DataScientest • com

Introduction to Machine Learning with scikit-learn

Linear regression

Introduction

> The objective of this exercise is to become familiar with linear regression. Linear regression was one of the first predictive models to be studied and is today one of the most popular models for practical applications thanks to its simplicity.

Univariate Linear Regression

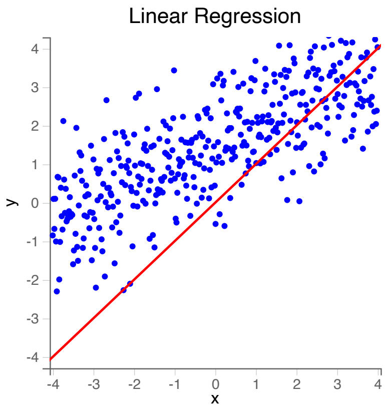
> In the univariate linear model, we have two variables: y called the **target** variable and x called the **explanatory variable**. Linear regression consists in modeling the link between these two variables by an **affine function**. Thus, the formula of the univariate linear model is given by:
>
> $$y \approx \beta_1 x + \beta_0$$
>
> where:
>
> - y is the variable we want to predict.
> - x is the explanatory variable.
> - β_1 and β_0 are the parameters of the affine function. β_1 will define its **slope** and β_0 will define its **y-intercept** (also called **bias**).
>
> **The goal of linear regression is to estimate the best parameters β_0 and β_1 to predict the variable y from a given value of x.**

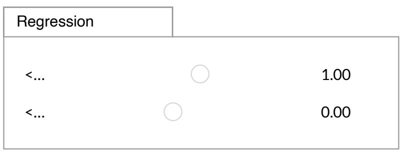
To get a feel for Univariate Linear Regression, let us look at the interactive example below.

- **(a)** Run the next cell to display the interactive figure. In this figure, we have simulated a dataset by the relation $y = \alpha_1 x + \alpha_0$.
- **(b)** Use the sliders on the `Regression` tab to find the parameters β_0 and β_1 that **best match** all the points in the data set.
- **(c)** What is the effect of each of the parameters on the regression function?

In [1]:
```
from widgets import regression_widget

regression_widget()
```





Multivariate Linear Regression

> Multivariate linear regression consists in modeling a linear link between a target variable y and **several explanatory variables** $x_1, x_2, ..., x_p$, often called *features*:
>
> $$\begin{aligned} y &\approx \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p \\ &\approx \beta_0 + \sum_{j=1}^{p} \beta_j x_j \end{aligned}$$
>
> There are now $p + 1$ parameters β_j to find.

We are now going to learn how to use the **`scikit-learn`** library in order to solve a Machine Learning problem with a **linear regression**.

During the following exercises, the objective will be to predict the **selling price of a car** based on its **characteristics**.

Importing the dataset

The dataset that we will use in the following contains many characteristics about different cars from 1985.

For simplicity, only the numeric variables have been kept and the lines containing missing values have been deleted.

- **(a)** Import the `pandas` module under the alias `pd` .
- **(b)** In a `DataFrame` named `df` , import the `automobiles.csv` dataset using the `read_csv` function of `pandas` . This file is located in the same folder as the runtime environment of the notebook.

In [2]:

```
# Insert your code here

import pandas as pd

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

df.head()
```

Out[2]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price
0	2	164	99.8	176.6	66.2	54.3	2337	109	3.19	3.4	10.0	102	5500	24	30	13950
1	2	164	99.4	176.6	66.4	54.3	2824	136	3.19	3.4	8.0	115	5500	18	22	17450
2	1	158	105.8	192.7	71.4	55.7	2844	136	3.19	3.4	8.5	110	5500	19	25	17710
3	1	158	105.8	192.7	71.4	55.9	3086	131	3.13	3.4	8.3	140	5500	17	20	23875
4	2	192	101.2	176.8	64.8	54.3	2395	108	3.50	2.8	8.8	101	5800	23	29	16430

Hide solution

In [3]:

```
import pandas as pd

df = pd.read_csv("automobiles.csv")
df.head(5)
```

Out[3]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price
0	2	164	99.8	176.6	66.2	54.3	2337	109	3.19	3.4	10.0	102	5500	24	30	13950
1	2	164	99.4	176.6	66.4	54.3	2824	136	3.19	3.4	8.0	115	5500	18	22	17450
2	1	158	105.8	192.7	71.4	55.7	2844	136	3.19	3.4	8.5	110	5500	19	25	17710
3	1	158	105.8	192.7	71.4	55.9	3086	131	3.13	3.4	8.3	140	5500	17	20	23875
4	2	192	101.2	176.8	64.8	54.3	2395	108	3.50	2.8	8.8	101	5800	23	29	16430

- The `symboling` variable corresponds to the degree of risk with respect to the insurer (risk of accident, breakdown, etc.).
- The `normalized_losses` variable is the relative average cost per year of vehicle insurance. This value is normalized with respect to cars of the same type (SUV, utility, sports, etc.).
- The following 13 variables concern the technical characteristics of the cars such as width, length, engine displacement, horsepower, etc ...
- The last variable `price` corresponds to the selling price of the vehicle. This is the variable that we will try to predict.

Separation of the explanatory variables from the target variable

We are now going to create two `DataFrames` , one containing the explanatory variables and another containing the target variable `price` .

- **(d)** In a `DataFrame` named `X` , make a copy of the explanatory variables of our data set, that is to say all the variables **except** `price` .
- **(e)** In a `DataFrame` named `y` , make a copy of the target variable `price` .

```
X = df.iloc[:,:-1]
#or
X = df.drop(['price'], axis = 1)

y = df.iloc[:, -1:]
#or
y = df['price']
```

Hide solution

In [5]:
```
# Explanatory variables
X = df.drop(['price'], axis = 1)

# Target variable
y = df['price']
```

Splitting of the data into training and test sets

> We are now going to split our dataset into two sets : A **training** set and a **test** set. This step is **extremely** important when doing Machine Learning.
>
> Indeed, as their names indicate:
>
> - The training set is used to train the model, ie to find the optimal $\beta_0, ..., \beta_p$ parameters for this datase t.
> - The test set is used to test the trained model by evaluating its ability to **generalize** its predictions on data that it has **never seen** .
>
> A very useful function for doing this is the `train_test_split` function of the `model_selection` submodule of **`scikit-learn`** .

- **(f)** Run the following cell to import the `train_test_split` function.

In [6]:
```
from sklearn.model_selection import train_test_split
```

> This function is used as follows:
>
> `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)`
>
> - `X_train` and `y_train` are the explanatory and target variables of the **training** dataset.
> - `X_test` and `y_test` are the explanatory and target variables of the **test** dataset.
> - The `test_size` parameter corresponds to the **proportion** of the dataset that we want to keep for the test set. In the previous example, this proportion corresponds to 20% of the initial dataset.

- **(g)** Using the `train_test_split` function, separate the dataset into a training set (`X_train` , `y_train`) and a test set (`X_test` , `y_test`) so that the test set contains **15% of the initial dataset**.

In [7]:
```
# Insert your code here

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.15)
```

Hide solution

In [8]:
```
# Splitting the dataset into a training set (85%) and a test set (15%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

Training the regression model

> To train a linear regression model on this dataset, we will use the **`LinearRegression`** class contained in the `linear_model` submodule of `scikit-learn` .

- **(h)** Run the following cell to import the `LinearRegression` class.

The `scikit-learn` API makes it easy to train and evaluate models. All scikit-learn model classes have the following two methods:

- **`fit`** : Train the model on the dataset given as input.
- **`predict`** : Make a prediction from a set of explanatory variables given as input.

Below is an example of training a model with scikit-learn:

```
# Instantiation of the model
linreg = LinearRegression()

# Training the model on the training set
linreg.fit(X_train, y_train)

# Prediction of the target variable for the test dataset. These predictions are stored in y_pred.
y_pred = linreg.predict(X_test)
```

- **(i)** Instantiate a `LinearRegression` model named **`lr`** .
- **(j)** Train `lr` on the training dataset.
- **(k)** Make a prediction on the training data. Store these predictions in `y_pred_train` .
- **(l)** Make a prediction on the test data. Store these predictions in `y_pred_test` .

In [10]:

```
# Insert your code here

lr = LinearRegression()

lr.fit(X_train, y_train)

y_pred_train = lr.predict(X_train)

y_pred_test = lr.predict(X_test)

dict1 = {'y_pred_train' : y_pred_train,
         'y_train_true' : y_train,
         'y_train_difference' : (y_pred_train - y_train),
         'difference_squared' : (y_pred_train - y_train)**2 }

df_train = pd.DataFrame(dict1)
df_train = df_train.astype(int)

print(df_train)

print(y_pred_train.shape, y_pred_test.shape)

print("MSE train : ",df_train.difference_squared.mean())
```

```
     y_pred_train  y_train_true  y_train_difference  difference_squared
55           9414          6989                2425             5882014
116          6905          7198                -292               85615
92          13104         12170                 934              873798
32          29187         32250               -3062             9379680
3           21302         23875               -2572             6615606
..            ...           ...                 ...                 ...
19           4308          6479               -2170             4710712
39          10317          8495                1822             3320893
105          9276          7463                1813             3289207
100          8773          7126                1647             2714076
10           5935          6575                -639              408879

[135 rows x 4 columns]
(135,) (24,)
MSE train :  4955113.088888889
```

Hide solution

In [11]:

```
# Instantiation of the model
lr = LinearRegression()

# Training the model
lr.fit(X_train, y_train)

# Prediction of the target variable for the TRAIN dataset
y_pred_train = lr.predict(X_train)

# Prediction of the target variable for the TEST dataset
y_pred_test = lr.predict(X_test)
```

In order to evaluate the **quality of the predictions of the model** obtained thanks to the parameters $\beta_0, ..., \beta_j$, there are several metrics already built in the `scikit-learn` library.

One of the most used metrics for regression is the **Mean Squared Error** (MSE) which is defined under the name of `mean_squared_error` in the `metrics` submodule of `scikit-learn`.

This function consists in calculating the average of the squared distances between the **target variables** and the **predictions obtained** thanks to the regression function.

The following interactive figure shows how this error is calculated according to β_1:

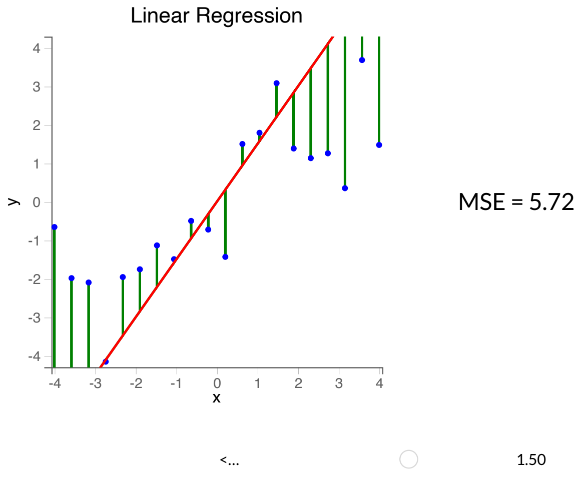
- The **blue** dots represent the **dataset** for which we want to evaluate the quality of the predictions. Usually this is the **test** dataset.
- The **red** line is the regression function configured by β_1. In this example, β_0 is set to 0 to simplify the illustration.
- The **green** lines are the **distances** between the **target variable** and the **predictions** obtained thanks to the regression function parameterized by β_1.

The mean squared error is just the average of these squared distances.

- **(m)** Run the next cell to display the interactive figure.
- **(n)** Using the cursor below the figure, try to find a value of β_1 that minimizes the Mean Squared Error. Is this value unique?

In [12]:
```
from widgets import interactive_MSE

interactive_MSE()
```



The `mean_squared_error` function of `scikit-learn` is used as follows:

```
mean_squared_error(y_true, y_pred)
```

where:

- `y_true` contains the true values of the target variable.
- `y_pred` contains the values **predicted** by our model for the same explanatory variables.

- **(o)** Import the **`mean_squared_error`** function from the `sklearn.metrics` submodule.
- **(p)** Evaluate the prediction quality of the model on **training data**. Store the result in a variable named `mse_train`.
- **(q)** Evaluate model prediction quality on **test data**. Store the result in a variable named `mse_test`.
- **(r)** Why is the MSE higher on the test dataset?

```
from sklearn.metrics import mean_squared_error

lr = LinearRegression()

lr.fit(X_train, y_train)

y_pred_train = lr.predict(X_train)

y_pred_test = lr.predict(X_test)

mse_train = mean_squared_error(y_train, y_pred_train)

mse_test = mean_squared_error(y_test, y_pred_test)

print(mse_train, 'MSE train')
print(mse_test, 'MSE test')
```

```
4955113.591604198 MSE train
6809280.471285641 MSE test
```

Hide solution

In [14]:

```
from sklearn.metrics import mean_squared_error

# Calculation of the MSE between the target variable and the predictions made on the training dataset
mse_train = mean_squared_error(y_train, y_pred_train)

# Calculation of the MSE between the target variable and the predictions made on the test dataset
mse_test = mean_squared_error(y_test, y_pred_test)

print("MSE train lr:", mse_train)
print("MSE test lr:", mse_test)
```

```
MSE train lr: 4955113.591604198
MSE test lr: 6809280.471285641
```

> The mean squared error you will find should be around millions on the test data, which can be difficult to interpret.
>
> This is why we are going to use another metric, the **Mean Absolute Error** which is at the same scale as the target variable.

- **(s)** Import the `mean_absolute_error` function from the `sklearn.metrics` submodule.
- **(t)** Evaluate the prediction quality on test and training data using the mean absolute error.
- **(u)** From the `DataFrame` `df`, calculate the average purchase price on all vehicles. Do the model's predictions seem reliable to you?

In [15]:

```
# Insert your code here

from sklearn.metrics import mean_absolute_error

mae_train = mean_absolute_error(y_train, y_pred_train)

mae_test = mean_absolute_error(y_test, y_pred_test)

print(mae_train, 'MAE train')
print(mae_test, 'MAE test')

print("Avg Price : ", df['price'].mean())
```

```
1685.244633500678 MAE train
1770.072002663633 MAE test
Avg Price :  11445.729559748428
```

Hide solution

```
# Calculation of the MAE between the target variable and the predictions made on the training dataset
mae_train = mean_absolute_error(y_train, y_pred_train)

# Calculation of the MAE between the target variable and the predictions made on the test dataset
mae_test = mean_absolute_error(y_test, y_pred_test)

print("MAE train lr:", mae_train)
print("MAE test lr:", mae_test)

mean_price = df['price'].mean()

print("\nRelative error", mae_test / mean_price)

# The mean absolute error is around 20% of the average price, which is not optimal
# but is still a good baseline for testing more advanced models.
```

```
MAE train lr: 1685.244633500678
MAE test lr: 1770.072002663633

Relative error 0.15464911986812122
```

2. Overfitting the data with another regression model

> We have just seen that with the `LinearRegression` class of `scikit-learn`, the model was able to learn on the training data and generalize on the test data with an error rate of 20% on average.
>
> In what follows we will create another regression model that **learns very well on training data but generalizes very poorly on test data**: this is called **overfitting**.
>
> For this we will use a Machine Learning model called **Gradient Boosting Regressor** known for its tendancy to overfit.

- **(a)** Run the following cell to import the `GradientBoostingRegressor` class contained in the `ensemble` submodule of `scikit-learn` and instantiate a `GradientBoostingRegressor` model named **`gbr`**.

In [17]:

```
from sklearn.ensemble import GradientBoostingRegressor

# These parameters have been chosen to overfit on purpose
# Do not use them in practice
gbr = GradientBoostingRegressor(n_estimators = 1000,
                                max_depth = 10000,
                                max_features = 15,
                                validation_fraction = 0)
```

- **(b)** Train the model **`gbr`** using its **`fit`** method.
- **(c)** Make predictions on the test and training datasets. Store these predictions in `y_pred_test_gbr` and `y_pred_train_gbr`.

In [18]:

```
# Insert your code here

gbr.fit(X_train, y_train)

y_pred_train_gbr = gbr.predict(X_train)

y_pred_test_gbr = gbr.predict(X_test)
```

Hide solution

In [19]:

```
# Training the model on the training dataset
gbr.fit(X_train, y_train)

# Prediction of the target variable for the TRAIN dataset
y_pred_train_gbr = gbr.predict(X_train)

# Prediction of the target variable for the TEST dataset
y_pred_test_gbr = gbr.predict(X_test)
```

> After instantiating our model, training it on the training data and making the predictions, we must then evaluate its performance.

- **(d)** Calculate the MSE on the **training** data and the **test** data using the `mean_squared_error` function then display the results.
- **(e)** Calculate the MAE for the **training** data and the **test** data using the `mean_absolute_error` function then display the results .
- **(f)** After having calculated the average of the `price` column, calculate the **relative error of the model** on the test set.

```
mse_train = mean_squared_error(y_train, y_pred_train_gbr)

mse_test = mean_squared_error(y_test, y_pred_test_gbr)

print(mse_train, "MSE train")
print(mse_test, "MSE test\n")

mae_train = mean_absolute_error(y_train, y_pred_train_gbr)

mae_test = mean_absolute_error(y_test, y_pred_test_gbr)

print(mae_train, "MAE train")
print(mae_test, "MAE test\n")

mean_price_gbr = df.price.mean()
print("Relative Error : ", mae_test / mean_price_gbr)
```

```
29882.92222222222 MSE train
5051790.565869872 MSE test

40.49629629630798 MAE train
1606.3345321880659 MAE test

Relative Error :  0.1403435686473945
```

Hide solution

In [21]:

```
### MSE

# Calculation of the MSE between the target variable and the predictions made on the training dataset
mse_train_gbr = mean_squared_error(y_train, y_pred_train_gbr)

# Calculation of the MSE between the target variable and the predictions made on the test dataset
mse_test_gbr = mean_squared_error(y_test, y_pred_test_gbr)

print("MSE train gbr:", mse_train_gbr)
print("MSE test gbr:", mse_test_gbr, "\n")


### MAE

# Calculation of the MAE between the target variable and the predictions made on the training dataset
mae_train_gbr = mean_absolute_error(y_train, y_pred_train_gbr)

# Calculation of the MAE between the target variable and the predictions made on the test dataset
mae_test_gbr = mean_absolute_error(y_test, y_pred_test_gbr)

print("MAE train gbr:", mae_train_gbr)
print("MAE test gbr:", mae_test_gbr, "\n")

mean_price_gbr = df ['price'].mean()

print("Relative error", mae_test_gbr / mean_price_gbr)
```

```
MSE train gbr: 29882.92222222222
MSE test gbr: 5051790.565869872

MAE train gbr: 40.49629629630798
MAE test gbr: 1606.3345321880659

Relative error 0.1403435686473945
```

Here is an example of results that we could obtain with these two models.

For the **linear regression with `LinearRegression`** we had:

- MAE train lr = 1588.131591267774
- MAE test lr = 2105.5002712214014

For the **regression with `GradientBoostingRegressor`** we have:

- MAE train gbr: 27.533333333339847
- MAE test gbr: 1393.013371545563

The mean absolute error obtained on the training set by the `GradientBoostingRegressor` model is only 27.5 against 1588 for the linear regression. The `GradientBoostingRegressor` model is very powerful and is able to learn the training data almost **"by heart"** which explains this difference in performance.

It is for this reason that the performance of the model should be evaluated on the **test** dataset. Indeed, the average absolute error of the `GradientBoostingRegressor` model is 1393, which is **very far** from the performance obtained on the training data.

This is an example of **blatant overfitting**. Even if the performance of the `GradientBoostingRegressor` is superior to that of the linear regression on the test data, you should always **be wary** of too high a performance.

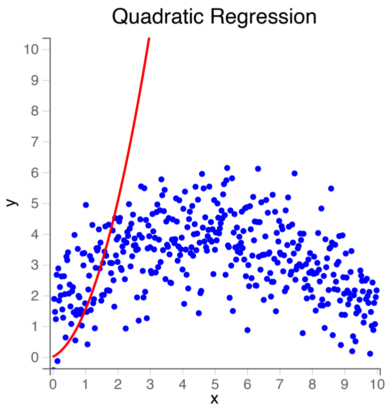
3. Going further: Polynomial Regression

$$y = \beta_0 + \beta_1 x + \beta_2 x^2$$

- **(a)** Run the next cell to display the interactive figure.

In [22]:
```
from widgets import polynomial_regression

polynomial_regression()
```



<... 0.00
<... 0.50
<... 1.00

> Polynomial regression is equivalent to performing a classical linear regression from **polynomial functions of the explanatory variable** of arbitrary degree. Polynomial regression is much more flexible than classical linear regression because it can approach any type of continuous function.
>
> When we have several explanatory variables, the polynomial variables can also be calculated by products between the explanatory variables. For example, if we have three variables, then the **second-order** polynomial regression model becomes:
>
> $$y \approx \beta_0 + \beta_1 x_1^2 + \beta_2 x_2^2 + \beta_3 x_3^2 + \beta_4 x_1 x_2 + \beta_5 x_2 x_3 + \beta_6 x_1 x_3$$
>
> If we had more explanatory variables or wanted to increase the degree of polynimial regression, the number of explanatory variables **would explode**, which could induce **overfitting**.

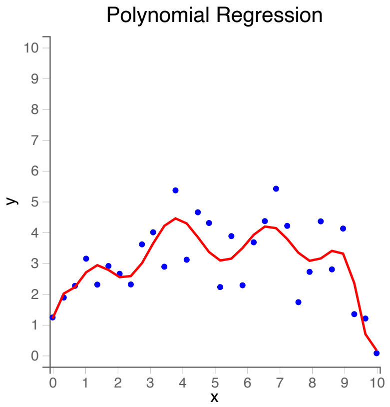
- **(d)** Run the next cell to display the interactive figure.

> The scatter plot was generated with the same trend as the previous figure. The red line corresponds to the optimal polynomial regression function obtained on these data.

- **(e)** Taking into account the scatter plot in the previous figure, find the **degree** of the polynomial regression that best captures the trend of the data.
- **(f)** Set `d` to 20. Do you think this regression function would give good predictions on the scatter plot in the previous figure?

In [23]:
```
from widgets import polynomial_regression2

polynomial_regression2()
```



<p>*d*</p> 12

PolynomialFeatures class of the `preprocessing` submodule:

```
from sklearn.preprocessing import PolynomialFeatures

poly_feature_extractor = PolynomialFeatures(degree = 2)
```

- The **`degree`** parameter defines the degree of the polynomial features to be calculated.

The `poly_feature_extractor` object **is not a prediction model**. This type of object is called a **`Transformer`** and it can be used with the following two methods:

> - **`fit`** : **does nothing** in this case. This method is generally used to calculate the parameters necessary to apply a transformation to the data.
> - **`transform`** : Applies the transformation to the dataset. In this case, the method returns the polynomial features of the dataset.

These two methods can be called **simultaneously** using the **`fit_transform`** method. We can compute the polynomial features on `X_train` and `X_test` as follows:

```
X_train_poly = poly_feature_extractor.fit_transform(X_train)

X_test_poly = poly_feature_extractor.transform(X_test)
```

- **(g)** Import the `PolynomialFeatures` class from the `preprocessing` submodule of `sklearn`.
- **(h)** Instantiate an object of class `PolynomialFeatures` with the argument **`degree = 3`** and name it `poly_feature_extractor`.
- **(i)** Apply the transformation of `poly_feature_extractor` on `X_train` and `X_test` and store the results in `X_train_poly` and `X_test_poly`.

In [24]:

```
# Insert your code here

from sklearn.preprocessing import PolynomialFeatures

poly_feature_extractor = PolynomialFeatures(degree = 3)

X_train_poly = poly_feature_extractor.fit_transform(X_train)

X_test_poly = poly_feature_extractor.transform(X_test)
```

Hide solution

In [25]:

```
from sklearn.preprocessing import PolynomialFeatures

poly_feature_extractor = PolynomialFeatures(degree = 3)

# Applying the transformation on X_train et X_test
X_train_poly = poly_feature_extractor.fit_transform(X_train)
X_test_poly = poly_feature_extractor.transform(X_test)
```

- **(j)** Train a linear regression model on the data (`X_train_poly` , `y_train`).
- **(k)** Evaluate its performance on training data and test data (`X_test_poly` , `y_test`). Are we in an overfitting regime?

In [26]:

```
# Insert your code here

lr = LinearRegression()

lr.fit(X_train_poly, y_train)

y_pred_train = lr.predict(X_train_poly)

mae_train = mean_absolute_error(y_train, y_pred_train)

print("MAE train : ",mae_train)

y_pred_test = lr.predict(X_test_poly)

mae_test = mean_absolute_error(y_test, y_pred_test)

print("MAE test : ", mae_test)
```

```
MAE train :  52.73664560603254
MAE test :  79094.70250124
```

Hide solution

```
polyreg = LinearRegression()

# Training of the model on polynomial features
polyreg.fit(X_train_poly, y_train)

# Evaluation of the model on the training data
y_pred_train = polyreg.predict(X_train_poly)
print("MAE Train:", mean_absolute_error(y_train, y_pred_train), '\n')


# Evaluation of the model on the test data
y_pred_test = polyreg.predict(X_test_poly)
print("MAE Test:", mean_absolute_error(y_test, y_pred_test), '\n')


print("We are absolutely in an overfitting regime.")
print("The polynomial regression model performs well on training data but not on test data.")
print("The third-order polynomial regression model performs significantly worse than a simple linear regression.")
```

```
MAE Train: 52.73664560603254

MAE Test: 79094.70250124

We are absolutely in an overfitting regime.
The polynomial regression model performs well on training data but not on test data.
The third-order polynomial regression model performs significantly worse than a simple linear regression.
```

Conclusion and recap

In this course, you have been introduced to solving a regression problem with machine learning.

We used the `scikit-learn` library to instantiate regression models like `LinearRegression` or `GradientBoostingRegressor` and also apply transformations on the data like extracting polynomial features.

The **different steps** that we have studied are the basis of any solution to a Machine Learning problem:

- The data is prepared by separating the **explanatory** variables from the **target** variable.
- We **split** the dataset into two sets (a **training** set and a **test** set) using the `train_test_split` function of the `sklearn.model_selection` submodule.
- We **instantiate** a model like `LinearRegression` or `GradientBoostingRegressor` thanks to the class' constructor.
- We **train** the model on the training dataset using the **`fit`** method.
- We perform a **prediction** on the test dataset using the **`predict`** method.
- We **evaluate the performance** of our model by calculating the error between these predictions and the true values of the target variable from the **test** data.

The performance evaluation for a regression model is easily done using the **`mean_squared_error`** or **`mean_absolute_error`** functions of the `metrics` submodule of `sklearn` .

Validate