```
# Script:
 #   des algorithmes Random Forest (RF) Gradient Boosting (GBM),
 #   extrem Gradient Boosting (XGB) et Deep learning (DL) Sous Scikit-learn

 # Description:
 #Ce script est dédié au développement des algorithme RF, GBM, XGB et DL,
 #le développement de ces algorithme suit étapes suivant:
 #  1- Importation des bibliothéques et les données;
 #  2- Création des fonctions d'évaluation et d'affichage des variables Important;
 #  3- Développement des 3 algorithmes avec des paramètres Par défaut:
 #      3-1). RF avec son évaluation et l'affichage de ses variables important;
 #      3-2). GBM avec son évaluation et l'affichage de ses variables important;
 #      3-3). XGB avec son évaluation et l'affichage de ses variables important;
 #      3-4). DL avec son évaluation.
 #  4- Optimisation des hyperparamètres avec Random Search et Grid search;
 #  5- Répétition de l'étape 3 mais avec les paramètres séléctionné au l'étape 4.

 # Version:
 #     Mohammed AMEKSA:Juin 2019Script Original
```

1.1 Importer les bibliothèques et les données

```
# importer les bibliothèques
 import numpy as np
 import pandas as pd
 # Importer le package de l'algorithme random forest
 from sklearn.ensemble import RandomForestRegressor
 # Importer le package de l'algorithme gradient boosting
 from sklearn.ensemble import GradientBoostingRegressor
 # Importer le package de l'algorithme xgboost
 import xgboost as XGBRegressor
 # # Importer le package de l'algorithme d'apprentissage profond
 from sklearn import neural_network
 # importer le package de standardisation pour deep learning
 from sklearn.preprocessing import StandardScaler
 #Pour l'évaluation
 from sklearn import metrics
 #Optimisation des hyperparamètres du modèle
 from sklearn.model_selection import GridSearchCV
 from sklearn.model_selection import RandomizedSearchCV

# Importer les deux fichiers de données
 dataset_train = pd.read_csv('Train-Equil-Lon-Lat-Hour-Month-RedVisi.csv')
 dataset_test = pd.read_csv('Test-Equil-Lon-Lat-Hour-Month-RedVisi.csv')

 #Détérminer pour chaque fichie les variables independants et le target
 #fichier d'entrainement
 X_train = dataset_train.iloc[:, 0:-1].values
```

```
y_train = dataset_train.iloc[:, -1].values
#fichier de test
X_test = dataset_test.iloc[:, 0:-1].values
y_test = dataset_test.iloc[:, -1].values
```

1.2 Création des fonctions d'évaluation

```
# évaluation du modèle en calculant les 6 indices de pérformance
 # coefficient de correlation, biais, variance, MAE, MSE et RMSE
 def evaluate(model):
     #prédire les données de test
     y_pred = model.predict(X_test)
     #évaluer le modèle
     #Calculer le coefficient de correlation
     cc=np.corrcoef(y_test, y_pred)[0, 1]
     # Calculer le bias
     bias=np.mean(y_pred)-np.mean(y_test)
     # Calculer la variance
     var = metrics.explained_variance_score(y_test,y_pred)
     # Calculer le MAE (mean absolute error)
     mea = metrics.mean_absolute_error(y_test, y_pred)
     # Calculer MSE (mean squared error)
     mse = metrics.mean_squared_error(y_test, y_pred)
     # Calculer RMSE (root mean squared error)
     rmse = np.sqrt(mse)
     #Afficher les résultats
     print('Coefficient de Correlation: %.2f'%cc)
     print('Biais: %.2f'%bias)
     print('Variance: %.2f'%var)
     print('MAE: %.5f' %mea)
     print('MSE: %.5f' %mse)
     print('RMSE: %.5f' %rmse)

# traçer la courbe des variable importantes et afficher les 5 les plus importants
 def importante_features(model):
     imp=model.feature_importances_
     df_imp=pd.DataFrame(columns=['variables','% d\'importance'])
     d={}
     for i in range(0, len(imp)):
  df_imp=df_imp.append({'variables':dataset_train.columns[i],
   '% d\'importance':imp[i]},ignore_index=True)
     df_imp_sort=df_imp.sort_values(by='% d\'importance', ascending=True,
 na_position='first')
     df_imp_sort.set_index('variables').plot( kind='barh',figsize=(10,10))
     print(df_imp_sort.tail(5))
```

1.3 1- Méthodes Ensemblistes

1.4 Random Forest

1.4.1 Par défaut

```
# Création d'un instance de random forest
 rf_defaut = RandomForestRegressor(random_state=42)
 # Ajuster le modèle sur les données d'entrainement
 rf_defaut.fit(X_train, y_train)

# évaluation du modèle Random Forest avec les paramètres par défauts
 evaluate(rf_defaut)
 # Afficher les variables importants du modèle Random Forest
 # avec les paramètres par défauts
 importante_features(rf_defaut)
```

1.4.2 Optimisation des hyperparamètres avec Grid Search et Random Search

les hyperparamètres les plus importants choisis pour Random forest sont:
* n_estimators = nombre des arbres
* max_features = nombre maximal d'entités prises en compte pour fractionner un nœud
* max_depth = nombre maximum de niveaux dans chaque arbre de décision
*min_samples_split = Nombre minimal d'échantillons requis pour scinder un nœud
* min_samples_leaf = nombre min de points de données autorisés pour une feuille
* bootstrap = méthode d'échantillonnage des points de données (avec ou sans remplacement)

```
# parameters
 #  nombre des arbres pour random forest
 n_estimators_rf = [int(x) for x in np.linspace(start = 60, stop = 80, num = 5)]
 # Nombre d'observations à prendre en compte à chaque scission n/3 pour la régression
 features = dataset_train.columns[:-1]
 max_features_rf = ['auto', 'sqrt', 'log2', int(len(features)/3)]
 # Nombre maximum de niveaux dans une arbre
 # dans cette étude nous avons choisie une plage entre 1 et n/2
 # avec n nombre d'observation
 max_depth_rf = [int(x) for x in np.linspace(1, 17, num = 4)]
 # Méthode d'échentillonage des points
 bootstrap_rf = [True, False]
 # Nombre minimal d'échantillons requis pour scinder un nœud
 min_samples_split_rf = list(range(2,5))
 # nombre min de points de données autorisés pour une feuille
 min_samples_leaf_rf= list(range(1,5))
```

Random Search

```
# Création de dictionnaire (random grid)
 rf_random_grid = {'n_estimators': n_estimators_rf,
'max_features': max_features_rf,
'max_depth': max_depth_rf,
```

```
'bootstrap':bootstrap_rf,
'min_samples_split': min_samples_split_rf,
'min_samples_leaf': min_samples_leaf_rf,
    }
 # Utiliser ce dictionaire (random grid) pour chercher la combinaison
 # des hyperparamètres optimale
 rf_random_search = RandomizedSearchCV(estimator = RandomForestRegressor(),
    param_distributions = rf_random_grid,
    n_iter = 10,
    cv = 3,
    verbose=2,
    scoring='neg_mean_squared_error')
 # Ajuster le modèle sur les données d'entrainement
 rf_random_search.fit(X_train, y_train)

# évaluation du modèle Random Forest avec les paramètres choisis
 # par random search
 evaluate(rf_random_search)
 # Afficher les variables importants du modèle Random Forest
 # avec les paramètres choisis par random search
 importante_features(rf_random_search)
```

Grid Search

Vue que Grid Search ça prend du temps nous avons éssayer de faire l'optimisation comme suit:
* premièrement nous avons optimiser (n_estimators) et (max_features)
* utiliser les valeurs optimales de n_estimators et max_features pour optimiser min_samples_split et min_samples_leaf
* Utiliser les quatres précidents pour optimiser le reste des paramètres.

```
# Optimiser n_estimators and max_features, puis utiliser les valeur optimales
 # choisis pour optimiser les autres
 param_grid_rf = [
 {'n_estimators': n_estimators_rf,
  'max_features': max_features_rf,
 }
 ]
 # Création du modèle
 grid_search_forest = GridSearchCV(RandomForestRegressor(),
param_grid_rf,
cv=3,
scoring='neg_mean_squared_error',
 #pour que les résultats soit toujours les mêmes
random_state = 42,
      )
 # Ajuster le modèle sur les données d'entrainement
 grid_search_forest.fit(X_train, y_train)
 # Afficher les mielleurs hyperparamètres choisis
 print(grid_search_forest.best_params_)
```

```
 # évaluation du modèle Random Forest avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_forest)
 # Afficher les variables importants du modèle Random Forest avec
 # les paramètres choisis par grid search
 importante_features(grid_search_forest)

# Optimiser min_samples_split et min_samples_leaf, puis utiliser
 # les valeur optimales choisis pour optimiser les autres
 param_grid_rf = [
 {'min_samples_split': min_samples_split_rf,
  'min_samples_leaf':min_samples_leaf_rf,
 }
 ]
 # Création du modèle
 grid_search_forest = GridSearchCV(RandomForestRegressor(
    # ici nous spécifions les valeurs choisis
    #pour n_estimators et max_feautures
    # notre cas les valeurs suivants sont les optimales
      n_estimators= 75 ,
      max_features=11
  ),
param_grid_rf,
cv=3,
scoring='neg_mean_squared_error',
    #pour que les résultats soit toujours les mêmes
random_state = 42,
      )
 # Ajuster le modèle sur les données d'entrainement
 grid_search_forest.fit(X_train, y_train)
 # Afficher les mielleurs hyperparamètres choisis
 print(grid_search_forest.best_params_)
 # évaluation du modèle Random Forest avec les paramètres
 # choisis par Grid search
 evaluate(grid_search_forest)
 # Afficher les variables importants du modèle Random Forest
 # avec les paramètres choisis par grid search
 importante_features(grid_search_forest)

# De même nous avons essayer d'optimiser max_depth et bootstrap,
 # puis utiliser les valeur optimales choisis pour optimiser les autres
 param_grid_rf = [
 {
  'max_depth': max_depth_rf,
  'bootstrap': bootstrap_rf,
 }
 ]
 # Création du modèle
```

```
 grid_search_forest = GridSearchCV(RandomForestRegressor(
# ici nous spécifions les valeurs choisis
# pour n_estimators et max_feautures
# notre cas les valeurs suivants sont les optimales
      n_estimators= 75 ,
      max_features=11,
      min_samples_split=2,
      min_samples_leaf=2,
  ),
param_grid_rf,
cv=3,
scoring='neg_mean_squared_error',
#pour que les résultats soit toujours les mêmes
random_state = 42,
      )
 # Ajuster le modèle sur les données d'entrainement
 grid_search_forest.fit(X_train, y_train)
 # Afficher les mielleurs hyperparamètres choisis
 print(grid_search_forest.best_params_)
 # évaluation du modèle Random Forest avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_forest)
 # Afficher les variables importants du modèle Random Forest avec
 # les paramètres choisis par grid search
 importante_features(grid_search_forest)
```

1.5 Gradient Boosting Machine

1.5.1 Par défaut

```
# Création d'un instance de gradient boosting machine
 gbm_defaut = GradientBoostingRegressor(random_state=42)
 # Ajuster le modèle sur les données d'entrainement
 gbm_defaut.fit(X_train, y_train)

# évaluation du modèle gradient boosting machine avec les paramètres
 # par défauts
 evaluate(gbm_defaut)
 # Afficher les variables importants du modèle gradient boosting machine
 # avec les paramètres par défauts
 importante_features(gbm_defaut)
```

1.5.2 Optimisation des hyperparamètres avec Grid Search et Random Search

les hyperparamètres les plus importants choisis pour Gradient Boosting Machine sont:

Les paramètres généraux peuvent être divisés en 3 catégories:

* Paramètres spécifiques aux arbres: ils affectent chaque arbre individuel du modèle.
* Paramètres d'amplification: ils affectent l'opération d'amplification dans le modèle.

* Paramètres divers: Autres paramètres pour le fonctionnement général.

Les paramètres utilisés pour définir un arbre
* min_samples_split: nombre minimal d'échantillons (ou d'observations) nécessaires dans un nœud à prendre en compte pour la scission.
* min_samples_leaf: minimum d'échantillons (ou d'observations) requis dans un nœud terminal ou une feuille
* max_depth: profondeur maximale d'un arbre.
* max_feature: nombre d'observations à prendre en compte lors de la recherche du meilleur split.

paramètres de gestion du boosting:
* n_estimators: nombre d'arbres séquentiels à modéliser
* learning_rate: détermine l'impact de chaque arbre sur le résultat final

```
# Les intérvale des paramètres
 # Nombre d'arbres
 n_estimators_gbm = [int(x) for x in np.linspace(start = 100, stop = 200, num = 5)]
 # nombre des observations à considirer pour un split
 #sqrt(n) pour classification et n/3 for régression
 features = dataset_train.columns[:-1]
 max_features_gbm = ['auto', 'sqrt', 'log2', int(len(features)/3)]
 # profondeur maximale d'un arbre
 max_depth_gbm = [int(x) for x in np.linspace(1, 15, num = 4)]
 # nombre minimal d'échantillons (ou d'observations) nécessaires
 # dans un nœud à prendre en compte pour la scission
 min_samples_split_gbm = list(range(2,5))
 # Taux d'apprentissage
 learning_rate_gbm = [0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2]
 min_samples_leaf_gbm= list(range(1,5))
```

Random Search

```
# Création de dictionnaire (random grid)
 GBM_random_grid = {'max_features': max_features_gbm,
      'max_depth': max_depth_gbm,
      'min_samples_split': min_samples_split_gbm,
      'min_samples_leaf':min_samples_leaf_gbm,

      'n_estimators': n_estimators_gbm,
      'learning_rate': learning_rate_gbm,
     }

 # Utiliser ce dictionaire (random grid) pour chercher la combinaison
 # des hyperparamètres optimale
 random_search_GBM = RandomizedSearchCV(estimator = GradientBoostingRegressor(),
    param_distributions = GBM_random_grid,
    n_iter = 5,
    cv = 3,
```

```
    verbose=2,
     random_state = 42,
    scoring='neg_mean_squared_error')
 # Ajuster le modèle sur les données d'apprentissage
 random_search_GBM.fit(X_train, y_train)

# évaluation du modèle Gradient Boosting Machine avec les paramètres choisis
 # par random search
 evaluate(random_search_GBM)
 # Afficher les variables importants du modèle Gradient Boosting Machine avec
 # les paramètres choisis par random search
 importante_features(random_search_GBM)
```

Grid Search

```
param_grid_gbm = [
     {'max_features': max_features_gbm,}
 ]
 # Création du modèle
 grid_search_GBM = GridSearchCV(GradientBoostingRegressor(),
     param_grid_gbm,
     cv=3,
     random_state = 42,
     scoring='neg_mean_squared_error')
 # Ajustement du modèle
 grid_search_GBM.fit(X_train, y_train)
 # évaluation du modèle Gradient Boosting Machine avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_GBM)
 # Afficher les variables importants du modèle Gradient Boosting Machine avec
 # les paramètres choisis par Grid search
 importante_features(grid_search_GBM)

param_grid_gbm = [
     {'max_depth': max_depth_gbm,}
 ]
 # Création du modèle avec les paramètres séléctionné précédement
 grid_search_GBM = GridSearchCV(GradientBoostingRegressor(max_features='auto'),
     param_grid_gbm,
     cv=3,
     random_state = 42,
     scoring='neg_mean_squared_error')

 grid_search_GBM.fit(X_train, y_train)

 # Ajustement du modèle
 grid_search_GBM.fit(X_train, y_train)
 # évaluation du modèle Gradient Boosting Machine avec les paramètres choisis
 # par Grid search
```

```
 evaluate(grid_search_GBM)
 # Afficher les variables importants du modèle Gradient Boosting Machine avec
 # les paramètres choisis par Grid search
 importante_features(grid_search_GBM)

param_grid_gbm = [
     {'min_samples_split': min_samples_split_gbm,}
 ]
 # Création du modèle avec les paramètres séléctionné précédement
 grid_search_GBM = GridSearchCV(GradientBoostingRegressor(max_features ='auto' ,
  max_depth =10 ),
     param_grid_gbm,
     cv=3,
     scoring='neg_mean_squared_error')
 # Ajustement du modèle
 grid_search_GBM.fit(X_train, y_train)
 # évaluation du modèle Gradient Boosting Machine avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_GBM)
 # Afficher les variables importants du modèle Gradient Boosting Machine avec
 # les paramètres choisis par Grid search
 importante_features(grid_search_GBM)

param_grid_gbm = [
     {'min_samples_leaf':min_samples_leaf_gbm,}
 ]
 # Création du modèle avec les paramètres séléctionné précédement
 grid_search_GBM = GridSearchCV(GradientBoostingRegressor(max_features= 'auto',
  max_depth= 10,
  min_samples_split = 3, ),
     param_grid_gbm,
     cv=3,
     scoring='neg_mean_squared_error')
 # Ajustement du modèle
 grid_search_GBM.fit(X_train, y_train)
 # évaluation du modèle Gradient Boosting Machine avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_GBM)
 # Afficher les variables importants du modèle Gradient Boosting Machine avec
 # les paramètres choisis par Grid search
 importante_features(grid_search_GBM)

param_grid_gbm = [
     {'n_estimators': n_estimators_gbm }
 ]
 # Création du modèle avec les paramètres séléctionné précédement
 grid_search_GBM = GridSearchCV(GradientBoostingRegressor(max_features='auto' ,
  max_depth = 10,
```

```
 min_samples_split =3 ,
 min_samples_leaf = 1 ,
),
    param_grid_gbm,
    cv=3,
    scoring='neg_mean_squared_error')
# Ajustement du modèle
grid_search_GBM.fit(X_train, y_train)
# évaluation du modèle Gradient Boosting Machine avec les paramètres choisis
# par Grid search
evaluate(grid_search_GBM)
# Afficher les variables importants du modèle Gradient Boosting Machine avec
# les paramètres choisis par Grid search
importante_features(grid_search_GBM)

param_grid_gbm = [
    {'learning_rate': learning_rate_gbm,}
]
# Création du modèle avec les paramètres séléctionné précédement
grid_search_GBM = GridSearchCV(GradientBoostingRegressor(max_features='auto',
 max_depth = 10,
 min_samples_split = 3,
 min_samples_leaf = 1,
 n_estimators = 200,
),
    param_grid_gbm,
    cv=3,
    scoring='neg_mean_squared_error')
# Ajustement du modèle
grid_search_GBM.fit(X_train, y_train)
# évaluation du modèle Gradient Boosting Machine avec les paramètres choisis
# par Grid search
evaluate(grid_search_GBM)
# Afficher les variables importants du modèle Gradient Boosting Machine avec
# les paramètres choisis par Grid search
importante_features(grid_search_GBM)
```

1.6 eXtrem Gradient Boosting

1.6.1 Par défaut

```
# Création d'un instance d'eXtrem Gradient Boosting
xgb_defaut = XGBRegressor(random_state=42)
# Ajuster le modèle sur les données d'entrainement
xgb_defaut.fit(X_train, y_train)

# évaluation du modèle eXtrem Gradient Boosting avec les paramètres
# par défauts
evaluate(xgb_defaut)
```

```
# Afficher les variables importants du modèle eXtrem Gradient Boosting
# avec les paramètres par défauts
importante_features(xgb_defaut)
```

1.6.2 Optimisation des hyperparamètres avec Grid Search et Random Search

les hyperparamètres les plus importants choisis pour Gradient Boosting Machine sont:

```
# Les intérvalles de variations des hyperparamètres choisis pour XGB
# Le taux d'apprentissage souvent entre 0.1 et 0.01
# or on peut essayer avec 0.15 0.2
learning_rate_xgb = [0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2]
# nombre d'arbres, généralemnet 100 si la taille des données est elevé
n_estimators_xgb =  [int(x) for x in np.linspace(start = 100, stop = 200, num = 11)]
# profondeur des arbres utilisé dans le modèle
max_depth_xgb = [int(x) for x in np.linspace(10, 15, num = 5)]
# % des ligne séléctionés pour construire chaque arbre,
# généralement prend des valeurs entre 0.8 et 1
subsample_xgb = [i/100.0 for i in range(80,101,5)]
# nombre de colonne utilisé par chaque arbre,
# généralement prend des valeurs entre 0.3 et 0.8
# pour des problèmes où y a un grand nombre des colonnes
# et entre 0.8 et 1 pour des problèmes avec un peu de colonnes
colsample_bytree_xgb = [i/100.0 for i in range(80,101,5)]
# il agit comme un paramètre de régularisation. 0, 1 ou 5
gamma_xgb = [i for i in range(0,6)]
```

Random Search

```
# Création de dictionnaire (random grid)
random_grid_XGB = {'max_depth': max_depth_xgb,
     'gamma': gamma_xgb,
     'colsample_bytree': colsample_bytree_xgb,
     'learning_rate': learning_rate_xgb,
     'n_estimators': n_estimators_xgb,
    }
# Utiliser ce dictionaire (random grid) pour chercher la combinaison
# des hyperparamètres optimales
random_search_XGB = RandomizedSearchCV(estimator = XGBRegressor(),
   param_distributions = random_grid_XGB,
   n_iter = 10,
   cv = 3,
   verbose=2,
   random_state = 42,
   scoring='neg_mean_squared_error')
# Ajuster le modèle sur les données d'apprentissage
random_search_XGB.fit(X_train, y_train)
```

```
# évaluation du modèle eXtrem Gradient Boosting avec les paramètres choisis
 # par random search
 evaluate(random_search_XGB)
 # Afficher les variables importants du modèle eXtrem Gradient Boosting
 # avec les paramètres choisis par random search
 importante_features(random_search_XGB)
```

Grid Search

```
# Sélectionne n_estimators à optimiser en premier temps
 param_grid_xgb = [
 {'n_estimators': n_estimators_xgb,}
 ]
 # Création du modèle
 grid_search_XGB = GridSearchCV(XGBRegressor(),
     param_grid_xgb,
     cv=3,
     random_state = 42,
     scoring='neg_mean_squared_error')
 # Ajustement du modèle
 grid_search_XGB.fit(X_train, y_train)
 # évaluation du modèle eXtrem Gradient Boosting avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_XGB)
 # Afficher les variables importants du modèle eXtrem Gradient Boosting avec
 # les paramètres choisis par Grid search
 importante_features(grid_search_XGB)
```

```
# Sélectionne max_depth à optimiser
 param_grid_xgb = [
 {'max_depth': max_depth_xgb, }
 ]
 # Création du modèle avec le nombre d'arbres séléctionné précédement
 # (dans notre cas =190)
 grid_search_XGB = GridSearchCV(XGBRegressor(n_estimators = 190),
     param_grid_xgb,
     cv=3,
    random_state = 42,
     scoring='neg_mean_squared_error')
 # Ajustement du modèle
 grid_search_XGB.fit(X_train, y_train)
 # évaluation du modèle eXtrem Gradient Boosting avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_XGB)
 # Afficher les variables importants du modèle eXtrem Gradient Boosting avec
 # les paramètres choisis par Grid search
 importante_features(grid_search_XGB)
```

```
# Sélectionne subsample à optimiser
 param_grid_xgb = [
```

```
{'subsample': subsample_xgb,}
]
# Création du modèle avec le nombre d'arbres séléctionné précédement
#(dans notre cas n_estimators=190 et max_depth=11)
grid_search_XGB = GridSearchCV(XGBRegressor(n_estimators =190 ,
  max_depth = 11,
 ),
    param_grid_xgb,
    cv=3,
    random_state = 42,
    scoring='neg_mean_squared_error')
# Ajustement du modèle
grid_search_XGB.fit(X_train, y_train)
# évaluation du modèle eXtrem Gradient Boosting avec les paramètres choisis
# par Grid search
evaluate(grid_search_XGB)
# Afficher les variables importants du modèle eXtrem Gradient Boosting avec
# les paramètres choisis par Grid search
importante_features(grid_search_XGB)

# Sélectionner gamma à optimiser
param_grid_xgb = [
{'gamma': gamma_xgb,}
]
# Création du modèle avec le nombre d'arbres séléctionné précédement
#(dans notre cas n_estimators=190 et max_depth=11 subsample est par défaut)
grid_search_XGB = GridSearchCV(XGBRegressor(n_estimators = 190 ,
  max_depth = 11,
  ),
    param_grid_xgb,
    cv=3,
    random_state = 42,
    scoring='neg_mean_squared_error')
# Ajustement du modèle
grid_search_XGB.fit(X_train, y_train)
# évaluation du modèle eXtrem Gradient Boosting avec les paramètres choisis
# par Grid search
evaluate(grid_search_XGB)
# Afficher les variables importants du modèle eXtrem Gradient Boosting avec
# les paramètres choisis par Grid search
importante_features(grid_search_XGB)

# Sélectionner colsample_bytree à optimiser
param_grid_xgb = [
{'colsample_bytree': colsample_bytree_xgb,}
]
# Création du modèle avec le nombre d'arbres séléctionné précédement
#(dans notre cas n_estimators=190, max_depth=11, subsample est par défaut
```

```
 # et gamma=3)
 grid_search_XGB = GridSearchCV(XGBRegressor(n_estimators = 190,
   max_depth = 11,
   gamma = 3 ,
   ),
     param_grid_xgb,
     cv=3,
     random_state = 42,
     scoring='neg_mean_squared_error')
 # Ajustement du modèle
 grid_search_XGB.fit(X_train, y_train)
 # évaluation du modèle eXtrem Gradient Boosting avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_XGB)
 # Afficher les variables importants du modèle eXtrem Gradient Boosting avec
 # les paramètres choisis par Grid search
 importante_features(grid_search_XGB)

# Sélectionner learning_rate à optimiser
 param_grid_xgb = [
 {'learning_rate': learning_rate_xgb,}
 ]
 # Création du modèle avec le nombre d'arbres séléctionné précédement
 #(dans notre cas n_estimators=190, max_depth=11, subsample et
 # colsample_bytree sont par défaut et gamma=3)
 grid_search_XGB = GridSearchCV(XGBRegressor(n_estimators =190 ,
   max_depth = 11,
   gamma = 3 ,
  ),
     param_grid_xgb,
     random_state = 42,
     cv=3,
     scoring='neg_mean_squared_error')
 # Ajustement du modèle
 grid_search_XGB.fit(X_train, y_train)
 # évaluation du modèle eXtrem Gradient Boosting avec les paramètres choisis
 # par Grid search
 evaluate(grid_search_XGB)
 # Afficher les variables importants du modèle eXtrem Gradient Boosting avec
 # les paramètres choisis par Grid search
 importante_features(grid_search_XGB)
```

1.7 2- Apprentissage profond

```
# Ici, nous avons standardisé les données en faisant appel
 # à un instance de StandardScaler
 # cette instance suit la régle "x-men/std"
 scaler = StandardScaler()
```

```
 scaler.fit(X_train)

 X_train = scaler.transform(X_train)
 X_test = scaler.transform(X_test)

# Création d'un instance de MLPRegressor avec les paramètres par défauts
 MLP_defaut=neural_network.MLPRegressor(random_state = 42)
 # Ajuster le modèle sur les données d'entrainement
 MLP_defaut.fit(X_train, y_train)

# évaluation du modèle MLP avec les paramètres choisis par défauts
 evaluate(MLP_defaut)
```

- Ensuite, nous créons une instance du modèle Multilayer perceptrons (réseau de neurons), nous définirons hidden_layer_sizes. Pour ce paramètre, nous transmettrons une paire composée du nombre de neurones que nous voulons au niveau de chaque couche, la nième entrée du tuple représente le nombre de neurones de la nième couche du modèle MLP.
- pour la simplicité, nous choisirons 2 couches avec le même nombre de neurones (68 nombre d'entrées fois 2) ainsi que 100 itérations maximum.

```
MLP_manuel=neural_network.MLPRegressor(hidden_layer_sizes=(68,68),
     max_iter=100,
     activation='relu',
     random_state = 42,
    )
 MLP_manuel.fit(X_train,y_train)

# évaluation du modèle MLP avec les paramètres choisis manuellement
 evaluate(MLP_manuel)
```