Kaggle - Housing Prices Competition

Tom GILGENKRANTZ
Charles LAMBERT
Paul GUILMIN

ENSIIE - 2A

March 13, 2025

Contents

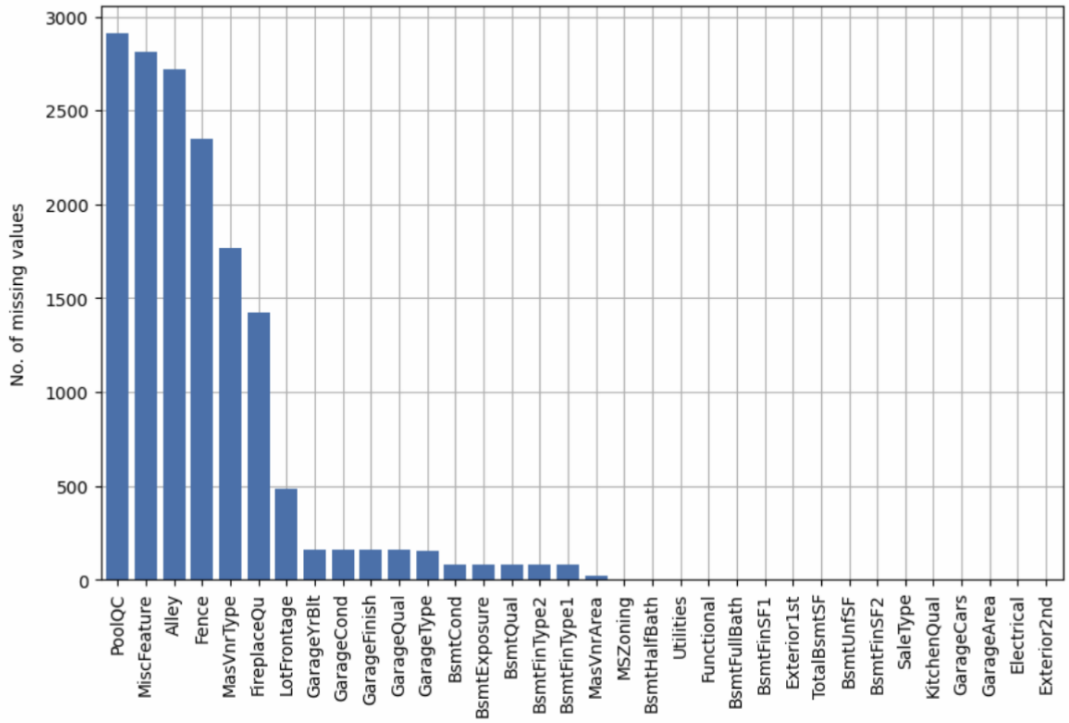
1 Introduction

Le but de ce projet est de prédire les prix de vente de maisons. Pour cela, nous avons à notre disposition un jeu de données d'entraînement contenant des observations à plusieurs variables. Nous avons également un jeu de données de test contenant d'autres observations. Notre objectif est de prédire le prix de vente de chaque maison du jeu de données de test. Pour cela, nous allons utiliser des méthodes de régression linéaire et des méthodes de machine learning. Nous allons également effectuer une analyse exploratoire des données pour mieux comprendre les variables et les relations entre elles.

2 Préparation des données

Avant de commencer à préparer les données, il est nécessaire de fusionner les données d'entraînement et de test pour effectuer les mêmes transformations sur les deux jeux de données.

On constate que certaines colonnes contiennent beaucoup de valeurs manquantes :



Le traitement de celles-ci se fera en fonction du type de la variable. On distingue deux types de variables : les variables catégoriques et les variables numériques.

2.1 Variables catégoriques

On commence par traiter les valeurs manquantes *NA* :

- Si elles sont trop nombreuses (plus de 10), on les remplace par la valeur *None*.
- Sinon, on les remplace par la valeur la plus fréquente au sein du même quartier avec le code suivant :

```
data[features_modefill] = data.groupby("Neighborhood")[features_modefill].transform(lambda x:x.fillna(x.mode()[0]))
```

De plus, les variables *MSSubClass* et *YrSold* sont des variables catégoriques mais sont représentées sous forme de nombres (année et type de maison). On les convertit en chaînes de caractères pour éviter que le modèle ne les traite comme des variables numériques.

2.2 Variables numériques

Les variables *LotFrontage* et *GarageYrBlt* ont de grosses quantités de valeurs manquantes (17.7% et 8%). On les remplace par la médiane des valeurs des maisons du même quartier. Pour les autres variables (qui ne comportent que quelques valeurs manquantes), on les remplace par des zéros.

Aussi, la variable *MoSold* correspond au mois auquel la maison a été vendue, ce qui en fait une variable cyclique de période 12. Pour que la valeur la plus petite (Janvier) apparaisse à côté de la plus grande (Décembre), on se sert de fonction trigonométrique. On transforme alors la colonne en deux variables *MoSoldSin* et *MoSoldCos*. Cela permet notamment de n'ajouter qu'une seule colonne supplémentaires au lieu de 11 si on avait utilisé un OneHotEncoding.

2.3 Ajout de variables

On va maintenant ajouté des variables qui pourraient être utiles pour la prédiction des prix et qui peuvent être simplement déduites des autres variables :

- $TotalArea = GrLiveArea + TotalBsmtSF$
- $TotalBaths = fullBath + BsmtFullBath + \frac{HalfBath}{2} + \frac{BsmtHalfBath}{2}$

- Pareil pour $TotalPorch$

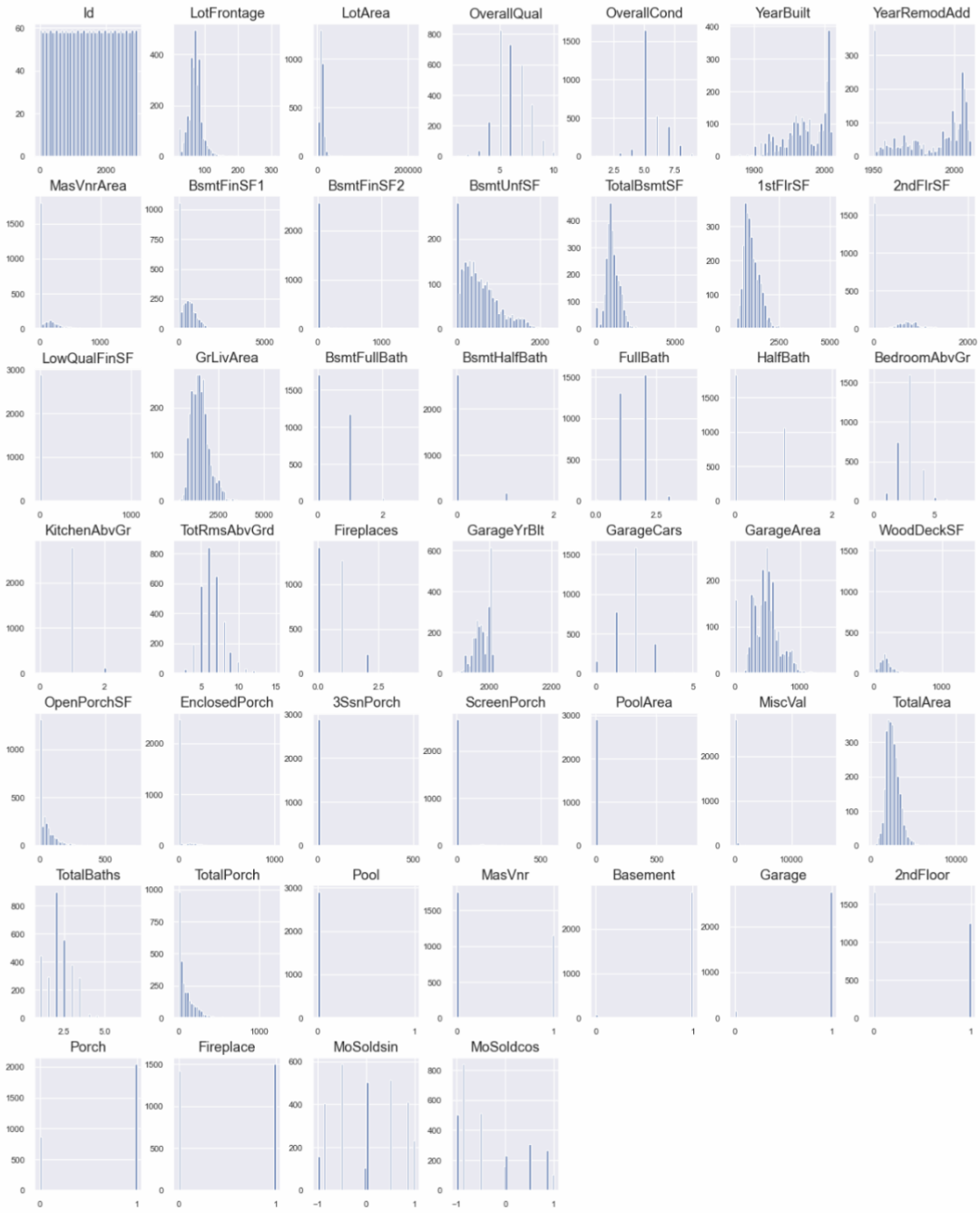
Ces colonnes supplémentaires permettent de mieux représenter les données et d'améliorer la prédiction.

On crée aussi des colonnes binaires pour les variables catégoriques :

- $Pool = \begin{cases} 1 & \text{si } PoolArea > 0 \\ 0 & \text{sinon} \end{cases}$
- Pareil pour l'existance d'une cheminée, d'un $2^{\text{ème}}$ étage, etc...

2.4 Distribution des variables

On remarque que certaines variables ne suivent pas une distribution normale.



On applique alors la transformation de Box-Cox :

$$y(\lambda) = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \text{si } \lambda \neq 0 \\ \log(y) & \text{si } \lambda = 0 \end{cases}$$

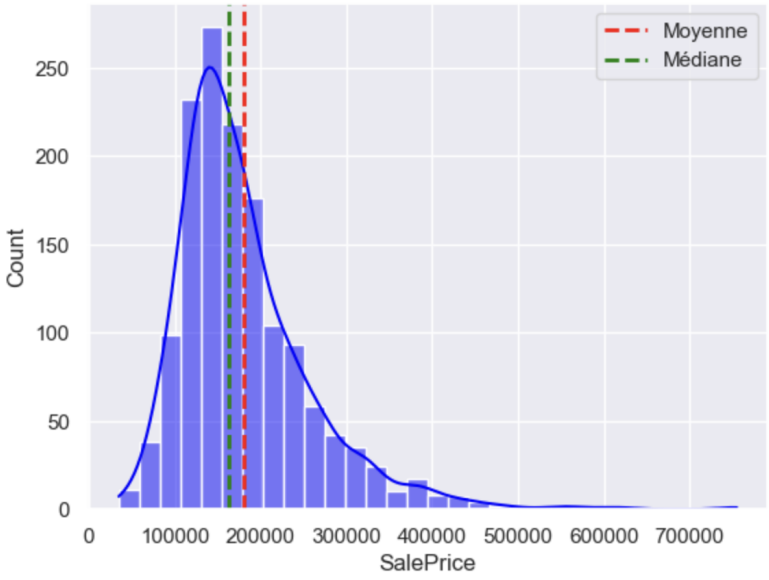
On choisit λ de manière à maximiser la vraisemblance des données. On applique cette transformation aux variables qui ne suivent pas une distribution normale.

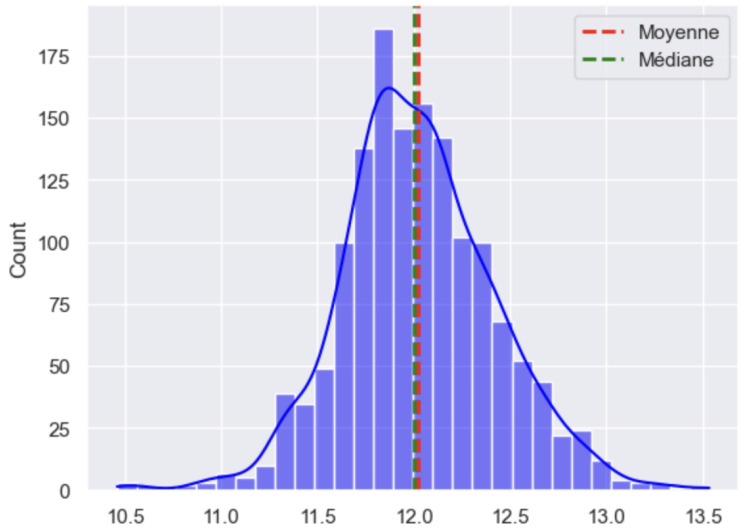
Cette transformation permet de réduire les effets des valeurs extrêmes et de rendre les données plus normales. Elle permet aussi de réduire l'asymétrie des distributions et de stabiliser la variance, ce qui est important pour les modèles de régression linéaire. Cette méthode est plus robuste que la transformation logarithmique car elle permet notamment de traiter les valeurs négative.

Enfin on se sert du Robust Scaler pour endurcir les données contre les valeurs extrêmes :

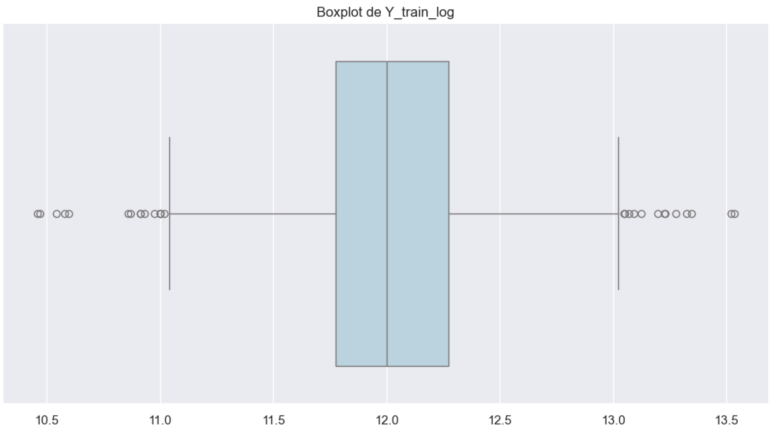
```
cols = data.select_dtypes(np.number).columns
data[cols] = RobustScaler().fit_transform(data[cols])
```

Pour le prix de vente, une transformation logarithmique suffit pour obtenir une distribution normale :



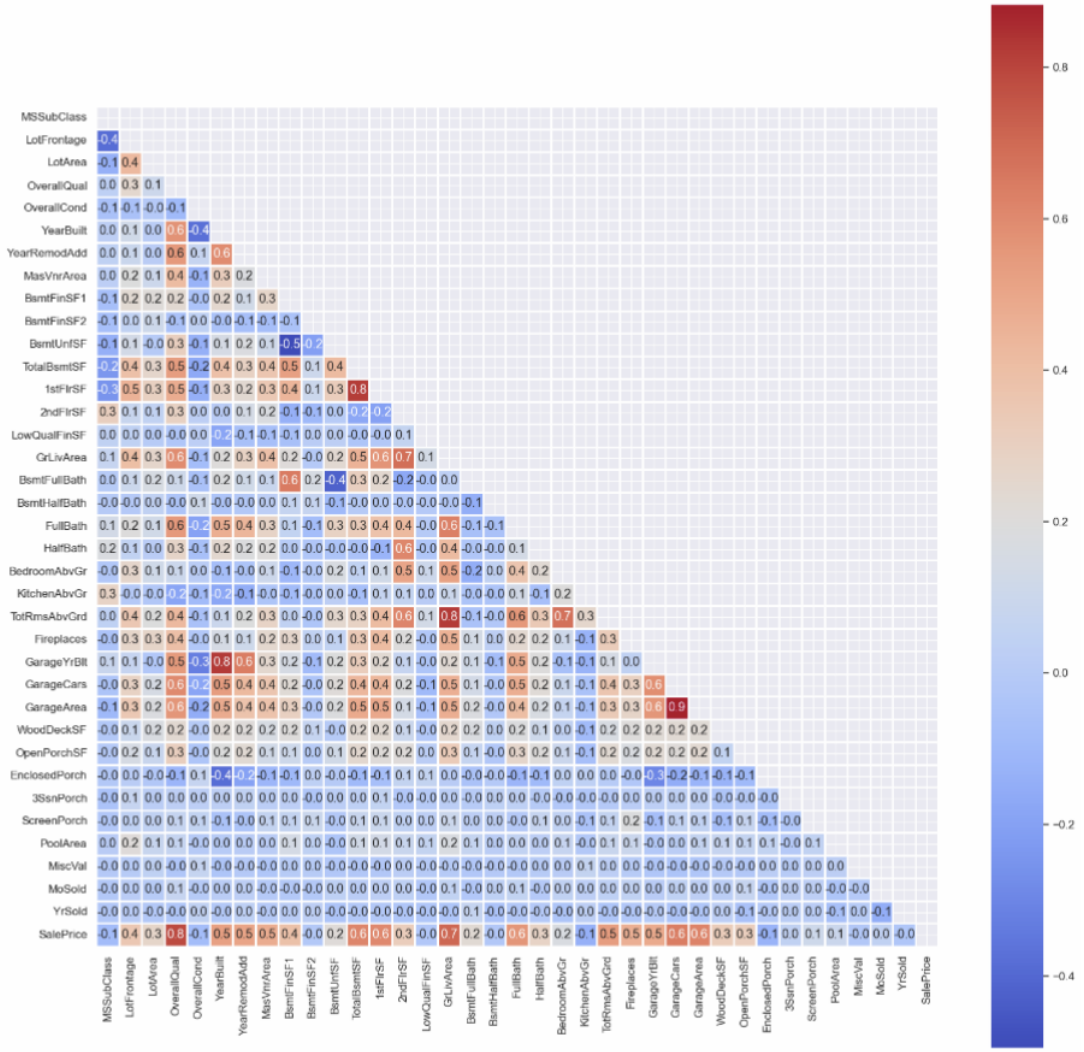


On remarque qu'il reste quelques outliers mais ce n'est pas problématique étant donné qu'on utilise des modèles robustes.

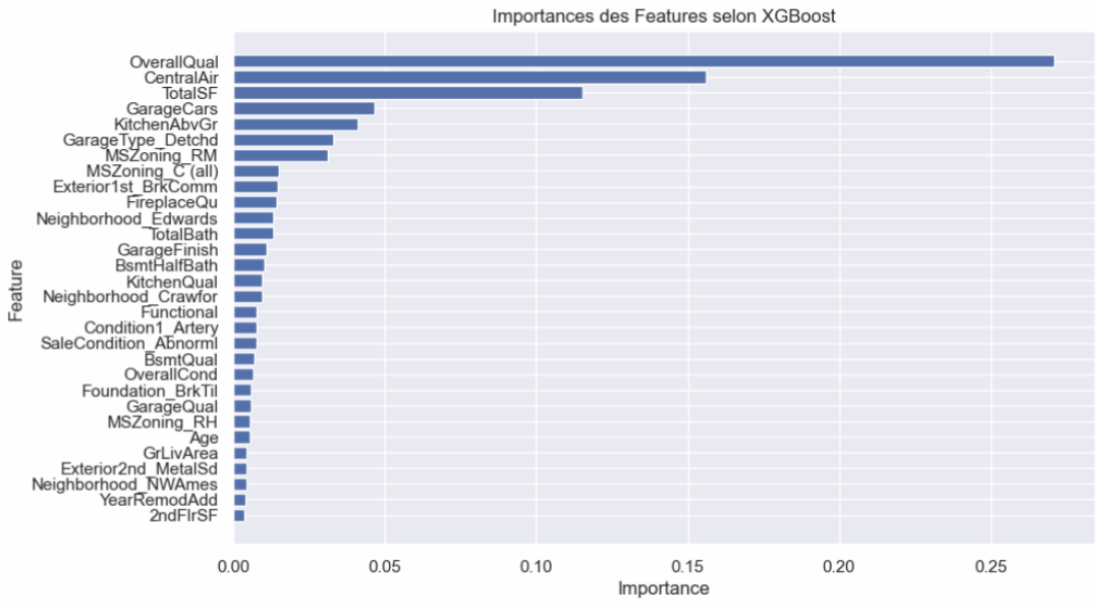


2.5 Sélection des variables

La matrice de corrélation permet de voir les relations entre les variables :



On se sert aussi du graphique des importances des variables selon le modèle XGBoost :



Cela permet de supprimer des variables trop fortement corrélées entre elles (en valeur absolue) et de ne garder que les variables les plus importantes pour la prédiction.

3 Modèles de régression

On se sert de plusieurs modèles de régression pour prédire les prix de vente des maisons. On utilise la validation croisée pour évaluer les performances des modèles (CV = 5). On utilise également la méthode de stacking pour combiner les modèles et obtenir une meilleure prédiction.

3.1 Optimisation des hyperparamètres

Dans tous les modèles utilisés, des hyperparamètres entrent en jeu pour régler la complexité du modèle. On utilise les méthodes de GridSearchCV ou RandomizedSearchCV pour trouver les meilleurs hyperparamètres. Ces deux méthodes permettent de tester plusieurs combinaisons d'hyperparamètres et de choisir celle qui donne la meilleure performance.

3.2 Support Vector Regressor

Le modèle Support Vector Regressor (SVB) est un modèle de régression qui se base sur les vecteurs de support pour prédire les valeurs. Le Kernel utilisé est le

Radial Basis Function (RBF) car il permet de modéliser des relations non linéaires complexes entre les variables. Les hyperparamètres du modèle sont les suivants :

- **C** : Paramètre de régularisation qui contrôle le compromis entre la complexité du modèle et la marge d'erreur. Une petite valeur ($\simeq 1$) impose une forte régularisation ce qui aboutit à un modèle plus simple mais potentiellement en underfitting. A l'inverse, une grande valeur ($\simeq 100$) impose une faible régularisation ce qui aboutit à un modèle plus complexe mais potentiellement en overfitting.
- Γ : Paramètre du noyau RBF qui contrôle l'influence des points individuels. Il définit à quelle distance les points d'entraînement ont un impact sur la prédiction. Une petite valeur ($\simeq 10^{-5}$) signifie que les points d'entraînement ont un impact sur une zone plus large ce qui aboutit à un modèle plus lisse mais potentiellement en underfitting. A l'inverse, une grande valeur ($\simeq 10^{-3}$) signifie que les points d'entraînement ont un impact sur une zone plus petite ce qui aboutit à un modèle plus complexe mais potentiellement en overfitting.
- ϵ : Marge d'erreur que le modèle accepte autour d'un point. Une petite valeur ($\simeq 10^{-2}$) signifie que le modèle accepte une faible marge d'erreur ce qui aboutit à un modèle potentiellement en overfitting. A l'inverse, une grande valeur ($\simeq 10^{-1}$) signifie que le modèle accepte une grande marge d'erreur ce qui aboutit à un modèle potentiellement en underfitting.

Après optimisation des hyperparamètres, on obtient les valeurs suivantes :

- $\mathbf{C} = 45$
- $\Gamma = 9.081 \times 10^{-5}$
- $\epsilon = 0.0039$

3.3 XGBoost

Le modèle XGBoost est un modèle de machine learning qui se base sur des arbres de décision pour prédire les valeurs. Les hyperparamètres du modèle sont les suivants :

- **n_estimators** : Nombre d'arbres de décision (ou modèles faibles) dans le modèle. Plus il y en a, plus le modèle est complexe, mais cela augmente aussi le risque d'overfitting ainsi que le temps de calcul.
- **max_depth** : Profondeur maximale des arbres de décision. Tout comme le nombre d'estimateurs, plus elle est grande, plus le modèle est complexe, mais cela augmente aussi le risque d'overfitting.

- **learning_rate** : Taux d'apprentissage qui contrôle la contribution de chaque arbre de décision. Une petite valeur de taux d'apprentissage implique de construire plus d'arbes pour converger vers la solution optimale, ce qui augmente le temps de calcul.
- **min_child_weight** : Poids minimum nécessaire pour créer un nouvel arbre de décision. Une grande valeur de ce paramètre empêche de créer des feuilles trop spécifiques, ce qui peut réduire le risque d'overfitting.

Après optimisation des hyperparamètres, on obtient les valeurs suivantes :

- **n_estimators** = 800
- **max_depth** = 3
- **learning_rate** = 0.05
- **min_child_weight** = 1

3.4 Light Gradient Boosting Machine

Le modèle Light Gradient Boosting Machine (LGBM) est un modèle de machine learning qui se base sur des arbres de décision pour prédire les valeurs. Les hyperparamètres du modèle sont les suivants :

- **colsample_bytree** : Fraction de colonnes à utiliser pour chaque arbre de décision. Une petite valeur de ce paramètre permet de réduire le risque d'overfitting en limitant la diversité des caractéristiques utilisées. A l'inverse, une grande valeur permet de construire des arbres plus complexes et donc potentiellement en overfitting.
- **learning_rate** : Taux d'apprentissage qui contrôle la contribution de chaque arbre au modèle final. Une grande valeur permet d'accélérer la convergence du modèle mais augmente le risque d'overfitting.

Après optimisation des hyperparamètres, on obtient les valeurs suivantes :

- **colsample_bytree** = 0.3 : ici seulement 30% des colonnes sont utilisées pour chaque arbre de décision, ce qui permet de réduire le risque d'overfitting en diversifiant les caractéristiques utilisées.
- **learning_rate** = 0.83 : : la valeur est modérement élevée, cela signifie que chaque arbre de décision contribue de manière significative à la prédiction finale.

3.5 Gradient Boosting

Le modèle Gradient Boosting est un modèle de machine learning qui se base sur des arbres de décision pour prédire les valeurs. Les hyperparamètres du modèle sont les suivants :

- **max_features** : Nombre de caractéristiques à considérer lors de la recherche de la meilleure partition. Une petite valeur de ce paramètre permet de réduire le risque d'overfitting en limitant la diversité des caractéristiques utilisées. A l'inverse, une grande valeur permet de construire des arbres plus complexes et donc potentiellement en overfitting.
- **learning_rate** : C'est le même hyperparamètre de vitesse d'apprentissage que pour le modèle LGBM.

Après optimisation des hyperparamètres, on obtient les valeurs suivantes :

- **max_features** $= 0.2$: ici seulement 20% des colonnes sont utilisées pour chaque arbre de décision, ce qui permet de réduire le risque d'overfitting en diversifiant les caractéristiques utilisées.
- **learning_rate** $= 0.83$: la valeur est modérement élevée, cela signifie que chaque arbre de décision contribue de manière significative à la prédiction finale.

3.6 Ridge

Le modèle Ridge est un modèle de régression linéaire qui se base sur la régression linéaire avec une régularisation L_2. L'hyperparamètre du modèle est le suivant :

- α : Paramètre de régularisation qui contrôle le compromis entre la complexité du modèle et la marge d'erreur. Une petite valeur ($\simeq 1$) impose une forte régularisation ce qui aboutit à un modèle plus simple mais potentiellement en underfitting. A l'inverse, une grande valeur ($\simeq 100$) impose une faible régularisation ce qui aboutit à un modèle plus complexe mais potentiellement en overfitting.

Après optimisation des hyperparamètres, on obtient la valeur suivante :

- $\alpha = 13.62$: la régularisation est forte, ce qui signifie que le modèle réduit forcément la complexité pour éviter l'overfitting.

3.7 Lasso

Le modèle Lasso est un modèle de régression linéaire qui se base sur la régression linéaire. La différence avec le Ridge est que la régularisation est L_1. L'hyperparamètre du modèle est le suivant :

- α : Paramètre de régularisation qui contrôle le compromis entre la complexité du modèle et la marge d'erreur. Une petite valeur ($\simeq 1$) impose une forte régularisation ce qui aboutit à un modèle plus simple mais potentiellement en underfitting. A l'inverse, une grande valeur ($\simeq 100$) impose une faible régularisation ce qui aboutit à un modèle plus complexe mais potentiellement en overfitting.

Après optimisation des hyperparamètres, on obtient la valeur suivante :

- $\alpha = 4.3 \times 10^{-4}$: la valeur indique que le modèle agit quasiment comme une régression linéaire classique.

4 Stacking des modèles

4.1 Principe

Le Stacking (ou Super Learning) est une méthode d'apprentissage automatique qui combine plusieurs modèles de régression pour améliorer la prédiction. On utilise un modèle de régression linéaire pour combiner les prédictions des modèles de base. On utilise la validation croisée pour évaluer les performances du modèle de stacking. Voici chacune des étapes du stacking :

- **Etape 1 :** On entraine chacun de nos modèles de base (XGBoost, Ridge, Lasso, etc...) sur les données d'entraînement. Chacun de ces modèles va proposer une prédiction des prix de vente des maisons du jeu de données de test.

- **Etape 2 :** On utilise ces prédictions comme variables d'entrée pour notre modèle de stacking. On entraine ce modèle de stacking sur les prédictions des modèles de base. Par exemple, on a dans notre cas 6 modèles de base donc notre nouveau set de données comportera 6 colonnes.

- **Etape 3 :** On utilise un modèle de régression linéaire pour combiner les prédictions des modèles de base. On entraine ce modèle de régression linéaire sur les prédictions des modèles de base.

4.2 Choix du Meta-Model

Le rôle du meta-model est de combiner les prédictions des modèles de base pour obtenir une prédiction finale. On a choisi le modèle Ridge pour ces raisons :

- Gestion de la colinéarité : les prédictions des modèles de base sont souvent fortement corrélées (par exemple XGBoost et LGBM). Ridge
- Simplicité : Ridge est rapide à entrainer et fonctionne efficacement même avec peu de données.
- Régularisation : le paramètre α contrôle la force de régulation L_2, ce qui réduit le risque d'overfitting

5 Résultats

Voici notre classement Kaggle sur le concours **Housing Prices Competition for Kaggle Learn Users** :

Figure 1: Screenshot pris le Jeudi 13 Mars à 08h28

La métrique utilisée est le Root Mean Squared Logarithmic Error (RMSLE) et le concours comptait 6961 participants. Notre équipe appartient donc au top 0.8% des participants !

6 Annexes

Importation des datasets:

```
train_df=pd.read_csv("train.csv")
test_df=pd.read_csv("test.csv")
```

On compte le nombre de valeurs manquantes pour les variables numériques

```
null_counts=train_df.isnull().sum()
null_columns = null_counts[null_counts > 0]
null_columns2 = null_columns.drop(["Alley","MasVnrType","MasVnrArea","GarageYrBlt","BsmtQual","BsmtCond","BsmtExposure","BsmtFinType1",
"BsmtFinType2","FireplaceQu","GarageType","GarageFinish","GarageQual","GarageCond","PoolQC","Fence","MiscFeature"])

print(null_columns2)
```

On concatène les datasets pour effectuer les modifications sur le set de test et d'entraînement en même temps.

```
data = pd.concat([train_df.drop("SalePrice",axis=1),test_df]) #Drop la variable cible
```

```
null_counts=data.isna().sum()
null_column = null_counts[null_counts > 0]

categorical_NA=["Alley","BsmtQual","BsmtCond","BsmtExposure","BsmtFinType1","BsmtFinType2","FireplaceQu","GarageType","GarageFinish",
"GarageQual","GarageCond","PoolQC","Fence","MiscFeature"]

null_column = null_column.drop(categorical_NA)

#print(null_column)
null_columns=["MasVnrType", "MSZoning", "Utilities", "Exterior1st", "Exterior2nd", "SaleType", "Electrical", "KitchenQual", "Functional"]
na_numerical=null_column.drop(null_columns)
print(na_numerical)
```

On remplace les NA des valeurs catégoriques par None pour pas induire en erreur les algorithmes

```
data[categorical_NA]=data[categorical_NA].fillna("None")
```

Python

On rempli avec le mode le plus présent par quartiers pour chaque valeur manquante

```
data[null_columns] = data.groupby("Neighborhood")[null_columns].transform(
    lambda x: x.fillna(x.mode().iloc[0] if not x.mode().empty else x) #Mode permet de remplir avec la valeur la plus présente parmis les quartiers
)
```

Python

Maintenant on va travailler sur les valeurs numériques manquantes LotFrontage=> 17.7% de valeurs manquantes -> pas besoin de KNN juste mettre la médiane par quartiers suffit GarageArea idem

```
numerical_median=["LotFrontage","GarageYrBlt"]
data[numerical_median] = data.groupby("Neighborhood")[numerical_median].transform(lambda x: x.fillna(x.median()))
```

Python

Très peu de valeurs manquantes => on met directement la valeur à 0

```
features_zerofill = ["GarageArea", "MasVnrArea", "BsmtHalfBath", "BsmtFullBath", "BsmtFinSF1", "BsmtFinSF2", "BsmtUnfSF", "TotalBsmtSF", "GarageCa
data[features_zerofill] = data[features_zerofill].fillna(0)
```

Python

On créé de nouvelles colonnes pour avoir des infos plus significatives

ICi on aura le nombre total

```
data["TotalArea"] = data["GrLivArea"] + data["TotalBsmtSF"] #Surface totale
data["TotalBaths"] = data["FullBath"] + data["BsmtFullBath"] + 0.5*(data["HalfBath"]+data["BsmtHalfBath"]) #Nombre total de salles de bain
data["TotalPorch"] = data["OpenPorchSF"] + data["EnclosedPorch"] + data["3SsnPorch"] + data["ScreenPorch"] #Nombre total de porche
```

Python

ICi on note la présence ou non de certains éléments

```
data['Pool'] = data['PoolArea'].apply(lambda x: 1 if x > 0 else 0) #Présence d'une piscine
data['MasVnr'] = data['MasVnrArea'].apply(lambda x: 1 if x > 0 else 0) #Présence de placage
data['Basement'] = data['TotalBsmtSF'].apply(lambda x: 1 if x > 0 else 0) #Présence d'un sous-sol
data['Garage'] = data['GarageArea'].apply(lambda x: 1 if x > 0 else 0) #Présence d'un garage
data['2ndFloor'] = data['2ndFlrSF'].apply(lambda x: 1 if x > 0 else 0) #Présence d'un 2nd étage
data['Porch'] = data['TotalPorch'].apply(lambda x: 1 if x > 0 else 0) #Présence d'un porche
data['Fireplace'] = data['Fireplaces'].apply(lambda x: 1 if x > 0 else 0) #Présence d'une cheminée
```

On remarque que MSSubClass et YrSold sont en fait des variables catégoriques, on doit donc les changer en type catégorique

```
data[["YrSold","MSSubClass"]]=data[["YrSold","MSSubClass"]].astype("category")
```

Valeurs cycliques à traiter MoSold_sin=sin(2π×MoSold/12) MoSold_ cos=cos(2π×MoSold/12)

```
data["MoSoldsin"] = np.sin(2 * np.pi * data["MoSold"] / 12)
data["MoSoldcos"] = np.cos(2 * np.pi * data["MoSold"] / 12) #Ajout de 2 nouvelles colonnes
data = data.drop("MoSold", axis=1) #On enlève MoSold
```

On a plusieurs variables qui ne sont pas distribuées normalement On les normalise

```
non_normalize=['LotFrontage', 'LotArea', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2','BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'Gr
non_normal_croiss = np.abs(data[non_normalize].apply(lambda x: skew(x)).sort_values(
    ascending=False))

non_normal_high = non_normal_croiss[non_normal_croiss > 0.3]

non_normal_index = non_normal_high.index

for i in non_normal_index:
    if (data[i] < 0).any():
        print(f"Attention : {i} contient des valeurs négatives !")
    if (data[i] == 0).any():
        print(f"Attention : {i} contient des valeurs nulles !")

for i in non_normal_index:
    data[i] = boxcox1p(data[i], boxcox_normmax(data[i] + 1)) #Boxcox permet de réduire l'asymétrie de façon efficace
```

On a plusieurs variables qui ne sont pas distribuées normalement On les normalise

```
non_normalize=['LotFrontage', 'LotArea', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2','BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF',
'GrLivArea','GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', 'LowQualFinSF', 'MiscVal']
non_normal_croiss = np.abs(data[non_normalize].apply(lambda x: skew(x)).sort_values(
    ascending=False))

non_normal_high = non_normal_croiss[non_normal_croiss > 0.3]

non_normal_index = non_normal_high.index

for i in non_normal_index:
    if (data[i] < 0).any():
        print(f"Attention : {i} contient des valeurs négatives !")
    if (data[i] == 0).any():
        print(f"Attention : {i} contient des valeurs nulles !")

for i in non_normal_index:
    data[i] = boxcox1p(data[i], boxcox_normmax(data[i] + 1)) #Boxcox permet de réduire l'asymétrie de façon efficace
```

Robust Scaling=> endurcir nos algorithmes face aux valeurs aberrantes

```
cols = data.select_dtypes(np.number).columns
data[cols] = RobustScaler().fit_transform(data[cols])
```

Sélection des variables en fonction de leurs importances avec XGB

```
from sklearn.feature_selection import SelectFromModel
best_xgb = xgb_search.best_estimator_
# Sélection basée sur l'importance des caractéristiques (exemple avec XGBoost)
selector = SelectFromModel(best_xgb, threshold='median')
selector.fit(X_train, y)
X_train_selected = selector.transform(X_train)
X_test_selected = selector.transform(X_test)
```