
Regression methods

Filippo Masi

The University of Sydney, Australia

Regression methods are a fundamental pillar of supervised Machine Learning, alongside classification. Designed for graduate students and researchers seeking to gain a solid understanding of regression methods, this chapter covers essential concepts ranging from linear to nonlinear regression, including regularization techniques and Bayesian approaches.

The chapter begins by delving into linear regression methods, exploring both closed-form solutions (when available) and the more general gradient descent algorithm. It also provides a brief overview of the necessary pre- and post-processing steps involved in identifying regression models with good generalization properties, such as feature scaling and hold-out and cross-validation.

Moving on, the second part focuses on nonlinear regression methods, with a specific emphasis on polynomial regression. Their advantages and drawbacks in comparison to linear regression are discussed, along with strategies for regularization and the development of interpretable and parsimonious models.

Lastly, Bayesian regression methods are introduced as a powerful and general alternative to the previous approaches.

- References to chapters in the same book ChI : Introduction – ChANN: Artificial Neural Networks – ChCL: Classification methods – ChTANN: Thermodynamics-based ANN – ChPINN: Physics-informed NN – ChDD: Data-driven computing

1 Introduction

Regression is one of the pillars, together with classification, composing supervised Machine Learning techniques. Whereas the outputs for classification are discrete class labels, regression is concerned with the prediction of continuous quantities. By identifying how changes in independent variables impact the outcome of a dependent variable, regression is a good ally in discovering hidden patterns from intricate piles of data.

To grasp the main core of regression methods, consider a scenario where we seek to

understand the influence of relative density and effective confining pressure on the shear strength and critical state behavior of a sand sample. Figure 1(a) presents 25 drained monotonic triaxial compression tests of Karlsruhe fine sand with different initial relative densities I_{D0} and effective confining pressures [WT].

To identify the intrinsic, general relationship between some independent variables \mathbf{x} (relative density and effective pressure), and a dependent variable y (shear strength and critical state line), we assume a model of the form

$$y = f_{\theta}(\mathbf{x}), \quad (1)$$

where f_{θ} represents a function with θ denoting its parameters (unknown). Then, we

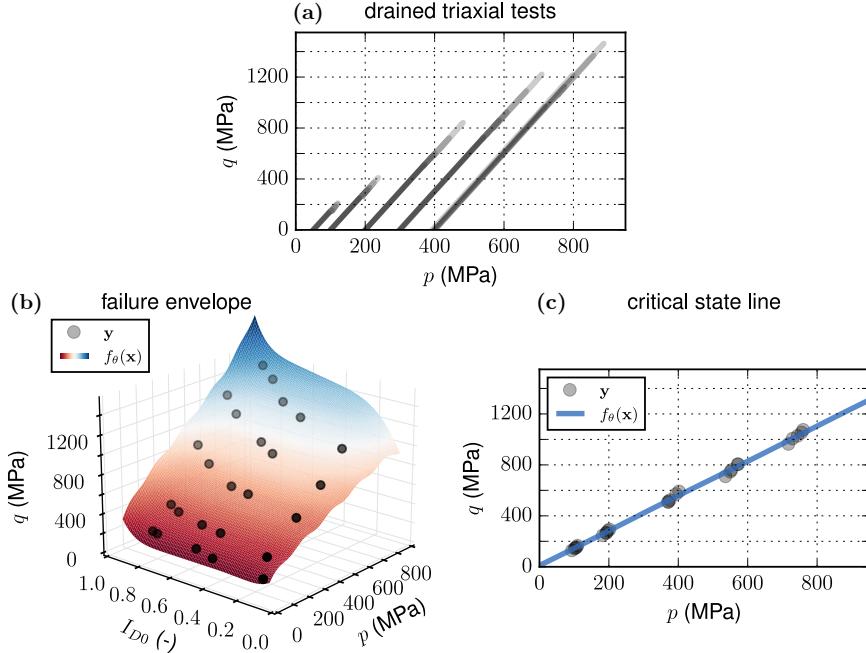


Figure 1: Drained monotonic triaxial compression tests of Karlsruhe fine sand with different initial relative densities I_{D0} and effective confining pressures [WT] (a). Non-linear regression of the failure envelope – locus of peak deviatoric stress – (b) and critical state line (c) using a basis of polynomial and trigonometric functions.

continue by formulating an objective function that measures the difference between the predictions of the model (1) and the values of the dependent variable. Via the minimization of the latter, we can determine an optimal fit and thus identify the parameters θ . By doing so, we obtain two models, $f_{\theta}(\mathbf{x})$, able to generate predictions of the failure envelope and critical state line, as depicted in Figure 1(b) and 1(c).

This process is the core of regression and enables us to uncover relationships between variables (data) and enhance our ability to make accurate predictions and gain insights

when dealing with either simple or complex systems.

This chapter provides the tools necessary for carrying such analyses by means of a comprehensive introduction to regression methods, using a “hands-on approach” – that is, providing essential theoretical aspects and prioritizing understanding through concrete examples. After studying this chapter, I hope that the reader will be able to:

- Understand the fundamental notions related to regression methods and gradient descent;
- Solve a regression problem and select the best (class of) method(s) depending on the particular task (linear, nonlinear, regularized, or Bayesian regression);
- Understand the importance of validation and test of a regression model (and in general of any Machine Learning model);
- Grasp the importance of regularization techniques in the presence of over- and/or under-determined regression problems.
- Understand the benefits and drawbacks of Bayesian regression as compared to the classical frequentist approach.

All codes used in this chapter are available on ALERT Geomaterial GitHub: repository “2023-doctoral-school” <https://github.com/alert-geomaterials/2023-doctoral-school> and updated versions can be found at <https://???.>

This chapter adopts the following notation: scalar values and function names are represented in *lowercase italic* font, while **lowercase bold** font denotes vectors. Matrices are denoted in **uppercase bold** font.

2 Linear regression

In linear regression [Man82], the idea is to select a linear function f_{θ} , whose parameters θ are identified to fit data by means of the minimization of some error metric between the predictions, i.e., the outputs of the function, and the data. As we will see, linear regression can be formulated as a simple solution of a linear system.

To fix the ideas, let us start by considering a *training* data set, with n data points or *snapshots*, composed of p dependent variables \mathbf{Y} and m independent variables \mathbf{X}

$$\mathbf{Y} \equiv \begin{bmatrix} | & | & | & | \\ \mathbf{y}^{(1)} & \mathbf{y}^{(2)} & \dots & \mathbf{y}^{(n)} \\ | & | & | & | \end{bmatrix}, \quad \mathbf{X} \equiv \begin{bmatrix} | & | & | & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \dots & \mathbf{x}^{(n)} \\ | & | & | & | \end{bmatrix}, \quad (2)$$

where $\mathbf{y}^{(k)}$ and $\mathbf{x}^{(k)}$ are the vector collecting the p dependent variables and the m independent variables, respectively, at the k -th snapshot of the data set.

In the following, we will consider for simplicity one-dimensional dependent variables, i.e., $p = 1$, and we will refer to the vector collecting the snapshots as \mathbf{y} . However the entire framework developed herein can be easily formulated in higher dimensions.

To find a best fit line through the training data points, we further assume a linear model – this is why we call it linear regression –

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_m x_m, \quad (3)$$

where \hat{y} is the value predicted by the model, x_k is the k -th independent variable value, and θ_k is the k -th model parameter – including the *bias* term, θ_0 , and the model's *weights*, $\theta_1, \theta_2, \dots, \theta_m$. The above equation can be formulated more concisely as

$$\hat{y} = f_{\boldsymbol{\theta}}(\mathbf{x}) \equiv \boldsymbol{\theta}^T \mathbf{x}^*, \quad (4)$$

where $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \dots, \theta_m]^T$ is the vector collecting the model's parameters, $\mathbf{x}^* = [x_0, x_1, x_2, \dots, x_m]^T$ is the augmented feature vector with $x_0 = 1$, $f_{\boldsymbol{\theta}}(\cdot)$ is the *hypothesis function* – a linear function, in this case –, and the superscript T denotes the transpose operator. To simplify notation, we will just write interchangeably \mathbf{x} for both \mathbf{x} and \mathbf{x}^* .

The next step is to identify the parameters $\boldsymbol{\theta}$ such that the linear regression model, $f_{\boldsymbol{\theta}}(\mathbf{x})$, best fits the data set (\mathbf{X}, \mathbf{y}) . This operation is called *training*. Training identifies the parameters value that optimize a measure of the *goodness-of-fit*, i.e., how well (or poorly) a model fits the data.

Various objective functions, often referred to as loss functions, can be adopted and their choice strongly determines the subsequent model. Three standard error metrics of hypothesis $f_{\boldsymbol{\theta}}$ on a set \mathbf{X} are often considered which are associated with the ℓ_1 and ℓ_2 norms, respectively defined as the mean absolute error (MAE) and the root mean square error (RMSE):

$$E_1(\mathbf{X}, f_{\boldsymbol{\theta}}) \equiv \frac{1}{n} \sum_{k=1}^n |f_{\boldsymbol{\theta}}(\mathbf{x}^{(k)}) - y^{(k)}| \quad (5a)$$

$$E_2(\mathbf{X}, f_{\boldsymbol{\theta}}) \equiv \left(\frac{1}{n} \sum_{k=1}^n (f_{\boldsymbol{\theta}}(\mathbf{x}^{(k)}) - y^{(k)})^2 \right)^{1/2}. \quad (5b)$$

One can also broadly define the error based on the ℓ_r norm, and namely

$$E_r(\mathbf{X}, f_{\boldsymbol{\theta}}) \equiv \left(\frac{1}{n} \sum_{k=1}^n |f_{\boldsymbol{\theta}}(\mathbf{x}^{(k)}) - y^{(k)}|^r \right)^{1/r}. \quad (6)$$

The higher the norm index r , the more the error metrics focuses on large values and neglect small ones, thus the best fit model intrinsically depends on r . In most cases, the differences between models based on different norms are small. However, when there are outliers¹ in the data, the choice of norm can have a significant impact: for instance, the RMSE is more sensitive to outliers than the MAE.

In linear regression, the most common choice is the root mean square error, but in

¹An outlier is an observation that lies an abnormal distance from the other values present in a data set.

practice, it is simpler to minimize the mean squared error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root). The MSE of a linear regression hypothesis f_{θ} on a set \mathbf{X} is calculated as

$$\text{MSE}(\mathbf{X}, f_{\theta}) \equiv E_2^2(\mathbf{X}, f_{\theta}) = \frac{1}{n} \sum_{k=1}^n (\theta^T \mathbf{x}^{(k)} - y^{(k)})^2. \quad (7)$$

Once the loss function is defined, MSE in our case, training requires to find the parameters θ that minimize that particular loss. This requires differentiation with respect to θ to identify the value of the latter such that a minimum of the error occurs – that is, find those θ for which $\partial \text{MSE}(\mathbf{X}, f_{\theta}) / \partial \theta = 0$.

Note that, although a zero derivative denotes either a minimum or a maximum, we know this must be a minimum of the error since there is no maximum error, i.e., we can always find a model that has a larger error.

Following the above procedure, the values of the parameters for which the error is minimum, $\hat{\theta}$, are the solution of the following linear system of equations

$$\mathbf{X}^T \mathbf{X} \hat{\theta}^T = \mathbf{X}^T \mathbf{y}. \quad (8)$$

If the matrix $\mathbf{X}^T \mathbf{X}$ is square and invertible (i.e., it has nonzero determinant), then there exists a unique solution $\hat{\theta}$ which is given by the *normal equation*,

$$\hat{\theta}^T = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}. \quad (9)$$

However, when $\mathbf{X}^T \mathbf{X}$ is singular, the normal equation does not hold. An alternative way to solve equation (10) consists of using the Moore-Penrose pseudoinverse \mathbf{X}^+ of \mathbf{X} to obtain the general and unique solution

$$\hat{\theta} = \mathbf{X}^+ \mathbf{y}, \quad (10)$$

where we have used the commutative property $(\mathbf{X}^+)^T = (\mathbf{X}^T)^+$. The pseudoinverse is computed using the matrix factorization technique called singular value decomposition that can decompose the independent variables matrix \mathbf{X} into the multiplication of three matrices $\mathbf{U} \Sigma \mathbf{V}^T$. The pseudoinverse is computed as $\mathbf{X} = \mathbf{V} \Sigma^+ \mathbf{U}$. To compute the matrix Σ^+ , the algorithm takes Σ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix.

With all the ingredients in our hands, is time to move to a concrete example.

2.1 Example

Let consider fitting the following data set (shown in Figure 2), generated by the function

$$y = \alpha_2 \exp(\alpha_1 x + \mathcal{U}(-1, 1)), \quad \alpha_1 = 1, \alpha_2 = e, \quad (11)$$

which is obviously nonlinear. $\mathcal{U}(-1, 1)$ is a uniformly distributed random variable that lies between -1 and 1 . The snippet hereinafter generates our data set:

```

import numpy as np
np.random.seed(42)
n_snapshots = 200 # set number of snapshots: n
a_1 = 1.; a_2 = np.exp(1) # set coefficients
noise = np.random.uniform(-1,1,(n_snapshots,1)) # uniform noise
X = np.random.uniform(0,4.,(n_snapshots,1)) # independent variable X
y = a_2*np.exp(a_1*X+noise) # dependent variable y

```

At this point, we can use the normal equation (9) and compute the values $\hat{\theta}$, using NumPy's linear algebra module (`np.linalg`) to calculate the inverse of a matrix and the `dot()` method for matrix multiplication:

```

X_p = np.c_[np.ones((n_snapshots, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_p.T.dot(X_p)).dot(X_p.T).dot(y)
print(theta_best) # parameters
array([-16.93873601,
       [ 28.05702557]])

```

The normal equation gives a linear model of the form $\hat{y} \approx -16.9 + 28.0x$. At this point, we can check if our linear model correctly fit the training data set, making predictions as follows

```

X_new = np.array([[0],[4]])
X_b = np.c_[np.ones((2,1)), X_new] # add x0 = 1 to each instance
y_predict = X_b.dot(theta_best)

```

Finally, we can plot the predictions (`y_predict`) with the training data set, see Figure 2. As one could have imagined from the very beginning, our regression model poorly fits the nonlinear data set.

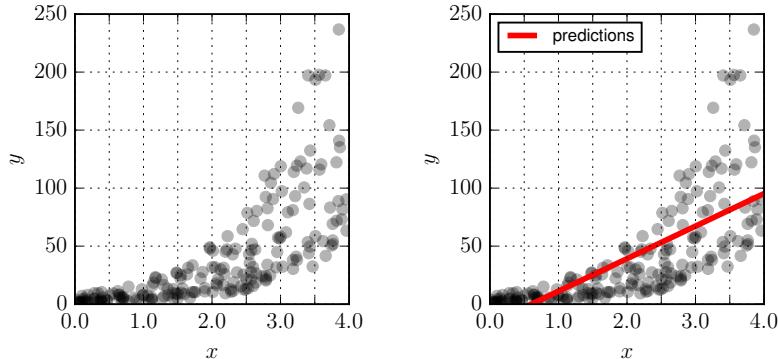


Figure 2: Linear regression: an example and a failure (!). Randomly generated data set (left) – $y = \alpha_2 e^{(\alpha_1 x + \mathcal{U}(-1,1))}$ – and linear regression model predictions (right).

But, as in most cases, data set preparation and transformation may help us to improve predictions. In this particular case, one has to simply perform a change of coordinates,

defining $y' = \ln(y)$ and $\alpha'_2 = \ln(\alpha_2)$. Such a transformation (*magically*) performs the linearization of the original data set, which from (11) now becomes

$$y' = \alpha'_2 + \alpha_1 x + \mathcal{U}(-1, 1), \quad \alpha_1 = 1, \quad \alpha'_2 = 1, \quad (12)$$

Let's try to apply the following changes to our original data set, repeating the linear regression fit, and plotting the predictions:

```

y_prime = np.log(y) # change of coordinate, y' = ln(y)
X_b = np.c_[np.ones((n_snapshots, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y_prime)
X_new = np.array([[0], [4]])
X_b = np.c_[np.ones((2,1)), X_new] # add x0 = 1 to each instance
y_predict = X_b.dot(theta_best) # parameters
print(theta_best)

array([0.99530983,
       [0.98646971]])

```

Done! The linear regression now correctly fits the transformed data set, identifying the following model: $y' = 0.98 + 0.99x$. Note that the presence of noise renders impossible to retrieve the exact values of the intercept and the angular coefficient.

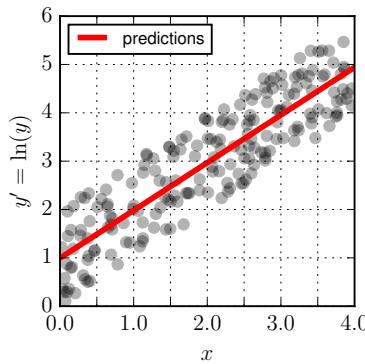


Figure 3: Comparison of the linear regression model predictions with the linearized data set, see equation (12).

Alternatively, a simpler (and more concise) way to perform a linear regression fit is to directly compute the pseudoinverse of the independent variable matrix \mathbf{X} , see Equation 10. This can be done, in few lines of code, using Scikit-Learn's library [BLB⁺13]:

```

from sklearn import linear_model
from sklearn.metrics import mean_squared_error
regr = linear_model.LinearRegression() # create linear regression object

```

```

regr.fit(X, y_prime) # train the model
y_predict = regr.predict(X) # make predictions
print("Parameters: ", regr.intercept_, regr.coef_) # parameters
print("Mean squared error: %.4f" % mean_squared_error(y_prime,
                                                       y_predict)) # compute MSE

```

Parameters: [0.99530983] [[0.98646971]]
Mean squared error: 0.3459

3 Gradient descent

We have seen that linear regression allow us to fit data sets with pre-identified linear models or hypothesis f_θ . The advantages are (i) the admission of analytically tractable, best-fit solutions and (ii) the reduced computational complexity. However, the major drawback is the impossibility to fit nonlinear functions, much more abundant in nature than linear ones (cf. chapters ChANN, ChCL).

To this end, the general theory of nonlinear regression² considers a nonlinear hypothesis function $f_\theta(x)$, in contrast with the linear function in Equation (3). In this case, if we proceed as we have done for linear regression – that is, requiring that $\partial \text{MSE}(\mathbf{X}, f_\theta) / \partial \boldsymbol{\theta} = 0$ – we obtain the following nonlinear system of equations

$$\frac{1}{n} \sum_{k=1}^n \left(f_\theta(\mathbf{x}^{(k)}) - y^{(k)} \right) \frac{\partial f_\theta}{\partial \boldsymbol{\theta}} = 0. \quad (13)$$

Unfortunately, there are no analytical methods to solve such a nonlinear system for a general nonlinear hypothesis f_θ . And sometimes Equation (13) may even not admit a solution or admit an infinity of solutions.

In such scenarios, the idea to solve the nonlinear system (13) is to resort to iterative approaches which, depending on the good (or bad) initial guess, may converge to the global (or a local) minimum error.

One of the most effective approaches to identify the roots of a nonlinear system of equations is *gradient descent* [BV04]. Gradient descent (GD), also called steepest descent, is an optimization algorithm for finding a local minimum of a differentiable function – that is, a set of optimal parameters $\boldsymbol{\theta}$ that minimizes a given (differentiable) loss function.

A quite easy way to grasp the essence of gradient descent is to imagine yourself lost in a foggy mountain, where visibility is so limited that you can only sense the slope beneath your feet. To find your way out quickly, a smart approach would be to head downhill, following the steepest slope (*WARNING, do not try this neither at home nor in Aussois!*). This is exactly what gradient descent does: it measures the local gradient of the loss function with regard to the parameters $\boldsymbol{\theta}$ and tweaks the latter to go in the direction of descending gradient. Once the gradient is zero, we have reached the

²Note, however, that if we formulate the nonlinear regression problem differently, within a reduced special setting, a seemingly linear hypothesis function can actually be formulated, cf. Section 5.

minimum.

Let's break it down further. At the beginning, θ is filled with random values, or with other predefined values. This process is called *initialization*. Then, in small incremental steps, often called *epochs*, the algorithm tweaks repeatedly the parameters to minimize the value of the loss function $E(\theta, f_\theta)$ (that must be differentiable for all θ – which also implies that f_θ must be differentiable for every θ) according to the regular gradient descent optimizer equation

$$\theta^{\text{next step}} = \theta - \eta \frac{\partial E}{\partial \theta}(\theta, f_\theta), \quad (14)$$

where η is the *learning rate*. In other words, we need to compute how much the loss function E will change if we change θ_i just a little bit, with $i = 1, 2, \dots, m$.

By resorting to the analogy of the foggy mountain, it is like asking “What is the slope of the mountain under my feet if I face east?” and then asking the same question facing north (and so on for all other dimensions i up to m , if you can imagine a universe with more than three dimensions).

After having computed the gradient vector $\partial E / \partial \theta$ which points uphill, we just have to head in the opposite direction to go downhill – that is, subtracting the gradient from the parameters θ , with a weighting factor η . This process continues until the algorithm converges, reaching a (global or local) minimum value, as shown in Figure 4.

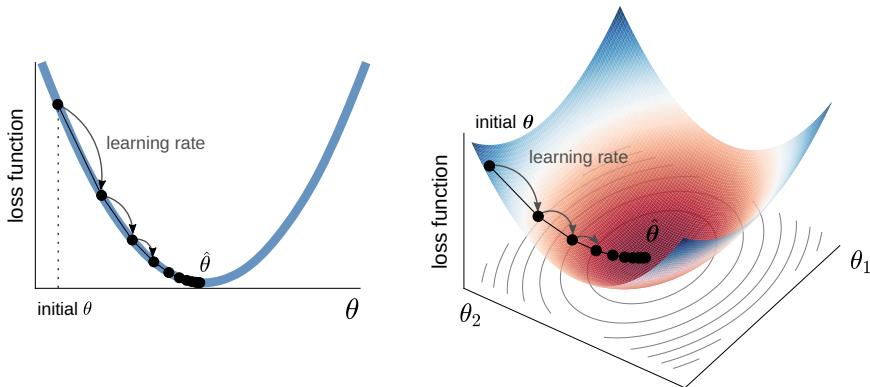


Figure 4: Schematic representation of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the parameters approach the minimum.

When optimizing an objective function using GD two important aspects must be considered:

- The size of the incremental steps, which is determined by the **learning rate**, η . The latter is a *hyperparameter*. A hyperparameter is a variable controlling the learning process (in contrast to the model parameters that are determined via

training). The learning rate, as such, is crucial in identifying the best value of the parameters θ within the smallest number of iterations.

Indeed, if η is set too small, the algorithm will require numerous iterations to converge, resulting in a time-consuming computations. Conversely, if η is excessively high, we might overshoot the minimum and end up on the opposite side, potentially at a point where error is higher than before. This scenario could even cause the algorithm to diverge, producing increasingly larger error values and failing to find an optimal solution.

- The **loss function**. Not all loss functions exhibit a smooth, bowl-like shape as the ones depicted in Figure 4. Depending on the choice of loss function and/or the hypothesis function, we may find irregularities such as holes, ridges, plateaus, and many local minima, creating challenges for convergence towards the (global) minimum. Also remember that the loss function must always be differentiable³, otherwise we cannot compute the optimizer equation (14).

Figure 5 highlights some of the primary obstacles that we may encounter with some loss functions and, in most of the cases, for nonlinear hypothesis f_θ – but also in neural networks, cf. chapter ChANN. When the algorithm starts on the left due to random initialization, it attempts to converge to a local minimum, which may not be as desirable as the global minimum. This same local minimum is a point of discontinuity which renders even more difficult the convergence. On the other hand, if the algorithm begins on the right, it finds a plateau and crossing it takes an extensive amount of time and if we prematurely stop the algorithm we won't succeed in reaching the global minimum.

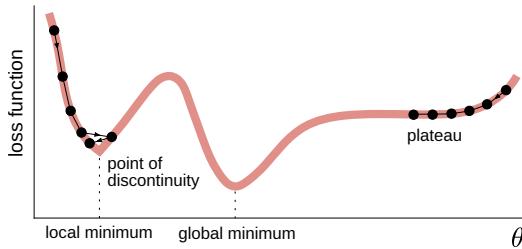


Figure 5: Gradient descent for a general non convex loss function.

The MSE for linear regression models happens to be always differentiable and convex: great news! This means that any two points on the curve can be connected by a line segment that never intersects the curve. Consequently, there are no local minima, only one single global minimum. Moreover, the MSE function is continuous, exhibiting a slope that changes gradually (differently from the MAE). These characteristics have a significant implication: given sufficient time and an appropriate value of the learning

³In reality, when dealing with a loss function that is not everywhere differentiable, an ad-hoc subgradient procedure can be used: instead of calculating the gradient, we use a subgradient that provides a valid lower bound on the slope of the loss function at that particular singular point.

rate, gradient descent is guaranteed to approach the global minimum closely.

In linear regression, the shape of the MSE loss function always resembles a bowl. However, we should note that the latter can be elongated if the parameters – that is, the features – have different scales. Figure 4 (right) demonstrates GD applied to a set with features θ_1 and θ_2 , where the latter has significantly smaller values than the former. Note that, in general, when features have quite different scales, the convergence to the global minimum requires a large number of iterations. Such a drawback can be easily overcome by appropriately scaling the features – that is, by scaling the independent and dependent variables (cf. Section 4).

It's worth noting that training a model, whether linear or nonlinear, involves searching for a perfect combination of model parameters that minimizes the loss function across the entire data set. This search takes place within the vast parameter space of the model. Now, imagine we are trying to train a model with an extensive set of parameters, surpassing the simplicity of the one- and two-parameters models in Figure 4. In this scenario, we are essentially following the same steps as before, but now in a higher-dimensional space. The dimension of this space is equal to the number of parameters, making the search much more challenging. Picture this: finding a needle in a haystack that has 1000 dimensions is significantly more intricate than doing so in just two dimensions. Thankfully, in the case of linear regression (unlike artificial neural networks, see chapter ChANN), the loss function has optimal properties and the needle (global minimum) resides comfortably at the bottom.

3.1 Batch gradient descent

Batch gradient descent is the most cumbersome implementation of the GD algorithm. It involves the use of Eq. (14) for the full training set \mathbf{X} at every step – that is, the whole batch of training data (and this is the reason of its name).

Applying batch GD for a linear regression problem starts with the computation of the partial derivative of the MSE function with respect to the model's parameter,

$$\frac{\partial}{\partial \theta} \text{MSE}(\mathbf{X}, f_{\theta}) = \frac{2}{n} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y}), \quad (15)$$

and continues with using recursively Equation (14) to update the model's parameters. The recursion stops whether we reach the prescribed number of steps (epochs) or the error is smaller than set threshold.

We can now use the training data set from the previous example and use batch GD to perform a linear regression:

```
eta = 0.05 # learning rate
n_epochs = 500 # no epochs
theta = np.array([[0.],[0.]]) # initialization
for epoch in range(n_epochs):
    gradients = 2/n_snapshots * X_p.T.dot(X_p.dot(theta) - y_prime) # Eq. (15)
    theta = theta - eta * gradients # optimizer Eq. (14)
print(theta)
```

```
[[0.99530187]
 [0.98647279]]
```

It should come as no surprise that we obtain the same result as using the normal equation. However, we might wonder what would happen if we had chosen a different learning rate. In Figure 6, we visualize the predictions of the same linear model within the first 10 iterations using different learning rates, all starting from the same initial point (represented by the dotted line). Similarly, Figure 7 illustrates the evolution of the model parameters, θ_1 and θ_2 , with the contours representing the loss function (MSE) that has a global minimum approximately located at $(1, 1)$.

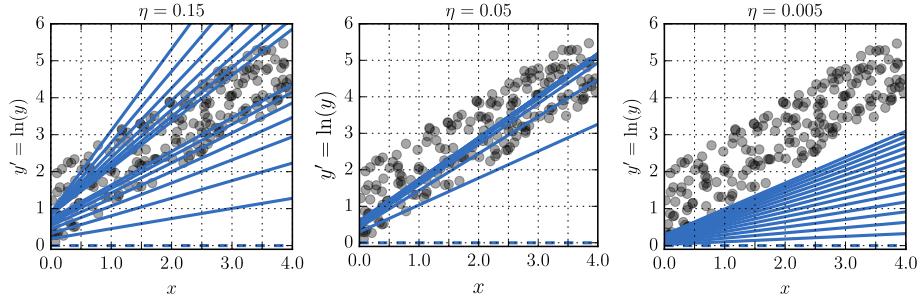


Figure 6: Batch gradient descent with various learning rates from the same starting point.

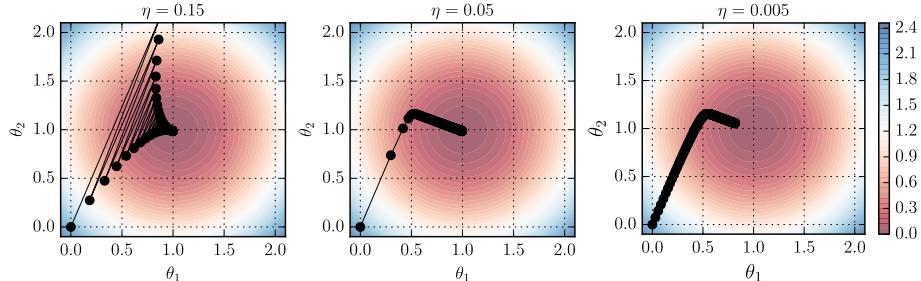


Figure 7: Evolution of the model's parameters during batch gradient descent with various learning rates (see Figure 6).

Starting from the left, we observe that with a relatively high learning rate, the algorithm bounces around during each iteration. In the middle, a moderately appropriate learning rate allows gradient descent to swiftly reach the global minimum. On the right, with a very low learning rate, the algorithm will eventually converge to the solution, but it will take a considerable amount of time.

To identify an appropriate learning rate value, we can employ two strategies: first, we can use an adaptive learning rate (see paragraph 3.2) or second, we can fine-tune the learning rate using grid search⁴ [Gér22].

⁴Grid search is a technique used in ML to systematically search for the optimal combination of hyperparameters.

Despite being much faster than using the normal equation or SVD decomposition for high-dimensional linear regression tasks, batch GD is the most computationally demanding among all gradient descent variants and the most vulnerable to “bad” initialization. The evaluation of the optimizer equation (14) is often computationally intractable, especially for deep neural networks (cf. chapter ChANN). This is primarily due to two reasons: (i) the parameter vector, collecting the parameters θ_k , can be quite large, and (ii) the number of data points n can also be large. Thus, utilizing the entire training set for computing the gradient can hinder convergence speed. As a result, we will explore two alternative algorithms that can enhance convergence speed to the minimum of the loss function: stochastic gradient descent and mini-batch gradient descent.

3.2 Stochastic gradient descent

Stochastic gradient descent is a variant of gradient descent that differs from batch gradient descent in how it estimates the gradient in equation (14). Instead of using all n training data points, stochastic GD randomly selects a single datum from the training set to estimate the gradient at each iteration. This choice of a single datum makes stochastic GD faster, as it only needs to process a single instance at a time. As a result, it is advantageous when dealing with large data sets, as it allows for efficient memory allocation (see, also, chapter ChANN).

However, due to its stochastic nature, stochastic GD exhibits less smooth behavior compared to batch GD. Instead of smoothly decreasing towards the minimum, the loss function experiences fluctuations, with only an average error decrease over iterations. Consequently, when the algorithm stops, the final parameter values are generally good but not necessarily optimal. Yet, it is worth noting that the randomness inherent in stochastic GD can actually aid in escaping eventual local minima, increasing the chances of finding the global minimum compared to batch GD.

Hereinafter, we implement the stochastic GD

```
eta = 0.05 #learning rate
n_epochs = 20 # no epochs
theta = np.array([[0],[0]]) #initialization
for epoch in range(n_epochs):
    for i in range(n_snapshots): # iterate over each snapshot
        random_index = np.random.randint(n_snapshots) # pick random snapshot
        xi = X_p[random_index:random_index+1]
        yi = y_prime[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi) # Eq. (15)
        theta = theta - eta * gradients # Eq. (14)
    print(theta)
[[0.83905266]
 [1.15041967]]
```

parameter values for a given model. It involves specifying a set of possible values for each hyperparameter and exhaustively evaluating the model performance using all possible combinations of these values. By evaluating the model accuracy for each combination, grid search helps identify the hyperparameter configuration that yields the best performance.

In Figure 8, we can observe that the algorithm struggles in converging to the global minimum, rather bouncing all around. This is to the significant dependence of stochastic GD on the learning rate, which in turn is due to inherent stochastic nature of the former.

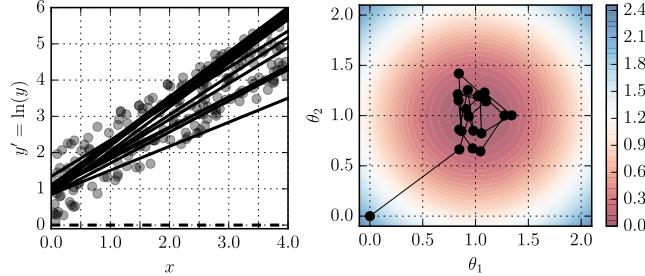


Figure 8: Stochastic gradient descent and evolution of the parameters, with constant learning rate.

The problem can be alleviated by gradually adapting the learning rate while training. Below, we repeat the learning process using a rather simple learning schedule that linearly decrease the learning rate at each epoch.

```
theta = np.array([[0],[0]]) #initialization
t0, t1 = 1, 100 # learning schedule hyperparameters
def learning_schedule(t):
    return t0 / (t + t1)
for epoch in range(n_epochs):
    for i in range(n_snapshots):
        random_index = np.random.randint(n_snapshots)
        xi = X_p[random_index:random_index+1]
        yi = y_prime[random_index:random_index+1]
        eta = learning_schedule(epoch * n_snapshots + i) # schedule
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        theta = theta - eta * gradients
    print(theta)
[[0.90607535]
 [1.02276101]]
```

Super effective! In 20 iterations only, the algorithm converges – relatively close – to the global minimum, as depicted in Figure 9. And this should be compared with the 500 iterations that were needed for the batch GD to reach the minimum.

3.3 Mini-batch gradient descent

Once we have a grasp both batch and stochastic GD, understanding mini-batch GD becomes straightforward if we get inspiration from Aristotle's (Latin) quote: "*In medio stat virtus*" – that is, "*the best option lies between two extremes*".

Instead of computing gradients based on the entire training set (batch GD) or a single

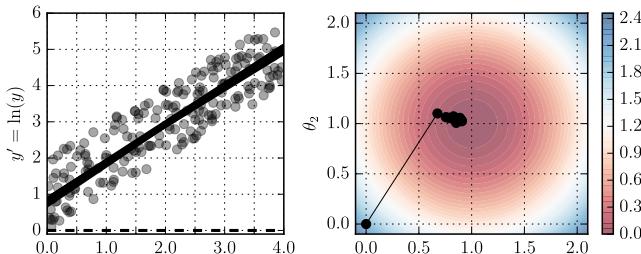


Figure 9: Stochastic gradient descent with learning schedule.

data point (stochastic GD), mini-batch GD computes gradients on small random sets of data points known as *mini-batches*. This approach offers the advantage of leveraging optimized matrix operations, resulting in improved performance compared to stochastic GD. At the same time, it avoids the need for large memory allocation required by batch GD.

The optimization path followed by mini-batch GD is less erratic than its stochastic counterpart, often leading to a more optimal minimum.

Implementing mini-batch GD is relatively simple. We only need to make slight modifications to the stochastic GD algorithm and provide an iterator (`iterate_minibatches`) to extract mini-batches.

```
def iterate_minibatches(inputs, targets, batchsize, shuffle=False):
    '''Iterator over mini-batches
    :param inputs, targets: independent and dependent variables
    :param batchsize: mini-batch size
    :param shuffle: shuffle mini-batches
    :return: mini-batches of inputs and targets'''
    # Check if no of samples in inputs and targets are equal
    assert inputs.shape[0] == targets.shape[0]
    if shuffle:
        indices = np.arange(inputs.shape[0]) # Generate array of indices
        np.random.shuffle(indices) # Shuffle the indices
    # Iterate over mini-batches
    for start_idx in range(0, inputs.shape[0] - batchsize + 1, batchsize):
        if shuffle: # subset of shuffled indices
            excerpt = indices[start_idx:start_idx + batchsize]
        else: # slice to select samples
            excerpt = slice(start_idx, start_idx + batchsize)
        yield inputs[excerpt], targets[excerpt] # return mini-batches
eta = 0.05 #learning rate
theta = np.array([[0],[0.]]) #initialization
batch_size = 30 # define mini-batches size
for epoch in range(n_epochs):
    for batch in iterate_minibatches(X_p, y_prime, batch_size, shuffle=True):
        x_batch, y_batch = batch
        gradients = 2/batch_size * x_batch.T.dot(x_batch.dot(theta) - y_batch)
        theta = theta - eta * gradients
print(theta)
```

```
[[0.94862053]
 [0.96679924]]
```

Done: much faster and accurate than the stochastic GD!

Figure 10 depicts the predictions of our model within the first 10 iterations of the algorithm and the parameters evolution. Note that the value of the minimum reached by the mini-batch GD intrinsically depends on the selected size of the mini-batch. The latter represents yet another important hyperparameter of our optimization problem.

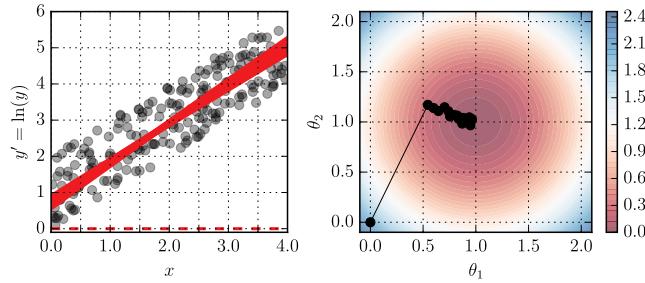


Figure 10: Mini-batch gradient descent.

To perform the same task, we could also use Scikit-Learn, and namely the `SGDRegressor` class, which by default minimizes the MSE. The following code runs for maximum 20 epochs or until the loss drops by less than 0.001 during one epoch (`max_iter=20`, `tol=1e-3`). It starts with a learning rate equal to `eta` and use the following learning schedule: $\eta^{(\text{next step})} = \eta t^{-1/4}$. The option `penalty=None` avoids the use of regularization strategies (cf. Section 6).

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=20, tol=1e-3, penalty=None, eta0=eta)
for epoch in range(n_epochs):
    for batch in iterate_minibatches(X_p, y_prime, batch_size, shuffle=True):
        x_batch, y_batch = batch
        sgd_reg.partial_fit(X, y_prime.ravel()) # partial_fit mini-batch
print(sgd_reg.intercept_, sgd_reg.coef_)
```

```
[1.00172812] [1.00508048]
```

Also note that if we were performing stochastic GD, we would not need to iterate over the mini-batches, and we should replace `partial_fit()` with `fit()`.

In conclusion, the three alternative algorithms for gradient descent – batch, stochastic, and mini-batch – are powerful tools for solving an optimization problem and they should be preferred to the normal equation and the SVD decomposition, when dealing with a high-dimensional linear regression problem (with large numbers of data sets and/or large number of parameters).

The main difference among the three alternatives is summarized by Figure 11 which provides a comparison for the parameters trajectory during training. The mini-batch GD is often preferred as it allows a good compromise in terms of the goodness of the

reached minimum and the computational speed, however it is difficult to provide strict recommendations for a general case. As a rule of thumb, we may opt for a mini-batch GD and train the model with different mini-batch sizes – if equal to 1, we will resort to stochastic GD, if equal to the training set size, we will end up with a batch GD.

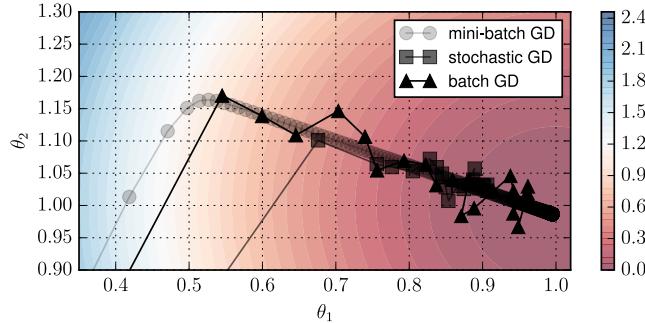


Figure 11: Gradient descent paths comparison in the parameters space.

Herein, we limited ourselves to the standard optimizer equation (14), however alternative (and most effective) solutions exist (cf. chapter ChANN).

4 Data preprocessing and model validation

In the above examples, we have seen how to perform linear regression given some training data sets. However, we have neglected two important aspects common to all ML models – regression included: *feature scaling* and *models validation*. The former is part of the preprocessing to train a model, while the latter can be interpreted as postprocessing of the learning process.

This Sections gives an overview of the two processes and the interested reader is invited to read chapter 2 in [Gér22].

4.1 Feature scaling

Feature scaling is a crucial step in preparing data for ML. Most algorithms struggle when the numerical attributes in the input have different scales (e.g. Figure 4). There are two common methods to address this issue: normalization and standardization. Both techniques transform a feature x into its scaled counterpart \bar{x} , according to

$$\bar{x} = \frac{1}{\alpha} (x - \beta). \quad (16)$$

Normalization shifts and re-scales the feature values to a range between -1 and 1, by considering $\alpha \equiv \frac{1}{2} (\max(x) - \min(x))$ and $\beta \equiv \alpha + \min(x)$.

On the other hand, standardization transform the feature in a distribution with a zero

mean and unit variance, by considering $\alpha \equiv \sigma(x)$ and $\beta \equiv \mu(x)$, with μ and σ being the mean and standard deviation, respectively.

Normalization is simple and confines values to a specific range, while standardization is not bound to a range and is less affected by outliers. The choice depends mainly on the type of loss function and the statistics of the data sets – when the latter are uniformly distributed, normalization is to be preferred, in all the other cases, standardization is the best choice.

Hereinafter an explicit implementation of both scaling techniques:

```
def scaling(x, fit=False, transform=False, inverse_transform=False,
           norm=True, param=None):
    ''' scale variable x
        :param x: variable
        :param fit: find scaling parameters
        :param transform: scale x
        :param inverse_transform: inverse scale x
        :param norm: inverse scale x
        :param fit: scaling parameters
        :return: scaling parameters (if fit=False)
                 scaled x (if transform=True)
                 inverse scaled x (if inverse_transform=True) '''
    if fit==True:
        if norm==True:
            min_ = np.amin(x); max_ = np.amax(x)
            a = 0.5*(max_-min_)
            b = 0.5*(max_+min_)
        else:
            a = np.std(x)
            b = np.mean(x)
        return [a,b]
    elif transform==True:
        return np.divide(x-param[1],param[0])
    elif inverse_transform==True:
        return np.multiply(x,param[0])+param[1]
```

One could also use directly Scikit-learn's preprocessing module.

4.2 Test and validation of a model

In the examples above, we discovered regression models that best fits some training data $\{X, y\}$. However, in the general case, and not only in the frame of regression methods, we need to evaluate how well our ML model performs on new data. To this end, let us define three important notions: *interpolation*, *generalization*, and *extrapolation*.

Definition. *Interpolation refers to the ability of predicting values within the range of observed data points.*

In the context of regression, it means predicting the value of the dependent variables for values of independent variables that lie within the range of the training data.

Definition. *Generalization refers to the ability of a model to adapt properly to new, previously unseen data, drawn from the same distribution as the ones used to train the model.*

Machine learning is all about having models that can generalize well. Trained to solve one problem, the model attempts to utilize the patterns/relationships learned from that task to solve the same task, with slight variations. Whilst the existence of a subtle difference between interpolation and generalization, both terms are often adopted interchangeably.

Definition. *Extrapolation involves predicting values beyond the range of observed, training data points.*

Extrapolation requires a solid understanding of the relationships identified by a model to apply the same outside the familiar data range. It is challenging for current ML models to reliably extrapolate, as they often specialize in specific tasks and struggle with broader applications. Many artificial intelligence methods are inherently interpolative, and it remains an open challenge to be able to construct an extrapolative or "intelligent" algorithm (cf. chapter ChANN and ChTANN).

From the aforementioned definitions, it is clear that we should **always test our models**. A cumbersome approach consists of dividing the original data set into two: a *training set* and a *test set*.

The training set is used to train the model, as we have already seen, while the test set is used to assess its performance. The latter is estimated by means of the generalization error, which measures the error of the trained model on the (unseen) test data set. Thus, if a model has low training error but high generalization error, it indicates *overfitting* to the training data. Overfitting is defined as the phenomenon where a model becomes overly complex and excessively fits the training data set, capturing noise and irrelevant patterns instead of learning the true underlying patterns and relationships.

However, evaluating the generalization error multiple times on the same test set and optimizing the model and hyperparameters (such as the learning rate in gradient descent, or the mini-batch size in mini-batch GD) specifically for that set may be suboptimal. Indeed, our model will unlikely perform as well on new data.

4.2.1 Hold-out validation

To mitigate the problem arising from testing a model on the same test set, a common solution is *hold-out validation*. A portion of the training set is reserved as a *validation set*. Multiple candidate models with different hyperparameters are trained on the reduced training set, and the one that performs best on the validation set is selected. The best model is then trained on the full training set, including the validation set, to create the final model. The performance of this final model is evaluated on the test set to estimate its generalization error.

While hold-out validation is effective, the size of the validation set is crucial. If it is too small, evaluations may be imprecise, leading to the selection of suboptimal mod-

els. Conversely, if the validation set is too large, the remaining training set becomes significantly smaller, making it unfair to compare models trained on a much smaller set.

As a rule of thumb (and a first good guess), we often split data in 80%-20%-20% for training-validating-testing. We can easily do it, by leveraging scikit-learn:

```
from sklearn.model_selection import train_test_split
# split into training+validation and test sets
X_tv, X_tv, y_train, y_test = train_test_split(X, y, test_size=0.2)
# split into training and validation sets - note: 0.25 * 0.8 = 0.2
X_train, X_val, y_train, y_val = train_test_split(X_tv, y_tv, test_size=0.25)
```

4.2.2 Cross-validation

Repeated hold-out validation addresses the issue related to the choice of the validation set size by using multiple small validation sets. This process is called k -fold cross-validation, where the training data set is further partitioned into k -folds, which are typically randomly selected portions of the original set. In k -fold cross validation, the training data is randomly partitioned into k partitions (or folds). Each partition is used to construct a regression model $\hat{y}_{(l)} = f_{\theta_{(l)}}(\mathbf{x}_{(l)})$ for $l = 1, 2, \dots, k$. Then, the final, cross-validated model is constructed by averaging the values of the parameters obtained from each fold $\boldsymbol{\theta} = 1/k \sum_{l=1}^k \boldsymbol{\theta}_{(l)}$.

Despite its great advantage over hold-out validation, cross-validation may increase tremendously the learning process as each model needs to be trained on different validation sets.

We will implement cross-validation both using Scikit-learn (cf. example in paragraph 5.1) and performing the aforementioned steps explicitly (cf. application in paragraph 7.1).

5 Nonlinear regression

Up to this point, we have seen how to train and validate linear models that best fits some data. However, what if our data are more complex and cannot be fitted, with good approximation, by a simple straight line?

In that case, we may simply resort (spoiler alert!) to artificial neural networks (see chapter ChANN). But, no worries! A much simpler alternative exists: we can actually fit a linear model to nonlinear data. To this end, we just need to replace our original set of independent variables, \mathbf{X} , with a library $\phi(\mathbf{x}) = [\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \dots, \phi_l(\mathbf{x})]^T$ of l candidate basis functions, e.g., polynomials, trigonometric or radial basis functions, or any other user-supplied function. By doing so, Equation (1) becomes

$$\hat{y} = \boldsymbol{\theta}^T \phi(\mathbf{x}). \quad (17)$$

In the following, we will consider as candidate basis functions only polynomial ones: the regression is thus polynomial.

5.1 End-to-end example

The best way is to move straightforward to an example. This time, we will also take care of preprocessing the data (see Section 4).

Let us consider a data set given by

$$y = \sin(|x|) + \sin(x^2) + \mathcal{N}(0, \sigma), \quad (18)$$

where $\mathcal{N}(0, \sigma)$ is a normally distributed random variable with mean zero and standard deviation σ .

```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(42)
n_snapshots = 200
noise = np.random.normal(0,0.5,(n_snapshots,1)) # generate normal noise
X = np.random.uniform(-np.pi,np.pi,(n_snapshots,1))
y = np.sin(np.abs(X))+np.sin(X)**2+ noise
```

Now, we proceed by splitting the data set into a training and test set, using Scikit-Learn's `train_test_split`. Then, we scale both independent and dependent variables based on the statistics of the training data set (see Section 4).

```
from sklearn.model_selection import train_test_split
# split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
# compute scaling params
param_x = scaling(X_train,fit=True) # scaling params for X
param_y = scaling(y_train,fit=True) # scaling params for y
# scale X (training and test sets)
norm_X_train = scaling(X_train,transform=True,param=param_x)
norm_X_test = scaling(X_test,transform=True,param=param_x)
# scale y (training and test sets)
norm_y_train = scaling(y_train,transform=True,param=param_y)
norm_y_test = scaling(y_test,transform=True,param=param_y)
```

At this point, we have to construct the polynomial basis functions, with Scikit-Learn's `PolynomialFeatures` (we stop at the sixth degree). Then, we can simply perform a linear regression

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
poly_features = PolynomialFeatures(degree=6, include_bias=False)
norm_X_poly = poly_features.fit_transform(norm_X_train)
lin_reg = LinearRegression()
lin_reg.fit(norm_X_poly,norm_y_train) # perform linear regression
lin_reg.intercept_, lin_reg.coef_ # theta_0 and theta_k

(array([-0.35987146]),
 array([[ 0.20041976,    7.23777176,   -0.72575099,  -16.10685101,
        0.48763306,   8.95138004]]))
```

Not so difficult! Our model finds the following best-fit equation

$$\hat{y} = -0.36 + 0.20x + 7.24x^2 - 0.73x^3 - 16.11x^4 + 0.49x^5 + 8.95x^6.$$

We can now deploy it to make predictions and compare the latter with the test set

```
x = np.expand_dims(np.linspace(-np.pi,np.pi,200),1)
norm_x = scaling(x,transform=True,param=param_x)
norm_x_poly = poly_features.fit_transform(norm_x)
norm_y_predicted = lin_reg.predict(norm_x_poly)
```

Figure 12(left) compares the model's predictions with the training and test sets and below we evaluate the MSE of the predictions for the training and test sets.

```
from sklearn.metrics import mean_squared_error
norm_X_test_poly = poly_features.fit_transform(norm_X_test)
norm_y_predicted_train = lin_reg.predict(norm_X_poly)
norm_y_predicted_test = lin_reg.predict(norm_X_test_poly)
print('MSE on train set: ', mean_squared_error(norm_y_predicted_train,
                                                norm_y_train))
print('MSE on test set: ', mean_squared_error(norm_y_predicted_test,
                                                norm_y_test))

MSE on training data set:  0.05173368072737507
MSE on test data set:  0.05142192371233674
```

The model performs quite well on both sets.

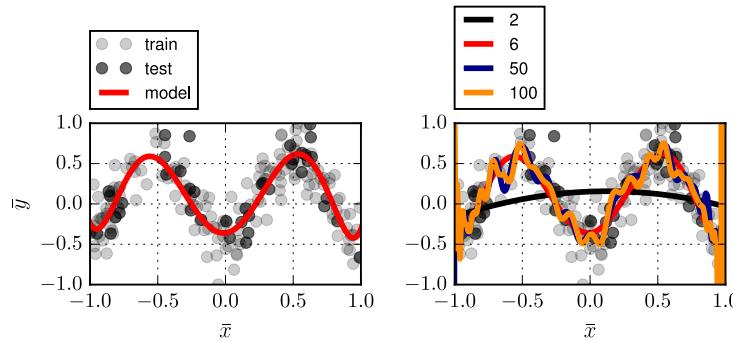


Figure 12: Nonlinear regression using polynomial basis functions with: (i) degree six (left) and (ii) different degrees (right). Features are scaled.

However, we skipped two important aspects: (i) hyperparameters selection and (ii) validation of the model.

5.1.1 Hyperparameters selection

The degree of the polynomial basis is actually a hyperparameter. Thus, we cannot a priori say that our analysis is completed – that is, whether or not we found the best model. To this end, we need to train multiple models with different degrees.

```

degree = [2,6,50,100] # different polynomial degrees
color = ['black','red','darkblue','darkorange']
fig = plt.figure(figsize=(3., 2.))
plt.plot(norm_X_train,norm_y_train,'ko',alpha=0.2)
plt.plot(norm_X_test,norm_y_test,'ko')
for i in range(len(degree)):
    poly_features = PolynomialFeatures(degree=degree[i], include_bias=False)
    norm_X_poly = poly_features.fit_transform(norm_X_train)
    lin_reg = LinearRegression() # linear regression model
    lin_reg.fit(norm_X_poly,norm_y_train) # fit model
    norm_x_poly = poly_features.fit_transform(norm_x)
    norm_y_predicted = lin_reg.predict(norm_x_poly)
    plt.plot(norm_x,norm_y_predicted,'-',color=color[i],
             linewidth=3,label=str(degree[i]))
plt.ylim(-1,1)
plt.xlim(-1,1)

```

The results are depicted in Figure 12 (right) and from them we can draw some conclusions: (i) a quadratic model is unable to fit our data and (ii) high-degree – 50 and 100, here – polynomial models wiggle around to get as close as possible to the training data points. This means the quadratic model is underfitting. Conversely, the higher the polynomial degree is, the more our model tends to overfit. So, how can we select the best models – that is, the one that will generalize best (cf. Section 4)? To answer we move to the second point of this example: model validation.

5.1.2 Validation

We have already seen in Section 4 that a crucial aspect in finding model that generalizes the best is to perform hold-out validation. To this end, we should assess the model’s performance by examining the validation error at varying of the size of the training set (and any eventual hyperparameter). By repeatedly training the model on subsets of varying sizes from the training set, we can generate informative plots that will guide us in selecting the model with the minimum error on the validation set. Below we implement a function that is designed to plot the training, validation, and test errors of models with different polynomial degrees (up to 40).

```

def learning_curves(X_train, y_train, X_test, y_test, degree):
    degree +=1
    X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                       test_size = 0.25) # split
    norm_X_val = scaling(X_val,transform=True,param=param_x) # scale
    model = LinearRegression()
    train_errors = np.zeros((degree-1,len(X_train)-1))
    val_errors = train_errors.copy()
    test_errors = train_errors.copy()
    for deg in range(1,degree):
        for m in range(1, len(X_train)):
            # Construct polynomial library up to degree = deg
            poly_features = PolynomialFeatures(degree=deg, include_bias=False)
            X_train_poly = poly_features.fit_transform(norm_X_train)
            X_val_poly = poly_features.fit_transform(norm_X_val)
            X_test_poly = poly_features.fit_transform(norm_X_test)

```

```

# Fit model
model.fit(X_train_poly[:m], y_train[:m])
# Predict training, validation, and test
y_train_predict = model.predict(X_train_poly[:m])
y_val_predict = model.predict(X_val_poly)
y_test_predict = model.predict(X_test_poly)
# Store errors
train_errors[deg-1,m-1] = mean_squared_error(y_train[:m],
                                              y_train_predict)
val_errors[deg-1,m-1] = mean_squared_error(y_val, y_val_predict)
test_errors[deg-1,m-1] = mean_squared_error(y_test, y_test_predict)
return train_errors, val_errors, test_errors
train, val, test = learning_curves(norm_X_train, norm_y_train,
                                    norm_X_test, norm_y_test, degree=40)

```

Let first have a look at the error on the training and validation sets for a sixth-degree polynomial model, depicted in Figure 13.

We start by analyzing the behavior of the training error. When there is a small bunch of training points, the model can perfectly fit those data and results in zero error. However, as more points are added, the model struggles to fit the data precisely due to the noise that we artificially added and the intrinsic non-linearity of the function we are trying to fit. Thus, the error on the training data increases until it reaches a plateau where additional data won't significantly affect the mean error.

An analysis on the validation error however rapidly reveals that things are quite different. When there is a small bunch of training points, the model clearly fails to generalize well and leads to relatively high validation errors. As the model is exposed to more training points, it gradually learns and improves, causing the validation error to decrease and, in this example, to reach approximately the same plateau of the training error.

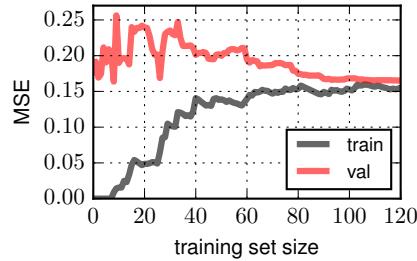


Figure 13: Training and validation mean squared errors at varying of the training set size for a sixth-degree polynomial model.

We are now ready to look at the big picture, namely Figure 14, where we can see the variation of the training, validation, and (optionally) test error at varying of both the polynomial degree and the size of the training set.

The training error presents the same aforementioned behavior. Additionally, we can

observe that for any polynomial degree smaller than 6, the error is high – that is, the model is underfitting the training data. Instead, as we increase the degree of the model, we observe a slight reduction of the error (caused by overfitting the training data).

If we move to the validation error, we observe that for training sets smaller than a certain size, the model does not generalize well (exactly as before). However, in addition, we (re-)discover overfitting. For degrees larger than 20, the model overfits the training data and cannot accurately predict any other data different from the training set (high validation error).

This shows the benefit of hold-out validation: the validation set allows us to pick up the best model(s) as the one(s) that presents the minimum validation error – approximately corresponding to a degree equal to 16, in this example. Once we have selected such model, we can deploy it to make prediction of new/unseen data – that is, the test set. And, indeed, the minimum error region for the test set is located within the region where the validation error is minimum (see Figure 14).

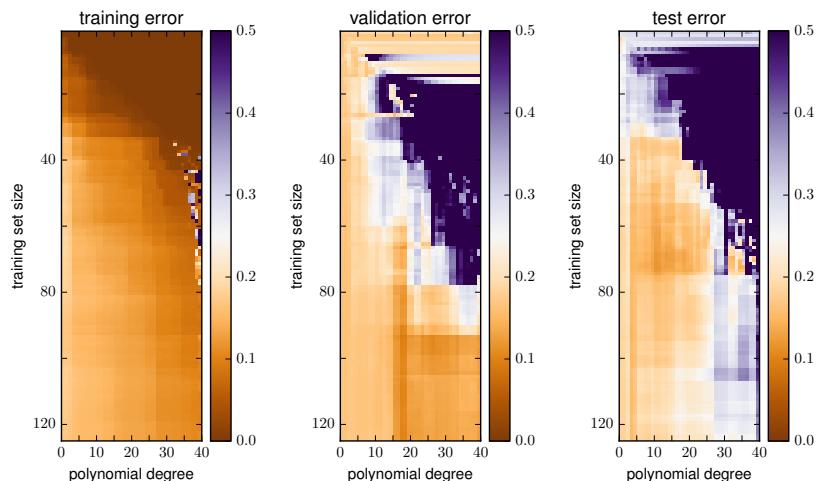


Figure 14: Training (top), validation (middle), and test (bottom) mean squared errors at varying of the training set size and the polynomial degree.

An alternative to the above laborious evaluation of the model’s performances at varying of the training set size consists of using Scikit-Learn’s k-fold cross-validation⁵. The code below, considering a polynomial regression of degree six, splits randomly the training set into five distinct folds, then it trains and evaluates the model five times, picking a different fold for evaluation every time and training on the other four folds. The result is an array containing the five evaluation scores:

⁵Note that `cross_val_score` expects an objective function to be maximized, so the scoring function is actually the opposite of the MSE.

```

from sklearn.model_selection import cross_val_score
poly_features = PolynomialFeatures(degree=6, include_bias=False)
X_train_poly = poly_features.fit_transform(norm_X_train)
scores = cross_val_score(lin_reg, X_train_poly, norm_y_train,
                        scoring="neg_mean_squared_error",
                        cv=5)
mse_scores = -scores
print('Scores: ', mse_scores)
print('Mean scores: ', np.mean(mse_scores))
print('Standard deviation scores: ', np.std(mse_scores))

Scores: [ 0.09159855  0.07245614  0.04068157  0.04406373  0.03980408]
Mean scores:  0.05772081499906291
Standard deviation scores:  0.0207975583120194

```

Our sixth-degree polynomial model has a MSE of approximately 0.057 ± 0.021 and we would have missed such information if we have just used one validation set. However, note that cross-validation carries some computational effort as it requires to train the model several times, and for much more complex methods (cf. chapter ChCL, ChANN, ChDR) may not be always a viable solution.

6 Regularization techniques

6.1 Over- and under-determined systems

In linear and nonlinear regression, most of time, we are confronted with a system of equations, either (4) or (17), that is under- or over-determined [Gen12]. To investigate these scenarios, let us rewrite Equation (10) as

$$\mathbf{A}\boldsymbol{\theta} = \mathbf{b}, \quad (19)$$

where $\mathbf{A} \equiv \mathbf{X}^T$ is a matrix with n rows and m columns and $\mathbf{b} \equiv \mathbf{y}^T$ is a matrix with n rows and p columns. Note that an analogous equation can also be formulated for nonlinear regression (13).

We define *overdetermined systems* as those systems that have more constraints (equations) than unknown variables – i.e., when the matrix \mathbf{A} is tall-skinny – while *under-determined systems* have more unknowns than constraints – i.e., \mathbf{A} is short-fat, see Figure 15.

Let us translate these notions from linear algebra to regression methods. If the number of data points is larger than the dimension of the independent variables ($n > m$), then we are dealing with an overdetermined systems of linear equations. Viceversa, if $n < m$, we are confronted with an underdetermined system. In the former case, there are generally no solutions satisfying the linear system, and instead, approximate solutions are found to minimize a given error. In the latter, there is an infinite number of solutions, and some additional constraints must be enforced in order to select an appropriate solution.

Note that in regression, we are most often dealing with overdetermined systems, even if sometimes overparametrization is preferred [BLLT20]. Analogous underdetermined

systems are instead met when dealing with overparametrized artificial neural networks (cf. chapter ChANN) [JNM⁺19].

Overdetermined or underdetermined optimization problems for linear systems (19) involve the minimization of the error of the solution, the MSE for instance, plus a constraint ($n < m$) or a penalty ($n > m$), which is also known as *regularization*, i.e.,

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(\text{MSE}(\mathbf{A}\boldsymbol{\theta} - \mathbf{b}) + \lambda w(\boldsymbol{\theta}) \right), \quad (20)$$

where λ a weighting parameter and $w(\boldsymbol{\theta})$ is a given function, depending whether the system is over- or under-determined. Below, we will see how ℓ_1 and ℓ_2 norm penalties/constraints can help in solving $\mathbf{A}\boldsymbol{\theta} = \mathbf{b}$. Spoiler alert: the choice of norm has a profound impact on the optimal solution found.

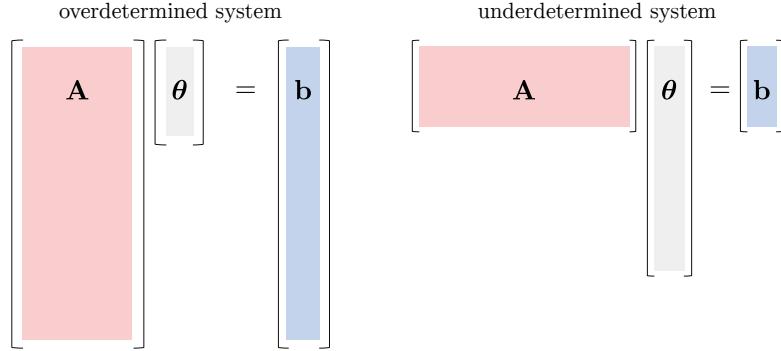


Figure 15: Overdetermined (left) and underdetermined (right) system.

6.1.1 Overdetermined systems

When dealing with overdetermined systems, there exists no solution that satisfy $\mathbf{A}\boldsymbol{\theta} = \mathbf{b}$. Thus, penalties based on the ℓ_1 and ℓ_2 norms are often considered and the optimization problem (20) reads

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(\text{MSE}(\mathbf{A}\boldsymbol{\theta} - \mathbf{b}) + \lambda_1 \|\boldsymbol{\theta}\|_1 + \lambda_2 \|\boldsymbol{\theta}\|_2 \right), \quad (21)$$

where $\|\boldsymbol{\theta}\|_r = (\|\boldsymbol{\theta}\|^r)^{1/r}$ denotes the ℓ_r norm and the parameters λ_1 and λ_2 control the penalization of the ℓ_1 and ℓ_2 norms, respectively. The above penalty is crucial for obtain optimal approximate solutions, which are directly affected by the value of the penalty parameters.

Let us consider an overdetermined system of equations. To fix the ideas, we draw \mathbf{A} and \mathbf{b} from normal distributions $\mathcal{N}[0, 1]$. We then obtain the parameters $\boldsymbol{\theta}$ by (i) using Equation (10) and (ii) solving the optimization problem (20), considering two

combinations of penalty parameters: $\{\lambda_1, \lambda_2\} = \{0.1, 0\}, \{0, 0.1\}$.

Figure 16 shows the obtained $\hat{\theta}$, as a $m \times p$ matrix. The key difference between the two regularization techniques is that ℓ_1 penalty shrinks the parameters associated with less important variables to zero. Indeed, regularization based on the ℓ_1 norm promotes a parsimonious solution, dominated by zero entries, i.e., sparse. While, regularization based on the ℓ_2 norm keeps the solution values as small as possible. We could also combine both kinds of regularization to promote both sparsity and decrease in the value of the parameters.

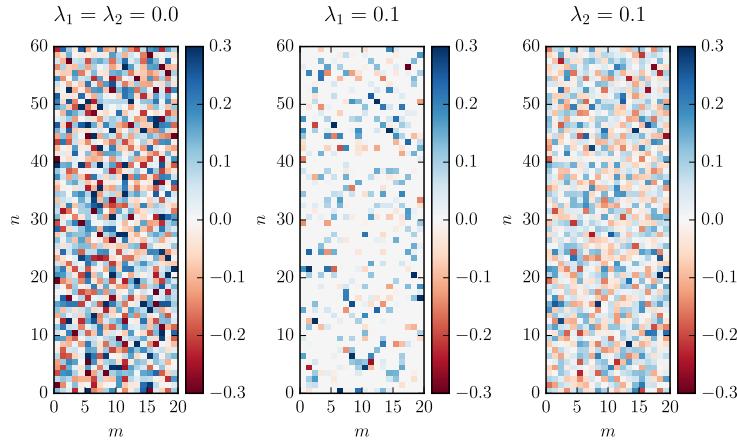


Figure 16: Overdetermined linear system of equations. Solutions obtained using the pseudoinverse (left), ℓ_1 (middle) and ℓ_2 (right) regularization.

6.1.2 Underdetermined systems

When dealing with undetermined systems, there exists an unlimited set of potential solutions that satisfy the equation $\mathbf{A}\theta = \mathbf{b}$. In such cases, the objective is to introduce constraints that would ideally result in a single, unique solution among the countless possibilities. Usually the optimization problem (20) is reformulated as

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left(\text{MSE}(\mathbf{A}\theta - \mathbf{b}) + \|\theta\|_r \right), \quad (22)$$

where the r denotes the r -norm. Once again, one could use either ℓ_1 or ℓ_2 norm, or combination of them, as for the overdetermined case.

6.2 Regularized regression

It is now time to focus on the reason why we discussed about over- and under-determined systems: regression!

As already mentioned, (non)linear regression deals, most of the time, with over-determined systems of equations – all examples we have looked up so far are indeed characterized by $n \gg m$. We should not be surprised, at this moment, that the same penalty terms that are used for the solution of overdetermined systems have their own counterparts (and names) in regression. Regularized regression models based on the ℓ_1 and ℓ_2 norms are called LASSO – Least Absolute Shrinkage and Selection Operator Regression –, and Ridge, respectively. The combination of both is often referred to as Elastic net.

6.2.1 LASSO

LASSO regression [Tib96] is the (non)linear regression version with a regularization term proportional to $\|\boldsymbol{\theta}\|_1$. This forces the learning algorithm to not only fit the data but also shrink the model parameters to zero. The loss function is thus defined as

$$\mathcal{L}^{\text{LASSO}}(\mathbf{X}, f_{\boldsymbol{\theta}}) = \text{MSE}(\mathbf{X}, f_{\boldsymbol{\theta}}) + \lambda_1 \|\boldsymbol{\theta}\|_1. \quad (23)$$

The above loss function is not differentiable at $\boldsymbol{\theta} = \mathbf{0}$, this means we cannot derive a closed form solution of the type (10). However, we can still use gradient descent provided that we define the following modified gradient, whenever $\boldsymbol{\theta}$ is equal to zero,

$$\text{grad}(\boldsymbol{\theta}, \mathcal{L}^{\text{LASSO}}) = \frac{\partial}{\partial \boldsymbol{\theta}} \text{MSE}(\mathbf{X}, f_{\boldsymbol{\theta}}) + \text{sign}(\boldsymbol{\theta}), \quad (24)$$

where $\text{sign}(\cdot)$ is the sign function.

Below, we show how to perform LASSO regression using Scikit-learn's implementation applied to the polynomial regression example (cf. Section 5), where we consider polynomials of degree 100:

```
from sklearn.linear_model import Lasso
lambda_1 = 0.0001
lasso_reg = Lasso(alpha=lambda_1, max_iter=100000) # define lambda_1
# Polynomial feature up to degree 100
poly_features = PolynomialFeatures(degree=100, include_bias=False)
lasso_reg.fit(norm_X_poly, norm_y_train) # fit LASSO reg model
norm_x_poly = poly_features.fit_transform(norm_x)
norm_y_predicted_lasso = lasso_reg.predict(norm_x_poly) # make predictions
```

The predictions are shown in Figure 17(left): quite powerful! For a clear comparison, the plot also shows the solution obtained without regularization (pinv). We basically rediscover the underlying function, Equation (18), used to generate the data, independently of the additive noise. The reason is that LASSO shrinks most of parameters to zero, hence we discover a quite precise series expansion of the combination of sinusoidal functions within the range $(-\pi, \pi)$.

For the sake of completeness, note that we could also use Scikit-learn's SGDRegressor with the following arguments: `penalty="l1"` and `alpha=lambda_1`:

```

from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=10000,
                      tol=1e-6,penalty="l1",alpha=lambda_1,
                      eta0=0.01)
sgd_reg.fit(norm_X_poly,norm_y_train.ravel())
norm_y_predicted_sgd = sgd_reg.predict(norm_X_poly) # make predictions

```

6.2.2 Ridge

Ridge regression [HK70] is the (non)linear regression version with a regularization term proportional to $\|\theta\|_2^2$ that forces the model parameters to be as small as possible. The loss function is defined as

$$\mathcal{L}^{\text{Ridge}}(\mathbf{X}, f_{\theta}) = \text{MSE}(\mathbf{X}, f_{\theta}) + \lambda_2 \|\theta\|_2^2. \quad (25)$$

As with linear regression – and differently from LASSO –, we can perform Ridge regression either by computing a closed-form equation [HK70] or by performing gradient descent. The code hereinafter implements the closed-form solution using Scikit-learn's Ridge:

```

from sklearn.linear_model import Ridge
lambda_2 = lambda_1
ridge_reg = Ridge(alpha=lambda_2,max_iter=100000) # define lambda_2
ridge_reg.fit(norm_X_poly,norm_y_train) # fit Ridge reg model
norm_y_predicted_ridge = ridge_reg.predict(norm_X_poly) # make predictions

```

We plot the predictions in Figure 17(middle). Once more, thanks to regularization, we are able to retrieve the original function, in contrast with the massively overfitted linear model (pinv).

6.2.3 Elastic net

Elastic net is the middle version between LASSO and Ridge. Accordingly, the regression model is regularized with respect to both the ℓ_1 and ℓ_2 norms. The regularization term is a mix of both Ridge and LASSO's controlled by a ratio ϱ :

$$\mathcal{L}^{\text{Elastic net}}(\mathbf{X}, f_{\theta}) = \text{MSE}(\mathbf{X}, f_{\theta}) + \varrho \lambda \|\theta\|_1 + \frac{1-\varrho}{\varrho} \lambda \|\theta\|_2^2. \quad (26)$$

Elastic net is equivalent to Ridge if $\varrho = 0$ and to LASSO if $\varrho = 1$. Below the code to define and fit an Elastic net model, whose predictions are shown in Figure 17(right):

```

from sklearn.linear_model import ElasticNet
net_reg = ElasticNet(alpha=2*lambda_2,l1_ratio=1.0,
                      max_iter=100000) # l1_ratio = \varrho
net_reg.fit(norm_X_poly,norm_y_train) # fit Elastic Net reg model
norm_y_predicted_net = net_reg.predict(norm_X_poly) # make predictions

```

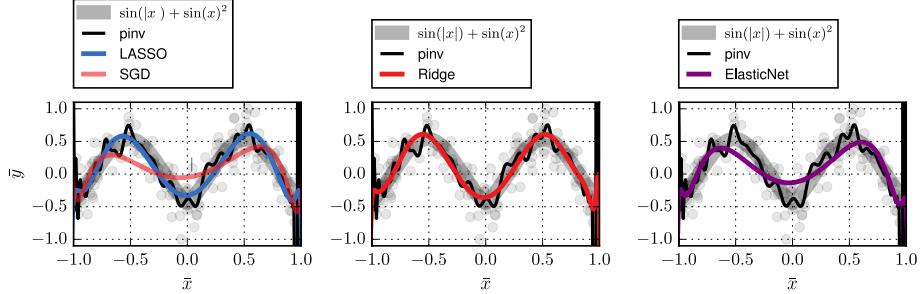


Figure 17: Nonlinear regression using polynomials of degree 100. Comparison between standard regression (pinv) – Eq. 10 – and regularized models: LASSO and stochastic GD (left), Ridge (middle), Elastic net (right). Features are scaled.

7 Challenges in generalization and extrapolation

Regression, like other supervised ML methods, is mainly designed for making predictions rather than determining existing relationships between some features. However, by post-processing the obtained model parameters θ , we can grasp some kind of rules buried under the mere data points we are analyzing. In this sense, we may even end up (re)discovering some laws of physics, in the form of governing equations, see e.g. [CLKB19]. To do so, the obtained model needs to be *interpretable* – that is, capable of explaining why data behaves in a certain manner. An interpretable model has, in general, not only good generalization capabilities, but also good expressive ability (extrapolation) [KB22].

Think to the example in Section 5 involving data generated from a combination of trigonometric functions with additive noise. The example highlighted that regression is actually more nuanced than simply choosing a regression model and performing a least-square fit. The selection of the model itself is crucial for achieving better predictions and interpretable descriptions of the data.

In the example, we have seen that if we pick up an “optimal” polynomial degree, then we will obtain a model able to interpolate quite well the test data (cf. Figure 12, left). However, if the polynomial degree is too large (or too small), we will end up having a model that massively overfits (or underfits) the training data – that is, unable to generalize for new data points (cf. Figure 12, right). The ultimate solution was to resort to regularized regression models (e.g., LASSO). In doing so, we succeeded in identifying a good approximation of the true (trigonometric) model hidden under the pile of noisy data (cf. Figure 17).

However, why a regularized regression model seems to have much more predictive power than a straightforward nonlinear regression one, with as many polynomial terms as we are able to count? The answer lies in Occam’s razor, a principle of parsimony (“*lex parsimoniae*”) attributed to William of Occam:

“Entia non sunt multiplicanda praeter necessitatem”

i.e. “entities must not be multiplied beyond necessity.” In other words, among different competing models that make the same predictions, the simpler one is the more likely. This principle is extensively used in science, where simpler explanations are preferred. An example is the method of dominant balance, which allows to determine the asymptotic behavior of solutions to an ordinary differential equation by identifying those terms that may be neglected in the limit. Dominant balance is, for instance, when we consider the evolution of shear stress in a sand sample, and we do not model neither its quantum state nor the effects of the relativistic warping of space-time caused by the grains.

Regularized regression adheres to this principle by discouraging the inclusion of numerous polynomial terms – or, in general, of features – thus promoting model simplicity. In addition, regularization helps in avoiding overfitting and results in high degree of stability. Indeed, high-dimensional polynomial models tend to suffer from instability due to multicollinearity that occurs when two or more independent variables have a high correlation with one another, see Figure 12(right).

While some of these benefits could also be obtained with low-degree polynomial models without the need to resort to regularization, these models tend to underfit the training data and exhibit insufficient expressive power. It is worth mentioning that successful machine learning models often exhibit a certain degree of *benign overfitting* (also called *overparametrization*) [BLLT20]. Therefore, regularized regression models, with their ability to balance complexity and simplicity, offer significant predictive power by selecting relevant features.

In the example involving LASSO, Ridge, and Elastic Net regression (paragraph 6.2), we demonstrated the interpolating power of regularized regression models. However, it is important to further assess their interpretability and potential to provide insights for the identification and/or discovery of patterns and equations hidden within data sets.

7.1 Interpretable models and *where* to find them

This application deals with the investigation of whether it is possible or not to discover interpretable models (governing equations) with extrapolative power using (regularized) regression methods. In particular, we mimic virtual experiments governed by a very simple equation and consider additive noise simulating virtual statistical errors in a hypothetical data acquisition system.

The underlying governing equation is the (one-dimensional) projectile motion – i.e., the solution of the second-order ordinary differential equation $\ddot{x}(t) + g = 0$,

$$x(t) = x(0) + \dot{x}(0)t - \frac{1}{2}gt^2,$$

where x , \dot{x} , and \ddot{x} are the trajectory, velocity, and acceleration of the projectile, respectively; t is the time, and g is the gravitational acceleration. The code hereinafter

integrate the differential equation with initial conditions: $x(0) = 0$ and $\dot{x} = 5$. Being interested in making predictions within the range of values of some training data and outside of it, we generate the data and split them in an interpolation (xi and yi) and extrapolation (xe and ye) sets, depicted in Figure 18(a):

```
import numpy as np
from scipy.integrate import solve_ivp
np.random.seed(42)

def projectile_motion(t, y):
    # Equations of motion
    dydt = y[1] # velocity
    dvdt = -g # acceleration
    return [dydt, dvdt]
# Initial conditions
n_snapshots = 200
x0 = 0.0 # meters
v0 = 5.0 # meters per second
g = 9.81 # acceleration due to gravity
t0 = 0.0 # initial t
tf = 1.0 # final t
# time points solution
t = np.linspace(t0, tf, n_snapshots)
# solve ODE
sol = solve_ivp(projectile_motion, [t0, tf], [x0, v0], t_eval=t)
# define independent and dependent variables
X = np.float32(t)
y = np.float32(sol.y[0])
n = n_snapshots//2
xi = X[:n] # train time
xe = X[n:2*n] # test time
ytrain = y[:n] # train y = [0,0.5]
ytest = y[n:2*n] # test y = [0.5,1]
```

We continue by training a set of n polynomial regression models of degree 20. Every model is trained against dependent variables lying between 0 and 1 (xi) and dependent variables with a normally distributed measurement noise, randomly selected at each loop (fni). In a nested for loop, we store the MSE values between the models predictions and the quadratic function, without measurement noise, for values of the independent variables in the interpolation (mse_int) and extrapolation (mse_ext) range.

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
degree = 20 # polynomial
mse_int = np.zeros((n,degree))
mse_ext = np.zeros((n,degree))
poly_features = PolynomialFeatures(degree=degree, include_bias=False)
xi_poly = poly_features.fit_transform(np.expand_dims(xi,1))
xe_poly = poly_features.fit_transform(np.expand_dims(xe,1))
for deg in range(degree):
    xi_poly_ = xi_poly[:, :deg+1]
    xe_poly_ = xe_poly[:, :deg+1]
```

```

for j in range(n):
    fni = yi + 0.3*np.random.normal(0,1,n) # y with normal noise
    lin_reg = LinearRegression() # model
    lin_reg.fit(xi_poly_,fni) # fit for t in [0,0.5]
    ynai = lin_reg.predict(xi_poly_)
    ynae = lin_reg.predict(xe_poly_) # fit for t in [0.5,1]
    mse_int[j,deg] = mean_squared_error(ytrain,ynai) # MSE train
    mse_ext[j,deg] = mean_squared_error(yval,ynae) # MSE val

```

Figure 18 shows the MSE of the set of model in the interpolation and extrapolation ranges. The results are clear: our set of models is massively overfitting the data with measurement noise, see Figure 18(b), and cannot generalize. When attempting to extrapolate beyond the range observed in the training data, we are confronted with substantial errors, as shown in Figure 18(c-d). The logarithmic plot in Figure 18(d) illustrates the exponential growth of the error as the polynomial degree increases, reaching magnitudes as high as 10^6 . This demonstrates the inability of overfitted models to make predictions in the extrapolation range. Thus, we can say that (non-regularized) polynomial regression does not offer an interpretable model for this simple case (except an obvious second-order polynomial).

Let us continue by employing cross-validation to mitigate the effects of overfitting. We extend our set of polynomial regression models – denoted with pinv, diminutive of pseudoinverse – with regularized ones. The code below performs 100-fold cross-validation for the same noisy data sets for the three different regression models (pinv, LASSO, and Ridge) and stores the obtained parameters. Note that, as the measurement noise is randomly select for each k fold, the training data points may be considered as virtual experimental measurements of our simplified system.

```

from sklearn.linear_model import Lasso,Ridge
from sklearn.model_selection import KFold
nfolds = 100
lin_reg = LinearRegression() # Linear regression (pinv) model
lasso = Lasso(alpha=3e-4) # Lasso regression model
ridge = Ridge(alpha=3e-4) # Ridge regression model
k_fold = KFold(nfolds) # k-fold cross-validation
# Initialize params
lasso_params = np.zeros((nfolds,degree+1))
ridge_params = np.zeros((nfolds,degree+1))
lin_params = np.zeros((nfolds,degree+1))
for k, (train, val) in enumerate(k_fold.split(xi_poly, ytrain)):
    fni = yi + 0.3*np.random.normal(0,1,n) # y with normal noise
    lin_reg.fit(xi_poly[train], fni[train]) # fit pinv
    ridge.fit(xi_poly[train], fni[train]) # fit Ridge
    lasso.fit(xi_poly[train], fni[train]) # fit Lasso
    # Store parameters for each model
    lin_params[k,0] = lin_reg.intercept_
    lin_params[k,1:] = lin_reg.coef_
    ridge_params[k,0] = ridge.intercept_
    ridge_params[k,1:] = ridge.coef_
    lasso_params[k,0] = lasso.intercept_
    lasso_params[k,1:] = lasso.coef_

```

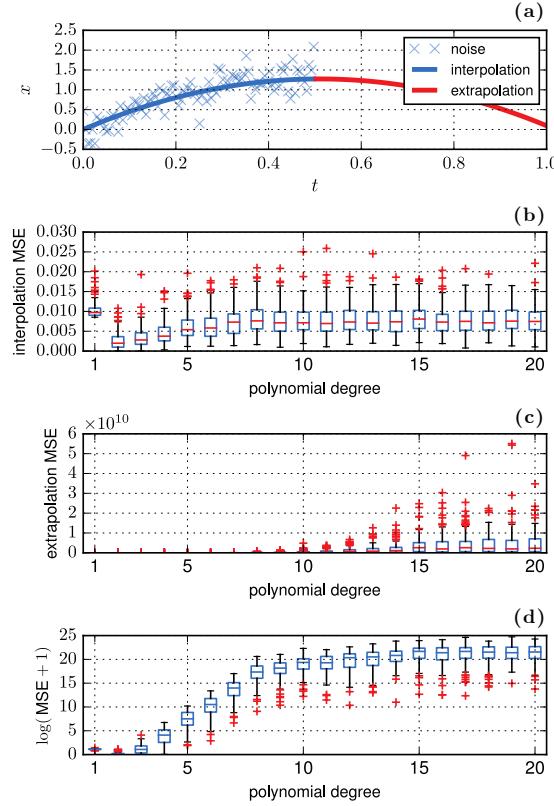


Figure 18: Regression of a quadratic function with measurement noise (a) using 100 different nonlinear regression models with polynomial basis (degree 20). Errors for values within the training, interpolation, range (b), and outside, extrapolation, (c-d).

Figure 19 plots the values of the parameters θ obtained from the above code, for k -th fold ($k = 2, 10$, and 100). Note how the parameters of the standard regression model explode, with values of the order of 150^3 , independently of the number of folds. The phenomenon is intrinsic to polynomial regression which blows up as the polynomial degree is increased and further enhanced by multicollinearity of the (polynomial) features. In contrast, Ridge and LASSO identify parameters with range of values comparable with underlying equation of motion. In addition, LASSO also successfully identifies a parsimonious model, with a (more or less) predominant quadratic term (depending on the number of folds).

In order to proceed to the evaluation of the three cross-validated models, we compute the average of the parameters obtained from the optimization performed at each fold (cf. paragraph 5.1.2):

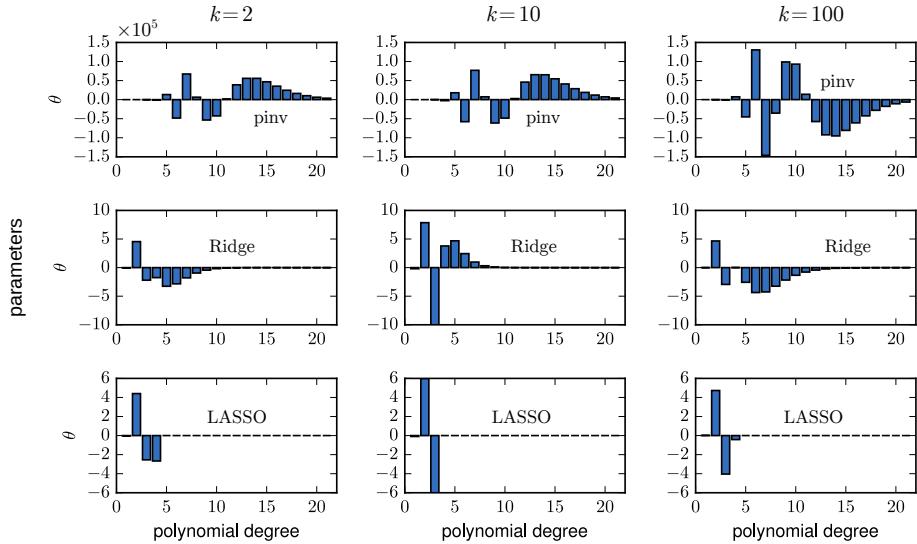


Figure 19: Values of the parameters θ obtained using standard (pinv), LASSO, and Ridge polynomial regression for different cross-validation folds: $k = 2$ (left), $k = 10$ (middle) and $k = 100$ (right).

```

# Compute models based on the mean over the 100-fold cross-validation
lin_theta = np.mean(lin_params[:, :, 0])
lasso_theta = np.mean(lasso_params[:, :, 0])
ridge_theta = np.mean(ridge_params[:, :, 0])
# Set mean params for each model
lin_reg.coef_ = lin_theta[1:]
lin_reg.intercept_ = lin_theta[0]
ridge.coef_ = ridge_theta[1:]
ridge.intercept_ = ridge_theta[0]
lasso.coef_ = lasso_theta[1:]
lasso.intercept_ = lasso_theta[0]
# Predictions within interpolation range
fy_lin_xi = lin_reg.predict(xi_poly)
fy_la_xi = lasso.predict(xi_poly)
fy_rid_xi = ridge.predict(xi_poly)
# Predictions within extrapolation range
fy_la_xe = lasso.predict(xe_poly)
fy_lin_xe = lin_reg.predict(xe_poly)
fy_rid_xe = ridge.predict(xe_poly)

```

Figure 20 compares the performances of the three models (pinv, LASSO, Ridge) in terms of the mean squared error, while Figure 21 shows the predictions, in the interpolation and extrapolation range. As we might have expected, LASSO excels over all other strategies by providing a model that can interpolate (without overfitting) and, even, extrapolate with good accuracy. We can also observe that Ridge performs better than the standard regression approach, although it rapidly fails in the extrapolation

range.

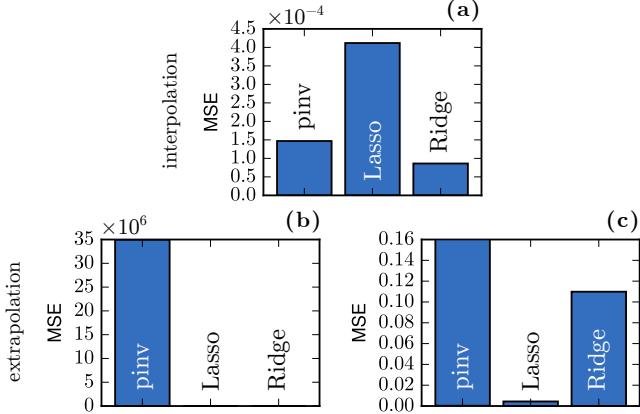


Figure 20: Errors of the cross-validated regression models: pinv, LASSO, Ridge within the: (a) interpolation range and (b-c) extrapolation range, where (c) presents a detailed view.

Whilst good generalization and extrapolation capabilities, LASSO has not identified the exact underlying governing equation but just a very good approximation of it,

$$x(t) \approx \theta_1 t + \theta_2 t^2 + \theta_3 t^3, \quad \theta_1 = 4.33, \quad \theta_2 = -2.93, \quad \theta_3 = 1.32.$$

Note that the true equation has $\theta_1 = 5$, $\theta_2 = -4.9$, and $\theta_3 = 0$. This implies that when we ask the model to make predictions for a significantly distant time point, i.e., $t \gg 1$, the accuracy of the extrapolation may be compromised.

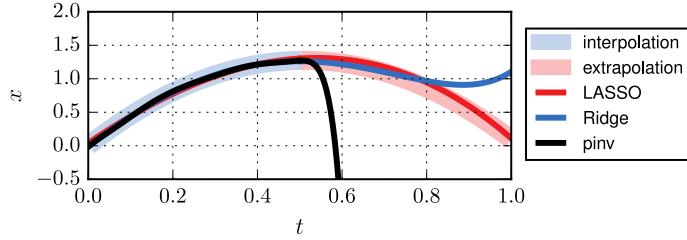


Figure 21: Comparison between the projectile motion equation and the predictions of the cross-validated models: pinv, LASSO and Ridge.

8 Bayesian regression

Up to this point, we have seen linear and nonlinear regression, with or without regularization. All these methods fall under the hat of *frequentist approaches* [Wak13] as

they assume that there are enough measurements to say something meaningful about θ and provide a single best estimate for a given training set. Such methods provide good models whenever the relationship between the independent and dependent variables is well understood and relatively stable. However, in real case scenarios, we are often confronted with uncertainty or variability in the data and deterministic methods are not appropriate. To this end, a powerful, alternative tool is provided by Bayesian regression methods.

Bayesian regression is able to provide predictions in the form of probability distributions incorporating uncertainty estimates. In this Section we set the basis for the understanding of such methodologies by focusing on two approaches: linear Bayesian and Gaussian process regression. For more, we refer to [BN06, Mur18].

8.1 Linear Bayesian Regression

In simple linear regression, we aim to find the best-fit line that explains the relationship between two variables. Bayesian Linear Regression extends this same concept by considering uncertainty in both the model parameters and the predictions. Instead of relying on a single line (cf. Section 2), it thus provides a distribution of possible lines, accounting for the uncertainty in the data.

Linear Bayesian regression starts by assuming a linear relationship between the independent variables \mathbf{X} and the dependent variable \mathbf{y}

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\epsilon}, \quad (27)$$

where $\boldsymbol{\theta}$ are the model parameters and $\boldsymbol{\epsilon}$ represents an error term assumed to follow a zero-mean Gaussian distribution, $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$, where \mathbf{I} is the identity matrix and σ^2 is the variance.

8.1.1 Bayesian inference

Bayesian inference allows to infer a posterior distribution $p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y})$ – in our case, that of the parameters – representing updated beliefs given the observed independent variables \mathbf{X} and dependent variable \mathbf{y} . To this end, we use Bayes' formula, i.e.,

$$p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y}) \propto p(\mathbf{X}, \mathbf{y}|\boldsymbol{\theta}) p(\boldsymbol{\theta}), \quad (28)$$

where $p(\boldsymbol{\theta})$ is the prior distribution over $\boldsymbol{\theta}$, incorporating prior beliefs about the parameters values before observing any data, while $p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$ is the likelihood function, quantifying instead the probability of observing the dependent variable \mathbf{y} given the independent variables \mathbf{X} and the parameters $\boldsymbol{\theta}$.

8.1.2 Prior and posterior

In principle, any prior $p(\boldsymbol{\theta})$ could be used, however the functional form of most priors, when multiplied by the functional form of the likelihood in (28), results in an posterior

$p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y})$ with no closed-form solution.

In such scenarios, the solution is to resort to approximate Bayesian inference techniques as Monte Carlo methods. But, certain priors are mathematically convenient because they result in posteriors with tractable, well-known densities.

The simplest and most widely used version of linear Bayesian regression is the *normal linear model*, in which \mathbf{y} given $\mathbf{X} - p(\mathbf{X}, \mathbf{y}|\boldsymbol{\theta})$ – is distributed Gaussian. In this model, and under a particular choice of the prior for the parameters – namely, when the prior is conjugate, meaning that the prior and posterior share the same functional form – the posterior can be found analytically. With more arbitrarily chosen priors, the posteriors generally have to be approximated.

8.1.3 Posterior predictive

Once we have the posterior distribution $p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y})$, we can make predictions for new, unseen data points.

One common approach is to generate posterior predictive samples by drawing parameter values from the posterior distribution and using them to make predictions [BN06]. To make predictions, we draw a posterior predictive distribution $p(\mathbf{y}_*|\mathbf{y})$, distribution over unobserved data \mathbf{y}_* .

8.1.4 Example

Let us consider an intuitive example drawn from [BN06] and implemented as in <https://github.com/gwgundersen/bayesian-linear-regression>. We start by generating 50 snapshots of the independent and dependent variables, according to

$$\mathbf{y} = \theta_1 \mathbf{x} + \theta_0,$$

where $\theta_1 = 0.5$ and the bias is $\theta_0 = -0.7$. Then, we perform Bayesian linear regression using as prior a normal-inverse-gamma distribution and consider an increasing number of observations in the training data set.

Figure 22 depicts the evolution of the prior and posterior distributions of the parameters $\boldsymbol{\theta}$ (left column) and six random samples of the vector $\boldsymbol{\theta} = [\theta_0, \theta_1]^T$ (right column). In the top row, the model has seen no data. The prior places high and equal probability on both θ_0 and θ_1 being zero. In subsequent rows, the model is fit to more data. With more observations, the model inferred posterior variance decreases, and the realizations of $\boldsymbol{\theta}$ from the posterior become more constrained.

8.2 Gaussian process regression

Gaussian process regression is a *nonparametric, Bayesian* approach to regression. Before continuing, let us decrypt this by briefly recalling and defining what nonparametric and Gaussian process mean.

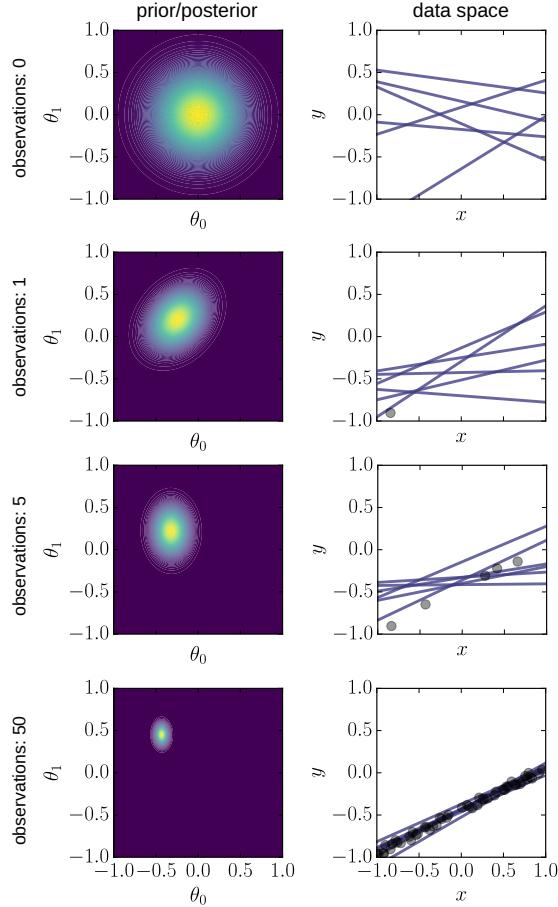


Figure 22: Evolution of the prior and posterior distributions of the parameters θ (left column) and random samples of the vector $\theta = [\theta_0, \theta_1]^T$ (right column) at varying of the number of observation (rows).

Parametric versus nonparametric model Parametric models assume that the distribution of the data can be entirely defined in terms of a finite set of parameters, θ . Thus, once we have trained a parametric model, every future prediction is independent of the particular set of new data for which we are making predictions. It should be evident that examples of parametric models are those models obtained with all regression methods we have seen so far – linear Bayesian regression included.

Nonparametric models do not prescribe any predetermined functional relationship (or hypothesis f_θ) between the dependent and independent variables [HM93]. Nonparametric models thus have the freedom to calculate the probability distribution over all admissible functions that fit the data, rather than, for instance, calculating the probabil-

ity distribution of parameters of a specific function. However, similar to the above, we must specify a prior (on the function space), calculate the posterior using the training data, and compute the predictive posterior distribution on our points of interest.

Gaussian process A Gaussian process (GP) is a collection of random variables, any finite number of which have a joint Gaussian distribution.

In the context of regression, we can think of a GP as an infinite-dimensional generalization of a multivariate Gaussian distribution. Instead of representing a mean vector and a covariance matrix, a GP is fully characterized by a mean function, denoted as $\mu(\mathbf{x})$, and a covariance function, also known as the kernel function, denoted as $k(\mathbf{x}, \mathbf{x})$, describing the relationship between any given two data points \mathbf{x}_i and \mathbf{x}_j , for $i, j = 1, 2, \dots, m$.

8.2.1 Inference

Gaussian process regression (GPR) is a *nonparametric, Bayesian* approach to regression (and, after the above digression, this should now be clear!). In particular, GPR uses a Gaussian process prior to infer the distribution of possible functions that could generate the observed data.

Given a set of training data $\mathcal{D} = (\mathbf{X}, \mathbf{y}) \equiv \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, the idea is to estimate the underlying function $f(\cdot)$ by placing a Gaussian process over $f(\mathbf{X})$. Typically, a mean of zero is assumed, leading to the following distribution

$$\mathbf{y} = f(\mathbf{X}) \sim \mathcal{N}(\mathbf{0}, \mathbf{K}), \quad (29)$$

where $\mathbf{K} \equiv k(\mathbf{X}, \mathbf{X})$ denotes the covariance matrix generated by the kernel function k chosen (e.g. periodic, linear, radial basis function) and describes the general shape of the function. Note that the choice of the kernel is crucial in identifying a good (or a bad) fit of the data.

To predict the values of a new set of dependent variables \mathbf{y}_* , given a new set of independent variables \mathbf{X}_* , we need to estimate the conditional distribution $p(\mathbf{y}_* | \mathbf{X}_*, \mathcal{D})$. According to the properties of GP [Mur18], the joint distribution of the training and test outputs follows a joint Gaussian distribution ()

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{y}_* \end{bmatrix} = f \left(\begin{bmatrix} \mathbf{x} \\ \mathbf{x}_* \end{bmatrix} \right) \sim \mathcal{N} \left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^\top & \mathbf{K}_{**} \end{bmatrix} \right), \quad (30)$$

where $\mathbf{K}_* \equiv k(\mathbf{X}, \mathbf{x}_*)$, $\mathbf{K}_{**} \equiv k(\mathbf{x}_*, \mathbf{x}_*)$. While the above joint distribution gives some insight as to how $f(\mathbf{x}_*)$ relates to $f(\mathbf{x})$, at this point no prediction for the new datum \mathbf{x}_* .

To obtain the posterior distribution of the predicted GP realizations $f(\mathbf{x}_*)$, we condition the prior distribution on the training data [Mur18]. This leads to the following mean and covariance predictions for the test point \mathbf{x}_* (see [Mur18] for the detailed derivation):

$$\begin{aligned} \mu(f(\mathbf{x}_*)) &= \mathbf{K}_*^\top \mathbf{K}^{-1} \mathbf{y}, \\ \text{Cov}(f(\mathbf{x}_*)) &= \mathbf{K}_{**} - \mathbf{K}_* \mathbf{K}^{-1} \mathbf{K}_*^\top. \end{aligned} \quad (31)$$

These equations provide the mean and covariance predictions for the output $f(\mathbf{x}_*)$ given the training data set and a new input \mathbf{x}_* .

8.2.2 Inference in the presence of noise

In real-world scenarios, we often encounter observations with measurement noise. In such scenarios, it is more appropriate to model the training targets \mathbf{y} to be noisy realizations of a Gaussian process $f(\mathbf{X})$,

$$\mathbf{y} = f(\mathbf{X}) \sim \mathcal{N}(\mathbf{0}, \mathbf{K}) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2), \quad (32)$$

where the noise ϵ is parameterized by a zero-mean Gaussian with positive noise covariance values given by σ^2 , which is a hyperparameter.

By repeating the steps above, we obtain the following joint distribution of the training and new outputs

$$f\left(\begin{bmatrix} \mathbf{x} \\ \mathbf{x}_* \end{bmatrix}\right) \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} + \sigma^2 \mathbf{I} & \mathbf{K}_* \\ \mathbf{K}_*^\top & \mathbf{K}_{**} \end{bmatrix}\right). \quad (33)$$

As before, but now with the noise term, we can obtain the predictive mean and variance

$$\begin{aligned} \mu(f(\mathbf{x}_*)) &= \mathbf{K}_*^\top (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{y}, \\ \text{Cov}(f(\mathbf{x}_*)) &= \mathbf{K}_{**} - \mathbf{K}_* (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{K}_*^\top + \sigma^2. \end{aligned} \quad (34)$$

8.2.3 Example

Let us see how we can perform GRP using Scikit-learn. We consider the same polynomial regression example in Section 5, with data generated from a combination of trigonometric functions with random additive noise. In the code hereinafter, we first fit a Gaussian process on the training data using a radial basis function (RBF) kernel and, additionally, a noise parameter (`alpha`). Then, we use the kernel to compute the mean predictions (`norm_y_mean`) for the train and test set and plot the 95% confidence interval (defined as `norm_y_mean ± 1.96 norm_y_std`).

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
# define a radial basis function kernel
kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(0.001, 10)) #
GPR = GaussianProcessRegressor(kernel=kernel,
                               alpha=0.5**2) # noise term
GPR.fit(norm_X_train, norm_y_train) # find mean and covariance matrix
# posterior distribution
norm_y_mean, norm_y_std = GPR.predict(norm_x, return_std=True) # see par. 5.1
```

The predictions are depicted in Figure 23 and compared with the training and test data. Not only the fit in terms of the expected (mean) value is extremely good, but in

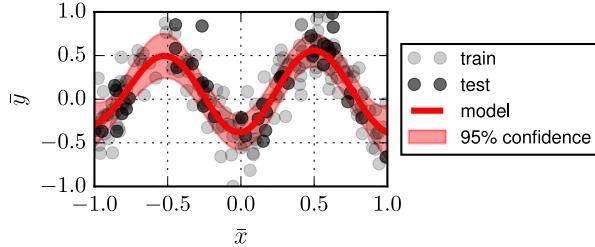


Figure 23: Gaussian process regression (with noise) for data generated from a combination of trigonometric functions with random additive noise, see paragraph 5.1: predictions and 95% confidence interval.

addition, due to the intrinsic probabilistic nature of GPR, we are also able to compute confidence intervals and thus quantify uncertainties.

In this Section, we have seen that linear Bayesian and Gaussian process regression offer unique advantages as compared to deterministic methods as far it concerns uncertainty estimation and Bayesian inference. However, we should also address some of the main drawbacks associated with such methods. First, both linear Bayesian regression and Gaussian process regression have high computational complexity – higher than that of deterministic methods – and may be computationally expensive, especially when dealing with large data sets. Second, we have seen that deterministic regression methods allow to identify interpretable model that directly relate to the impact of each parameter on the dependent variable predictions – especially in the case of regularized strategies, cf. Section 7. However, this is not usually true for linear Bayesian and Gaussian process regression: indeed, the parameters are expressed in the form of posterior distributions or covariance matrices and their interpretation can be quite challenging.

9 Conclusions

This chapter provided a comprehensive introduction to regression methods, which serve as one of the foundational pillars of Machine Learning, together with classification methods (cf. chapter ChCL).

We covered linear and nonlinear regression models and their Bayesian counterparts in Sections 2, 5, and 8. Together, this kind of supervised ML approaches allow to identify relationships between inputs and outputs and offer predictive models. The starting point consists of defining a hypothesis – in linear and nonlinear regression – or a prior distribution and some kernel – in Gaussian process regression. Then, the process continues with the identification of the hypothesis parameters – in the form of fixed values or posterior distributions – or functionals, obtained through optimization strategies (gradient descent) and/or as closed-form solutions, to finally obtain a

predictive model by relying on a training data set.

We also introduced concepts and strategies related to model validation and generalization, as well as features scaling (cf. Section 4). These are extremely important and useful not only for regression methods but, in general, for any Machine Learning approach – e.g., classification methods, artificial neural networks, and dimensionality reduction techniques.

Then, we emphasized the significance of regularization, particularly in the context of nonlinear regression (cf. Section 6). Regularization techniques play a vital role in preventing overfitting, where the model becomes too complex and fits noise or irrelevant patterns, leading to poor generalization. Note that regularization strategies extend well beyond regression models and are a good ally for enhancing the performance and generalization abilities of Machine Learning models.

Additionally, regularization strategies facilitate the identification of interpretable models by promoting simplicity and a small number of dominant parameters. Interpretable models have the potential to reveal hidden physical relationships or equations that may be obscured within vast amounts of data (cf. Section 7).

However, we should pay attention in understanding the limitations of (non) regularized regression models when addressing complex phenomena. For instance, if we were asked to predict occurrences and magnitudes of earthquakes and landslides or characterize the constitutive behavior of intricate materials like sand or clay, the interpretable, but simplistic, approach developed in the case of the projectile motion (cf. paragraph 7.1) will not be effective. The aforementioned scenarios pose, indeed, challenges due to the intrinsic chaotic nature and a high-dimensional state space. In such scenarios, achieving reliable generalization and extrapolation performance becomes increasingly arduous when relying solely on Machine Learning approaches that lack any knowledge bias – even in the case of sophisticated neural networks, cf. chapter ChANN. It becomes thus imperative to resort to physics-informed and thermodynamics-based machine learning models (as explored in chapters ChANN and ChTANN) or to leverage data-driven computing approaches (presented in chapter ChDD). These alternative methodologies incorporate domain knowledge and physical principles into the learning process, resulting in more robust and accurate predictions.

References

- [BLB⁺13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

- [BLLT20] Peter L Bartlett, Philip M Long, Gábor Lugosi, and Alexander Tsigler. Benign overfitting in linear regression. *Proceedings of the National Academy of Sciences*, 117(48):30063–30070, 2020.
- [BN06] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.
- [BV04] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- [CLKB19] Kathleen Champion, Bethany Lusch, J Nathan Kutz, and Steven L Brunton. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, 116(45):22445–22451, 2019.
- [Gen12] James E Gentle. *Numerical linear algebra for applications in statistics*. Springer Science & Business Media, 2012.
- [Gér22] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. ”O’Reilly Media, Inc.”, 2022.
- [HK70] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.
- [HM93] Wolfgang Händle and Enno Mammen. Comparing nonparametric versus parametric regression fits. *The Annals of Statistics*, pages 1926–1947, 1993.
- [JNM⁺19] Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. Fantastic generalization measures and where to find them. *arXiv preprint arXiv:1912.02178*, 2019.
- [KB22] J Nathan Kutz and Steven L Brunton. Parsimony as the ultimate regularizer for physics-informed machine learning. *Nonlinear Dynamics*, 107(3):1801–1817, 2022.
- [Man82] John Mandel. Use of the singular value decomposition in regression analysis. *The American Statistician*, 36(1):15–24, 1982.
- [Mur18] Kevin P Murphy. Machine learning: A probabilistic perspective (adaptive computation and machine learning series), 2018.
- [Tib96] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.
- [Wak13] Jon Wakefield. *Bayesian and frequentist regression methods*, volume 23. Springer, 2013.
- [WT] Torsten Wichtmann and Theodoros Triantafyllidis. An experimental database for the development, calibration and verification of constitutive

models for sand with focus to cyclic loading: part I—tests with monotonic loading and stress cycles.