Lasso Regularization

Module 1 | Chapter 1 | Notebook 9

After getting to know *Ridge* in the previous lesson, we will look at lasso regularization in this lesson. It is particularly practical for *feature selection*. This notebook will cover the following concept:

- Lasso Regression

Lasso Regression

Scenario: A Taiwanese investor comes to you to find out how much his properties in Taiwan are actually worth. He might want to resell them. The data for the houses he needs predictions for is located in *Taiwan_real_estate_prediction_data.xlsx*.

After the overfitting problems in the lesson before last, he is now looking for an optimal solution with as many features as feasibly possible, but not more than that. The training data is in *Taiwan_real_estate_training_data.xlsx*.

The model quality should be evaluated with the data in *Taiwan_real_estate_test_data.xlsx*. If it is good, the investor wants to know how much his properties are worth.

In order to make rapid progress, let's import the data and divide it into `df_train` (training data), `df_test` (test data to validate the models) and `df_aim` (prediction data not including property prices).

In [59]:

```
import pandas as pd
import numpy as np
from sklearn.metrics import *
df_train = pd.read_excel('Taiwan_real_estate_training_data.xlsx', index_col='No')
df_test = pd.read_excel('Taiwan_real_estate_test_data.xlsx', index_col='No')
df_aim = pd.read_excel('Taiwan_real_estate_prediction_data.xlsx', index_col='No')

col_names = ['house_age',
             'metro_distance',
             'number_convenience_stores',
             'number_parking_spaces',
             'air_pollution',
             'light_pollution',
             'noise_pollution',
             'neighborhood_quality',
             'crime_score',
             'energy_consumption',
             'longtitude',
             'price_per_ping']
df_train.columns = col_names
```

```
df_test.columns = col_names
df_aim.columns = col_names

print(df_aim.loc[:, 'price_per_ping'])
print(df_aim.loc[:, 'price_per_ping'].unique())

df_train.loc[:, 'price_per_m2'] = df_train.loc[:, 'price_per_ping'] / 3.3
df_test.loc[:, 'price_per_m2'] = df_test.loc[:, 'price_per_ping'] / 3.3
df_aim.loc[:, 'price_per_m2'] = df_aim.loc[:, 'price_per_ping'] / 3.3

df_train = df_train.drop('price_per_ping', axis=1)
df_test = df_test.drop('price_per_ping', axis=1)
df_aim = df_aim.drop('price_per_ping', axis=1)

features_train = df_train.drop('price_per_m2', axis=1)
features_test = df_test.drop('price_per_m2', axis=1)

target_train = df_train.loc[:,'price_per_m2']
target_test = df_test.loc[:,'price_per_m2']
```

```
No
1     NaN
2     NaN
3     NaN
4     NaN
5     NaN
6     NaN
7     NaN
8     NaN
9     NaN
10    NaN
Name: price_per_ping, dtype: float64
[nan]
```

Once again, the data dictionary for this data is as follows:

Column number	Column name	Type	Description
0	`'house_age'`	continuous (`float`)	age of the house in years
1	`'metro_distance'`	continuous (`float`)	distance in meters to the next metro station
2	`'number_convenience_stores'`	continuous (`int`)	Number of convenience stores nearby
3	`'number_parking_spaces'`	continuous (`int`)	Number of parking spaces nearby
4	`'air_pollution'`	continuous (`float`)	Air pollution value near the house
5	`'light_pollution'`	continuous (`float`)	Light pollution value near the house
6	`'light_pollution'`	continuous (`float`)	Light pollution value near the house

Column number	Column name	Type	Description
7	`'neighborhood_quality'`	continuous (`float`)	average quality of life in the neighborhood
8	`'crime_score'`	continuous (`float`)	crime score according to police
9	`'energy_consumption'`	continuous (`float`)	The property's energy consumption
10	`'longitude'`	continuous (`float`)	The property's longitude
11	'price_per_ping'	continuous (`float')	House price in Taiwan dollars per ping, one ping is 3.3 m^2
12	`'price_per_ping'`	continuous (`'float'`)	House price in Taiwan dollars per m^2

In the last lesson we saw that too many features in the linear regression model led to overfitting. The trained parameters can then no longer be used to predict new, independent data. The only way to prevent overfitting is regularization and to simplify the model. In principle, the fewer features a model considers, the simpler it is. But how do you choose the best features in a data-driven way? Lasso regularization is a very practical option.

A lasso regression is actually barely different from a ridge regression. The only difference is that in a lasso regression, the absolute slope values are minimized instead of the square of the slope values, as is the case with ridge regression. Lasso regression therefore adjusts the parameters to the data with two objectives:

- Keep the difference between predicted and actual target values as small as possible
- Keep the sum of the absolute slopes (e.g. $|slope_1| + |slope_2|$) as small as possible.

As with ridge regression, the α hyperparameter controls how the two objectives are balanced. With `alpha=0` the second objective (regularization) is disregarded. Then the lasso regression would be a normal linear regression. With an infinitely high `alpha`, the first objective disregarded. In this case all slopes are zero.

If you are interested, the official `sklearn` documentation provides all the information you need. We have summarized the most important information for you here:

```
Lasso(
    alpha= float,        #strength of penalty for regularization
    fit_intercept=True, #fit intercept in underlying linear regression
    random_state=None,  #random seed used for data shuffling
)
```

Lasso regression tends to set a lot of slopes to zero. This means that the features with a slope of zero will not even be used for the prediction. So you can generally disregard them and select the most important features.

Now use a lasso regression with `alpha=1`. `Lasso` is located in `sklearn.linear_model`. Store the model in `model_lasso`. Fit the model to the data with all eleven features. Note that the features should be standardized, as with the ridge regression. Print the slopes at the end. Are any of the slopes actually zero?

In [15]:
```
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(features_train)
features_train_standardized = scaler.transform(features_train)

model_lasso = Lasso(alpha=1)
model_lasso.fit( features_train, target_train)
model_lasso.fit(features_train_standardized, target_train)

print(model_lasso.coef_)
print(features_train.columns)
```

```
[-0.         -1.47427896  0.46369063  0.          -0.          -0.
 -0.04670917  0.          0.          0.          0.        ]
Index(['house_age', 'metro_distance', 'number_convenience_stores',
       'number_parking_spaces', 'air_pollution', 'light_pollution',
       'noise_pollution', 'neighborhood_quality', 'crime_score',
       'energy_consumption', 'longtitude'],
      dtype='object')
```

Almost all features were set to zero. Only `metro_distance`, `number_convenience_stores` and `noise_pollution` have survived and would be used to make predictions with this model.

Next you can evaluate the `model_lasso` model using the test data. Note that the features of the test data should be standardized exactly as the training data was before. Print the *mean squared error*, *RMSE* and R^2 values of `model_lasso` and the test data.

In [21]:
```
from sklearn.metrics import mean_squared_error, r2_score

features_test_standardized = scaler.transform(features_test)
target_test_pred_lasso = model_lasso.predict(features_test_standardized)

print('MSE: ', mean_squared_error(target_test, target_test_pred_lasso))
print('RMSE: ', np.sqrt(mean_squared_error(target_test, target_test_pred_lasso)))
print('R2: ', r2_score(target_test, target_test_pred_lasso))
```

```
MSE:  8.571168547622161
RMSE:  2.927655810989769
R2:  0.3520342491334303
```

If we now compare these values with those of the other models, it becomes apparent that the lasso model delivers the best performance.

Model	Test: *MSE*	Test: R^2
`model_age`	11.89	10.1%
`model_metro`	10.27	22.4%

Model	Test: *MSE*	Test: R^2
`model_stores`	11.29	14.7%
`model_multiple`	9.72	26.5%
`model_multiple_all`	31.77	-140.2%
`model_ridge`	10.83	18.1%
`model_lasso`	8.57	35.2%

Congratulations: You have now learned about two regression models with regularization. In practice, ridge regression is used more against overfitting and correlated features and lasso regression is used to identify the most important features. Lasso regressions are also often used when there is a very large number of features. In our case the lasso model is also the best model according to quality metrics. So let's use it to predict the real estate investor's house prices as best we can.

Predicting house prices with lasso regression

Scenario: The Taiwanese investor would now like to know how much his properties are worth. Predict their value with the best model you have found (`model_lasso`). You will find the data in `df_aim` .

In [61]:
```
df_aim =  df_aim.dropna(axis=0, how='all')

features_aim = df_aim.drop('price_per_m2', axis=1)
target_aim = df_aim.loc[:,'price_per_m2']

features_aim_standardized = scaler.transform(features_aim)
target_aim_pred_lasso = model_lasso.predict(features_aim_standardized)
print(target_aim_pred_lasso.mean())

#print('MSE: ', mean_squared_error(target_aim, target_aim_pred_lasso))
#print('RMSE: ', np.sqrt(mean_squared_error(target_aim, target_aim_pred_lasso)))
#print('R2: ', r2_score(target_aim, target_aim_pred_lasso))
```

```
12.876347921105019
```

Congratulations: You got an impression of overfitting and how to avoid it. In the end, the best model was a lasso regression that predicted an average house price of $12.88 / m^2. The real estate investor is very happy with this. He thanks you for all your hard work and hopes you enjoy the rest of the course!

Remember:

- `Lasso` minimizes the sum of the absolute slope values.
- `Lasso` is suitable for preventing overfitting and for *feature selection*.

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.
