UNIVERSITETET
I OSLO

Project 1: Linear regression methods, regularization, resampling

Applied Data Analysis and Machine Learning

UiO (FYS-STK4155)

Pietro PERRONE

pietrope@uio.no

October 6, 2025

GitHub link to the project repository:
https://github.com/p-perrone/UiO_MachineLearning/tree/main/ml_project1
Link to DeepSeek chat:
https://chat.deepseek.com/share/h2ifare1m1c31ud8vf

Contents

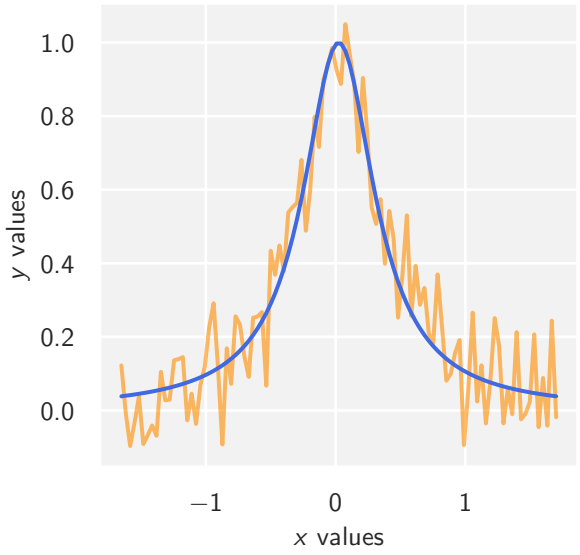

Figure 1: Runge function | In orange with an added noise of distribution $\mathcal{N}(0,0.1)$. The x values have already been scaled

1 Introduction

The aim of this project is to implement functions that could help in the understanding of the behaviors of different linear regression methods and resampling techniques, with a view to potential machine learning applications.

This will be done by creating a own Python 3 library, following the typical workflows that characterize existing Python modules such as `scikit-learn`.

In the frame of this work, all the different algorithms will be tested for a simple 1D function, the Runge function (Runge, 1901):

$$\text{Runge}(x) = y = \frac{1}{1+25x^2} + \epsilon \tag{1}$$

where $x \in [-1,1]$ and ϵ represent a stochastic noise, distributed according to a normal law $\mathcal{N}(0,1)$ in our case. The curve of this function is shown in Figure 1.

First of all, a code for the Runge function will be provided, and the methods such as the Ordinary Least Squares (OLS) and Ridge regression will be coded and adapted to this function. All the necessary preprocessing steps will be included, particularly the scaling an train-test splitting of the data. Secondly, we will move from regressions whose fit can be computed analytically to iterative numerical methods, namely the Gradient Descent. In this context, different types of this technique will be analyzed, dealing with its simplest form initially and then adding momentum and stochastic sampling. The Lasso regression will be implemented in this part as well.

Lastly, we will perform an analysis of the bias-variance tradeoff as a function of the complexity of the models, comparing two different resampling techniques, the bootstrap and the cross-validation.

2 Analytical regressions: Ordinary Least Squares and Ridge

2.1 Theory

Most of the following informations has been retrieved by textbooks such as Hastie et al., 2009, and from the Jupyter Book referred to this course available on GitHub.

2.1.a Ordinary Least Squares

Regression modeling aims to sample how a random variable y is distributed and varies as a function of another variable x. In this model, therefore, the input is going to be the vector $\boldsymbol{x}^\intercal = [x_0, x_1, x_2, \ldots, x_{n-1}]$ and the output, *i.e.* the value that we want to model, $\boldsymbol{y}^\intercal = [y_0, y_1, y_2, \ldots, y_{n-1}]$.

Particularly, an Ordinary Least Squares linear regression approximates the unknown output values with another continuous function $\tilde{\boldsymbol{y}}(\boldsymbol{x})$ which depends linearly on parameters $\boldsymbol{\theta}^\intercal = [\theta_0, \theta_1, \theta_2, \ldots, \theta_{p-1}]$:

$$\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\theta} \tag{2}$$

where $\boldsymbol{X} \in \mathbb{R}^{n\times p}$ is the so called *design matrix*, whose p columns represent the features, i.e. the variables that can be used to predict $\boldsymbol{y}$, and the rows the n input observations used for the modeling. In the case of a polynomial regression, the features are

polynomials obtained by elevating $\boldsymbol{x}$ to powers that range from 0 (or 1) to p:

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_0^1 & x_0^2 & \dots & x_0^{p-1} \\ 1 & x_1^1 & x_1^2 & \dots & x_1^{p-1} \\ 1 & x_2^1 & x_2^2 & \dots & x_2^{p-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1}^1 & x_{n-1}^2 & \dots & x_{n-1}^{p-1} \end{bmatrix} \tag{3}$$

The aim is to *fit* on the unknown function $\boldsymbol{y}$ these parameters, *i.e.* finding the ones that can minimize the spread between the true expected values $\boldsymbol{y}$ and the predicted ones $\tilde{\boldsymbol{y}}$, through a process called regularization.

This spread can be defined in terms of a *cost function* $C(\boldsymbol{\theta})$:

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \frac{1}{n}\left\{(\boldsymbol{y} - \tilde{\boldsymbol{y}})^\intercal(\boldsymbol{y} - \tilde{\boldsymbol{y}})\right\} \tag{4}$$

that can be rewritten in terms of Equation (2):

$$C(\boldsymbol{\theta}) = \frac{1}{n}\left\{(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^\intercal(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})\right\}. \tag{5}$$

Optimizing $\boldsymbol{\theta}$ means minimize this spread function:

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p}\frac{1}{n}\left\{(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^\intercal(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})\right\}. \tag{6}$$

which is done by taking its derivative with respect to $\boldsymbol{\theta}$ and setting it to 0:

$$\frac{\partial C(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0 = \boldsymbol{X}^\intercal(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) \tag{7}$$

which yields

$$\boldsymbol{X}^\intercal\boldsymbol{y} = \boldsymbol{X}^\intercal\boldsymbol{X}\boldsymbol{\theta}, \tag{8}$$

If the matrix $\boldsymbol{X}^\intercal\boldsymbol{X}$ is invertible the solution for the optimal parameters is

$$\hat{\boldsymbol{\theta}} = (\boldsymbol{X}^\intercal\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{9}$$

In most of the cases, $\boldsymbol{X}^\intercal\boldsymbol{X}$ cannot be inverted directly, because it tends to be low-dimensional. In this situations algorithms such as the Singular Value Decomposition can be applied.

2.1.b Ridge regression

When the number of observations and the complexity (maximum polynomial degree) of the fit are close, there might be some *overfitting* in the regression. This means that, even if the regularization shrinks as much as possible the cost function because of the high polynomial degree chosen, applying the same fit to different datasets can lead to a high variance between the predicted results and to a worse generalization of the problem. One way to fix this issue is to add a regularization term to the matrix we have to invert:

$$\boldsymbol{X}^\intercal\boldsymbol{X} \to \boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I}, \tag{10}$$

λ is a *hyperparameter*, or penalty term, ranging typically from 10^{-5} to 10^2. The higher λ, the higher the shrinkage of the fit to a a more general, and of course biased, one. This term characterize the Ridge regression

The minimization for the Ridge problem is therefore

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p}\frac{1}{n}||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}||_2^2 + \lambda||\boldsymbol{\theta}||_2^2 \tag{11}$$

by applying the definition of the *norm-2* vector:

$$||\boldsymbol{x}||_2 = \sqrt{\sum_i x_i^2}. \tag{12}$$

By minimizing this problem as previously done, the optimal parameters for the Ridge regression can be obtained:

$$\hat{\boldsymbol{\theta}}_{\text{Ridge}} = \left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\intercal\boldsymbol{y} \tag{13}$$

The differences between OLS and Ridge regression will be more deeply analyzed in the following sections.

2.2 Preprocessing

In the frame of machine learning, the regularization process requires some preprocessing steps before actually dealing with the problem itself.

As previously said, we will deal with 1-dimensional $\boldsymbol{x}$ dataset and the function we want to model $\boldsymbol{y} = \text{Runge}(\boldsymbol{x})\,(+\boldsymbol{\epsilon})$

First of all, with the aim of "training" a model, it is necessary to define which data will be used for training and which ones for testing the behavior of the model. For this purpose, `scikit-learn` provides a function from the module `model_selection` : `train_test_split` . This function splits the input dataset and the one given by the true values $\boldsymbol{y}$ in a train set an in a test set. Usually the dimension of the test cluster is set at 20-30% of the former number of observations:

```
x_train, x_test, y_train, y_test
= train_test_split(x, y, test_size=0.2)
```

Secondly, a scaling of the data has to be done. In most of the real cases the application of high polynomial degree to an input dataset can yield values of very different order of magnitudes, which can lead to numerical stability issues while implementing the regression. Also, we often deal with real dimensional variables, and in the regularization having a-dimensional quantities is much more convenient.

Different types of scaling can solve this problems. The *standard scaling* can be used: every element of the input dataset is subtracted by the mean value of $\boldsymbol{x}$ and divided by its standard deviation:

$$x_i \to \frac{x_i - \overline{x}}{\sigma(\boldsymbol{x})} \tag{14}$$

One other scaling technique is the *min-max* scaling.

`scikit-learn` provides a class, `StandardScaler(with_mean, with_std)` , that performs a standard scaling, and that will be used in this workflow. Actually, in our case, the input dataset consists of a range of a-dimensional values quite limited ($[-1, 1]$). In this particular case, we don't expect therefore to observe huge differences in the results obtained scaling and others obtained without performing this operation. Nevertheless, we still want to implement this step ih the process in order make it more general and potentially more robust if applied to more complex systems.

2.3 OLS implementation

All the steps presented in Section 2.1 have been coded on an own class called `LinearRegression_own` . This class is structured following the typical OLS workflow (and Ridge, as well), similar to the one of `scikit_learn` :

1. creation of the design (features) matrices $\boldsymbol{X}_{\text{train}}$ and $\boldsymbol{X}_{\text{test}}$, starting from the input `x_train` and `x_test` input arrays, and declaring the desired polynomial degree `p` $\longleftrightarrow$ `polynomial_features(x, p)` ;
2. fit of the model on the train set $\longleftrightarrow$ `fit(X, y, method, lbda)` , either for obtaining $\boldsymbol{\theta}_{\text{OLS}}$ or $\boldsymbol{\theta}_{\text{Ridge}}$;
3. computation of the predicted value $\tilde{\boldsymbol{y}}$ $\longleftrightarrow$ `predict()`

An example of the implementation for the OLS is shown in Figure 2.

2.3.a Behavior of the own-coded OLS

We now want to analyze the performance of this type of regression as a function of the complexity (polynomial degree). For the moment, we stick to the original dataset with 100 samples.

For this purpose, we will look at two statistical quantities. The first is the *Mean Squared Error* (MSE), that, in this case, correspond to the cost function $C(\boldsymbol{\theta})$:

$$MSE(\boldsymbol{y}, \tilde{\boldsymbol{y}}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 \tag{15}$$

This estimator represents the spread between the true values and the predicted ones.

The second quantity is the *determination coefficient* R^2:

$$R^2(\boldsymbol{y}, \tilde{\boldsymbol{y}}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2} \tag{16}$$

This score provides a measure of the effectiveness of the model on potential future samples taken from the former population. An $0.8 < R^2 < 1$ is generally representative of a fit that can generalize very well the true value. This estimator can also be close to 0:

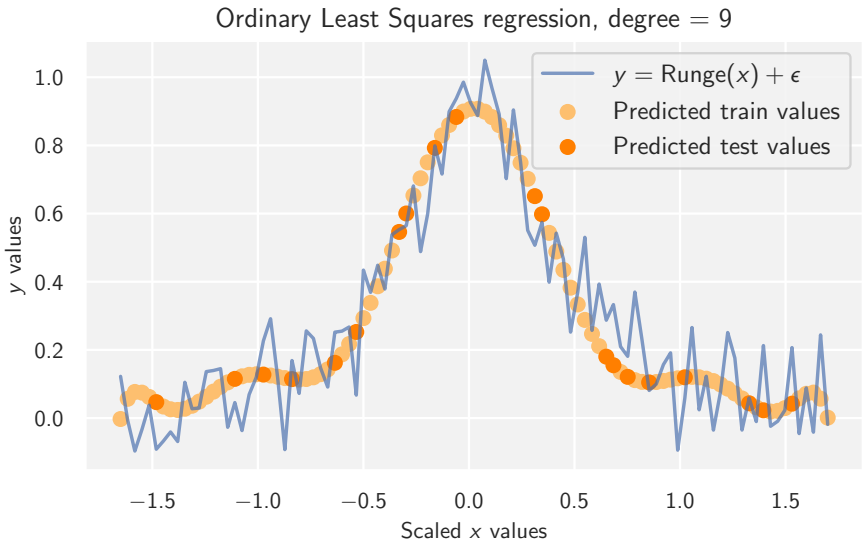

Figure 2: OLS regression | For a polynomial fit of degree 10, from a former dataset of $n = 100$ observations. The result of `train_test_split` can also be observed here.

in this case, it means that the model can't provide a better prediction than a constant, which indicates a poor quality of the fit. If the model is even worse than a constant model, then we can expect a $R^2 < 0$.

Both these estimators have been implemented with the two respective functions of `scikit_learn` : `mean_squared_error` and `r2_score` .

In Figure 3 we can observe the trend of this estimators as a function of the model's complexity. With 100 samples, good values are typical of fit of polynomial degree $7 < p < 25$. For smaller p, the model is too generic. For higher p, overfitting occurs, so the model become extremely sample-specific and cannot be generalized. We can equally observe a discrepancy between the two sets, that is still reasonably small (about 10%), so the model is performing well on the test set as well.

Subsequently, we want to analyze the trend of these estimators as a function of both the dimensionality n and p. The result is shown in Figure 4. Compared to the previous experiment, we can observe that increasing the dimension of n has solved the overfitting problem – at least, the MSE and the R^2 show good values for higher polynomial degrees. For this second experiment, the own function `MSE_R2_pn()` has been implemented, with a condition that set the value of the estimators to NaN in the case where $p \geq n$, in order to increase the readability of the heatmaps. We can still observe an isolated overfitting phenomenon for $n = 10$ and $p = 6$.

2.4 Ridge regression implementation

We now want to find the optimal parameters solving the Ridge regularization (Equation (13)). The penalty coefficient λ determines the shrinkage of the fit towards a constant model: its effects can be observed in Figure 5.

We can now explore the dependency of the two estimators as a function of p and λ. Since we already analyzed the influence of the input dataset dimensionality n, we will keep the former 100 sample $\boldsymbol{x}$. The result can be seen in Figure 6. We can clearly see that there is a dependence on p of the fit's goodness only for for values of λ up to $\sim 10^2$. For stronger penalties, the regression will always perform as a constant model (very high MSE and very poor R^2). We can equally observe the effect of overfitting for smaller λ and higher p, which is logical considering the relatively small amount of input samples.

3 Gradient descent methods

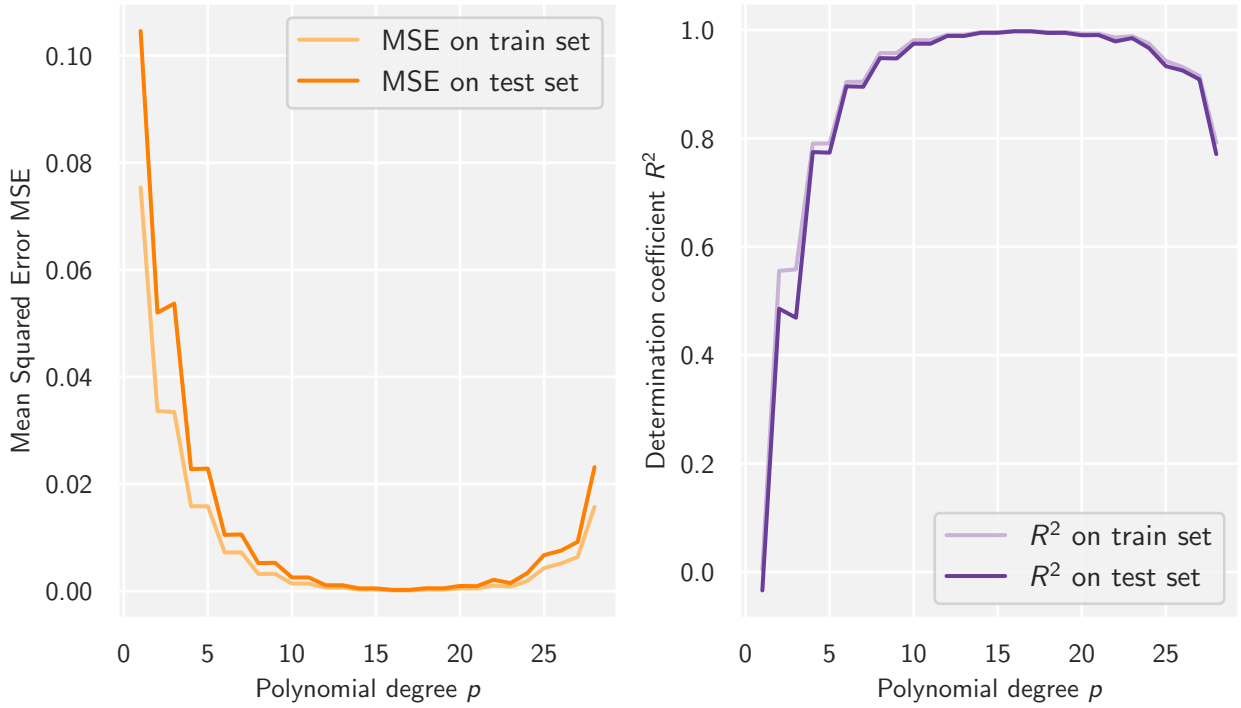

Figure 3: MSE and R^2 for OLS regression as a function of p | Statistical estimators of model effectiveness with fixed $n = 100$.

3.1 Theory

3.1.a Newton-Raphson method

The analytical solutions shown in Section 2.1 are not always available in machine learning. In fact, inverting the $\boldsymbol{X}^\intercal\boldsymbol{X}$ can sometimes lead to challenging computational issues. In these situations, an iterative, numerical approach should be preferred in order to find the minimum of the cost function.

The most common iterative method for such mathematical problems is the Newton-Raphson formula, that consists in substituting to a curve $y = f(x)$ the tangent of the curve, and by successive iteration, approximating the solution $s|f(s) = 0$.

It basically consists of a Taylor expansion, generally approximated to the first order, expressed for the function we want to model $f(x)$ and close the solution of the problem s:

$$f(x) + (s - x)f'(x) \approx 0 \tag{17}$$

that yields

$$s \approx x - \frac{f(x)}{f'(x)} \tag{18}$$

This problem can be translated into an iterative process, where we try to approximate s:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{19}$$

3.1.b Gradient descent fundamentals

For a multivariate function $F(\mathbf{x})$ with $\mathbf{x} = (x_0, x_1, \dots x_p)$, the fastest way to fine the global minimum is to follow the direction of the negative gradient $-\boldsymbol{\nabla} F(\mathbf{x})$.

Adopting an iterative approach as shown in Equation (19), we can write the following expression:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \boldsymbol{\nabla} F(\mathbf{x}_k) \tag{20}$$

Here $\gamma_k > 0$ and if this term is small enough, then it can be demonstrated that $F(\mathbf{x_{k+1}}) \leq F(\mathbf{x_k})$ and therefore at each iteration we are approaching the minimum of F. Particularly, this result can be achieved only if $F(\mathbf{x})$ is a convex function.

3.1.c Application to the cost function

In our case, the cost function $C(\boldsymbol{\theta})$ is the multivariate function that we want to minimize. In the

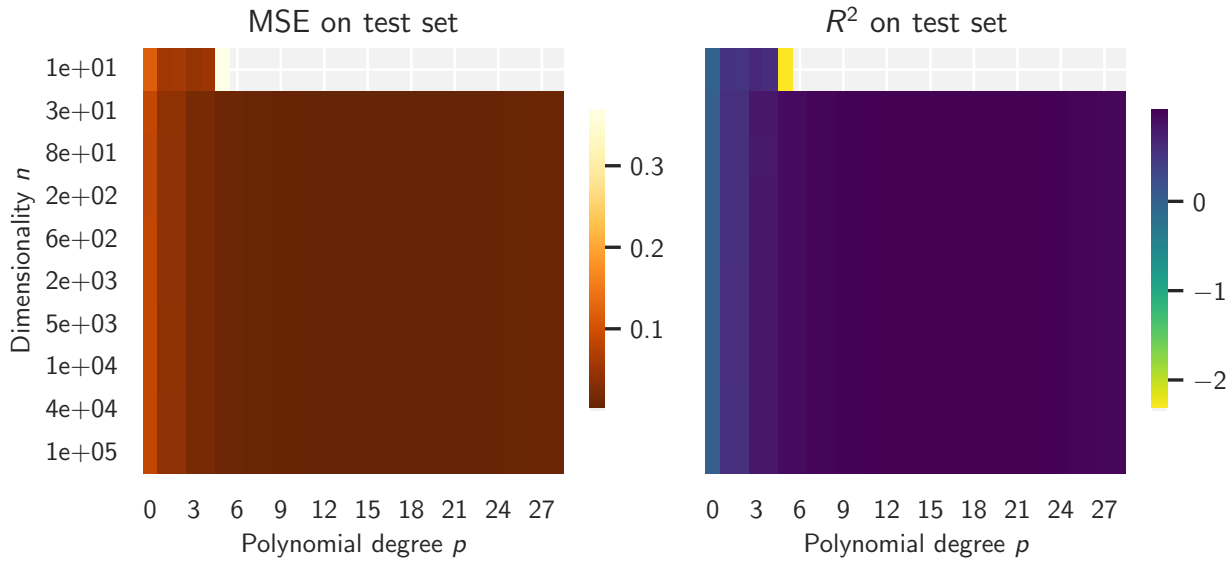

Figure 4: MSE and R^2 for OLS regression as a function of p and n | Statistical estimators of model effectiveness varying both parameters.

simple case where $\boldsymbol{\theta} = (\theta_0, \theta_1)$ (*i.e*, an intercept and a slope), the gradient of a simple OLS $C(\theta)$ can be expressed as following:

$$\begin{aligned}\boldsymbol{\nabla}_\theta C(\theta) &= \frac{2}{n}\begin{bmatrix}\sum_{i=0}^{n-1}(\theta_0+\theta_1 x_i - y_i)\\ \sum_{i=0}^{n-1}(x_i(\theta_0+\theta_1 x_i) - y_i x_i)\end{bmatrix}\\ &= \frac{2}{n}\boldsymbol{X}^\intercal(\boldsymbol{X}\boldsymbol{\theta} - \mathbf{y}) \quad (21)\end{aligned}$$

The Hessian matrix of $C(\boldsymbol{\theta})$ is given by

$$\boldsymbol{H} \equiv \begin{bmatrix}\frac{\partial^2 C(\boldsymbol{\theta})}{\partial \theta_0^2} & \frac{\partial C(\boldsymbol{\theta})}{\partial \theta_0}\theta_1\\ \frac{\partial C(\boldsymbol{\theta})}{\partial \theta_0}\theta_1 & \frac{\partial^2 C(\boldsymbol{\theta})}{\partial \theta_1^2}\end{bmatrix} = \frac{2}{n}\boldsymbol{X}^\intercal\boldsymbol{X} \quad (22)$$

Similarly, for Ridge's cost function, the gradient can be expressed as $\frac{2}{n}(\boldsymbol{X}^\intercal(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}) + \lambda\boldsymbol{\theta})$.

Since $\boldsymbol{X}^\intercal\boldsymbol{X}$ is always positive semi-definite, we can assess that the cost function is always convex.

We can finally express the formula that will be the starting point for implementing the simple Gradient Descent algorithm:

$$\theta_{k+1} = \theta_k - \eta\boldsymbol{\nabla}_\theta C(\theta_k), \;\; k = 0, 1, \cdots p \quad (23)$$

The initial θ_0 can be randomly chosen. In the context of regularization, η defines the *learning rate*, *i.e.* the rate at which we approximate the minimum at each iteration.

3.2 Simple Gradient descent

The heart of the most basic Gradient Descent (GD) algorithm is given by the following loop:

```
for k in range(iterations):
  gradient = grad(cost)(theta)
  theta -= eta * gradient

  if eta * gradient_norm <= converge:
    break
```

That is, the optimal parameter `theta` is updated at each `k` iteration until the norm of the gradient $\|\boldsymbol{\nabla}_k C(\theta_k)\|$ (`gradient_norm`) is smaller than a previously defined convergence criterion, usually set at 10^{-8}, namely when the algorithm approaches 0.

In order to test the behavior of this algorithm, the function `theta_gd` has been implemented. The parameters that influence the gradient descent performance are `eta` and `iterations`. The first one, the learning rate, is constant in this simple case, and usually set $10^{-4} < \eta < 10^{-1}$; the number of iterations can vary from 1000 to more than 100 000.

The gradient in Python can also be computed numerically using the function `grad` from the module `autograd`. Nevertheless, for this simple case that

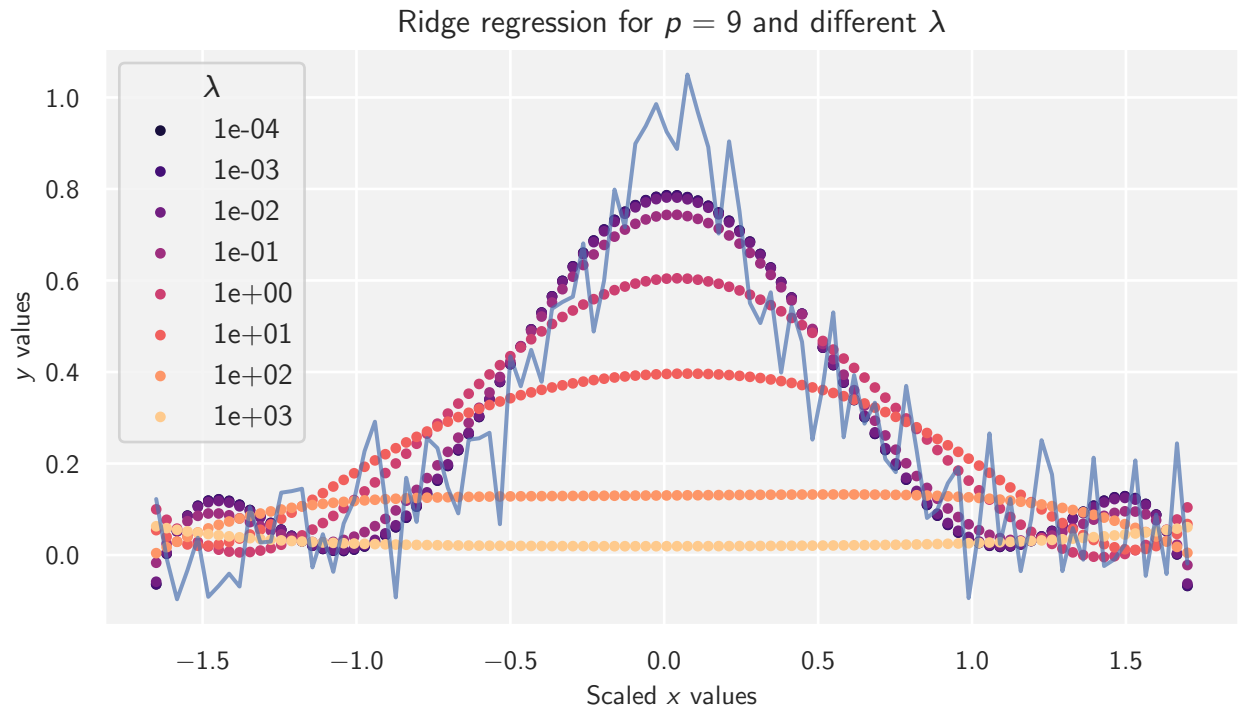

Figure 5: Ridge regression | For different values of λ (penalty coefficient), from a former dataset of $n = 100$ observations. When λ approaches 0, the fit approximates to a simple OLS regression. Runge function with noise is also shown in blue.

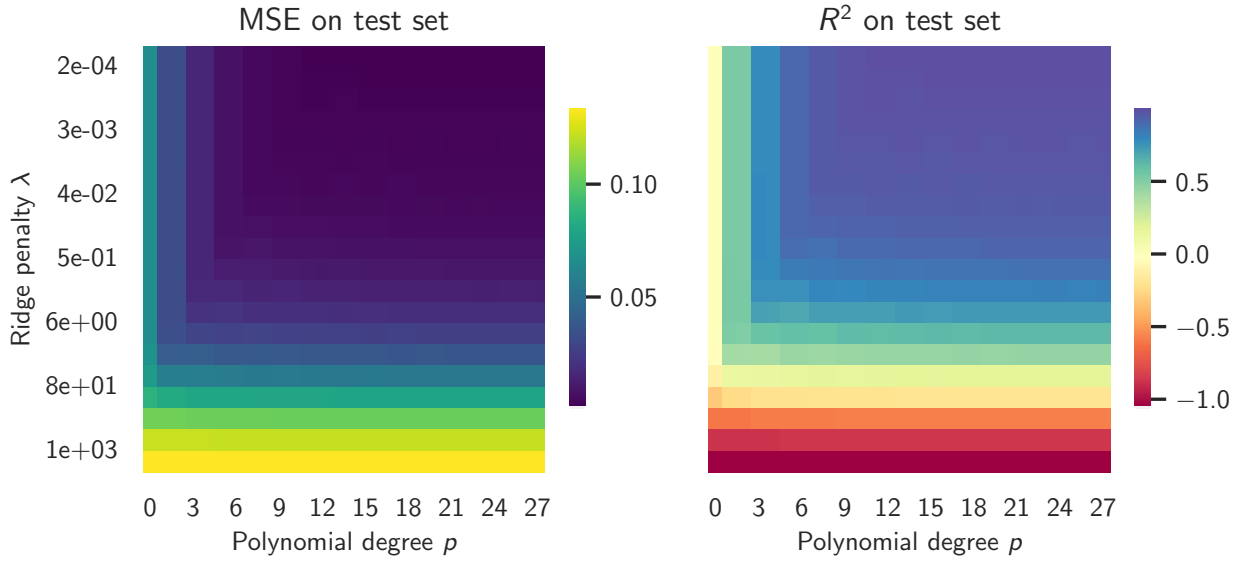

Figure 6: Ridge regression | For different values of λ (penalty coefficient), from a former dataset of $n = 100$ observations. When λ approaches 0, the fit approximates to a simple OLS regression. Runge function with noise is also shown in blue.

only includes methods such as OLS and Ridge regression, we implement a simple function that uses the analytical solution of the gradient of the cost function (Equation (21)), `grad_analytical` .

3.2.a Considerations on simple GD

In a first moment, by manually regulating the above parameters, the solution that can better approximate a simple OLS regression is obtained with a fairly high number of iterations (200 000) and with a medium learning rate of 10^{-3}. This first result can be seen in Figure 7.

For a more comprehensive understanding of the influence of the number of iterations, learning rate an polynomial degree together, another experiment has been run, in order to know how MSE and R^2 evolve as functions of these three parameters. Again, the former 100 sample dataset will be used.

The result is shown in Figure 8. We can observe that some problems occur with $\eta = 10^3$: this could be probably related to a potential overflow, caused by the excessive learning rate. The best values of R^2 are obtained by increasing the iterations. The limitations of a constant learning rate are indirectly shown in this figure: the maximum polynomial degree is set to 8, because higher complexity always caused overflows in the experiments, returning NaN values.

3.3 Updating the learning rate in Gradient Descent

3.3.a Momentum Gradient Descent

The simple gradient descent uses a constant learning rate throughout the process. This can turn in a slower algorithm and increase the probability that the program will be stuck in local minima of the cost function. In order to avoid this, several ways to update the learning rate can be implemented. In general, all these techniques keep memory of the previous step and, based on that, they increase or decrease the value of the learning rate.

The simplest case is the *momentum* gradient descent. The key idea of this method is to update the optimal parameter by an inertial term $\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \boldsymbol{\nabla}_k C(\boldsymbol{\theta}_k)$:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k \tag{24}$$

A typical pseudo-code for implementing this algorithm could be:

```
for k in range(iterations):
  v = momentum * v - eta * gradient
  update = v
  theta += update
```

For this analysis, we set `momentum` to 0.9. This parameter represents the amount of inertia applied to each iteration.

3.3.b More complex ways to update the learning rate: AdaGrad, RMSProp, ADAM

Others adaptive gradient methods can adjust the learning rate dynamically for a further increase in convergence and stability of the algorithm.

AdaGrad

This method scales the learning rate for each parameter by maintaining the cumulative squared sum of previous gradients. For a parameter θ_j at iteration k, the update rule is:

$$\theta_{j,k+1} = \theta_{j,k} - \frac{\eta}{\sqrt{r_{j,k}} + \epsilon} g_{j,k} \tag{25}$$

where $g_{j,k}$ is the gradient of the loss with respect to θ_j,

$$r_k = r_{k-1} + g_k \circ g_k \tag{26}$$

is the accumulated squared gradient, with $g_t \circ g_t$ the element-wise square of the gradient vector, η is the base learning rate, and ϵ is a small constant for avoiding division by 0. AdaGrad is effective for sparse data and features, but the accumulation of squared gradients can lead to a strong shrinking of the learning rate over time, slowing potentially too much the progress.

RMSProp

This algorithm (Goodfellow et al., 2016) modifies AdaGrad by substituting a decaying average of the

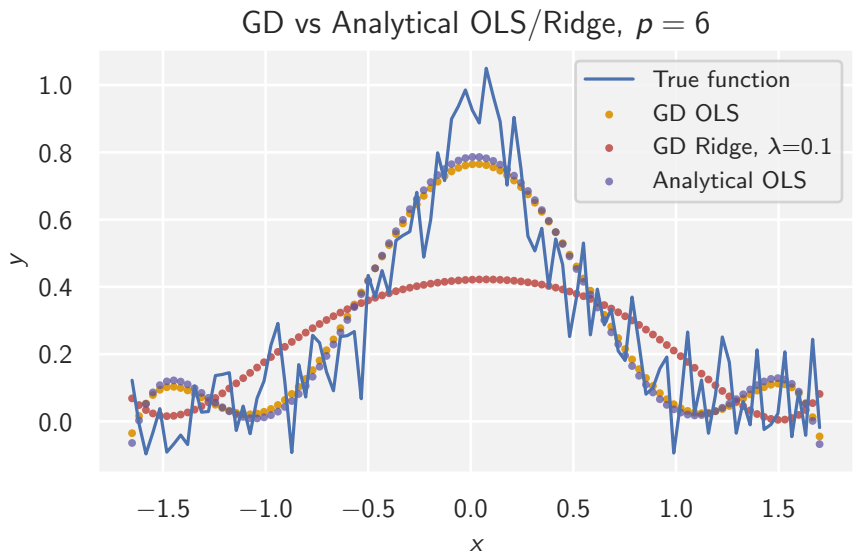

Figure 7: GD and OLS | Comparison between a classical OLS algorithm and Gradient Descent methods.

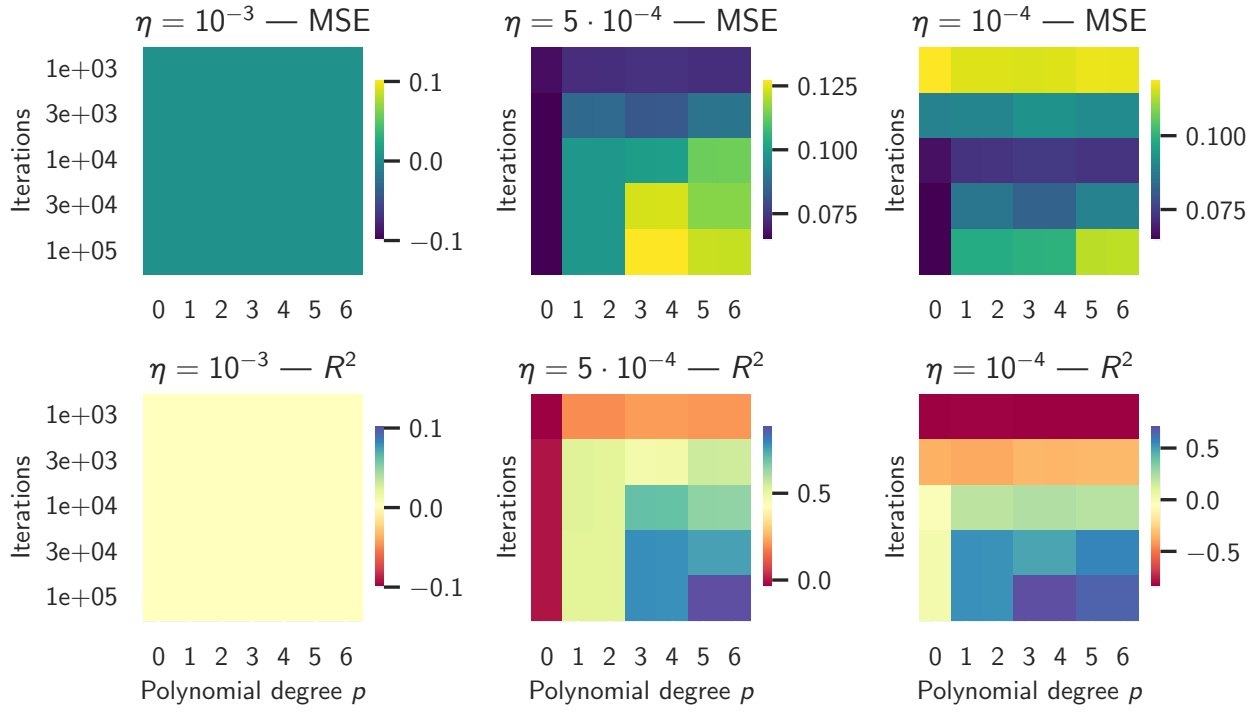

Figure 8: Gradient Descent | **For different η, iterations and p**

squared gradients instead of the cumulative squared sum:

$$r_j, k \to v_{j,k} = \rho v_{j,k-1} + (1-\rho)(\boldsymbol{\nabla}_k C(\theta_k))^2 \quad (27)$$

ADAM (Adaptive Moment Estimation)

ADAM (Kingma & Ba, 2017) combines RMSProp with momentum by maintaining two moving averages: the first momentum m_k (mean of gradients) and the second moment (*uncentered variance*) v_k:

$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\boldsymbol{\nabla}_k C(\theta_k) \quad (28)$$
$$v_k = \beta_2 v_{k-1} + (1-\beta_2)(\boldsymbol{\nabla}_k C(\theta_k))^2 \quad (29)$$

with bias-corrected versions:

$$\hat{m}_k = \frac{m_k}{1-\beta_1^k} \quad (30)$$
$$\hat{v}_k = \frac{v_k}{1-\beta_2^k} \quad (31)$$

with typical $\beta_1 = 0.9$, $\beta_2 = 0.999$. Initialize $m_0 = 0$, $v_0 = 0$.

The update is then:

$$\theta_{k+1} = \theta_k - \frac{\alpha}{\sqrt{\hat{v}_k} + \epsilon}\hat{m}_k \quad (32)$$

3.3.c Gradient Descent with learning rate update implementation

The function `theta_gd_mom` includes all the four techniques of above for updating the learning rate. For reasons of numerical instability, the function is yielding MSEs and R^2s that don't fall in an "acceptable" range. This problem could potentially be fixed by increasing iterations or decreasing the polynomial degree. This is a common issue of other functions implemented in the frame of this work, such as the LASSO regression function (see next Sections). Nevertheless, the effects of the different types of updates between AdaGRad, RMSProp and ADAM have been observed: the three methods showed better convergence progressively, with ADAM being much faster than the other two techniques.

3.4 LASSO regression

3.4.a Theory

LASSO (Least Absolute Shrinkage and Selection Operator) regression is similar to Ridge in the way it adds a regularization term to the former OLS-type cost function in order to define a penalty. In this case, nevertheless, this term is an L1-type regularization term, and not an L2-type.

The L1 (norm-1) term is generally defined as:

$$\|\boldsymbol{x}\|_1 = \sum_i |x_i| \quad (33)$$

The minimization of the cost function for LASSO becomes therefore:

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \frac{1}{n}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1 \quad (34)$$

This problem, because of the presence of an absolute value, is not differentiable for any $\theta_i = 0$. Hence, it cannot be solved via analytical expressions. For this reason, we can apply the gradient descent methods of above for solving the LASSO problem.

3.4.b Considerations on LASSO regression

In Figure 9 we can clearly observe that the L1-term of LASSO regression has a stronger impact the the Ridge's L2-term on the fit, by comparing the curves of same λ with the ones of Figure 5

3.5 Stochastic Gradient Descent

3.5.a Theory

One of the main problems of regular Gradient Descent is that, with high dimensionalities and with high complexity, running the algorithm can take a large amount of time. Stochastic Gradient Descent (SGD) partially fix this issue by performing the approximation, at each iteration, only on a subset of the former input dataset, called *minibatch*. The simplest case of SGD is the one where we compute the gradient on one sample only at each iteration.

Given that the gradient of the cost function can be expressed in terms of the sum of the gradients of every cost function c_i computed for the respective $\mathbf{x}_i$ (*i.e.*, row of the design matrix):

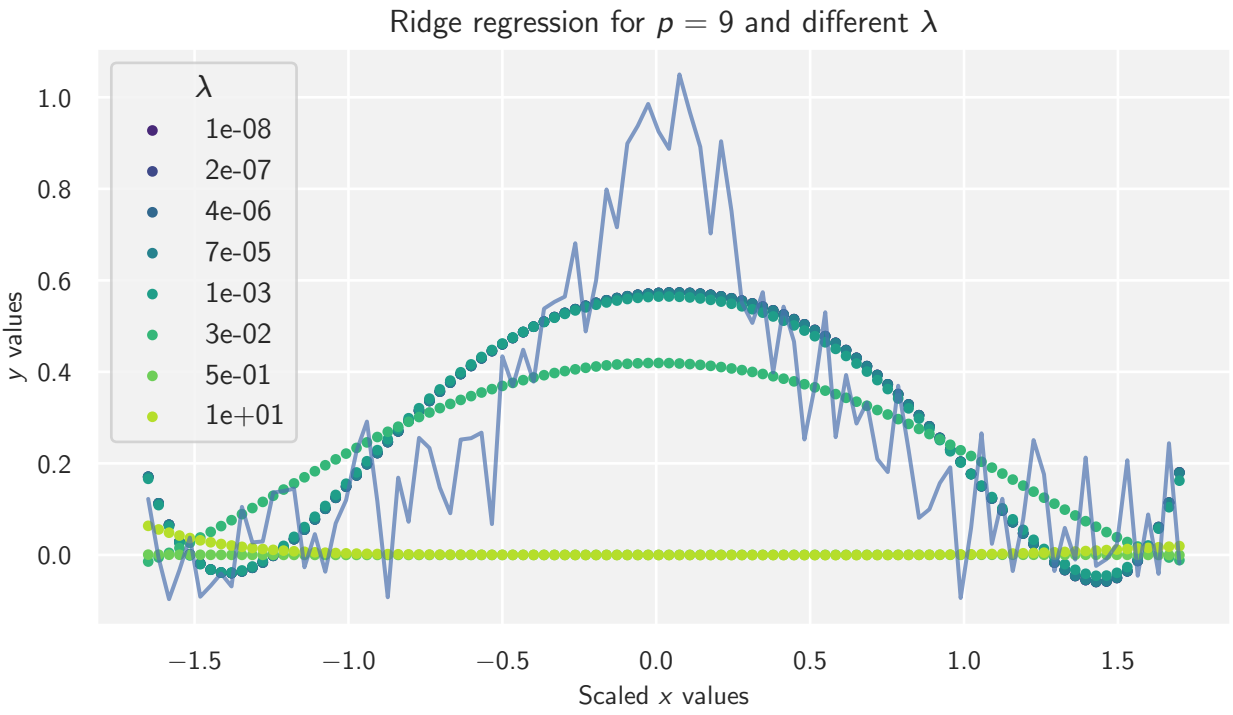

Figure 9: Lasso regression | With a Gradient Descent, 100000 iterations and $\eta = 10^{-4}$

$$\nabla_\theta C(\theta) = \sum_i^n \nabla_\theta c_i(\mathbf{x}_i, \theta) \tag{35}$$

By taking the gradient on a random subset of the data B_k, we effectively add stochasticity to the algorithm. If M is the size of the minibatches, n the former dimensionality of the dataset, then $m = n/M$ is the number of minibatches:

$$\nabla_\theta C(\theta) = \sum_{i=1}^n \nabla_\theta c_i(\mathbf{x}_i, \theta) \to \sum_{i \in B_k}^n \nabla_\theta c_i(\mathbf{x}_i, \theta) \tag{36}$$

Thus, at each step the update becomes:

$$\theta_{j+1} = \theta_j - \eta_j \sum_{i \in B_k}^n \nabla_\theta c_i(\mathbf{x}_i, \theta) \tag{37}$$

The advantage of taking random samples at each iteration has the following advantages:

— it accelerates the algorithm;

— it adds noise to the descent "path" followed by the algorithm, decreasing the probability to be stuck in local minima.

3.5.b Implementation

The iteration scheme consists of first iterate over the number of `M`-sized minibatches `m`, then repeat the loop for a number of times called *epochs*. One epoch correspond to the iteration over every single minibatch. A pseudocode can possibly be:

```
for epoch in range(epochs):
  for i in range(m):
    # pick M random samples
    # = create minibatch Bk
    k = M * random_index
    # compute grad of data in Bk
    # update
```

This is fully implemented in a function `theta_sgd_mom`, which is very similar to `theta_gd_mom`, a part from the fact that it incorporates this loop scheme at the beginning.

4 Bias – Variance decomposition and tradeoff, resampling techniques

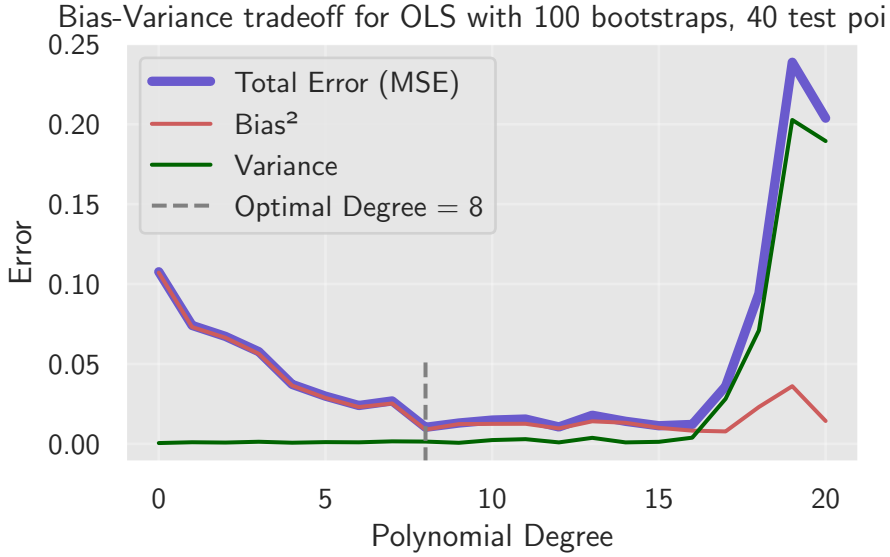

Figure 10: Bias-Variance tradeoff | Obtained with bootstrapping

4.1 Theory

In order to have a proper understanding of how the total accumulated error of a model evolves as a function of the model complexity, a decomposition of this error is necessary.

The total error (the cost function, namely) is expressed as:

$$C(\boldsymbol{X},\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}\left[(\boldsymbol{y}-\tilde{\boldsymbol{y}})^2\right] \quad (38)$$

where $\mathbb{E}\left[(\boldsymbol{y}-\tilde{\boldsymbol{y}})^2\right]$ is the expectation value of the error. This last expression can be rewritten in terms of the true function of $\boldsymbol{x}$ and its error, $\boldsymbol{y}+\boldsymbol{\epsilon}$:

$$\mathbb{E}\left[(\boldsymbol{y}+\boldsymbol{\epsilon}-\tilde{\boldsymbol{y}})^2\right] \quad (39)$$

By adding and subtracting $\mathbb{E}[\tilde{\boldsymbol{y}}]$, we obtain:

$$\mathbb{E}\left[(\boldsymbol{y}-\tilde{\boldsymbol{y}})^2\right] = \mathbb{E}\left[(\boldsymbol{y}+\boldsymbol{\epsilon}-\tilde{\boldsymbol{y}}+\mathbb{E}\left[\tilde{\boldsymbol{y}}\right]-\mathbb{E}\left[\tilde{\boldsymbol{y}}\right])^2\right] \quad (40)$$

and by expanding the second degree polynomial we obtain the following expression:

$$\begin{aligned}\mathbb{E}\Big[(\boldsymbol{y}-\mathbb{E}[\hat{\boldsymbol{y}}]+\mathbb{E}[\hat{\boldsymbol{y}}]-\hat{\boldsymbol{y}}+\boldsymbol{\epsilon})^2\Big] = \\ (\boldsymbol{y}-\mathbb{E}[\hat{\boldsymbol{y}}])^2+\mathbb{E}\big[(\hat{\boldsymbol{y}}-\mathbb{E}[\hat{\boldsymbol{y}}])^2\big]+\mathbb{E}[\boldsymbol{\epsilon}^2] \\ +2\,\mathbb{E}[\boldsymbol{y}-\mathbb{E}[\hat{\boldsymbol{y}}])(\mathbb{E}[\hat{\boldsymbol{y}}]-\hat{\boldsymbol{y}})] \\ +2\,\mathbb{E}[\boldsymbol{\epsilon}(\boldsymbol{y}-\mathbb{E}[\hat{\boldsymbol{y}}])]+2\,\mathbb{E}[\boldsymbol{\epsilon}(\mathbb{E}[\hat{\boldsymbol{y}}]-\hat{\boldsymbol{y}})]\end{aligned}$$

All the cross terms can be simplified, because $\mathbb{E}[\hat{\boldsymbol{y}}-\mathbb{E}[\hat{\boldsymbol{y}}]]=0$ and $\mathbb{E}[\boldsymbol{\epsilon}]=0$, and we assume $\boldsymbol{\epsilon}$ is independent of $\hat{\boldsymbol{y}}$. Therefore we obtain

$$\mathbb{E}\left[(\boldsymbol{y}-\hat{\boldsymbol{y}})^2\right] = (\boldsymbol{y}-\mathbb{E}[\hat{\boldsymbol{y}}])^2+\mathbb{E}\left[(\hat{\boldsymbol{y}}-\mathbb{E}[\hat{\boldsymbol{y}}])^2\right]+\sigma^2.$$

where σ^2 is the noise variance, the irreducible error. In this expression we can distinguish the three following terms:

$$\mathbb{E}\left[(\boldsymbol{y}-\hat{\boldsymbol{y}})^2\right] = \text{Bias}^2+\text{Variance}+\sigma^2 \quad (41)$$

We could define the *bias* as the inability of a model to reproduce the true value, which is commonly caused by *underfitting*. It is therefore the term that contributes the most to total error when the model complexity is too poor.

The *variance* is, on the opposite side, a measure of how much the model is influenced by the input data. In case of overfitting, as already previously mentioned, the model becomes very train set - specific, and it will hardly adapt properly to different input samples.

The aim is, therefore, to asses which degree of complexity is able to minimize the error or, in other words, to represent a good tradeoff between bias and variance. In this section we will look therefore at how the MSE evolves as a function of p. To have a more robust and generalized estimator, we will resample

the former dataset and compute the MSE on multiple subsets of it, and eventually compute the mean of all the obtained MSEs.

4.2 Resampling techniques and bias-variance tradeoff

In order to run this analysis, two resampling techniques have been employed for resampling the input dataset $\boldsymbol{x}$: the *bootstrap* and the *cross-validation*.

The `BiasVarianceTradeoff` is the class we implemented for these purposes. Only the analytical OLS regression has been used to test it.

4.2.a The bootstrap

The idea of the bootstrap is to sample different subsets of the whole input dataset, for a defined number of times (bootstraps), fit the model on these subsets at each iteration and compute the desired statistic on the obtained fit, in our case the MSE. A typical pseudocode of the workflow is:

```
x_train, x_test, y_train, y_test =
train_test_split(x)

for boostrap in range(bootstraps):
  x_sample, y_sample =
  resample(x_train, y_train)

  ## fit on x_sample
  ## predict: compute y_pred_sample
  ## store the the predicted values in
    ## a matrix of  shape (number of
    ## samples, number of bootstraps)
```

This is the core of the `bootstrap` function that has been implemented in the `BiasVarianceTradeoff` class. Note that it uses the `resample` function from `scikit-learn`, that automatically resamples random values from the former dataset, with the possibility to pick the same value more than once.

The function `decompose_mse` decomposes the total MSE in its components (Equation **??**), and finally `degree_range_simul` performs the both the bootstrapping and the decomposition for a defined range of polynomial degrees.

The result of this class's workflow is shown in Figure 10. What was previously explained is here well visible: with simple problems, the bias is the major contributor to the total error. With too complex problems, it is the variance instead.

4.2.b Cross validation

Cross validation is an other resampling technique that consists of dividing the former dataset in a fixed number k of subsets (*folds*); at any iteration, different combinations of folds are used as training and test sets. Like before, at each iteration the value pf the MSE resulting from the fit performed on the respective train folds is stored, and eventually the average of all the k MSEs will be computed.

A own function `k_fold_cv` has been developed for computing the MSE over k folds (and, therefore, k iterations) and a range of polynomial degrees. This function implements `scikit-learn`'s `KFold` for sampling the folds. The resulting MSE trends can be observed in Figure 11. We can see that, in order to obtain a proper range of values for the MSE, a high number of folds is required.

5 Conclusions

In the frame of this work, several techniques have been implemented in order to study the behavior of regressions model, both with analytical and numerical solutions. In order to assess the effectiveness of the implemented functions, the means squared error MSE and the determination coefficient have been computed. While regular models with analytical solution showed a fairly good response and adaptability to the application of different parameters, iterative methods have shown more numerical stability and runtime issues. A Bias-Variance tradeoff analysis has been performed in order to define the best complexity – simplicity tradeoff for a simple Ordinary Least Squares model.

References

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press. https://www.deeplearningbook.org/contents/optimization.html

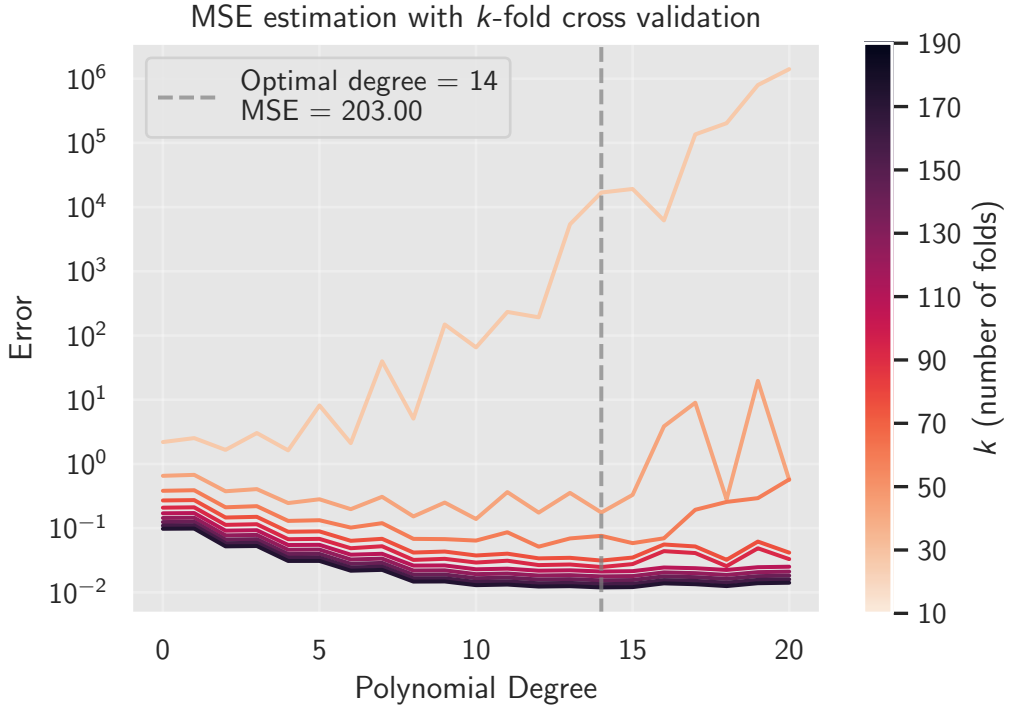

Figure 11: Cross Validation | Trends of MSE curves for different number of folds

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer. https://doi.org/10.1007/978-0-387-84858-7

Kingma, D. P., & Ba, J. (2017). ADAM: A method for stochastic optimization. https://arxiv.org/abs/1412.6980

Runge, C. (1901). Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten. *Zeitschrift für Mathematik und Physik*, *46*, 224–243.