Introduction

Objectif : Prédire une éruption volcanique à moyen/long terme.

Les scientifiques identifient souvent le temps avant une éruption en surveillant les secousses volcaniques à partir des signaux sismiques. Dans certains volcans, ces secousses s'intensifient lorsque les volcans se réveillent et se préparent à entrer en éruption. L'algorithme doit ainsi être en mesure d'identifier les signatures des formes d'ondes sismiques qui caractérisent le développement d'une éruption pour prédire le temps avant la prochaine éruption.

Dans ce contexte, les données proviennent de 10 capteurs placés autour d'un même volcan. Une instance de donnée correspond à plusieurs séquence de données. Au sein de cette séquence de données de dix minutes (6001 valeurs), on retrouve les valeurs détectées par 10 capteurs (sismomètres) placés autour du volcan. À chaque instance de donnée est associée un temps avant éruption exprimé en centième de secondes.

Librairies et Données ¶

`segment_id` : ID d'un segment de données.

`time_to_eruption` : Temps avant la prochaine éruption, qu'on cherche à prédire ici. (en 0.01s).

`[train|test]/*.csv` : Les fichiers de données. Chaque fichier contient dix minutes de logs provenant de dix capteurs différents disposés autour du volcan.

In [1]:

```
import numpy as np
import pandas as pd
from multiprocessing import Pool
import random
import os

from astropy.stats import biweight_location, biweight_scale
from scipy import signal, stats
import scipy

from plotly.subplots import make_subplots
import plotly.graph_objects as go

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.compose import TransformedTargetRegressor
from sklearn import linear_model
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor, GradientBoosting
Regressor, StackingRegressor
from sklearn.model_selection import cross_val_score, GridSearchCV, KFold, train_test_sp
lit, RandomizedSearchCV, RepeatedKFold
from sklearn.metrics import mean_squared_error, r2_score
import xgboost as xgb
from sklearn import svm
from hyperopt import fmin, tpe, hp, STATUS_OK, Trials
```

La ligne ci-dessous permet d'importer les données de la compétition à partir de l'API de Kaggle. L'ensemble des données étant volumineux (30GB), je présenterai une seule instance de donnée lors de l'analyse exploiratoire des données et j'importerai directement les bases d'entraînements et de test sur lesquelles j'ai travaillé à partir du Features Engineering.

In [2]:

```
# !kaggle competitions download -c gan-getting-started
```

In [3]:

```
path_to_cwd = os.getcwd()

train = pd.read_csv(path_to_cwd + '/data/train.csv')
sample_submission = pd.read_csv(path_to_cwd + '/data/sample_submission.csv')
train.head(5)
```

Out[3]:

	segment_id	time_to_eruption
0	1136037770	12262005
1	1969647810	32739612
2	1895879680	14965999
3	2068207140	26469720
4	192955606	31072429

Chaque séquence d'un capteur dure 10 minutes et contient 60001 valeurs. Sur certains segments, certains capteurs ne semblent pas fonctionner. Il manque quelques échantillons pour certains, cela étant probablement lié à des pannes de batterie.

Analyse exploratoire des données

J'ai représenté ci-dessous une instance de donnée (une observation, c'est-à-dire 10 minutes de logs provenant de 10 capteurs différents).

In [4]:

```
sample_segment = pd.read_csv(path_to_cwd + '/data/1000015382.csv')
sample_segment.head()
```

Out[4]:

	sensor_1	sensor_2	sensor_3	sensor_4	sensor_5	sensor_6	sensor_7	sensor_8	senso
0	260.0	64.0	-232.0	-36.0	-2.0	-35.0	103.0	389.0	6
1	233.0	175.0	146.0	160.0	-4.0	29.0	-120.0	498.0	5
2	216.0	236.0	321.0	202.0	2.0	113.0	-230.0	554.0	9
3	156.0	205.0	382.0	6.0	12.0	70.0	-228.0	580.0	14
4	158.0	101.0	272.0	-154.0	16.0	45.0	-162.0	624.0	14

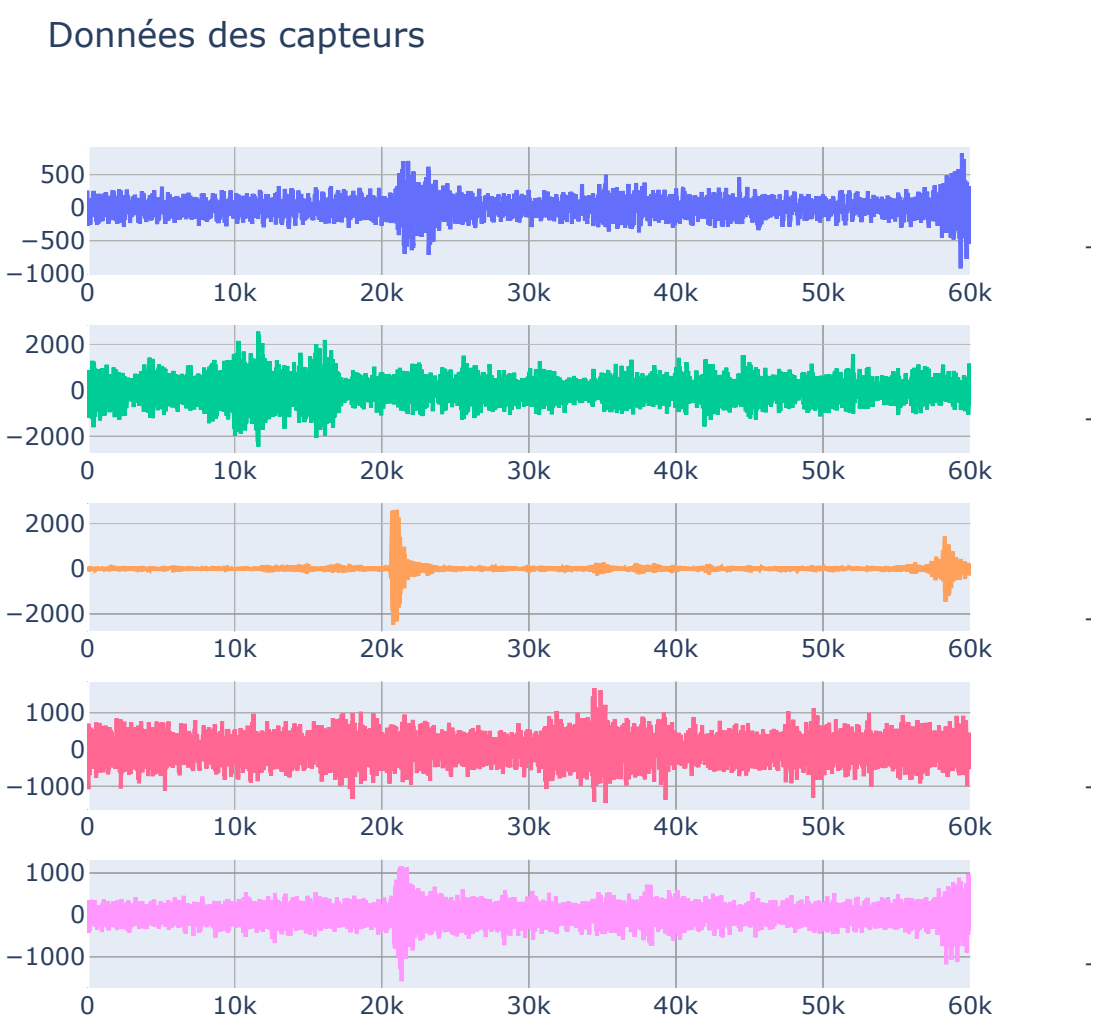
In [5]:

```
fig = make_subplots(rows=5, cols=2)

traces = [go.Scatter(x=sample_segment.index, y=sample_segment[col], name=col) for col i
n sample_segment.columns]

for i in range(0, len(traces)) :
    fig.append_trace(traces[i], (i // 2) + 1, (i % 2) + 1)

fig.update_layout(height=600, width=1200, title_text="Données des capteurs")
fig.show()
```



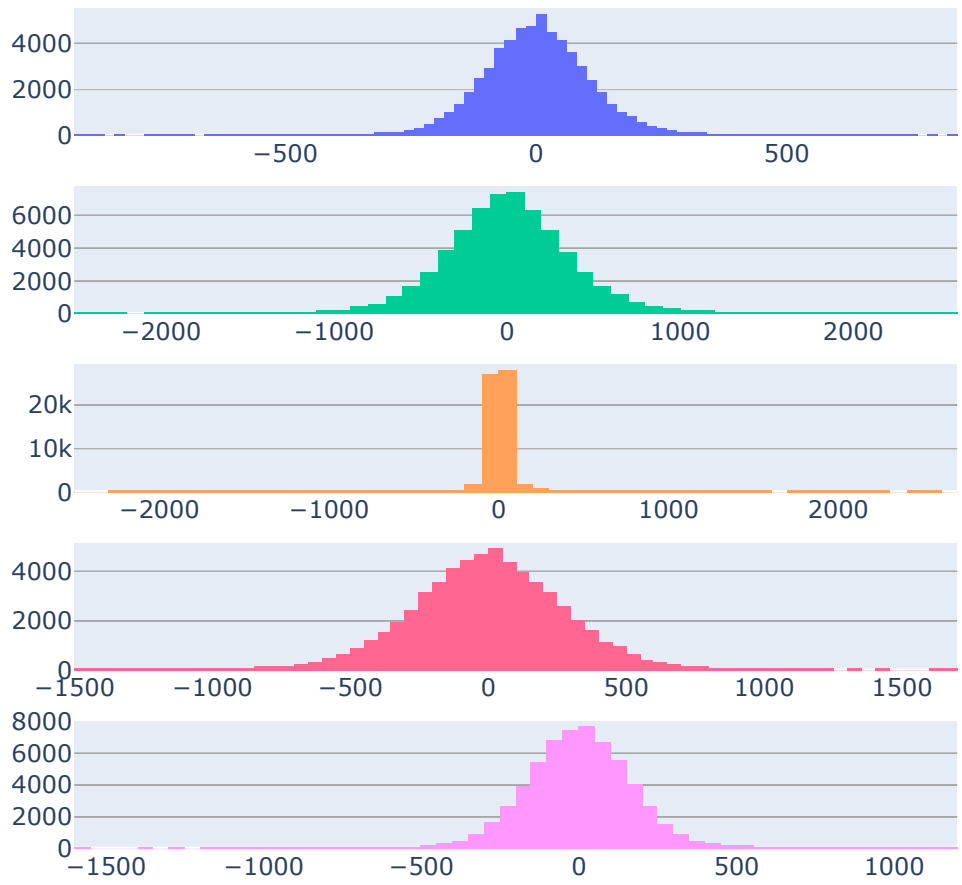
In [6]:

```
fig = make_subplots(rows=5, cols=2)

traces = [go.Histogram(x=sample_segment[col], nbinsx=100, name=col) for col in sample_s
egment.columns]

for i in range(0, len(traces)) :
    fig.append_trace(traces[i], (i // 2) + 1, (i % 2) + 1)

fig.update_layout(height=600, width=1200, title_text="Distribution des capteurs")
fig.show()
```



Il y a parfois du **bruit**, **quelques anomalies** et des **données manquantes**. Le bruit peut être corrigé par filtre médian ou encore par transformée en ondelettes et les anomalies par écart interquartile, z-score ou à l'aide d'intervalle de confiance, par exemple. Lorsqu'il manque une séquence entière de données (le plus souvent), on impute par 0 les données manquantes et par la moyenne sinon. Cela peut être l'objet d'une étude plus approfondie ultérieurement. Toutefois il conviendra d'ajuster au mieux les hyperparamètres de ces méthodes (seuil sur le Z-Score, taille de la fenêtre du filtre médian...) et les données de test peuvent parfois être distribuées différement des données d'entrainement.

Features Engineering

Comme pour une instance on a 10 séquences, on regroupe chaque instance sur une seule ligne à partir d'un ensemble de features décrivant au mieux nos capteurs. Pour cela, j'ai calculé **plusieurs statistiques** décrivant chaque séquence de 10 capteurs ; moyenne, médiane, écart-type, maximum, minimum, skewness, kurtosis, quantile, écart inter-quartile sur le **signal original et quelques modifications** ; filtre médian avec différentes fenêtres, transformée de Fourier, transformée en sinus et cosinus discrète.

Pour accélérer le temps de formation des données, j'ai parallélisé le processus à partir du nombre de coeurs

In [7]:

```
def df_parallelize_run(func, t_split, cores=4):
    num_cores = np.min([cores, len(t_split)])
    pool = Pool(processes=num_cores)
    df = pd.concat(pool.starmap(func, t_split), axis=0)
    pool.close()
    pool.join()

    return df
```

La fonction `get_basic_features` calcule les statistiques pour une instance de données (un fichier).

In [37]:

```
def get_basic_features(df, direct='train'):
    j = 0
    result = pd.DataFrame(dtype=np.float32)
    result['segment_id'] = df['segment_id']
    result['time_to_eruption'] = df['time_to_eruption']
    result.set_index('segment_id', inplace=True)

    for ids in df['segment_id']:
        f = pd.read_csv(f'/kaggle/input/predict-volcanic-eruptions-ingv-oe/{direct}/{id
s}.csv')
        if j%100 == 0 :
            print(j)                                        #progress bar
        for sensor in f.columns:

            s_ = f[sensor].ffill().fillna(0).values         # filling nans
            raw_s = s_                                      # raw signal
            filtered_5 = signal.medfilt(s_, 5)              # median filtered signal wi
th window=5
            filtered_11 = signal.medfilt(s_, 11)            # median filtered signal wi
th window=11
            filtered_33 = signal.medfilt(s_, 33)            # median filtered signal wi
th window=33
            filtered_55 = signal.medfilt(s_, 55)            # median filtered signal wi
th window=55
            cos_s = scipy.fft.dct(s_)                       # direct cosine transformed
signal
            sin_s = scipy.fft.dst(s_)                       # direct sine transformed s
ignal
            abs_s = np.abs(s_)                              # abs signal
            fft = np.fft.fft(s_)                            # fft
            fft_real = np.real(fft)                         # real part
            fft_img = np.imag(fft)                          # imaginary part

            for num, data in enumerate([raw_s, cos_s, sin_s, filtered_5, filtered_11, f
iltered_33, filtered_55, abs_s, fft_real, fft_img]):

                sum_ = np.sum(data)                         # sum
                mean = np.mean(data)                        # mean
                med = np.median(data)                       # median
                std = np.std(data, ddof=0)                  # std
                var_ = np.std(data)                         # var
                min_ = np.min(data)                         # minimum
                max_ = np.max(data)                         # maximum
                mr = (min_ + max_)*0.5                      # midrange
                ptp = max_ - min_                           # range
                q1, q3 = np.quantile(data, q=[0.25, 0.75])  # 1st & 3rd quartile
                mh = (q3 + q1)*0.5                          # midhinge
                iqr = q3 - q1                               # IQR
                mad_1 = np.median(np.abs(data - mean))      # Median deviation from the
mean
                mad_2 = np.mean(np.abs(data - mean))        # Mean deviation from the m
ean
                bwl = biweight_location(data)               # Biweight location
                bws = biweight_scale(data)                  # Biweight scale
                skew = scipy.stats.skew(data)               # skew
                kurtosis = scipy.stats.kurtosis(data)       # kurtosis

                result.loc[ids, f'{sensor}_sum_{num}'] = sum_
                result.loc[ids, f'{sensor}_mean_{num}'] = mean
```

```
                result.loc[ids, f'{sensor}_med_{num}'] = med
                result.loc[ids, f'{sensor}_std_{num}'] = std
                result.loc[ids, f'{sensor}_var_{num}'] = var_
                result.loc[ids, f'{sensor}_midrange_{num}'] = mr
                result.loc[ids, f'{sensor}_midhinge_{num}'] = mh
                result.loc[ids, f'{sensor}_min_{num}'] = min_
                result.loc[ids, f'{sensor}_max_{num}'] = max_
                result.loc[ids, f'{sensor}_ptp_{num}'] = ptp
                result.loc[ids, f'{sensor}_q1_{num}'] = q1
                result.loc[ids, f'{sensor}_q3_{num}'] = q3
                result.loc[ids, f'{sensor}_iqr_{num}'] = iqr
                result.loc[ids, f'{sensor}_med_abs_dev_mean_{num}'] = mad_1
                result.loc[ids, f'{sensor}_mean_abs_dev_mean_{num}'] = mad_2
                result.loc[ids, f'{sensor}_biweight_location_{num}'] = bwl
                result.loc[ids, f'{sensor}_biweight_scale_{num}'] = bws
        j += 1
    return result
```

On sépare ensuite les jeux de données selon le nombre de coeurs de façon à segmenter la formation sur chaque coeur.

In [ ]:

```
N_CORES = 4
chunk_size = train.shape[0] // N_CORES
splits_train = [train[:chunk_size], train[chunk_size:2*chunk_size], train[2*chunk_size:
3*chunk_size], train[3*chunk_size:]]
splits_test = [sample_submission[:chunk_size], sample_submission[chunk_size:2*chunk_siz
e], sample_submission[2*chunk_size:3*chunk_size], sample_submission[3*chunk_size:]]
```

In [ ]:

```
# %%time

# train_df = df_parallelize_run(get_basic_features,
#                               [(chunk, 'train') for chunk in splits_train],
#                               N_CORES)
```

In [ ]:

```
# %%time

# test_df = df_parallelize_run(get_basic_features,
#                              [(chunk, 'test') for chunk in splits_test],
#                              N_CORES)
```

Il a fallu tout de même environ 3 heures pour former les jeux de données d'entraînement et de formation. Pour la suite, j'ai sauvegardé les bases en local que je re-upload à chaque fois.

In [9]:

```
train_df = pd.read_csv(path_to_cwd + '/data/train_df.csv', index_col=0)
test_df  = pd.read_csv(path_to_cwd + '/data/test_df.csv', index_col=0)
train_df
```

Out[9]:

segment_id	time_to_eruption	sensor_1_sum_0	sensor_1_mean_0	sensor_1_med_0	sensor_1
1136037770	12262005	-96621.0	-1.610323	0.0	303.
1969647810	32739612	85569.0	1.426126	0.0	438.
1895879680	14965999	150278.0	2.504592	0.0	241.
2068207140	26469720	129950.0	2.165797	0.0	221.
192955606	31072429	4429.0	0.073815	0.0	261.
...	...	...	...	...	
873340274	15695097	54405.0	0.906735	0.0	613.
1297437712	35659379	476221.0	7.936884	0.0	649.
694853998	31206935	85261.0	1.420993	0.0	110.
1886987043	9598270	54350.0	0.905818	0.0	478.
1100632800	20128938	-248575.0	-4.142848	0.0	272.

4431 rows × 1871 columns

On a donc 1870 variables explicatives. Le fait d'avoir trop de variables explicatives peut mener à **sur-apprendre** et l'algorithme d'apprentissage nécessitera beaucoup **plus de temps** pour se former. Pour réduire la dimension de mes données, j'ai utilisé la **régression Lasso**. La méthode d'analyse en composantes principales fonctionne également très bien. La transformation de la variable cible en racine permet d'obtenir une distribution plus simple à estimer et à réduire le rang des valeurs possible. La régression Lasso est une procédure de régularisation qui vise à éviter que le modèle ne surcharge les données et traite ainsi les problèmes de variance élevée. Dans le cadre d'un modèle linéaire standard, les coefficients sont obtenus par minimisation de la somme des carrés des résidus. Avec la méthode lasso, le vecteur de coefficients est également obtenu en minimisant la somme des carrés des résidus mais sous une contrainte supplémentaire qui est une **pénalisation de la norme L1** des coefficients.

$$\min_{\beta} \frac{1}{2} \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{i,j})^2 \quad s.c. \quad \sum_{j=1}^{p} |\beta_j| \leq t$$

Avant d'utiliser Lasso, je peux supprimer les variables pour lesquelles la **variance est faible**.

In [10]:

```
low_var = train_df.columns[train_df.var() <= 1].tolist()

train_df = train_df.drop(low_var, axis=1)
test_df = test_df.drop(low_var, axis=1)
```

In [11]:

```
print(f'Low variation features in train are {train_df.columns[train_df.var() <= 1].toli
st()}')
print(f'Low variation features in test are {test_df.columns[test_df.var() <= 1].tolist
()}')
```

```
Low variation features in train are []
Low variation features in test are ['time_to_eruption', 'sensor_8_med_0']
```

La **transformation de la variable cible** `time_to_eruption` réduit l'étandue de ses valeurs et permet d'obtenir une distribution plus simple à estimer. Il existe plusieurs fonctions à tester : sqrt, log, 1/x...

In [12]:

```
x_train = train_df.drop('time_to_eruption', axis=1)
x_test  = test_df.drop('time_to_eruption', axis=1)
y_train = train_df['time_to_eruption']
y_test = test_df['time_to_eruption']
```

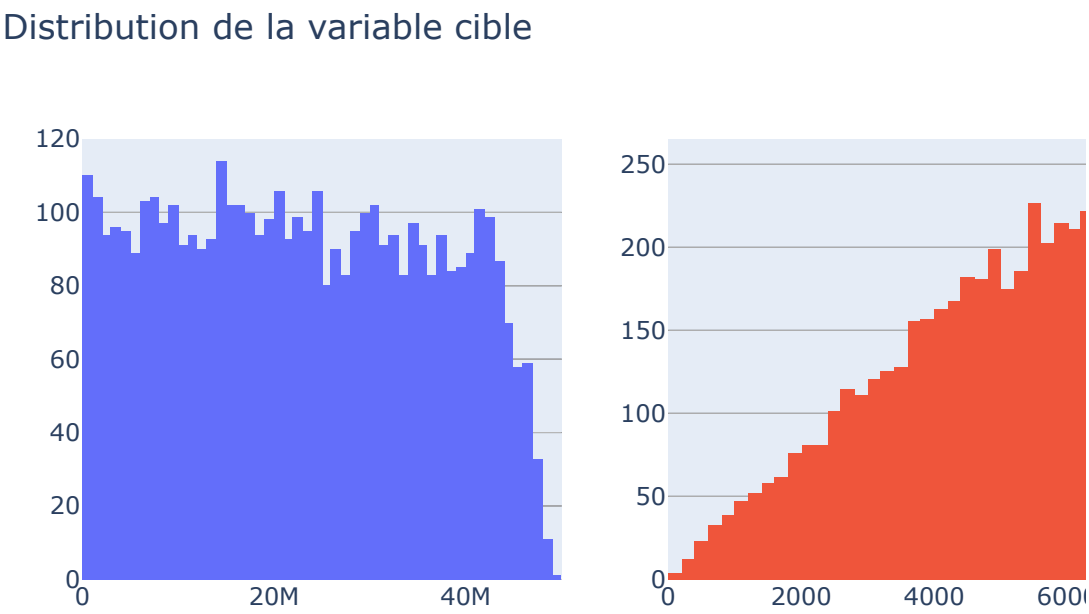
In [13]:

```
fig = make_subplots(rows=1, cols=2)

traces = []
traces.append(go.Histogram(x=y_train, name = 'Ø'))
traces.append(go.Histogram(x=np.sqrt(y_train), name = 'Transformation sqrt'))

for i in range(0, len(traces)) :
    fig.append_trace(traces[i], (i // 2) + 1, (i % 2) + 1)

fig.update_layout(height=400, width=800, title_text="Distribution de la variable cible"
)
fig.show()
```



Je cherche donc à trouver le meilleur alpha pour ma régression Lasso. Pour cela, j'utilise la recherche par grille complète.

In [ ]:

```
lasso_params = {'transformer__regressor__alpha': np.arange(0.01,25, 2)}
scaler = StandardScaler()

lasso = linear_model.Lasso(fit_intercept=True, normalize=False, max_iter=20_000, tol=1e
-3, positive=False, random_state=1, selection='random')
tt_lasso = TransformedTargetRegressor(regressor=lasso, func=np.sqrt, inverse_func=np.sq
uare)

pipe = Pipeline([('scaler', scaler), ('transformer', tt_lasso)])

rkf = RepeatedKFold(n_splits=3, n_repeats=4, random_state=1)
gs=GridSearchCV(pipe, param_grid=lasso_params, n_jobs=-1, cv=rkf).fit(x_train, y_train)
```

In [15]:

```
gs.best_params_
```

Out[15]:

```
{'transformer__regressor__alpha': 24.009999999999998}
```

Une fois alpha estimé, on ajuste la régression lasso et on garde les variables pour lesquelles on a un coefficient associé à la variable supérieur à 0. La contrainte contracte la valeur des coefficients et la forme de la pénalité va permettre à certains coefficients d'être à 0. En ne gardant que les variables avec un coefficient de la Lasso supérieur à 0, **je passe de 1871 à 78 variables**.

In [16]:

```
gs_alpha = 24

scaler = StandardScaler()
lasso = linear_model.Lasso(alpha = gs_alpha, fit_intercept=True, normalize=False, max_i
ter=20_000, tol=1e-3, positive=False, random_state=1, selection='random')
tt_lasso = TransformedTargetRegressor(regressor=lasso, func=np.sqrt, inverse_func=np.sq
uare)
pipe = Pipeline([('scaler', scaler), ('transformer', tt_lasso)])

pipe.fit(x_train, y_train)
lasso_coefs = pipe.named_steps['transformer'].regressor_.coef_
important_cols = x_train.columns[lasso_coefs != 0]
```

On a largement réduit notre jeu d'entrainement aux variables les plus utiles, on peut passser à la modélisation. Commençon par construire notre table où on va stocker nos résultats.

In [17]:

```
evaluation = pd.DataFrame(columns=['Model', 'Details', 'MAE', 'R2'])
```

Modélisation

Forêt Aléatoire

Le principe d'un arbre de décision est de construire itérativement une partition basée sur des **divisions successives d'hyperplans orthogonaux** aux axes de manière à séparer au mieux nos données. Chaque division définit deux noeuds, les noeuds fils à gauche et à droite et dont l'ensemble R^p constitue le noeud racine. À chaque noeud ou à chaque nouvelle division, l'algorithme choisit au hasard une variable et un seuil de séparation. On appelle noeud terminal ou feuille de l'arbre, un noeud pour lequel il y a plus de partitions admissibles, c'est-à-dire lorsqu'un noeud est totalement homogène ou encore lorsque le nombre d'observations qu'il contient est inférieur à une valeur seuil.

La forêt aléatoire est un algorithme de prédiction, constitué d'un **ensemble d'arbres de décision**. À chaque itération, l'algorithme construit un arbre selon un algorithme variant de l'arbre CART à partir d'un échantillon boostrap tiré avec remise. Pour chaque noeud, l'algorithme tire m variables explicatives et partitionne le noeud à partir de la meilleure de ces variables (gain d'homogénéité).

In [19]:

```
rfr = RandomForestRegressor()
tt_rfr = TransformedTargetRegressor(regressor=rfr, func=np.sqrt, inverse_func=np.square
)
pipe_rfr = Pipeline([('scaler', scaler), ('transformer', tt_rfr)])

cv = float(format((cross_val_score(pipe_rfr,
                                   x_train[important_cols],
                                   y_train,
                                   scoring='neg_mean_absolute_error',
                                   error_score='raise')*-1).mean(),'.3f'))

R2 = float(format(cross_val_score(pipe_rfr,
                                   x_train[important_cols],
                                   y_train).mean(),'.3f'))

evaluation.loc[0] = ['Random Forest', 'No Hyperopt', cv, R2]
evaluation.head()
```

Out[19]:

	Model	Details	MAE	R2
0	Random Forest	No Hyperopt	4515880.794	0.765

Pour ajuster au mieux mes **hyperparamètres** (nombres d'arbres, profondeur maximale, nombres de variables à considérer, nombre d'échantillons min. pour séparer/par feuille), j'utilise **l'optimisation bayésienne** qui permet d'estimer au mieux la fonction de perte (l'ensemble des combinaisons d'hyperparamètres et leurs scores associés) en explorant à chaque itération la combinaison qui offre le meilleur compromis en une combinaison situé dans une zone non exploitée par l'algorithme et le point minimum de la fonction de perte. L'avantage de cette méthode par rapport aux autres (grille aléatoire et complète) est qu'elle ne perd pas de temps à chercher des hyperparamètres dans un espace qui n'améliore pas notre modèle et s'avère être tout aussi efficace qu'une recherche complète sur le plan gain de temps/performance.

In [20]:

```
params_grid = {
    "n_estimators" : range(200, 1800),
    "max_depth" : range(10, 110),
    "max_features": ['auto', 'sqrt'],
    "min_samples_split" : [2,5,10],
    "min_samples_leaf" : [1,2,4],
    "bootstrap" : [True, False]
}
```

In [21]:

```
space = {
    'n_estimators': hp.choice('n_estimators', range(200, 1800)),
    'max_features': hp.choice('max_features', ['auto', 'sqrt']),
    'max_depth' : hp.choice('max_depth', range(10, 110)),
    'min_samples_leaf' : hp.choice('min_samples_leaf', [1,2,4]),
    'min_samples_split' : hp.choice('min_samples_split', [2,5,10]),
    'bootstrap' : hp.choice('bootstrap', [True, False])
        }

def f_nn(params):

    print ('Params testing: ', params)
    rfr = RandomForestRegressor(n_estimators=params['n_estimators'], max_depth=params[
'max_depth'], min_samples_split=params['min_samples_split'], min_samples_leaf=params['m
in_samples_leaf'], max_features=params['max_features'], bootstrap=params['bootstrap'])
    tt_rfr = TransformedTargetRegressor(regressor=rfr, func=np.sqrt, inverse_func=np.sq
uare)
    pipe_rfr = Pipeline([('scaler', scaler), ('transformer', tt_rfr)])

    cv = float(format((cross_val_score(pipe_rfr,
                                       x_train[important_cols],
                                       y_train,
                                       cv=rkf,
                                       n_jobs=-1,
                                       scoring='neg_mean_absolute_error',         #fonc
tion de perte
                                       error_score='raise')*-1).mean(),'.3f'))

    print('model_score:', cv)
    return {'loss': cv, 'status': STATUS_OK, 'model': pipe}

trials = Trials()
best = fmin(f_nn, space, algo=tpe.suggest, max_evals=25, trials=trials)
print('best: ', best)
```

```
Params testing:
{'bootstrap': False, 'max_depth': 95, 'max_features': 'sqrt', 'min_samp
les_leaf': 4, 'min_samples_split': 10, 'n_estimators': 1610}
model_score:
4764353.378
Params testing:
{'bootstrap': True, 'max_depth': 10, 'max_features': 'auto', 'min_sampl
es_leaf': 2, 'min_samples_split': 10, 'n_estimators': 900}
model_score:
5460045.498
Params testing:
{'bootstrap': False, 'max_depth': 78, 'max_features': 'sqrt', 'min_samp
les_leaf': 2, 'min_samples_split': 10, 'n_estimators': 459}
model_score:
4637393.571
Params testing:
{'bootstrap': True, 'max_depth': 95, 'max_features': 'auto', 'min_sampl
es_leaf': 1, 'min_samples_split': 5, 'n_estimators': 252}
model_score:
4741799.166
Params testing:
{'bootstrap': True, 'max_depth': 69, 'max_features': 'auto', 'min_sampl
es_leaf': 2, 'min_samples_split': 10, 'n_estimators': 1098}
model_score:
4797337.652
Params testing:
{'bootstrap': True, 'max_depth': 17, 'max_features': 'auto', 'min_sampl
es_leaf': 1, 'min_samples_split': 2, 'n_estimators': 1678}
model_score:
4731036.838
Params testing:
{'bootstrap': False, 'max_depth': 64, 'max_features': 'auto', 'min_samp
les_leaf': 4, 'min_samples_split': 2, 'n_estimators': 457}
model_score:
5489480.166
Params testing:
{'bootstrap': True, 'max_depth': 104, 'max_features': 'sqrt', 'min_samp
les_leaf': 1, 'min_samples_split': 10, 'n_estimators': 585}
model_score:
5321731.099
Params testing:
{'bootstrap': True, 'max_depth': 15, 'max_features': 'sqrt', 'min_sampl
es_leaf': 1, 'min_samples_split': 5, 'n_estimators': 1625}
model_score:
5319311.038
Params testing:
{'bootstrap': False, 'max_depth': 70, 'max_features': 'auto', 'min_samp
les_leaf': 1, 'min_samples_split': 5, 'n_estimators': 636}
model_score:
5311708.987
Params testing:
{'bootstrap': True, 'max_depth': 42, 'max_features': 'sqrt', 'min_sampl
es_leaf': 1, 'min_samples_split': 2, 'n_estimators': 1761}
model_score:
5165105.029
Params testing:
{'bootstrap': True, 'max_depth': 73, 'max_features': 'sqrt', 'min_sampl
es_leaf': 4, 'min_samples_split': 5, 'n_estimators': 315}
model_score:
5468095.875
Params testing:
```

```
{'bootstrap': True, 'max_depth': 80, 'max_features': 'sqrt', 'min_sampl
es_leaf': 4, 'min_samples_split': 10, 'n_estimators': 1591}
model_score:
5497790.151
Params testing:
{'bootstrap': True, 'max_depth': 15, 'max_features': 'auto', 'min_sampl
es_leaf': 4, 'min_samples_split': 5, 'n_estimators': 645}
model_score:
4899823.121
Params testing:
{'bootstrap': True, 'max_depth': 74, 'max_features': 'auto', 'min_sampl
es_leaf': 1, 'min_samples_split': 2, 'n_estimators': 1099}
model_score:
4717918.812
Params testing:
{'bootstrap': False, 'max_depth': 82, 'max_features': 'sqrt', 'min_samp
les_leaf': 4, 'min_samples_split': 2, 'n_estimators': 629}
model_score:
4733152.524
Params testing:
{'bootstrap': False, 'max_depth': 13, 'max_features': 'auto', 'min_samp
les_leaf': 2, 'min_samples_split': 10, 'n_estimators': 1077}
model_score:
5623795.332
Params testing:
{'bootstrap': True, 'max_depth': 77, 'max_features': 'auto', 'min_sampl
es_leaf': 1, 'min_samples_split': 10, 'n_estimators': 1179}
model_score:
4783010.15
Params testing:
{'bootstrap': False, 'max_depth': 52, 'max_features': 'auto', 'min_samp
les_leaf': 1, 'min_samples_split': 2, 'n_estimators': 1342}
model_score:
5279625.112
Params testing:
{'bootstrap': False, 'max_depth': 25, 'max_features': 'auto', 'min_samp
les_leaf': 1, 'min_samples_split': 10, 'n_estimators': 842}
model_score:
5408377.097
Params testing:
{'bootstrap': False, 'max_depth': 74, 'max_features': 'sqrt', 'min_samp
les_leaf': 2, 'min_samples_split': 2, 'n_estimators': 1080}
model_score:
4528493.621
Params testing:
{'bootstrap': False, 'max_depth': 41, 'max_features': 'sqrt', 'min_samp
les_leaf': 2, 'min_samples_split': 2, 'n_estimators': 1597}
model_score:
4521517.144
Params testing:
{'bootstrap': False, 'max_depth': 40, 'max_features': 'sqrt', 'min_samp
les_leaf': 2, 'min_samples_split': 2, 'n_estimators': 1349}
model_score:
4522646.833
Params testing:
{'bootstrap': False, 'max_depth': 83, 'max_features': 'sqrt', 'min_samp
les_leaf': 2, 'min_samples_split': 2, 'n_estimators': 1719}
model_score:
4523237.74
Params testing:
{'bootstrap': False, 'max_depth': 41, 'max_features': 'sqrt', 'min_samp
```

```
les_leaf': 2, 'min_samples_split': 2, 'n_estimators': 914}
model_score:
4526579.2
100%|██████████████████████████████████████████████| 25/25 [1:33:
10<00:00, 223.61s/trial, best loss: 4521517.144]
best:  {'bootstrap': 1, 'max_depth': 31, 'max_features': 1, 'min_sample
s_leaf': 1, 'min_samples_split': 0, 'n_estimators': 1397}
```

In [22]:

```
best = {'bootstrap': 1, 'max_depth': 31, 'max_features': 1, 'min_samples_leaf': 1, 'min
_samples_split': 0, 'n_estimators': 1397}
```

On ajuste à nouveau notre modèle avec les hyperparamètres sélectionnés sur l'ensemble du jeu d'entraînement.

In [23]:

```
rfr = RandomForestRegressor(bootstrap=params_grid['bootstrap'][best['bootstrap']], max_
depth=params_grid['max_depth'][best['max_depth']], max_features=params_grid['max_featur
es'][best['max_features']], min_samples_leaf=params_grid['min_samples_leaf'][best['min_
samples_leaf']], min_samples_split=params_grid['min_samples_split'][best['min_samples_s
plit']], n_estimators=params_grid['n_estimators'][best['n_estimators']])
tt_rfr = TransformedTargetRegressor(regressor=rfr, func=np.sqrt, inverse_func=np.square
)
pipe_rfr = Pipeline([('scaler', scaler), ('transformer', tt_rfr)])

cv = float(format((cross_val_score(pipe_rfr,
                                   x_train[important_cols],
                                   y_train,
                                   scoring='neg_mean_absolute_error',
                                   error_score='raise')*-1).mean(),'.3f'))

R2 = float(format(cross_val_score(pipe_rfr,
                                   x_train[important_cols],
                                   y_train).mean(),'.3f'))

evaluation.loc[1] = ['Random Forest', 'Hyperopt', cv, R2]
evaluation.head()
```

Out[23]:

	Model	Details	MAE	R2
0	Random Forest	No Hyperopt	4515880.794	0.765
1	Random Forest	Hyperopt	4312057.098	0.791

Support Vector Regression

La **régression par SVR** consiste à trouver une fonction $f(x)$ qui a **au plus une déviation** ϵ par rapport aux exemples d'apprentissage (x_i, y_i) et qui est la **plus plate** possible. Cela revient donc à ne pas considérer les erreurs inférieures à ϵ (se trouvant dans la marge) et à interdire celles supérieures à ϵ. Maximiser la platitude permet de minimiser la complexité du modèle qui influe sur ses performances. Le problème revient donc à minimiser la norme des poids en autorisant certaines erreurs et en garantissant que les erreurs sont inférieures à $\epsilon + \xi_i$:

$$min \frac{1}{2} ||w||^2 + C \sum_{i=1}^{N} (\xi_i + \xi_i^*)$$

sous les contraintes $y_i - \overrightarrow{w} \cdot \overrightarrow{x} - b \leq \epsilon + \xi_i$, $\overrightarrow{w} \cdot \overrightarrow{x} + b - y_i \leq \epsilon + \xi_i^*$ et $\xi_i, \xi_i^* \geq 0$ pour tout $i = 1, \ldots, N$. \

où ξ_i et ξ_i^* représentent respectivement les erreurs positives et négatives. C est dans ce cas de figure un hyperparamètre permettant de régler le compromis entre la quantité d'erreur autorisée et la platitude de la fonction f. \

Comme pour la méthode précédente, je détermine les hyperparamètres (C et epsilon) par validation croisée et optimisation bayésienne.

In [24]:

```
svr=svm.LinearSVR()
scaler = StandardScaler()
tt_svr = TransformedTargetRegressor(regressor=svr, func=np.sqrt, inverse_func=np.square
)
pipe_svr = Pipeline([('scaler', scaler), ('transformer', tt_svr)])

cv = float(format((cross_val_score(pipe_svr,
                                   x_train[important_cols],
                                   y_train,
                                   scoring='neg_mean_absolute_error',
                                   error_score='raise')*-1).mean(),'.3f'))

R2 = float(format(cross_val_score(pipe_svr,
                                   x_train[important_cols],
                                   y_train).mean(),'.3f'))

evaluation.loc[2] = ['SVR', 'No Hyperopt', cv, R2]
evaluation.head()
```

Out[24]:

	Model	Details	MAE	R2
0	Random Forest	No Hyperopt	4.515881e+06	0.765
1	Random Forest	Hyperopt	4.312057e+06	0.791
2	SVR	No Hyperopt	1.606734e+07	-1.172

LinearSVR revient à utiliser la SVR classique avec un noyau linéaire mais la fonction implémente plus de paramètres spécifiques aux noyaux linéaires tel que la **fonction de perte** (L1 ou L2) ou encore la **tolérance** (l'algorithme se stoppe dès lors que le score ne s'améliore pas plus que la tolérance).

In [25]:

```
params_grid = {
          "C": np.arange(0, 500, 1),
          "loss": ['epsilon_insensitive', 'squared_epsilon_insensitive'],
          "epsilon" : np.arange(0, 1, 0.1),
          "tol" : [1, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
        }
```

In [26]:

```
space = {
          'C': hp.choice('C', np.arange(0, 500, 1).tolist()),
          'loss' : hp.choice('loss', ['epsilon_insensitive', 'squared_epsilon_insensiti
ve']),
          'tol' : hp.choice('tol', [1, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]),
          'epsilon' : hp.choice('epsilon', np.arange(0, 1, 0.1).tolist()),
        }

def f_nn(params):
    print ('Params testing: ', params)

    svr = svm.LinearSVR(loss=params['loss'],
                        tol=params['tol'],
                        epsilon=params['epsilon'],
                        C=params['C'],
                        fit_intercept=True,
                        intercept_scaling=1,
                        dual=True,
                        verbose=0,
                        random_state=1,
                        max_iter=1000)

    tt_svr = TransformedTargetRegressor(regressor=svr, func=np.sqrt, inverse_func=np.sq
uare)
    pipe_svr = Pipeline([('scaler', scaler), ('transformer', tt_svr)])

    cv = float(format((cross_val_score(pipe_svr,
                                       x_train[important_cols],
                                       y_train,
                                       cv=4,
                                       n_jobs=-1,
                                       scoring='neg_mean_absolute_error',
                                       error_score='raise')*-1).mean(),'.3f'))

    print('model_score:', cv)
    return {'loss': cv, 'status': STATUS_OK, 'model': pipe}

trials = Trials()
best = fmin(f_nn, space, algo=tpe.suggest, max_evals=50, trials=trials)
print('best: ', best)
```

```
Params testing:
{'C': 438, 'epsilon': 0.0, 'loss': 'squared_epsilon_insensitive', 'tol':
0.0001}
model_score:
23042241.691
Params testing:
{'C': 302, 'epsilon': 0.30000000000000004, 'loss': 'epsilon_insensitive',
'tol': 0.1}
model_score:
10182296.634
Params testing:
{'C': 28, 'epsilon': 0.30000000000000004, 'loss': 'epsilon_insensitive',
'tol': 1e-05}
model_score:
9747910.096
Params testing:
{'C': 247, 'epsilon': 0.7000000000000001, 'loss': 'squared_epsilon_insensi
tive', 'tol': 0.1}
model_score:
21946983.401
Params testing:
{'C': 470, 'epsilon': 0.9, 'loss': 'squared_epsilon_insensitive', 'tol':
0.1}
model_score:
23128820.025
Params testing:
{'C': 110, 'epsilon': 0.0, 'loss': 'epsilon_insensitive', 'tol': 1}
model_score:
14979161.108
Params testing:
{'C': 48, 'epsilon': 0.9, 'loss': 'epsilon_insensitive', 'tol': 1e-06}
model_score:
9749140.557
Params testing:
{'C': 226, 'epsilon': 0.0, 'loss': 'epsilon_insensitive', 'tol': 0.001}
model_score:
9759751.595
Params testing:
{'C': 493, 'epsilon': 0.1, 'loss': 'epsilon_insensitive', 'tol': 1e-06}
model_score:
9754972.596
Params testing:
{'C': 83, 'epsilon': 0.2, 'loss': 'epsilon_insensitive', 'tol': 1}
model_score:
13477487.706
Params testing:
{'C': 49, 'epsilon': 0.5, 'loss': 'epsilon_insensitive', 'tol': 1e-06}
model_score:
9748212.288
Params testing:
{'C': 100, 'epsilon': 0.30000000000000004, 'loss': 'epsilon_insensitive',
'tol': 1e-06}
model_score:
9747642.254
Params testing:
{'C': 456, 'epsilon': 0.4, 'loss': 'epsilon_insensitive', 'tol': 0.01}
model_score:
9755951.001
Params testing:
{'C': 16, 'epsilon': 0.2, 'loss': 'epsilon_insensitive', 'tol': 1e-05}
model_score:
```

```
9800629.941
Params testing:
{'C': 112, 'epsilon': 0.30000000000000004, 'loss': 'epsilon_insensitive',
'tol': 1e-05}
model_score:
9748864.563
Params testing:
{'C': 114, 'epsilon': 0.9, 'loss': 'squared_epsilon_insensitive', 'tol':
0.01}
model_score:
19508305.034
Params testing:
{'C': 418, 'epsilon': 0.2, 'loss': 'squared_epsilon_insensitive', 'tol':
0.1}
model_score:
22968800.416
Params testing:
{'C': 334, 'epsilon': 0.4, 'loss': 'squared_epsilon_insensitive', 'tol': 1
e-06}
model_score:
22586237.806
Params testing:
{'C': 111, 'epsilon': 0.8, 'loss': 'epsilon_insensitive', 'tol': 0.001}
model_score:
9752534.329
Params testing:
{'C': 272, 'epsilon': 0.6000000000000001, 'loss': 'squared_epsilon_insensi
tive', 'tol': 1e-06}
model_score:
22168304.557
Params testing:
{'C': 103, 'epsilon': 0.30000000000000004, 'loss': 'epsilon_insensitive',
'tol': 1e-05}
model_score:
9747482.075
Params testing:
{'C': 209, 'epsilon': 0.30000000000000004, 'loss': 'epsilon_insensitive',
'tol': 0.0001}
model_score:
9756444.232
Params testing:
{'C': 100, 'epsilon': 0.30000000000000004, 'loss': 'epsilon_insensitive',
'tol': 1e-05}
model_score:
9747642.254
Params testing:
{'C': 103, 'epsilon': 0.30000000000000004, 'loss': 'epsilon_insensitive',
'tol': 1e-05}
model_score:
9747482.075
Params testing:
{'C': 74, 'epsilon': 0.1, 'loss': 'epsilon_insensitive', 'tol': 1e-05}
model_score:
9747987.918
Params testing:
{'C': 472, 'epsilon': 0.6000000000000001, 'loss': 'epsilon_insensitive',
'tol': 1e-05}
model_score:
9739480.681
Params testing:
{'C': 41, 'epsilon': 0.6000000000000001, 'loss': 'epsilon_insensitive', 't
```

```
ol': 1e-05}
model_score:
9745653.28
Params testing:
{'C': 29, 'epsilon': 0.6000000000000001, 'loss': 'epsilon_insensitive', 't
ol': 1e-05}
model_score:
9749966.24
Params testing:
{'C': 293, 'epsilon': 0.6000000000000001, 'loss': 'epsilon_insensitive',
'tol': 0.0001}
model_score:
9750543.169
Params testing:
{'C': 441, 'epsilon': 0.6000000000000001, 'loss': 'epsilon_insensitive',
'tol': 1e-05}
model_score:
9756303.256
Params testing:
{'C': 407, 'epsilon': 0.6000000000000001, 'loss': 'squared_epsilon_insensi
tive', 'tol': 0.01}
model_score:
22921216.967
Params testing:
{'C': 472, 'epsilon': 0.6000000000000001, 'loss': 'epsilon_insensitive',
'tol': 1}
model_score:
20202797.573
Params testing:
{'C': 41, 'epsilon': 0.8, 'loss': 'epsilon_insensitive', 'tol': 0.001}
model_score:
9753854.843
Params testing:
{'C': 413, 'epsilon': 0.7000000000000001, 'loss': 'squared_epsilon_insensi
tive', 'tol': 0.0001}
model_score:
22942478.918
Params testing:
{'C': 384, 'epsilon': 0.5, 'loss': 'epsilon_insensitive', 'tol': 1e-05}
model_score:
9743813.793
Params testing:
{'C': 416, 'epsilon': 0.5, 'loss': 'epsilon_insensitive', 'tol': 0.1}
model_score:
10015020.109
Params testing:
{'C': 144, 'epsilon': 0.5, 'loss': 'epsilon_insensitive', 'tol': 1e-05}
model_score:
9751445.496
Params testing:
{'C': 125, 'epsilon': 0.5, 'loss': 'squared_epsilon_insensitive', 'tol':
1}
model_score:
20220096.342
Params testing:
{'C': 288, 'epsilon': 0.0, 'loss': 'epsilon_insensitive', 'tol': 0.001}
model_score:
9752591.513
Params testing:
{'C': 256, 'epsilon': 0.7000000000000001, 'loss': 'epsilon_insensitive',
'tol': 1e-05}
```

```
model_score:
9751280.496
Params testing:
{'C': 311, 'epsilon': 0.5, 'loss': 'epsilon_insensitive', 'tol': 0.1}
model_score:
10180750.993
Params testing:
{'C': 364, 'epsilon': 0.1, 'loss': 'epsilon_insensitive', 'tol': 0.01}
model_score:
9737278.732
Params testing:
{'C': 150, 'epsilon': 0.1, 'loss': 'squared_epsilon_insensitive', 'tol':
0.01}
model_score:
20528525.145
Params testing:
{'C': 10, 'epsilon': 0.1, 'loss': 'epsilon_insensitive', 'tol': 0.01}
model_score:
9886585.115
Params testing:
{'C': 198, 'epsilon': 0.1, 'loss': 'epsilon_insensitive', 'tol': 0.01}
model_score:
9788257.335
Params testing:
{'C': 95, 'epsilon': 0.9, 'loss': 'epsilon_insensitive', 'tol': 0.01}
model_score:
9744258.753
Params testing:
{'C': 77, 'epsilon': 0.4, 'loss': 'squared_epsilon_insensitive', 'tol': 0.
01}
model_score:
17807558.904
Params testing:
{'C': 446, 'epsilon': 0.0, 'loss': 'epsilon_insensitive', 'tol': 1}
model_score:
20201147.618
Params testing:
{'C': 395, 'epsilon': 0.1, 'loss': 'epsilon_insensitive', 'tol': 0.0001}
model_score:
9760353.03
Params testing:
{'C': 168, 'epsilon': 0.8, 'loss': 'epsilon_insensitive', 'tol': 0.01}
model_score:
9757952.415
100%|██████████████████████████████████████████████████| 50/50 [01:06
<00:00,  1.33s/trial, best loss: 9737278.732]
best:  {'C': 364, 'epsilon': 1, 'loss': 0, 'tol': 2}
```

In [27]:

```
best = {'C': 364, 'epsilon': 1, 'loss': 0, 'tol': 2}
```

In [28]:

```
svr = svm.LinearSVR(loss=params_grid['loss'][best['loss']],
                    tol=params_grid['tol'][best['tol']],
                    epsilon=params_grid['epsilon'][best['epsilon']],
                    C=params_grid['C'][best['C']],
                    fit_intercept=True,
                    intercept_scaling=1,
                    dual=True,
                    verbose=0,
                    random_state=1,
                    max_iter=10000)

tt_svr = TransformedTargetRegressor(regressor=svr, func=np.sqrt, inverse_func=np.square
)
pipe_svr = Pipeline([('scaler', scaler), ('transformer', tt_svr)])

cv = float(format((cross_val_score(pipe_svr,
                                   x_train[important_cols],
                                   y_train,
                                   scoring='neg_mean_absolute_error',
                                   error_score='raise')*-1).mean(),'.3f'))

R2 = float(format(cross_val_score(pipe_svr,
                                  x_train[important_cols],
                                  y_train).mean(),'.3f'))

evaluation.loc[3] = ['SVR', 'Hyperopt', cv, R2]
evaluation.head()
```

Out[28]:

	Model	Details	MAE	R2
0	Random Forest	No Hyperopt	4.515881e+06	0.765
1	Random Forest	Hyperopt	4.312057e+06	0.791
2	SVR	No Hyperopt	1.606734e+07	-1.172
3	SVR	Hyperopt	9.761188e+06	0.189

Extrem Gradient Boosting (XGBOOST)

Le **boosting** adopte la même stratégie que la bagging (forêt aléatoire) : construction d'une famille de modèles qui sont ensuite agrégés par une moyenne pondérée des estimations ou d'un vote. La différence apparaît ici sur la façon de construire la famille. Chaque version du modèle est adaptative du précédent en donnant plus de poids lors de l'estimation suivante aux observations mal classées ou mal ajustées. L'algorithme concentre ainsi ses efforts sur ces données. L'idée du boosting est de construire un classifieur fort (dont l'erreur est très petite) à partir d'un grand ensemble de classifieurs faibles (ex : nos arbres).

L'algorithme XGBOOST utilise **la méthode de descente par gradient** combinée à un **terme de régularisation** qui minimise la valeur de la fonction de perte. La fonction objectif s'écrit :

$$L(\theta) = \sum_{i=1}^{n} l(y_i, F(x_i)) + \sum_{i=1}^{T} \Omega(f)$$

avec :

$$\Omega(f) = \gamma T + \frac{1}{2}\beta \sum_{i=j}^{T} w_j^2$$

où T correspond au nombre de feuilles de l'arbre f_m, w le vecteur des valeurs attribuées à chacune de ses feuilles et l la fonction de perte.

Ce terme de régularisation implique une différence dans la construction des arbres par rapport à un arbre CART que je ne développerai pas ici (gain d'une séparation, score de similarité, exact/approximate greedy enumaration...). On retrouve quasiment les hyperparamètres de la méthode de forêt aléatoire avec **deux paramètres de régularisation L1, L2** et le **taux d'apprentissage** qui permet d'échelonner la contribution d'un nouvel arbre.

In [29]:

```
xgb_r = xgb.XGBRegressor()
scaler = StandardScaler()
tt_xgb = TransformedTargetRegressor(regressor=xgb_r, func=np.sqrt, inverse_func=np.squa
re)
pipe_xgb = Pipeline([('scaler', scaler), ('transformer', tt_xgb)])

cv = float(format((cross_val_score(pipe_xgb,
                                   x_train[important_cols],
                                   y_train,
                                   scoring='neg_mean_absolute_error',
                                   error_score='raise')*-1).mean(),'.3f'))

R2 = float(format(cross_val_score(pipe_xgb,
                                   x_train[important_cols],
                                   y_train).mean(),'.3f'))

evaluation.loc[4] = ['XGBOOST', 'No Hyperopt', cv, R2]
evaluation.head()
```

Out[29]:

	Model	Details	MAE	R2
0	Random Forest	No Hyperopt	4.515881e+06	0.765
1	Random Forest	Hyperopt	4.312057e+06	0.791
2	SVR	No Hyperopt	1.606734e+07	-1.172
3	SVR	Hyperopt	9.761188e+06	0.189
4	XGBOOST	No Hyperopt	5.085187e+06	0.730

- booster : le type d'apprenant. Cela peut-être un arbre ou un modèle linaire.
- learning_rate : taux d'apprentissage du modèle qui permet d'échelonner la contribution d'un nouvel arbre.
- max_depth : profondeur max de l'arbre.
- min_child_weight : poids (ou nombre) d'échantillons requis pour créer une nouvelle partition (sum hi).
- colsample_bytree : pourcentage de variables à considérer pour un arbre.
- n_estimators : nombre d'arbres.
- subsample : ratio de sous-echantillonnage des données pour construire un arbre.
- lambda : paramètre L2 de régularisation (beta).
- alpha : paramètre L1 de régularisation (gamma).
- tree-method : exact greedy algorithm, approximate greedy algorithm used quantile sketch.
- max_bin : nombre bin maximum.
- objective : fonction de perte à optimiser.

In [30]:

```
params_grid = {
    "n_estimators" : range(100, 2000),
    "max_depth" : range(1, 16),
    "learning_rate" : np.arange(0.05, 0.55, 0.05).tolist(),
    "objective" : ['reg:squarederror', 'reg:squaredlogerror', 'reg:pseudohubererror'],
    "min_child_weight" : range(1, 8),
    "colsample_bytree" : np.arange(0.2, 0.9, 0.1).tolist(),
    "subsample" : np.arange(0.5, 1.1, 0.1).tolist(),
    "reg_lambda" : np.arange(0.1, 1.1, 0.1).tolist(),
    "reg_alpha" : range(0, 10)
}
```

In [32]:

```
space = {
    'n_estimators': hp.choice('n_estimators', range(100, 2000)),
    'max_depth' : hp.choice('max_depth', range(1, 16)),
    'learning_rate' : hp.choice('learning_rate', np.arange(0.05, 0.55, 0.05).tolist()),
    'objective' : hp.choice('objective', ['reg:squarederror', 'reg:squaredlogerror', 'r
eg:pseudohubererror']),
    'min_child_weight' : hp.choice('min_child_weight', range(1, 8)),
    'colsample_bytree' : hp.choice('colsample_bytree', np.arange(0.2, 0.9, 0.1).tolist
()),
    'subsample' : hp.choice('subsample', np.arange(0.5, 1, 0.1).tolist()),
    'reg_lambda' : hp.choice('reg_lambda', np.arange(0.1, 1.1, 0.1).tolist()),
    'reg_alpha' : hp.choice('reg_alpha', range(0, 10))
}

def f_nn(params):

    print ('Params testing: ', params)

    xgb_r = xgb.XGBRegressor(tree_method='auto', #fastest method
                             n_estimators=params['n_estimators'],
                             max_depth=params['max_depth'],
                             learning_rate=params['learning_rate'],
                             objective=params['objective'],
                             min_child_weight=params['min_child_weight'],
                             colsample_bytree=params['colsample_bytree'],
                             subsample=params['subsample'],
                             reg_lambda=params['reg_lambda'],
                             reg_alpha=params['reg_alpha'])

    tt_xgb = TransformedTargetRegressor(regressor=xgb_r, func=np.sqrt, inverse_func=np.
square)
    pipe_xgb = Pipeline([('scaler', scaler), ('transformer', tt_xgb)])

    cv = float(format((cross_val_score(pipe_xgb,
                                       x_train[important_cols],
                                       y_train,
                                       cv=5,
                                       n_jobs=-1,
                                       scoring='neg_mean_absolute_error',
                                       error_score='raise')*-1).mean(),'.3f'))

    print('model_score:', cv)
    return {'loss': cv, 'status': STATUS_OK, 'model': pipe}

trials = Trials()
best = fmin(f_nn, space, algo=tpe.suggest, max_evals=100, trials=trials)
print('best: ', best)
```

```
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.2, 'max_dep
th': 5, 'min_child_weight': 3, 'n_estimators': 1063, 'objective': 'reg:
pseudohubererror', 'reg_alpha': 6, 'reg_lambda': 0.4, 'subsample': 0.79
99999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.35000000000000003, 'max_de
pth': 15, 'min_child_weight': 7, 'n_estimators': 1114, 'objective': 're
g:pseudohubererror', 'reg_alpha': 3, 'reg_lambda': 0.9, 'subsample': 0.
7999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.05, 'max_depth': 3, 'min_c
hild_weight': 6, 'n_estimators': 1893, 'objective': 'reg:squarederror',
'reg_alpha': 7, 'reg_lambda': 0.5, 'subsample': 0.7999999999999999}
model_score:
5263437.791
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.25, 'max_de
pth': 6, 'min_child_weight': 2, 'n_estimators': 1567, 'objective': 're
g:squarederror', 'reg_alpha': 9, 'reg_lambda': 0.1, 'subsample': 0.7999
999999999999}
model_score:
5178471.538
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.25, 'max_de
pth': 11, 'min_child_weight': 3, 'n_estimators': 206, 'objective': 're
g:squaredlogerror', 'reg_alpha': 4, 'reg_lambda': 0.2, 'subsample': 0.8
999999999999999}
model_score:
22842349.331
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.2, 'max_de
pth': 9, 'min_child_weight': 6, 'n_estimators': 233, 'objective': 'reg:
squarederror', 'reg_alpha': 0, 'reg_lambda': 0.5, 'subsample': 0.7}
model_score:
4787971.622
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.1, 'max_dep
th': 3, 'min_child_weight': 3, 'n_estimators': 984, 'objective': 'reg:p
seudohubererror', 'reg_alpha': 3, 'reg_lambda': 0.1, 'subsample': 0.899
9999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.15000000000
000002, 'max_depth': 2, 'min_child_weight': 7, 'n_estimators': 1549, 'o
bjective': 'reg:pseudohubererror', 'reg_alpha': 6, 'reg_lambda': 0.3000
0000000000004, 'subsample': 0.7999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.1, 'max_dep
th': 2, 'min_child_weight': 1, 'n_estimators': 978, 'objective': 'reg:s
quarederror', 'reg_alpha': 1, 'reg_lambda': 0.7000000000000001, 'subsam
ple': 0.7999999999999999}
model_score:
5860233.304
```

```
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.4, 'max_depth': 5, 'min_ch
ild_weight': 7, 'n_estimators': 1983, 'objective': 'reg:pseudohubererro
r', 'reg_alpha': 8, 'reg_lambda': 0.8, 'subsample': 0.5}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.15000000000000002, 'max_de
pth': 15, 'min_child_weight': 3, 'n_estimators': 1692, 'objective': 're
g:squaredlogerror', 'reg_alpha': 1, 'reg_lambda': 0.7000000000000001,
'subsample': 0.5}
model_score:
22845160.427
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.25, 'max_de
pth': 13, 'min_child_weight': 3, 'n_estimators': 1730, 'objective': 're
g:pseudohubererror', 'reg_alpha': 9, 'reg_lambda': 0.5, 'subsample': 0.
5}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.3, 'max_de
pth': 3, 'min_child_weight': 1, 'n_estimators': 242, 'objective': 'reg:
squarederror', 'reg_alpha': 3, 'reg_lambda': 0.9, 'subsample': 0.7}
model_score:
5805399.963
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.35000000000
000003, 'max_depth': 2, 'min_child_weight': 6, 'n_estimators': 784, 'ob
jective': 'reg:squarederror', 'reg_alpha': 2, 'reg_lambda': 1.0, 'subsa
mple': 0.5}
model_score:
6597122.075
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.4, 'max_dep
th': 14, 'min_child_weight': 1, 'n_estimators': 1371, 'objective': 're
g:squaredlogerror', 'reg_alpha': 9, 'reg_lambda': 0.8, 'subsample': 0.
6}
model_score:
22837768.365
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.45, 'max_de
pth': 9, 'min_child_weight': 5, 'n_estimators': 1569, 'objective': 're
g:squaredlogerror', 'reg_alpha': 0, 'reg_lambda': 0.5, 'subsample': 0.
6}
model_score:
22846289.31
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.15000000000
000002, 'max_depth': 5, 'min_child_weight': 3, 'n_estimators': 1393, 'o
bjective': 'reg:pseudohubererror', 'reg_alpha': 1, 'reg_lambda': 0.7000
000000000001, 'subsample': 0.8999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.2, 'max_dep
th': 3, 'min_child_weight': 6, 'n_estimators': 590, 'objective': 'reg:p
seudohubererror', 'reg_alpha': 2, 'reg_lambda': 0.8, 'subsample': 0.6}
model_score:
22848977.764
Params testing:
```

```
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.15000000000
000002, 'max_depth': 4, 'min_child_weight': 6, 'n_estimators': 1233, 'o
bjective': 'reg:pseudohubererror', 'reg_alpha': 1, 'reg_lambda': 0.5,
'subsample': 0.8999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.3, 'max_dep
th': 11, 'min_child_weight': 3, 'n_estimators': 1148, 'objective': 're
g:squarederror', 'reg_alpha': 2, 'reg_lambda': 0.2, 'subsample': 0.5}
model_score:
5905480.221
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.2, 'max_de
pth': 6, 'min_child_weight': 2, 'n_estimators': 1067, 'objective': 're
g:squarederror', 'reg_alpha': 0, 'reg_lambda': 0.1, 'subsample': 0.7}
model_score:
5052730.889
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.5, 'max_de
pth': 10, 'min_child_weight': 2, 'n_estimators': 1770, 'objective': 're
g:squarederror', 'reg_alpha': 0, 'reg_lambda': 0.6, 'subsample': 0.7}
model_score:
6040141.906
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.2, 'max_de
pth': 6, 'min_child_weight': 4, 'n_estimators': 1002, 'objective': 're
g:squarederror', 'reg_alpha': 5, 'reg_lambda': 0.1, 'subsample': 0.7}
model_score:
5133294.61
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.2, 'max_de
pth': 9, 'min_child_weight': 2, 'n_estimators': 1134, 'objective': 're
g:squarederror', 'reg_alpha': 0, 'reg_lambda': 0.30000000000000004, 'su
bsample': 0.7}
model_score:
4839508.477
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.2, 'max_de
pth': 9, 'min_child_weight': 2, 'n_estimators': 679, 'objective': 'reg:
squarederror', 'reg_alpha': 0, 'reg_lambda': 0.30000000000000004, 'subs
ample': 0.7}
model_score:
4839508.422
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.2, 'max_de
pth': 9, 'min_child_weight': 5, 'n_estimators': 1188, 'objective': 're
g:squarederror', 'reg_alpha': 0, 'reg_lambda': 0.30000000000000004, 'su
bsample': 0.7}
model_score:
4812545.37
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.5, 'max_de
pth': 1, 'min_child_weight': 5, 'n_estimators': 160, 'objective': 'reg:
squarederror', 'reg_alpha': 7, 'reg_lambda': 1.0, 'subsample': 0.7}
model_score:
7861615.268
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.05, 'max_d
epth': 8, 'min_child_weight': 5, 'n_estimators': 1518, 'objective': 're
g:squarederror', 'reg_alpha': 0, 'reg_lambda': 0.30000000000000004, 'su
```

```
bsample': 0.7}
model_score:
4407388.955
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.05, 'max_d
epth': 8, 'min_child_weight': 4, 'n_estimators': 442, 'objective': 're
g:squarederror', 'reg_alpha': 5, 'reg_lambda': 0.4, 'subsample': 0.7}
model_score:
4422229.519
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.05, 'max_de
pth': 8, 'min_child_weight': 4, 'n_estimators': 442, 'objective': 'reg:
squaredlogerror', 'reg_alpha': 5, 'reg_lambda': 0.4, 'subsample': 0.7}
model_score:
22845713.933
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 8, 'min_child_weight': 4, 'n_estimators': 405, 'objective': 'reg:
squarederror', 'reg_alpha': 5, 'reg_lambda': 0.4, 'subsample': 0.7}
model_score:
4319509.946
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 8, 'min_child_weight': 5, 'n_estimators': 167, 'objective': 'reg:
squarederror', 'reg_alpha': 5, 'reg_lambda': 0.4, 'subsample': 0.6}
model_score:
4511230.227
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 838, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7}
model_score:
4110359.668
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 1031, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7999
999999999999}
model_score:
4083165.317
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.45, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 838, 'objective': 're
g:squaredlogerror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7
999999999999999}
model_score:
22844403.569
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 1846, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7999
999999999999}
model_score:
4083151.548
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.35000000000
000003, 'max_depth': 12, 'min_child_weight': 4, 'n_estimators': 1090,
'objective': 'reg:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'su
bsample': 0.7999999999999999}
model_score:
4949152.816
```

```
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 7, 'min_child_weight': 4, 'n_estimators': 1846, 'objective': 're
g:squarederror', 'reg_alpha': 4, 'reg_lambda': 0.6, 'subsample': 0.7999
999999999999}
model_score:
4456866.425
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.25, 'max_de
pth': 12, 'min_child_weight': 7, 'n_estimators': 738, 'objective': 're
g:squaredlogerror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7
999999999999999}
model_score:
22845957.746
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.1, 'max_dep
th': 12, 'min_child_weight': 4, 'n_estimators': 169, 'objective': 'reg:
squarederror', 'reg_alpha': 6, 'reg_lambda': 0.6, 'subsample': 0.799999
9999999999}
model_score:
4225705.282
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.05, 'max_depth': 15, 'min_
child_weight': 4, 'n_estimators': 367, 'objective': 'reg:squarederror',
'reg_alpha': 8, 'reg_lambda': 0.9, 'subsample': 0.7999999999999999}
model_score:
4617692.634
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.4, 'max_dep
th': 14, 'min_child_weight': 7, 'n_estimators': 734, 'objective': 'reg:
pseudohubererror', 'reg_alpha': 7, 'reg_lambda': 0.2, 'subsample': 0.79
99999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.3, 'max_dep
th': 13, 'min_child_weight': 1, 'n_estimators': 374, 'objective': 'reg:
squaredlogerror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.799
9999999999999}
model_score:
22833990.148
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.35000000000
000003, 'max_depth': 10, 'min_child_weight': 4, 'n_estimators': 808, 'o
bjective': 'reg:squarederror', 'reg_alpha': 4, 'reg_lambda': 1.0, 'subs
ample': 0.7999999999999999}
model_score:
5215394.446
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.1, 'max_dep
th': 4, 'min_child_weight': 4, 'n_estimators': 961, 'objective': 'reg:p
seudohubererror', 'reg_alpha': 6, 'reg_lambda': 0.9, 'subsample': 0.899
9999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.45, 'max_de
pth': 12, 'min_child_weight': 1, 'n_estimators': 1265, 'objective': 're
g:squarederror', 'reg_alpha': 3, 'reg_lambda': 0.6, 'subsample': 0.7999
999999999999}
model_score:
```

```
5724485.9
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.5, 'max_depth': 7, 'min_ch
ild_weight': 7, 'n_estimators': 1019, 'objective': 'reg:squaredlogerro
r', 'reg_alpha': 8, 'reg_lambda': 0.7000000000000001, 'subsample': 0.79
99999999999999}
model_score:
22846481.673
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.25, 'max_de
pth': 11, 'min_child_weight': 6, 'n_estimators': 1677, 'objective': 're
g:squarederror', 'reg_alpha': 9, 'reg_lambda': 0.8, 'subsample': 0.5}
model_score:
5570840.742
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.05, 'max_de
pth': 1, 'min_child_weight': 4, 'n_estimators': 1444, 'objective': 're
g:pseudohubererror', 'reg_alpha': 8, 'reg_lambda': 0.2, 'subsample': 0.
8999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.4, 'max_dep
th': 5, 'min_child_weight': 3, 'n_estimators': 1584, 'objective': 'reg:
squarederror', 'reg_alpha': 3, 'reg_lambda': 0.6, 'subsample': 0.6}
model_score:
6346488.941
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.15000000000
000002, 'max_depth': 2, 'min_child_weight': 6, 'n_estimators': 944, 'ob
jective': 'reg:squarederror', 'reg_alpha': 7, 'reg_lambda': 0.5, 'subsa
mple': 0.7999999999999999}
model_score:
5884246.596
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.3, 'max_dep
th': 12, 'min_child_weight': 1, 'n_estimators': 1559, 'objective': 're
g:pseudohubererror', 'reg_alpha': 8, 'reg_lambda': 0.1, 'subsample': 0.
7999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.05, 'max_depth': 15, 'min_
child_weight': 4, 'n_estimators': 182, 'objective': 'reg:squaredlogerro
r', 'reg_alpha': 2, 'reg_lambda': 0.7000000000000001, 'subsample': 0.5}
model_score:
22846536.296
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.1, 'max_dep
th': 3, 'min_child_weight': 2, 'n_estimators': 265, 'objective': 'reg:s
quarederror', 'reg_alpha': 1, 'reg_lambda': 1.0, 'subsample': 0.8999999
999999999}
model_score:
5819665.728
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.35000000000
000003, 'max_depth': 13, 'min_child_weight': 3, 'n_estimators': 425, 'o
bjective': 'reg:squarederror', 'reg_alpha': 4, 'reg_lambda': 0.9, 'subs
ample': 0.6}
model_score:
5728800.254
```

```
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.45, 'max_de
pth': 14, 'min_child_weight': 7, 'n_estimators': 1586, 'objective': 're
g:pseudohubererror', 'reg_alpha': 9, 'reg_lambda': 0.6, 'subsample': 0.
7999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.05, 'max_de
pth': 6, 'min_child_weight': 4, 'n_estimators': 255, 'objective': 'reg:
squarederror', 'reg_alpha': 6, 'reg_lambda': 0.8, 'subsample': 0.5}
model_score:
4867552.006
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.5, 'max_dep
th': 4, 'min_child_weight': 6, 'n_estimators': 1082, 'objective': 'reg:
squaredlogerror', 'reg_alpha': 8, 'reg_lambda': 0.2, 'subsample': 0.799
9999999999999}
model_score:
22846368.549
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.25, 'max_de
pth': 12, 'min_child_weight': 5, 'n_estimators': 1719, 'objective': 're
g:squarederror', 'reg_alpha': 3, 'reg_lambda': 0.1, 'subsample': 0.8999
999999999999}
model_score:
4464015.066
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.15000000000
000002, 'max_depth': 10, 'min_child_weight': 1, 'n_estimators': 237, 'o
bjective': 'reg:squarederror', 'reg_alpha': 2, 'reg_lambda': 0.5, 'subs
ample': 0.6}
model_score:
4681510.701
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.4, 'max_depth': 11, 'min_c
hild_weight': 2, 'n_estimators': 1032, 'objective': 'reg:pseudohubererr
or', 'reg_alpha': 1, 'reg_lambda': 0.6, 'subsample': 0.799999999999999
9}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.05, 'max_de
pth': 1, 'min_child_weight': 3, 'n_estimators': 1383, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.7000000000000001, 'sub
sample': 0.7999999999999999}
model_score:
7893668.897
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.3, 'max_dep
th': 5, 'min_child_weight': 4, 'n_estimators': 584, 'objective': 'reg:s
quarederror', 'reg_alpha': 7, 'reg_lambda': 0.30000000000000004, 'subsa
mple': 0.5}
model_score:
6062354.266
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.05, 'max_de
pth': 2, 'min_child_weight': 4, 'n_estimators': 706, 'objective': 'reg:
squaredlogerror', 'reg_alpha': 9, 'reg_lambda': 1.0, 'subsample': 0.799
9999999999999}
model_score:
```

```
22845208.272
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.5, 'max_dep
th': 12, 'min_child_weight': 5, 'n_estimators': 529, 'objective': 'reg:
squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.899999
9999999999}
model_score:
5380140.873
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 1373, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7}
model_score:
4110313.936
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 1629, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7}
model_score:
4110307.395
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 557, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7}
model_score:
4110563.977
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 209, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7}
model_score:
4122015.154
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 7, 'min_child_weight': 4, 'n_estimators': 1658, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.6}
model_score:
4491406.966
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.05, 'max_de
pth': 3, 'min_child_weight': 4, 'n_estimators': 495, 'objective': 'reg:
squarederror', 'reg_alpha': 4, 'reg_lambda': 0.4, 'subsample': 0.7}
model_score:
5691473.245
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.35000000000
000003, 'max_depth': 12, 'min_child_weight': 4, 'n_estimators': 561, 'o
bjective': 'reg:squarederror', 'reg_alpha': 6, 'reg_lambda': 0.8, 'subs
ample': 0.7999999999999999}
model_score:
5060406.052
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.45, 'max_de
pth': 6, 'min_child_weight': 2, 'n_estimators': 383, 'objective': 'reg:
squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7}
model_score:
6162069.792
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.2, 'max_dep
th': 9, 'min_child_weight': 4, 'n_estimators': 1031, 'objective': 'reg:
squarederror', 'reg_alpha': 2, 'reg_lambda': 0.9, 'subsample': 0.799999
```

```
9999999999}
model_score:
4734317.94
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.1, 'max_dep
th': 15, 'min_child_weight': 7, 'n_estimators': 699, 'objective': 'reg:
squarederror', 'reg_alpha': 5, 'reg_lambda': 0.1, 'subsample': 0.5}
model_score:
4630045.125
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 6, 'n_estimators': 925, 'objective': 're
g:squarederror', 'reg_alpha': 1, 'reg_lambda': 0.5, 'subsample': 0.7999
999999999999}
model_score:
4132292.726
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.25, 'max_depth': 13, 'min_
child_weight': 4, 'n_estimators': 1803, 'objective': 'reg:pseudohuberer
ror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.7}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.15000000000
000002, 'max_depth': 4, 'min_child_weight': 3, 'n_estimators': 1648, 'o
bjective': 'reg:squarederror', 'reg_alpha': 3, 'reg_lambda': 0.2, 'subs
ample': 0.6}
model_score:
5374802.71
Params testing:
{'colsample_bytree': 0.8000000000000003, 'learning_rate': 0.4, 'max_dep
th': 10, 'min_child_weight': 1, 'n_estimators': 1048, 'objective': 're
g:squaredlogerror', 'reg_alpha': 0, 'reg_lambda': 0.30000000000000004,
'subsample': 0.7999999999999999}
model_score:
22830341.52
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 1055, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.8999
999999999999}
model_score:
4029323.155
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.3, 'max_dep
th': 14, 'min_child_weight': 5, 'n_estimators': 1980, 'objective': 're
g:squarederror', 'reg_alpha': 7, 'reg_lambda': 0.4, 'subsample': 0.8999
999999999999}
model_score:
4514254.495
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.05, 'max_de
pth': 1, 'min_child_weight': 4, 'n_estimators': 767, 'objective': 'reg:
squarederror', 'reg_alpha': 9, 'reg_lambda': 0.6, 'subsample': 0.899999
9999999999}
model_score:
8440114.246
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.2, 'max_dep
th': 11, 'min_child_weight': 7, 'n_estimators': 1318, 'objective': 're
g:pseudohubererror', 'reg_alpha': 8, 'reg_lambda': 0.7000000000000001,
```

```
'subsample': 0.8999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.05, 'max_de
pth': 5, 'min_child_weight': 2, 'n_estimators': 1331, 'objective': 're
g:squaredlogerror', 'reg_alpha': 4, 'reg_lambda': 0.8, 'subsample': 0.8
999999999999999}
model_score:
22840934.717
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.35000000000
000003, 'max_depth': 2, 'min_child_weight': 4, 'n_estimators': 1142, 'o
bjective': 'reg:squarederror', 'reg_alpha': 6, 'reg_lambda': 1.0, 'subs
ample': 0.8999999999999999}
model_score:
6019235.325
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.45, 'max_de
pth': 7, 'min_child_weight': 6, 'n_estimators': 1202, 'objective': 're
g:squarederror', 'reg_alpha': 5, 'reg_lambda': 0.6, 'subsample': 0.8999
999999999999}
model_score:
5509314.761
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.1, 'max_de
pth': 12, 'min_child_weight': 4, 'n_estimators': 1720, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.9, 'subsample': 0.8999
999999999999}
model_score:
4288923.874
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.5, 'max_dep
th': 12, 'min_child_weight': 1, 'n_estimators': 408, 'objective': 'reg:
pseudohubererror', 'reg_alpha': 8, 'reg_lambda': 0.1, 'subsample': 0.79
99999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.25, 'max_de
pth': 3, 'min_child_weight': 3, 'n_estimators': 1630, 'objective': 're
g:squarederror', 'reg_alpha': 1, 'reg_lambda': 0.5, 'subsample': 0.8999
999999999999}
model_score:
5449888.677
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.15000000000
000002, 'max_depth': 8, 'min_child_weight': 5, 'n_estimators': 434, 'ob
jective': 'reg:squaredlogerror', 'reg_alpha': 2, 'reg_lambda': 0.6, 'su
bsample': 0.5}
model_score:
22846842.986
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.05, 'max_de
pth': 9, 'min_child_weight': 4, 'n_estimators': 325, 'objective': 'reg:
squarederror', 'reg_alpha': 0, 'reg_lambda': 0.2, 'subsample': 0.799999
9999999999}
model_score:
4179762.593
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.4, 'max_depth': 6, 'min_ch
```

```
ild_weight': 7, 'n_estimators': 815, 'objective': 'reg:squarederror',
'reg_alpha': 3, 'reg_lambda': 0.30000000000000004, 'subsample': 0.6}
model_score:
6203937.196
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.05, 'max_d
epth': 15, 'min_child_weight': 4, 'n_estimators': 628, 'objective': 're
g:pseudohubererror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.
7999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.7000000000000002, 'learning_rate': 0.3, 'max_dep
th': 13, 'min_child_weight': 2, 'n_estimators': 1395, 'objective': 're
g:squarederror', 'reg_alpha': 7, 'reg_lambda': 0.4, 'subsample': 0.8999
999999999999}
model_score:
4582182.058
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 6, 'n_estimators': 1070, 'objective': 're
g:squaredlogerror', 'reg_alpha': 9, 'reg_lambda': 1.0, 'subsample': 0.7
999999999999999}
model_score:
22846416.106
Params testing:
{'colsample_bytree': 0.5000000000000001, 'learning_rate': 0.2, 'max_dep
th': 14, 'min_child_weight': 4, 'n_estimators': 856, 'objective': 'reg:
squarederror', 'reg_alpha': 4, 'reg_lambda': 0.7000000000000001, 'subsa
mple': 0.5}
model_score:
5226209.006
Params testing:
{'colsample_bytree': 0.4000000000000001, 'learning_rate': 0.5, 'max_dep
th': 4, 'min_child_weight': 1, 'n_estimators': 1069, 'objective': 'reg:
squarederror', 'reg_alpha': 8, 'reg_lambda': 0.6, 'subsample': 0.799999
9999999999}
model_score:
6170911.947
Params testing:
{'colsample_bytree': 0.2, 'learning_rate': 0.35000000000000003, 'max_de
pth': 10, 'min_child_weight': 4, 'n_estimators': 1216, 'objective': 're
g:squarederror', 'reg_alpha': 6, 'reg_lambda': 0.8, 'subsample': 0.8999
999999999999}
model_score:
5296690.171
Params testing:
{'colsample_bytree': 0.6000000000000001, 'learning_rate': 0.05, 'max_de
pth': 12, 'min_child_weight': 3, 'n_estimators': 371, 'objective': 're
g:pseudohubererror', 'reg_alpha': 5, 'reg_lambda': 0.6, 'subsample': 0.
7999999999999999}
model_score:
22848977.764
Params testing:
{'colsample_bytree': 0.30000000000000004, 'learning_rate': 0.45, 'max_d
epth': 11, 'min_child_weight': 4, 'n_estimators': 114, 'objective': 're
g:squarederror', 'reg_alpha': 8, 'reg_lambda': 0.9, 'subsample': 0.6}
model_score:
6389661.546
100%|████████████████████████████████████████████████| 100/100 [4
1:30<00:00, 24.90s/trial, best loss: 4029323.155]
```

```
best:  {'colsample_bytree': 4, 'learning_rate': 0, 'max_depth': 11, 'mi
n_child_weight': 3, 'n_estimators': 955, 'objective': 0, 'reg_alpha':
8, 'reg_lambda': 5, 'subsample': 4}
```

In [33]:

```
best={'colsample_bytree': 4, 'learning_rate': 0, 'max_depth': 11, 'min_child_weight': 3
, 'n_estimators': 955, 'objective': 0, 'reg_alpha': 8, 'reg_lambda': 5, 'subsample': 4}
```

In [34]:

```
xgb_r = xgb.XGBRegressor(tree_method='auto', colsample_bytree=params_grid['colsample_by
tree'][best['colsample_bytree']],learning_rate=params_grid['learning_rate'][best['learn
ing_rate']], max_depth=params_grid['max_depth'][best['max_depth']], min_child_weight=pa
rams_grid['min_child_weight'][best['min_child_weight']], n_estimators=params_grid['n_es
timators'][best['n_estimators']], objective=params_grid['objective'][best['objective'
]], reg_alpha=params_grid['reg_alpha'][best['reg_alpha']], reg_lambda=params_grid['reg_
lambda'][best['reg_lambda']], subsample=params_grid['subsample'][best['subsample']])
tt_xgb = TransformedTargetRegressor(regressor=xgb_r, func=np.sqrt, inverse_func=np.squa
re)
pipe_xgb = Pipeline([('scaler', scaler), ('transformer', tt_xgb)])

cv = float(format((cross_val_score(pipe_xgb,
                                   x_train[important_cols],
                                   y_train,
                                   scoring='neg_mean_absolute_error',
                                   error_score='raise')*-1).mean(),'.3f'))

R2 = float(format(cross_val_score(pipe_xgb,
                                  x_train[important_cols],
                                  y_train).mean(),'.3f'))

evaluation.loc[5] = ['XGBOOST', 'Hyperopt', cv, R2]
```

Out[34]:

	Model	Details	MAE	R2
0	Random Forest	No Hyperopt	4.515881e+06	0.765
1	Random Forest	Hyperopt	4.312057e+06	0.791
2	SVR	No Hyperopt	1.606734e+07	-1.172
3	SVR	Hyperopt	9.761188e+06	0.189
4	XGBOOST	No Hyperopt	5.085187e+06	0.730

In [36]:

```
evaluation
```

Out[36]:

	Model	Details	MAE	R2
0	Random Forest	No Hyperopt	4.515881e+06	0.765
1	Random Forest	Hyperopt	4.312057e+06	0.791
2	SVR	No Hyperopt	1.606734e+07	-1.172
3	SVR	Hyperopt	9.761188e+06	0.189
4	XGBOOST	No Hyperopt	5.085187e+06	0.730
5	XGBOOST	Hyperopt	4.029323e+06	0.807

Améliorations

- Appliquer un **filtrage du bruit** et une **détection des anomalies**.
- Approfondir l'analyse et l'**imputation** des valeurs manquantes.
- Réduction de la dimension par **ACP**.
- Tester **LightGBM, CatBoost** (méthodes récentes et très utilisés lors de compétitions Kaggle).
- Tester les **modèles de mélange** (utiliser les prédictions de différents modèles et puis prédire à nouveau à l'aide d'un dernier modèle à partir des prévisions).
- Utiliser d'autres **modifications du signal** (MFCC/Cepstrum, STA/LTA...).