Importer les bibliothèques nécessaires

Nous importons les bibliothèques nécessaires pour travailler avec les données et entraîner notre modèle.

- pandas (pd) : utilisé pour manipuler et analyser les données tabulaires.
- RandomForestRegressor : l'algorithme d'apprentissage automatique que nous allons utiliser pour créer notre modèle.
- files de google.colab : utilisé pour téléverser des fichiers dans l'environnement de Colab.

Nous avons préférer utiliser Pandas au lieu de pySpark au vu du peu de ligne que contient le jeu de donnée (environ 1600)

```
# Importer pandas pour la manipulation de données
import pandas as pd

# Importer RandomForestRegressor de scikit-learn pour créer notre modèle
from sklearn.ensemble import RandomForestRegressor

# Importer files de google.colab pour téléverser des fichiers
from google.colab import files

# Téléverser des fichiers depuis l'appareil local vers l'environnement de Colab
files.upload()
```

Charger les données à partir d'un fichier CSV

Nous utilisons la fonction read_csv de pandas pour charger les données à partir d'un fichier CSV. Le fichier "wine.csv" est chargé et stocké dans le DataFrame data_wine.

Ensuite, nous affichons les types de données de chaque colonne à l'aide de la méthode dtypes.

```
# Charger les données à partir d'un fichier CSV
data_wine = pd.read_csv("wine.csv")

# Afficher les types de données de chaque colonne
print(data_wine.dtypes)
```

```
fixed acidity           float64
volatile acidity        float64
citric acid             float64
residual sugar          float64
chlorides               float64
free sulfur dioxide     float64
```

```
total sulfur dioxide    float64
density                 float64
pH                      float64
sulphates               float64
alcohol                 float64
quality                   int64
dtype: object
```

Nous allons donc travailler avec des données exclusivement numériques. Cependant, il est notable que la colonne "quality" peut être traitée de deux manières différentes : soit comme un système linéaire où nous prendrions la valeur entière la plus proche, soit au contraire comme des classes distinctes.

I - Analyse exploratoire des données (EDA) à l'aide de visualisations

Nous utilisons les bibliothèques seaborn (sns) et matplotlib.pyplot (plt) pour créer différentes visualisations afin d'explorer les caractéristiques et les relations dans notre jeu de données.

```
# Importer les bibliothèques de visualisation
import seaborn as sns
import matplotlib.pyplot as plt
```

Affichage du nombre de valeurs nulles pour chaque colonne.

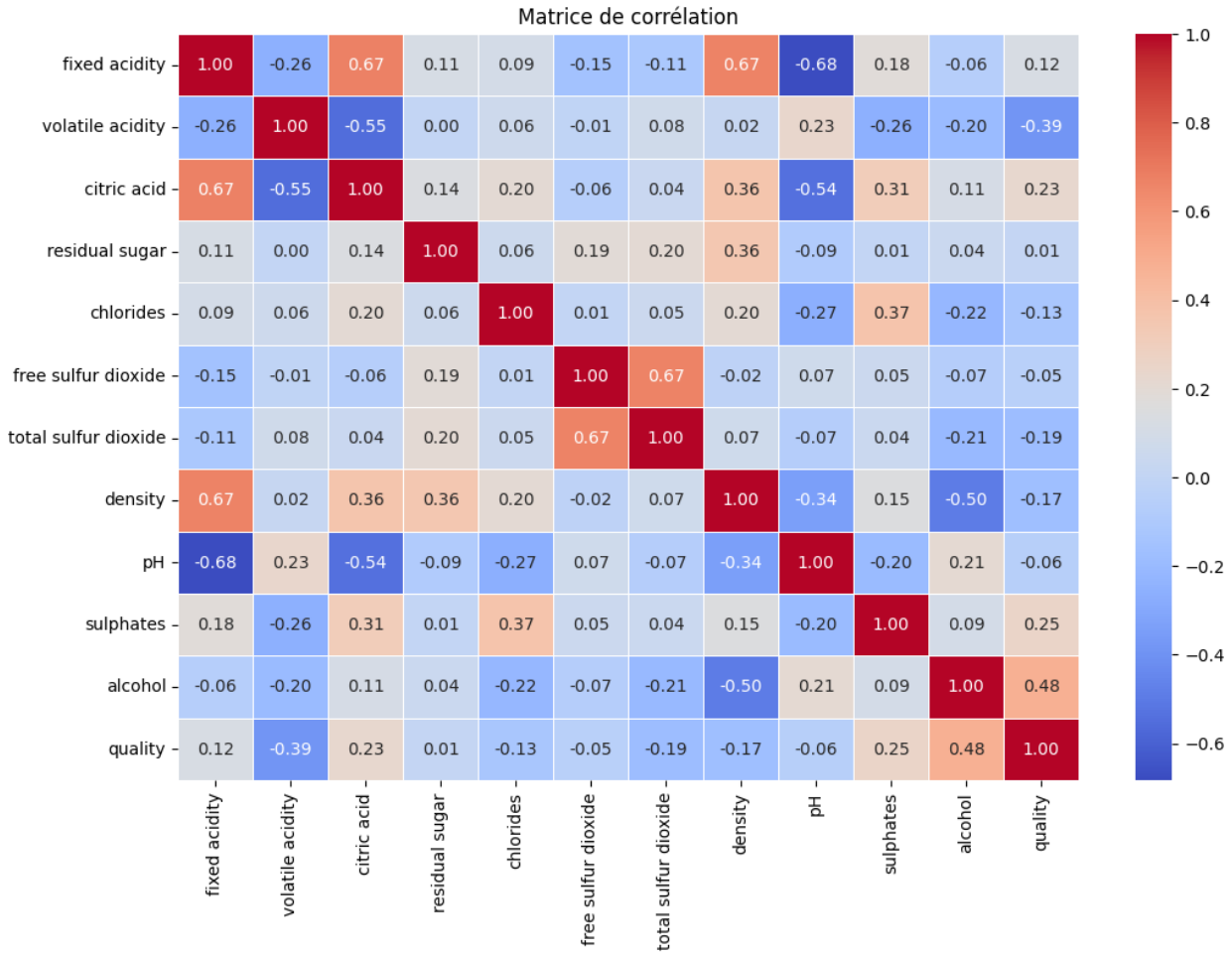
```
# Afficher le nombre de valeurs nulles pour chaque colonne
print("Valeurs nulles:")
print(data_wine.isnull().sum())
```

```
Valeurs nulles:
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

Nous constatons que le jeux de données est complet, il ne manque aucune valeur, que qui va simplifier son analyse.

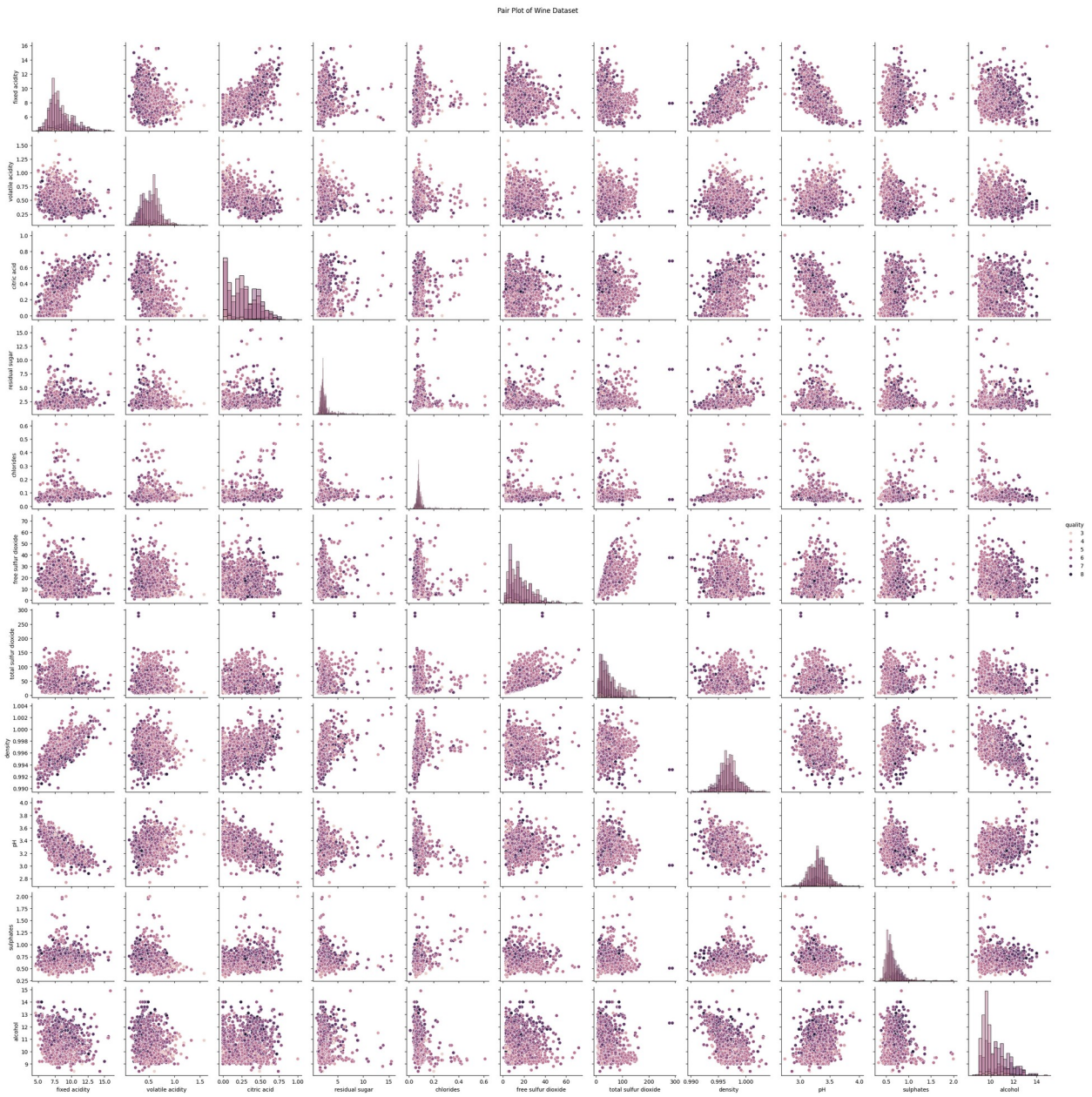
Création et affichage d'une matrice de corrélation sous forme de heatmap.

```
# Créer et afficher une matrice de corrélation sous forme de heatmap
correlation_matrix = data_wine.corr()
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm",
fmt=".2f", linewidths=.5)
plt.title("Matrice de corrélation")
plt.show()
```



Création et affichage d'un pair plot pour explorer les relations entre les variables, coloré par la qualité.

```
# Créer et afficher un pair plot coloré par la qualité
sns.pairplot(data_wine, hue='quality', diag_kind='hist')
plt.suptitle("Pair Plot of Wine Dataset", y=1.02)
plt.show()
```



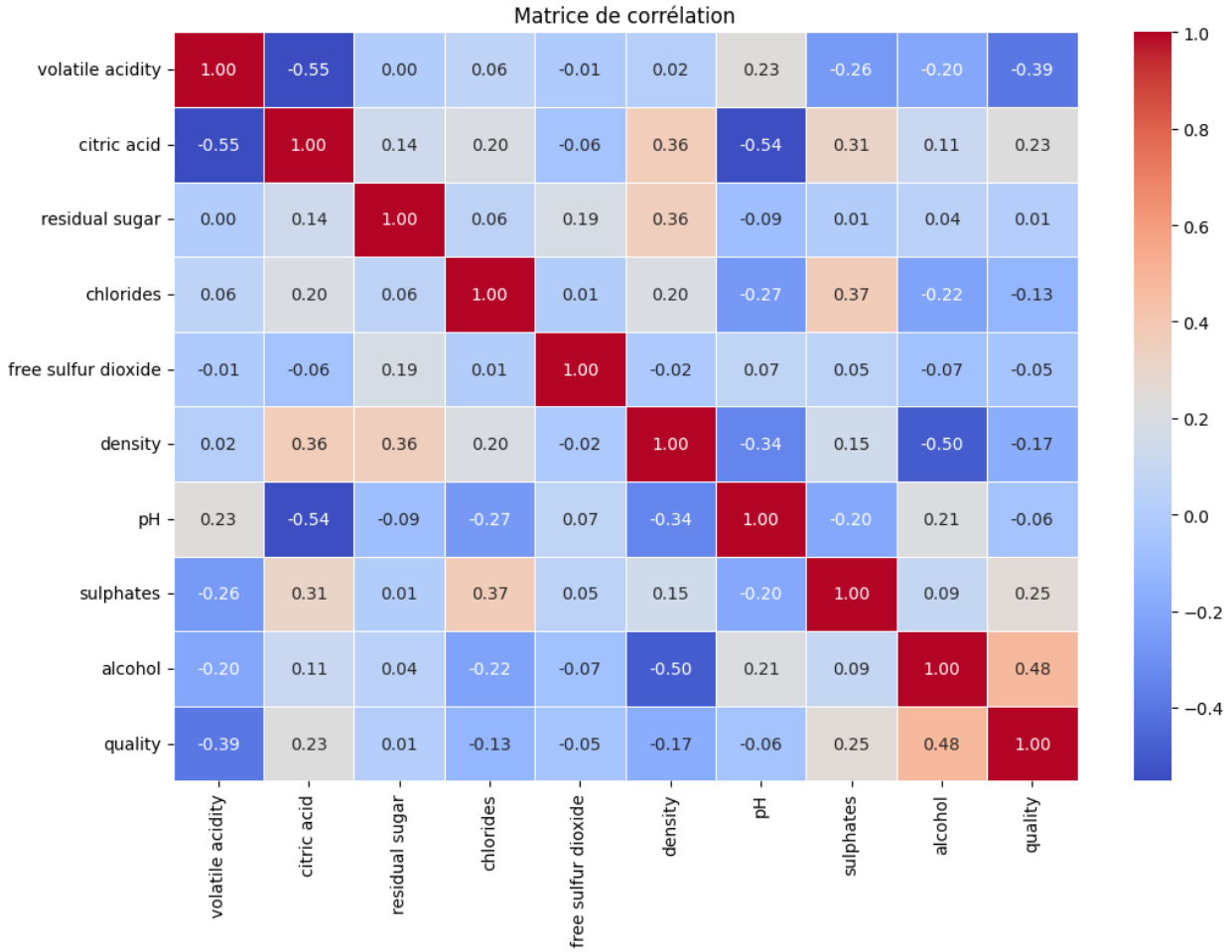
D'après les deux graphes précédents, nous pouvons constater que certains critères sont fortement corrélés. Il est donc judicieux, pour mener à bien la suite de cette analyse, de supprimer la colonne "fixed acidity", qui est fortement corrélée avec les critères suivants : "pH", "density" et "citric acid". De même, nous supprimons la colonne "total sulfur dioxide", qui est redondante avec le critère "free sulfur dioxide".

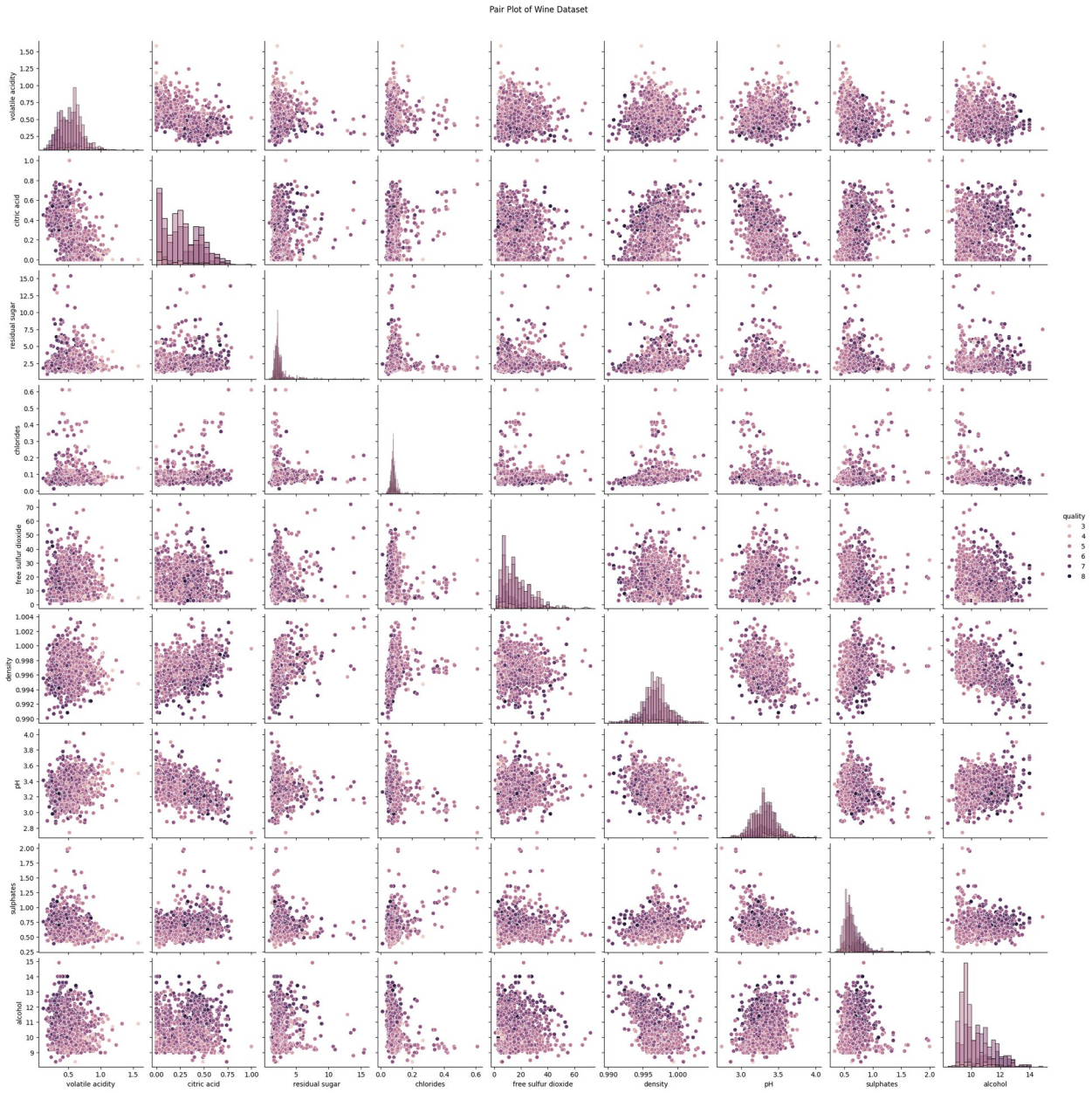
Nous répétons donc l'opération, cette fois-ci en supprimant les deux colonnes citées précédemment, pour s'assurer que cette manipulation est suffisante.

```
#suppression des deux colonnes
data_wine = data_wine.drop(columns=['fixed acidity', 'total sulfur
dioxide']);
```

```
# Créer et afficher une matrice de corrélation sous forme de heatmap
correlation_matrix = data_wine.corr()
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm",
fmt=".2f", linewidths=.5)
plt.title("Matrice de corrélation")
plt.show()

# Créer et afficher un pair plot coloré par la qualité
sns.pairplot(data_wine, hue='quality', diag_kind='hist')
plt.suptitle("Pair Plot of Wine Dataset", y=1.02)
plt.show()
```





Voilà qui est beaucoup mieux : il n'y a plus de valeurs supérieures à 0.6 (ou inférieures à -0.6). De plus, il n'y a plus de réelle relation linéaire directe entre les colonnes restantes deux à deux.

Maintenant, il est indispensable de diviser les données en ensembles d'entraînement et de test. Les données d'entraînement servent à entraîner les différents modèles, tandis que le jeu de test sert à mesurer les performances des modèles précédemment entraînés.

Utilisons la fonction train_test_split de scikit-learn pour diviser notre jeu de données en deux parties :

- X_train : ensemble d'entraînement pour les caractéristiques (features)
- X_test : ensemble de test pour les caractéristiques (features)
- y_train : ensemble d'entraînement pour la variable cible (label)

- y_test : ensemble de test pour la variable cible (label)
- La colonne "quality" est exclue de X et utilisée comme variable cible y.

Paramètres :

- test_size = 0.2 : spécifie que 20% des données seront utilisées comme ensemble de test, tandis que 80% seront utilisées comme ensemble d'entraînement.
- random_state = 42 : garantit que la séparation est reproductible en reprenant la même valeur.

Sans standardisation, les variables avec des échelles différentes pourraient fausser l'apprentissage du modèle, en attribuant involontairement plus de poids à certaines caractéristiques simplement en raison de leur magnitude. Ainsi, nous standardiserons afin de garantir une modélisation plus robuste et des prédictions plus fiables.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
X = data_wine.drop("quality", axis=1)
y = data_wine["quality"]
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardisez les fonctionnalités à l'aide de StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

II - Entraînement et comparaison de modèles de regession

I - Entraîner un modèle RandomForestRegressor

Nous utilisons ici l'algorithme RandomForestRegressor de scikit-learn pour entraîner notre modèle. RandomForestRegressor est un modèle d'ensemble basé sur des arbres de décision.

Pour déterminer les paramètres optimaux, nous utilisons une recherche sur grille (grid_search) afin d'obtenir la meilleure combinaison possible.

```
from sklearn.model_selection import GridSearchCV
# Définir les hyperparamètres à tester dans la grille
param_grid = {
    'n_estimators': [20, 50, 100, 200, 300],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

```
# Initialiser le modèle RandomForestRegressor
rf = RandomForestRegressor()

# Créer l'objet GridSearchCV
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5,
n_jobs=-1, verbose=2)

# Lancer la recherche de grille sur les données
grid_search.fit(X, y)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres:", grid_search.best_params_)
```

```
Fitting 5 folds for each of 135 candidates, totalling 675 fits
Meilleurs paramètres: {'max_depth': None, 'min_samples_leaf': 4,
'min_samples_split': 10, 'n_estimators': 300}
```

Maintenant que la recherche sur grille a identifié la meilleure combinaison parmi les possibilités qui lui ont été fournies, nous pouvons procéder à l'entraînement de la forêt aléatoire avec ces paramètres optimisés. En utilisant ces paramètres optimisés, nous nous attendons à ce que notre modèle RandomForestRegressor soit mieux ajusté aux données et produise des prédictions plus précises.

```
# Créer une instance du modèle RandomForestRegressor
rf_model = RandomForestRegressor(**grid_search.best_params_)

# Entraîner le modèle sur les données d'entraînement
rf_model.fit(X_train_scaled, y_train)
```

```
RandomForestRegressor(min_samples_leaf=4, min_samples_split=10,
                      n_estimators=300)
```

Évaluation du modèle RandomForestRegressor

Nous utilisons différentes métriques pour évaluer les performances de notre modèle RandomForestRegressor. Les métriques incluses sont :

- Root Mean Square Error (RMSE)
- Mean Square Error (MSE)
- Mean Absolute Error (MAE)
- R-squared (R2)
- Précision (Accuracy)

Ces métriques nous donnent une indication de la qualité de prédiction du modèle sur l'ensemble de test.

```
# Importer les bibliothèques pour les métriques d'évaluation
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score
```

```
import numpy as np

# Faire des prédictions sur l'ensemble de test
y_pred = rf_model.predict(X_test_scaled)

# Calculer les métriques d'évaluation
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Calculer la précision du modèle
accuracy = rf_model.score(X_test_scaled, y_test)

# Afficher les métriques d'évaluation
print("Root Mean Square Error (RMSE):", rmse)
print("Mean Square Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("R-squared (R2):", r2)

# Afficher la précision du modèle
print("Accuracy:", accuracy)
```

```
Root Mean Square Error (RMSE): 0.5758217730732651
Mean Square Error (MSE): 0.33157071434523877
Mean Absolute Error (MAE): 0.4532716833930818
R-squared (R2): 0.4926277865934571
Accuracy: 0.4926277865934571
```

II - Entraîner un modèle LinearRegression

Testons un second modele de regression afin de les comparer

```
from sklearn.linear_model import LinearRegression

# Define and train the linear regression model
lr_model = LinearRegression()
lr_model.fit(X_train_scaled, y_train)
```

```
LinearRegression()
```

Évaluation du modèle LinearRegression

Nous utilisons les mêmes métriques que précedemment afin de les comparer dans un second temps. Les métriques incluses sont :

- Root Mean Square Error (RMSE)
- Mean Square Error (MSE)
- Mean Absolute Error (MAE)

- R-squared (R2)
- Précision (Accuracy)

```
# Faire des prédictions sur l'ensemble de test
y_pred = lr_model.predict(X_test_scaled)

# Calculer les métriques d'évaluation
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Calculer la précision du modèle
accuracy = lr_model.score(X_test_scaled, y_test)

# Afficher les métriques d'évaluation
print("Root Mean Square Error (RMSE):", rmse)
print("Mean Square Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("R-squared (R2):", r2)

# Afficher la précision du modèle
print("Accuracy:", accuracy)
```

```
Root Mean Square Error (RMSE): 0.6245843903311801
Mean Square Error (MSE): 0.3901056606453719
Mean Absolute Error (MAE): 0.5111287339346319
R-squared (R2): 0.403057133996532
Accuracy: 0.403057133996532
```

III - Comparaison des 2 modèles

Il est clair que le modèle de forêt aléatoire surpasse la régression linéaire dans toutes les mesures de performance. Il présente des erreurs plus faibles, indiquant une meilleure capacité à prédire les valeurs cibles. De plus, le coefficient de détermination (R^2) est plus élevé pour le modèle de forêt aléatoire, ce qui signifie qu'il explique une plus grande partie de la variance des données. Enfin, la précision du modèle de forêt aléatoire est également supérieure à celle de la régression linéaire. En résumé, le modèle de forêt aléatoire semble être le meilleur choix pour ce problème de régression.

III - Entraîner un réseaux de neurones

Nous utilisons à nouveau la bibliothèque scikit-learn pour créer cette fois un classificateur Multi-Layer Perceptron (MLP) en utilisant la méthode GridSearchCV pour rechercher les meilleurs hyperparamètres pour le modèle parmi ceux que nous auront fournit.

```
from sklearn.neural_network import MLPClassifier
```

```
# Créer un classificateur MLP
mlp = MLPClassifier()

# Définir les hyperparamètres à rechercher
param_grid = {
    'hidden_layer_sizes': [(50,), (100,), (50, 50), (100, 100)],
    'activation': ['relu', 'tanh', 'logistic'],
    'solver': ['sgd', 'adam'],
    'learning_rate': ['constant', 'adaptive'],
}

# Créer un objet GridSearchCV
grid_search = GridSearchCV(mlp, param_grid, cv=5, n_jobs=-1)

# Exécuter la recherche sur la grille
grid_search.fit(X_train_scaled, y_train)

# Afficher les meilleurs paramètres
print("Meilleurs paramètres trouvés:")
print(grid_search.best_params_)

# Évaluer le modèle sur l'ensemble de test
accuracy = grid_search.best_estimator_.score(X_test_scaled, y_test)
print("Précision sur l'ensemble de test:", accuracy)
```

```
Meilleurs paramètres trouvés:
{'activation': 'relu', 'hidden_layer_sizes': (100, 100),
'learning_rate': 'constant', 'solver': 'adam'}
Précision sur l'ensemble de test: 0.628125

/usr/local/lib/python3.10/dist-packages/sklearn/neural_network/
_multilayer_perceptron.py:686: ConvergenceWarning: Stochastic
Optimizer: Maximum iterations (200) reached and the optimization
hasn't converged yet.
  warnings.warn(
```

Les meilleurs paramètres trouvés pour le modèle MLP (Multi-Layer Perceptron) sont les suivants :

- Activation : 'relu'
- Tailles des couches cachées : (100, 100)
- Taux d'apprentissage : 'adaptive'
- Solveur : 'adam'

Interprétation des paramètres :

- Activation 'relu' : Cela signifie que la fonction d'activation de la couche cachée est Rectified Linear Unit (ReLU), qui est une fonction non linéaire qui permet au réseau de mieux capturer les relations complexes dans les données.

- Tailles des couches cachées (100, 100) : Le modèle a deux couches cachées, chacune avec 100 neurones. Cela signifie que le modèle est relativement complexe, avec une capacité suffisante pour apprendre des modèles complexes dans les données.
- Taux d'apprentissage 'adaptive' : Cela signifie que le taux d'apprentissage est adaptatif, ce qui permet au modèle d'ajuster automatiquement le taux d'apprentissage au fur et à mesure de l'entraînement.
- Solveur 'adam' : Adam est un algorithme d'optimisation qui adapte le taux d'apprentissage pour chaque paramètre du modèle individuellement, ce qui permet une convergence plus rapide pendant l'entraînement.

La précision sur l'ensemble de test est de 0.628125. Cela signifie que le modèle classificateur MLP atteint une précision de 62.81% lorsqu'il est évalué sur l'ensemble de test. Cela peut être interprété comme la proportion d'observations correctement classées par le modèle sur l'ensemble de test.