

TP 1 - Part. 2

Support Vector Machines & Decision Trees

MADAD Sarra

NOUAR Manelle

BERNARDOU Elliott

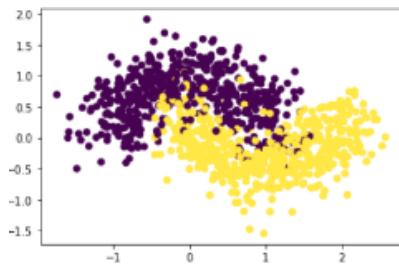
Enseignant : M. Lionel PREVOST

Matière : Machine Learning & Application

Introduction

Nous commençons par créer un jeu de données aléatoires binaires : nous avons 1000 observations pour 2 features. On ajoute du bruit car si on ne le met pas ou si on le laisse à 0, les données sont trop bien séparées (même non linéairement). Il sera donc difficile pour un SVM non linéaire de ne pas bien généraliser.

Elles sont donc non linéairement séparables, avec un bruit de 0.28.



Pas besoin de normaliser nos données (les centrer-réduire) ni de faire une PCA puisque nous n'avons que 2 descripteurs.

Le but ici est d'essayer de bien séparer ce problème bi-classes, à l'aide de modèles non linéaires adéquates.

Modèles utilisés :

- **SVM non linéaire à noyau gaussien**

Ce modèle a pour vocation de séparer les données en déterminant un hyperplan optimisé à l'aide de marges. Ici, il fait intervenir un kernel non linéaire capable de séparer notre type de données, ainsi que **2 paramètres à optimiser : C et Gamma**.

Le paramètre C de la classe SVC est le paramètre qui régit le degré de souplesse d'une marge. Par défaut, il a une valeur de 1.

- Plus la valeur de C est petite, plus les marges sont larges - ce qui peut conduire à plus de mauvaises classifications.
- Inversement, plus la valeur de C est grande, plus les marges du classificateur sont étroites - ce qui peut conduire à moins d'erreurs de classification mais une mauvaise généralisation.

En décidant de fixer C, on fait varier Gamma. On remarque que plus Gamma augmente, plus on passe en surapprentissage et par conséquent l'erreur en training tend vers 0 et **l'erreur en test décolle lorsque Gamma est trop haut (et celui en apprentissage plafonne au maximum)**.

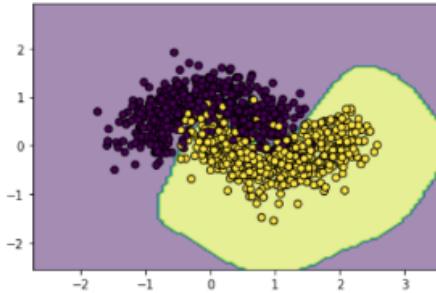
Gamma définit dans quelle mesure le modèle s'adapte aux données d'apprentissage, ce qui permet d'éviter l'overfitting. Le gamma définit la portée de l'influence d'un seul exemple de formation. Plus la valeur est faible, plus la portée d'un point de formation est grande.

Inversement, plus la valeur est grande, plus la portée du point de formation est faible.

Le nombre de vecteurs-supports (ce sont les points les plus proches de la séparatrice) joue un rôle important : il ne faut pas qu'il y ait trop de vecteurs-supports sinon cela voudrait dire que un trop grand nombre de points sont proches des frontières, ce qui peut mener à une mauvaise généralisation.

En utilisant une optimisation par recherche en grille, les paramètres optimaux et leur performance sont :

- **C = 1**
- **Gamma = 2**
 - **Accuracy en apprentissage = 0.86**
 - **Accuracy en test = 0.87**
 - **Temps d'inférence/exécution = 9e-06**
 - **Temps d'apprentissage = 10e-06**
 - **Nombre de vecteurs supports = 176**



- *Décision Trees*

L'arbre de décision est un diagramme de flux, et peut aider à prendre des décisions sur la base d'expériences passées. On doit chercher un attribut discriminant, lui attribuer un seuil et mettre en place un critère d'arrêt (indice de Gini, entropie...).

On choisit ici le critère “Entropy” et on l’entraîne selon deux schémas, avec **2 paramètres à optimiser (max_depth et min_samples_split)** :

- Faire varier max_depth et noter les performances
- Faire varier min_samples_split et noter les performances

L'entropie caractérise le degré de désorganisation, ou d'imprédictibilité, du contenu en information d'un système. Si on continue jusqu'à une entropie à 0 (cf. arbre complet) on est en sur-apprentissage. On va donc prendre des critères d'arrêt lors de l'apprentissage comme une entropie à une valeur minimum.

La "maximum tree depth" qui signifie profondeur maximale de l'arbre, il s'agit d'arrêter le développement de l'arbre une fois qu'il a atteint une certaine profondeur, cela évitera que l'arbre ne construise des branches avec trop peu d'exemples et donc permettra d'éviter un sur apprentissage (les noeuds sont développés jusqu'à ce que toutes les feuilles soient pures ou jusqu'à ce que toutes les feuilles contiennent moins de min_samples_split échantillon).

L'erreur en test décolle lorsque ce paramètre est trop haut (et celui en apprentissage plafonne au maximum).

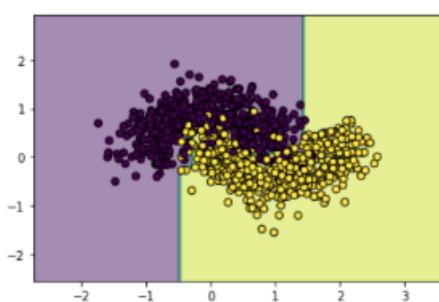
Le "minimum sample split" ou encore nombre d'exemples minimum pour un split consiste à ne pas splitter une branche si la décision concerne trop peu d'exemples. Cela permet également d'empêcher le surapprentissage.

L'erreur en test décolle lorsque ce paramètre est trop bas (et celui en apprentissage plafonne au maximum).

En utilisant une optimisation par recherche en grille, les paramètres optimaux et leur performance sont :

- **max_depth = 6**
- **min_samples_split = 20**
 - **Accuracy en apprentissage = 0.92**

- **Accuracy en test = 0.91**
- **Temps d'apprentissage = 3e-06**



- Extra Trees

Les Extra-Trees divisent en choisissant des noeuds seuils entièrement au hasard. Le principal avantage des arbres supplémentaires est la réduction du biais et du temps de calcul. Imaginons que l'on a un dataset de 1000 descripteurs, c'est compliqué/long de trouver le meilleur descripteur avec le meilleur seuil !

On utilise ici deux schémas :

- Lancement d'un seul extra trees 20 fois
- Lancement de 10, 20 et 50 extra-trees

On peut comparer le temps d'apprentissage et en combinant cela à la précision, on peut jauger si notre arbre de décision optimal ou nos extra-trees sont les plus performants.

Pour les Extra-Trees lancé en même temps : **la performance en test la plus intéressante est celle de 10 Extra-Trees avec un score de 90%. Le temps d'apprentissage associé est 0.000014.**

Pour notre Extra-Trees lancé 20 fois : **au bout de la 15ème fois, la performance en test la plus intéressante est à 90%. Le temps d'apprentissage associé est 9.663105e-07.**

Nous pouvons donc conclure qu'il est plus intéressant d'utiliser une combinaison d'Extra-Trees plutôt que de lancer plusieurs fois un même Extra-Tree (temps d'apprentissage moins long).

Conclusion

A l'issue de ce TP, nous pouvons dire que les 3 modèles sont efficaces pour traiter un modèle de classification non linéaire bi-classes, mais le plus performant au regard de la performance en test et du temps de calcul semble l'Extra-Tree.

Vous trouverez en annexe le notebook avec le code et les résultats.

TP 2 . Support vector Machines & Decision trees

```
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn.svm import LinearSVC

#on génère un jeu de données de 1000 observations
X,y = make_moons(n_samples=1000, shuffle=True, noise = 0.28)
print((X.shape)) #1000 données de 2 features

#on sépare nos données en entrainement/test
#70% training et 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

#on visualise nos données
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()
```

A scatter plot illustrating a binary classification problem. The x-axis ranges from -1.5 to 2.0, and the y-axis ranges from -1.5 to 1.0. Two classes of data points are shown: purple dots representing one class and yellow dots representing the other. A vertical decision boundary is drawn at approximately x = 0.5, where the model's output is zero. The region to the left of this boundary (x < 0.5) contains mostly purple dots, while the region to the right (x > 0.5) contains mostly yellow dots. Some overlap exists between the two classes, particularly around the decision boundary.

```
print(X)
print(y)

[[[-0.41844906  1.17863288]
 [ 0.82566809 -0.37591651]
 [ 1.68706743 -0.43966847]
 ...
 [ 0.05710037 -0.09659999]]
```

```

[ 1.29043767 -0.78295909]
[0 0 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1 0
1 1 0 0 0 1 1 0 1 0 1 0 0 1 0 1 1 1 0 0 0 0 0 0 1 1
1 0 1 0 0 1 0 0 1 1 0 0 1 0 0 0 1 1 1 1 1 0 0 0 0
0 1 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 0 0 0 1 1
0 1 0 1 0 1 1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 1 0 0 1 1
0 1 0 1 0 0 0 0 1 1 1 1 1 0 0 1 0 1 1 0 0 0 1 1 0 1 1 0 0 1
0 1 1 1 1 0 1 0 0 1 0 0 1 1 1 0 1 1 0 1 1 1 0 0 0 1 1 0 0 0 1
0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1 0 1 1 0 1 1 0 1 0 0 0 0 0
1 1 1 0 1 0 1 0 0 0 0 1 0 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1
1 0 0 0 0 1 0 1 1 1 0 1 1 0 0 1 0 1 0 1 0 0 1 0 0 1 0 1 1 0
0 1 0 0 0 0 1 1 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1
1 0 0 0 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 1 0 0 1 0 0 1 0 0 1
0 0 0 0 0 0 1 0 1 1 1 0 1 0 1 1 1 0 0 1 0 0 0 1 1 0 0 0 1 0 0 1
0 0 1 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 0 1 1 1 1 1 0 1 1 1 1 0
1 1 1 0 1 1 1 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 0
1 0 0 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0
1 1 1 0 1 0 0 0 1 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 1 0 1 1 1 0
0 1 1 1 1 1 0 0 0 1 0 0 1 0 1 1 1 1 0 1 1 1 0 1 0 1 1 0 1 1
0 0 0 1 1 1 0 1 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 1 0 1 1 0 1 0 1
0 0 0 1 0 1 1 0 0 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 0 0 1
1 1 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1 1 1 0 0 1 0 0
1 1 1 0 1 0 1 1 0 1 0 1 0 1 1 1 1 1 0 0 0 1 0 0 1 0 1 0 0 1 0 0
1 1 0 0 0 0 1 0 1 1 0 1 0 1 1 1 1 1 0 0 0 1 0 0 1 0 1 0 0 1 0 0
0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 0 1 0 1 0 0 0 1 0 0 0 1 0 0
1 1 0 0 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 0 1 1 0 1 0 0 0 0 0
1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0
0 1 0 1 0 0 0 0 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 0
1]

```

```

from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler

'''
#on normalise nos données => pas besoin ici
scaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.fit_transform(X_test)

'''

'\n#on normalise nos données => pas besoin ici\nscaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.fit_transform(X_test)'

from sklearn import svm
from sklearn import metrics
from sklearn.metrics import ConfusionMatrixDisplay

#créer une SVM avec un noyau gaussien de paramètre rbf
classifier = svm.SVC(kernel='rbf', C=10, gamma=0.01)

#entraîner la SVM sur le jeu d'entraînement
classifier.fit(X_train, y_train)

#prédiction des labels pour les données de test
y_pred_test = classifier.predict(X_test)

#prédiction des labels pour les données d'apprentissage
y_pred_app = classifier.predict(X_train)

#erreur en apprentissage = 1 - accuracy
print("Accuracy:",metrics.accuracy_score(y_train, y_pred_app))
print("Erreur en apprentissage :",(1-(metrics.accuracy_score(y_train, y_pred_app))))*100

#erreur en test = 1 - accuracy
print("Accuracy:",metrics.accuracy_score(y_test, y_pred_test))
print("Erreur en test :",(1-(metrics.accuracy_score(y_test, y_pred_test))))*100

Accuracy: 0.87
Erreure en apprentissage : 0.13
Accuracy: 0.86
Erreure en test : 0.14

```

```

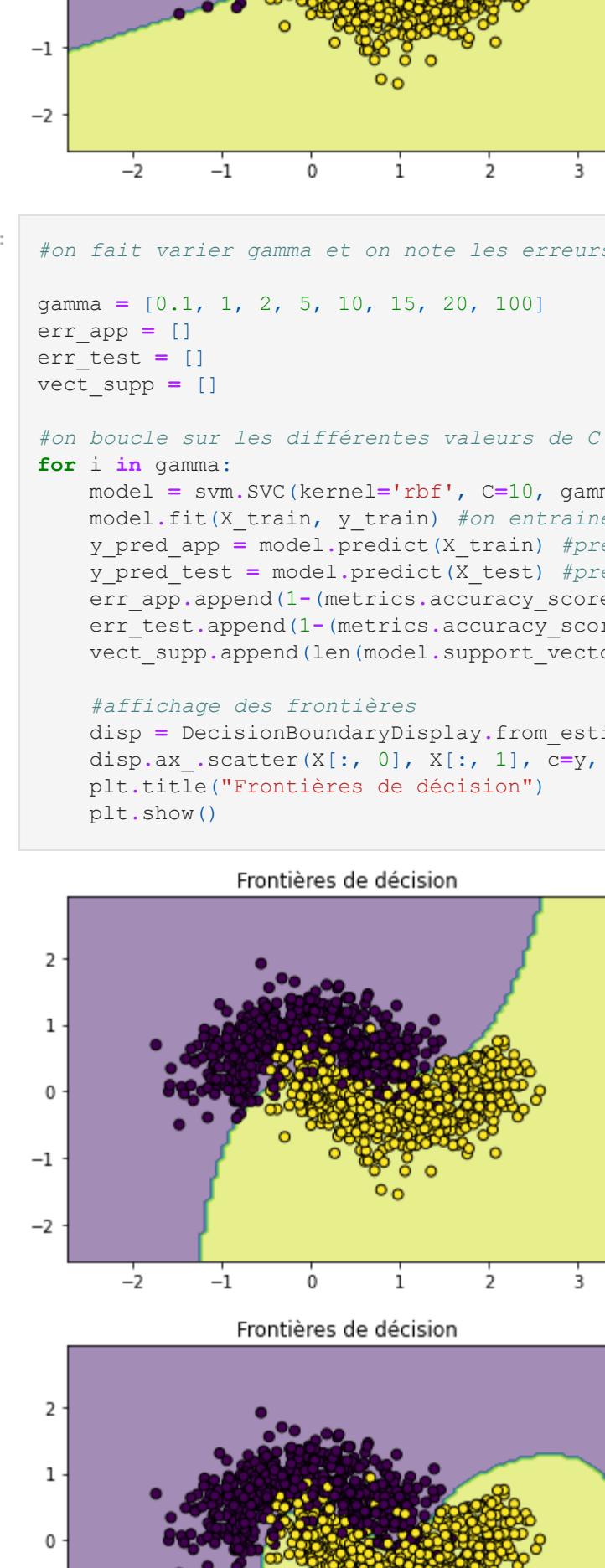
from sklearn.inspection import DecisionBoundaryDisplay

#on affiche le nombre de vecteurs supports
print("Il y a ",len(classifier.support_vectors_), "vecteurs supports")

#on dessine les frontières de décision
disp = DecisionBoundaryDisplay.from_estimator(
    classifier, X, response_method="predict",
    alpha=0.5,
)
disp.ax_.scatter(X[:, 0], X[:, 1], c=y, edgecolor="black")
plt.show()

Il y a 271 vecteurs supports.

```



The figure displays a scatter plot of data points colored by class (purple for one class, yellow for the other) against two features. A green shaded region represents the decision boundary, where the logistic regression model's predicted probability is above 0.5. The x-axis ranges from -2 to 3, and the y-axis ranges from -2 to 2.

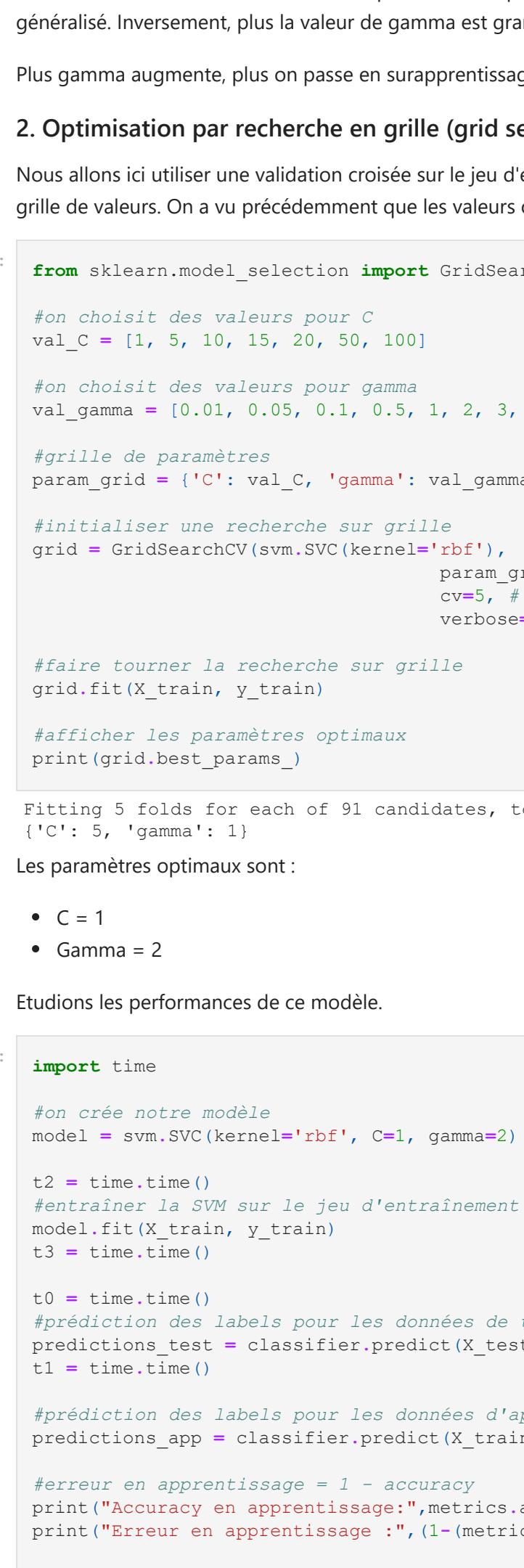
```
#résultats selon gamma
df = pd.DataFrame(list(zip(gamma, err_app, err_tes)))
df
```

	Gamma	Erreur en training	Erreur en test	Nombre de Vecteurs
0	0.1	0.097143	0.103333	
1	1.0	0.077143	0.086667	
2	2.0	0.074286	0.080000	
3	5.0	0.080000	0.086667	
4	10.0	0.071429	0.083333	
5	15.0	0.068571	0.086667	
6	20.0	0.067143	0.096667	
7	100.0	0.022857	0.130000	

Le paramètre C de la classe SVC est le paramètre qui régit le degré des fonctions de base. Les valeurs peuvent être considérées comme suit :

- Plus la valeur de C est petite, plus les marges sont larges - ce qui entraîne une mauvaise classification.
- Inversement, plus la valeur de C est grande, plus les marges sont étroites - ce qui entraîne une meilleure classification.

Gamma définit dans quelle mesure le modèle s'adaptera aux données.



```

#on affiche le nombre de vecteurs supports
print("Il y a ",len(model.support_vectors_), "vecteurs supports.")

#on affiche le temps d'inférence
total1 = (t3-t2)/len(X)
total = (t1-t0)/len(X)
print("Temps d'exécution/d'inférence : ", total)
print("Temps d'apprentissage : ", total1)

#on dessine les frontières de décision
disp = DecisionBoundaryDisplay.from_estimator(
    model, X, response_method="predict",
    alpha=0.5)
disp.ax_.scatter(X[:, 0], X[:, 1], c=y, edgecolor="k")
plt.show()

```

Accuracy en apprentissage: 0.8657142857142858
 Erreur en apprentissage : 0.13428571428571423
 Accuracy en test: 0.87
 Erreur en test : 0.13
 Il y a 176 vecteurs supports.
 Temps d'exécution/d'inférence : 8.987903594970703e-06
 Temps d'apprentissage : 9.974002838134766e-06

vecteurs-supports = ce sont les points les plus proches de la séparatrice

Il ne faut pas qu'il y ait trop de vecteurs-supports sinon cela voudrait dire que un trop grand nombre de points sont proches des frontières, ce qui peut mener à une mauvaise généralisation.

Temps d'inférence ?

Decision Trees

Fonctionnement :

L'arbre de décision est un diagramme de flux, et peut vous aider à prendre des décisions sur la base d'expériences antérieures.

Dans cet exemple, une personne va essayer de décider si elle doit aller à un spectacle de comédie ou non.

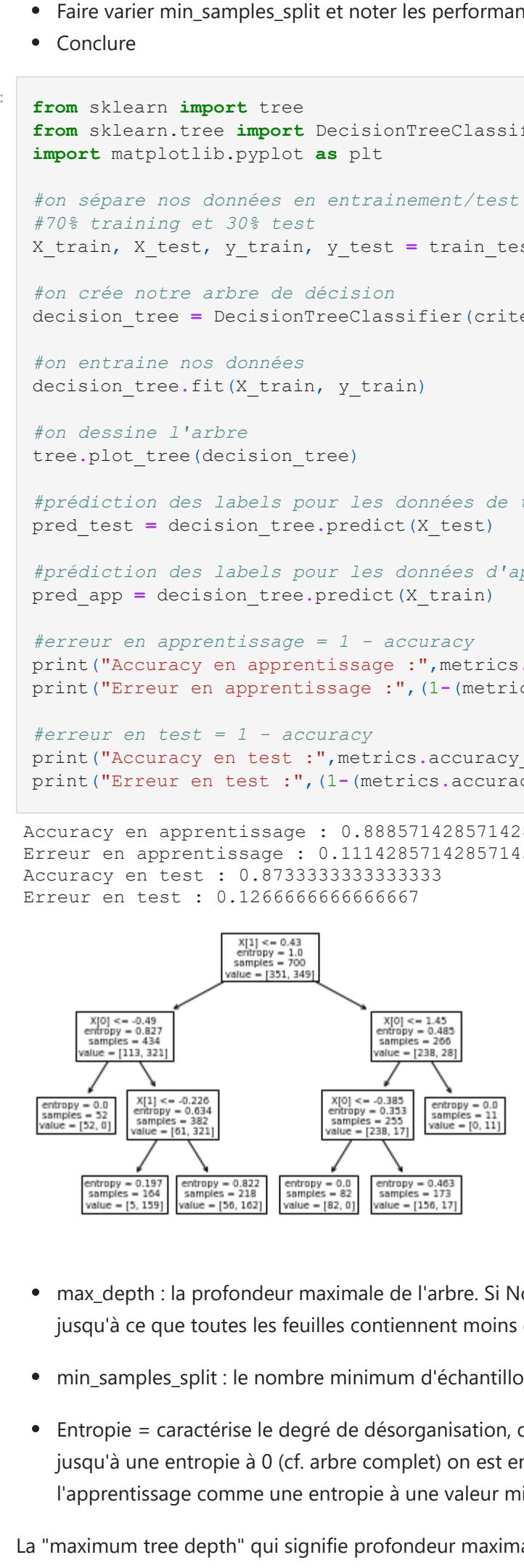
Heureusement, la personne de notre exemple s'est inscrite à chaque fois qu'un spectacle comique a eu lieu en ville, a enregistré des informations sur le comédien et a également enregistré si elle y est allée ou non. Maintenant, sur la base de cet ensemble de données, Python peut créer un arbre de décision qui peut être utilisé pour décider si de nouveaux spectacles valent la peine d'être vus.

- toutes les variables doivent être numériques
- séparer features et target
- il faut normalement rechercher un attribut discriminant et lui attribuer un seuil
- Noeud pur => une classe contient tous les samples // les autres à 0
- Si le noeud terminal est pur => STOP, sinon on passe au noeud 2 avec un second attribut discriminant et son seuil (le choix se fait selon le critère de Gini/Entropie/Gain d'information).
- Un noeud est terminal si tous les éléments associés à ce nœud appartiennent à une même classe

Pour éviter le sur-apprentissage, on contrôle la profondeur et l'entropie.

1. Définir un arbre de décision en utilisant comme critère l'entropie (criterion='entropy') et l'entrainer. Analyser l'influence des paramètres suivants sur les performances en apprentissage, en test et sur les frontières de décision.

- Faire varier max_depth et noter les performances



concerne trop peu d'exemples. Cela permet également d'empêcher le surapprentissage.

```
#on fait varier max_depth et min_samples_split
max_depth = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30]
tx_err_app = []
tx_err_test = []

#on commence par faire varier max_depth pour observer son influence
for i in max_depth:
    model = DecisionTreeClassifier(criterion='entropy', max_depth=i)
    model.fit(X_train, y_train) #on entraîne le modèle sur les données d'entraînement
    y_pred_app = model.predict(X_train) #prédiction des labels pour les données d'entraînement
    y_pred_test = model.predict(X_test) #prédiction des labels pour les données de test
    tx_err_app.append(1-(metrics.accuracy_score(y_train, y_pred_app))) #calcul du taux d'erreur en apprentissage
    tx_err_test.append(1-(metrics.accuracy_score(y_test, y_pred_test))) #calcul du taux d'erreur en test

df = pd.DataFrame(list(zip(max_depth, tx_err_app, tx_err_test)), columns = ['max_depth','Erreur en training', 'Erreur en test'])
df
```

	max_depth	Erreur en training	Erreur en test
0	1	0.201429	0.190000
1	2	0.111429	0.126667
2	3	0.111429	0.126667
3	4	0.111429	0.126667
4	5	0.104286	0.116667
5	6	0.075714	0.096667
6	7	0.060000	0.106667
7	8	0.047143	0.120000
8	9	0.040000	0.123333
9	10	0.030000	0.123333
10	20	0.000000	0.130000
11	30	0.000000	0.133333

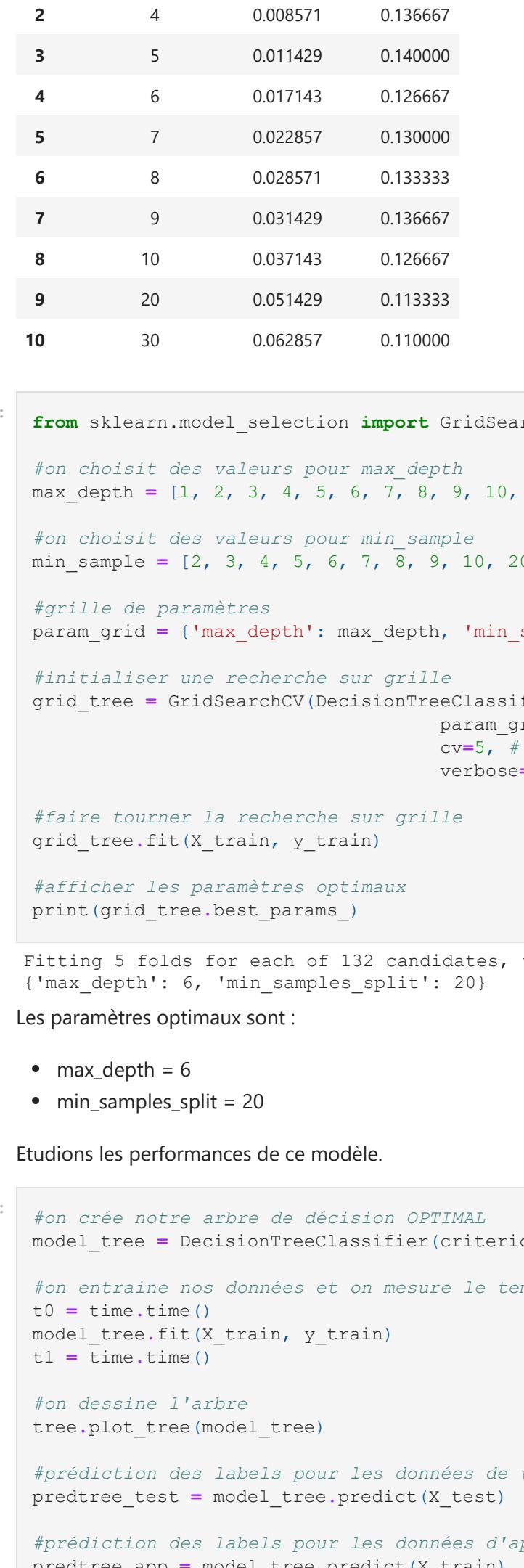
```
#on fait varier min_sample maintenant pour observer son influence

min_sample = [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30]
tx_err_app2 = []
tx_err_test2 = []

for j in min_sample:
    model = DecisionTreeClassifier(criterion='entropy', min_samples_split=j) #max_depth=None par défaut
    model.fit(X_train, y_train) #on entraîne le modèle sur les données d'entraînement
    y_pred_app2 = model.predict(X_train) #prédiction des labels pour les données d'entraînement
    y_pred_test2 = model.predict(X_test) #prédiction des labels pour les données de test
    tx_err_app2.append(1-(metrics.accuracy_score(y_train, y_pred_app2))) #calcul du taux d'erreur en apprentissage
    tx_err_test2.append(1-(metrics.accuracy_score(y_test, y_pred_test2))) #calcul du taux d'erreur en test

df = pd.DataFrame(list(zip(min_sample, tx_err_app2, tx_err_test2)), columns = ['min_sample','Erreur en training', 'Erreur en test'])
df
```

	min_sample	Erreur en training	Erreur en test
0	2	0.000000	0.133333
1	3	0.005714	0.136667



The figure consists of two parts. The top part is a decision tree diagram showing the structure of a classification model. The root node has a Gini coefficient of 0.45 and splits into two branches. Each branch further splits into four nodes, resulting in a total of 16 leaf nodes. The bottom part is a scatter plot showing the decision boundary. The x-axis ranges from 0 to 2, and the y-axis ranges from 0 to 2. A vertical line at approximately x=1.25 separates the plot into two regions: a purple region on the left and a yellow-green region on the right. Data points are scattered across the plot, with most points falling into their respective colored regions.

```

#erreur en apprentissage = 1 - accuracy
print("Accuracy en apprentissage:",metrics.accuracy_score(y_train, predtree_app))
print("Erreur en apprentissage :",(1-(metrics.accuracy_score(y_train, predtree_app)))))

#erreur en test = 1 - accuracy
print("Accuracy en test:",metrics.accuracy_score(y_test, predtree_test))
print("Erreur en test :",(1-(metrics.accuracy_score(y_test, predtree_test)))))

#on dessine les frontières de décision
disp = DecisionBoundaryDisplay.from_estimator(
    model_tree, X, response_method="predict",
    alpha=0.5)
disp.ax_.scatter(X[:, 0], X[:, 1], c=y, edgecolor="k")
plt.show()

```

Temps d'apprentissage : 2.994537353515625e-06
Accuracy en apprentissage: 0.92
Erreur en apprentissage : 0.07999999999999996
Accuracy en test: 0.9133333333333333
Erreur en test : 0.08666666666666667

A scatter plot illustrating a classification boundary. The horizontal axis (x) and vertical axis (y) both range from -2 to 3. The plot is divided into two main regions: a purple region on the left and a yellow region on the right. A vertical green line at approximately x = -0.8 serves as the decision boundary. Data points are represented by small circles: most are yellow in the yellow region and purple in the purple region, with some overlap near the boundary.

Extra Trees

1. Expliquer le principe et l'avantage des extra trees.

Les Extra-Trees divisent en choisissant des noeuds seuils entièrement au hasard. Le principal avantage des arbres supplémentaires est la réduction du biais. Il s'agit de l'échantillonnage de l'ensemble des données pendant la construction des arbres. Différents sous-ensembles de données peuvent introduire différents biais dans les résultats obtenus. Extra Trees évite cela en échantillonnant l'ensemble des données. Nous constatons que les ExtraTrees présentent un intérêt, notamment lorsque le coût de calcul est un problème. Plus précisément, lorsque l'on construit des modèles qui comportent des étapes importantes de pré-modélisation d'ingénierie et de sélection de caractéristiques, et que le coût de calcul est un problème, ExtraTrees serait un bon choix par rapport à d'autres modèles basés sur des arbres d'ensemble.

==> imaginons que l'on a un dataset de 1000 descripteurs, c'est compliqué/long de trouver le meilleur descripteur avec le meilleur seuil

2. Lancer 20 fois l'apprentissage d'un unique extra tree (ExtraTreesClassifier) en utilisant le paramètre splitter='random'. Afficher les performances en apprentissage et en test en utilisant une boîte à moustache. Commenter ces résultats.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import ExtraTreeClassifier

#on a déjà nos données brutes, d'apprentissage et de test
perf_app = []
perf_test = []
liste = [i for i in range(1,20+1)]
temps_app = []

for i in liste:
    #on crée notre classifieur ExtraTrees
    extra_trees = ExtraTreeClassifier(splitter='random')

    #on mesure le temps d'apprentissage
    #on entraîne le modèle
    t0 = time.time()
    extra_trees.fit(X_train, y_train)
    t1 = time.time()

    #temps d'apprentissage
    temps_app.append((t1-t0)/len(X))

    #prédiction des labels pour les données de test
    predXtratree_test = extra_trees.predict(X_test)
    #prédiction des labels pour les données d'apprentissage
    predXtratree_app = extra_trees.predict(X_train)

    #on note nos scoring
    perf_app.append(extra_trees.score(X_train, y_train))
    perf_test.append(extra_trees.score(X_test, y_test))

df = pd.DataFrame(list(zip(liste, perf_app, perf_test, temps_app)), columns = ['Itération','Performance en train', 'Performance en test', 'Temps d'apprentissage'])
df
```

Out[68]:

	Itération	Performance en training	Performance en test	Temps d'apprentissage
0	1	1.0	0.886667	3.984928e-06
1	2	1.0	0.856667	2.990961e-06
2	3	1.0	0.823333	1.994133e-06
3	4	1.0	0.823333	2.002478e-06
4	5	1.0	0.840000	1.990080e-06
5	6	1.0	0.863333	1.994610e-06
6	7	1.0	0.846667	1.998425e-06
7	8	1.0	0.860000	1.988649e-06
8	9	1.0	0.870000	1.991749e-06
9	10	1.0	0.830000	1.036167e-06
10	11	1.0	0.873333	9.999275e-07
11	12	1.0	0.860000	9.999275e-07
12	13	1.0	0.836667	9.756088e-07
13	14	1.0	0.866667	1.004696e-06
14	15	1.0	0.900000	9.663105e-07
15	16	1.0	0.860000	9.949207e-07
16	17	1.0	0.846667	9.973049e-07
17	18	1.0	0.860000	9.989738e-07
18	19	1.0	0.856667	9.965897e-07
19	20	1.0	0.880000	1.018286e-06

In [38]:

```
import matplotlib.pyplot as plt
import numpy as np

df2 = pd.DataFrame(list(zip(perf_app, perf_test)), columns = ['Erreur en training', 'Erreur en test'])

fig = plt.figure(figsize=(10, 7))

#ecrite d moustache
plt.boxplot(df2)
plt.show()
```



In [51]:

```
print(df2["Erreur en test"].min())
print(df2["Erreur en test"].max())
print(df2["Erreur en test"].median())
print(np.quantile(df2["Erreur en test"], 0.25)) #1er quartile
print(np.quantile(df2["Erreur en test"], 0.75)) #3ème quartile
```

```
0.8366666666666667
0.8933333333333333
0.8525
0.8808333333333334
```

4. Lancer l'apprentissage avec 10, puis 20 puis 50 extra-trees. Comment est réalisée la combinaison de leur prédition ? Rassemblez les performances en appentissage, en test dans un tableau. Comparer le modèle optimal avec celui trouvé en 2).

In [53]:

```
from sklearn.ensemble import ExtraTreesClassifier
import time
```

```
total = t1-t0
```

```
precisionTrain = []
precisionTest = []
nbr_arbre = []
tps_apprentissage = []
```

```
for i in (1, 10, 20, 50):
    cl = ExtraTreesClassifier(n_estimators=i), splitter='random')
```

```
    t0 = time.time()
    clf.fit(X_train, y_train)
    t1 = time.time()
```

```
    predictTrain = clf.predict(X_train)
    predictTest = clf.predict(X_test)
    precisionTrain = metrics.accuracy_score(predictTrain, y_train)
```

```
    precisionTest = metrics.accuracy_score(predictTest, y_test)
```

```
    precisionTrain.append(predictionTrain)
    precisionTest.append(predictionTest)
    tps_apprentissage.append((t1-t0)/len(X))
```

```
    nbr_arbre.append(i)
```

```
colonnes = ["Nombre d'arbres", "Précision Train", "Précision Test", "Temps d'apprentissage"]
```

```
df = pd.DataFrame(columns=colonnes)
```

```
df["Nombre d'arbres"] = nbr_arbre
```

```
df["Précision Train"] = precisionTrain
```

```
df["Précision Test"] = precisionTest
```

```
df["Temps d'apprentissage"] = tps_apprentissage
```

```
df
```

Out[53]:

	Nombre d'arbres	Précision Train	Précision Test	Temps d'apprentissage
0	1	1.0	0.843333	0.000003
1	10	1.0	0.900000	0.000014
2	20	1.0	0.893333	0.000024
3	50	1.0	0.893333	0.000052

On peut comparer le temps d'apprentissage et en combinant cela à la précision, on peut juger si notre arbre de décision optimal ou nos extraites sont les plus performants.

- Pour les Extra-Trees lancé en même temps : la performance en test la plus intéressante est celle de 10 Extra-Trees avec un score de 90%. Le temps d'apprentissage associé est 0.000014.
- Pour notre Extra-Trees lancé 20 fois : au bout de la 15ème fois, la performance en test la plus intéressante est à 90%. Le temps d'apprentissage associé est 9.663105e-07.

Nous pouvons donc conclure qu'il est plus intéressant d'utiliser une combinaison d'ExtraTrees plutot que de lancer plusieurs fois un même ExtraTrees (temps d'apprentissage moins long).

Pour des différences plus probantes, nous pourrions tester avec un dataset beaucoup plus large que nos 1000 données de base.

In []:

```
#sur le anaconda prompt, taper :
#jupyter-nbconvert --to PDFviaHTML TP2_SocialNetwork_MADAD_NOUAR.ipynb
```

