SY09 Printemps 2020
TP 10 — Régression logistique

La régression logistique est implémentée dans `scikit-learn` à travers la classe `LogisticRegression` et est disponible avec l'instruction suivante

```
from sklearn.linear_model import LogisticRegression
```

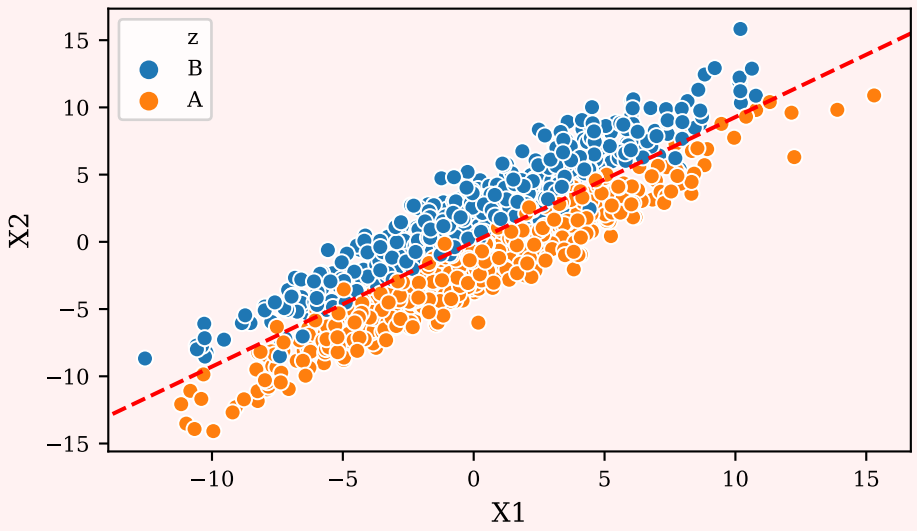
Les arguments intéressants lors de l'instanciation de la classe sont les suivants :
- `penalty` : le type de régularisation ajoutée à la fonction objectif à minimiser. Par défaut, une régularisation ℓ_2 est utilisée. Pour pouvoir comparer à notre implémentation on utilisera `penalty="none"`. Sinon, il est en effet conseillé d'utiliser la régularisation par défaut.

Après apprentissage, les attributs suivants sont disponibles :
- `coef_` : les coefficients appliqués aux descripteurs,
- `intercept_` : le coefficient constant.

1 Régression logistique

(1) Apprendre un modèle de régression logistique sur le jeu de données présent dans le fichier `Synth1-2000.csv` du TP07 et visualiser la frontière de décision.

```
In [1]: Xy =
        ↪ pd.read_csv("../TP09_Analyse_discriminante_de_donnees_gaussiennes/data/SynthPara_n1000
        X = Xy.iloc[:, :-1]
        y = Xy.iloc[:, -1]

        from sklearn.linear_model import LogisticRegression
        cls = LogisticRegression(penalty="none")
        cls.fit(X, y)
        sns.scatterplot(x="X1", y="X2", hue="z", data=Xy)
        add_decision_boundary(cls)
        plt.show()
```



(2) En utilisant les attributs `coef_` et `intercept_`, retrouver la frontière de décision.

Le modèle logistique s'écrit

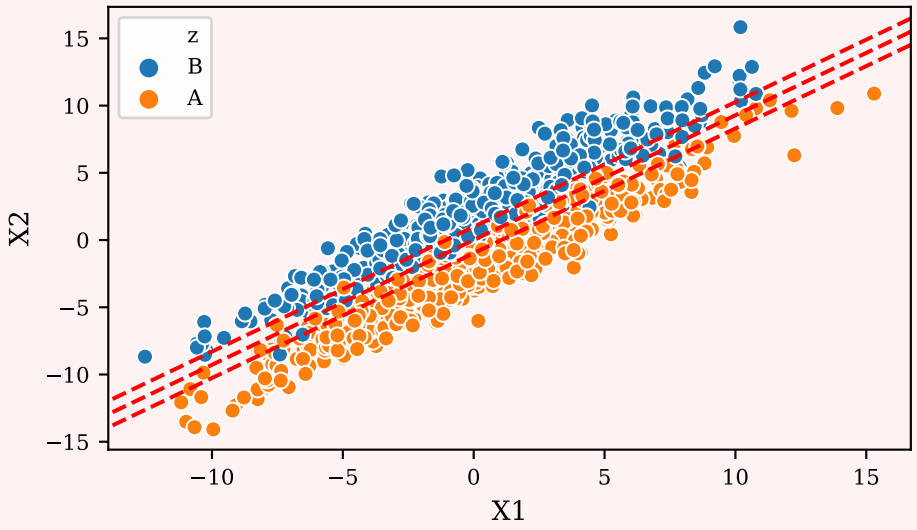
$$\sigma^{-1}(p) = w_0 + w_1 x + w_2 y,$$

avec $w_0 = -0,0033986$, $w_1 = -2,0777143$ et $w_2 = 2,2389711$. On retrouve approximativement la frontière de décision $y = x$.

(3) On souhaite ajouter des lignes de niveaux correspondant aux probabilités 0.1 et 0.9. En adaptant l'argument `levels` qui porte sur la partie linéaire, tracer ces lignes de niveaux.

On pourra utiliser la fonction `logit` du module `scipy.special` ou calculer directement les niveaux correspondant.

```
In [2]: sns.scatterplot(x="X1", y="X2", hue="z", data=Xy)
        from scipy.special import logit
        levels = logit(np.array([.1, .5, .9]))
        add_decision_boundary(cls, levels=levels)
        plt.show()
```



2 Régression logistique quadratique

Il est possible de généraliser le modèle de régression logistique de manière très simple. La stratégie consiste à transformer les données dans un espace plus complexe, dans lequel les classes peuvent être séparées par un hyperplan. La régression logistique est alors effectuée dans cet espace.

Le modèle ainsi défini est donc plus flexible (il permet de construire une frontière de décision plus complexe) ; mais il peut également s'avérer moins robuste que le modèle classique déterminé dans l'espace des caractéristiques initiales, puisqu'il nécessite d'estimer davantage de paramètres.

Par exemple, dans le cas où les individus sont décrits par les variables naturelles X^1, X^2 et X^3, la régression logistique quadratique consiste à apprendre un modèle de régression logistique classique dans l'espace $\mathcal{X}^2 = \{X^1, X^2, X^3, X^1X^2, X^1X^3, X^2X^3, (X^1)^2, (X^2)^2, (X^3)^2\}$, plutôt que dans l'espace $\mathcal{X} = \{X^1, X^2, X^3\}$ constitué des trois variables naturelles.

`Scikit-learn` permet de réaliser tout type de transformations de données qui peuvent être interprétées comme un modèle qu'on apprend avant de soumettre les données en entrée et récupérer la sortie. La classe `PCA` est un exemple d'un tel modèle. L'appel de la méthode `fit_transform` permet d'apprendre le modèle et de renvoyer les données transformées.

Les transformations polynômiales (et donc quadratiques) sont déjà implémentées dans `scikit-learn` grâce au modèle `PolynomialFeatures` disponible grâce à l'instruction suivante

```
from sklearn.preprocessing import PolynomialFeatures
```

Cette classe prend en argument le degré de l'expansion polynômiale à réaliser. Par example, le modèle suivant

```
poly = PolynomialFeatures(degree=3)
```

réalise l'expansion polynômiale de degré 3 lors de l'appel à `fit_transform`.

```
In  [3]: X = np.array([[1], [2], [3]])
         from sklearn.preprocessing import PolynomialFeatures
         poly = PolynomialFeatures(degree=3)
         poly.fit_transform(X)
Out [3]: array([[ 1.,  1.,  1.,  1.],
                [ 1.,  2.,  4.,  8.],
                [ 1.,  3.,  9., 27.]])
```

④ Réaliser l'expansion polynômiale de degré 2 du jeu de données et apprendre une régression logistique sur le nouveau jeu de données.

```
In  [4]: Xy = pd.read_csv("../TP07_K_plus_proches_voisins/data/Synth1-2000.csv")
         X = Xy.iloc[:, :-1]
         y = Xy.iloc[:, -1]

         poly = PolynomialFeatures(degree=2)
         Y = poly.fit_transform(X)
         cls = LogisticRegression()
         cls.fit(Y, y)
Out [4]: LogisticRegression(C=1.0, class_weight=None, dual=False,
         ↪  fit_intercept=True,
                            intercept_scaling=1, l1_ratio=None, max_iter=100,
                            multi_class='auto', n_jobs=None, penalty='l2',
                            random_state=None, solver='lbfgs', tol=0.0001,
                            ↪  verbose=0,
                            warm_start=False)
```

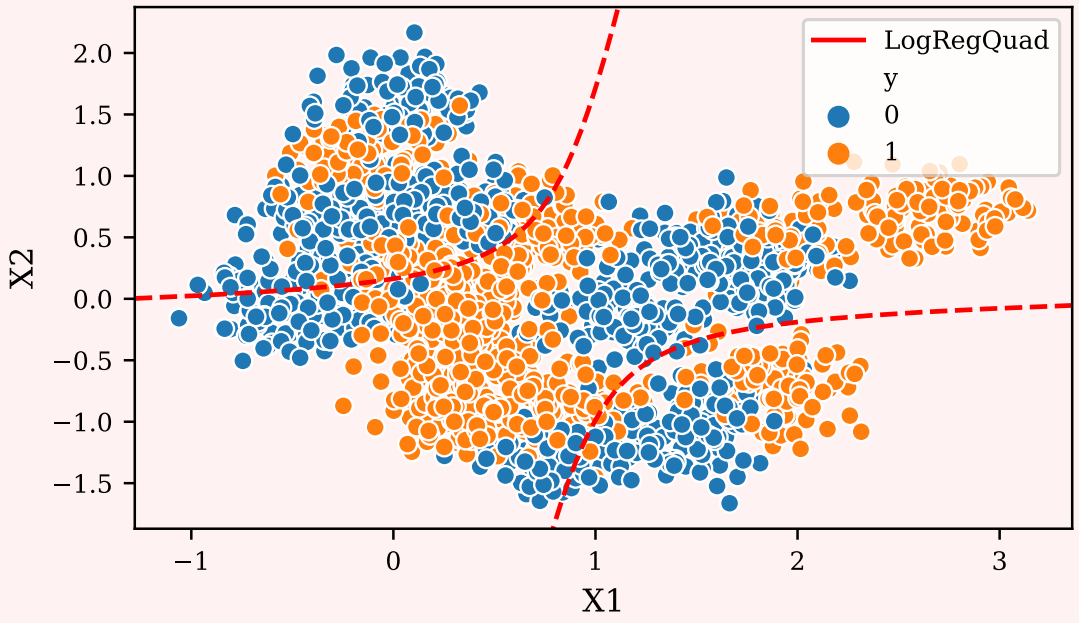
Afin de grouper ces deux opérations successives au sein d'un même modèle qui réalisera la transformation ainsi que l'apprentissage de la régression logistique lors d'un seul appel à une méthode `fit`, on va utiliser un *pipeline*. On peut créer un *pipeline* en utilisant la fonction `make_pipeline` et en spécifiant la liste des modèles à appliquer successivement.

```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(cls1, cls2, cls3)
pipe.fit(X, y)
```

On peut alors utiliser le méthode `predict` sur le *pipeline* `pipe` de façon transparente sans avoir à nous soucier des transformations intermédiaires.

(5) Créer un *pipeline* réalisant l'expansion polynômiale suivie de la régression logistique et visualiser dans le plan l'algorithme de discrimination résultant avec la fonction `add_decision_boundary`.

```
In [5]: from sklearn.pipeline import make_pipeline
        poly = PolynomialFeatures(degree=2)
        cls = LogisticRegression()
        pipe = make_pipeline(poly, cls)

        pipe.fit(X, y)
        sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
        add_decision_boundary(pipe, label="LogRegQuad")
        plt.show()
```



(6) Quelle est la forme des frontières de décision ?

La fonction de décision d'une régression logistique est linéaire en les prédicteurs qui sont une expansion polynômiale de degré 2. On a donc des frontières de décision qui sont des coniques.

3 Implémentation de la régression logistique binaire

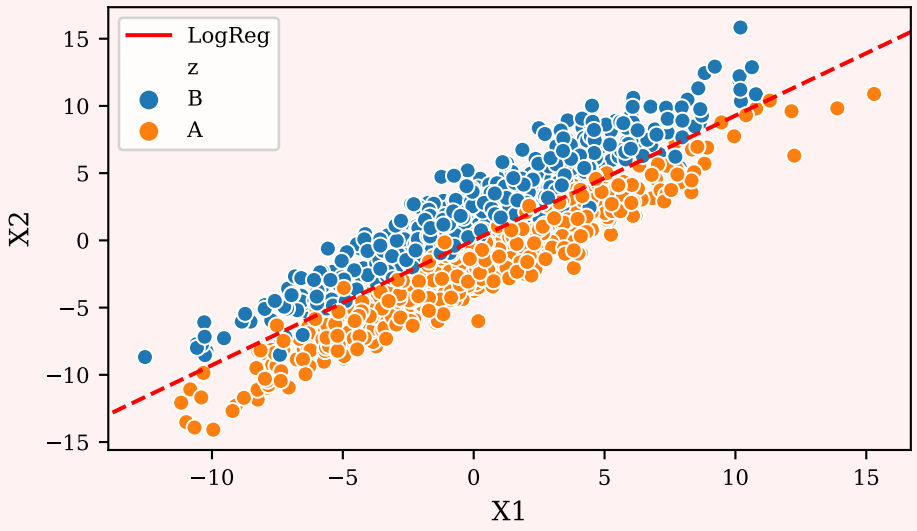
Dans cette section, on se propose d'implémenter une régression logistique binaire. On va utiliser la méthode de Newton-Raphson décrite en détails dans le polycopié de cours.

On pourra comparer avec les coefficients obtenus pour l'implémentation de `scikit-learn` avec les instructions suivantes.

```
from sklearn.linear_model import LogisticRegression as
↪  SklearnLogisticRegression
sk_cls = SklearnLogisticRegression(penalty="none")
sk_cls.fit(X, y)
sk_cls.coef_
sk_cls.intercept_
```

(7) Compléter le fichier `src/logistic_regression.py`.

```
In  [6]: from sklearn.linear_model import LogisticRegression as
         ↪  SklearnLogisticRegression
         sk_cls = SklearnLogisticRegression(penalty="none")
         sk_cls.fit(X, y)
         sk_cls.coef_
Out [6]: array([[-2.07771425,  2.23897111]])
In  [7]: sk_cls.intercept_
Out [7]: array([-0.00339858])
In  [8]: cls = LogisticRegression()
         cls.fit(X, y)

         np.isclose(sk_cls.coef_, cls.coef_)
Out [8]: array([[ True,  True]])
In  [9]: np.isclose(sk_cls.intercept_, cls.intercept_)
Out [9]: array([False])
In  [10]: sns.scatterplot(x="X1", y="X2", hue="z", data=Xy)
          add_decision_boundary(cls, label="LogReg")
          plt.show()
```



4 Régression logistique sur données transformées

Dans cette section, nous allons appliquer une régression logistique sur des données qui vont être d'abord transformées de la manière suivante :

1. on se donne k points parmi le jeu de données qu'on appelle des centres,

2. pour chaque centre, on génère les distances de chaque exemple du jeu de données à ce centre,
3. on rassemble ces k caractéristiques dans un nouveau jeu de données.

(8) En utilisant le jeu de données présent dans le fichier `Synth1-2000.csv` du TP07, déterminer k centres à l'aide de l'algorithme des *k-means*.

```
In [11]: from sklearn.cluster import KMeans
         cls = KMeans(n_clusters=20)
         cls.fit(X)
         centers = cls.cluster_centers_
```

(9) Former un nouveau jeu de données en calculant toutes les inter-distances entre le jeu de données d'origine et les centres calculés par la méthode des *k-means*.

On pourra utiliser la fonction `pairwise_distances`.

```
In [12]: from sklearn.metrics import pairwise_distances
         Y = pairwise_distances(X, centers)
```

(10) À l'aide du transformeur `FunctionTransformer`, créer un modèle `scikit-learn` qui réalise la transformation de la question précédente.

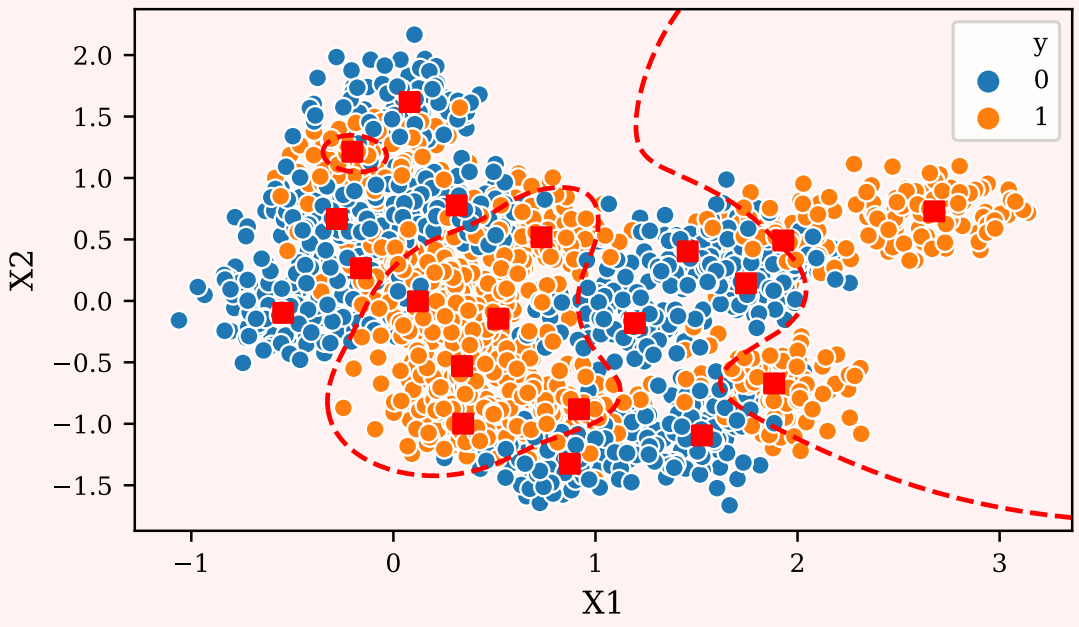
On pourra s'inspirer du code suivant

```
from sklearn.preprocessing import FunctionTransformer
from sklearn.metrics import pairwise_distances


def distances_to_centers(centers, metric="euclidean"):
    def distances_to_centers0(X):
        # Calcul des inter-distances entre `X` et `centers`
        return ...
    return distances_to_centers0

# Fonction qui prend en argument un jeu de données et le transforme.
func = distances_to_centers(centers)

# Création d'un modèle Scikit-learn qui réalise la transformation
# voulue.
transformer = FunctionTransformer(func)
```

```
In [13]: from sklearn.preprocessing import FunctionTransformer
         from sklearn.metrics import pairwise_distances


         def distances_to_centers(centers, metric="euclidean"):
             def distances_to_centers0(X):
                 # Calcul des inter-distances entre `X` et `centers`
                 return pairwise_distances(X, centers, metric=metric)
             return distances_to_centers0

         # Fonction qui prend en argument un jeu de données et le transforme.
         func = distances_to_centers(centers)

         # Création d'un modèle Scikit-learn qui réalise la transformation
         # voulue.
         transformer = FunctionTransformer(func)
```

(11) Créer un *pipeline* `scikit-learn` pour définir un modèle qui réalise d'abord la transformation précédente et applique ensuite une régression logistique.

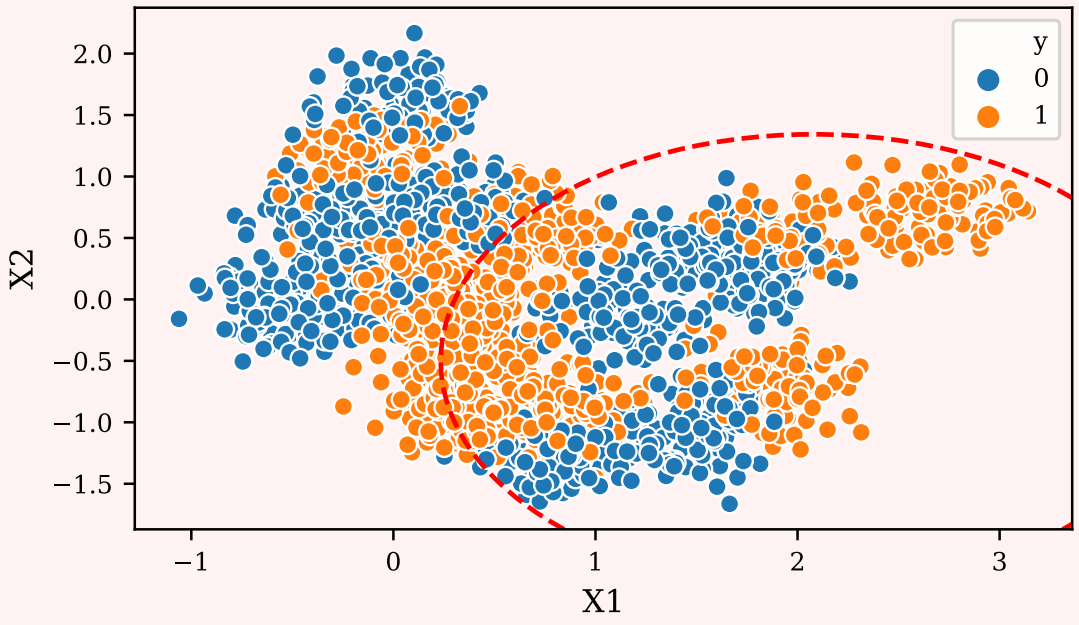
```
In [14]: from sklearn.pipeline import make_pipeline

         pipe = make_pipeline(transformer, LogisticRegression())
         pipe.fit(X, y)
         ax = sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
         plt.scatter(*centers.T, marker="s", c="r")
         add_decision_boundary(pipe)
         plt.show()
```



(12) Qu'aurait-on obtenu comme frontière de décision si au lieu de prendre la distance euclidienne on avait pris le carré de cette dernière ?

La fonction de décision est linéaire en les données transformées qui sont ici quadratiques en les données de départ. On se retrouve donc avec une conique.

```
In [15]: func = distances_to_centers(centers, metric="sqeuclidean")
         transformer = FunctionTransformer(func)
         pipe = make_pipeline(transformer, LogisticRegression())
         pipe.fit(X, y)
         sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
         add_decision_boundary(pipe)
         plt.show()
```



La méthode précédente présente l'inconvénient d'avoir à effectuer un algorithme des *k-means* avant de construire le *pipeline*. Il est donc impossible d'utiliser `scikit-learn` pour rechercher automatiquement l'hyperparamètre fixant le nombre de centres.

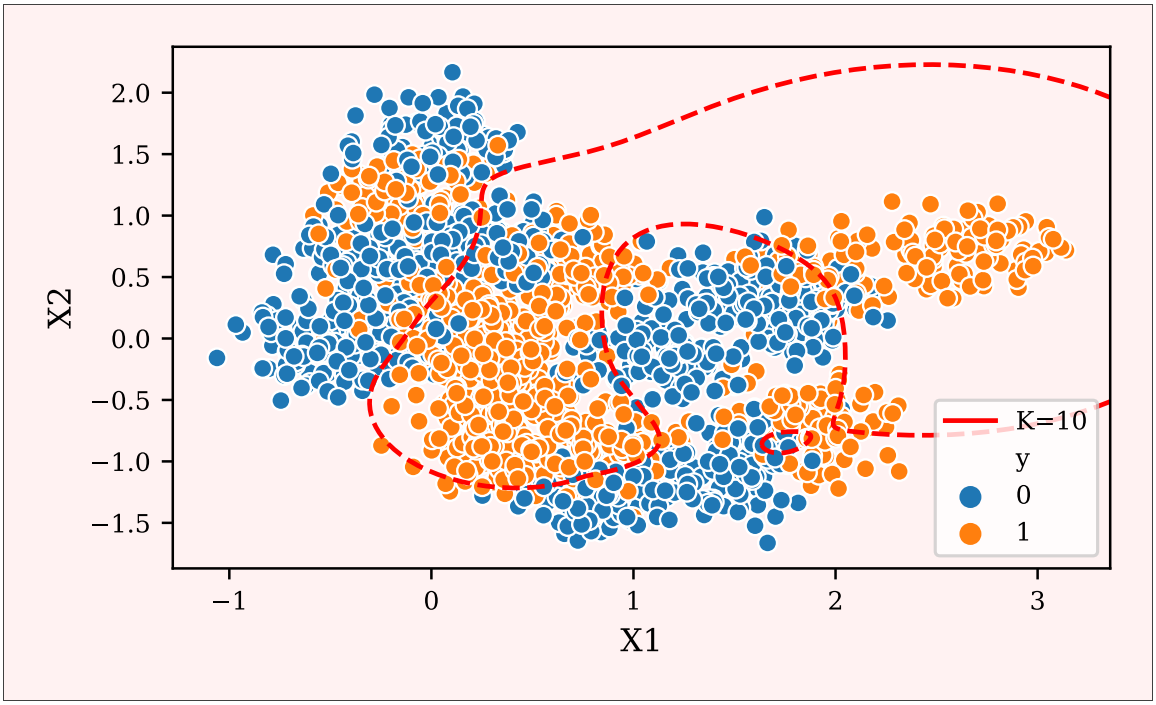
Afin d'inclure cet apprentissage dans l'apprentissage du *pipeline*, nous allons définir un *transformer* `scikit-learn` en héritant des classes `BaseEstimator` et `TransformerMixin`.

(13) Compléter le fichier `src/custom_transformer.py` qui définit un transformeur qui apprend lui-même les centres.

Voir fichier `src/custom_transformer.py`

(14) Créer un *pipeline* qui réalise la transformation proposée en calculant lui-même les centres suivie d'une régression logistique.

```
In [16]: pipe = make_pipeline(CustomTransformer(10), LogisticRegression())
         pipe.fit(X, y)
         sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
         add_decision_boundary(pipe, label="K=10")
         plt.show()
```



On peut maintenant utiliser `scikit-learn` pour rechercher le nombre optimal de centres par validation croisée en utilisant la fonction `GridSearchCV`.

Afin d'identifier les hyperparamètres de chaque modèle faisant partie du *pipeline*, il faut tout d'abord nommer chacun de ces modèles en utilisant la classe `Pipeline` au lieu de la fonction `make_pipeline` :

```
from sklearn.pipeline import Pipeline

pipe = Pipeline(
    (
        ("transformer1", Transformer1()),
        ("transformer2", Transformer2()),
        ("classifier", Classifier()),
    )
)
```

Les hyperparamètres de chaque modèle sont alors disponibles dans le *pipeline* sous le nom suivant

```
<nom du modèle>__<nom de l'hyperparamètre du modèle>
```

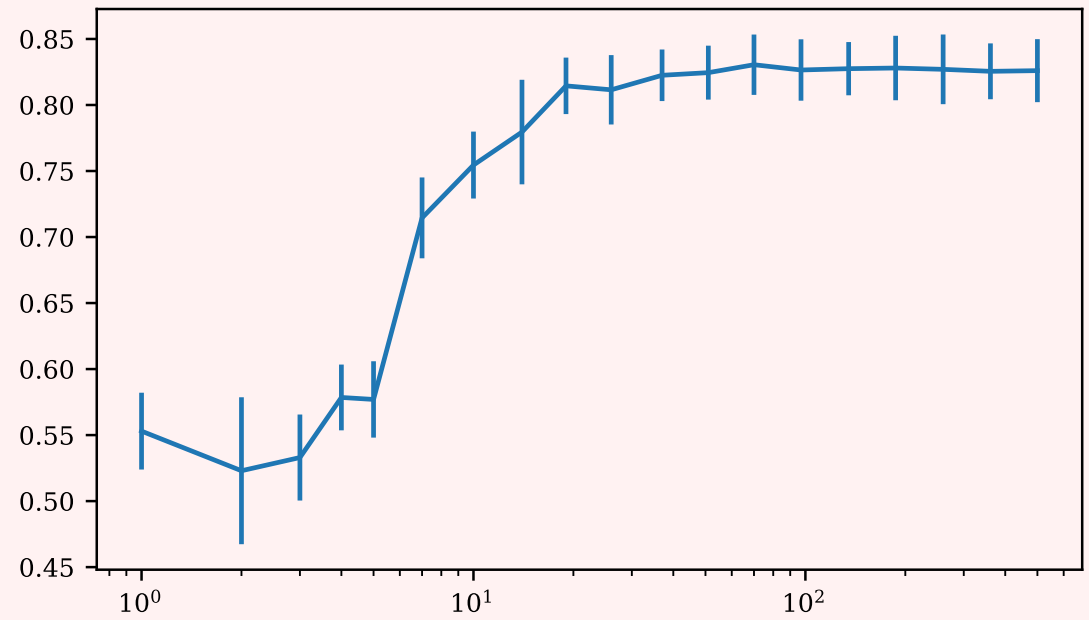
avec un double trait souligné séparant le nom du modèle donné dans la définition du *pipeline* et le nom d'un hyperparamètre de ce modèle.

(15) Définir un *pipeline* nommé ainsi qu'une grille de recherche sur le nombre de centres et déterminer le nombre de centres optimal par validation croisée.

Afin de déterminer le meilleur modèle, on utilisera la procédure suivante :

1. déterminer le meilleur modèle en considérant la moyenne des précisions sur les ensembles de validation et calculer un intervalle de confiance à 68% sur cette précision,
2. sélectionner le modèle le plus simple dont la moyenne des précisions est contenue dans cet intervalle de confiance.

```
In  [17]: from sklearn.model_selection import GridSearchCV
          from sklearn.pipeline import Pipeline

          pipe = Pipeline([("tf", CustomTransformer(10)), ("logreg",
          ↪  LogisticRegression())])
          n_clusters_list = np.unique(np.round(np.geomspace(1, 500,
          ↪  20)).astype(int))

          param_grid = {"tf__n_clusters": n_clusters_list}
          search = GridSearchCV(pipe, param_grid, scoring="accuracy", cv=10)
          search.fit(X, y)
          search.best_params_
Out [17]: {'tf__n_clusters': 70}
In  [18]: df = pd.DataFrame(
              (
                  dict(n_clusters=d["tf__n_clusters"], accuracy=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_clusters"], df["accuracy"], yerr=df["std"])
          plt.xscale("log")
          plt.show()

          # Recherche du modèle le plus simple dont la précision estimée est
          # contenue dans l'intervalle de confiance
```




```
In  [19]: Kopt = df.loc[df.accuracy.idxmax(), "n_clusters"]
          acc_opt = df.loc[df.accuracy.idxmax(), "accuracy"]
          std_opt = df.loc[df.accuracy.idxmax(), "std"]
          Kopt = min(df.loc[df.accuracy >= acc_opt - std_opt, "n_clusters"])
          Kopt
```

```
Out [19]: 19
In  [20]: pipe = make_pipeline(CustomTransformer(Kopt), LogisticRegression())
          pipe.fit(X, y)
          sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
          centers = pipe[0].centers
          plt.scatter(*centers.T, marker="s", c="r")
          add_decision_boundary(pipe, label=f"K={Kopt}")
          plt.show()
```