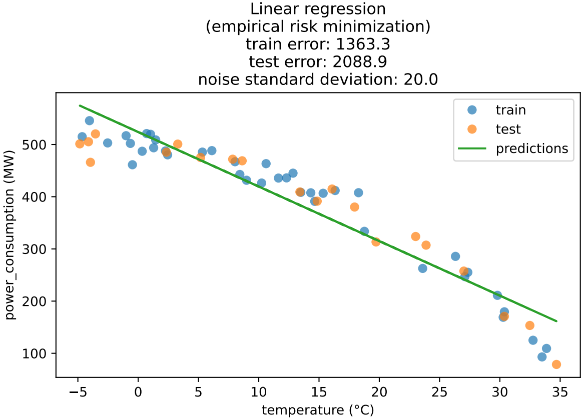
Machine Learning Tek 5: train and test set, regression.



Content

Train sets and test sets, loss functions

Linear regression in one dimension

Polynomial regression and overfitting

General linear regression : Ordinary least squares, ridge regression and hyperparameters

Linear regression

Linear regression is one of the most elementary methods used in ML regression problems. It is useful for many applications, and is often a component of more complex methods.
We will use is to illustrate several important aspects of ML that are also encountered when using other methods (kernels, trees, neural networks, etc.), namely :

- the train and test sets.
- overfitting (when the dimension d is not really small).

Example problem

We want to predict the power that needs to be produced by a power plant in a city, as a function of the temperature only.

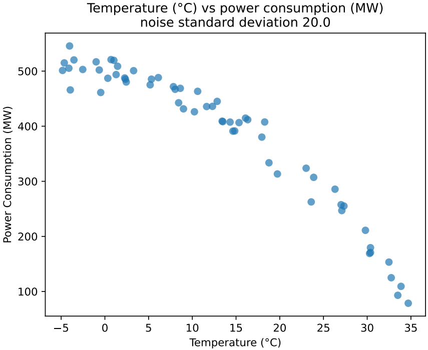


Figure – Dataset

The only information we have is a collection of n **samples**, called the **dataset**. Each sample consists in two values :

- the temperature in °C (the **feature**).
- the power consumption in W (the **label**).

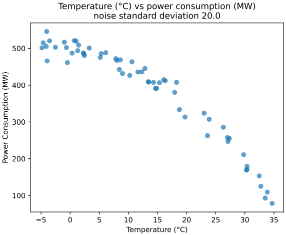


Figure – Dataset

Exercice 1 : What is d in this example ?

Supervised learning

Based on this **finite** dataset, our objective is to produce a function, noted $\tilde{f}$, called the **model**, the **predictor**, or the **estimator**, that will map a temperature to a power consumption.

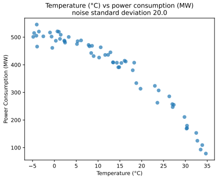


Figure – Dataset

The problem of supervised learning is that we want this function to perform well on **new, previously unseen samples**. It is not useful in itsself to perform well on the dataset !

Supervised learning

Based on this **finite** dataset, our objective is to produce a function, noted $\tilde{f}$, that will map a temperature to a power consumption.

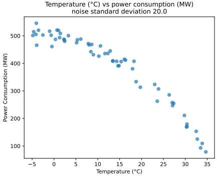


Figure – Dataset

Exercice 2 : How could we **estimate** the performance of $\tilde{f}$ on unseen data, using the dataset ?

Train set and set set

We split the dataset in two parts :

- the **train set** will be used to learn (train $\tilde{f}$)
- the **test set** will be used to estimate the performance of $\tilde{f}$ on unseen data.

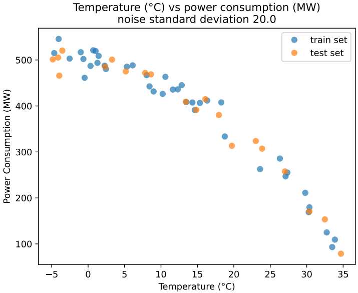


Figure – Splitted Dataset

Loss functions

To evaluate the performance of $\tilde{f}$, we use a **loss function**. The loss function is be a measure of the **discrepancy** between a prediction $\tilde{f}(x)$ and a label y (that are both real numbers). In regression, the most common loss function is the squared loss :

$$(\tilde{f}(x) - y)^2 \tag{1}$$

Averaging over the whole dataset, this leads to :

- The train error

$$\frac{1}{n_{\text{train}}} \sum_{(x_i, y_i) \in (X_{train} \times y_{train})} (\tilde{f}(x_i) - y_i)^2 \tag{2}$$

- The test error

$$\frac{1}{n_{\text{test}}} \sum_{(x_i, y_i) \in (X_{test} \times y_{test})} (\tilde{f}(x_i) - y_i)^2 \tag{3}$$

Empirical risk minimization

The most common supervised learning process is then naturally to :

- Find $\tilde{f}$ the as a small training error.
- Compute the test error of $\tilde{f}$ in order to estimate its performance on unseen data.

Remark : many of these two aspects will be discussed more deeply later in the couse (e.g. notions of validation set, or optimization error)

Train sets and test sets, loss functions

Linear regression in one dimension

Polynomial regression and overfitting

General linear regression : Ordinary least squares, ridge regression and hyperparameters

Linear regression

The remaining step is then to find $\tilde{f}$. The first step, called **model selection**, is to decide the type of function / model use. Many types of models exist, each one having potential drawbacks, for instance :

- linear functions (which we will use in this first example).
- polynomial functions
- kernels
- neural networks
- support vector machines

`https://scikit-learn.org/stable/machine_learning_map.html`

Exercice 3 : Why are the samples not on a line (and not on a curved line either) ?

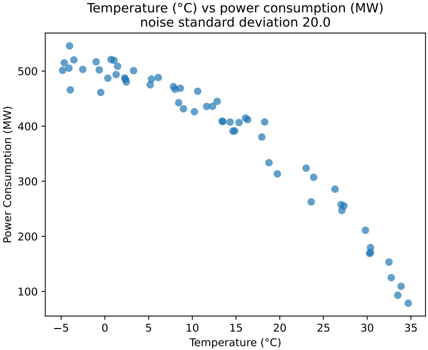


Figure – Dataset

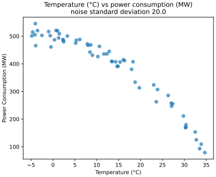


Figure – Dataset

The power consumption does not depend **only** on the temperature, but also on many other variables, that we do not have access to here :

- time in the day
- humidity, wind
- period of the year (holidays or not)
- other variables

Linear regression

Formalization :

- input space (temperature in °C) : $\mathcal{X} = \mathbb{R}$
- output space (power consumption in W) : $\mathcal{Y} = \mathbb{R}$
- dataset : $D = \{(x_1, y_1), \dots, (x_n, y_n), i \in [1, n]\}$.

linear regression in dimension 1 (as $x \in \mathbb{R}$), means that our estimator is of the form :

$$\tilde{f}(x) = \theta x + b \tag{4}$$

with $\theta \in \mathbb{R}$, $b \in \mathbb{R}$.

Empirical risk minimization

With the squared loss, and this function $\tilde{f}$ the **train error** is :

$$R_{n_{train}}(\theta, b) = \frac{1}{n_{train}} \sum_{(x_i, y_i) \in (X_{train} \times y_{train})} (\theta x_i + b - y_i)^2 \qquad (5)$$

The **optimization problem** is to find θ and b such that $R_{n_{train}}(\theta, b)$ has the **smallest possible value**.

Numpy

Numpy demo.

Exercice 4 :

cd code/day_1/1_linear_regression /1D_linear_regression/

- Generate some data using **main_create_data.py**. You can choose the amplitude of the noise by setting STD_NOISE in **constants.py** (we will discuss the importance of this constant shortly). The images are generated in the **images/datasets/** folder.
- fix **utils.py** in order to correctly compute the train error and test error.

Launch **main_random_params.py** in order to test several values for θ and b by evaluating their empirical risk.

Remark : In VSCode, you need to execute the python script frop the correct location (namely from **1D_linear_regression**) in order for the paths to properly work.

Analytic solutions

For some problems, like this one, it is possible to explicitely compute the optimal solution.
For some advanced reasons (convexity and differentiability of $R_n(\theta)$), the points optimizing the empirical risk are obtained by finding (θ^*, b^*) such that the gradient cancels (more on that later in the course).

$$\nabla_{(\theta,b)} R_n(\theta^*, b^*) = 0 \tag{6}$$

Derivatives

We drop the $\frac{1}{n}$ as it does not change the final result :

$$\begin{aligned}\frac{\partial R_n}{\partial \theta}(\theta, b) &= \sum_{i=1}^{n} 2(\theta x_i + b - y_i)x_i \\ &= 2[\theta \sum_{i=1}^{n} x_i^2 + b \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} x_i y_i]\end{aligned} \tag{7}$$

$$\begin{aligned}\frac{\partial R_n}{\partial b}(\theta, b) &= \sum_{i=1}^{n} 2(\theta x_i + b - y_i) \\ &= 2[\theta \sum_{i=1}^{n} x_i + nb - \sum_{i=1}^{n} y_i]\end{aligned} \tag{8}$$

Hence we have a system of 2 equations with 2 unknowns (dropping the θ^* notation)

$$\theta \sum_{i=1}^{n} x_i^2 + b \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} x_i y_i = 0 \tag{9}$$

$$\theta \sum_{i=1}^{n} x_i + nb - \sum_{i=1}^{n} y_i = 0 \tag{10}$$

Which means

$$b = \frac{1}{n}(\sum_{i=1}^{n} y_i - \theta \sum_{i=1}^{n} x_i) \tag{11}$$

$$\theta \sum_{i=1}^{n} x_i^2 + \frac{1}{n}(\sum_{i=1}^{n} y_i - \theta \sum_{i=1}^{n} x_i) \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} x_i y_i = 0 \tag{12}$$

Finally :

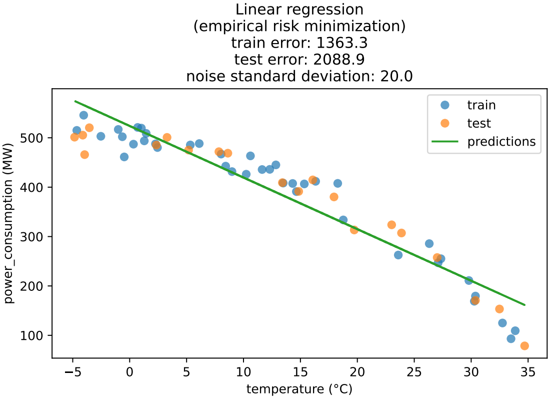
$$\theta(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2) + \frac{1}{n}\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} x_i y_i = 0 \qquad (13)$$

or

$$\theta^* = \frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n}\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}[\sum_{i=1}^{n} x_i]^2} \qquad (14)$$

Exercice 5 :

Fix **main_optimal_params.py** in order to plot the linear regression found with the analytic solution on the same plot as the raw dataset.

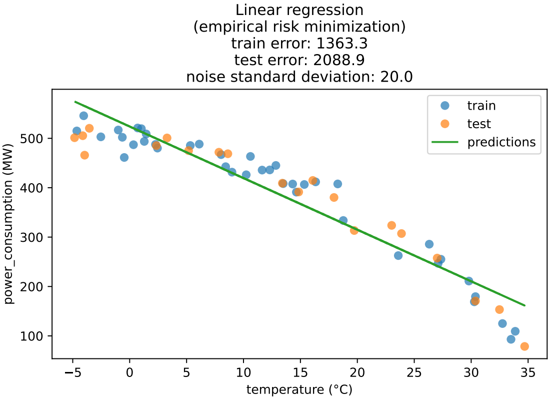


Scikit

With **main_scikit.py**, we can perform the same computation using scikit, and verify that the obtained estimator is identical.

Different model ?

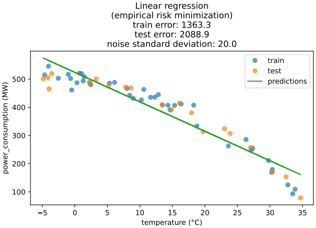
Exercice 6 : Could we use a better **model** for this dataset ?



Instead of a linear predictor, we could use a polynom. If t is the temperature, the polynomial predictor writes :

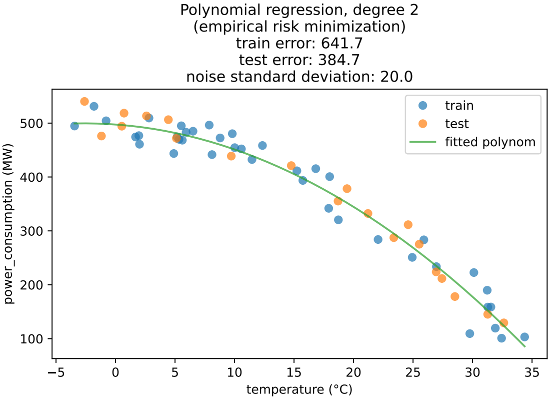
$$\tilde{f}(t) = a_0 + a_1 t + a_2 t^2 + \cdots + a_d t^d \tag{15}$$

with the **degree** d being a constant to determine.

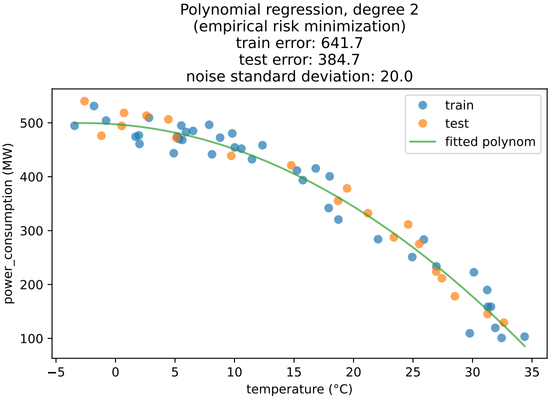


Fitting polynomials

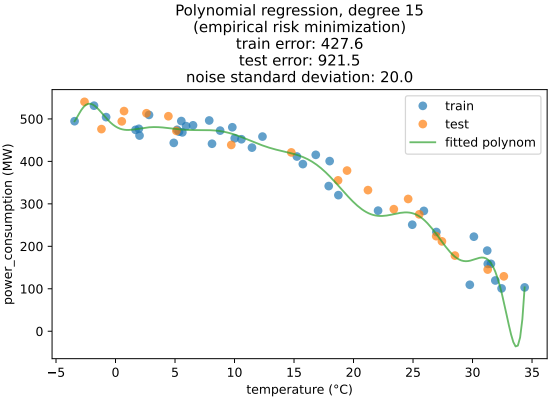
Using **main_polynomial_regression.py**, in which the optimization problem is directly handeled by numpy, we will make several observations.



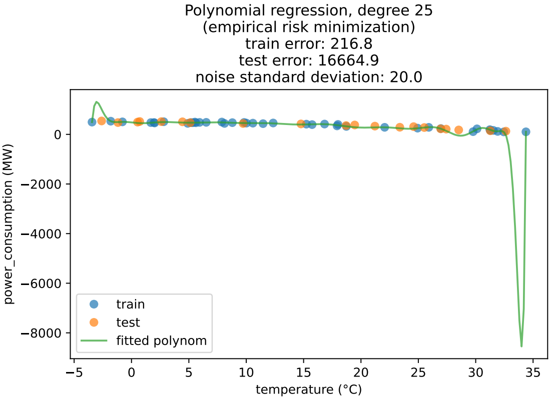
The test error is smaller with this polynomial fit of degree 2 than for the linear regression.



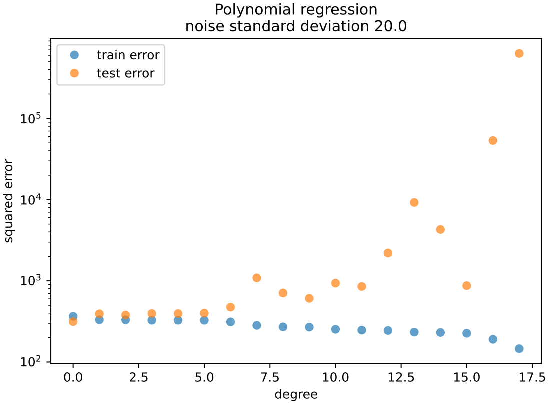
If the degree of the polynom is too large, **overfitting** happens.



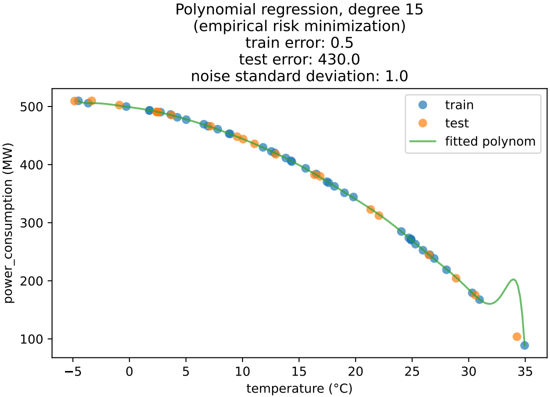
If the degree of the polynom is too large, **overfitting** happens.



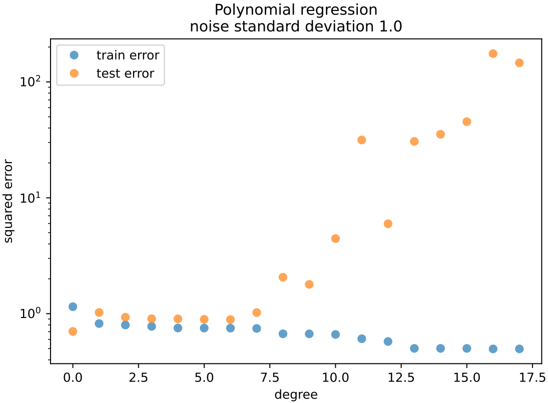
If the degree of the polynom is too large, **overfitting** happens.



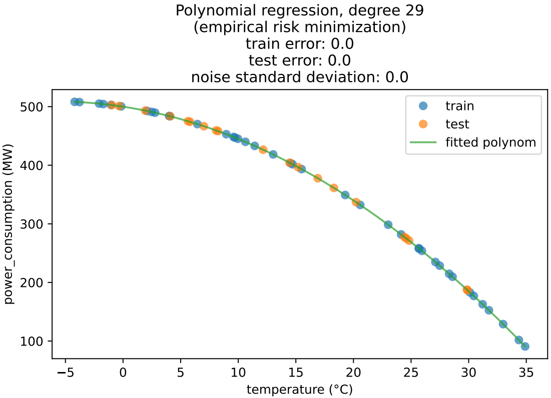
Reducing the noise standard deviation reduces overfitting



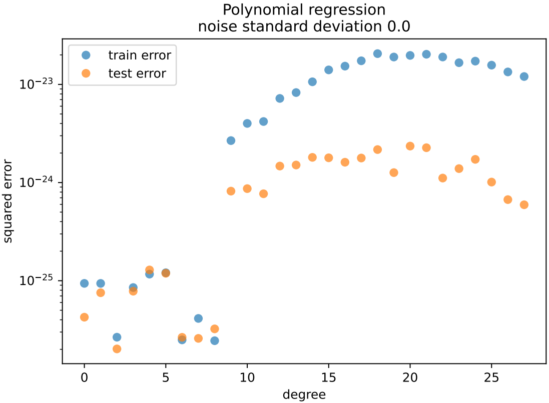
Reducing the noise standard deviation reduces overfitting



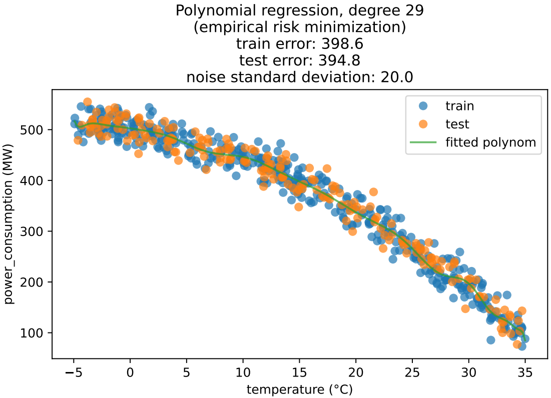
If there is no noise, there is no overfitting !



If there is no noise, there is no overfitting !



Having more samples also reduces overfitting.



Conclusion

The amount of overfitting depend on the **balance** between :

- the number of parameters / features / dimension d of the problem
- the number of available samples

Neural networks show some specific behavior on that topic, that we will study later.
Important remark : most of the time, it is not possible to have such a direct **visualisation** of the data (as soon as $d > 3$) !

Train sets and test sets, loss functions

Linear regression in one dimension

Polynomial regression and overfitting

General linear regression : Ordinary least squares, ridge regression and hyperparameters

Generalization

Linear regression also works in higher dimenions, when the inputs are multidimensional. For instance in dimension 3, $x = (x_1, x_2, x_3)$ and :

$$h(x) = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + b \tag{16}$$

The parameter is now $(\theta, b) = (\theta_1, \theta_2, \theta_3, b)$.
Example : x contains the age, the profession, and the gender.

Empirical risk

The empirical risk now writes (with adaptation to the relevant train / test dataset) :

$$R_n(\theta, b) = \frac{1}{n}\sum_{i=1}^{n}(\langle\theta, x_i\rangle + b - y_i)^2 \tag{17}$$

Almost similarly to the 1D case ($\theta \in \mathbb{R}$, $x_i \in \mathbb{R}$), where it was :

$$R_n(\theta, b) = \frac{1}{n}\sum_{i=1}^{n}(\theta x_i + b - y_i)^2 \tag{18}$$

Matrix notations

If we store the input data in a matrix X (called the **design matrix**) with n lines and d columns, and the labels in a vector y with n lines, the empirical risk writes :

$$X = \begin{pmatrix} x_1^T \\ \dots \\ x_i^T \\ \dots \\ x_n^T \end{pmatrix} = \begin{pmatrix} x_{11}, \dots, x_{1j}, \dots x_{1d} \\ \dots \\ x_{i1}, \dots, x_{ij}, \dots x_{id} \\ \dots \\ x_{n1}, \dots, x_{nj}, \dots x_{nd} \end{pmatrix} \tag{19}$$

$$R_n(\theta, b) = \frac{1}{n} ||X\theta - y + b||^2 \tag{20}$$

(more on the notion of norm later in the course)

OLS estimator

It is possible to show, with some maths notions, that the θ that minimizes the empirical risk is :

$$\hat{\theta} = (X^T X)^{-1} X^T y \tag{21}$$

T is the transposition.
$\hat{\theta}$ is called the **OLS estimator** (Ordinary least squares).

Scikit in 1D

We can use scikit-learn in order to obtain the OLS estimator directly.
`https://scikit-learn.org`
main_scikit.py computes the OLS for the previous power consumption example.
`https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html`

Overfitting

When d is of the same order of magnitude or larger than the number of samples n, it is possible to have a low **train error** and a high **test error**. As before, this is overfitting.
cd ../dD_linear_regression/

- Generate some data using **generate_data.py**
- Example with **OLS_scikit.py**

Ridge regression

Ridge regression is a variation of OLS. For some advanced reasons, minimizing the Ridge risk might reduce overfitting. We can witness this with **Ridge_scikit.py**.

- OLS risk :

$$R_{n,OLS}(\theta, b) = \sum_{i=1}^{n} (\langle \theta, x_i \rangle + b - y_i)^2 \tag{22}$$

- Ridge risk :

$$R_{n,Ridge}(\theta, b) = \sum_{i=1}^{n} (\langle \theta, x_i \rangle + b - y_i)^2 + \lambda ||\theta||^2 \tag{23}$$

with $\lambda > 0$ a real number (regularization parameter).

Ridge regression

`https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ridge.html`

Importantly, we can see that Ridge() has some parameters, called **hyperparameters**. Almost all machine learning algorithms have hyperparameters.

Choice of the hyperparameters

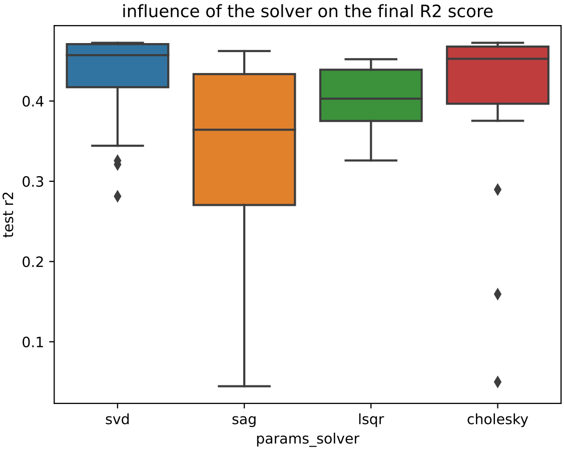
- The choice of the hyperparameters is very important. Most of the time, it is guided by experimentation and / or theoretical results.
- Some methods and libraries are helpful to look for good hyperparameters, such as **optuna** `https://optuna.org/`
- other classical methods : gridsearch, random search.

Using optuna to tune Ridge regression

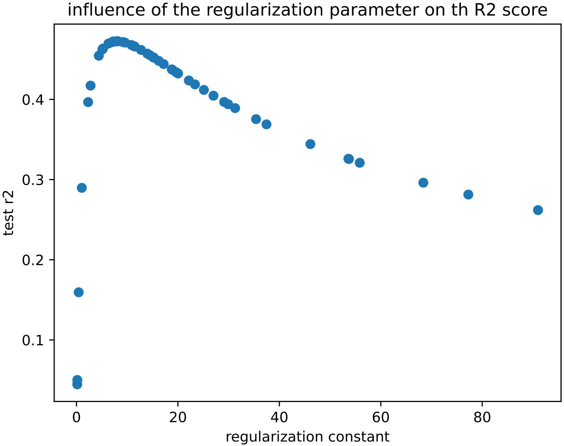
Exercice 7 :

Use **optuna_ridge_scikit.py** in order to choose some good hyperparameters (alpha, solver) for ridge. You will need to study the optuna API and to edit the **objective()** function.

Analysis of the hyperparameters

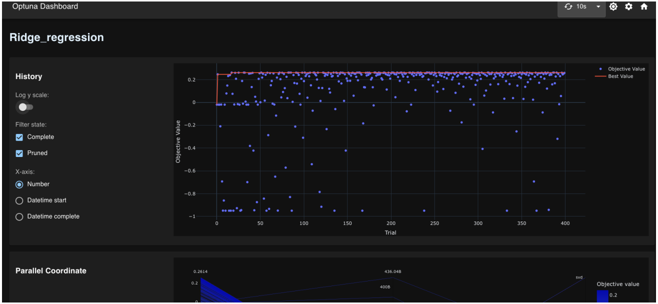


Analysis of the hyperparameters



Optuna dashboard

`https://github.com/optuna/optuna-dashboard`



Optuna Dashboard

Ridge_regression

History

Log y scale:

Filter state:

☑ Complete

☑ Pruned

X-axis:

● Number

○ Datetime start

○ Datetime complete

Parallel Coordinate

Multi-objective optimization

Optuna can be used to optimize several objectives, e.g. : optimize the score and minimize the computation time (both depend on the hyperparameters).

Notion of Pareto front.