SY09 Printemps 2020
TP 11 — Arbres de décision

Une implémentation des arbres de décision est disponible dans `scikit-learn`. Il faut importer la classe `DecisionTreeClassifier`

```
from sklearn.tree import DecisionTreeClassifier
```

Les arguments intéressants lors de l'instanciation de la classe sont les suivants :
- `criterion` : le critère utilisé pour évaluer la qualité d'une séparation,
- `max_leaf_nodes` : le nombre maximum de feuilles autorisée lors de la construction de l'arbre,
- `max_depth` : la profondeur maximale autorisée lors de la construction de l'arbre,
- `max_features` : le nombre de caractéristiques à retenir de manière aléatoire pour la recherche de la meilleure séparation,
- `ccp_alpha` : paramètre λ utilisé pour l'élagage coût–complexité.

1 Arbres de décision

(1) Retrouver l'arbre construit dans le polycopié de cours en utilisant le jeu de données `data/toy2.csv`. Pour visualiser l'arbre construit, on pourra utiliser la fonction `plot_tree` avec

```
from sklearn.tree import plot_tree
```

(2) Retrouver le critère de Gini du nœud racine.

(3) Montrer que le critère de Gini de la première séparation qui est donc minimal pour tout autre séparation vaut $\simeq 0.347985$.

2 *Bagging*

Dans cette section, nous allons recréer manuellement les différentes étapes de la technique du « bagging » appliqué aux arbres de décision. La première étape consiste à créer plusieurs jeux de données avec la technique du « bootstrap ».

(4) Créer trois jeux de données échantillonnés par *bootstrap* grâce à la fonction `resample` disponible avec

```
from sklearn.utils import resample
```

On utilisera le jeu de données `Synth1-2000.csv` du TP07.

(5) Apprendre un arbre de décision limité à 50 feuilles maximum sur chaque jeu de données. Afficher leurs régions de décision respectives.

(6) Utiliser le classifieur par vote majoritaire pour combiner les décisions des trois arbres. On utilisera le modèle `VotingClassifier` disponible avec

```
from sklearn.ensemble import VotingClassifier
```

(7) Réaliser le même travail en utilisant cette fois la classe `BaggingClassifier`.

3 Forêt aléatoire

Une implémentation des forêts aléatoires est disponible dans `scikit-learn`. Il faut importer la classe `RandomForestClassifier`

```
from sklearn.ensemble import RandomForestClassifier
```

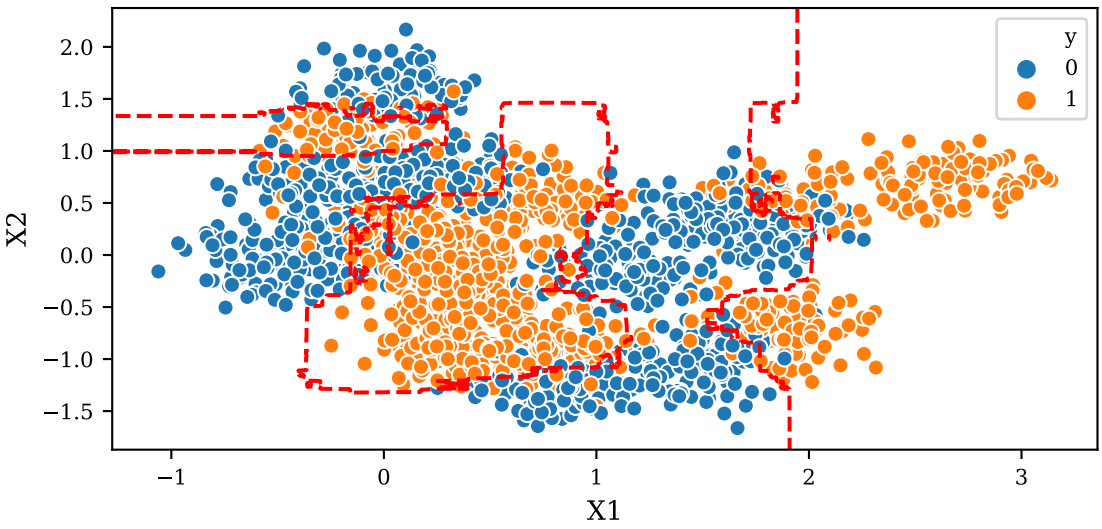
En plus des arguments des arbres de décision, on trouve l'argument `n_estimators` qui fixe le nombre d'arbres utilisés dans la forêt. De plus, l'argument `max_features` est cette fois fixé par défaut à $\sqrt{p}$ avec p le nombres de caractéristiques.

(8) Apprendre une forêt aléatoire sur le jeu de données précédent et visualiser le résultat.

In [1]:
```
from sklearn.ensemble import RandomForestClassifier

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

clf = RandomForestClassifier(n_estimators=100, max_leaf_nodes=50)
clf.fit(X, y)
sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
add_decision_boundary(clf)
plt.show()
```



4 Élagage coût–complexité et validation croisée

Dans cette section, on se propose d'implémenter la recherche du paramètre λ optimal par validation croisée telle que décrite aux pages 127 à 128 du polycopié de cours.

(9) On commence par déterminer les λ_k de la séquence $(\mathcal{A}_k, \lambda_k)$ des arbres appris sur l'ensemble d'apprentissage total $\mathcal{T}$.

La série des λ_k est disponible grâce à la fonction `cost_complexity_pruning_path`.

(10) Calculer les $\overline{\lambda}_k = \sqrt{\lambda_k \lambda_{k+1}}$.

(11) Compléter le générateur suivant qui génère les erreurs $\widehat{\varepsilon}_v\left(\mathcal{A}^{\overline{\lambda_k},(v)}\right)$ pour chaque k et chaque v.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from sklearn.utils import check_X_y


def decision_tree_cross_validation_accuracies(X, y, n_folds, lambdas):
    X, y = check_X_y(X, y)

    # Création d'un object `KFold` pour la validation croisée
    kf = ...

    for train_index, val_index in kf:
        # Création de `X_train`, `y_train`, `X_val` et `y_val`
        raise NotImplementedError

        for k, lmb in enumerate(lambdas):
            # Création d'un arbre avec un coefficient coût-complexité
            # égal à `lmb`
            clf = ...

            # Apprentissage sur l'ensemble d'apprentissage et calcul
            # de la précision sur l'ensemble de validation
            ...
            yield k, acc
```

(12) Créer le jeu de données issu du générateur précédent. Agréger les erreurs de tous les plis ensemble et afficher les erreurs de validation en fonction des $\overline{\lambda}_k$.

(13) En déduire le sous-arbre optimal obtenu par cette procédure et afficher sa frontière de décision.

5 Comparaison de la diversité des arbres

Dans cette section, on se propose d'illustrer le bénéfice du paramètre `max_features` en terme de diversité des arbres appris.

Pour évaluer la diversité des arbres, on choisit de comparer les ensembles de caractérisques sélectionnées pour effectuer des séparations jusqu'à une certaine profondeur.

On pourra utiliser la fonction suivante qui génère les indices des caractéristiques utilisées dans les séparations jusqu'à une certaine profondeur.

```
def features_depth(model, depth):
    tree = model.tree_
    def gen_id(i, depth):
        if tree.feature[i] >= 0:
            yield tree.feature[i]
        if depth != 0:
```

```
            yield from gen_id(tree.children_left[i], depth - 1)
            yield from gen_id(tree.children_right[i], depth - 1)

    yield from gen_id(0, depth)
```

(14) Afficher la distribution des caractéristiques retenues jusqu'à la profondeur 1 pour une forêt de 100 arbres en faisant varier `max_features`. Que peut-on constater ? Quelle est la distribution lorsque `max_features` vaut 1 ?

On utilisera le jeu de données `spams` disponible avec le TP09.