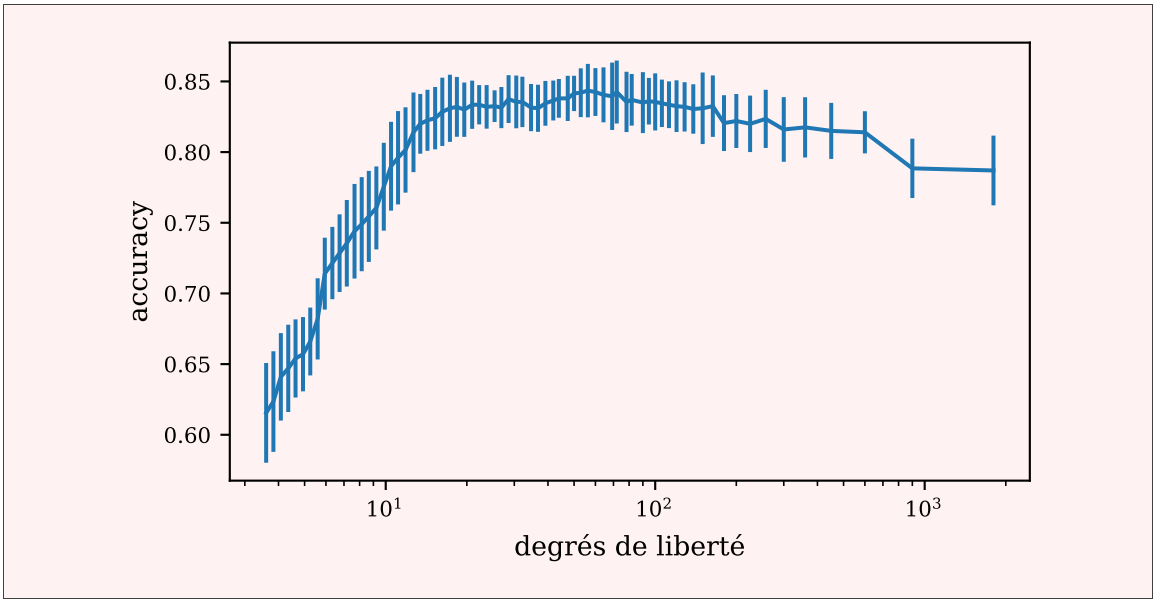
(7) Compléter la fonction `knn_cross_validation`. Visualiser les résultats obtenus.

```
In  [15]: n_folds = 10
          gen = knn_cross_validation(X, y, n_folds, n_neighbors_list)

          df = pd.DataFrame(gen, columns=["# neighbors", "accuracy", "degrés de
          ↪  liberté"])
          Kopt = df.loc[df.accuracy.idxmax(), "# neighbors"]
          Kopt
Out [15]: 46
In  [16]: sp = sns.lineplot(x="degrés de liberté", y="accuracy",
          ↪  err_style="bars", ci="sd", data=df)
          sp.set(xscale="log")
Out [16]: [None]
In  [17]: plt.show()
```



(8) Le calcul des erreurs de validation croisée peut être automatisé en utilisant la fonction `cross_val_score`. Réécrire en cinq lignes la fonction `knn_cross_validation`.

```
In  [18]: from sklearn.model_selection import cross_val_score
          n_folds = 10
          gen = knn_cross_validation2(X, y, n_folds, n_neighbors_list)

          df = pd.DataFrame(gen, columns=["# neighbors", "accuracy", "degrés de
          ↪  liberté"])
          Kopt = df.loc[df.accuracy.idxmax(), "# neighbors"]
          Kopt
Out [18]: 26
In  [19]: sp = sns.lineplot(x="degrés de liberté", y="accuracy",
          ↪  err_style="bars", ci="sd", data=df)
          sp.set(xscale="log")
Out [19]: [None]
In  [20]: plt.show()
```



`Scikit-learn` permet de calculer automatiquement une validation croisée mais il permet également de sélectionner directement le meilleur hyperparamètre. Pour cela, on utilise la classe `GridSearchCV` du module `sklearn.model_selection`.

La classe `GridSearchCV` s'utilise comme les classes `scikit-learn` déjà vues. Il faut instancier la classe avec des paramètres et ensuite appeler la méthode `fit`.

Les deux premiers paramètres sont les suivants :
- `estimator` : Le modèle (instancié) sur lequel on veut recherchre les hyperparamètres les plus performants.
- `param_grid` : Les hyperparamètres à tester.

Parmi les autres paramètres nommés utiles, on trouve
- `scoring` : le critère utilisé pour évaluer le classifieur,
- `cv` : le nombre de plis à utiliser pour la validation croisée.

Une fois l'apprentissage terminé, les attributs suivants sont disponibles :
- `best_estimator_` : le meilleur estimateur trouvé,
- `best_params_` : un dictionnaire des meilleurs paramètres trouvés,
- `cv_results_` : la synthèse de tous les résultats des validations croisées pour tous les paramètres testés.

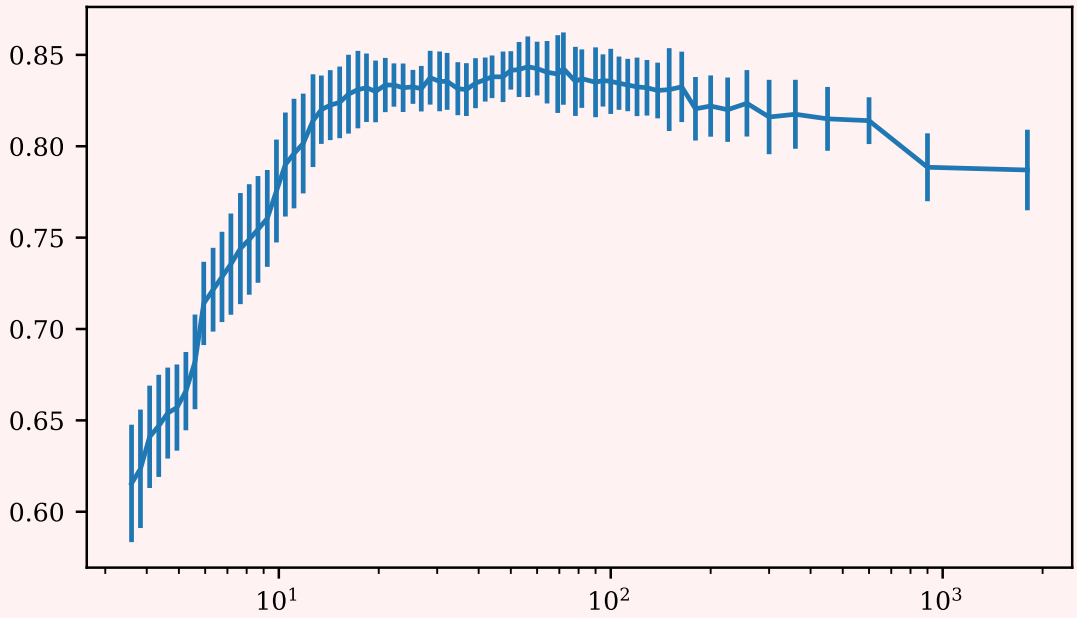
(9) Créer un objet de classe `GridSearchCV` pour rechercher le nombre de voisins optimal.

```
In  [21]: from sklearn.model_selection import GridSearchCV

          # De 1 à 500 voisins (exclu), échelle logarithmique
          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, y)
          search.best_params_
Out [21]: {'n_neighbors': 32}
```

(10) En utilisant l'attribut `cv_results_`, regénérer la figure précédente.

On pourra utiliser la fonction `errorbar` de `matplotlib`.

```
In [22]: df = pd.DataFrame(
             (
                 dict(n_neighbors=d["n_neighbors"], error=e, std=s)
                 for d, e, s in zip(
                     search.cv_results_["params"],
                     search.cv_results_["mean_test_score"],
                     search.cv_results_["std_test_score"],
                 )
             )
         )
         n = 9/10 * len(y)
         plt.errorbar(n/df["n_neighbors"], df["error"], yerr=df["std"])
         plt.xscale("log")
         plt.show()
```



1.2 Estimation des performances

(11) En utilisant `train_test_split` et `GridSearchCV`, donner une estimation non biaisée de la précision du modèle sélectionné.

```
In  [23]: from sklearn.model_selection import GridSearchCV
          from sklearn.model_selection import train_test_split
          from sklearn.metrics import accuracy_score

          X_train, X_test, y_train, y_test = train_test_split(X, y,
          ↪  train_size=0.66)

          # De 1 à 500 voisins (exclu), échelle logarithmique
          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, y_train)
          y_pred = search.predict(X_test)
          accuracy_score(y_pred, y_test)
Out [23]: 0.8323529411764706
```

2 Méthode des « K plus proches prototypes »

La méthode des K plus proches voisins présente des propriétés intéressantes, mais cette stratégie reste coûteuse : elle nécessite, en phase de test, de calculer la distance entre chaque individu de test et tous les individus d'apprentissage. On souhaite ici en tester une variante, dans laquelle l'ensemble d'apprentissage sera résumé par un ensemble de points caractéristiques que nous appellerons *prototypes*.

Le bénéfice attendu d'une telle opération est évidemment calculatoire ; notons qu'elle a également une influence sur le plan des performances, en fonction du nombre de prototypes choisi pour résumer une classe et de la manière dont ces prototypes sont déterminés.

2.1 Apprentissage des prototypes

Cette variante de la méthode des K plus proches voisins comporte à présent une phase d'apprentissage : le calcul des prototypes qui résument les individus d'apprentissage dans chaque classe.

Pour réaliser cet apprentissage, on utilisera l'algorithme des « C_k-means » [1] : pour *chaque classe* ω_k, on déterminera ainsi C_k centres qui résumeront la classe. L'ensemble de ces centres (étiquetés) sera ensuite utilisé à la place de l'ensemble d'apprentissage pour classer les individus de test.

Les paramètres C_k, qui fixent pour chaque classe ω_k le nombre de prototypes qui la résument, doivent bien être différenciés du paramètre K, qui détermine le nombre de plus proches prototypes utilisés en phase de test pour classer les individus.

2.2 Questions

(12) Supposons que l'on fixe $C_k = 1$ pour tout $k = 1, \ldots, g$, et $K = 1$: à quel classifieur correspond alors la méthode des K plus proches prototypes ?

Il s'agit alors du classifieur euclidien.

(13) Si l'on fixe à présent $C_k = n_k = \sum_{i=1}^{n} z_{ik}$, quel classifieur retrouve-t-on ?

1. Il se peut que l'on veuille utiliser un indicateur de tendance centrale plus robuste aux points atypiques que la moyenne ; cela revient à remplacer l'algorithme des C_k-means par une autre méthode de partitionnement, comme par exemple la stratégie des C_k-*médoïdes* (dans laquelle on substitue la médiane à la moyenne).

On retrouve évidemment la méthode des K plus proches voisins.

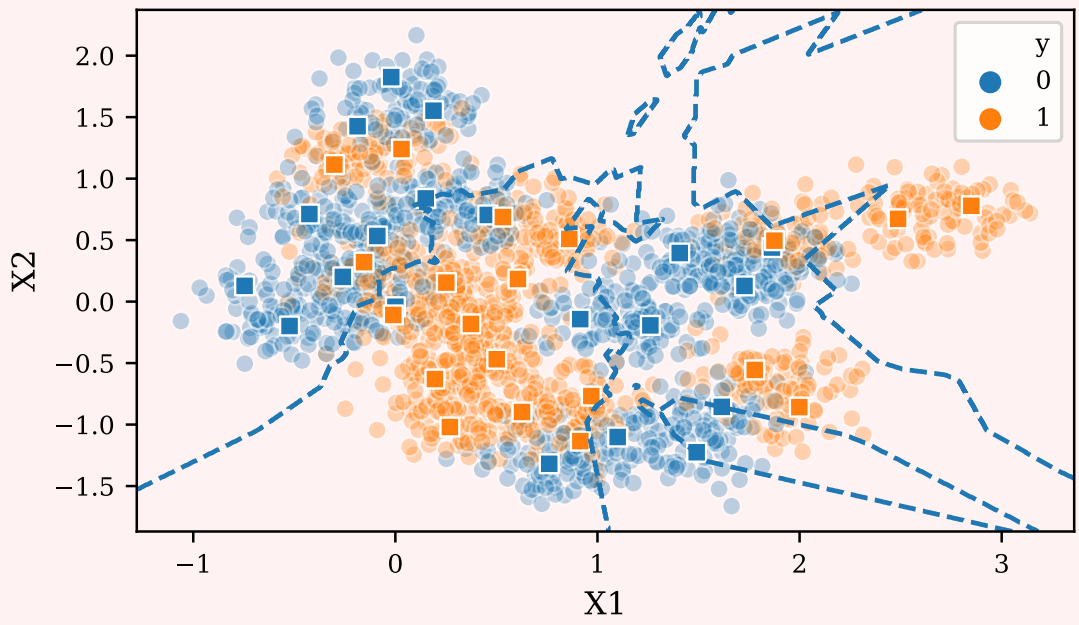
(14) Quelle relation entre les C_k et K doit-on avoir pour que l'algorithme soit bien défini ?

On doit au moins avoir $\sum_k C_k \geq K$.

(15) Compléter le fichier `src/nearest_prototypes.py` qui implémente les K plus proches prototypes.

Tester l'algorithme sur le jeu de données `Synth1-2000.csv`.

```
In  [24]: Xy = pd.read_csv("data/Synth1-2000.csv")
          X = Xy.iloc[:, :-1]
          y = Xy.iloc[:, -1]

          # 20 prototypes pour chaque classe, un 7-NN sur ces 40 prototypes
          knp = NearestPrototypes([20, 20], 7)
          knp.fit(X, y)

          ax = sns.scatterplot(x="X1", y="X2", hue="y", data=Xy, alpha=0.3,
          ↪  legend=False)
          df = pd.DataFrame(
              dict(X1=knp.prototypes_[:, 0], X2=knp.prototypes_[:, 1],
              ↪  y=knp.labels_)
          )
          sns.scatterplot(x="X1", y="X2", hue="y", marker="s", data=df)
Out [24]: <matplotlib.axes._subplots.AxesSubplot object at 0x1a24b36e90>
In  [25]: add_decision_boundary(knp)
          plt.show()
```

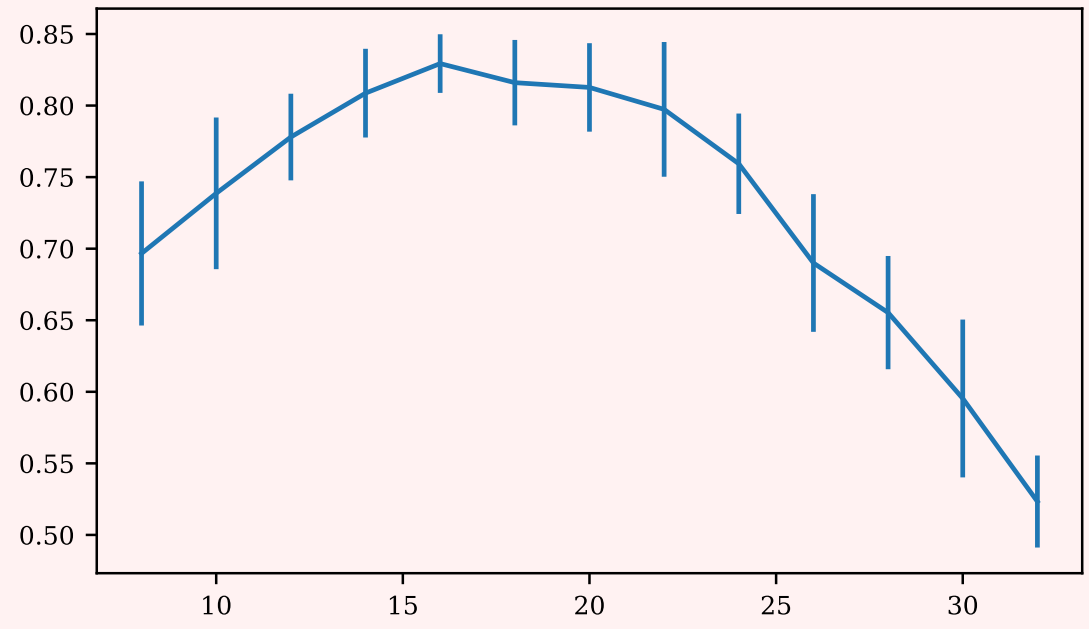


(16) Tester l'algorithme sur les jeux de données `Synth2-1500.csv` et `Synth3-1500.csv`.

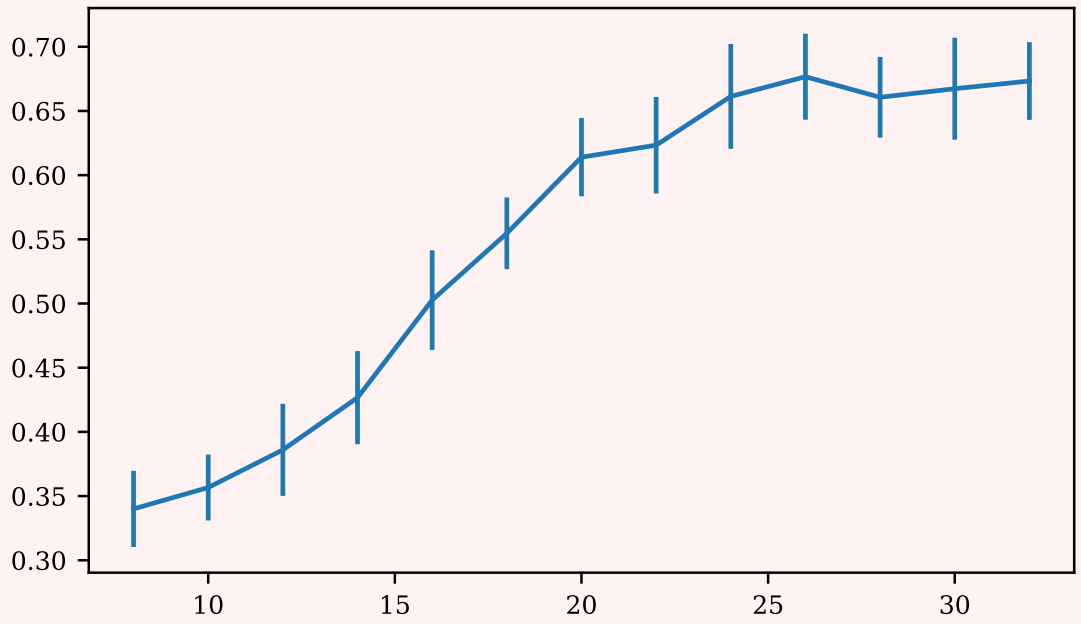
In [26]:
```
Xy1 = pd.read_csv("data/Synth2-1500.csv")
X1 = Xy1.iloc[:, :-1]
y1 = Xy1.iloc[:, -1]

Xy2 = pd.read_csv("data/Synth3-1500.csv")
X2 = Xy2.iloc[:, :-1]
y2 = Xy2.iloc[:, -1]


from sklearn.model_selection import GridSearchCV

param_grid = {
    "n_neighbors": [3],
    "n_prototypes_list": [[20 - i, 20 + i] for i in range(-12, 14, 2)],
}

cls = NearestPrototypes(n_prototypes_list=[20, 20], n_neighbors=3)
search = GridSearchCV(cls, param_grid, scoring="accuracy", cv=10)
search.fit(X1, y1)
search.best_params_
```

Out [26]:
```
{'n_neighbors': 3, 'n_prototypes_list': [16, 24]}
```

In [27]:
```
df = pd.DataFrame(
    (
        dict(n_prototypes0=d["n_prototypes_list"][0], error=e, std=s)
        for d, e, s in zip(
            search.cv_results_["params"],
            search.cv_results_["mean_test_score"],
            search.cv_results_["std_test_score"],
        )
    )
)
plt.errorbar(df["n_prototypes0"], df["error"], yerr=df["std"])
plt.show()
```



```
In [28]: search.fit(X2, y2)
         df = pd.DataFrame(
             (
                 dict(n_prototypes0=d["n_prototypes_list"][0], error=e, std=s)
                 for d, e, s in zip(
                     search.cv_results_["params"],
                     search.cv_results_["mean_test_score"],
                     search.cv_results_["std_test_score"],
                 )
             )
         )
         plt.errorbar(df["n_prototypes0"], df["error"], yerr=df["std"])
         plt.show()
```



2.3 Recherche aléatoire dans l'espace des hyperparamètres

Dans cette section, on souhaite déterminer les hyperparamètres optimaux avec pour seule restriction le coût à l'évaluation de l'algorithme. Les hyperparamètres sont donc

- n_1 et n_2, les nombres des prototypes pour chacune des classes,
- K le nombre de voisins pour l'algorithme des plus proches voisins.

On suppose que le coût à l'évaluation de l'algorithme des plus proches voisins est $(n_1 + n_2)K$.

(17) En utilisant les contraintes de l'algorithme des plus proches voisins. Montrer que le choix d'hyperparamètres valides est controlé par les relations suivantes :

$$\begin{cases} 1 \leq K \leq \lfloor \sqrt{A} \rfloor \\ 1 \leq n_1 \leq \lfloor \frac{A}{K} \rfloor - 1 \\ 1 \leq n_2 \leq \lfloor \frac{A}{K} \rfloor - 1 \\ K \leq n_1 + n_2 \leq \lfloor \frac{A}{K} \rfloor \end{cases}$$

avec A le coût maximum autorisé.

Résulte directement du fait que le nombre de voisins doit être inférieur à la taille du jeu de données et que $(n_1 + n_2)K \leq A$.

On peut montrer que le nombre d'hyperparamètres varie comme $A^{3/2}$. Plutôt que de les tester tous, nous allons adopter une stratégie de recherche aléatoire dans l'espace des hyperparamètres. On se sert de la classe `RandomizedSearchCV` qui s'utilise quasiment comme `GridSearchCV` à la différence près qu'au lieu de spécifier tous les hyperparamètres à tester, on donne un objet possédant une méthode nommée `rvs` qui renvoie un hyperparamètre tiré au hasard.

(18) Compléter le code ci-après qui génère une distribution des paramètres à l'aide d'une fonction stochastique.

On pourra utiliser la fonction `np.random.randint`.

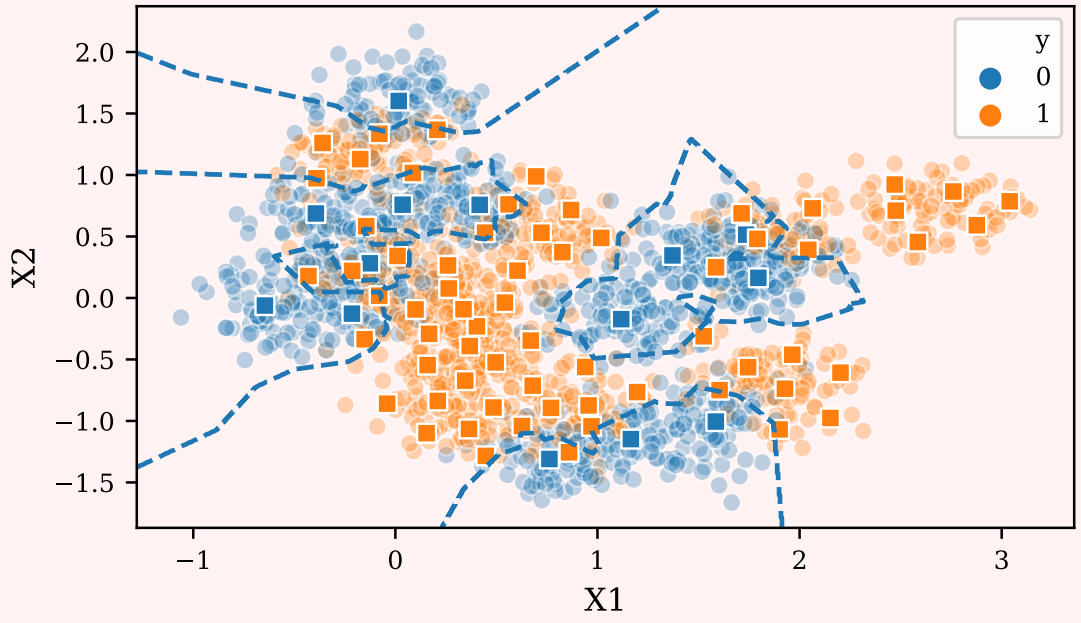
```
import math


class StochasticProtList:
    def __init__(self, n_neighbors, A):
        self.n_neighbors = n_neighbors
        self.A = A

    def rvs(self, *args, **kwargs):
        # Création de `n1` et `n2`
        n1 = ...
        n2 = ...

        # Retour du couple de prototypes ou rejet et appel récursif
        if ...:
            return ...
        else:
            return ...


A = 200

param_grid = [
    {"n_neighbors": [n_neighbors], "n_prototypes_list":
    ↪  StochasticProtList(n_neighbors, A)}
    for n_neighbors in range(1, math.floor(math.sqrt(A)) + 1)
]
```

```
In  [29]: import math


          class StochasticProtList:
              def __init__(self, n_neighbors, A):
                  self.n_neighbors = n_neighbors
                  self.A = A

              def rvs(self, *args, **kwargs):
                  # Création de `n1` et `n2`
                  n1 = np.random.randint(1, math.floor(self.A /
                  ↪  self.n_neighbors))
                  n2 = np.random.randint(1, math.floor(self.A /
                  ↪  self.n_neighbors))

                  # Retour du couple de prototypes ou rejet et appel récursif
                  if n1 + n2 < self.n_neighbors or n1 + n2 > math.floor(
                      self.A / self.n_neighbors
                  ):
                      return self.rvs(*args, **kwargs)
                  else:
                      return [n1, n2]


          A = 200

          param_grid = [
              {"n_neighbors": [n_neighbors], "n_prototypes_list":
              ↪  StochasticProtList(n_neighbors, A)}
              for n_neighbors in range(1, math.floor(math.sqrt(A)) + 1)
          ]
```

(19) En utilisant la variable `param_grid` définie précédemment, créer un objet de classe `RandomizedSearchCV`. On utilisera l'argument `n_iter` pour spécifier le nombre de jeu de d'hyperparamètres à échantillonner.

```
In  [30]: from sklearn.model_selection import RandomizedSearchCV

          knp = NearestPrototypes()
          search = RandomizedSearchCV(knp, param_grid, scoring="accuracy",
          ↪  n_iter=200)
          search.fit(X, y)
          search.best_params_
Out [30]: {'n_neighbors': 2, 'n_prototypes_list': [14, 65]}
In  [31]: best = search.best_estimator_
          df = pd.DataFrame(
              dict(X1=best.prototypes_[:, 0], X2=best.prototypes_[:, 1],
              ↪  y=best.labels_)
          )
          sns.scatterplot(x="X1", y="X2", hue="y", data=Xy, alpha=0.3,
          ↪  legend=False)
Out [31]: <matplotlib.axes._subplots.AxesSubplot object at 0x11c0ce7d0>
In  [32]: sns.scatterplot(x="X1", y="X2", hue="y", marker="s", data=df)
Out [32]: <matplotlib.axes._subplots.AxesSubplot object at 0x11c0ce7d0>
```

```
In [33]: add_decision_boundary(best)
         plt.show()
```



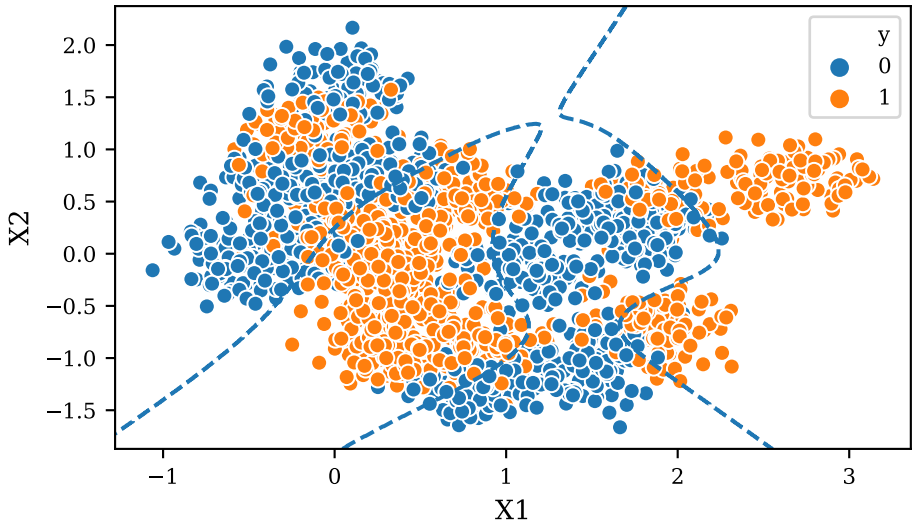


FIGURE 1 – Frontière de décision optimale (au sens de Bayes). Erreur de Bayes : $\simeq 13.81\%$