Vaulry Yannis
Seri Kevin

TP : Car Evaluation Data Set

GITHUB :
https://github.com/Keevin350/YannisVAULRY_KevinSERI_TPArbreDeicision.git

Objectif : Le but de ce TP est de développer une méthode pour lire et importer le dataset *car.data* pour ensuite commencer à résoudre le problème de prédiction en proposant un arbre de décision robuste.

Pour analyser les données de *car.data* nous avons décidé d'importer le dataset en utilisant la méthode de pandas « read_csv ». Nous utilisons cette fonction en précisant le séparateur = « , » et les noms des colonnes = ["buying","maint","doors","persons","lug_boot","safety","target"].
En effet, nous sommes obligés de préciser l'en-tête, car elle n'est pas disponible par défaut dans le dataset *car.data*.

Si nous traduisons les noms de colonne en Français nous obtenons ceci :
["Achat","Entretien","Portes","Personnes","Taille du coffre","Sécurité","Cible"]
Nous avons donc plusieurs features correspondant à différentes données sur différents véhicules.

La dernière colonne « Target » correspond à nos cibles. D'après la documentation issue du site : https://archive.ics.uci.edu/ml/datasets/car+evaluation, cette colonne peut prendre 4 valeurs distinctes : unacc, acc, good, vgood .

```
Entrée [44]: import pandas as pd
             import matplotlib.pyplot as plt

Entrée [45]: donnees = pd.read_csv("CAR EVALUATION DATASET/car.data",sep=",",names =
```

Afin d'analyser les données des colonnes nous pouvons utiliser les différentes fonctions de pandas telles que describe et info.

Avec ces fonctions nous pouvons voir que nous avons 7 colonnes en comptant la colonne Target. Nous avons **1728** données. Via le describe nous pouvons voir combien de **valeurs uniques** disposent les différentes colonnes (exemple pour la colonne Buying nous pouvoir que cette dernière dispose de 4 valeurs uniques). Nous pouvons aussi voir les valeurs les plus récurrentes avec leurs taux de fréquences.

```
Entrée [46]: donnees.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1728 entries, 0 to 1727
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   buying     1728 non-null   object
 1   maint      1728 non-null   object
 2   doors      1728 non-null   object
 3   persons    1728 non-null   object
 4   lug_boot   1728 non-null   object
 5   safety     1728 non-null   object
 6   target     1728 non-null   object
dtypes: object(7)
memory usage: 94.6+ KB

Entrée [47]: donnees.describe()
```

Out[47]:

	buying	maint	doors	persons	lug_boot	safety	target
count	1728	1728	1728	1728	1728	1728	1728
unique	4	4	4	3	3	3	4
top	vhigh	vhigh	2	2	small	low	unacc
freq	432	432	432	576	576	576	1210

Afin d'être sur de ne pas avoir des valeurs NaN (non numériques/absentes) nous avons utiliser la fonction dropna() qui supprime les lignes pour lesquelles des données sont manquantes. D'après ce que nous avons remarqué cette fonction n'a pas impacté notre dataset, nous restons à 1728 lignes.

Nous avons ensuite séparer notre dataset, c'est à dire que la colonne Target se trouve maintenant dans une variable nommée y et les autres données des autres colonnes se trouvent dans une variable nommée X.

```
Entrée [48]: donnees = donnees.dropna()
             y = donnees.drop(["buying", "maint", "doors", "persons", "lug_boot", "sa
             X = donnees.drop(["target"], axis=1)

Entrée [49]: X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1728 entries, 0 to 1727
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   buying    1728 non-null   object
 1   maint     1728 non-null   object
 2   doors     1728 non-null   object
 3   persons   1728 non-null   object
 4   lug_boot  1728 non-null   object
 5   safety    1728 non-null   object
dtypes: object(6)
memory usage: 94.5+ KB
```

Si nous visualisons notre variable y nous pouvons voir que nous avons uniquement les données de la colonne Target.

```
Entrée [50]: print(y.info())
             print(y)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1728 entries, 0 to 1727
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   target  1728 non-null   object
dtypes: object(1)
memory usage: 27.0+ KB
None
     target
0     unacc
1     unacc
2     unacc
3     unacc
4     unacc
...     ...
1723   good
1724  vgood
1725  unacc
1726   good
1727  vgood

[1728 rows x 1 columns]
```

Étant donné que plusieurs colonnes du dataset comportent des données catégoriques (non numériques) nous avons décider de remplacer ces données par des données numériques.

Pour la variable y nous avons décider de visualiser les différentes valeurs uniques de la colonne Target. Pour faire cela nous avons généré une fonction valeurUnique().

Cette fonction nous retourne donc les valeurs uniques de y :

```
#Verifier les différentes valeurs du dataset :

def valeurUnique(y, colonne):
    valeur = []
    for i in range(len(y)):
        if y.iloc[i,colonne] not in valeur:
            valeur.append(y.iloc[i,colonne])
    return valeur

valeurUnique(y,0)
```

```
['unacc', 'acc', 'vgood', 'good']
```

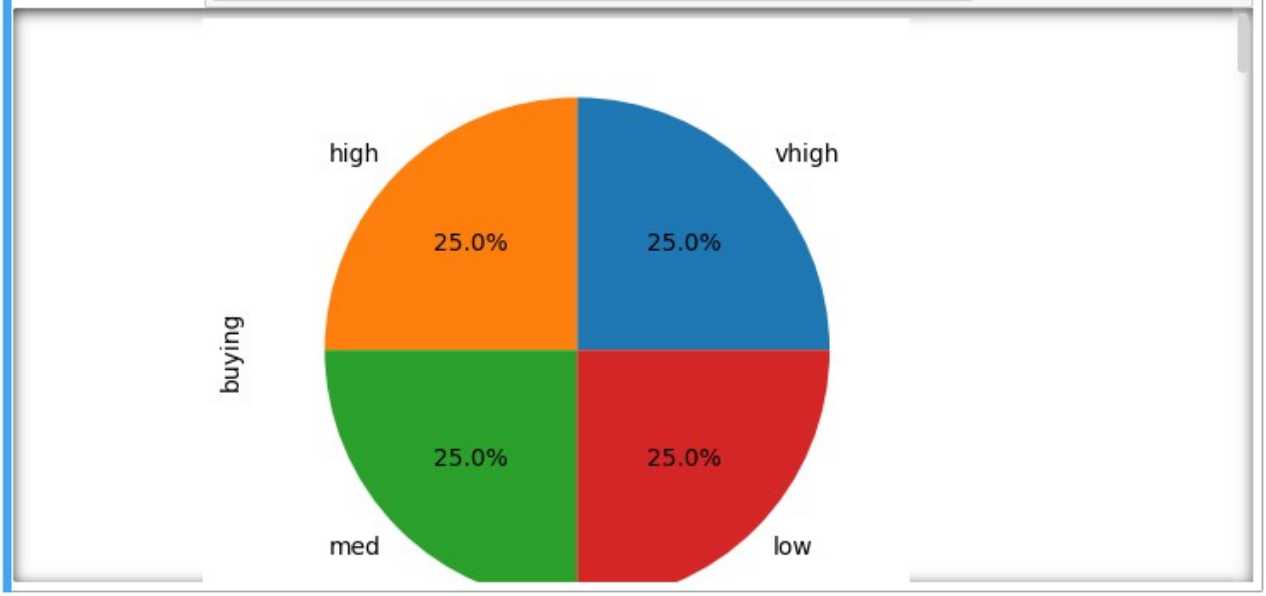
Cette fonction est utile car maintenant nous allons remplacer les valeurs trouvées par 0, 1, 3, 2.

Voici ci-dessous la boucle qui nous permet de faire cela.
Si nous relançons la fonctions nous pouvons voir que les données sont maintenant numériques.

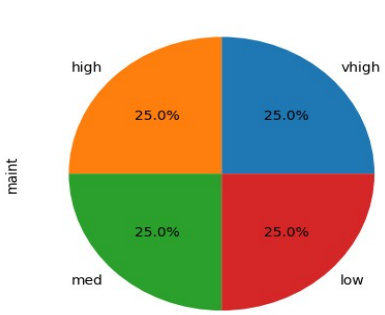
```
Entrée [52]: #Attribuer des valeurs numériques aux données du dataset (target)

             for i in range(len(y)):
                 if y.iloc[i,0] == "unacc":
                     y.iloc[i,0] = 0
                 elif y.iloc[i,0] == "acc":
                     y.iloc[i,0] = 1
                 elif y.iloc[i,0] == "good":
                     y.iloc[i,0] = 2
                 elif y.iloc[i,0] == "vgood":
                     y.iloc[i,0] = 3

             valeurUnique(y,0)

  Out[52]: [0, 1, 3, 2]
```

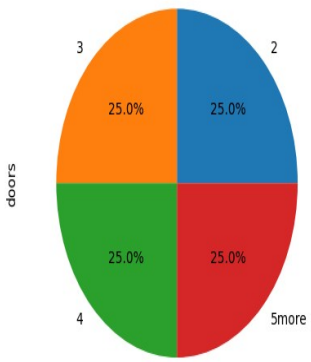
Nous décidons de faire de même pour les données de la variable X car précédemment nous avons aussi remarqué que les données de cette variable ne sont pas numériques. Afin de ne pas faire plusieurs fois manuellement la même boucle pour transformer les données non catégoriques en numériques, nous décidons de faire une fonction « attribution ». Cette dernière transforme les données en données numériques et affiche avec des graphiques de type « pie» la répartition des différentes données uniques. :

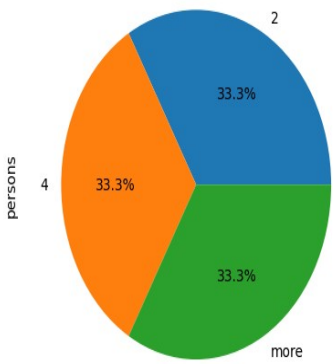
```
Entrée [53]: #Attribuer des valeurs numériques aux données du dataset X

#valeurUnique(X,0) #valeurs pour la colonne Buying


def attribution(X,nombreColonnes):
    for i in range(nombreColonnes):
        liste= valeurUnique(X, i)
        nomColonne = X.columns[i]
        for j in range(len(X)):
            if X.iloc[j,i] in liste:
                X.iloc[j,i] = liste.index(X.iloc[j,i])
        donnees[nomColonne].value_counts(normalize=True).plot(kind='pie'
        plt.show()

attribution(X,len(X.axes[1]))
```

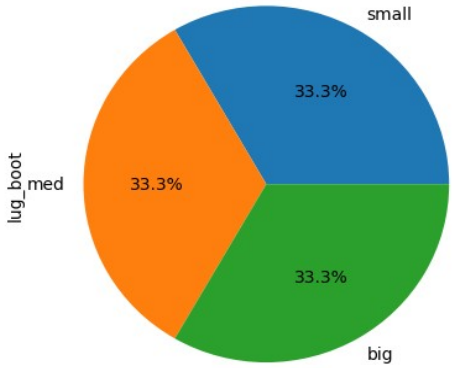


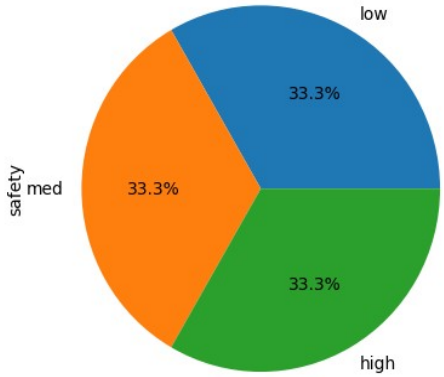
Voici ci-dessous les autres graphique :











Si nous affichons les données de la variable X nous pouvons voir que nous disposons maintenant de données numériques :

Entrée [54]:
```
X
```

Out[54]:

	buying	maint	doors	persons	lug_boot	safety
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	0	2
3	0	0	0	0	1	0
4	0	0	0	0	1	1
...	...	...	...	...	...	...
1723	3	3	3	2	1	1
1724	3	3	3	2	1	2
1725	3	3	3	2	2	0
1726	3	3	3	2	2	1
1727	3	3	3	2	2	2

1728 rows × 6 columns

Notre but maintenant est d’utiliser ce jeu de données pour prédire si les voitures sont Non acceptables, acceptables, Bonnes , Très bonnes en fonction des features que disposent la variable X.

Nous devons donc prédire si les voitures sont :

- Non Acceptable (Unacc)
- Acceptable (Acc)
- Bonnes (Good)
- Très bonnes (Vgood)

Pour faire cela nous allons utiliser comme préciser dans l'objectif un algorithme proposant un arbre de décision robuste qui nous permettra de classifier les différentes données correctement.

Nous importons les libraires nécessaires pour ce type d'algorithme. Nous importons Scikit-Learn (sklearn).

Entrée [92]:
```
from sklearn import tree
from sklearn.model_selection import train_test_split, GridSearchCV
```

Question : Combien y-a t-il d'exemples de chaque classe ?

Voici ci-dessous le nombre d'exemples pour chaque classe :

Entrée [170]:
```
#Attribuer des valeurs numériques aux données du dataset (target)

unacc = 0
acc = 0
good = 0
vgood = 0

for i in range(len(y)):
    if y.iloc[i,0] == "unacc":
        y.iloc[i,0] = 0
        unacc+=1
    elif y.iloc[i,0] == "acc":
        y.iloc[i,0] = 1
        acc+=1
    elif y.iloc[i,0] == "good":
        y.iloc[i,0] = 2
        good+=1
    elif y.iloc[i,0] == "vgood":
        y.iloc[i,0] = 3
        vgood+=1

valeurUnique(y,0)
print("Unacc :",unacc,"\nacc : ",acc,"\ngood : ",good,"\nvgood : ", vgood)
```

```
Unacc : 1210
acc :  384
good :  69
vgood :  65
```

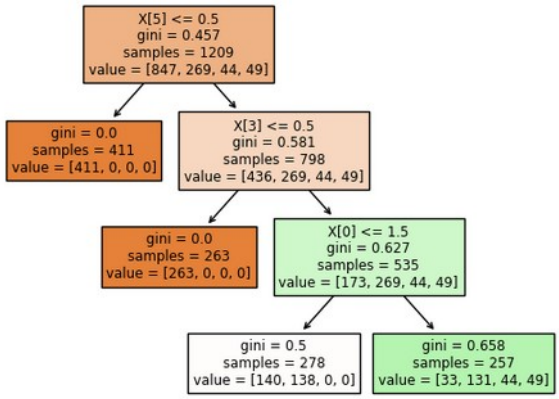
Avant de construire le modèle, séparons le jeu de données en deux : 70% pour l'apprentissage, 30% pour le test via la fonction Train_Test_Split.

```
#création de l'arbre de décision
X_train, X_test, Y_train, Y_test = train_test_split(X, y.astype('int'), train_size=0.7, random_state=0)
clf = tree.DecisionTreeClassifier(max_depth = 3, min_samples_leaf=60)
```

Nous pouvons désormais construire un arbre de décision sur ces données. Nous plaçons des paramètres aléatoires. Une fois l'apprentissage terminé, nous pouvons visualiser l'arbre, avec matplotlib en passant par la méthode plot_tree :

Entrée [108]:
```
#création de l'arbre de décision
X_train, X_test, Y_train, Y_test = train_test_split(X, y.astype('int'), train_size=0.7, random_state=0)
clf = tree.DecisionTreeClassifier(max_depth = 3, min_samples_leaf=60)
clf.fit(X_train, Y_train)
tree.plot_tree(clf, filled=True)
print(clf.score(X_test, Y_test))
```

```
0.815028901734104
```



Notre Score est actuellement celui-ci : **0.815028901734104**

Nous remarquons dans notre arbre, les indices Gini ne dépassent pas 0,65. De plus, à certains endroits on remarque que l'indice Gini est égal à 0, ce qui signifie que la répartition des classe est parfaite.

Afin d'obtenir un meilleur score nous utilisons la méthode GridSearchCV. Nous précisons plusieurs valeurs pour max_depth et min_samples_leaf afin de voir si un meilleur score peut être obtenu en précisant des paramètres plus précis pour notre jeu de données.

Précision sur les paramètres : Le paramètre max_depth est un seuil sur la profondeur maximale de l'arbre. Le paramètre min_samples_leaf donne le nombre minimal d'échantillons dans un nœud feuille. Ils permettent de mettre des contraintes sur la construction de l'arbre et donc de contrôler indirectement le phénomène de sur-apprentissage.

Attention nous voulons trouver le meilleur score possible sans tomber dans le sur-apprentissage.

Voici ci-dessous le score obtenu avec le GridSearchCV : **0,97**. Nous obtenons aussi nos paramètres optimaux que nous allons utiliser pour réaliser à nouveau un arbre de décision.

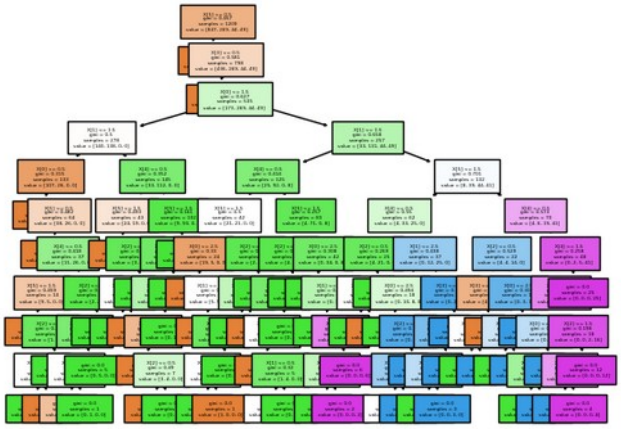
Entrée [104]:
```
#on fait un gridSearch
tuned_params = {"max_depth":[1, 2, 3, 4, 5, 6, 10, 20, 30], "min_samples_leaf":[1, 2, 3, 5, 10, 15, 20, 25, 30, 35,
grid_search = GridSearchCV(tree.DecisionTreeClassifier(), param_grid=tuned_params, cv=10)
grid_search.fit(X_train, Y_train)
print(grid_search.best_estimator_.score(X_test, Y_test))
print(grid_search.best_params_)
```

```
0.9788053949903661
{'max_depth': 10, 'min_samples_leaf': 1}
```

En utilisant les paramètres optimaux nous obtenons le score : **0.9865125240847784**. Notre nouvel arbre ressemble à ceci (*nous avons beaucoup de feuilles à cause du paramètre min_sample_leaf = 1)*:

Entrée [117]:
```
clf = tree.DecisionTreeClassifier(max_depth = 10, min_samples_leaf=1)
clf.fit(X_train, Y_train)

tree.plot_tree(clf, filled=True, fontsize=3)
print(clf.score(X_test, Y_test))
```

```
0.9865125240847784
```

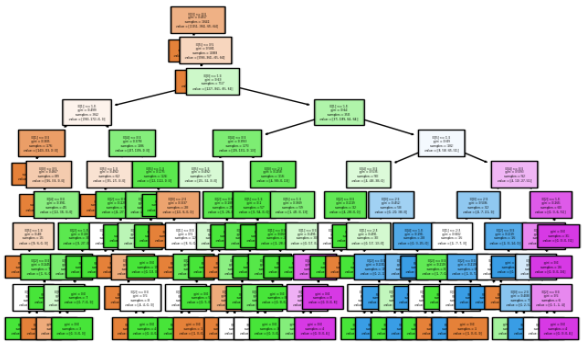


Question : Le problème ici étant particulièrement simple, refaites une division apprentissage/test avec 5% des données en apprentissage et 95% test. Calculez le taux d'éléments mal classifiés sur l'ensemble de test.
Faites varier (ou mieux, réalisez une recherche par grille avec GridSearchCV) les valeurs des paramètres max_depth et min_samples_leaf pour mesurer leur impact sur ce score.

En faisant un train_split, avec train_size = 0,95 et les paramètres optimaux trouvés précédemment, nous obtenons un score de **0,896551724137931**. Ce score est plus faible que celui obtenu précédemment.

Entrée [121]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, y.astype('int'), train_size=0.95, ran

clf = tree.DecisionTreeClassifier(max_depth = 10, min_samples_leaf=1)
clf.fit(X_train, Y_train)

tree.plot_tree(clf, filled=True, fontsize=2)
print(clf.score(X_test, Y_test))
```

```
0.896551724137931
```



SI nous faisons, à nouveau un gridSearch nous obtenons les paramètres optimaux ci-dessous et un score de : **0.9195402298850575**.
Le score obtenu est donc plus faible que le score du gridsearch avec un train_size à **0,7**. Les paramètres optimaux sont les mêmes que pour le précédent gridsearchCV.

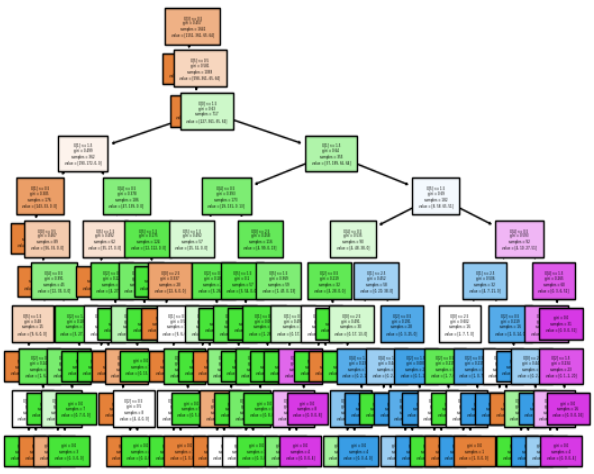
Entrée [122]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, y.astype('int'), train_size=0.95, rar

#on fait un gridSearch
tuned_params = {"max_depth":[1, 2, 3, 4, 5, 6, 10, 20, 30], "min_samples_leaf":[1, 2, 3, 5,
grid_search = GridSearchCV(tree.DecisionTreeClassifier(), param_grid=tuned_params, cv=10)
grid_search.fit(X_train, Y_train)
print(grid_search.best_estimator_.score(X_test, Y_test))
print(grid_search.best_params_)
```

```
0.9195402298850575
{'max_depth': 10, 'min_samples_leaf': 1}
```

En effectuant à nouveau un apprentissage avec les paramètres optimaux. Nous obtenons, un score de **0.9770114942528736** avec cet arbre de décision :

Entrée [123]:
```
clf = tree.DecisionTreeClassifier(max_depth = 10, min_samples_leaf=1)
clf.fit(X_train, Y_train)

tree.plot_tree(clf, filled=True, fontsize=2)
print(clf.score(X_test, Y_test))
```

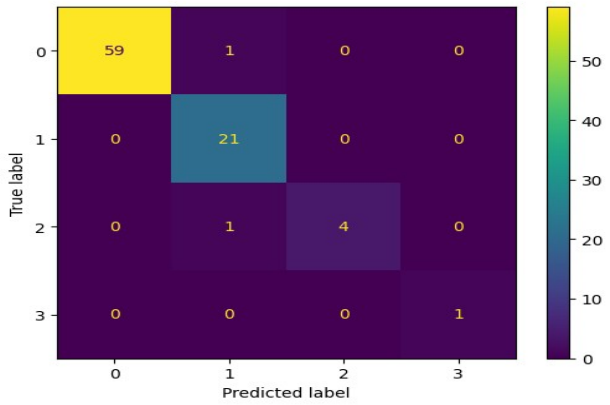
```
0.9770114942528736
```



Nous avons construit une matrice de confusion afin de visualiser les éléments mal classifiés. Cette matrice a donc été réalisée avec X_test. Nous obtenons la matrice de confusion ci-dessous.

```
: from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay


  resultat = clf.predict(X_test)
  clf.score(X_test, Y_test)
  #score en test de l'arbre de décision 0.9777777777777777
  print(clf.score(X_test, Y_test))
  cm =confusion_matrix(resultat, Y_test)
  print(confusion_matrix(resultat, Y_test))


  disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
  disp.plot()
  plt.show()
```

```
0.9770114942528736
[[59  1  0  0]
 [ 0 21  0  0]
 [ 0  1  4  0]
 [ 0  0  0  1]]
```



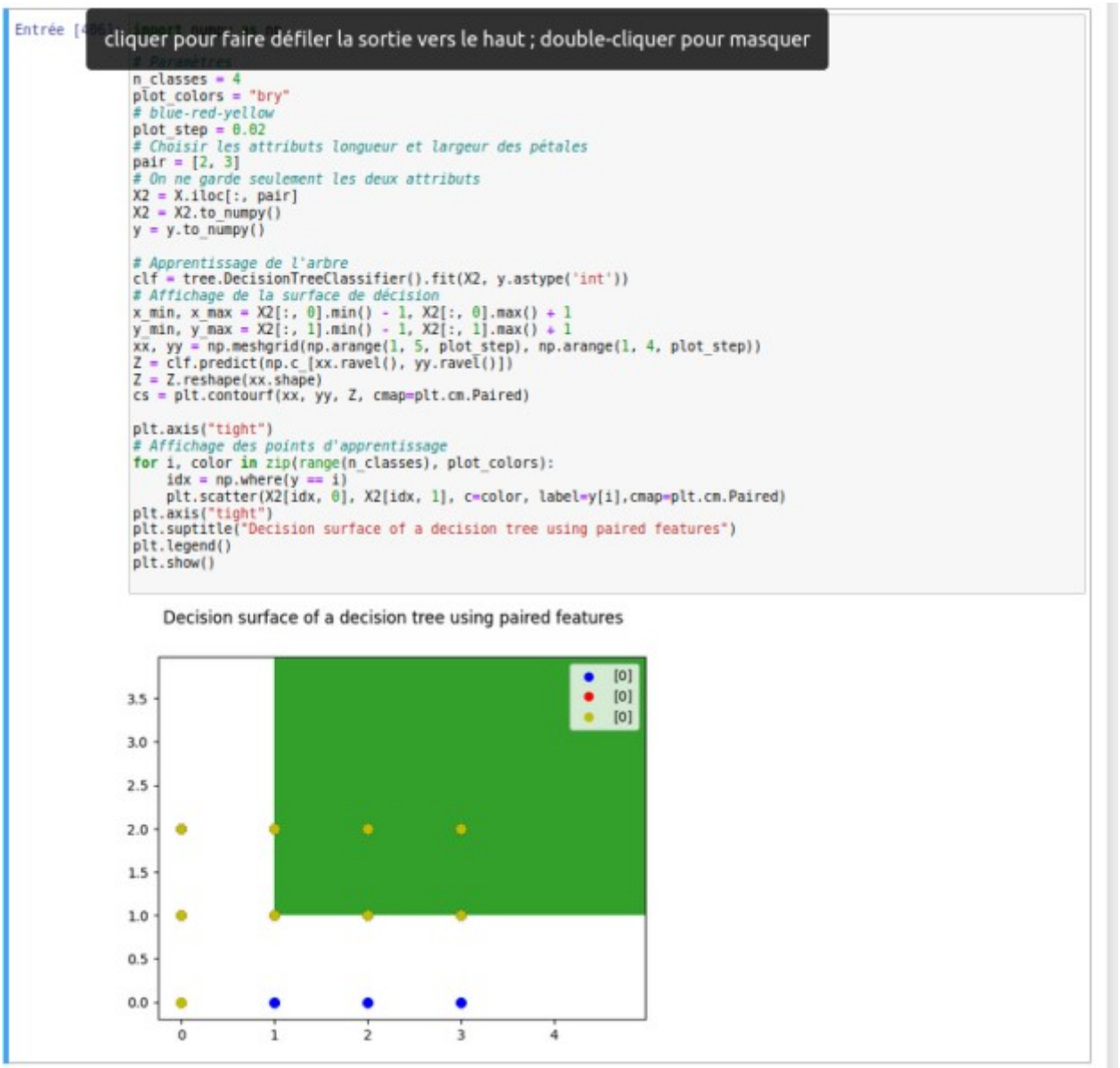
Nous savons que nous avons un test_size de 0,95 cela veut dire que nous prenons 1641 enregistrements pour effectuer l'apprentissage il nous reste donc **1728-1641 = 87**.

D'après cette matrice nous pouvons voir que l'arbre de décision a prédit correctement **59** valeurs pour la classe 0 c'est à dire la classe « Non acceptable », il y' a 1 enregistrement que nous avons mal prédit (nous avons prédit « acceptable » au lieu de « non acceptable »).

L'arbre a prédit **23** données acceptables, il y' a 2 enregistrements en plus qui ont mal été prédit.

Pour la classe Bien (good) l'arbre a prédit **4** enregistrements, il nous en manque 1 et pour la classe très bien l'arbre a prédit correctement la **seule** donnée de cette classe.

Nous avons affiché la surface de décision afin de trouver la paire de séparation entre les classes la plus marquée. Le problème est que nous avons ds points superposés.



Nous pensons que même si nous affichons la surface de décision, nous ne pourrons trouvé la paire de séparation entre les classes la plus marquée car il n' y a pas assez de données à répartir sur les plans, c'est à dire que nous avons trop peu de valeurs uniques. **Les points sont donc superposés.** Lorsque nous avions réalisé ce code sur le dataset Iris, nous avions plus de valeurs unique. Nous pouvions donc visuellement interprété les différentes surface de décision. Les points n'étaient pas superposés.