notebook_Minh

April 30, 2023

{r setup, include=FALSE} title: "Untitled" author: "John Doe, Jane Doe"

[2]:
```
"""
Installation de Tensorflow et Keras est nécessaire pour lancer ce notebook
"""
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import seaborn as sns
from sklearn.linear_model import LinearRegression, Ridge, RidgeCV
from sklearn.kernel_ridge import KernelRidge
from sklearn.kernel_approximation import Nystroem
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense
from sklearn.metrics import r2_score
from sklearn.model_selection import cross_validate
%matplotlib inline
```

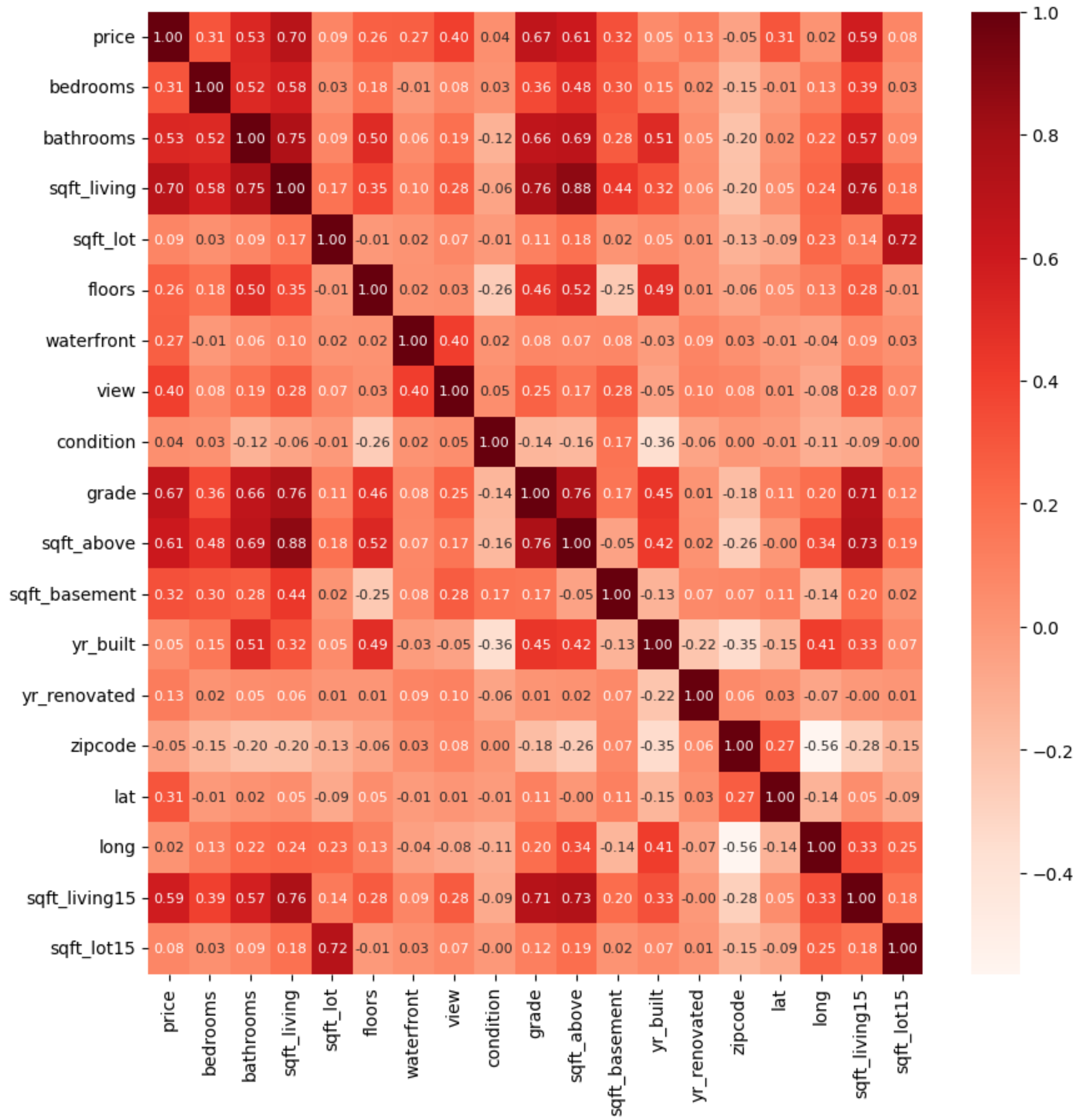
1 Exploration de données

[3]:
```
df = pd.read_csv('kc_house_data.csv', index_col='id')
df
```

[3]:
```
                       date     price  bedrooms  bathrooms  sqft_living  \
id
7129300520  20141013T000000  221900.0         3       1.00         1180
6414100192  20141209T000000  538000.0         3       2.25         2570
5631500400  20150225T000000  180000.0         2       1.00          770
2487200875  20141209T000000  604000.0         4       3.00         1960
1954400510  20150218T000000  510000.0         3       2.00         1680
…                         …         …       …        …            …
263000018   20140521T000000  360000.0         3       2.50         1530
6600060120  20150223T000000  400000.0         4       2.50         2310
1523300141  20140623T000000  402101.0         2       0.75         1020
291310100   20150116T000000  400000.0         3       2.50         1600
```

```
1523300157  20141015T000000  325000.0         2       0.75         1020

            sqft_lot  floors  waterfront  view  condition  grade  sqft_above  \
id
7129300520      5650     1.0           0     0          3      7        1180
6414100192      7242     2.0           0     0          3      7        2170
5631500400     10000     1.0           0     0          3      6         770
2487200875      5000     1.0           0     0          5      7        1050
1954400510      8080     1.0           0     0          3      8        1680
…                …     …         …   …       …    …        …
263000018       1131     3.0           0     0          3      8        1530
6600060120      5813     2.0           0     0          3      8        2310
1523300141      1350     2.0           0     0          3      7        1020
291310100       2388     2.0           0     0          3      8        1600
1523300157      1076     2.0           0     0          3      7        1020

            sqft_basement  yr_built  yr_renovated  zipcode      lat     long  \
id
7129300520              0      1955             0    98178  47.5112 -122.257
6414100192            400      1951          1991    98125  47.7210 -122.319
5631500400              0      1933             0    98028  47.7379 -122.233
2487200875            910      1965             0    98136  47.5208 -122.393
1954400510              0      1987             0    98074  47.6168 -122.045
…                     …       …           …      …      …      …
263000018               0      2009             0    98103  47.6993 -122.346
6600060120              0      2014             0    98146  47.5107 -122.362
1523300141              0      2009             0    98144  47.5944 -122.299
291310100               0      2004             0    98027  47.5345 -122.069
1523300157              0      2008             0    98144  47.5941 -122.299

            sqft_living15  sqft_lot15
id
7129300520           1340        5650
6414100192           1690        7639
5631500400           2720        8062
2487200875           1360        5000
1954400510           1800        7503
…                     …          …
263000018            1530        1509
6600060120           1830        7200
1523300141           1020        2007
291310100            1410        1287
1523300157           1020        1357

[21613 rows x 20 columns]
```

[4]:
```python
# Matrix de coorelation de DataFrame
coor = df.corr()
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(coor, annot=True, cmap=plt.cm.Reds, fmt='.2f',annot_kws={"fontsize":
 ↪8})
plt.show()
```



[5]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21613 entries, 7129300520 to 1523300157
Data columns (total 20 columns):
```

```
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   date           21613 non-null  object
 1   price          21613 non-null  float64
 2   bedrooms       21613 non-null  int64
 3   bathrooms      21613 non-null  float64
 4   sqft_living    21613 non-null  int64
 5   sqft_lot       21613 non-null  int64
 6   floors         21613 non-null  float64
 7   waterfront     21613 non-null  int64
 8   view           21613 non-null  int64
 9   condition      21613 non-null  int64
 10  grade          21613 non-null  int64
 11  sqft_above     21613 non-null  int64
 12  sqft_basement  21613 non-null  int64
 13  yr_built       21613 non-null  int64
 14  yr_renovated   21613 non-null  int64
 15  zipcode        21613 non-null  int64
 16  lat            21613 non-null  float64
 17  long           21613 non-null  float64
 18  sqft_living15  21613 non-null  int64
 19  sqft_lot15     21613 non-null  int64
dtypes: float64(5), int64(14), object(1)
memory usage: 3.5+ MB
```

2 Nettoyage de données

Nous avons supprimé les colonnes qui ne nous intéressent pas pour notre analyse.

[6]:
```python
new_df = df.drop(['date', 'zipcode', 'yr_renovated'], axis=1)
```

Nous avons décidé d'utiliser le `StandardScaler` de sklearn pour normaliser les données. Nous obtenons ainsi des données centrées réduites.

[7]:
```python
# Rescale data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
new_df['sqft_living'] = scaler.fit_transform(new_df['sqft_living'].values.
 ↪reshape(-1,1))
new_df['sqft_lot'] = scaler.fit_transform(new_df['sqft_lot'].values.
 ↪reshape(-1,1))
new_df['sqft_above'] = scaler.fit_transform(new_df['sqft_above'].values.
 ↪reshape(-1,1))
new_df['sqft_living15'] = scaler.fit_transform(new_df['sqft_living15'].values.
 ↪reshape(-1,1))
new_df['sqft_lot15'] = scaler.fit_transform(new_df['sqft_lot15'].values.
 ↪reshape(-1,1))
```

```
new_df['sqft_basement'] = scaler.fit_transform(new_df['sqft_basement'].values.
 ↪reshape(-1,1))
new_df['yr_built'] = scaler.fit_transform(new_df['yr_built'].values.
 ↪reshape(-1,1))
new_df['lat'] = scaler.fit_transform(new_df['lat'].values.reshape(-1,1))
new_df['long'] = scaler.fit_transform(new_df['long'].values.reshape(-1,1))
new_df.describe()
```

[7]:

```
              price      bedrooms     bathrooms   sqft_living      sqft_lot  \
count  2.161300e+04  21613.000000  21613.000000  2.161300e+04  2.161300e+04
mean   5.400881e+05      3.370842      2.114757  3.174253e-16  3.281921e-17
std    3.671272e+05      0.930062      0.770163  1.000023e+00  1.000023e+00
min    7.500000e+04      0.000000      0.000000 -1.948891e+00 -3.521759e-01
25%    3.219500e+05      3.000000      1.750000 -7.108948e-01 -2.430487e-01
50%    4.500000e+05      3.000000      2.250000 -1.849914e-01 -1.808075e-01
75%    6.450000e+05      4.000000      2.500000  5.118578e-01 -1.066880e-01
max    7.700000e+06     33.000000      8.000000  1.247807e+01  3.950434e+01

             floors    waterfront          view     condition         grade  \
count  21613.000000  21613.000000  21613.000000  21613.000000  21613.000000
mean       1.494309      0.007542      0.234303      3.409430      7.656873
std        0.539989      0.086517      0.766318      0.650743      1.175459
min        1.000000      0.000000      0.000000      1.000000      1.000000
25%        1.000000      0.000000      0.000000      3.000000      7.000000
50%        1.500000      0.000000      0.000000      3.000000      7.000000
75%        2.000000      0.000000      0.000000      4.000000      8.000000
max        3.500000      1.000000      4.000000      5.000000     13.000000

         sqft_above  sqft_basement      yr_built           lat          long  \
count  2.161300e+04   2.161300e+04  2.161300e+04  2.161300e+04  2.161300e+04
mean   3.892022e-16  -2.022801e-15  3.592925e-15 -3.432958e-14 -3.663944e-14
std    1.000023e+00   1.000023e+00  1.000023e+00  1.000023e+00  1.000023e+00
min   -1.809494e+00  -6.586810e-01 -2.417383e+00 -2.916795e+00 -2.166543e+00
25%   -7.226314e-01  -6.586810e-01 -6.810785e-01 -6.426977e-01 -8.102505e-01
50%   -2.758102e-01  -6.586810e-01  1.360059e-01  8.478232e-02 -1.143518e-01
75%    5.091458e-01   6.066704e-01  8.849999e-01  8.512345e-01  6.312541e-01
max    9.204044e+00   1.023238e+01  1.497813e+00  1.570054e+00  6.383070e+00

       sqft_living15    sqft_lot15
count   2.161300e+04  2.161300e+04
mean   -1.506632e-16  1.235382e-16
std     1.000023e+00  1.000023e+00
min    -2.316325e+00 -4.438052e-01
25%    -7.244971e-01 -2.808593e-01
50%    -2.138280e-01 -1.885636e-01
75%     5.448802e-01 -9.835556e-02
max     6.162239e+00  3.144029e+01
```

[8]:
```python
X = new_df.drop('price', axis=1)
y = new_df['price']
```

2.1 Split des données en train et test

[9]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=0)
X_train
```

[9]:
```
            bedrooms  bathrooms  sqft_living  sqft_lot  floors  waterfront  \
id
5100402668         3       1.00    -0.555193 -0.231701     1.0           0
7856560480         3       2.50    -0.326539 -0.099155     1.0           0
2872900010         3       1.50    -1.077829 -0.126630     1.0           0
3216900070         4       2.50     0.141657 -0.193821     2.0           0
976000790          3       2.50    -0.304762 -0.249736     2.0           0
…                …          …            …         …      …           …
2322069010         5       5.00     2.047104  1.906878     2.0           0
2114700368         2       2.50    -0.740293 -0.334262     2.0           0
5469501200         3       2.25     0.304981 -0.003790     1.0           0
3751602797         4       2.00     0.315869  1.486207     2.0           0
4038600260         4       2.25     0.326757  0.027258     1.0           0

            view  condition  grade  sqft_above  sqft_basement  yr_built  \
id
5100402668     0          4      7   -0.867546       0.471097 -1.055576
7856560480     0          4      8   -0.698479       0.629266  0.306232
2872900010     0          3      8   -0.843394      -0.658681  0.544548
3216900070     0          3      8    0.509146      -0.658681  0.748819
976000790      0          3      7   -0.662250       0.606670  0.476458
…            …          …      …           …              …         …
2322069010     0          3     10    2.622489      -0.658681  0.919045
2114700368     0          3      8   -0.758860      -0.116388  1.259497
5469501200     0          4      9    0.690290      -0.658681  0.238141
3751602797     0          4      8    0.702366      -0.658681  0.238141
4038600260     0          3      7   -0.299963       1.239346 -0.340627

                 lat      long  sqft_living15  sqft_lot15
id
5100402668  0.968151 -0.746341      -0.315962   -0.233979
7856560480 -0.019143  0.460830       0.471927   -0.112383
2872900010  0.473060  1.263244      -0.403505   -0.106450
3216900070 -1.006438  0.219396      -0.024151   -0.211271
976000790   0.620288 -1.051685      -0.286781   -0.293202
…                …         …              …           …
2322069010 -1.299451  1.440769       0.369794    1.893510
2114700368 -0.186579 -0.959372      -1.351890   -0.411831
```

```
5469501200 -1.259035  0.396921       1.070140    0.059316
3751602797 -1.998784 -0.462301       0.180117    0.240465
4038600260  0.379239  0.666759       0.355203   -0.140768

[17290 rows x 16 columns]
```

3 Implementation de K-fold Cross Validation

[10]:
```
def cross_validation(model, _x, _y, _cv=5):
    """
    Parameters
    ----------
    model        : model to be used for cross validation
    _x           : features to train
    _y           : target to train
    _cv          : int
                   number of folds

    Returns
    -------
    results      : dictionary of average cross validation results

    """
    _scoring = ['r2', 'neg_mean_squared_error']
    results = cross_validate(model, _x, _y, cv=_cv, scoring=_scoring)
    results['test_mean_squared_error'] = -results['test_neg_mean_squared_error']
    results['mean_r2'] = results['test_r2'].mean()
    return results
```

4 Création des modèles

4.1 Linear regression model

[11]:
```
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)
```

[11]:
```
LinearRegression()
```

[12]:
```
res_validation = cross_validation(linear_model, X_train, y_train)
res_validation
```

[12]:
```
{'fit_time': array([0.01049662, 0.06285238, 0.00599885, 0.00799727,
0.00498366]),
 'score_time': array([0.00219822, 0.01600718, 0.00200152, 0.00101733,
0.00100088]),
 'test_r2': array([0.67505938, 0.66494874, 0.6991278 , 0.71386052, 0.70856267]),
```

```
 'test_neg_mean_squared_error': array([-5.80908872e+10, -3.82063063e+10,
-4.63087741e+10, -3.57464713e+10,
        -3.54090643e+10]),
 'test_mean_squared_error': array([5.80908872e+10, 3.82063063e+10,
4.63087741e+10, 3.57464713e+10,
        3.54090643e+10]),
 'mean_r2': 0.6923118242624663}
```

4.2 Prediction of Liear Regression Model

[13]:
```python
# Data scatter of predicted values
y_pred = linear_model.predict(X_test)
y_test = np.array(y_test)
y_pred, y_test
```

[13]:
```
(array([ 379824.39845475, 1520132.09686934,  535192.63307498, …,
         348266.82410121,  202975.57538713,  403521.86733752]),
 array([ 297000., 1578000.,  562100., …,  369950.,  300000.,  575950.]))
```

[14]:
```python
# p-value on test set
r2_score(y_test, y_pred)
```

[14]:
```
0.6884620663104776
```

4.3 Ridge regression

4.3.1 Intuition de Ridge Regression

{python} Il faut la régularisation pour la Regression Ridge, même s'il faut que alpha ne soit pas tout petit. Sinon, il risque que les colonnes sont très corrélées. Si alpha est trop grand, la p-valeur ne descend pas beaucoup. En conclusion, nous pouvons dire que ce jeu de données est adapté pour la Regression Ridge.

[15]:
```python
ridge_reg = Ridge(alpha=1e8, solver='svd')
```

[16]:
```python
ridge_reg.fit(X_train, y_train)
```

[16]:
```
Ridge(alpha=100000000.0, solver='svd')
```

[17]:
```python
# p-value de Ridge Regression
ridge_res_validation = cross_validation(ridge_reg, X_train, y_train)
ridge_res_validation['mean_r2']
```

[17]:
```
-9.473387491352181e-05
```

4.3.2 Perform Leave-One-Out Cross Validation

[18]:
```
ridgeCV = RidgeCV(alphas=[1e-6, 1e-3, 1, 1000], cv=5)
```

[19]:
```
ridgeCV.fit(X_train, y_train)
```

[19]:
```
RidgeCV(alphas=array([1.e-06, 1.e-03, 1.e+00, 1.e+03]), cv=5)
```

Meilleur α: 1.0

[20]:
```
ridgeCV.alpha_
```

[20]:
```
1.0
```

[21]:
```
ridgeCV.score(X_test, y_test)
```

[21]:
```
0.6885181988335431
```

4.4 Prediction of Ridge Regression Model

[21]:
```
def ridge_score(alpha=1.0):
    ridge_reg = Ridge(alpha=alpha, solver="svd")
    ridge_reg.fit(X_train, y_train)
    r2_ridge = cross_validation(ridge_reg, X_train, y_train)['mean_r2']
    return r2_ridge
```

[22]:
```
## Iterate over alpha values
alpha_values = np.linspace(0.01, 1, 100)
r2_ridge = []
for alpha in alpha_values:
    r2_ridge.append(ridge_score(alpha))
```

Nous pouvons voir que la R^2 ne varie pas beaucoup si nous faisons varier α. Cela veut dire que le modèle est assez stable.

[23]:
```
r2_ridge
```

[23]:
```
[0.6923119789388008,
 0.6923121331338864,
 0.69231228684803,
 0.6923124400815378,
 0.6923125928347165,
 0.6923127451078722,
 0.6923128969013108,
 0.6923130482153381,
 0.6923131990502597,
 0.6923133494063809,
 0.692313499284007,
 0.6923136486834429,
```

0.6923137976049933,
0.6923139460489631,
0.692314094015656,
0.692314241505377,
0.6923143885184295,
0.6923145350551175,
0.6923146811157446,
0.6923148267006141,
0.6923149718100292,
0.692315116444293,
0.6923152606037084,
0.6923154042885775,
0.6923155474992031,
0.6923156902358875,
0.6923158324989324,
0.6923159742886396,
0.692316115605311,
0.6923162564492479,
0.6923163968207513,
0.6923165367201225,
0.6923166761476625,
0.6923168151036714,
0.6923169535884497,
0.6923170916022983,
0.6923172291455166,
0.6923173662184047,
0.6923175028212623,
0.6923176389543885,
0.692317774618083,
0.6923179098126446,
0.6923180445383723,
0.6923181787955649,
0.6923183125845206,
0.6923184459055378,
0.6923185787589148,
0.6923187111449491,
0.6923188430639389,
0.6923189745161814,
0.692319105501974,
0.6923192360216139,
0.6923193660753978,
0.6923194956636228,
0.6923196247865852,
0.6923197534445814,
0.6923198816379076,
0.6923200093668598,
0.6923201366317338,

```
0.6923202634328252,
0.6923203897704291,
0.6923205156448411,
0.6923206410563562,
0.6923207660052688,
0.692320890491874,
0.692321014516466,
0.6923211380793391,
0.6923212611807876,
0.692321383821105,
0.692321506000585,
0.6923216277195212,
0.6923217489782071,
0.6923218697769354,
0.6923219901159994,
0.6923221099956915,
0.6923222294163044,
0.6923223483781304,
0.6923224668814619,
0.6923225849265905,
0.6923227025138081,
0.6923228196434063,
0.6923229363156764,
0.69232305253091,
0.6923231682893976,
0.6923232835914305,
0.6923233984372988,
0.6923235128272935,
0.6923236267617046,
0.6923237402408222,
0.6923238532649361,
0.6923239658343363,
0.6923240779493118,
0.6923241896101524,
0.6923243008171472,
0.692324411570585,
0.6923245218707546,
0.6923246317179446,
0.6923247411124432,
0.692324850054539,
0.6923249585445197]
```

4.5 Kernel Ridge Regression

[22]:
```
feature_map_nystroem = Nystroem(gamma=.2,
                                random_state=1,
                                n_components=16)
feature_map_nystroem
```

[22]:
```
Nystroem(gamma=0.2, n_components=16, random_state=1)
```

[25]:
```
X_train_transformed = feature_map_nystroem.fit_transform(X_train)
X_train_transformed.shape
```

[25]:
```
(17290, 16)
```

[28]:
```
kernel_ridge = KernelRidge(alpha=1, kernel='rbf', gamma=.2)
```

[29]:
```
kernel_ridge.fit(X_train_transformed, y_train)
```

[29]:
```
KernelRidge(gamma=0.2, kernel='rbf')
```

[30]:
```
kernel_ridge.score(X_train_transformed, y_train)
```

[30]:
```
0.37826677245632423
```

Nous avons décidé d'implémenter le `Deep Neural Network` pour obtenir un meilleur modèle.

4.6 Neural Network

[24]:
```
neural_network = Sequential(
    [
        Dense(units=128, kernel_initializer='normal', input_dim=X_train.
 ↪shape[1], activation="relu"),
        Dense(units=256, kernel_initializer='normal', activation="relu"),
        Dense(units=256, kernel_initializer='normal', activation="relu"),
        Dense(units=1, activation="linear"),
    ], name="kc_model"
)
```

[25]:
```
neural_network.summary()
```

```
Model: "kc_model"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               2176


 dense_1 (Dense)             (None, 256)               33024


 dense_2 (Dense)             (None, 256)               65792
```

```
 dense_3 (Dense)             (None, 1)                 257

=================================================================
Total params: 101,249
Trainable params: 101,249
Non-trainable params: 0
_________________________________________________________________
```

[26]:
```
#### Examine Weights shapes
[layer1, layer2, layer3, layer4] = neural_network.layers
W1,b1 = layer1.get_weights()
W2,b2 = layer2.get_weights()
W3,b3 = layer3.get_weights()
W4,b4 = layer4.get_weights()
print(f"W1 shape = {W1.shape}, b1 shape = {b1.shape}")
print(f"W2 shape = {W2.shape}, b2 shape = {b2.shape}")
print(f"W3 shape = {W3.shape}, b3 shape = {b3.shape}")
print(f"W4 shape = {W4.shape}, b4 shape = {b4.shape}")
```

```
W1 shape = (16, 128), b1 shape = (128,)
W2 shape = (128, 256), b2 shape = (256,)
W3 shape = (256, 256), b3 shape = (256,)
W4 shape = (256, 1), b4 shape = (1,)
```

[28]:
```
X_train_numpy = np.array(X_train)
y_train_numpy = np.array(y_train)
X_train_numpy.shape
```

[28]: (17290, 16)

[29]:
```
# Compile and fit the model
neural_network.compile(
    loss=tf.keras.losses.MeanAbsoluteError(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    metrics=[tf.keras.metrics.MeanAbsoluteError()]
)

history = neural_network.fit(
    X_train_numpy, y_train_numpy,
    epochs=50,
    batch_size=50,
    validation_split=0.2
)
```

```
Epoch 1/50
277/277 [==============================] - 1s 2ms/step - loss: 361758.7500 -
mean_absolute_error: 361758.7500 - val_loss: 155124.0938 -
val_mean_absolute_error: 155124.0938
```

```
Epoch 2/50
277/277 [==============================] - 0s 2ms/step - loss: 141654.6094 -
mean_absolute_error: 141654.6094 - val_loss: 124997.5156 -
val_mean_absolute_error: 124997.5156
Epoch 3/50
277/277 [==============================] - 0s 2ms/step - loss: 126970.5391 -
mean_absolute_error: 126970.5391 - val_loss: 120336.3203 -
val_mean_absolute_error: 120336.3203
Epoch 4/50
277/277 [==============================] - 0s 2ms/step - loss: 123883.9844 -
mean_absolute_error: 123883.9844 - val_loss: 117404.7734 -
val_mean_absolute_error: 117404.7734
Epoch 5/50
277/277 [==============================] - 0s 2ms/step - loss: 121623.1641 -
mean_absolute_error: 121623.1641 - val_loss: 115586.2656 -
val_mean_absolute_error: 115586.2656
Epoch 6/50
277/277 [==============================] - 0s 2ms/step - loss: 120109.9141 -
mean_absolute_error: 120109.9141 - val_loss: 114070.8047 -
val_mean_absolute_error: 114070.8047
Epoch 7/50
277/277 [==============================] - 0s 2ms/step - loss: 118616.9297 -
mean_absolute_error: 118616.9297 - val_loss: 112459.3359 -
val_mean_absolute_error: 112459.3359
Epoch 8/50
277/277 [==============================] - 0s 2ms/step - loss: 117205.8750 -
mean_absolute_error: 117205.8750 - val_loss: 112344.6250 -
val_mean_absolute_error: 112344.6250
Epoch 9/50
277/277 [==============================] - 0s 2ms/step - loss: 116026.7969 -
mean_absolute_error: 116026.7969 - val_loss: 110621.2578 -
val_mean_absolute_error: 110621.2578
Epoch 10/50
277/277 [==============================] - 0s 2ms/step - loss: 114882.4219 -
mean_absolute_error: 114882.4219 - val_loss: 109255.5938 -
val_mean_absolute_error: 109255.5938
Epoch 11/50
277/277 [==============================] - 0s 2ms/step - loss: 113941.2109 -
mean_absolute_error: 113941.2109 - val_loss: 110929.6172 -
val_mean_absolute_error: 110929.6172
Epoch 12/50
277/277 [==============================] - 0s 2ms/step - loss: 113460.0547 -
mean_absolute_error: 113460.0547 - val_loss: 107895.9688 -
val_mean_absolute_error: 107895.9688
Epoch 13/50
277/277 [==============================] - 0s 2ms/step - loss: 112663.3047 -
mean_absolute_error: 112663.3047 - val_loss: 107421.8125 -
val_mean_absolute_error: 107421.8125
```

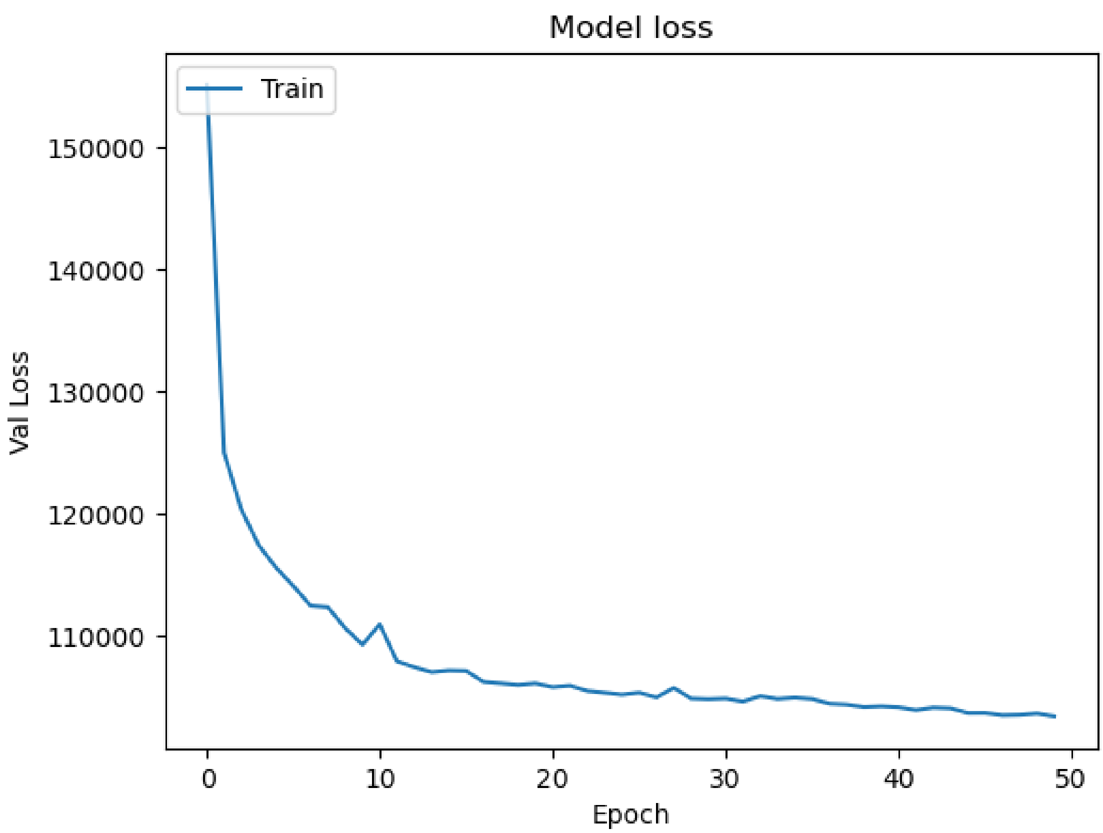
```
Epoch 14/50
277/277 [==============================] - 0s 2ms/step - loss: 112046.0859 -
mean_absolute_error: 112046.0859 - val_loss: 107013.5000 -
val_mean_absolute_error: 107013.5000
Epoch 15/50
277/277 [==============================] - 0s 2ms/step - loss: 111549.4531 -
mean_absolute_error: 111549.4531 - val_loss: 107145.0156 -
val_mean_absolute_error: 107145.0156
Epoch 16/50
277/277 [==============================] - 0s 2ms/step - loss: 111215.2266 -
mean_absolute_error: 111215.2266 - val_loss: 107112.7891 -
val_mean_absolute_error: 107112.7891
Epoch 17/50
277/277 [==============================] - 0s 2ms/step - loss: 110785.0938 -
mean_absolute_error: 110785.0938 - val_loss: 106216.1562 -
val_mean_absolute_error: 106216.1562
Epoch 18/50
277/277 [==============================] - 0s 2ms/step - loss: 110588.5000 -
mean_absolute_error: 110588.5000 - val_loss: 106097.2109 -
val_mean_absolute_error: 106097.2109
Epoch 19/50
277/277 [==============================] - 0s 2ms/step - loss: 110157.4609 -
mean_absolute_error: 110157.4844 - val_loss: 105967.0469 -
val_mean_absolute_error: 105967.0469
Epoch 20/50
277/277 [==============================] - 0s 2ms/step - loss: 109995.2031 -
mean_absolute_error: 109995.1875 - val_loss: 106093.3125 -
val_mean_absolute_error: 106093.3125
Epoch 21/50
277/277 [==============================] - 0s 2ms/step - loss: 109714.9219 -
mean_absolute_error: 109714.9219 - val_loss: 105784.0781 -
val_mean_absolute_error: 105784.0781
Epoch 22/50
277/277 [==============================] - 0s 2ms/step - loss: 109561.1406 -
mean_absolute_error: 109561.1406 - val_loss: 105901.7266 -
val_mean_absolute_error: 105901.7266
Epoch 23/50
277/277 [==============================] - 0s 2ms/step - loss: 109206.2266 -
mean_absolute_error: 109206.2266 - val_loss: 105472.8359 -
val_mean_absolute_error: 105472.8359
Epoch 24/50
277/277 [==============================] - 0s 2ms/step - loss: 108979.5156 -
mean_absolute_error: 108979.5156 - val_loss: 105321.5703 -
val_mean_absolute_error: 105321.5703
Epoch 25/50
277/277 [==============================] - 0s 2ms/step - loss: 108751.5781 -
mean_absolute_error: 108751.5781 - val_loss: 105163.6484 -
val_mean_absolute_error: 105163.6484
```

```
Epoch 26/50
277/277 [==============================] - 0s 2ms/step - loss: 108611.1406 -
mean_absolute_error: 108611.1406 - val_loss: 105323.1953 -
val_mean_absolute_error: 105323.1953
Epoch 27/50
277/277 [==============================] - 0s 2ms/step - loss: 108587.8906 -
mean_absolute_error: 108587.8906 - val_loss: 104946.6172 -
val_mean_absolute_error: 104946.6172
Epoch 28/50
277/277 [==============================] - 0s 2ms/step - loss: 108280.2109 -
mean_absolute_error: 108280.2109 - val_loss: 105723.8750 -
val_mean_absolute_error: 105723.8750
Epoch 29/50
277/277 [==============================] - 1s 2ms/step - loss: 108220.4688 -
mean_absolute_error: 108220.4688 - val_loss: 104851.5234 -
val_mean_absolute_error: 104851.5234
Epoch 30/50
277/277 [==============================] - 0s 2ms/step - loss: 108115.5625 -
mean_absolute_error: 108115.5625 - val_loss: 104801.7500 -
val_mean_absolute_error: 104801.7500
Epoch 31/50
277/277 [==============================] - 1s 2ms/step - loss: 107901.3047 -
mean_absolute_error: 107901.3047 - val_loss: 104851.4219 -
val_mean_absolute_error: 104851.4219
Epoch 32/50
277/277 [==============================] - 1s 2ms/step - loss: 107854.3438 -
mean_absolute_error: 107854.3438 - val_loss: 104605.5625 -
val_mean_absolute_error: 104605.5625
Epoch 33/50
277/277 [==============================] - 0s 2ms/step - loss: 107688.7266 -
mean_absolute_error: 107688.7266 - val_loss: 105052.9297 -
val_mean_absolute_error: 105052.9297
Epoch 34/50
277/277 [==============================] - 0s 2ms/step - loss: 107713.3906 -
mean_absolute_error: 107713.3906 - val_loss: 104819.6484 -
val_mean_absolute_error: 104819.6484
Epoch 35/50
277/277 [==============================] - 0s 2ms/step - loss: 107379.9453 -
mean_absolute_error: 107379.9453 - val_loss: 104927.8359 -
val_mean_absolute_error: 104927.8359
Epoch 36/50
277/277 [==============================] - 0s 2ms/step - loss: 107327.0156 -
mean_absolute_error: 107327.0156 - val_loss: 104815.0391 -
val_mean_absolute_error: 104815.0391
Epoch 37/50
277/277 [==============================] - 0s 2ms/step - loss: 107233.3906 -
mean_absolute_error: 107233.3906 - val_loss: 104432.8594 -
val_mean_absolute_error: 104432.8594
```

```
Epoch 38/50
277/277 [==============================] - 0s 2ms/step - loss: 107128.2656 -
mean_absolute_error: 107128.2656 - val_loss: 104360.8672 -
val_mean_absolute_error: 104360.8672
Epoch 39/50
277/277 [==============================] - 0s 2ms/step - loss: 106943.2500 -
mean_absolute_error: 106943.2500 - val_loss: 104153.7422 -
val_mean_absolute_error: 104153.7422
Epoch 40/50
277/277 [==============================] - 0s 2ms/step - loss: 106985.8750 -
mean_absolute_error: 106985.8750 - val_loss: 104217.3281 -
val_mean_absolute_error: 104217.3281
Epoch 41/50
277/277 [==============================] - 0s 2ms/step - loss: 106825.0859 -
mean_absolute_error: 106825.0859 - val_loss: 104136.8672 -
val_mean_absolute_error: 104136.8672
Epoch 42/50
277/277 [==============================] - 0s 2ms/step - loss: 106819.1484 -
mean_absolute_error: 106819.1484 - val_loss: 103906.4375 -
val_mean_absolute_error: 103906.4375
Epoch 43/50
277/277 [==============================] - 0s 2ms/step - loss: 106654.2969 -
mean_absolute_error: 106654.2969 - val_loss: 104109.7891 -
val_mean_absolute_error: 104109.7891
Epoch 44/50
277/277 [==============================] - 0s 2ms/step - loss: 106452.1953 -
mean_absolute_error: 106452.1953 - val_loss: 104054.7500 -
val_mean_absolute_error: 104054.7500
Epoch 45/50
277/277 [==============================] - 0s 2ms/step - loss: 106468.8359 -
mean_absolute_error: 106468.8359 - val_loss: 103671.9609 -
val_mean_absolute_error: 103671.9609
Epoch 46/50
277/277 [==============================] - 0s 2ms/step - loss: 106318.0859 -
mean_absolute_error: 106318.0859 - val_loss: 103678.5391 -
val_mean_absolute_error: 103678.5391
Epoch 47/50
277/277 [==============================] - 0s 2ms/step - loss: 106125.0234 -
mean_absolute_error: 106125.0234 - val_loss: 103490.8594 -
val_mean_absolute_error: 103490.8672
Epoch 48/50
277/277 [==============================] - 0s 2ms/step - loss: 106151.3828 -
mean_absolute_error: 106151.3828 - val_loss: 103520.8672 -
val_mean_absolute_error: 103520.8672
Epoch 49/50
277/277 [==============================] - 1s 2ms/step - loss: 106128.6484 -
mean_absolute_error: 106128.6484 - val_loss: 103634.1016 -
val_mean_absolute_error: 103634.1016
```

```
Epoch 50/50
277/277 [==============================] - 0s 2ms/step - loss: 105923.1953 -
mean_absolute_error: 105923.1953 - val_loss: 103399.5391 -
val_mean_absolute_error: 103399.5391
```

[30]:
```python
# Visualisation
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Val Loss')
plt.xlabel('Epoch')
plt.legend(['Train'], loc='upper left')
plt.show()
```



4.7 Prediction of Neural Network Model

[31]:
```python
X_test_numpy = np.array(X_test)
y_test_numpy = np.array(y_test)
```

[32]:
```python
# Data scatter of predicted values
y_pred = neural_network.predict(X_test_numpy)
y_pred = np.reshape(y_pred, (y_pred.shape[0],))
```

```
y_test_numpy, y_pred
```

```
136/136 [==============================] - 0s 805us/step
```

[32]:
```
(array([ 297000., 1578000.,  562100., …,  369950.,  300000.,  575950.]),
 array([ 438204.66, 1385815.5 ,  560890.  , …,  401254.9 ,  251818.45,
         390145.7 ], dtype=float32))
```

[33]:
```
# p-value
r2_score(y_test_numpy, y_pred)
```

[33]:
```
0.7394895122908761
```

4.8 Implementation des Neural Network modèles avec des différentes learning rates

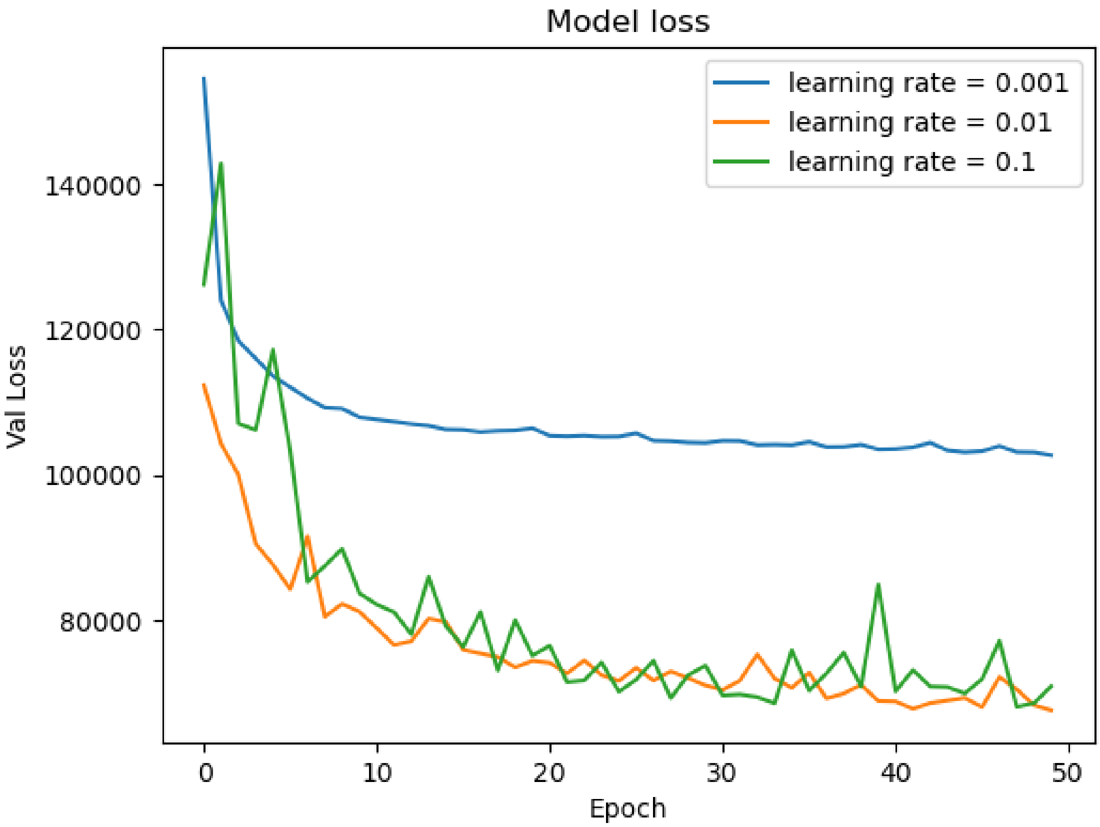
[34]:
```
def neural_network(num_layer, units, learning_rate=0.001):
    """
    Parameters
    ----------
    num_layer       : int
                      number of layers
    units           : list
                      number of units in each layer
    learning_rate   : float
                      learning rate of the optimizer of the model
    """
    if (len(units) != num_layer):
        raise ValueError("Number of list of units must be equal to number of␣
↪layers")
    model = Sequential()
    model.add(Dense(units=units[0], kernel_initializer='normal',␣
↪input_dim=X_train.shape[1], activation="relu"))
    for i in range(1, num_layer-1):
        model.add(Dense(units=units[i], kernel_initializer='normal',␣
↪activation="relu"))
    model.add(Dense(units=1, activation="linear"))
    model.compile(
        loss=tf.keras.losses.MeanAbsoluteError(),
        optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
        metrics=[tf.keras.metrics.MeanAbsoluteError()]
    )
    return model
```

[35]:
```
# Itération
histories_neural_network = []
def neural_network_score(num_layer, units, learning_rate=[0.001, 0.01, 0.1]):
    """
```

```
    Parameters
    ----------
    num_layer       : int
                      number of layers
    units           : list
                      number of units in each layer
    learning_rate   : list
                      learning rate of the optimizer of the model
    """
    if (len(units) != num_layer):
        raise ValueError("Number of list of units must be equal to number of␣
↪layers")
    for i in range(len(learning_rate)):
        model = neural_network(num_layer, units, learning_rate=learning_rate[i])
        model.save(f"neural_network_model_{i}.h5")
        history = model.fit(
            X_train_numpy, y_train_numpy,
            epochs=50,
            batch_size=50,
            validation_split=0.2
        )
        histories_neural_network.append(history)
        y_pred = model.predict(X_test)
        y_pred = np.reshape(y_pred, (y_pred.shape[0],))
        print(f"R2 score on test set with learning rate {learning_rate[i]} is␣
↪{r2_score(y_test_numpy, y_pred)}")
        print(f"Mean absolute error on test set with learning rate␣
↪{learning_rate[i]} is {history.history['val_loss'][-1]}")
```

[ ]:
```
num_layer = 4
units = [128, 256, 256, 1]
neural_network_score(num_layer, units)
```

[37]:
```
# Visualisation
plt.plot(histories_neural_network[0].history['val_loss'])
plt.plot(histories_neural_network[1].history['val_loss'])
plt.plot(histories_neural_network[2].history['val_loss'])
plt.title('Model loss')
plt.ylabel('Val Loss')
plt.xlabel('Epoch')
plt.legend(["learning rate = 0.001", "learning rate = 0.01", "learning rate = 0.
↪1"], loc='upper right')
plt.show()
```



Nous pouvons voir que avec `learning rate = 0.01`, le modèle converge plus vite que le modèle avec `learning rate = 0.001`. Il est plus stable que le modèle avec `learning rate = 0.1`. Nous choisissons donc le modèle avec `learning rate = 0.01` comme modèle plus optimisé.