Notebook - UE Apprentissage supervisé

Quentin Fouché

13 novembre 2022

1 Projet 1

1.1 Exercice 1

[1]:
```
# (1.1) Import des packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import plot_tree
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error as MSE
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
import sys
!{sys.executable} -m pip install palmerpenguins
from palmerpenguins import load_penguins
```

[2]:
```
# (1.2) Import du jeu de données sur les pingouins
penguins_orig = load_penguins()
penguins_orig.head()
```

[2]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	male	2007
1	Adelie	Torgersen	39.5	17.4	186.0			
2	Adelie	Torgersen	40.3	18.0	195.0			
3	Adelie	Torgersen	NaN	NaN	NaN			
4	Adelie	Torgersen	36.7	19.3	193.0			

```
1       3800.0  female  2007
2       3250.0  female  2007
3          NaN     NaN  2007
4       3450.0  female  2007
```

[3]:
```
# (1.3) Inspection des variables
print(penguins_orig.info())
print()
print("Nom des espèces :")
print(penguins_orig["species"].unique())
print()
print("Nom des îles :")
print(penguins_orig["island"].unique())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 344 entries, 0 to 343
Data columns (total 8 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   species            344 non-null    object
 1   island             344 non-null    object
 2   bill_length_mm     342 non-null    float64
 3   bill_depth_mm      342 non-null    float64
 4   flipper_length_mm  342 non-null    float64
 5   body_mass_g        342 non-null    float64
 6   sex                333 non-null    object
 7   year               344 non-null    int64
dtypes: float64(4), int64(1), object(3)
memory usage: 21.6+ KB
None

Nom des espèces :
['Adelie' 'Gentoo' 'Chinstrap']

Nom des îles :
['Torgersen' 'Biscoe' 'Dream']
```

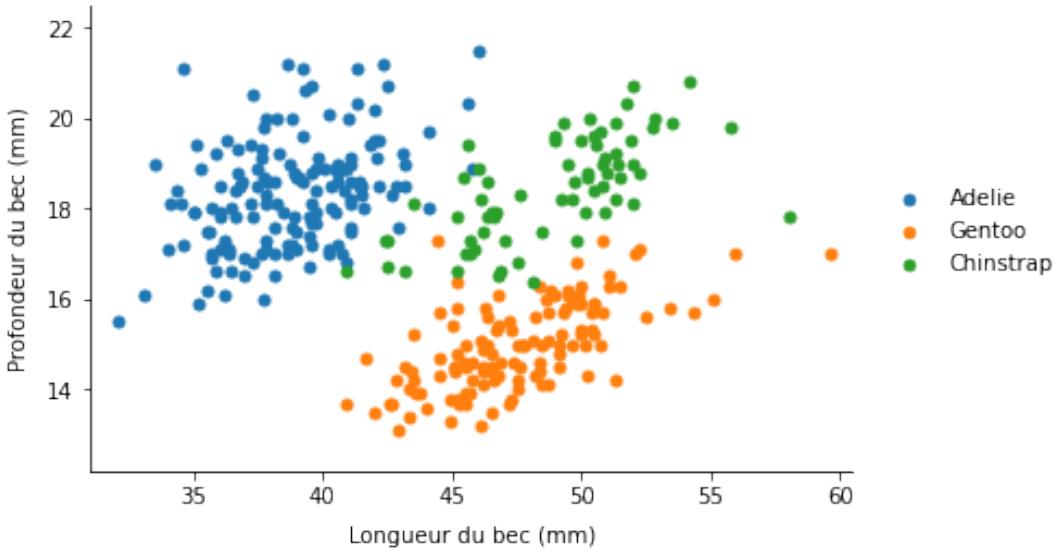
[4]:
```
# (1.4) Suppression des valeurs manquantes dans la longueur et la profondeur␣
↪du bec
penguins = penguins_orig.dropna(subset = ["bill_length_mm", "bill_depth_mm"])
```

[5]:
```
# (1.5) Création et entraînement d'un arbre de classification ("dt") ayant␣
↪une profondeur de 1
X = penguins[["bill_length_mm", "bill_depth_mm"]].to_numpy()
y = penguins["species"].to_numpy()
dt = DecisionTreeClassifier(max_depth = 1, random_state = 1)
dt.fit(X, y)
```

[5]:
```
DecisionTreeClassifier(max_depth=1, random_state=1)
```

1.2 Exercice 2

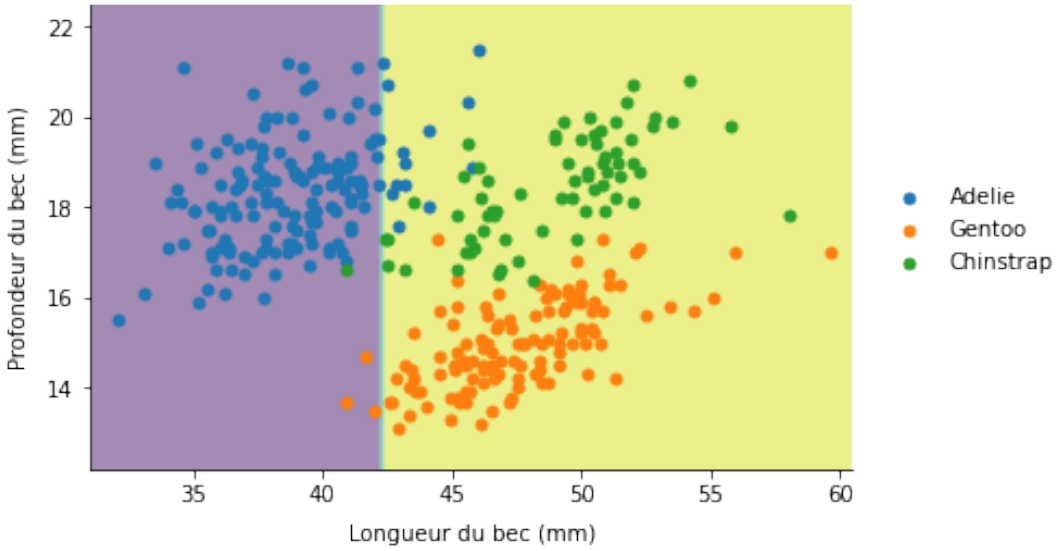
[6]:
```
# (2.1) Représentation de la profondeur du bec en fonction de la longueur du␣
↪bec, en indiquant l'espèce en couleur
species = list(penguins["species"].unique())
colors = ["C0", "C1", "C2"]
fig, ax = plt.subplots(figsize = (16/2.54, 10/2.54))
for target, color in zip(species, colors):
    indicesToKeep = penguins["species"] == target
    ax.scatter(penguins.loc[indicesToKeep, "bill_length_mm"], penguins.
↪loc[indicesToKeep, "bill_depth_mm"], c = color, s = 25)
ax.set_xlabel("Longueur du bec (mm)", labelpad = 8)
ax.set_ylabel("Profondeur du bec (mm)", labelpad = 8)
ax.set_xlim(31,60.5)
ax.set_ylim(12.2,22.5)
ax.legend(species, bbox_to_anchor = (1.02, 0.65), frameon = False)
ax.spines[["top", "right"]].set_visible(False)
plt.show()
```



1.3 Exercice 3

[7]:
```
# (3.1) Même représentation en affichant la partition faite par l'arbre (une␣
↪couleur par groupe créé)
legend = {}
fig, ax = plt.subplots(figsize = (16/2.54, 10/2.54))
disp = DecisionBoundaryDisplay.from_estimator(dt, X, response_method =␣
↪"predict", alpha = 0.5, ax = ax)
for target, color in zip(species, colors):
    indicesToKeep = penguins["species"] == target
    disp.ax_.scatter(penguins.loc[indicesToKeep, "bill_length_mm"], penguins.
↪loc[indicesToKeep, "bill_depth_mm"], c = color, s = 25)
```

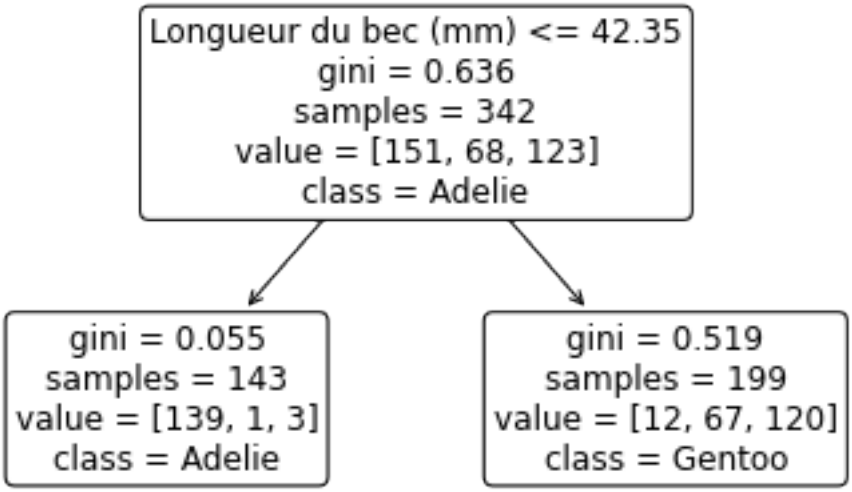
```
    legend[str(target)] = ax.scatter(x = 100, y = 100, c = color, s = 25,␣
 ↪label = target)
ax.set_xlabel("Longueur du bec (mm)", labelpad = 8)
ax.set_ylabel("Profondeur du bec (mm)", labelpad = 8)
ax.set_xlim(31,60.5)
ax.set_ylim(12.2,22.5)
ax.legend(handles = [legend["Adelie"], legend["Gentoo"],␣
 ↪legend["Chinstrap"]], bbox_to_anchor = (1.02, 0.65), frameon = False)
ax.spines[["top", "right"]].set_visible(False)
plt.show()
```



La variable utilisée par l'arbre de classification pour réaliser la partition est la longueur du bec : les individus sont classés dans une espèce ou une autre selon que leur longueur de bec est en-dessous ou au-dessus de 42mm.

1.4 Exercice 4

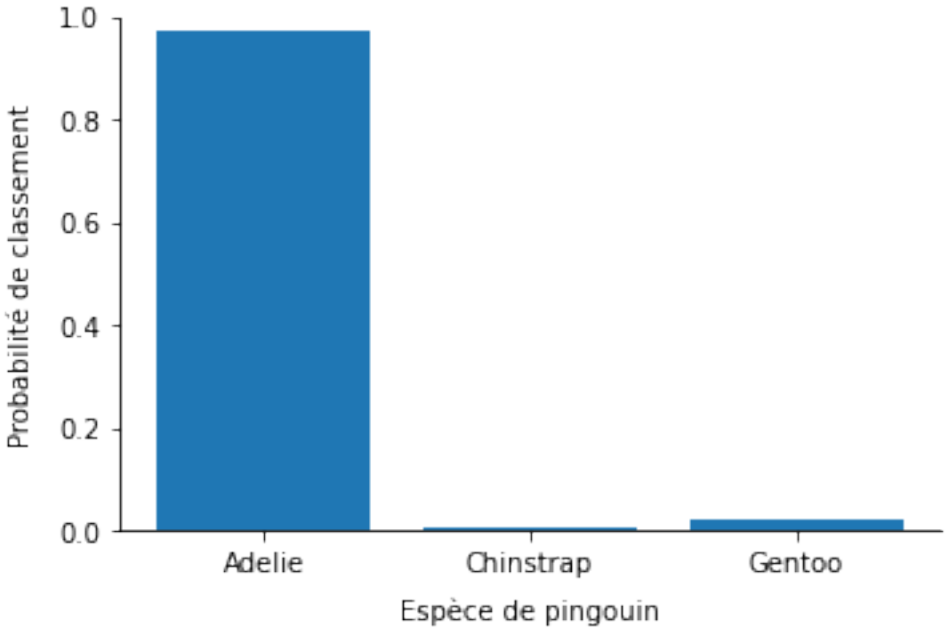
[8]:
```
# (4.1) Représentation de l'arbre de classification
plt.figure(figsize = (17/2.54, 10/2.54))
plot_tree(dt, feature_names = ["Longueur du bec (mm)", "Profondeur du bec␣
 ↪(mm)"], class_names = ["Adelie", "Chinstrap", "Gentoo"], fontsize = 12,␣
 ↪rounded = True)
plt.show()
```



Le label retenu pour la feuille de gauche est “Adelie” et celui pour la feuille de droite est “Gentoo”. Ces labels correspondent au nom de l’espèce dont le nombre d’individus est le plus élevé dans chaque feuille. Il est cohérent pour la feuille de gauche, dans la mesure où 91% des individus de cette feuille appartiennent bien à l’espèce “Adelie”. Le label de la feuille de droite est moins cohérent car, si la majorité des individus de l’espèce “Gentoo” se trouvent effectivement dans cette feuille, la majorité des individus de l’espèce “Chinstrap” s’y trouvent aussi mais sans figurer dans le label. Si ce modèle était utilisé pour des prédictions, les individus des espèces “Adelie” et “Gentoo” seraient correctement classés mais ceux de l’espèce “Chinstrap” seraient ignorés.

1.5 Exercice 5

[9]:
```
# (5.1) Représentation des probabilités de classer dans chaque espèce un␣
↪pingouin ayant un bec de 35mm de long et 17mm de profondeur
proba = dt.predict_proba(np.array([[35, 17]]))[0]
fig, ax = plt.subplots(figsize = (14/2.54, 9/2.54))
ax.bar(x = ["Adelie", "Chinstrap", "Gentoo"], height = proba)
ax.set_xlabel("Espèce de pingouin", labelpad = 8)
ax.set_ylabel("Probabilité de classement", labelpad = 8)
ax.set_ylim(0,1)
ax.spines[["top", "right"]].set_visible(False)
plt.show()
```



D'après ce graphique, un individu ayant un bec de 35mm de long et 17mm de profondeur aura 97% de chances d'appartenir à l'espèce "Adelie", selon l'arbre de classification. Cet arbre semble donc approprié pour classer des individus ayant cette taille de bec.

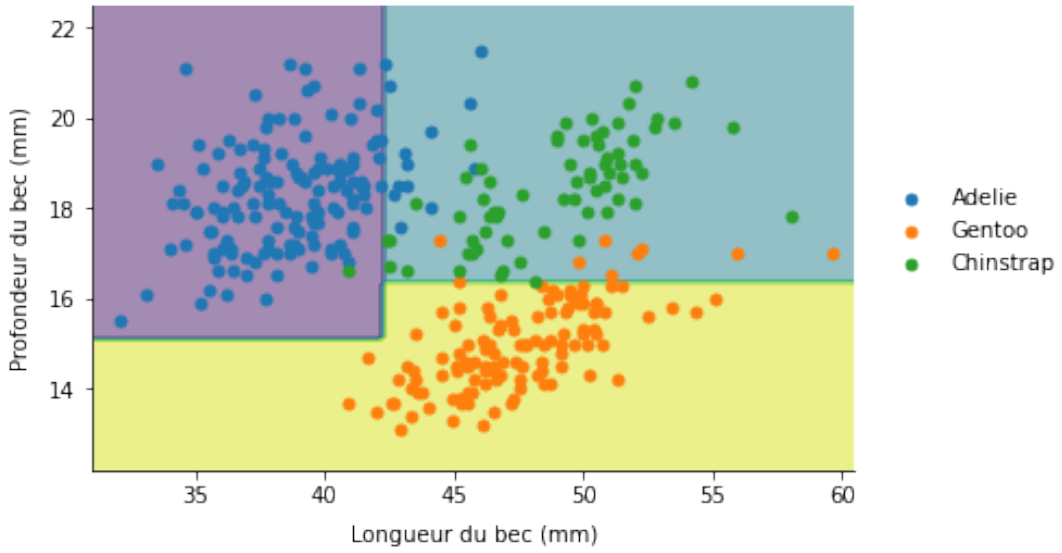
1.6 Exercice 6

[10]:
```
# (6.1) Entraînement d'un arbre de classification ("dt") ayant une profondeur␣
↪de 2
dt_2 = DecisionTreeClassifier(max_depth = 2, random_state = 1)
dt_2.fit(X, y)
```

[10]:
```
DecisionTreeClassifier(max_depth=2, random_state=1)
```

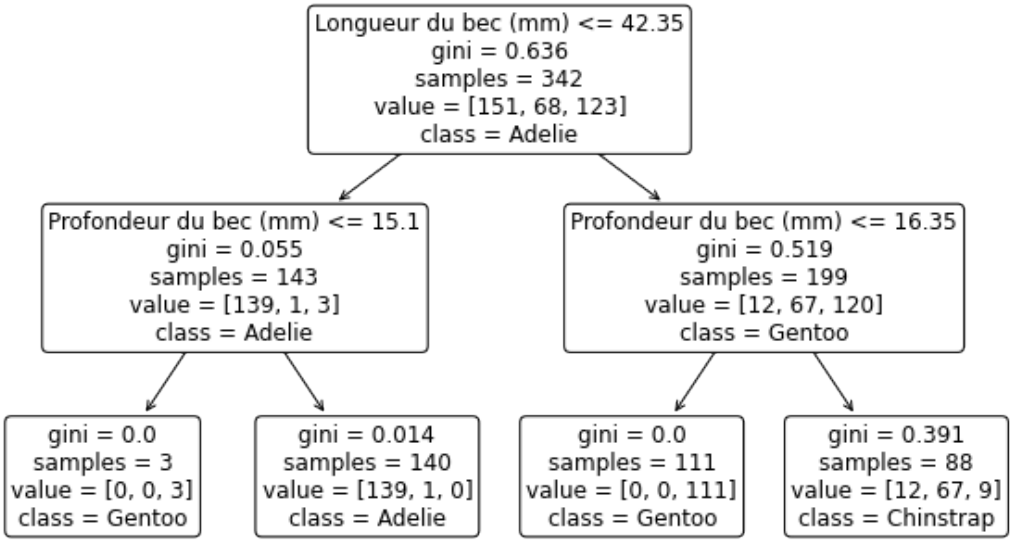
[11]:
```
# (6.2) Représentation de la profondeur du bec en fonction de la longueur du␣
↪bec, en indiquant en couleur d'une part l'espèce, d'autre part la partition␣
↪faite par l'arbre (une couleur par groupe créé)
legend = {}
fig, ax = plt.subplots(figsize = (16/2.54, 10/2.54))
disp_2 = DecisionBoundaryDisplay.from_estimator(dt_2, X, response_method =␣
↪"predict", alpha = 0.5, ax = ax)
for target, color in zip(species, colors):
    indicesToKeep = penguins["species"] == target
    disp_2.ax_.scatter(penguins.loc[indicesToKeep, "bill_length_mm"],␣
↪penguins.loc[indicesToKeep, "bill_depth_mm"], c = color, s = 25)
    legend[str(target)] = ax.scatter(x = 100, y = 100, c = color, s = 25,␣
↪label = target)
ax.set_xlabel("Longueur du bec (mm)", labelpad = 8)
ax.set_ylabel("Profondeur du bec (mm)", labelpad = 8)
ax.set_xlim(31,60.5)
ax.set_ylim(12.2,22.5)
```

```
ax.legend(handles = [legend["Adelie"], legend["Gentoo"],␣
 ↪legend["Chinstrap"]], bbox_to_anchor = (1.02, 0.65), frameon = False)
ax.spines[["top", "right"]].set_visible(False)
plt.show()
```



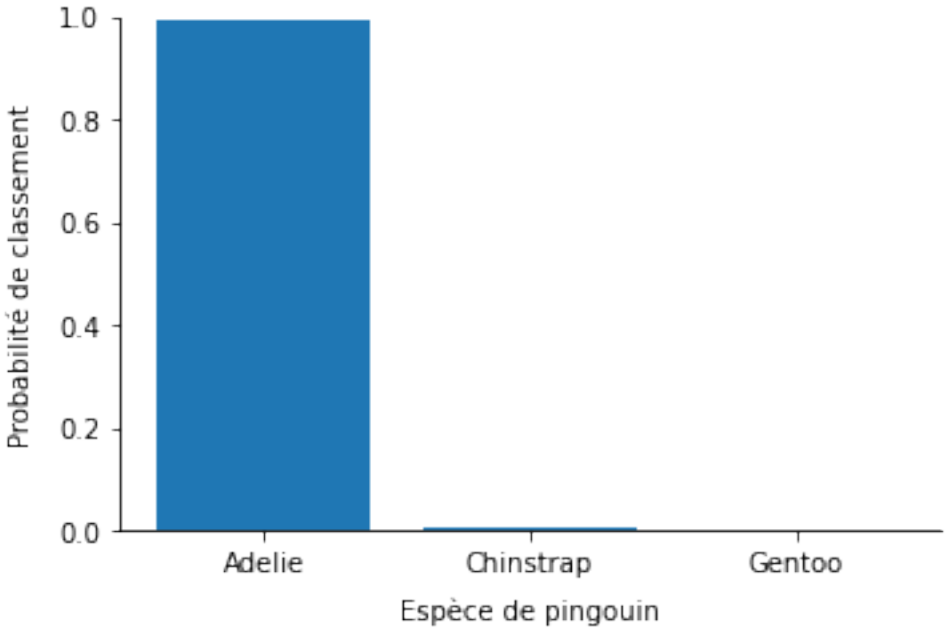
Les variables utilisées par l'arbre de classification pour réaliser la partition sont en premier la longueur du bec (seuil autour de 42mm) et en second la profondeur de bec (seuil autour de 15.1 ou 16.4mm selon que la longueur de bec est respectivement inférieure ou supérieure à 42mm).

[12]:
```
# (6.3) Représentation de l'arbre de classification
plt.figure(figsize = (26/2.54, 15/2.54))
plot_tree(dt_2, feature_names = ["Longueur du bec (mm)", "Profondeur du bec␣
 ↪(mm)"], class_names = ["Adelie", "Chinstrap", "Gentoo"], fontsize = 12,␣
 ↪rounded = True)
plt.show()
```



Les labels retenus pour chaque feuille sont, de gauche à droite, “Gentoo”, “Adelie”, “Gentoo” et “Chinstrap”. Ces labels apparaissent plus cohérent que ceux de l’abre de classification précédent, puisque cette fois-ci aucune espèce n’est ignorée et chaque groupe contient bien une seule espèce largement majoritaire par rapport aux deux autres.

[13]:
```
# (6.4) Représentation des probabilités de classer dans chaque espèce un␣
↪pingouin ayant un bec de 35mm de long et 17mm de profondeur
proba = dt_2.predict_proba(np.array([[35, 17]]))[0]
fig, ax = plt.subplots(figsize = (14/2.54, 9/2.54))
ax.bar(x = ["Adelie", "Chinstrap", "Gentoo"], height = proba)
ax.set_xlabel("Espèce de pingouin", labelpad = 8)
ax.set_ylabel("Probabilité de classement", labelpad = 8)
ax.set_ylim(0,1)
ax.spines[["top", "right"]].set_visible(False)
plt.show()
```



D'après ce graphique, un individu ayant un bec de 35mm de long et 17mm de profondeur aura plus de 99% de chances d'appartenir à l'espèce "Adelie", un pourcentage plus élevé que celui obtenu avec un arbre de profondeur 1.

2 Projet 2

2.1 Exercice 1 : Entraîner un arbre de classification

[14]:
```
# (1.1) Import du jeu de données "breast cancer.csv"
breast_cancer_orig = pd.read_csv(r"C:\Users\quent\Documents\DU Data Analyst␣
↪2022\UE n°6 - Introduction au Machine Learning\Jeux de données\breast␣
↪cancer.csv", low_memory = False, encoding = "latin-1")
breast_cancer = breast_cancer_orig.copy(deep = True)
breast_cancer.head()
```

[14]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	\
0	842302	M	17.99	10.38	122.80	1001.0	
1	842517	M	20.57	17.77	132.90	1326.0	
2	84300903	M	19.69	21.25	130.00	1203.0	
3	84348301	M	11.42	20.38	77.58	386.1	
4	84358402	M	20.29	14.34	135.10	1297.0	

	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	\
0	0.11840	0.27760	0.3001	0.14710	
1	0.08474	0.07864	0.0869	0.07017	
2	0.10960	0.15990	0.1974	0.12790	
3	0.14250	0.28390	0.2414	0.10520	
4	0.10030	0.13280	0.1980	0.10430	

	...	texture_worst	perimeter_worst	area_worst	smoothness_worst	\
0	...	17.33	184.60	2019.0	0.1622	
1	...	23.41	158.80	1956.0	0.1238	
2	...	25.53	152.50	1709.0	0.1444	
3	...	26.50	98.87	567.7	0.2098	
4	...	16.67	152.20	1575.0	0.1374	

	compactness_worst	concavity_worst	concave points_worst	symmetry_worst	\
0	0.6656	0.7119	0.2654	0.4601	
1	0.1866	0.2416	0.1860	0.2750	
2	0.4245	0.4504	0.2430	0.3613	
3	0.8663	0.6869	0.2575	0.6638	
4	0.2050	0.4000	0.1625	0.2364	

	fractal_dimension_worst	Unnamed: 32
0	0.11890	NaN
1	0.08902	NaN
2	0.08758	NaN
3	0.17300	NaN
4	0.07678	NaN

[5 rows x 33 columns]

[15]:
```
# (1.2) Inspection des variables
print(breast_cancer.info())
print()
```

```
print("Catégories de diagnostics :")
print(breast_cancer["diagnosis"].unique())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 33 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   id                       569 non-null    int64
 1   diagnosis                569 non-null    object
 2   radius_mean              569 non-null    float64
 3   texture_mean             569 non-null    float64
 4   perimeter_mean           569 non-null    float64
 5   area_mean                569 non-null    float64
 6   smoothness_mean          569 non-null    float64
 7   compactness_mean         569 non-null    float64
 8   concavity_mean           569 non-null    float64
 9   concave points_mean      569 non-null    float64
 10  symmetry_mean            569 non-null    float64
 11  fractal_dimension_mean   569 non-null    float64
 12  radius_se                569 non-null    float64
 13  texture_se               569 non-null    float64
 14  perimeter_se             569 non-null    float64
 15  area_se                  569 non-null    float64
 16  smoothness_se            569 non-null    float64
 17  compactness_se           569 non-null    float64
 18  concavity_se             569 non-null    float64
 19  concave points_se        569 non-null    float64
 20  symmetry_se              569 non-null    float64
 21  fractal_dimension_se     569 non-null    float64
 22  radius_worst             569 non-null    float64
 23  texture_worst            569 non-null    float64
 24  perimeter_worst          569 non-null    float64
 25  area_worst               569 non-null    float64
 26  smoothness_worst         569 non-null    float64
 27  compactness_worst        569 non-null    float64
 28  concavity_worst          569 non-null    float64
 29  concave points_worst     569 non-null    float64
 30  symmetry_worst           569 non-null    float64
 31  fractal_dimension_worst  569 non-null    float64
 32  Unnamed: 32              0 non-null      float64
dtypes: float64(31), int64(1), object(1)
memory usage: 146.8+ KB
None

Catégories de diagnostics :
['M' 'B']
```

[16]:
```
# (1.3) Création de jeux de données d'entraînement et de test ; pour les labels, les valeurs M et B sont remplacées par 1 et 0, respectivement
X = breast_cancer[["radius_mean", "concave points_mean"]].to_numpy()
```

```
y = []
for i in breast_cancer["diagnosis"]:
    if i == "M":
        y.append(1)
    else:
        y.append(0)
y = np.array(y)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪stratify = y, random_state = 1)
```

[17]:
```
# (1.4) Import de la fonction DecisionTreeClassifier à partir du package␣
↪sklearn.tree
from sklearn.tree import DecisionTreeClassifier
```

[18]:
```
# (1.5) Création d'un arbre de classification nommé dt, avec une profondeur␣
↪maximale de 6 et une "seed" égale à 1
dt = DecisionTreeClassifier(max_depth = 6, random_state = 1)
```

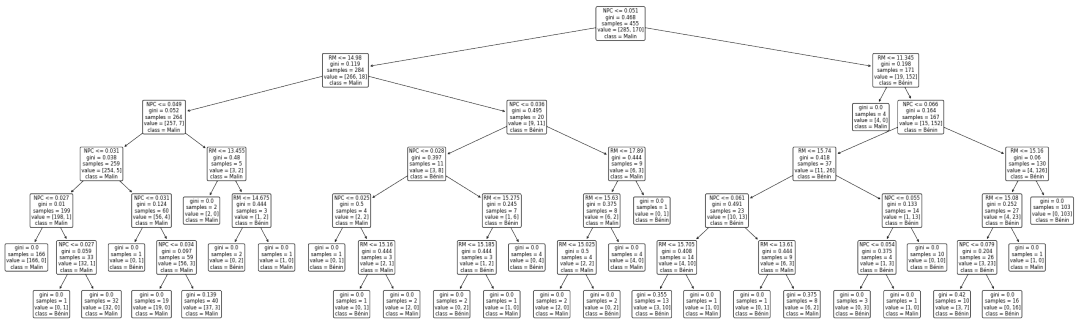
[19]:
```
# (1.6) Entraînement de l'arbre sur le jeu de données d'entraînement
dt.fit(X_train, y_train)
```

[19]:
```
DecisionTreeClassifier(max_depth=6, random_state=1)
```

[20]:
```
# (1.7) Utilisation de l'arbre pour prédire le caractère bénin ou malin des␣
↪tumeurs sur la totalité du jeu de données (affichage uniquement des 5␣
↪premières valeurs)
dt.predict(X)[0:5]
```

[20]:
```
array([1, 1, 1, 1, 1])
```

[21]:
```
# (1.8) Représentation de l'arbre de classification (les variables sont␣
↪désignées par leurs initiales : "RM" pour le rayon moyen et "NPC" pour le␣
↪nombre de points concaves)
plt.figure(figsize = (80/2.54, 25/2.54))
plot_tree(dt, feature_names = ["RM", "NPC"], class_names = ["Malin",␣
↪"Bénin"], fontsize = 8, rounded = True)
plt.show()
```



2.2 Exercice 2 : Evaluation d'un arbre, choix du critère d'information

[22]:
```
# (2.1) Import de la fonction accuracy_score à partir du package sklearn.
↪metrics
from sklearn.metrics import accuracy_score
```

[23]:
```
# (2.2) Utilisation de l'arbre pour prédire le caractère bénin ou malin des␣
↪tumeurs du jeu de données test
y_pred = dt.predict(X_test)
```

[24]:
```
# (2.3) Affichage de la valeur de la métrique de performance
print("Accuracy = {:.2f}".format(accuracy_score(y_test, y_pred)))
```

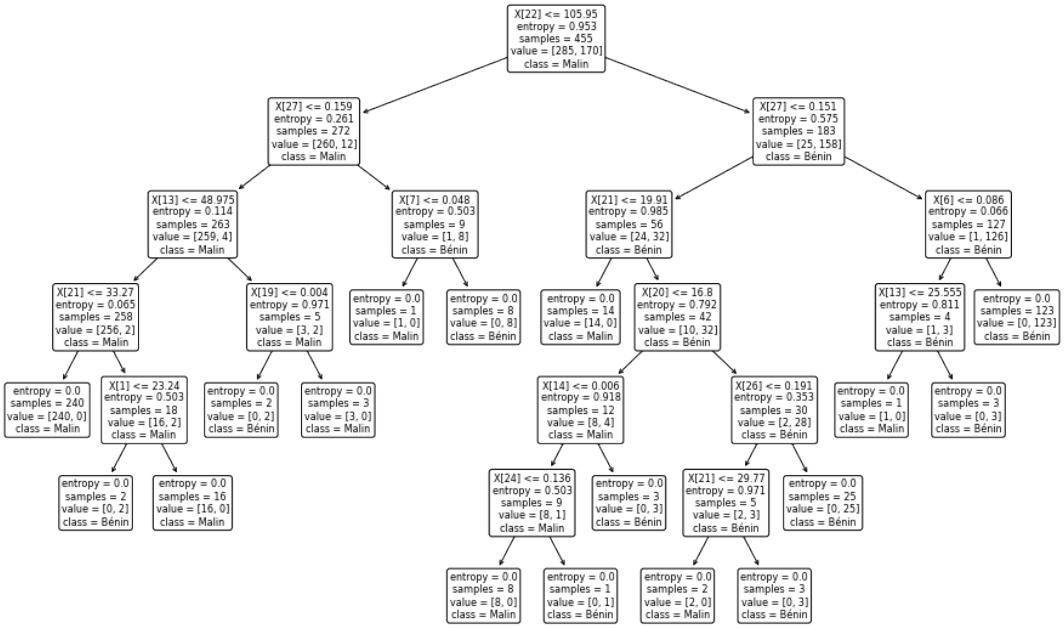
```
Accuracy = 0.89
```

[25]:
```
# (2.4) Création de 2 nouveaux arbres en prenant en compte l'ensemble des␣
↪variables du jeu de données (avec toujours une profondeur de 6)
# (2.4.1) Le premier arbre utilise l'entropie comme critère d'information
X_tot = breast_cancer.iloc[:,2:32].to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X_tot, y, test_size = 0.
↪2, stratify = y, random_state = 1)
dt_entropy = DecisionTreeClassifier(criterion = "entropy", max_depth = 6,␣
↪random_state = 1)
dt_entropy.fit(X_train, y_train)
# (2.4.2) Le second arbre mobilise l'indice de Gini
dt_gini = DecisionTreeClassifier(criterion = "gini", max_depth = 6,␣
↪random_state = 1)
dt_gini.fit(X_train, y_train)
```

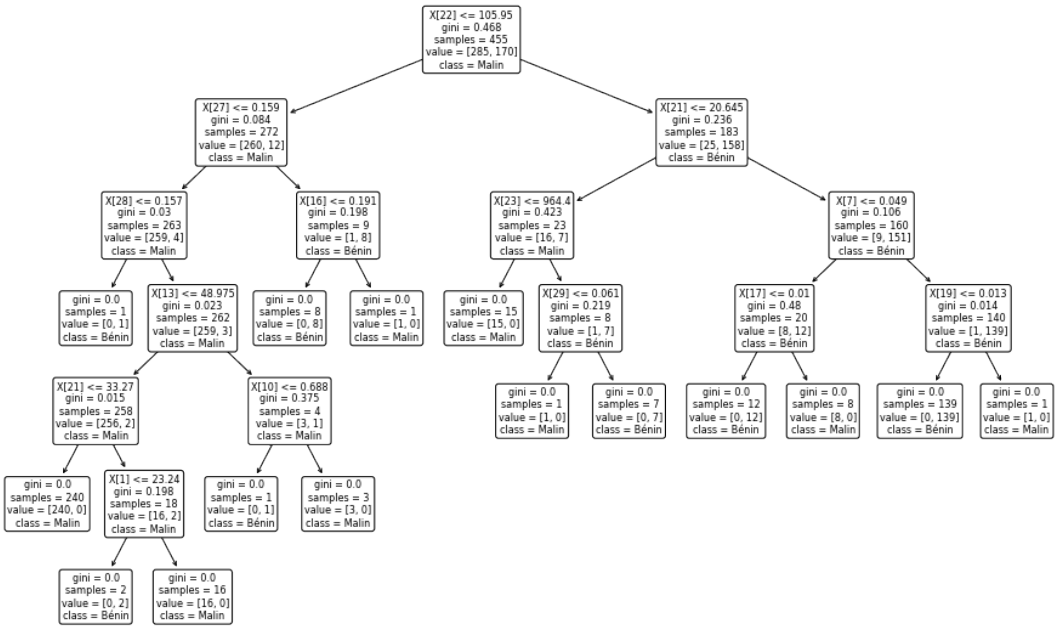
[25]:
```
DecisionTreeClassifier(max_depth=6, random_state=1)
```

L'entropie et l'indice de Gini sont deux critères d'information utilisés dans la création d'arbres de décision. Ces critères sont mobilisés à chaque noeud de l'arbre pour choisir d'une part quelle variable utiliser pour réaliser une partition des données, d'autre part quelle valeur de cette variable utiliser comme point de scission ("split-point"). L'entropie correspond à une mesure du désordre : plus sa valeur est grande, plus le groupe de données est hétérogène (i.e. plus la proportion de chaque classe d'individus dans le groupe est élevée). L'indice de Gini correspond davantage à un indice d'impureté ; de même que pour l'entropie, plus sa valeur est grande, plus le groupe de données est hétérogène (ou "impur"). Chacun de ces indices a une formule de calcul qui lui est propre.

[26]:
```
# (2.5) Représentation de l'arbre mobilisant l'entropie
plt.figure(figsize = (40/2.54, 25/2.54))
plot_tree(dt_entropy, class_names = ["Malin", "Bénin"], fontsize = 8, rounded␣
↪= True)
plt.show()
```



[27]:
```
# (2.6) Représentation de l'arbre mobilisant l'indice de Gini
plt.figure(figsize = (40/2.54, 25/2.54))
plot_tree(dt_gini, class_names = ["Malin", "Bénin"], fontsize = 8, rounded =␣
↪True)
plt.show()
```



[28]:
```
# (2.7) Affichage de la valeur de la métrique de performance pour chacun de␣
↪ces deux arbres
y_pred_entropy = dt_entropy.predict(X_test)
y_pred_gini = dt_gini.predict(X_test)
print("Performance de l'arbre mobilisant l'entropie = {:.2f}".
↪format(accuracy_score(y_test, y_pred_entropy)))
print("Performance de l'arbre mobilisant l'indice de Gini = {:.2f}".
↪format(accuracy_score(y_test, y_pred_gini)))
```

```
Performance de l'arbre mobilisant l'entropie = 0.93
Performance de l'arbre mobilisant l'indice de Gini = 0.93
```

La performance des deux arbres créés est égale. Cela signifie que ces deux critères d'information que sont l'entropie et l'indice de Gini sont ici autant efficaces l'un que l'autre pour créer un arbre classant les tumeurs de cancers du sein.

2.3 Exercice 3 : Arbre de régression

[29]:
```
# (3.1) Import du jeu de données "auto-mpg.csv"
auto_mpg_orig = pd.read_csv(r"C:\Users\quent\Documents\DU Data Analyst␣
↪2022\UE n°6 - Introduction au Machine Learning\Jeux de données\auto-mpg.
↪csv", low_memory = False, encoding = "latin-1")
auto_mpg_orig.head()
```

[29]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year␣ ↪ \
0	18.0	8	307.0	130	3504	12.0	70
1	15.0	8	350.0	165	3693	11.5	70
2	18.0	8	318.0	150	3436	11.0	70
3	16.0	8	304.0	150	3433	12.0	70
4	17.0	8	302.0	140	3449	10.5	70

	origin	car name
0	1	chevrolet chevelle malibu
1	1	buick skylark 320
2	1	plymouth satellite
3	1	amc rebel sst
4	1	ford torino

[30]:
```
# (3.2) Inspection des variables
print(auto_mpg_orig.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64
 2   displacement  398 non-null    float64
 3   horsepower    398 non-null    object
 4   weight        398 non-null    int64
```

```
 5   acceleration  398 non-null    float64
 6   model year    398 non-null    int64
 7   origin        398 non-null    int64
 8   car name      398 non-null    object
dtypes: float64(3), int64(4), object(2)
memory usage: 28.1+ KB
None
```

[31]:
```
# (3.3) Conversion de la variable "horsepower" en nombres entiers et␣
↪suppression des lignes contenant des valeurs manquantes
horsepower = []
for i in list(auto_mpg_orig["horsepower"]):
    if i == "?":
        horsepower.append(np.nan)
    else:
        horsepower.append(int(i))
auto_mpg_orig["horsepower"] = horsepower
auto_mpg = auto_mpg_orig.dropna()
```

[32]:
```
# (3.4) Création de jeux de données d'entraînement et de test
X = auto_mpg[["cylinders", "displacement", "horsepower", "weight",␣
↪"acceleration", "model year"]].to_numpy()
y = auto_mpg["mpg"].to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 1)
```

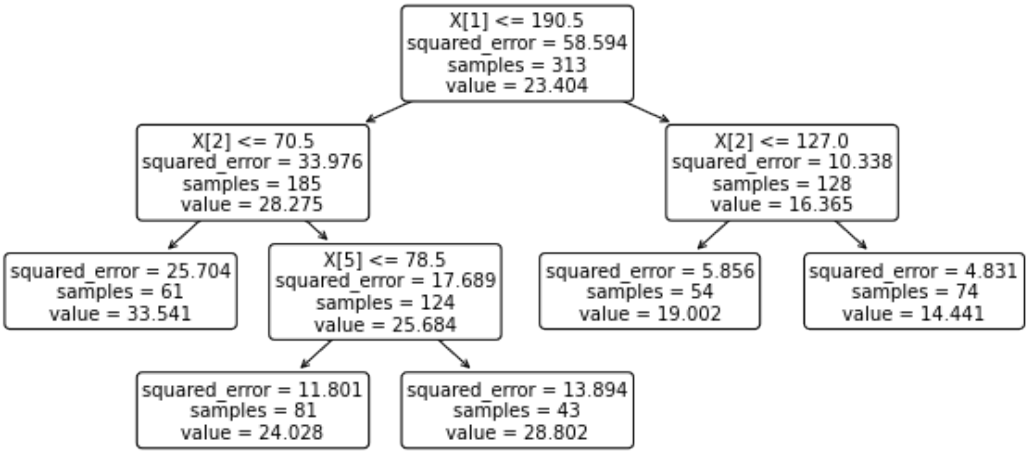
[33]:
```
# (3.5) Import de la fonction DecisionTreeRegressor à partir du package␣
↪sklearn.tree
from sklearn.tree import DecisionTreeRegressor
```

[34]:
```
# (3.6) Création d'un arbre de régression nommé dt, avec une profondeur␣
↪maximale de 8 et un pourcentage minimum d'observations par feuille égal à␣
↪13%
dt = DecisionTreeRegressor(max_depth = 8, min_samples_leaf = 0.13,␣
↪random_state = 1)
```

[35]:
```
# (3.7) Entraînement de l'arbre sur le jeu de données d'entraînement
dt.fit(X_train, y_train)
```

[35]:
```
DecisionTreeRegressor(max_depth=8, min_samples_leaf=0.13, random_state=1)
```

[36]:
```
# (3.8) Représentation de l'arbre de classification
plt.figure(figsize = (26/2.54, 12/2.54))
plot_tree(dt, fontsize = 10, rounded = True)
plt.show()
```



[37]:
```
# (3.9) Import de la fonction mean squared error à partir du package sklearn.metrics
from sklearn.metrics import mean_squared_error as MSE
```

[38]:
```
# (3.10) Utilisation de l'arbre pour prédire les valeurs de la variable "mpg" du jeu de données test
y_pred = dt.predict(X_test)
```

[39]:
```
# (3.11) Calcul du MSE et du RMSE
mse_dt = MSE(y_test, y_pred)
rmse_dt = mse_dt**(1/2)
print("MSE = {:.2f}".format(mse_dt))
print("RMSE = {:.2f}".format(rmse_dt))
```

```
MSE = 18.63
RMSE = 4.32
```

Les valeurs réelles (y_test) et prédites par le modèle (y_pred) doivent être mobilisées ensemble pour calculer le MSE et le RMSE, car ces métriques sont justement calculées à partir de la différence entre les valeurs réelles et prédites. Par rapport au MSE, le RMSE a pour avantage d'avoir la même unité que ces valeurs.

[40]:
```
# (3.12) Calcul du RMSE à partir d'une régression linéaire simple
from sklearn.linear_model import LinearRegression
lr = LinearRegression().fit(X_train, y_train)
lr_y_pred = lr.predict(X_test)
rmse_lr = MSE(y_test, lr_y_pred)**(1/2)
print("RMSE = {:.2f}".format(rmse_lr))
```

```
RMSE = 3.59
```

2.4 Exercice 4 : Biais, variance, erreur de généralisation

Le biais et la variance sont deux sources d'erreurs rencontrées lorsqu'on essaie d'optimiser les paramètres d'un modèle. Le biais est une erreur consistant à choisir des paramètres qui induisent un sous-entraînement du modèle, c'est-à-dire une imprécision dans la prédiction des labels des

données d'entraînement. A l'inverse, l'erreur de variance consiste à choisir des paramètres qui permettent une prédiction très précise des labels des données d'entraînement (du fait d'une forte sensibilité du modèle aux faibles fluctuations dans ces données), mais qui échouent à prédire les labels de nouvelles données dès lors que celles-ci diffèrent de celles d'entraînement (sur-entraînement du modèle). Ces deux erreurs varient de manière opposée avec la complexité d'un modèle : l'erreur de variance augmente avec la complexité tandis que le biais diminue.

La validation croisée est une méthode permettant de limiter le risque de réaliser un échantillonnage non-représentatif dans le jeu de données initial (par exemple la sélection d'une seule catégorie parmi deux pour construire le jeu d'entraînement). Le principe de cette méthode est de réaliser plusieurs échantillonnages des données de sorte à obtenir des jeux d'entraînement légèrement différents. Les métriques de performance du modèle sont ensuite calculées à partir de chacun des jeux d'entraînement, puis la moyenne de ces métriques est calculée pour obtenir la performance globale du modèle. Par exemple, dans le cas de dix échantillonnages ("10-fold"), la moyenne est calculée sur les dix métriques obtenues à partir de chaque échantillon. Cette approche permet de minimiser l'erreur de variance et donc d'éviter un sur-entraînement du modèle.

[41]:
```
# (4.1) Création de jeux de données d'entraînement (70% des données) et de␣
↪test (30%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
↪random_state = 1)
```

[42]:
```
# (4.2) Création d'un arbre de régression nommé dt, avec une profondeur␣
↪maximale de 4 et un pourcentage minimum d'observations par feuille égal à␣
↪26%
dt = DecisionTreeRegressor(max_depth = 4, min_samples_leaf = 0.26,␣
↪random_state = 1)
```

[43]:
```
# (4.3) Calcul du MSE et du RMSE issus de la validation croisée (10 fold)
MSE_CV = - cross_val_score(dt, X_train, y_train, cv = 10, scoring =␣
↪"neg_mean_squared_error", n_jobs = -1)
print("MSE de la validation croisée = {:.2f}".format(MSE_CV.mean()))
print("RMSE de la validation croisée = {:.2f}".format(MSE_CV.mean()**(1/2)))
```

```
MSE de la validation croisée = 18.48
RMSE de la validation croisée = 4.30
```

[44]:
```
# (4.4) Calcul du RMSE sur le jeu d'entraînement (RMSE_train)
dt.fit(X_train, y_train)
RMSE_train = MSE(y_train, dt.predict(X_train))**(1/2)
print("RMSE_train = {:.2f}".format(RMSE_train))
```

```
RMSE_train = 4.19
```

2.5 Exercice 5 : Bagging

Un bootstrap est une itération d'un arbre de décision, qui porte sur un échantillon du jeu d'entraînement. Le bootstrapping consiste à réaliser plusieurs bootstraps en faisant varier à chaque fois l'échantillon d'entraînement (tirage aléatoire avec remise), tout en gardant dans chaque bootstrap l'intégralité des variables initiales qui seront utilisées comme critères de décision. Le bagging désigne l'agrégation des bootstraps : dans le cas d'une classification, le résultat

final est le label majoritaire parmi l'ensemble des labels prédits par chaque bootstrap, tandis que pour une régression, le résultat final est la moyenne des résultats de chaque bootstrap.

[45]:
```
# (5.1) Import du jeu de données "indian_liver_patient.csv"
indian_liver_patient_orig = pd.read_csv(r"C:\Users\quent\Documents\DU Data␣
↪Analyst 2022\UE n°6 - Introduction au Machine Learning\Jeux de␣
↪données\indian_liver_patient.csv", low_memory = False, encoding = "latin-1")
indian_liver_patient_orig.head()
```

[45]:

	Age	Gender	Total_Bilirubin	Direct_Bilirubin	Alkaline_Phosphotase	\
0	65	Female	0.7	0.1	187	
1	62	Male	10.9	5.5	699	
2	62	Male	7.3	4.1	490	
3	58	Male	1.0	0.4	182	
4	72	Male	3.9	2.0	195	

	Alamine_Aminotransferase	Aspartate_Aminotransferase	Total_Protiens	\
0	16	18	6.8	
1	64	100	7.5	
2	60	68	7.0	
3	14	20	6.8	
4	27	59	7.3	

	Albumin	Albumin_and_Globulin_Ratio	Dataset
0	3.3	0.90	1
1	3.2	0.74	1
2	3.3	0.89	1
3	3.4	1.00	1
4	2.4	0.40	1

[83]:
```
# (5.2) Inspection des variables
print(indian_liver_patient_orig.info())
print()
print("Nombre de malades (1) et de non-malades (2)")
print(indian_liver_patient_orig["Dataset"].value_counts())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 583 entries, 0 to 582
Data columns (total 11 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   Age                         583 non-null    int64
 1   Gender                      583 non-null    object
 2   Total_Bilirubin             583 non-null    float64
 3   Direct_Bilirubin            583 non-null    float64
 4   Alkaline_Phosphotase        583 non-null    int64
 5   Alamine_Aminotransferase    583 non-null    int64
 6   Aspartate_Aminotransferase  583 non-null    int64
 7   Total_Protiens              583 non-null    float64
 8   Albumin                     583 non-null    float64
 9   Albumin_and_Globulin_Ratio  579 non-null    float64
 10  Dataset                     583 non-null    int64
```

```
dtypes: float64(5), int64(5), object(1)
memory usage: 50.2+ KB
None

Nombre de malades (1) et de non-malades (2)
1    416
2    167
Name: Dataset, dtype: int64
```

[47]:
```
# (5.3) Suppression des valeurs manquantes
indian_liver_patient = indian_liver_patient_orig.dropna()
```

[48]:
```
# (5.4) Conversion des labels 1 et 2 de la colonne "Dataset" en 1 (patient␣
 ↪malade) et 0 (patient non-malade)
dataset = []
for i in indian_liver_patient.index:
    if indian_liver_patient["Dataset"][i] == 1:
        dataset.append(1)
    else:
        dataset.append(0)
indian_liver_patient["Dataset"] = dataset
```

[49]:
```
# (5.5) Création de jeux de données d'entraînement (70% des données) et de␣
 ↪test (30%)
X = indian_liver_patient.iloc[:,2:10].to_numpy()
y = indian_liver_patient["Dataset"].to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
 ↪random_state = 1)
```

[50]:
```
# (5.6) Import de la fonction BaggingClassifier à partir du package sklearn.
 ↪ensemble
from sklearn.ensemble import BaggingClassifier
```

[51]:
```
# (5.7) Création et entraînement d'un arbre de classification nommé dt (avec␣
 ↪une "seed" égale à 1)
dt = DecisionTreeClassifier(random_state = 1)
dt.fit(X_train, y_train)
```

[51]:
```
DecisionTreeClassifier(random_state=1)
```

[52]:
```
# (5.8) Création et entraînement d'un bagging, puis calcul de son score de␣
 ↪performance
bc = BaggingClassifier(base_estimator = dt, n_estimators = 300, oob_score =␣
 ↪True, n_jobs = -1, random_state = 2)
bc.fit(X_train, y_train)
y_pred_bc = bc.predict(X_test)
print("Performance du bagging = {:.2f}".format(accuracy_score(y_test,␣
 ↪y_pred_bc)))
```

```
Performance du bagging = 0.72
```

[53]:
```
# (5.9) Calcul du score de performance d'un arbre de décision simple
y_pred_dt = dt.predict(X_test)
print("Performance d'un arbre de décision simple = {:.2f}".
 ↪format(accuracy_score(y_test, y_pred_dt)))
```

```
Performance d'un arbre de décision simple = 0.69
```

Comme attendu, la performance du bagging est un peu plus élevée que celle d'un arbre de décision simple.

L'OOB accuracy, ou performance "Out-Of-Bag" du bagging, désigne la performance moyenne calculée sur les données non-incluses dans les échantillons de chaque bootstrap. Par exemple, si un bootstrap est réalisé sur 70% des données d'entraînement, les 30% restant vont être utilisés pour calculer la performance "Out-Of-Bag" du bootstrap. L'OOB accuracy du bagging sera égal à la moyenne des performances OOB de chaque bootstrap.

[54]:
```
# (5.10) Calcul de l'OOB accuracy du bagging
print("OOB accuracy = {:.2f}".format(bc.oob_score_))
```

```
OOB accuracy = 0.67
```

La performance OOB du bagging est ici inférieure à la performance calculée sur le jeu de données test ($0.67 < 0.72$). La valeur ajoutée de la performance OOB est de prendre en compte l'intégralité des données dans le calcul. En effet, certaines données d'entraînement peuvent ne pas être incluses dans les échantillons des bootstraps, du fait du caractère aléatoire de l'échantillonnage, ce qui aboutit alors à une perte d'information lors du calcul de la performance sur le jeu de données test. Le calcul de la performance OOB est surtout utile lorsque le jeu de données est de petite taille, car dans ce cas la non-prise en compte de certaines données peut biaiser considérablement les prédictions du modèle.

2.6 Exercice 6 : Forêt aléatoire

Une forêt aléatoire fonctionne de la même manière qu'un bagging sauf en ce qui concerne le choix des variables : pour chaque arbre et à chaque noeud de l'arbre, la variable utilisée comme critère de décision est choisie parmi un sous-ensemble des variables candidates. Ce sous-ensemble est défini aléatoirement à chaque noeud de l'arbre, et comprend un nombre de variables égal à la racine carrée du nombre total de variables candidates. L'intérêt de cette méthode est de limiter la corrélation entre les arbres, qui est particulièrement forte dans le cas du bagging. Un autre avantage est la diminution du temps de calcul par l'algorithme.

[55]:
```
# (6.1) Import du jeux de données "bikeshare.csv"
bikeshare_orig = pd.read_csv(r"C:\Users\quent\Documents\DU Data Analyst␣
 ↪2022\UE n°6 - Introduction au Machine Learning\Jeux de données\bikeshare.
 ↪csv", low_memory = False, encoding = "latin-1")
bikeshare_orig.head()
```

[55]:

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395					
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635					
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635					
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395					
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395					

```
0        81        0.0       3          13      16
1        80        0.0       8          32      40
2        80        0.0       5          27      32
3        75        0.0       3          10      13
4        75        0.0       0           1       1
```

[56]:
```
# (6.2) Inspection des variables
print(bikeshare_orig.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    10886 non-null  object
 1   season      10886 non-null  int64
 2   holiday     10886 non-null  int64
 3   workingday  10886 non-null  int64
 4   weather     10886 non-null  int64
 5   temp        10886 non-null  float64
 6   atemp       10886 non-null  float64
 7   humidity    10886 non-null  int64
 8   windspeed   10886 non-null  float64
 9   casual      10886 non-null  int64
 10  registered  10886 non-null  int64
 11  count       10886 non-null  int64
dtypes: float64(3), int64(8), object(1)
memory usage: 1020.7+ KB
None
```

[57]:
```
# (6.3) Import de la fonction RandomForestRegressor à partir du package␣
↪sklearn.ensemble
from sklearn.ensemble import RandomForestRegressor
```

[58]:
```
# (6.4) Création de jeux de données d'entraînement (80% des données) et de␣
↪test (20%)
X = bikeshare_orig.iloc[:,1:9].to_numpy()
y = bikeshare_orig["count"].to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 1)
```

[59]:
```
# (6.5) Création et entraînement d'une forêt aléatoire nommée rf
rf = RandomForestRegressor(n_estimators = 300, random_state = 1)
rf.fit(X_train, y_train)
```

[59]:
```
RandomForestRegressor(n_estimators=300, random_state=1)
```

[60]:
```
# (6.6) Calcul du RMSE de la forêt aléatoire sur le jeu de données␣
↪d'entraînement
RMSE_rf = MSE(y_test, rf.predict(X_test))**(1/2)
print("RMSE de la forêt aléatoire = {:.2f}".format(RMSE_rf))
```

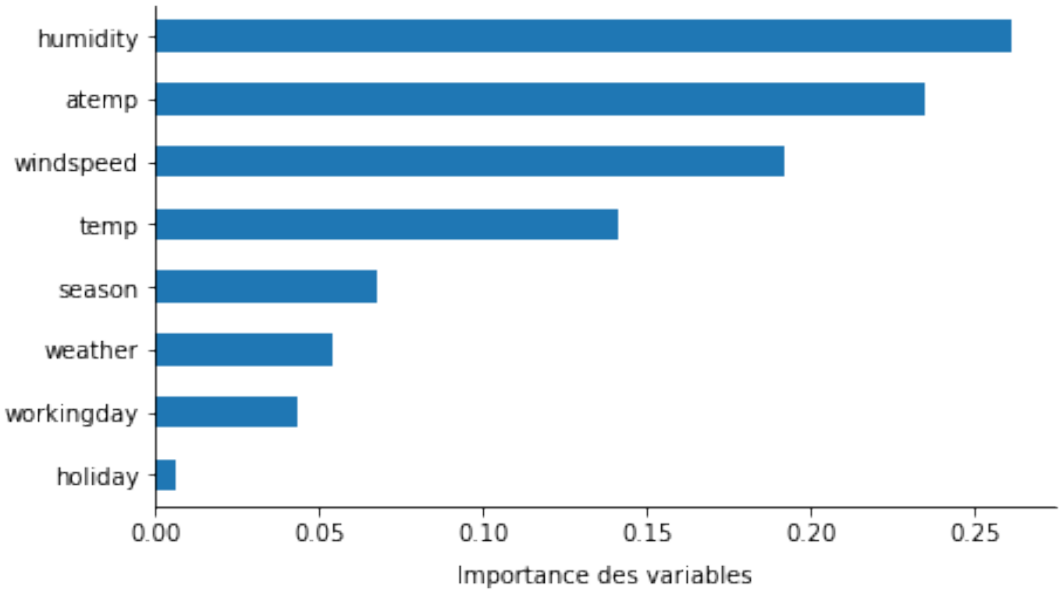
```
RMSE de la forêt aléatoire = 153.84
```

[61]:
```
# (6.7) Calcul du RMSE d'un arbre de régression simple
dt = DecisionTreeRegressor(random_state = 1)
dt.fit(X_train, y_train)
RMSE_dt = MSE(y_test, dt.predict(X_test))**(1/2)
print("RMSE d'un arbre de régression simple = {:.2f}".format(RMSE_dt))
```

```
RMSE d'un arbre de régression simple = 198.00
```

Comme attendu, la performance de l'arbre de régression simple est inférieure (RMSE plus faible) à celle de la forêt aléatoire.

[62]:
```
# (6.8) Calcul et représentation de l'importance des variables dans la␣
↪création de la forêt
plt.figure(figsize = (17.5/2.54, 10/2.54))
importances_rf = pd.Series(rf.feature_importances_, index = bikeshare_orig.
↪iloc[:,1:9].columns).sort_values()
importances_rf.plot(kind = "barh")
plt.xlabel("Importance des variables", labelpad = 8)
plt.gca().spines[["top", "right"]].set_visible(False)
plt.show()
```



2.7 Exercice 7 : Hyperparamètres et grid search, une introduction

[65]:
```
# (7.1) Création d'un arbre de classification ayant les caractéristiques␣
↪suivantes : indice de Gini, au moins 1 échantillon restant dans les␣
↪feuilles, au moins 2 échantillons dans un noeud et une seed égale à 1
dt = DecisionTreeClassifier(criterion = "gini", min_samples_leaf = 1,␣
↪random_state = 1, min_samples_split = 2)
```

[66]:
```
# (7.2) Entraînement de l'arbre sur le jeu de données d'entraînement␣
↪"indian_liver_patient"
X = indian_liver_patient.iloc[:,2:10].to_numpy()
y = indian_liver_patient["Dataset"].to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
↪random_state = 1)
dt.fit(X_train, y_train)
```

[66]:
```
DecisionTreeClassifier(random_state=1)
```

[67]:
```
# (7.3) Création d'un dictionnaire (params_dt) comprenant deux vecteurs,␣
↪définissant la profondeur maximale de l'arbre (max_depth) et le nombre␣
↪minimum d'échantillons par feuille (min_samples_leaf)
params_dt = {"max_depth": [2, 3, 4], "min_samples_leaf": [0.12, 0.14, 0.16, 0.
↪18]}
```

La sensibilité d'un modèle est le rapport entre le nombre de cas positifs correctement prédits et le nombre total de cas positifs réels. A l'inverse, la spécificité est le rapport entre le nombre de cas négatifs correctement prédits et le nombre total de cas négatifs réels. L'AUC d'un modèle est une métrique de performance égale à l'aire sous la courbe ROC du modèle, qui est une courbe représentant le taux de vrais positifs (sensibilité) en fonction du taux de faux positifs (1 - spécificité). Un modèle ayant un AUC proche de 0.5 est un modèle relativement inutile, car équivalent à une classification aléatoire. Un modèle très performant aura un AUC proche de 1.

[68]:
```
# (7.4) Import de la fonction GridSearchCV à partir du package sklearn.
↪model_selection
from sklearn.model_selection import GridSearchCV
```

[69]:
```
# (7.5) Création d'une grid search puis application sur les données␣
↪d'entraînement
grid_dt = GridSearchCV(estimator = dt, param_grid = params_dt, scoring =␣
↪"roc_auc", cv = 5, n_jobs = -1)
grid_dt.fit(X_train, y_train)
```

[69]:
```
GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=1),␣
↪n_jobs=-1,
             param_grid={'max_depth': [2, 3, 4],
                         'min_samples_leaf': [0.12, 0.14, 0.16, 0.18]},
             scoring='roc_auc')
```

[70]:
```
# (7.6) Extraction du meilleur modèle (best_model)
best_model = grid_dt.best_estimator_
print(best_model)
```

```
DecisionTreeClassifier(max_depth=3, min_samples_leaf=0.12, random_state=1)
```

[71]:
```
# (7.7) Prédiction des probabilités d'avoir une maladie du foie pour les␣
↪patients du jeu de données test
y_pred_proba = best_model.predict_proba(X_test)[:,1]
print(y_pred_proba)
```

```
[0.85185185 0.9625     0.44897959 0.65306122 0.85714286 0.85714286
```

```
0.58666667 0.9625     0.9625     0.58666667 0.85714286 0.65306122
0.44897959 0.9625     0.58666667 0.85714286 0.85714286 0.44897959
0.44897959 0.9625     0.9625     0.44897959 0.65306122 0.44897959
0.85714286 0.58666667 0.65306122 0.44897959 0.65306122 0.9625
0.44897959 0.44897959 0.44897959 0.65306122 0.85714286 0.85185185
0.85185185 0.58666667 0.58666667 0.44897959 0.9625     0.9625
0.58666667 0.85185185 0.85185185 0.44897959 0.58666667 0.85714286
0.85185185 0.85714286 0.9625     0.58666667 0.65306122 0.58666667
0.85714286 0.58666667 0.44897959 0.9625     0.44897959 0.85714286
0.44897959 0.85185185 0.44897959 0.9625     0.65306122 0.85714286
0.44897959 0.44897959 0.9625     0.85185185 0.9625     0.85714286
0.85714286 0.65306122 0.58666667 0.85185185 0.58666667 0.65306122
0.44897959 0.65306122 0.85714286 0.44897959 0.58666667 0.44897959
0.65306122 0.65306122 0.44897959 0.58666667 0.58666667 0.9625
0.85714286 0.85185185 0.65306122 0.65306122 0.44897959 0.44897959
0.58666667 0.58666667 0.58666667 0.58666667 0.9625     0.85714286
0.85185185 0.58666667 0.44897959 0.58666667 0.58666667 0.9625
0.9625     0.44897959 0.44897959 0.58666667 0.65306122 0.65306122
0.44897959 0.85185185 0.85185185 0.85185185 0.85714286 0.9625
0.44897959 0.65306122 0.58666667 0.85185185 0.44897959 0.65306122
0.58666667 0.85714286 0.44897959 0.9625     0.85185185 0.44897959
0.85185185 0.9625     0.85714286 0.9625     0.44897959 0.44897959
0.9625     0.58666667 0.85714286 0.85714286 0.58666667 0.58666667
0.65306122 0.65306122 0.58666667 0.9625     0.85714286 0.58666667
0.44897959 0.65306122 0.9625     0.58666667 0.85185185 0.9625
0.85714286 0.85714286 0.85185185 0.65306122 0.44897959 0.9625
0.9625     0.85714286 0.65306122 0.44897959 0.85714286 0.85185185
0.65306122 0.44897959 0.44897959 0.9625     0.44897959 0.85185185]
```

[72]:
```
# (7.8) Calcul de l'AUC du meilleur modèle
best_model_roc_auc_score = roc_auc_score(y_test, y_pred_proba)
print("AUC = {:.2f}".format(best_model_roc_auc_score))
```

```
AUC = 0.74
```

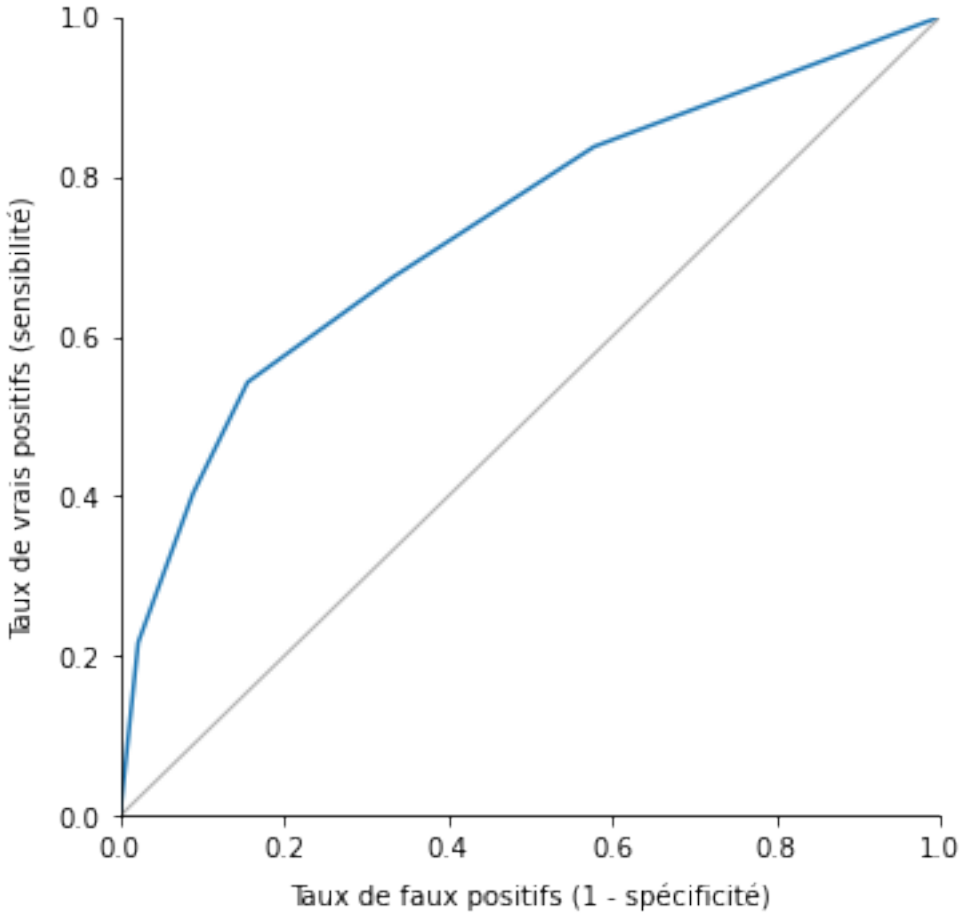
[73]:
```
# (7.9) Affichage de la matrice de confusion du meilleur modèle (sur le jeu␣
↪de données test)
y_pred = best_model.predict(X_test)
TP_list = []
FP_list = []
FN_list = []
TN_list = []
for i in range(len(list(zip(y_pred, y_test)))):
    if list(zip(y_pred, y_test))[i] == (1, 1):
        TP_list.append(1)
    elif list(zip(y_pred, y_test))[i] == (1, 0):
        FP_list.append(1)
    elif list(zip(y_pred, y_test))[i] == (0, 1):
        FN_list.append(1)
    else:
        TN_list.append(1)
TP = sum(TP_list)
```

```
FP = sum(FP_list)
FN = sum(FN_list)
TN = sum(TN_list)
df_conf_matrix = pd.DataFrame({"Malade": [TP, FN, TP+FN], "Non-malade": [FP,␣
↪TN, FP+TN], "Total": [TP+FP, FN+TN, TP+FN+FP+TN]})
df_conf_matrix.columns = pd.MultiIndex.from_product([["Cas réels"],␣
↪["Malade", "Non-malade", "Total"]])
df_conf_matrix.index = pd.MultiIndex.from_product([["Cas prédits"],␣
↪["Malade", "Non-malade", "Total"]])
df_conf_matrix
```

[73]:

		Cas réels		
		Malade	Non-malade	Total
Cas prédits	Malade	108	26	134
	Non-malade	21	19	40
	Total	129	45	174

[74]:
```
# (7.10) Représentation de la courbe ROC du modèle
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
plt.figure(figsize = (14/2.54, 14/2.54))
plt.plot(fpr,tpr)
plt.axline((0, 0), slope = 1, linewidth = 1, color = "darkgrey")
plt.ylabel("Taux de vrais positifs (sensibilité)", labelpad = 8)
plt.xlabel("Taux de faux positifs (1 - spécificité)", labelpad = 8)
plt.ylim(0,1)
plt.xlim(0,1)
plt.gca().spines[["right", "top"]].set_visible(False)
plt.show()
```



Le rappel et la précision sont deux autres métriques de performance d'un arbre de classification. Le rappel est une mesure de qualité, qui est calculée de la même manière que la sensibilité dans le cas d'une classification binaire (rapport entre le nombre de cas positifs correctement prédits et le nombre total de cas positifs réels). La précision est une mesure de quantité, qui correspond à la proportion de cas positifs correctement prédits sur l'ensemble des prédictions de cas positifs.

[75]:
```
# (7.11) Calcul du rappel et de la précision du meilleur modèle
recall = TP / (TP+FN)
precision = TP / (TP+FP)
print("Rappel = {:.2f}".format(recall))
print("Précision = {:.2f}".format(precision))
```

```
Rappel = 0.84
Précision = 0.81
```

2.8 Exercice 8 : Une grid search en autonomie

[76]:
```
# (8.1) Création d'un dictionnaire (params_rf) comprenant trois vecteurs :␣
↪n_estimators, max_features et min_samples_leaf
params_rf = {"n_estimators": [100, 350, 500], "max_features": ["log2",␣
↪"auto", "sqrt"], "min_samples_leaf": [2, 10, 30]}
```

Les paramètres sur lesquels la grid search est effectuée correspondent au nombre d'arbres dans

la forêt (n_estimators), au nombre maximum de variables à prendre en compte à chaque noeud pour choisir le critère de décision (max_features) et au nombre minimum d'échantillons qu'il doit rester dans chaque feuille (min_samples_leaf).

[77]:
```
# (8.2) Création de jeux de données d'entraînement (80% des données) et de␣
↪test (20%)
X = bikeshare_orig.iloc[:,1:9].to_numpy()
y = bikeshare_orig["count"].to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 1)
```

[78]:
```
# (8.3) Création et entraînement d'une forêt aléatoire nommée rf
rf = RandomForestRegressor(random_state = 1)
rf.fit(X_train, y_train)
```

[78]:
```
RandomForestRegressor(random_state=1)
```

[79]:
```
# (8.4) Création d'une grid search puis application sur les données␣
↪d'entraînement
grid_rf = GridSearchCV(estimator = rf, param_grid = params_rf, scoring =␣
↪"neg_root_mean_squared_error", cv = 3, n_jobs = -1)
grid_rf.fit(X_train, y_train)
```

[79]:
```
GridSearchCV(cv=3, estimator=RandomForestRegressor(random_state=1), n_jobs=-1,
             param_grid={'max_features': ['log2', 'auto', 'sqrt'],
                         'min_samples_leaf': [2, 10, 30],
                         'n_estimators': [100, 350, 500]},
             scoring='neg_root_mean_squared_error')
```

[80]:
```
# (8.5) Extraction du meilleur modèle (best_model)
best_model = grid_rf.best_estimator_
print(best_model)
```

```
RandomForestRegressor(max_features='sqrt', min_samples_leaf=2, n_estimators=500,
                      random_state=1)
```

[81]:
```
# (8.6) Calcul du RMSE du meilleur modèle sur le jeu de données d'entraînement
rmse_best_model = MSE(y_test, best_model.predict(X_test))**(1/2)
print("RMSE = {:.2f}".format(rmse_best_model))
```

```
RMSE = 147.25
```