DECEMBRE 2024

Rapport projet IA

Projet valeurs foncière biens immobilier

Presented to

Mr Yamak

Presented by

Gabin Niel

ESIGELEC
ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

Compte rendu projet IA

Il nous a été demandé de répondre à un exercice basé sur les algorithmes d'apprentissage.
Le but de cet exercice est de déterminer, à l'aide d'un dataset fourni par le gouvernement sur les biens immobiliers, la valeur foncière des biens immobiliers en fonction de plusieurs paramètres tels que la superficie, le nombre de pièces, la localisation, le type de local...
Il fallait, dans un premier temps, suivre plusieurs étapes.

Dans ce document explicatif, vous retrouverez tous les éléments qui vous permettront de comprendre mes choix pour le prétraitement, le choix du modèle, des différents graphiques et tests, ainsi que les décisions pour la fonction finale.

J'ai dû recommencer ce projet plusieurs fois en raison de difficultés liées au prétraitement des données et à une compréhension initiale insuffisante du dataset. Voici les étapes clés et les réflexions qui ont guidé mon travail :
1) Approche initiale : utilisation de toutes les colonnes
Dans un premier temps, j'ai choisi de conserver l'ensemble des colonnes du dataset. Pour gérer les valeurs manquantes, j'ai créé des colonnes "missing" prenant la valeur 1 lorsqu'une donnée était manquante dans les colonnes d'origine.

- Avantage : Cette méthode me permettait de conserver 305 000 lignes et environ 70 colonnes, formant un ensemble de données conséquent. Cela semblait idéal pour exploiter les capacités des algorithmes de deep learning.
- Problème : Malgré ce volume de données, les résultats obtenus avec le deep learning et le machine learning étaient médiocres, suggérant une inadéquation entre le modèle et les données.

2) Analyse approfondie de la colonne "valeur foncière"
En approfondissant l'analyse, j'ai remarqué que la colonne "valeur foncière" représentait le montant de la transaction et non la valeur réelle du bien. Cette incohérence biaisait les résultats des modèles.

- Solution envisagée : J'ai alors décidé de créer une nouvelle colonne, valeur_foncière_bien en fonction de la surface réelle bâtie.
- Problème : Cette approche réduisait considérablement le volume des données utilisables à seulement 30 000 lignes, ce qui était insuffisant pour réaliser efficacement un modèle de deep learning.

Enfin, après plusieurs essais et erreurs, j'ai élaboré un nouveau prétraitement, celui que j'applique actuellement à mon modèle.

Etape 1 : Import du DataSet

```
import pandas as pd
df = pd.read_csv('data_immobiliers.csv')
df.head()
```

Tout d'abord l'importation
Une fois le fichier importé, il faut le travailler, ce qui nous amène à l'une des étapes les plus cruciales : l'analyse du dataset.
En effet, il est crucial de comprendre les données et leurs relations entre elles avant de les prétraiter. On doit voir les différents éléments, comprendre leurs relations entre eux et avec la cible, c'est-à-dire la valeur foncière.
J'ai donc analysé le jeu de données fourni par le gouvernement. Il y avait 305 000 lignes et 41 colonnes.

```
df.columns
```

```
Index(['id_mutation', 'date_mutation', 'numero_disposition', 'nature_mutation',
       'valeur_fonciere', 'adresse_numero', 'adresse_suffixe',
       'adresse_nom_voie', 'adresse_code_voie', 'code_postal', 'code_commune',
       'nom_commune', 'code_departement', 'ancien_code_commune',
       'ancien_nom_commune', 'id_parcelle', 'ancien_id_parcelle',
       'numero_volume', 'lot1_numero', 'lot1_surface_carrez', 'lot2_numero',
       'lot2_surface_carrez', 'lot3_numero', 'lot3_surface_carrez',
       'lot4_numero', 'lot4_surface_carrez', 'lot5_numero',
       'lot5_surface_carrez', 'nombre_lots', 'code_type_local', 'type_local',
       'surface_reelle_bati', 'nombre_pieces_principales',
       'code_nature_culture', 'nature_culture', 'code_nature_culture_speciale',
       'nature_culture_speciale', 'surface_terrain', 'longitude', 'latitude',
       'section_prefixe'],
      dtype='object')
```

Nous allons reprendre point par point les colonnes nécessaires selon moi pour l'élaboration du modèle d'apprentissage.

1) Id_mutation : C'est un élément important. En effet, lors de l'analyse du dataset, j'ai remarqué que des lignes se ressemblaient sur beaucoup de caractéristiques : la date, la ville, la rue, l'id_mutation, la valeur foncière, et d'autres.
En effet, une ligne ne correspond pas à un bien avec une valeur foncière, mais une ligne est un élément d'une transaction qui partage le même id_mutation. Cela signifie qu'on peut avoir 5 fois la même valeur foncière. C'est donc la valeur foncière de la transaction. Il faudra donc, plus tard, rassembler les éléments par transaction et non par ligne, mais nous verrons ça dans le pré-traitement.
2) date_mutation : Pas utile élément qui est unique par transaction
3) numero_disposition : Pas utile, numéro administratif sans lien direct avec la valeur foncière.
4) nature_mutation : Utile, type de transaction
5) valeur_fonciere : Cible à prédire
6) adresse_numero : Pas utile
7) adresse_suffixe : Pas utile, Suffixe de rue, rarement pertinent pour la valeur
8) adresse_nom_voie : Utile nom de rue précis
9) adresse_code_voie : Pas utile, code administratif peu pertinent
10) code_postal : Pas utile, déjà code_commune
11) code_commune : Utile, code de la commune
12) nom_commune : Pas utile, redondant avec code_commune
13) code_departement : Pas utile, déjà code_commune
14) ancien_code_commune : Pas utile, ancien code
15) ancien_nom_commune : Pas utile, ancien nom
16) id_parcelle : Pas utile, identifiant unique
17) ancien_id_parcelle : Pas utile, ancien id_pacerelle
18) numero_volume : Pas utile, spécifique à certaines transactions, rarement pertinent
19) lot1_numero : Pas utile, administratif
20) lot1_surface_carrez : Utile, surface habitable impacte directement la valeur

21) lot2_numero : Pas utile
22) lot2_surface_carrez : Utile, surface habitable impacte directement la valeur
23) lot3_numero : Pas utile
24) lot3_surface_carrez : Utile, surface habitable impacte directement la valeur
25) lot4_numero : Pas utile
2**6) lot4_surface_carrez** : Utile, surface habitable impacte directement la valeur
27) lot5_numero : Pas utile
28) lot5_surface_carrez : Utile, surface habitable impacte directement la valeur
29) nombre_lots : Utile, plusieurs lots peuvent affecter la valeur
30) code_type_local : Utile, indique le type de bien
31) type_local : Pas utile, redondant avec code_type_local
32) surface_reelle_bati : Utile, la surface bâtie est cruciale pour estimer la valeur
33) nombre_pieces_principales : Utile. (Le nombre de pièces est un critère essentiel).
34) code_nature_culture : Pas utile, pertinent uniquement pour les terrains agricoles
35) nature_culture : Utile, pertinent uniquement pour les terrains agricoles
36) code_nature_culture_speciale : Pas utile, spécifique aux terrains spéciaux
37) nature_culture_speciale : Pas utile, spécifique aux terrains spéciaux
38) surface_terrain : Utile, la taille du terrain influence la valeur
39) longitude : Utile, coordonnées géographiques pour localisation précise
40) latitude : Utile, coordonnées géographiques pour localisation précise
41) section_prefixe : Pas utile

Maintenant que j'ai présenté tous les éléments que je considère utiles ou non, je débute le prétraitement.

Etape 2 : Traitement de données

Tout d'abord, il est d'usage de faire .info() sur le dataset pour obtenir des informations importantes, notamment tous les types d'éléments, mais aussi le nombre de valeurs non nuls par colonne, ce qui est très utile.
Ensuite, on fait un .describe() pour avoir une description du dataset, avec le nombre d'éléments non nuls par colonne, mais aussi le nombre d'éléments uniques, la fréquence d'apparition des éléments, etc.
Maintenant voici la liste des éléments que j'ai décidé de conserver pour mon dataset :
surface_reelle_bati, nombre_pieces_principales, surface_terrain
code_type_local, id_mutation,
nature_mutation, valeur_foncière,
adresse_nom_voie, code_commune,
lot1_surface_carrez, lot2_surface_carrez,
lot3_surface_carrez,
lot4_surface_carrez, lot5_surface_carrez
nombre_lots, longitude, latitude,nature_culture

```
colonne_keep = ["surface_reelle_

df = df[colonne_keep]
```

Il nous reste maintenant 18 colonnes

```
print(len(df.columns))
```

```
18
```

Ensuite, il faut lister les éléments nuls pour savoir quelles colonnes il faut compléter, car il faut savoir que les modèles de régression linéaire et de réseaux de neurones ne prennent pas en compte les éléments nuls ou NaN. De ce fait, il faut compléter ces colonnes.

```
df.isnull().sum()
```

```
surface_reelle_bati          157369
nombre_pieces_principales     81487
surface_terrain               92008
code_type_local               81271
id_mutation                       0
nature_mutation                   0
valeur_fonciere                2929
adresse_nom_voie               2442
code_commune                      0
lot1_surface_carrez          282269
lot2_surface_carrez          296747
lot3_surface_carrez          304313
lot4_surface_carrez          305097
lot5_surface_carrez          305216
nombre_lots                       0
longitude                      7997
latitude                       7997
nature_culture                92003
dtype: int64
```

Comme on peut le constater, il y a énormément de valeurs nulles.

Lors de l'analyse du jeu de données, j'ai remarqué que les lots_surface_carrez sont plus précis que les surface_reelle_bati , ce qui m'a poussé à créer une fonction permettant de remplir la colonne surface_reelle_bati par les lots_surface_carrez en priorité et, s'il n' y en a pas, par les surfaces déjà présentes dans surface_reelle_bati . Cela permet de remplir correctement les surfaces de la colonne surface_reelle_bati et de compléter les éléments nulles

```
def traiter_surface_reelle_bati(df, col_cible, colonnes_surface, groupby_col):

    df[col_cible] = df[col_cible].replace(['', 'NA', 'na', 'N/A'], pd.NA)

    df[col_cible] = pd.to_numeric(df[col_cible], errors='coerce')

    for col in colonnes_surface:
        df[col_cible] = df[col_cible].combine_first(df[col])

    df[col_cible] = pd.to_numeric(df[col_cible], errors='coerce')

    if df[col_cible].isnull().sum() > 0:
        moyennes = df.groupby(groupby_col)[col_cible].transform('mean')

        df[col_cible] = df[col_cible].fillna(moyennes)

    return df

# Exemple d'utilisation
colonnes_surface = ['lot1_surface_carrez','lot2_surface_carrez','lot3_surface_carrez','lot4_surface_carrez','lot5_surface_carrez']


df = traiter_surface_reelle_bati(df, 'surface_reelle_bati', colonnes_surface, 'code_type_local')
```

Il reste après cette fonction 80 000 valeurs nulles dans la colonne surface_reelle_bati

Ensuite, il faut remplir les colonnes en faisant la moyenne des éléments. Cependant, faire des moyennes en fonction des autres lignes n'a pas vraiment de sens. Pour préciser la moyenne, il faut rassembler les éléments par groupby, par exemple faire la moyenne des latitudes et longitudes en fonction de ‘adresse_nom_voie’, permettra d'être bien plus précis, au lieu d'utiliser une moyenne générale.

```
def clean_and_fill_column(df, column_to_clean, group_column):

    df[column_to_clean] = df[column_to_clean].replace(['', 'NA', 'na', 'N/A'], pd.NA)

    df[column_to_clean] = pd.to_numeric(df[column_to_clean], errors='coerce')

    print(f"Valeurs manquantes avant le remplissage de '{column_to_clean}' : {df[column_to_clean].isnull().sum()}")

    moyennes = df.groupby(group_column)[column_to_clean].transform('mean')

    df[column_to_clean] = df[column_to_clean].fillna(moyennes)

    print(f"Valeurs manquantes après le remplissage de '{column_to_clean}' : {df[column_to_clean].isnull().sum()}")

    return df
clean_and_fill_column(df,'latitude', 'adresse_nom_voie')
clean_and_fill_column(df,'longitude', 'adresse_nom_voie')
clean_and_fill_column(df,'surface_terrain', 'nombre_lots')
```

```
Valeurs manquantes avant le remplissage de 'latitude' : 8003
Valeurs manquantes après le remplissage de 'latitude' : 783
Valeurs manquantes avant le remplissage de 'longitude' : 8003
Valeurs manquantes après le remplissage de 'longitude' : 783
Valeurs manquantes avant le remplissage de 'surface_terrain' : 92014
Valeurs manquantes après le remplissage de 'surface_terrain' : 25984
```

Comme on peut le voir, les colonnes latitude , longitude et surface_terrain ont vu leurs éléments nuls diminuer.

Ensuite, on passe à la conversion des éléments du jeu de données.
La conversion des colonnes et des fonctionnalités est primordiale pour les incorporer dans le modèle de régression linéaire et dans le réseau de neurones. En effet, ces modèles ne prennent pas en compte les types d'objets , il est nécessaire de les convertir.

Etape 3 : La conversion

Tout d'abord, il faut convertir toutes les colonnes de type objet contenant des valeurs numeriques en type numerique

```
def conversion_num(df,columns_list):
    for col in columns_list:
       df[col] = pd.to_numeric(df[col], errors='coerce')
    return df

columns_list=['valeur_fonciere','lot1_surface_carrez','lot2_surface_carrez','l
df= conversion_num(df,columns_list)
```

Ensuite, il faut convertir tous les éléments objets avec des string en utilisant LabelEncoder() .
Cela permet d'attribuer à chaque chaîne d'éléments une valeur entière.

```
from sklearn.preprocessing import LabelEncoder

def label_encoder(df,object_columns):
    label_encoder = LabelEncoder()
    for col in object_columns:
           df[col] = label_encoder.fit_transform(df[col].astype(str))
    return df

object_columns = ['id_mutation','nature_mutation','adresse_nom_voie','nombre_lots']

df=label_encoder(df,object_columns)
```

Ensuite on vérifie le type des données

```
X = df[[
    'id_mutation',
    'nature_mutation',
    'valeur_fonciere',
    'adresse_nom_voie',
    'code_commune',
    'lot1_surface_carrez',
    'lot2_surface_carrez',
    'lot3_surface_carrez',
    'lot4_surface_carrez',
    'lot5_surface_carrez',
    'nombre_lots',
    'code_type_local',
    'surface_reelle_bati',
    'nombre_pieces_principales',
    'surface_terrain',
    'longitude',
    'latitude']]

print(X.dtypes)
```

```
id_mutation                 int32
nature_mutation             int32
valeur_fonciere           float64
adresse_nom_voie            int32
code_commune              float64
lot1_surface_carrez       float64
lot2_surface_carrez       float64
lot3_surface_carrez       float64
lot4_surface_carrez       float64
lot5_surface_carrez       float64
nombre_lots                 int32
code_type_local           float64
surface_reelle_bati       float64
nombre_pieces_principales float64
surface_terrain           float64
longitude                 float64
latitude                  float64
dtype: object
```

J'ai remarqué que dans le jeu de données, de nombreuses caractéristiques étaient identiques sur plusieurs lignes, notamment pour des colonnes comme **id_mutation** , **valeur_foncière** , **code_commune** , etc. Cela signifie que plusieurs lignes représentent une seule transaction avec le même **id_mutation** et la même **valeur_foncière** . Ce phénomène pose problème, car la répétition de la **valeur_foncière** pour chaque ligne d'un même groupe peut perturber le modèle.

En effet, lors de l'entraînement, le modèle analyse les données ligne par ligne. Par exemple, il pourrait voir sur la première ligne un appartement de 20 m² avec une **valeur_foncière** de 300 000 euros, puis sur la ligne suivante une maison de 300 m² avec la même **valeur_foncière** de 300 000 euros. Cela créerait une confusion, car le modèle pourrait conclure que les deux biens (appartement et maison) ont les mêmes caractéristiques, ce qui est incorrect.

Pour résoudre ce problème, nous regroupons les transactions d'un même groupe (**id_mutation**) en une seule ligne. Les caractéristiques additives, comme **surface_terrain** , **surface_reelle_bati** et **nombre_de_pieces_principales** , sont additionnées. Les autres caractéristiques répétitives, comme la **valeur_foncière** , sont simplement conservées. Ainsi, au lieu d'avoir deux lignes distinctes, nous obtenons une seule ligne consolidée, par exemple : "appartement + maison, 320 m², 300 000 euros" ce qui représente la véritable valeur foncière du lot.

En complément, nous ajoutons une nouvelle colonne nommée **dep** (pour dépendance). En analysant les données, nous avons constaté que les lignes correspondantes aux dépendances n'avaient aucune valeur dans les colonnes **surface_reelle_bati** ou **nombre_de_pieces_principales** . Ces lignes, initialement présentes dans la colonne **type_local** , contiennent différents types de biens, tels que maison, appartement, local industriel et dépendance.

Pour améliorer le modèle, nous avons supprimer les dépendances de la colonne **type_local** et créer la colonne **dep** . Cette colonne indique, pour chaque groupe, s'il existe une ou plusieurs dépendances (valeur 1) ou aucune dépendance (valeur 0). Ainsi, le regroupement des transactions est complété, tout en intégrant cette information importante dans l'ensemble de données.

```python
unique_check_columns = ["id_mutation","nature_mutation","valeur_fonciere","adresse_numero","adresse_nom_voie","code_commune","lot1_numero","lot1_surface_

first_columns = ['code_type_local']

sum_columns = ['surface_reelle_bati', 'nombre_pieces_principales', 'surface_terrain']


def process_dataframe(df, unique_check_columns, first_columns, sum_columns):

    df['dep'] = df.groupby('id_mutation')['code_type_local'].transform(lambda x: 1 if 3 in x.values else 0)

    df_filtered = df[df['code_type_local'] != 3]

    agg_dict = {}

    for col in sum_columns:
        agg_dict[col] = 'sum'

    for col in first_columns:
        agg_dict[col] = 'first'

    for col in unique_check_columns:
        agg_dict[col] = lambda x: x.iloc[0]

    agg_dict['dep'] = 'max'

    df_processed = df_filtered.groupby('id_mutation', as_index=False).agg(agg_dict)

    return df_processed

df= process_dataframe(df, unique_check_columns, first_columns, sum_columns)
```

On arrive ensuite à ce nombre d’erreur

```python
df.isnull().sum()
```

```
surface_reelle_bati              0
nombre_pieces_principales        0
surface_terrain                  0
code_type_local              23117
id_mutation                      0
nature_mutation                  0
valeur_fonciere                392
adresse_nom_voie                 0
code_commune                     1
lot1_surface_carrez          83467
lot2_surface_carrez          93219
lot3_surface_carrez          96742
lot4_surface_carrez          97061
lot5_surface_carrez          97120
nombre_lots                      0
longitude                      258
latitude                       258
dep                              0
dtype: int64
```

On constate qu'il nous reste 23 117 éléments nuls dans la colonne code_type_local . J'ai donc créé une fonction qui permet la suppression des éléments nulles de la colonne code_type_local . Comme les lignes représentent désormais uniquement les transactions, on peut supprimer les lignes sans problème. Supprimer une ligne reviendra à supprimer une transaction entière, et non un fragment de transaction, ce qui ne faussera pas le jeu de données.

```
def drop_null(df,target_column):
    df[target_column] = df[target_column].replace(['', 'NA', 'na', 'N/A'], pd.NA)
    suppr_vide_type = df[df[target_column].isna()].index
    df = df.drop(suppr_vide_type, axis=0)
    return df

df = drop_null(df,'code_type_local')
```

Parfois, il est important de garder des éléments nuls, car ils sont présents de manière intentionnelle, comme les lots_surface_carrez des différents lots. Il faut donc les préciser au modèle. J'ai donc créé des colonnes missing pour certaines fonctionnalité, ce qui permet, lorsqu'il n'y a rien dans la colonne, de mettre 1 dans les colonnes manquantes et 0 dans les colonnes de base, afin que les modèles comprennent que les valeures nulles ne sont pas une erreur.

```
import numpy as np

columns = ['lot1_surface_carrez', 'lot2_surface_carrez', 'lot3_surface_carrez', 'lot4_surface_carrez', 'lot5_surface_carrez']

def missing_value(df, columns, replacement_value=0):

    for col in columns:
        df[f'missing_{col}'] = df[col].isnull()
        df[col] = df[col].fillna(replacement_value)

    return df

missing_value(df, columns, replacement_value=0)

print(df.head())
```

Ensuite, il faut supprimer les colonnes **id_mutation**, qui ne sont pas nécessaires pour la cible, et **code_type_local**, qui ne l'est pas non plus.
Id_mutation est un élément unique et ne traduis aucune tendance.
Je précise que, **code_type_local** n'est pas très utile étant donné qu'on a rassemblé les groupes en une ligne. Si je garde cette colonne en rassemblant les lignes, la fonction prendra un des biens aléatoirement de la transaction, ce qui faussera les résultats.
Enfin on termine par une suppression générale qui supprime les quelques éléments (environ 150) qu'il reste.

```
colonnes_1 = ['id_mutation','code_type_local']
df = df.drop(columns=colonnes_1)
```

```
Ensuite on remplace les derniers petits éléments null qui sont au nombre d'une trentaine pour bien netoyer le dataset
```

```
df= df.dropna(axis=0, how='any')
```

```
On vérifie une fois de plus les dimenssions du dataset
```

```
df.shape
```

```
(73817, 21)
```

il nous reste un dataset de 74000 lignes

```
df.isnull().sum()
```

```
surface_reelle_bati            0
nombre_pieces_principales      0
surface_terrain                0
nature_mutation                0
valeur_fonciere                0
adresse_nom_voie               0
code_commune                   0
lot1_surface_carrez            0
lot2_surface_carrez            0
lot3_surface_carrez            0
lot4_surface_carrez            0
lot5_surface_carrez            0
nombre_lots                    0
longitude                      0
latitude                       0
dep                            0
missing_lot1_surface_carrez    0
missing_lot2_surface_carrez    0
missing_lot3_surface_carrez    0
missing_lot4_surface_carrez    0
missing_lot5_surface_carrez    0
dtype: int64
```

Nous n’avons plus de valeurs nulles dans le dataset, ce qui permet maintenant aux modèles de pouvoir prédire les éléments.

Donc, on a prétraité le dataset en collectant les colonnes les plus pertinentes, on a comblé les valeurs nulles pour nettoyer le dataset et pour qu'il soit lisible par les modèles, et on a converti tous les éléments objet en type float ou int pour que les modèles puissent être entraînés.

Étape 4 : Mise en place des modèles

On choisit les différentes entrées et la cible, puis on sépare la partie d'entraînement et de test du dataset pour les modèles. On choisit 10% des données pour le test et 90% pour l'entraînement, en effet avec seulement 70000 ligne il est préférable de mettre plus de données à contribution de l'entrainement que pour le test. On utilise également la fonction StandardScaler() pour mettre les grandes et petites valeurs à la même échelle afin que les modèles comprennent mieux.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X = df[[
    'nature_mutation',
    'adresse_nom_voie',
    'code_commune',
    'nombre_lots',
    'surface_reelle_bati',
    'nombre_pieces_principales',
    'surface_terrain',
    'longitude',
    'latitude',
    'dep',
    'missing_lot1_surface_carrez',
    'missing_lot2_surface_carrez',
    'missing_lot3_surface_carrez',
    'missing_lot4_surface_carrez',
    'missing_lot5_surface_carrez',
    'lot1_surface_carrez',
    'lot2_surface_carrez',
    'lot3_surface_carrez',
    'lot4_surface_carrez',
    'lot5_surface_carrez'
]]
y = df['valeur_fonciere']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)

print("Dimensions des données d'entraînement transformées : ", X_train_scaled.shape)
print("Dimensions des données de test transformées : ", X_test_scaled.shape)
```

J'ai mis en place une fonction qui permet de passer en paramètre, dans une boucle, tous les modèles pour déterminer celui avec le meilleur score R² pour mon jeu de données.

```
from sklearn.tree import DecisionTreeRegressor
from xgboost import XGBRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.ensemble import GradientBoostingRegressor, AdaBoostRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import Ridge, Lasso
from sklearn.neural_network import MLPRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error


models = {
    'Linear Regression': LinearRegression(),
    'Decision Tree': DecisionTreeRegressor(),
    'Random Forest': RandomForestRegressor(),
    'XGBoost': XGBRegressor(),
    'Support Vector Machine': SVR(),
    'Ridge Regression': Ridge(alpha=1.0),
    'Lasso Regression': Lasso(alpha=0.01),
    'K-Nearest Neighbors': KNeighborsRegressor(n_neighbors=5),
    'Gradient Boosting': GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3),
    'AdaBoost': AdaBoostRegressor(n_estimators=50, learning_rate=1.0),
    'Neural Network': MLPRegressor(hidden_layer_sizes=(100,), max_iter=500, random_state=42)
}

results = {}

for name, model in models.items():
    model.fit(X_train, y_train)

    y_test_pred = model.predict(X_test)
    train_score = model.score(X_train, y_train)
    test_score = model.score(X_test, y_test)

    rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    mae = mean_absolute_error(y_test, y_test_pred)

    results[name] = {
        'Train R²': train_score,
        'Test R²': test_score,
        'RMSE': rmse_test,
        'MAE': mae
    }

for name, metrics in results.items():
    print(f"\n{name}:")
    for metric, value in metrics.items():
        print(f"{metric}: {value:.4f}")
```

J’ai obtenu un résultat et j’ai demandé a chat gpt de le mettre sous forme de tableau

Modèle	Train R^2	Test R^2	RMSE	MAE
Régression linéaire	0,2207	0,7562	323531.1677	87160.3062
Arbre de décision	0,9952	0,3407	531978.2503	88442.3563
Forêt aléatoire	0,8772	0,3604	523966.3974	66482.6971
XGBoost	0,9433	0,2062	583735.3668	74293.3926
Machine à vecteurs de support	-0,0082	-0,0045	656657.0927	101833.4129
Régression de crête	0,2207	0,7562	323530.9732	87158.3880
Régression Lasso	0,2207	0,7562	323531.1679	87160.2970
K-Les voisins les plus proches	0,4025	0,1892	589947.6974	92548.2429
Amplification du gradient	0,7588	0,1864	590967.5900	83169.3745
AdaBoost	-0,7342	-0,2873	743361.3893	339648.3911
Réseau neuronal	0,2276	0,7185	347587.4913	89416.8958

on constate que la régression linéaire est le meilleur modele.

Modèle machine learning, régression linéaire

Explication des choix de paramètres :

On entraine le modèle avec la fonction '**fit**'
Ensuite on fait une prediction avec la fonction '**predict**'

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
import pandas as pd
import numpy as np

model_machine_learning = LinearRegression()
model_machine_learning.fit(X_train, y_train)

y_test_pred = model_machine_learning.predict(X_test)

train_score = model_machine_learning.score(X_train, y_train)
test_score = model_machine_learning.score(X_test, y_test)

print(f"R² Score sur les données d'entraînement : {train_score}")
print(f"R² Score sur les données de test : {test_score}")

rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
mae = mean_absolute_error(y_test, y_test_pred)

print(f"RMSE sur les données de test : {rmse_test}")
print(f"Erreur absolue moyenne (MAE) : {mae}")
```

```
R² Score sur les données d'entraînement : 0.2161191800003608
R² Score sur les données de test : 0.7549762596798248
RMSE sur les données de test : 324310.97183576267
Erreur absolue moyenne (MAE) : 89477.6832254474
```

Pour le modèle de machine learning en régression, nous avons obtenu 76 % de score R^2 et un MAE de 89477 ainsi qu'un RMSE de 324310

R^2 : Mesure le pourcentage de variance expliquée
MAE : Mesure la magnitude moyenne des erreurs
RMSE : Pénalise les grosses erreurs plus lourdement que MAE

Nous utilisons la validation croisée (cross-validation) pour diviser l'ensemble de données en plusieurs parties et vérifier si le modèle est performant pour toutes ces sections

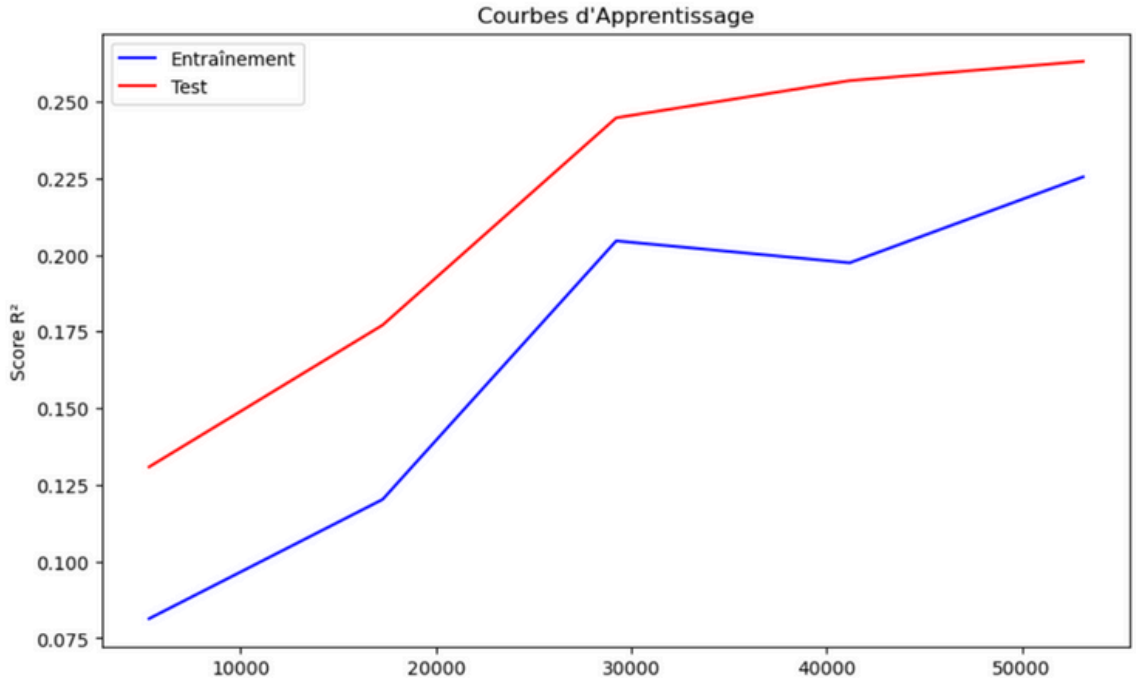
```
from sklearn.model_selection import cross_val_score

cv_results = cross_val_score(model_machine_learning, X, y, cv=5)
print(cv_results)
```

```
[ 0.79805617  0.2449592  -0.80946409  0.15497958  0.18905896]
```

Les résultats montrent que le modèle est instable d'un sous-ensemble à l'autre

On vérifie la courbe d'apprentissage pour voir si le modèle s'ajuste bien et prédit correctement.



On remarque sur la courbe d’apprentissage que le score du test est bien meilleure que celui d’entrainement

Modèle deep learning , réseau de neuronne

Explication des choix de paramètres :

J'ai décidé de mettre en place 5 couches cachées (hidden layer) car c'était le meilleur choix pour obtenir un meilleur score en précisant la taille par puissance de 2.
input_dim=X_train.shape[1] correspond à la dimension de mon jeu de données d'entraînement, c'est-à-dire le nombre d'entrées.
J'ai choisi également 'relu' comme fonction d'activation car elle est principalement utilisée dans les couches cachés.
Adam : Optimiseur polyvalent, qui fonctionne bien dans la plupart des situations et est efficace pour minimiser les fonctions de perte comme le MSE .
L'optimiseur permet d'améliorer le changement des poids et biais
MSE (Mean Squared Error) : Approprié pour mesurer l'erreur dans les prédictions de valeurs continues.
BatchNormalization : Permet d'améliorer l'entraînement en particulier la variance interne des activations.
EarlyStopping : Fonction qui permet d'arrêter les épochs lorsque le poids ne change plus et que le modèle ne s'améliore plus, ce qui empêche le sur-apprentissage.
On a choisi 50 epochs pour laisser le temps au modèle de corriger les poids ainsi que les biais tout au long de l'entrainement

```
from tensorflow.keras.layers import Dropout
from tensorflow.keras.regularizers import l2
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.layers import BatchNormalization
from sklearn.metrics import mean_absolute_error, r2_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

model = Sequential()

model = Sequential()
model.add(Dense(512, input_dim=X_train.shape[1], activation='relu'))
model.add(BatchNormalization())
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(32, activation='relu'))
model.add(Dense(1))

model.compile(optimizer='adam', loss='mean_squared_error')

early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

history = model.fit(X_train_scaled, y_train, epochs=50, validation_data=(X_test_scaled, y_test), callbacks=[early_stopping])

preds = model.predict(X_test_scaled)

mae = mean_absolute_error(y_test, preds)
print(f"Mean Absolute Error (MAE): {mae}")
r2 = r2_score(y_test, preds)
print(f"R² Score : {r2}")

rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
print(f"RMSE sur les données de test : {rmse_test}")
```

```
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 178945507328.0000 - val_loss: 410077724672.0000
Epoch 2/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 179442286592.0000 - val_loss: 299001872384.0000
Epoch 3/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - loss: 152703647744.0000 - val_loss: 169427566592.0000
Epoch 4/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 93613219840.0000 - val_loss: 179673939968.0000
Epoch 5/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 116952088576.0000 - val_loss: 384455606272.0000
Epoch 6/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 100149936128.0000 - val_loss: 258327560192.0000
Epoch 7/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 142668821760.0000 - val_loss: 160630358016.0000
Epoch 8/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 139710431232.0000 - val_loss: 356115087360.0000
Epoch 9/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 128949444608.0000 - val_loss: 282087456768.0000
Epoch 10/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 154194575360.0000 - val_loss: 304957980672.0000
Epoch 11/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 202343989248.0000 - val_loss: 358084346880.0000
Epoch 12/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 123213078528.0000 - val_loss: 154702790656.0000
Epoch 13/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 149665857536.0000 - val_loss: 113675657216.0000
Epoch 14/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 244484273152.0000 - val_loss: 108617897216.0000
Epoch 15/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 101601976320.0000 - val_loss: 95152259072.0000
Epoch 16/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 126906130432.0000 - val_loss: 396716343296.0000
Epoch 17/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 153098616832.0000 - val_loss: 339801210880.0000
Epoch 18/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 107341291520.0000 - val_loss: 382579703808.0000
Epoch 19/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 182335324160.0000 - val_loss: 148163802368.0000
Epoch 20/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 235037491200.0000 - val_loss: 245868789760.0000
Epoch 21/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 122307059712.0000 - val_loss: 188096462848.0000
Epoch 22/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 227334094848.0000 - val_loss: 376881774592.0000
Epoch 23/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 188719742976.0000 - val_loss: 395374788608.0000
Epoch 24/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 235013570560.0000 - val_loss: 400265347072.0000
Epoch 25/50
2077/2077 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 102347923456.0000 - val_loss: 399463481344.0000
231/231 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Mean Absolute Error (MAE): 74807.64680796617
R² Score : 0.7783315786222647
RMSE sur les données de test : 323634.9953680799
```

On obtient un R^2 score de 78% et 74000 de MAE pour le modèle deep learning

En conclusion, avec le nombre de données dont je dispose, le meilleur modèle capable de faire une prédiction précise est le modèle de machine learning en régression linéaire, qui obtient un R^2 de 75%, contrairement au deep learning qui atteint 78%. Le MAE du modèle de machine learning est de 90 000, tandis que celui du deep learning est de 74 000. On pourrait penser que le deep learning est plus adapté, cependant, dans ce cas précis, avec si peu de données, le machine learning le plus performant.

Étape 5 : Fonction de création d'excel avec prédiction

J'ai créé une fonction nommée clean file qui reprend tout le prétraitement utilisé pour l'entrainement afin que le fichier soit lisible pour le modèle

```
def clean_file(df):

    df = pd.read_csv('data_immobiliers.csv')

    colonne_keep = [
        "surface_reelle_bati", "nombre_pieces_principales", "surface_terrain",
        "code_type_local", "id_mutation", "date_mutation", "nature_mutation",
        "valeur_fonciere", "adresse_nom_voie", "code_commune",
        "lot1_surface_carrez", "lot2_surface_carrez",
         "lot3_surface_carrez", "lot4_surface_carrez",
         "lot5_surface_carrez", "nombre_lots", "longitude",
        "latitude", "nature_culture"
    ]
    df = df[colonne_keep]

    colonnes_surface = [
        'lot1_surface_carrez', 'lot2_surface_carrez', 'lot3_surface_carrez',
        'lot4_surface_carrez', 'lot5_surface_carrez'
    ]
    df_new = traiter_surface_reelle_bati(df, 'surface_reelle_bati', colonnes_surface, 'code_type_local')

    df = clean_and_fill_column(df_new, 'latitude', 'adresse_nom_voie')
    df = clean_and_fill_column(df_new, 'longitude', 'adresse_nom_voie')
    df = clean_and_fill_column(df_new, 'surface_terrain', 'nombre_lots')

    columns_list = [
        'valeur_fonciere', 'adresse_numero', 'lot1_surface_carrez',
        'lot2_surface_carrez', 'lot3_surface_carrez', 'lot4_surface_carrez'
```

J’ai créé une fonction nommée predict_file qui importe le fichier csv ,il définit les entrées. J’ai créé une liste feature car il fallait que je présente les entrées sans la valeur foncière, ensuite il créer une nouvelle colonne ‘valeur_fonciere_predite’ et il génère un fichier csv avec les prédictions par ligne

```
def predict_file(fichier, model):
    df_new = pd.read_csv(fichier)
    df_new=clean_file(df_new)

    features = [
    'nature_mutation',
    'adresse_nom_voie',
    'code_commune',
    'nombre_lots',
    'surface_reelle_bati',
    'nombre_pieces_principales',
    'surface_terrain',
    'longitude',
    'latitude',
    'dep',
    'missing_lot1_surface_carrez',
    'missing_lot2_surface_carrez',
    'missing_lot3_surface_carrez',
    'missing_lot4_surface_carrez',
    'missing_lot5_surface_carrez',
    'lot1_surface_carrez',
    'lot2_surface_carrez',
    'lot3_surface_carrez',
    'lot4_surface_carrez',
    'lot5_surface_carrez'
]

    df_features = df_new[features]

    predictions = model_machine_learning.predict(df_features)
    df_new['Valeur_fonciere_predite'] = predictions

    df_new.to_csv('property_data_with_predictions.csv', index=False)
    print("Prédictions sauvegardées dans 'property_data_with_predictions.csv'")

predict_file('data_immobiliers.csv', model)
```

Le modèle utilisé pour la prédiction est le modèle de machine learning en regression lineaire qui est bien plus performant à 70000 lignes