

# 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  $\lambda$  utilisé pour l'élagage coût-complexité.

### 1 Arbres de décision

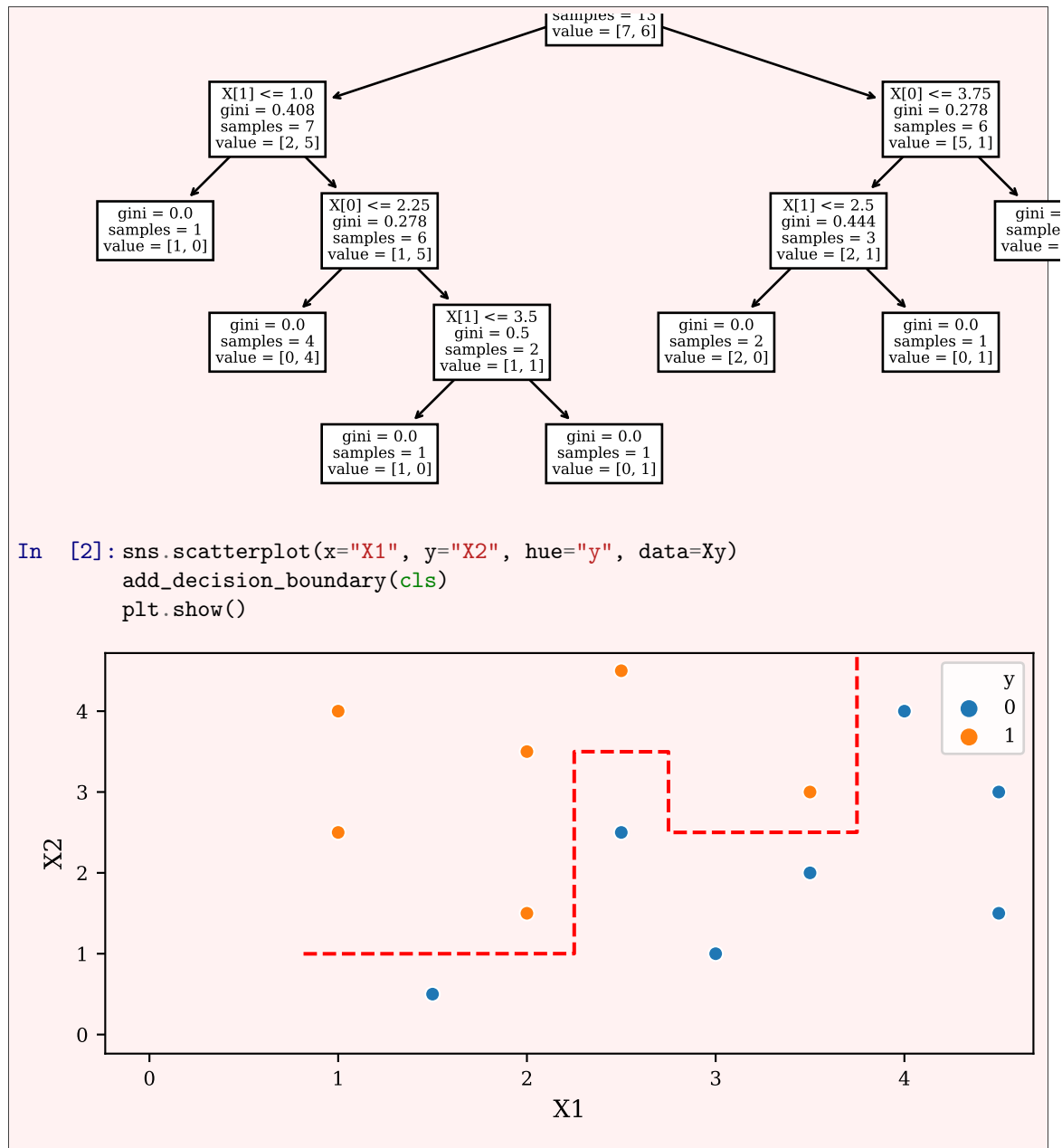
- ① Retrouver l'arbre construit dans le polycopié de cours en utilisant le jeu de données `data/toy2.csv`. Pour visualiser l'arbre construit, on pourra utiliser la fonction `plot_tree` avec

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

```
In [1]: from sklearn.tree import DecisionTreeClassifier
        from sklearn.tree import plot_tree

        Xy = pd.read_csv("data/toy2.csv")
        X = Xy.iloc[:, :-1]
        y = Xy.iloc[:, -1]

        cls = DecisionTreeClassifier()
        cls.fit(X, y)
        plot_tree(cls)
        plt.show()
```



② Retrouver le critère de Gini du nœud racine.

Les proportions des classes sont  $6/13$  et  $7/13$ . Le critère de Gini est donc  $6/13(1 - 6/13) + 7/13(1 - 7/13) = 84/169 \simeq 0.497$ .

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

On pondère les critères de Gini des deux cotés de la séparation par leur nombre de points

```
In [3]: gini_left = cls.tree_.impurity[cls.tree_.children_left[0]]
        gini_left
Out [3]: 0.40816326530612246
In [4]: gini_right = cls.tree_.impurity[cls.tree_.children_right[0]]
        gini_right
Out [4]: 0.27777777777777778
```

```
In [5]: count_left = cls.tree_.n_node_samples[cls.tree_.children_left[0]]
        count_left
Out [5]: 7
In [6]: count_right = cls.tree_.n_node_samples[cls.tree_.children_right[0]]
        count_right
Out [6]: 6
In [7]: (count_left * gini_left + count_right * gini_right) / (count_left +
        ↪ count_right)
Out [7]: 0.34798534798534797
```

## 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.

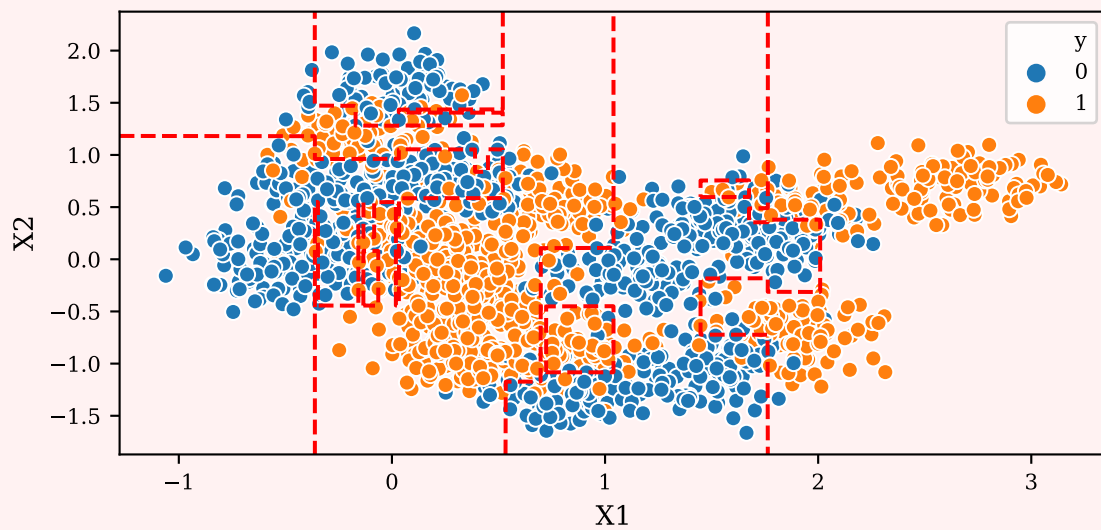
```
In [8]: Xy = pd.read_csv("../TP07_K_plus_proches_voisins/data/Synth1-2000.csv")
        X = Xy.iloc[:, :-1]
        y = Xy.iloc[:, -1]

        from sklearn.utils import resample
        X1, y1 = resample(X, y)
        X2, y2 = resample(X, y)
        X3, y3 = resample(X, y)
```

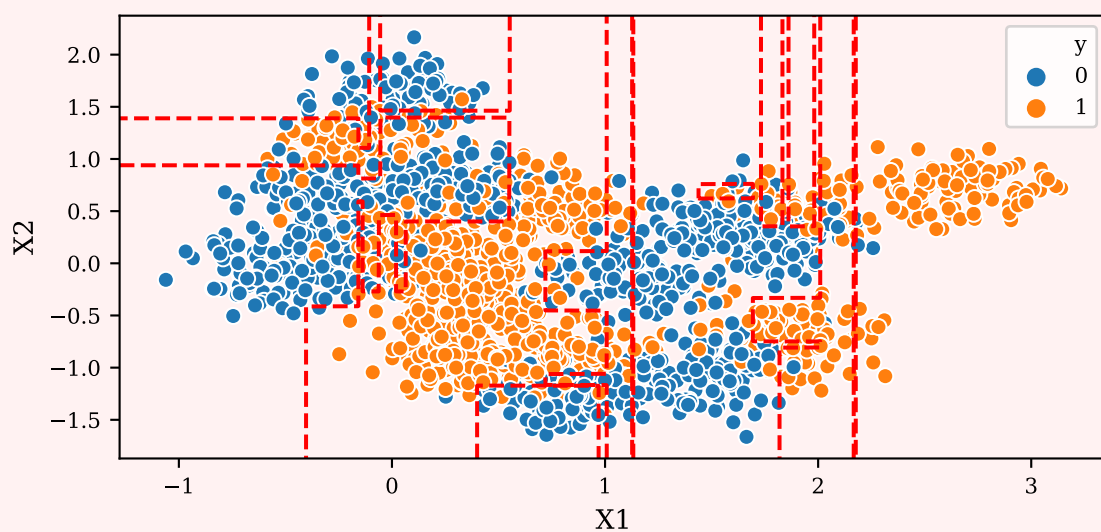
- ⑤ Apprendre un arbre de décision limité à 50 feuilles maximum sur chaque jeu de données. Afficher leurs régions de décision respectives.

```
In [9]: from sklearn.tree import DecisionTreeClassifier

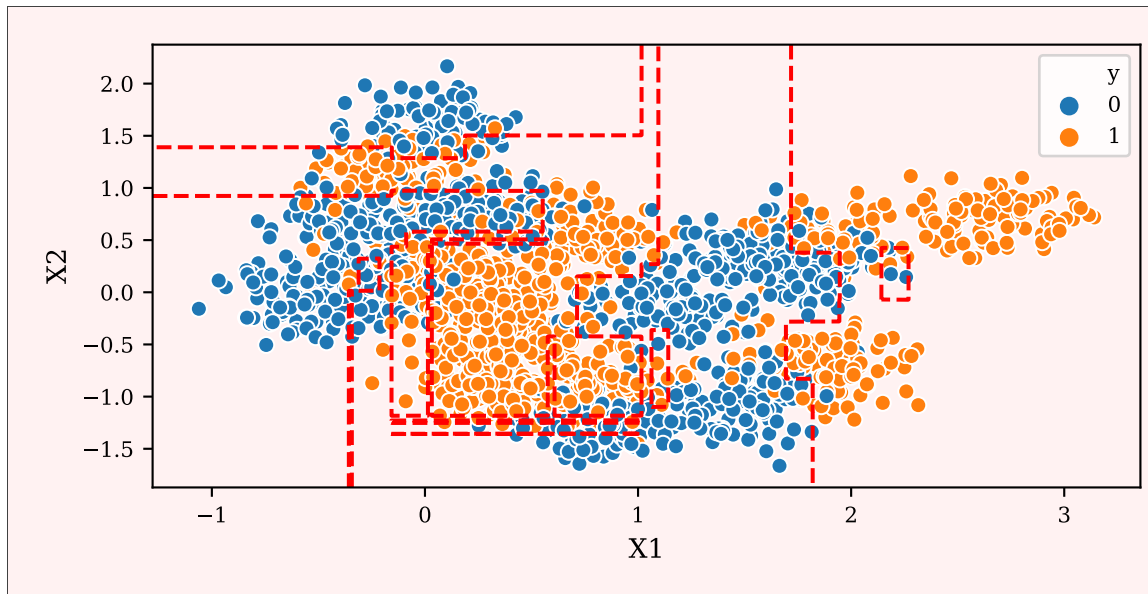
        params = {"max_leaf_nodes": 50}
        clf1 = DecisionTreeClassifier(**params)
        clf1.fit(X1, y1)
        sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
        add_decision_boundary(clf1)
        plt.show()
```



```
In [10]: clf2 = DecisionTreeClassifier(**params)
          clf2.fit(X2, y2)
          sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
          add_decision_boundary(clf2)
          plt.show()
```



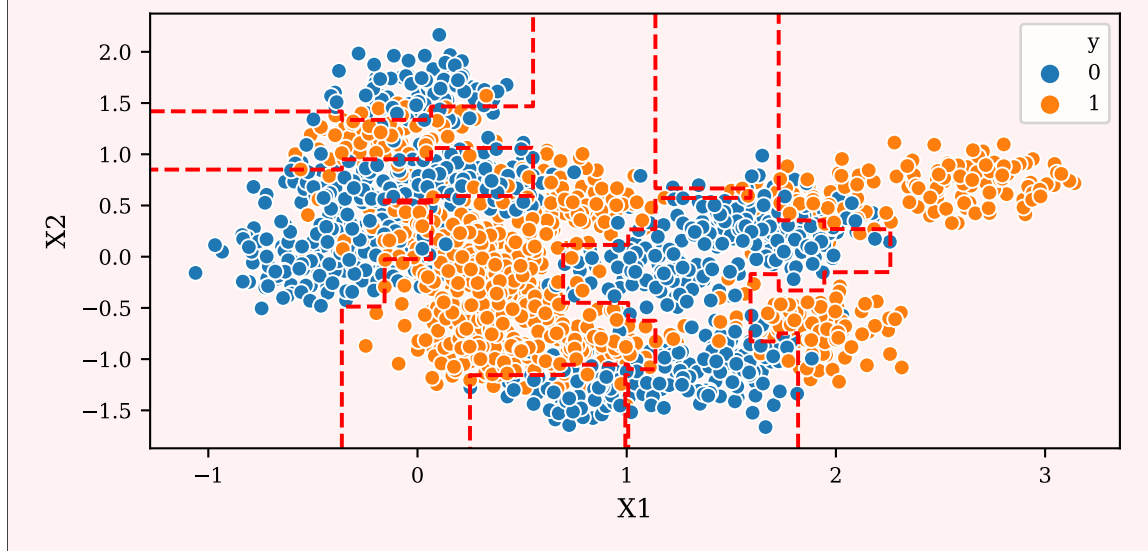
```
In [11]: clf3 = DecisionTreeClassifier(**params)
          clf3.fit(X3, y3)
          sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
          add_decision_boundary(clf3)
          plt.show()
```



- 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
```

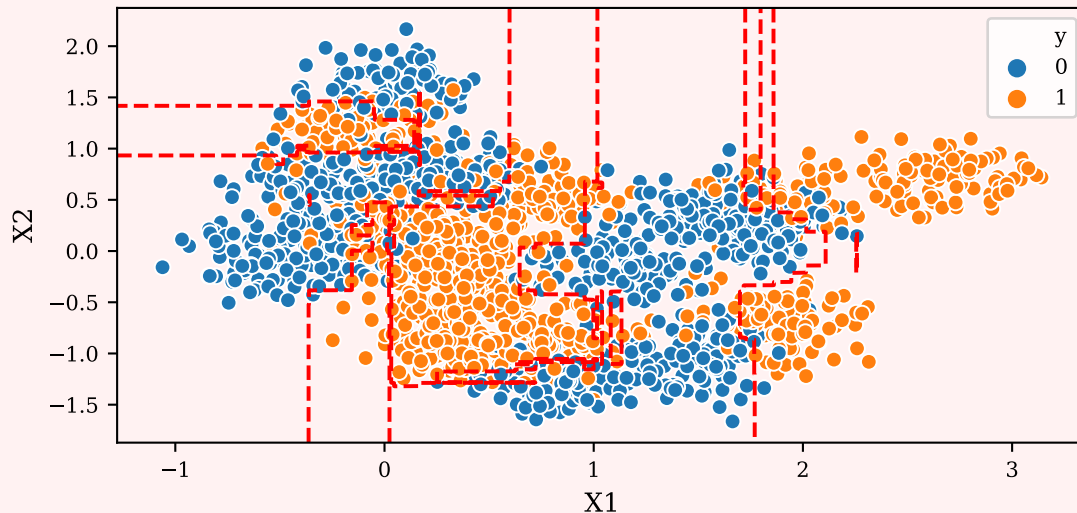
```
In [12]: from sklearn.ensemble import VotingClassifier
         eclf1 = VotingClassifier(
             estimators=[("tree1", clf1), ("tree2", clf2), ("tree3", clf3)]
         )
         eclf1.fit(X, y)
         sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
         add_decision_boundary(eclf1)
         plt.show()
```



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

```
In [13]: from sklearn.ensemble import BaggingClassifier
```

```
model = DecisionTreeClassifier(max_leaf_nodes=50)
clf = BaggingClassifier(model, n_estimators=3, bootstrap=True)
clf.fit(X, y)
sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
add_decision_boundary(clf)
plt.show()
```



### 3 Forêt aléatoire

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

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

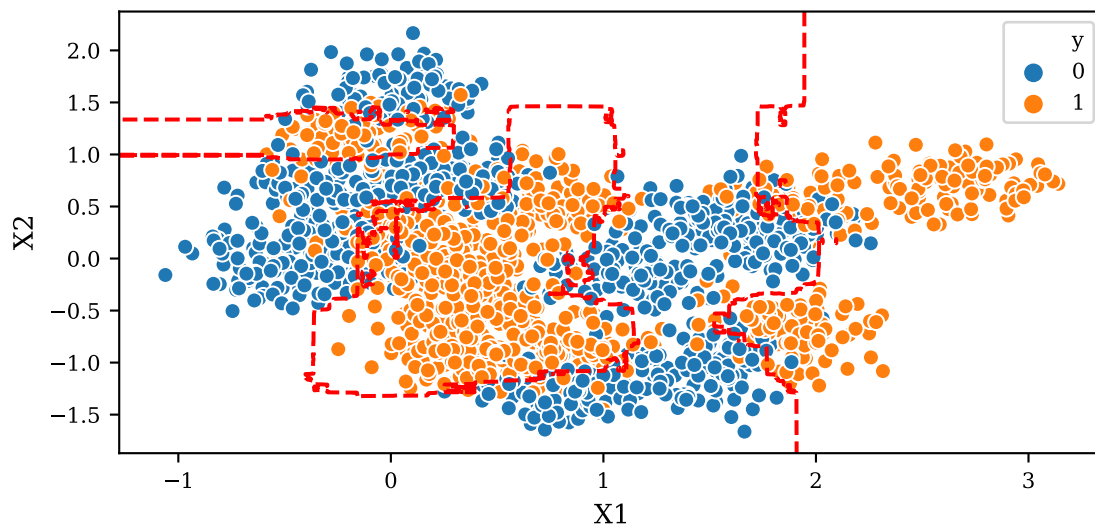
En plus des arguments des arbres de décision, on trouve l'argument `n_estimators` qui fixe le nombre d'arbres utilisés dans la forêt. De plus, l'argument `max_features` est cette fois fixé par défaut à  $\sqrt{p}$  avec  $p$  le nombre de caractéristiques.

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

```
In [14]: 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  $\lambda$  optimal par validation croisée telle que décrite aux pages 127 à 128 du polycopié de cours.

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

La série des  $\lambda_k$  est disponible grâce à la fonction `cost_complexity_pruning_path`.

```
In [15]: from sklearn.tree import DecisionTreeClassifier

Xy = pd.read_csv("../TP07_K_plus_proches_voisins/data/Synth1-2000.csv")

X = Xy.iloc[:, :-1]
y = Xy.iloc[:, -1]

clf = DecisionTreeClassifier()
clf.fit(X, y)

# Détermination des  $\lambda_k$  sur le jeu de données global
lambdas = clf.cost_complexity_pruning_path(X, y)["ccp_alphas"]
lambdas = np.unique(lambdas)
```

- ⑩ Calculer les  $\bar{\lambda}_k = \sqrt{\lambda_k \lambda_{k+1}}$ .

```
In [16]: lambdas_moy = np.sqrt(lambdas[:-1] * lambdas[1:])
```

- ⑪ Compléter le générateur suivant qui génère les erreurs  $\hat{\varepsilon}_v(\mathcal{A}^{\bar{\lambda}_k, (v)})$  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 objet `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
```

```
In [17]: 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 objet `KFold` pour la validation croisée
            kf = KFold(n_splits=n_folds, shuffle=True).split(X)

            for train_index, val_index in kf:
                # Création de `X_train`, `y_train`, `X_val` et `y_val`
                X_train = X[train_index, :]
                y_train = y[train_index]
                X_val = X[val_index, :]
                y_val = y[val_index]

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

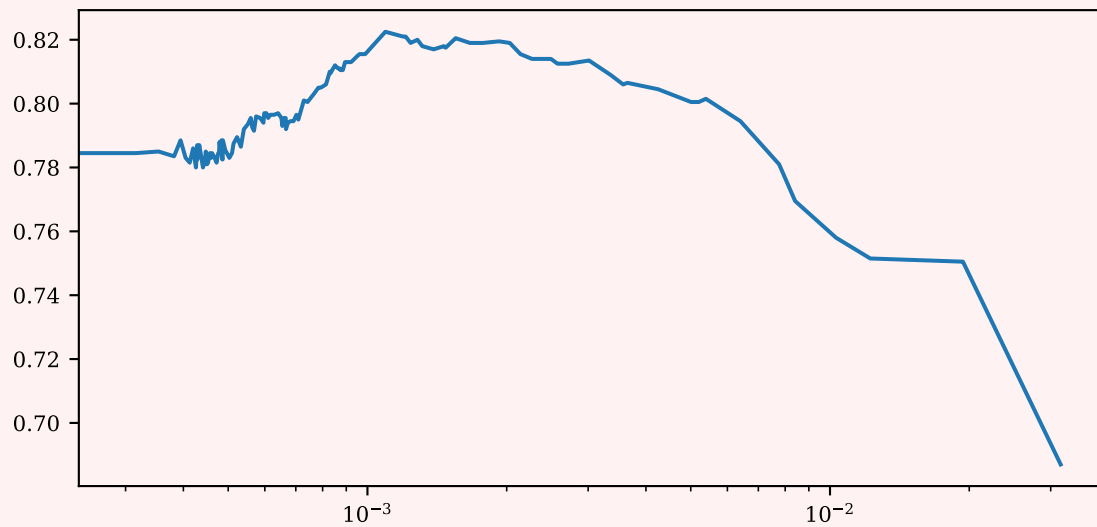
                    # Apprentissage sur l'ensemble d'apprentissage et calcul
                    # de la précision sur l'ensemble de validation
                    clf.fit(X_train, y_train)
                    pred = clf.predict(X_val)
                    acc = accuracy_score(y_val, pred)
                    yield k, acc
```

- 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  $\bar{\lambda}_k$ .



```
In [18]: n_folds = 10
```

```
df = pd.DataFrame(
    decision_tree_cross_validation_accuracies(X, y, n_folds,
        ↪ lambdas_moy),
    columns=["k", "accuracy"],
)
valid = df.groupby("k").mean()
plt.plot(lambdas_moy[valid.index], valid.accuracy)
plt.xscale('log')
plt.show()
```

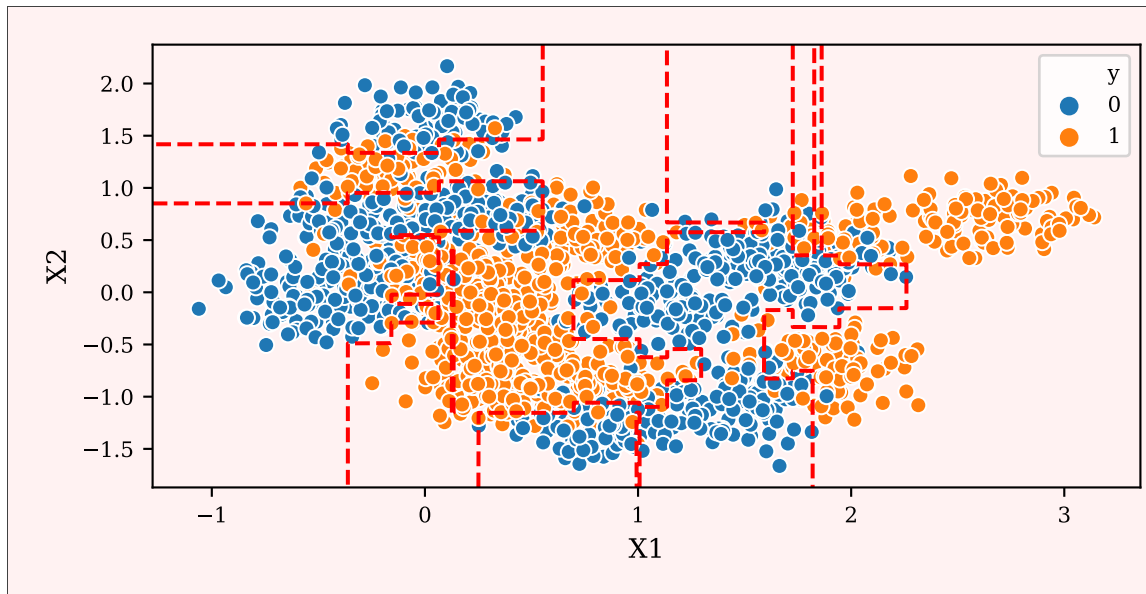


- 13) En déduire le sous-arbre optimal obtenu par cette procédure et afficher sa frontière de décision.

Le sous-arbre optimal est obtenu pour la valeur de  $\lambda$  qui maximise la précision.

```
In [19]: lmb = lambdas_moy[valid.accuracy.idxmax()]
```

```
clf = DecisionTreeClassifier(ccp_alpha=lmb)
clf.fit(X, y)
sns.scatterplot(x="X1", y="X2", hue="y", data=Xy)
add_decision_boundary(clf)
plt.show()
```



## 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 caractéristiques 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.

```

In [20]: from sklearn.ensemble import RandomForestClassifier
        from sklearn import datasets

spam =
    ↪ pd.read_csv("../TP09_Analyse_discriminante_de_donnees_gaussiennes/data/spambase.csv",
    ↪ index_col=0)
X = spam.iloc[:, :-1]
y = spam.iloc[:, -1]

models = [
    ("One feature", RandomForestClassifier(**params, max_features=1)),
    ("sqrt", RandomForestClassifier(**params)),
    ("All features", RandomForestClassifier(**params,
    ↪ max_features=None)),
]

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)

def gen_data(models, X, y):
    for name, model in models:
        model.fit(X, y)
        for ct in model.estimators_:
            for feat in features_depth(ct, 1):
                yield (name, feat)

df = pd.DataFrame(gen_data(models, X, y), columns=["Modèle",
    ↪ "Feature"])

g = sns.FacetGrid(df, col="Modèle")
g.map(sns.distplot, "Feature")
plt.show()

```

