Project 1 on Computational Physics and Machine Learning

I. Poquet[1,2]

[1] Institute for Theoretical Astrophysics, University of Oslo
e-mail: ijpoquet@uio.no
[2] Rosseland Center for Solar Physics

ABSTRACT

Machine learning models are widely used today to solve problems where we need to fit the data. The most basic models are the so-called Linear Regressions, where the models are able to reproduce the observed data with very good accuracy. There are several ways to train and compute these Linear Regressions: by directly inverting matrices or by using different types of gradient descent methods. In addition, these methods are used to solve Logistic Regressions, another type of simple regressions to solve binary classification problems. In this paper we study and analyse the main methods used to compute such regressions.

1. Introduction

Artificial Intelligence is a computation field with the main goal of develop software able to solve tasks that are easy for people to perform but hard for people to describe formally. To do this, Artifical Intelligence needs of systems with the ability to acquire their own knowledge, by extracting patterns from war data. This capability is known as Machine Learning (ML) and consists on that the models used automatically adapt to the problem, learning from it. The most popular techniques today are those involving Deep Learning and very complicated Neural networks. However, there are much simpler ML techniques that can perfectly solve our problem with a much lower computational cost. Moreover, these techniques are the prelude to today's modern ML. It is therefore essential to understand them. When they became insufficient, NN and Deep Learning models were developed in response to the need to tackle more complicated problems.

One of these basic techniques are the so-called *Regression methods* where, using very simple concepts, it is possible to create a model that adapts to the problem. One of the best known examples is the *Least squares adjustment*, used to fit a model as best as possible to the data available. Just to clarify, this technique, like many others, is normally included in the field of numerical optimisation. ML is a label that is fast becoming in a field that involves all the techniques that correct themselves to solve better the problem.

It is essential to know the limits and what our models are capable of. The optimal solution to a problem is always the simplest solution, and we can only achieve it by knowing the techniques we use. Is for this that in this project we study the following kinds of Least Squares Regression methods: Ordinary Least Squares, Ridge Regression, Lasso Regression and Logistic Regression. Also, we will go through different ways to solve the regressions, using both direct methods and *Gradient Descent* methods.

In the second section, we introduce the formalism of the Least Squares Regression, followed by the formalism of the different kinds of regression and the study of resampling techniques in sections 3 and 4 respectively. In the section 5, gradient descent methods will be introduced and applied to solve the regressions. Section 6 explains the formalism for the Logistic regression and its application over a practical case. Eventually, section 7 will show the conclusions on the whole project and section 8 will explain the code used.

2. Least Squares Regression

Consider a set of D pairs,

$$D = \{(x_0, y_0), (x_1, y_1), ..., (x_{N-1}, y_{N-1})\}, \tag{1}$$

where x_i and y_i are inputs and outputs respectively. The regression methods aim is to infer an optimal model $\hat{y} = \hat{y}(x)$ that reproduces as best as possible the outputs y_i given the inputs x_i. To achieve this, we make the assume that the pairs of D follow a function:

$$y = y(x) = f(x) + \epsilon, \tag{2}$$

where ϵ denotes noise $\sim N(0, \sigma^2)$. We assume that $f(x)$ is analytic and a deterministic continuous function, which means that is not stochastic. Without this assumption, we could not fit $f(x)$ with a linear model because it would have a quasi-random behavior. Now we propose the model:

$$\hat{y}_i = \hat{y}(x_i) \simeq f(x_i) \cong y(x_i) = y_i,$$

and assume that it is polynomial. Hence, we can express it as a power series:

$$\hat{y}_i = \sum_{j=0}^{p-1} \beta_j x_i^j = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + ... + \beta_{p-1} x_i^{p-1}. \tag{3}$$

We call each polynomial degree a feature. With this model, we want to describe each output as a polynomial model of each input. Value p denotes the complexity/degree of the model. The higher the degree, the greater the complexity the model can acquire. Realize that, without the assumptions over (2), in particular the analytic assumption, we could not do the polynomial expansion either.

In the case of having multiple inputs and outputs, as in D, we can rewrite $\hat{y}$ as:

$$\begin{bmatrix} \tilde{y}_0 = \beta_0 + \beta_1 x_0 + \beta_2 x_0^2 + \beta_3 x_0^3 + ... + \beta_{p-1} x_0^{p-1} \\ \tilde{y}_1 = \beta_1 + \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_1^3 + ... + \beta_{p-1} x_1^{p-1} \\ \cdot \\ \cdot \\ \cdot \\ \tilde{y}_{n-1} = \beta_0 + \beta_1 x_{n-1} + \beta_2 x_{n-1}^2 + ... + \beta_{p-1} x_{n-1}^{p-1} \end{bmatrix}$$

where:

$\tilde{\mathbf{y}}^T = [\tilde{y}_0, \tilde{y}_1, ..., \tilde{y}_{n-1},]$, $\tilde{\mathbf{y}} \in \mathcal{R}^n$

$\boldsymbol{\beta}^T = \left[\beta_0, \beta_1, ..., \beta_{p-1}\right]$, $\boldsymbol{\beta} \in \mathcal{R}^p$

Eventually, we can rewrite everything in a very compact way as:

$$\tilde{\mathbf{y}} = X\boldsymbol{\beta}, \tag{4}$$

where we define X as *Feature matrix*:

$$X = \begin{bmatrix} 1 & x_0 & x_0^2 & ... & x_0^{p-1} \\ 1 & x_1 & x_1^2 & ... & x_1^{p-1} \\ \cdot & & & & \\ \cdot & & & & \\ \cdot & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & ... & x_{n-1}^{p-1} \end{bmatrix} = \left[1\ X^{(1)}\ X^{(2)}, ..., X^{(P-1)}\right] \tag{5}$$

Because of Eq. 4, our model is linear in the unknown parameters $\boldsymbol{\beta}$. For this reason, the group of regressions methods that follow the above derivation is called *linear regressions*.

The main goal, then, of the regression methods, is to find the optimal parameters $\hat{\boldsymbol{\beta}}$ that best reproduces the outputs given the inputs. Therefore, we need a criterion to quantify how good is a model given by a particular $\boldsymbol{\beta}$. This is done by the so-called *Cost Function* $C(\boldsymbol{\beta})$, which assesses the model. Although there is a lot of theory and a large number of possible Cost Functions, we won't go into detail because is not the aim of this project. Thus, we want to find the parameters $\boldsymbol{\beta}$ that minimize the error, what means finding the minimum of $C(\boldsymbol{\beta})$. According to Fermat's theorem for optimization, we can find the minimum where the derivative vanishes, i.e:

$$\frac{\partial C(\beta)}{\partial \beta_j} = 0. \tag{6}$$

Depending on the $C(\beta)$ we choose, Eq. 6 will have an analytical solution or not.

Summarising, we aim to find:

$$\hat{\boldsymbol{\beta}} = \underset{\beta \in \mathbb{R}^p}{argmin}\ C(\boldsymbol{\beta}) \tag{7}$$

Throughout this project, we will study different methods to solve the above problem.

3. Models and data processing

3.1. OLS regression

To start to solve problem (7), we need to choose a $C(\boldsymbol{\beta})$. Our first choice will be the *Mean Squared Error* (MSE):

$$MSE = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2, \tag{8}$$

where y_i is the real value and $\tilde{y}$ the output from the model. Then for a good model, the MSE will tend to be small. For the cost function:

$$C_{OLS}(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j\right)^2 = \frac{1}{n}||(\mathbf{y} - X\boldsymbol{\beta})||_2^2, \tag{9}$$

where $||x||_2 = \sqrt{\sum_{i=0}^{n-1} x_i^2}$. This choice is very general and widely used in the field, so it can be usefull for many situations.

For this election, equation (6) has analytical solution:

$$\hat{\beta}_{OLS} = (X^T X)^{-1} X^T y. \tag{10}$$

Hence, for a given set of inputs and outputs, we could find the proper model solving Eq. 10. This linear regression method is called *Ordinary Least Squares* (OLS).

3.2. Ridge Regression

However, there is a sensitive part in that equation, the inverse $(X^T X)^{-1}$, if $\det(X^T X) = 0$. A cheap trick to avoid this is adding a λ parameter.

$$\hat{\boldsymbol{\beta}} = (X^T X + \lambda \mathbb{I})^{-1} X^T \mathbf{y}. \tag{11}$$

The cost function that would lead us to the above result applying equation (6) is:

$$C_{Ridge}(\beta) = \frac{1}{n}||(\mathbf{y} - X\hat{\boldsymbol{\beta}})||_2^2 + \lambda||\beta||_2^2 \tag{12}$$

This regression is called *Ridge Regression*.

The second derivative of Eq. 10,11 are, respectively:

$$\frac{\partial^2 C_{OLS}(\boldsymbol{\beta})}{\partial\beta\partial\beta^T} = \frac{2}{n}X^T X \text{ and } \frac{\partial^2 C_{Ridge}(\boldsymbol{\beta})}{\partial\beta\partial\beta^T} = \frac{2}{n}(X^T X + \lambda\mathbb{I}). \tag{13}$$

The eigenvalues of $X^T X$ are positive, meaning that it is positive definite. This implies that Eq. 6 applied to OLS and Ridge is a convex problem with a global minimum. This feature will be relevant in section 5.

3.3. Lasso Regression

Another well known regression method is called *Lasso Regression*. The cost function is:

$$C_{Lasso}(\beta) = \frac{1}{n}||(\mathbf{y} - X\hat{\boldsymbol{\beta}})||_2^2 + \lambda||\beta||_1, \tag{14}$$

where $||x||_1 = \sqrt{\sum_{i=0}^{n-1}|x_i|}$. Unlike the previous two methods, this one has not an analytical solution for equation (6), so we need to compute the result by numerical methods.

In the first part of the project, we studied these three regressions (10,11,14), solving them and testing different resampling methods.

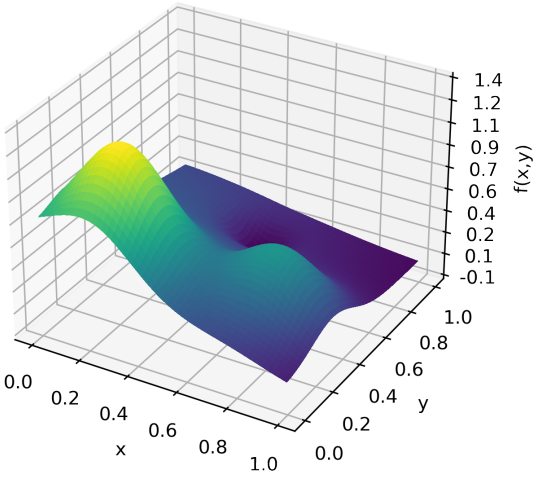

Fig. 1. Plot of the Franke Function in all its domain, $x, y \in [0, 1]$.

3.4. Franke function

To be able to test the regression methods, we need data pairs as in (1). Usually, these pairs are obtained from measurements/experiments with labeled data. Nonetheless, as we want to study the methods themselves, we will use a *Test function*, a function that we know perfectly so we can generate as many pairs of points as we want. In this section we will use as test function the so-called *Franke Function* (FF) defined as:

$$\begin{aligned} f(x,y) =& \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+\frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+\frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &-\frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right). \end{aligned} \tag{15}$$

See representation of the FF in Fig. 1. The domain of this function is $x, y \in [0, 1]$. So we will try to apply the regression methods shown to create a model.

From now until Sect. 5, we will be working with $n = 1000$ datapoints, generated $\sim U[0, 1]$

3.5. Pre-processing

Once we have the data, we can start our computing of the regressions. To solve OLS, first we have to compute the proper parameters $\hat{\beta}$ following equation (10). After, we can compute the predicted output with equation (4) and use the MSE to check how good is the model. But, before, we need to pre-process the data. For that, we need to:

- Split the data into both Training and Test dataset.
- Add stochastic noise to the data.
- Study the scaling of the data.

To choose the value for the splitting, we have trained a model and tested it for different splitting sizes. We used values between 50% and 90% of the total data size for the training set, and the remaining for the test set. We found that the best size is 70% of the data for the training data set and 30% for the test data (top panel of Fig. 2). But, why do we want to perform this split? This is the key to understand the difference between *optimization* and machine learning (Goodfellow et al. 2016). With optimization, we aim to find the minimum of a function. Then, we use all the available data. But, with machine learning methods such as linear regressions, what we want is to build a model to predict outputs given new inputs. So we need some way to check how good is our model. How to do this is with the validation or test set. This set works as "new" data for the model. So, once the model is trained with a training set, we will test it with the test set. Since we only have a given amount of data available, we need to split it between train and test sets. We quantify how good is the model computing the MSE_{train} and the MSE_{test}. The first one is computed with the known outputs of the training set and the predictions of the model over inputs of the same set. The second one, is computed in the same way but using the test set.

Regarding the noise, we generate it with a random normal distribution: $Noise \sim N(0, 1)$. To scale it, we need to multiply it by a factor. We want a noise, at least, one order of magnitude below the order of magnitude of our data to avoid the loss of information. We proceed as with splitting size, but with different noise factors. The best value for the factor is ~ 0.4, as we can see in the low panel of Fig. 2.

The scaling of the data is a very powerful tool depending on the situation. Many methods are sensitive to the scales of the features. So, to avoid problems such as overflow, underflow, strange behavior, or a lot of variation, we can re-scale the data. In this case, we will apply the standard scaling over the feature matrix:

$$X_j^{(i)} \to \frac{X_j^{(i)} - \bar{X}^{(i)}}{\sigma(X^{(i)})}, \tag{16}$$

where $X^{(i)}$ refers to the i column of the feature matrix 2. This scaling centers the data in 0 and his standard deviation in 1. Usually, this reduces the size of the data. But, in our case, using it only increases the size of our data (see bottom panel of Fig. 2). This would be useful if we were working with very little data to avoid underflow, but that is not the case. Our data is of the same order of magnitude as the scaling. Therefore, we won't use scaling.

In the following section, we will study different methods with OLS, Ridge and Lasso.

4. Bias-variance trade-off and resampling techniques

4.1. Overfitting vs Underfitting

The central challenge in machine learning is to generate models that perform well on new, unseen inputs, not just those on which our model was trained. In other words, we want that our model trained with the training set to fit well also the test set. When we compute the proper parameters $\hat{\boldsymbol{\beta}}$, we minimize the training error. But we also want to reduce the test error. This feature is called *capacity*, and can be understood as the ability of a model to fit new data. We can find three different scenarios:

- Underfitting: this is the case where we have low capacity. The model is unable to fit properly both training data and test

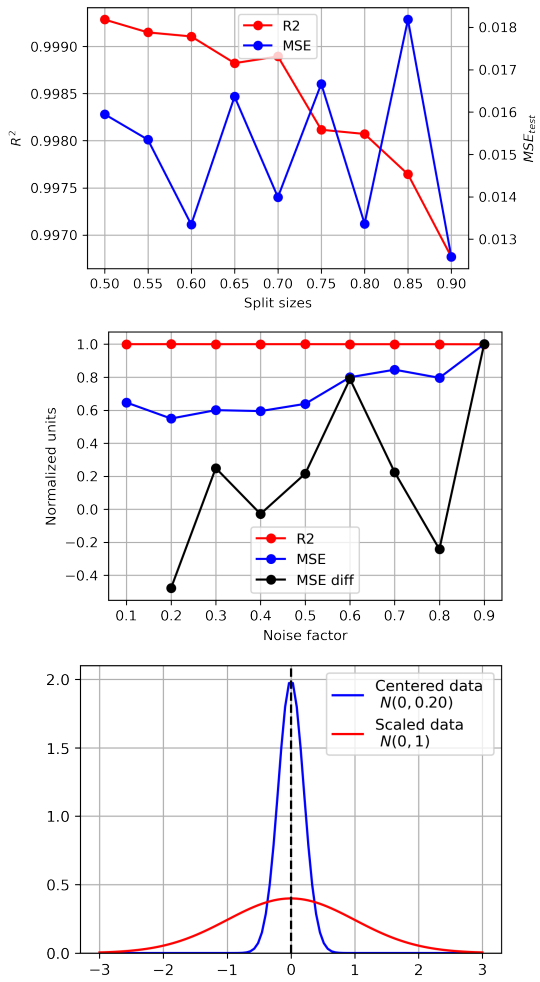

Fig. 2. Top panel: R^2 function and MSE for predicted data by OLS model trained for different split sizes of the training dataset. As can be seen, the best split is 70% of the total data in training set and the other 30% in test set. Afte this point, the MSE increases dramatically while the R^2 decreases in the same way. **Mid. panel**: R^2 and MSE for predicted data by OLS model trained for different noise factor. There is also plotted the difference of the MSE with the previous one. As we can see, the MSE is stable until factor 5 is reached. At this point, start to increase. With respect to the R^2 score, there is not significant variation. Therefore, we choose factor 0.4 for the noise. **Bot. panel:** Comparison of two normal distribution. The red one with the same parameters than the scaling of Eq. 16, $N \sim (0, 1)$. The blue one has, as parameters, the average σ of the of the generated data and 0 mean, $N \sim (0, 0.20)$. We have centered it to compare it with the red one. We can see than our data is less sparse than the scaled data. This means the scaling only would increase the size, thing that is not necessary in this problem.

data. This means that the model needs more complexity to fit. This scenario is recognizable because of large MSE_{train} and MSE_{test} and is since the model cannot fit the data sufficiently.

– Appropriate capacity: this is the case where we have a model that fits well both train and test set. This zone is characterized by the lower MSE_{train} and MSE_{test}. Is the optimal point for the model and refers to a high capacity.

– Overfitting: this case means low capacity as well. The model fits very well the train test but is not able to reproduce the test set. This happens because the model tailors too much to the train data, being unable to fit different data.

The ability of the model to tailor the data is due to its complexity degree. As more complexity, more tailoring. Then, we want to find the adequate degree of complexity that gives us an appropriate capacity, avoiding under and overfitting (see Fig. 5.2 of Goodfellow et al. 2016). To find the proper degree, we will reproduce Fig. 2.11 of Hastie et al. (2009) for our data and models (Fig. 3 upper left panel). This figure represents the training error and the test error of a given model in function of the complexity. On the left side, we identify the *underfitting* zone, which corresponds with low complexity. In the middle, we see the change in the behavior of the test error. Stop the decrease to increasing, reaching the minimum value for the test error. This change marks the optimal complexity, and corresponds to an appropriate capacity. Finally, on the right side, we can observe a growth in the test error while the train error is stills decreasing. This behavior indicates the *'overfitting'* zone.

To do the computations for the regressions and the MSE_{Train} and MSE_{Test} we followed, for OLS and Ridge, the algorithm 1, and for Lasso, the algorithm 2. In both algorithms, is taken for granted that the data is already split into Test and Train, referring $X_{train/test}$ to the feature matrices and $y_{train/test}$ to the data outputs.

Algorithm 1 OLS and Ridge regression

Do:
$X,\ y \leftarrow X_{train},\ y_{train}$
if OLS **then**
  $0 \leftarrow \lambda$
end if
Compute:
$\hat{\boldsymbol{\beta}} \leftarrow (X^T X - \lambda \mathbb{I})^{-1} X^T y$
$\tilde{\mathbf{y}} \leftarrow X\hat{\boldsymbol{\beta}}_{OLS}$
$\text{MSE}_{train} \leftarrow \text{MSE}(y, \tilde{y})$
Do:
$X,\ y \leftarrow X_{test},\ y_{test}$
Compute:
$\tilde{\mathbf{y}} \leftarrow X\hat{\boldsymbol{\beta}}$
$\text{MSE}_{test} = \text{MSE}(y, \tilde{y})$

Algorithm 2 Lasso regression

Do:
$X,\ y \leftarrow X_{train},\ y_{train}$
Compute:
$C(\beta) \leftarrow \frac{1}{n}\|(\mathbf{y} - X\hat{\boldsymbol{\beta}})\|_2^2 + \lambda\|\beta\|_2^2$ ▷ Lasso cost function (Eq. 14)
$\hat{\boldsymbol{\beta}} \leftarrow \underset{\beta \in \mathbb{R}^p}{argmin}\ C(\beta)$ ▷ Solve numerically
$\tilde{\mathbf{y}} \leftarrow X\hat{\boldsymbol{\beta}}$
$\text{MSE}_{train} \leftarrow \text{MSE}(y, \tilde{y})$
Do:
$X,\ y \leftarrow X_{test},\ y_{test}$
Compute:
$\tilde{\mathbf{y}} \leftarrow X\hat{\boldsymbol{\beta}}_{OLS}$
$\text{MSE}_{test} \leftarrow \text{MSE}(y, \tilde{y})$

At this point, we find with one problem. Since we are generating the data randomly, for each generation, we get different optimal parameters. To solve this, we run each regression 100 times with a new generated dataset each time. At each iteration, we perform the regression and compute the errors following Alg. 1 and 2 for different complexities and, eventually, do the average over the 100 different results for each complexity. In this first part of the project, to compute the inverse in OLS and Ridge

(Alg. 1), we have computed it directly using the inverse function or the pseudo-inverse function. The difference between both is that the second holds even if the determinant of the matrix is zero. (*inv* or *pinv* of the *Numpy.linAlg.package*)

With this method, we obtain a much more realistic result than for only one iteration, and we can also detect a general tendency. Figure 3 upper right the train and test error comparation for OLS. We can see that the proper complexity is $n = 5$. In the bottom row, there are the same but for Ridge and Lasso regression. As these two methods have an additional parameter λ, we have computed the optimal value previously. For this purpose, we have calculated the test error for a fixed complexity $n = 5$ (we chose this complexity because it is optimal for OLS case) and several λ's in the interval $[0.001, 10]$. For Ridge, the best performance is given by $\lambda = 0.17$ and, for Lasso, $\lambda = 0.001$. In the bottom left panel of Fig. 3 is represented the train and test error for Ridge regression. The same in the right panel for Lasso regression.

In all the three plots clearly show the same behavior, which coincides with that described in the first plot (upper left). For the three methods, complexity $n = 5$ is the best. It is not surprising that, for the three methods, we get the same complexity as the best one. The origin of this is the data itself. More specifically, the number of points that we are using. We are working with 1000 data points split into 700 points for training and 300 for testing. As more points we use, we get more information, so we need more complexity to fit them. We have the inverse phenomenon when we go to less and less data. We get less information of the function we want to fit. Less information means that the surface to be tailored is simpler. So we can reach that with a lower complexity.

4.2. Bias-variance trade off analysis

In this subsection, we will study the so-called *Bias-variance trade-off* using resampling methods. Resampling methods serve to generate different training and testing sets from the same data. This allows us to get more information about the model and asses it. When resampling methods are applied and we study the result with some technique, we obtain information, above all, about the quality of the data and the capacity of the model. With respect to the quality of the data, if the data is good, we expect to get a similar behavior when using the different resamples. If we get very different behavior, means that the data is not representative, so we should not use that model to make predictions. We don't have a representative sample of data points. If the data is good enough, then we can extract as well information about the capacity of the model as we did in the previous subsection. We will use the so-called *bias-variance trade-off*. It is computed by comparing the outputs of the model when it is trained for different complexities, with different data points from the same sample, i.e, changing the train set and, depending on the particular method, the test set. To perform this, we need some way to resample the training data set. We will use the *Bootstrap method* and the *k-Fold Cross-correlation method* in the following subsections.

The Bias-variance trade-off expression is derived from the MSE (Eq. 8):

$$\mathbb{E}[(y-\tilde{y})^2] = \frac{1}{n}\sum_i (f_i - \mathbb{E}[\tilde{y}])^2 + \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}[\tilde{y}])^2 + \sigma^2.$$

This is the theoretical expression. Nonetheless, since we don't know Eq. 2, we have to do the approach $y \simeq f$, so we get:

$$\mathbb{E}[(y-\tilde{y})^2] = \frac{1}{n}\sum_i (y_i - \mathbb{E}[\tilde{y}])^2 + \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}[\tilde{y}])^2 + \sigma^2, \quad (17)$$

where σ denotes the standard desviation of y.

The left term, $\frac{1}{n}\sum_i (y_i - \mathbb{E}[\tilde{y}])^2$, is the bias. This quantity measures how biased is the model, i.e, how different the values predicted are by the model w.r.t the real values when it is trained with different resamples. The more points we have, the more information we have about the objective function. Therefore, we need in general a larger complexity due to we have to fit a more complex surface. If the model is good, the bias should be minimal. On the other hand, a large bias means that the model is not good at generalizing because, changing the training set, the predicted values change too much.

The right term, $\frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}[\tilde{y}])^2$, represents the variance. This gives the variance of the predicted values with the same input. It is a measure of how different a predicted value is when using different training set. If the variance is large, it means that for different training data sets, we get very different predicted outputs for the same input. This is also a bad indicator of generalisation. A good model has to give the same or similar output for the same input. Like bias, this depends on the number of data points and complexity.

For a given amount o data points, there will be a proper complexity. However, we cannot normally control the amount of data we have, it is fixed. So it is usual to work only with the complexity because it is the only parameter we can play with.

4.3. Bootsrap method

Bootstrap method was first proposed in Efron (1979) and is a variant of the Jacknife method (see Mosteller & Tukey 1968). According the bootstrap theorem, it is a form of resampling that converges to the mean value and variance of the data if we do it enough times. It consists on generating M different training sets from the original training set. We create the new sets by randomly taking points from the original training set and allowing replacement, i.e, choosing the same point more than once. Algorithm is represented in Alg. 3.

Algorithm 3 Bootstrap resampling

Require:
$D = (x_0, y_0), ..., (x_{n-1}, y_{n-1})$
$M \leftarrow$ number of Bootstraps resamples
$S \leftarrow$ number of samples per resample
$D_{train}, D_{test} \leftarrow D$
Do:
for i=1,...,M **do**
  $D^i_{train} \leftarrow S$ random points with replacement from D_{train}
end for

Once we have all the new generated train sets, we train the model from scratch for each one and compute the MSE_{test} with the same test set always. Then, we give as final MSE_{test} the mean over the M resamples. We have done this for all three OLS, Ridge and Lasso methods using 100 resamples. For each resample, we have chosen the same number of points than the original train set. The results are shown in Fig. 4. There is represented the bias-variance trade-off. In all three plots, yellow line marks the bias, cyan one the variance and the green one the MSE_{test}. We can observe that the behavior of these quantities match with the zones indicated in the upper left panel of Fig. 3. The left sides of the plots of Fig. 4 show a *high bias* and *low variance* zone. This indicates us that, with such complexity, model is not able to generalise. The right sides belong to the *low bias* and *high variance* zone. It is obvious than the variance increases with respect the

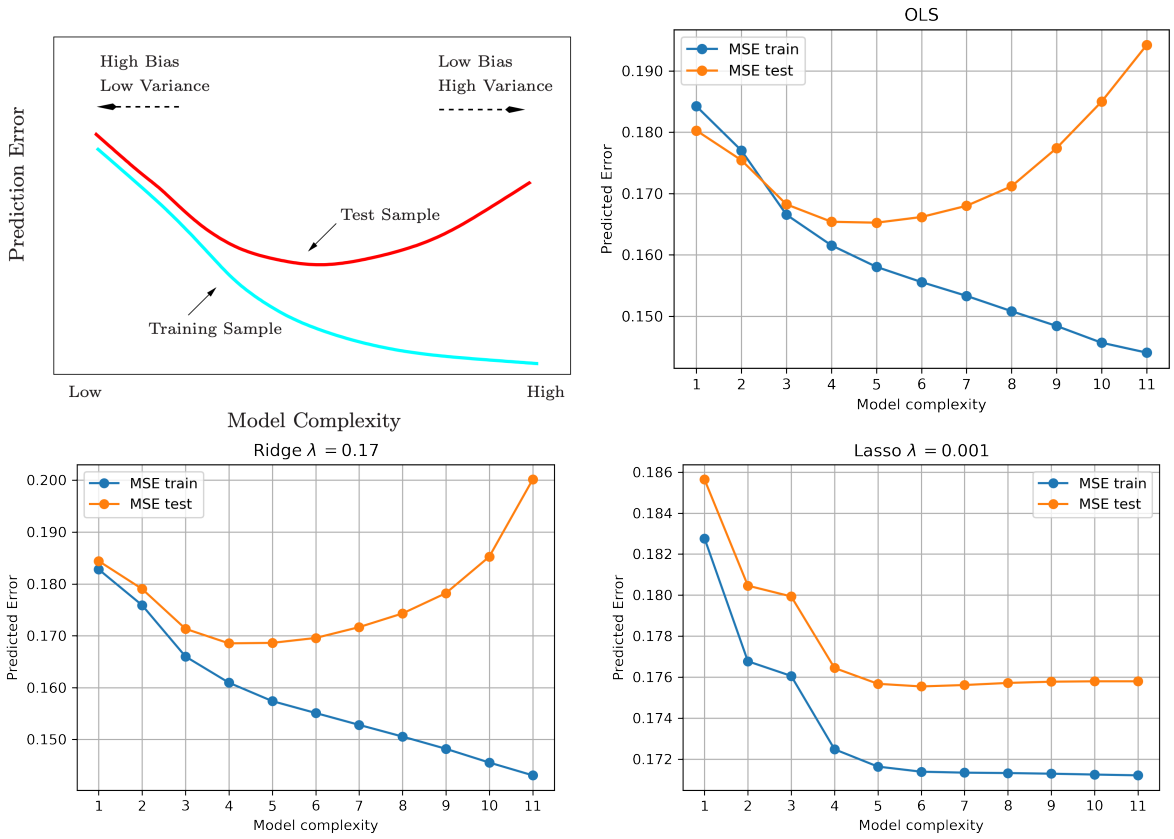

Fig. 3. Upper left: Fig. 2.11 of Hastie et al. (2009). There is represented the Bias-Variance trade-off and the behavior of the test error and training error with the complexity of the model. The left side belongs to the *underfitting* zone or the High Bias/Low Variance zone. Here, the model is not able to fit. The middle zone represents the appropriate complexity, where the model achieves the lowest test error. This is the optimal complexity. The right zone represents the *overfitting zone* or Low Bias/High Variance. In this zone, the test error starts to increase while the training error stills decreasing. **Upper right**: Bias-Variance trade-off of OLS regression. We can identify the same behavior as the upper left plot. The optimal complexity is $n = 5$. **Bottom left**: Bias-Variance trade-off of Ridge regression using $\lambda = 0.17$. The optimal complexity is $n = 5$. This plot reproduces as well the expected trend. **Bottom right**: Bias-Variance trade-off of Lasso regression with λ =0.001. The optimal complexity is $n = 5$. In this case, we see again the expected behavior of the test and train error w.r.t the model complexity.

left side, indicating the overfitting of the model. The bias, for the top and bottom panel shows a growing tendency. However, is not as dominant when is compared with the MSE as in the left side. Proportionally, therefore, it has declined. The area with lowest bias and variance, and hence MSE, is in the center of the panels, with complexities of 4 and 5. This is the expected proper complexity because we got the same values in the previous analysis.

4.4. K-folds Cross Validation

K-Folds Cross Validation (CV) is another technique for data resampling. Unlike Bootstrap, in this method, both test and train datasets are changed at each iteration. Here the procedure consists on split the data into a number of "k" folds of the same or similar size. Then, we train **k** models from scratch. For each model, we choose a different fold as test set and the others as train set. As final MSE_{test} we present the average:

$$\mathrm{MSE}_{\mathrm{Test}} = \frac{1}{k}\sum_{i=0}^{k-1} \mathrm{MSE}_{Test}(i). \tag{18}$$

As an example, see Fig. 5.

We have performed this method for all three OLS, Ridge and Lasso as well. We compared the MSE_{Test} for different number of folders. Also, to achieve the maximum possible generality, we have computed the MSE_{Test} for each folder 100 times, generating new data each time and using the mean of all the iterations. As number of folds, we chose multiples of 1000, the total amount of data, to obtain folds of equal size. For the three methods, we used a complexity of degree n = 5 and $\lambda = 0.17$ and 0.001 respectively for Ridge and Lasso.

Results are shown in Fig. 6. We can see that, independently on the method, there is a trend. For all three, the best $\mathrm{MSE}_{\mathrm{Test}}$ is gotten for 4 folds. After this, the $\mathrm{MSE}_{\mathrm{Test}}$ grows monotonously. This expected behavior can be explained with the ratio train to test data. As we increase the number of folds, the ratio increases a lot. This means that even if we have a lot of data available to train, the test set is not a good representative of the data because it has too few points. On the other hand, for a little number of folders, we don't have either the optimal proportion between train and test data. The ratio is too small. Then, we don't have data enough to train the model. The obtained result in 4 folders means a splitting of $\sim \frac{1}{4} \simeq 0.25$. This value is close to the value

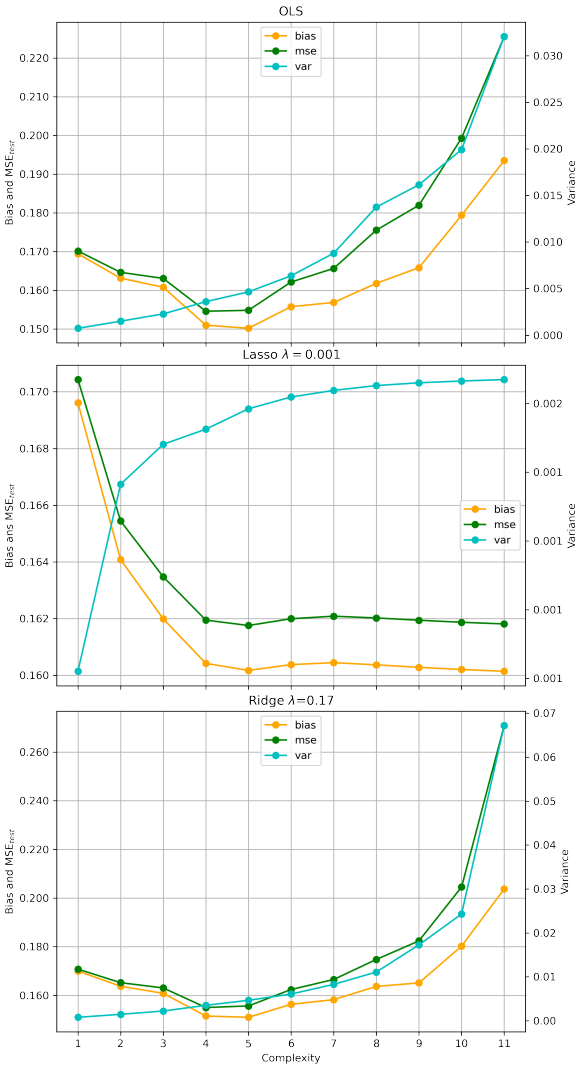

Fig. 4. Bias-variance trade-off representation in function of model's complexity for, from top to bottom, OLS, Lasso, and Ridge regressions. In the three panels we can see the *high bias* and *low variance* zone on the left side, the *low bias* and *high variance* zone on the right side, and the optimum zone at the middle. Realize that the variance scale is on the right side of the panels.

we have used to split the data, $\frac{train}{test} = 0.3$, defined in the section 3.5. So this is another indicator that we chose a good ratio.

4.5. Comparation of methods.

To finish this section, we will compare the MSE_{Test} of OLS, Ridge, and Lasso methods with no resampling, Bootstrap resampling, and CV resampling. We have used the results obtained in the previous computations, so the parameters are the ones decribed throughout the section. The result is summarized in table 7. The best result is achieved by OLS, followed by Ridge and Lasso, three of all using Bootstrap. This is followed by CV and no resampling respectively. Is to be expected that, with resampling, we get better results. However, the difference between the

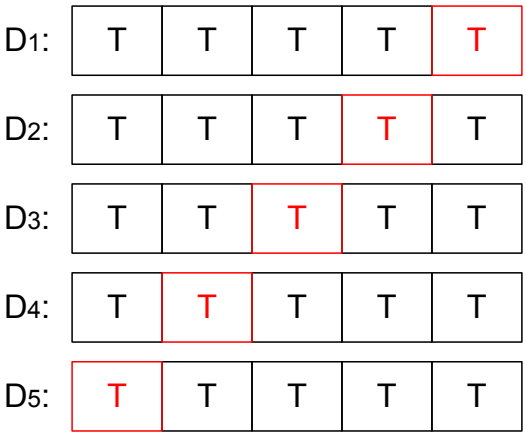

Fig. 5. Example of data resampling using CV. We divide the data into **k**=5 folds. Then we train 5 independent models. For each one, we select as test set a different fold, marked with the red color for each different model ($D_1, ..., D_5$). Therefore, we train the models using the black folds as the training set and the red one as test set. As final MSE_{Test} we provide the average over the MSE_{Test} of all the models, following Eq. 18.

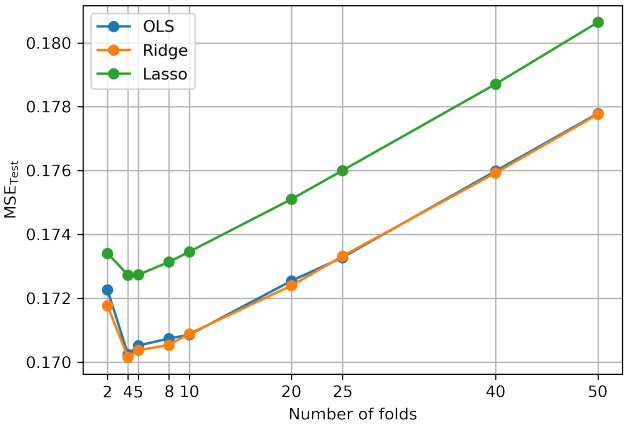

Fig. 6. Caption

values is not significant. This means that the data we are working with has a good quality, so when we split with an adequate ratio between train and test, both train and test are representative data. This is to be expected since we are generating data from a uniform distribution, so the data is well distributed.

5. Stochastic Gradient Descent methods

5.1. Gradient Descent methods

In this section, we have solved OLS and Ridge using the so-called **Gradient Descent** (GD) methods. To understand them, we first have to explain, very briefly, the *Newton-Rapshon method*, commonly known only as the *Newton method* unfortunately for Raphson.

We want to find the zero of a given function $f(x)$, which is analytical. Then, we do the Taylor expansion of f centered at x_n and truncating after the linear term and equal it to zero:

$$f(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n) = 0.$$

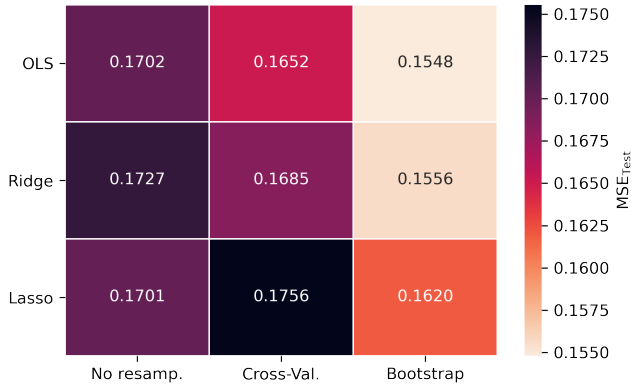

Fig. 7. Table for the MSE_{Test} for OLS, Ridge and Lasso methods with no resampling, CV and Bootstrap. The best score is obtained by Bootstraping with OLS, while the worst one is obtained by CV with Lasso.

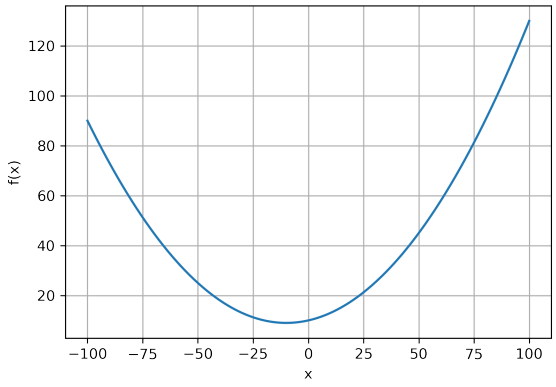

Fig. 8. Representation of function used for GD (Eq. 23)

Solving this equation, we get:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \ n = 0, 1, 2, ..., \tag{19}$$

where n is the iteration. It is thus an iterative method that over a given number of iterations approaches to the minimum of $f(x)$ (Gautschi 2011). On the other hand, if we want to find the minimum of a function $f(x)$, we know that this is located in the point x where holds:

$$\frac{\partial f}{\partial x} = f'(x) = 0,$$

as in the Eq. 6. To find this zero, we can apply the Newton method (Eq. 19) to the first derivative:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} = x_n - \mathcal{H}(x_n)^{-1} g(x_n),$$

where $\mathcal{H}(x_n)$ is second derivative known as well as *Hessian* and $g(x_n)$ the first derivative or gradient. However, in real cases where we have to handle a big amount of data and multivariate functions, computing the Hessian is not affordable because is too computationally expensive. Then, we will replace the Hessian by the so-called *Learning rate* γ getting:

$$x_{n+1} = x_n - \gamma_n g(x_n). \tag{20}$$

This is the expression for the GD methods. Depending on the treatment we give to γ or to the data, we face different versions, but Eq. 20 is the basis.

We can apply GD methods to solve Eq. 7. Iteratively, we can find the optimal $\hat{\beta}$ and this is what we will do throughout this section. Since we want to find $\hat{\beta}$ that meets Eq. 6, we will solve:

$$\beta_{n+1} = \beta_n - \gamma_n g(\beta_n) \tag{21}$$

where $g(\beta)$ represents the gradient of the Loss function with respect to β:

$$g_{OLS}(\beta) = \nabla_\beta C_{OLS}(\beta_k)^T = X^T[X\beta - y] \tag{22}$$

To work, we will use a function other than Franke Function, an easier one:

$$f(x) = 10 + 0.2x + 0.01x^2. \tag{23}$$

The values of the exponents are arbitrary. The function is represented in the Fig. 8.

For the computations, we have used 1000 points generated following a uniform distribution in the interval $[-100, 100]$. Also, unlike the previous case, here we have scaled the feature matrix following Eq. 16. Without this, the GD based methods do not converge even for very good learning rates. With the scaling, we have removed the interception, i.e, the first column of the feature matrix. This is done because, if we remove the mean of each column, the first columns of 2 becomes in a column of zeros. So it is useless information.

5.2. Plain GD and GD with momentum

GD with momentum (GDM) is an extension of the GD presented in Polyak (1964). It consist on the plain GD (Eq. 20) with an extra term called *momentum* δ:

$$\beta_{n+1} = \beta_n - \gamma_n g(\beta_n) + \delta(\beta_n - \beta_{i-1}), \tag{24}$$

where the last term is the momentum. The function of the momentum is to accelerate learning. It accumulates an exponentially decaying moving average of past gradients and continues to move in their direction (Goodfellow et al. 2016).

We have compared the performance of GD with GDM, for different values of γ and δ. We have used the absolute value of the gradient to consider if the method has converged or not. If, for a given iteration n, the gradient $\|\nabla_\beta C(\beta_n)\|$ is equal or lower than a given ϵ, we consider the model trained. We set also a maximum number of iterations *maxiter* and an initial value for B, $B_{initial}$. The values we have used for all this section are:

- $maxiter = 100000$
- $\epsilon = 0.000001$
- $\boldsymbol{\beta_{initial}} = \mathbf{0.1}_p$,

where $\mathbf{0.1}_p$ represents a vector of the same dimension as $\boldsymbol{\beta}$ consisting of 0.1. We have used as well a complexity $n = 3$ for the feature matrix.

The algorithm can be seen in Alg. 4. Realize that the GD and GDM are the same algorithm if $\delta = 0$.

To set an initial range of γ, we explore a wide interval using $\delta = 0$. The result is that, for values larger than $\gamma = 1 \cdot 10^{-3}$ GD does not converge. For values smaller than $\gamma = 4 \cdot 10^{-5}$ the algorithm exceeds the maximum number of iterations. Hence, we have used the interval $\gamma \in [4 \cdot 10^{-5}, 1 \cdot 10^{-3}]$. For δ, we have used the interval $\delta \in [0, 0.9]$, corresponding the first value $\delta = 0$

Algorithm 4 GD and GDM algorithm

Require: Global γ
Require: Global ϵ
Require: Global *maxiter*
Initialize $\boldsymbol{\beta}_0 \leftarrow \boldsymbol{\beta}_{initial}$ and $n \leftarrow 0$
if GD **then**
  $\delta \leftarrow 0$
end if
while $\|\nabla_\beta C(\boldsymbol{\beta}_n)\| > \epsilon$ or $n < maxiter$ **do**:
  $\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n - \gamma \nabla_\beta C(\boldsymbol{\beta}_n) + \delta(\beta_n - \beta_{n-1})$
  $n \leftarrow n + 1$
end while

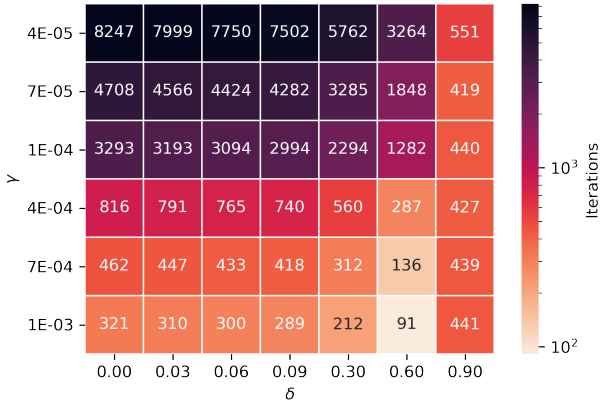

Fig. 9. Table with the iterations needed to converge for different combination of γ and δ using GDM. The best value is achieved for $\{\gamma, \delta\} = \{0.001, 0.6\}$. The first row corresponds to GD due to $\delta = 0$. The colour is represented in log scale to improve the visualization.

with the GD. The results of the comparison are in the Fig. 9. There are represented the iterations for each combination of γ and δ. The first row corresponds to the plain GD.

We can clearly see that the best performance, the lower number of iterations, is achieved by the pair $\{\gamma, \delta\} = \{0.001, 0.6\}$. Also, all the scores with non-zero momentum beat the GD. Therefore, GDM has a better performance than GD. Also, we see that the number of iterations change much more with the δ than for the γ. This means that the GD is very sensitive to choices of learning rates, while δ just increases the speed of convergence. But the learning rate is much more determinant.

5.3. Stochastic GD

Algorithms that use the entire training set, as GD or GDM, are called batch gradient methods, because they process all the training examples simultaneously in a large batch. But there are, also, algorithms that use only a single example of data at a time. These algorithms are called *stochastic* and use the technique of *Mini Batches* (MB). This consists on divide the data in a given number M of equal-sized or almost equal-sized batches. Then, at each iteration, we only use one batch to compute the gradient and the update of the parameter, $\boldsymbol{\beta}$ in our case. We define an *epoch* as the set of iterations where we have used the M batches. Then, for a number k of epochs, we have done $k \cdot M$ iterations, where in each one the gradient and update is done only with one batch at a time. Depending on the code used, the MB can change at each iteration or be the same along all the computation. In our case, we change the MB in each iteration, so it is very unlikely that we use the same exact batch twice. Stochastic Gradient Descent (SGD) consists on performing gradient descent methods using MB. Although at the end of the epoch we have gone throughout all the data points as in GD and GDM, we compute the gradients of only one batch at a time. For a large amount of data, as in the real cases, this is much more computationally efficient and requires much less memory than using the whole set for each gradient calculation (Goodfellow et al. 2016).

A very important point here is how to assess the model and the convergence. In the previous section, our criterion was the absolute value of $\boldsymbol{g}(\boldsymbol{\beta})$, i.e, $\|\nabla_\beta C(\boldsymbol{\beta}_n)\|$. This was valid because we where using all the points to compute the gradient and the update at each step. Nonetheless, we can not use that with SGD. When we use MBs, we are splitting the data. This means that, for each MB, we have a different feature matrix X. The shape of our objective function depends of X, i.e, $C_{OLS} = C_{OLS}(\boldsymbol{\beta}, \boldsymbol{X})$. Therefore, at each MB, we change the function, changing as well the minimum. So, if for one MB the gradient is very small, for another using the same $\boldsymbol{\beta}$ it can be very large. We want to use a proper amount of batches such that the different minimums are close enough to make possible the convergence. We can see an example of this divergence in the Fig. 10. There is represented, in one hand, the gradient (Eq. 22) as a vector map. We have inverted the sign of the gradients so they point towards the minimum. We have used a complexity $n = 2$, so $\boldsymbol{\beta} = (\beta_1, \beta_2)$ On the other hand, there are represented, for the data we are using, the minimum of the C_{OLS} in function of the batches used. The red point indicates the minimum point of the function. The blue, orange and green dots represent the minimum for different sizes of MB. For each size, we plotted the minimum of 8 different MB. For the larger number of MB, the blue points, we see a wide dispersion. This means that we don't have enough information in each MB, so it is very difficult to converge to some optimal result. The orange and green dots have a similar dispersion and are much more closer to the red point than the blue points. This means that we have approximately the same information in batches of 100 points as in batches of 50 points. Therefore, if we choose 20 MB instead of 10MB, we will have a faster computation and be sure that the algorithm converges.

Then, to check if our algorithm works properly, we will compute the MSE between the $\boldsymbol{\beta}$ computed by solving Eq. 10 and the $\boldsymbol{\beta}$ computed by our SGD algorithms.

5.4. Learning rate schedulers

In addition to the MB, it is very common, if not always, to use *learning rate schedulers*. What this schedulers do is to modify the value of γ with each epoch. When we performed the GD and GDM, we used a fixed γ until the convergence. However, it is not a good technique because γ modifies the distance we move in each iteration, and we are not as close to the minimum in the later iterations as in the earlier ones. With too large γ, we can get stuck around the minimum, as represented by the red arrows in Fig. 11. On the other hand, we can get stuck in some local but not global minimum or reach the minimum after a huge number of iterations, how is represented by the blue arrows in the Fig. 11. This is not efficient due to the time it consumes. Modifying γ at each epoch improves the algorithm, making it smaller and smaller as it gets closer to the minimum. This is represented by the green arrow in Fig. 11.

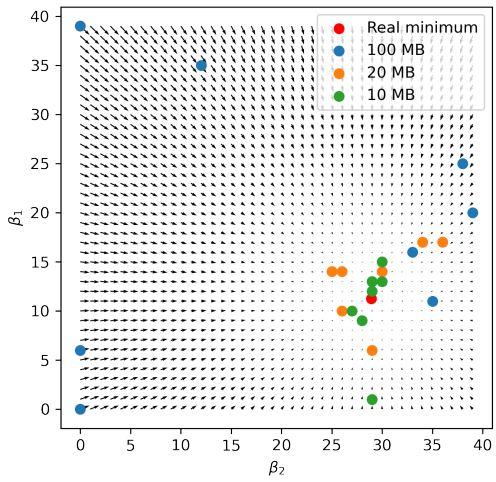

Fig. 10. Minimum of the C_{OLS} in function of the batches used. As more MB, the minimum of the function diverges more for the given MB composition. This means that, as more MB we use, less information we get from the data.

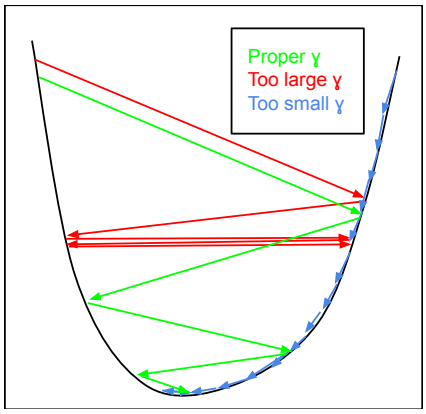

Fig. 11. Representation of the convergence for a scheduled γ (green), a fixed too large γ and a fixed to short γ (blue).

Since we are working with a convex problem with a global minimum, as explained in the section 3, and we will use MB, we don't have to worry about getting stuck in some local minimum. Therefore, we can describe a general algorithm for SGD as follows:

Algorithm 5 General SGD

Require: Global γ
Require: Initial $\boldsymbol{\beta_0}$
while stopping criterion not met **do**
  Sample a MB of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
  Algorithm to compute gradient $\boldsymbol{g}(\boldsymbol{\beta})$
  Apply update: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \Delta\boldsymbol{\beta}$
end while

Our *stopping criterion* would be the number of epochs we use. For each epoch, we sample all the M batches and update $\boldsymbol{\beta}$ M times. Depending also on the particular algorithm used, the computation of the gradient will differ, including the change of γ.

5.5. AdaGrad, RmsProp and Adam schedulers

We have tested different the three most used learning rate schedulers. These are *AdaGrad* (Duchi et al. 2011), *RMSProp* (RMS) (Hinton et al. 2012) and *Adam* (Kingma & Ba 2014). These methods are described in algorithms 6,7 and 8 respectively. As a curiosity, the last paper mentioned is one of the most cited ever. We will not study the algorithm in detail, as this is not the aim of the project. However, we will study their performance and compare them (Goodfellow et al. 2016).

Algorithm 6 AdaGrad algorithm

Require: Global γ
Require: Initial $\boldsymbol{\beta_0}$
Require: Small constant δ, usually 10^{-6}
Initialize $r = 0$
while stopping criterion not met **do**
  Sample a MB of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
  Compute gradient $\boldsymbol{g}(\boldsymbol{\beta})$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow r + \boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter: $\Delta\beta = -\frac{\gamma}{\delta + \sqrt{r}} \odot \boldsymbol{g}$
  Apply update: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \Delta\boldsymbol{\beta}$
end while

Algorithm 7 RMSProp algorithm

Require: Global γ, decay rate ρ
Require: Initial $\boldsymbol{\beta_0}$
Require: Small constant δ, usually 10^{-6}
Initialize $r = 0$
while stopping criterion not met **do**
  Sample a MB of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
  Compute gradient $\boldsymbol{g}(\boldsymbol{\beta})$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho r + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter: $\Delta\beta = -\frac{\gamma}{\sqrt{\delta + r}} \odot \boldsymbol{g}$
  Apply update: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \Delta\boldsymbol{\beta}$
end while

We have compared all three algorithms using different numbers of MB: 1, 10, 20, and 50; Three different initial γ, γ_0 and two momentums, $\delta = 0$ and $\delta = 0.6$. We chose this value for σ because was the proper one for GDM. The result is presented in Fig. 12. Solid lines represent $\delta = 0$ and dotted ones $\delta = 0.6$.

From left to right, we see how for small MB the first algorithm in converge is, for all three γ_0, RMS, followed by AdaGrad and Adam. However, as we increase the number of MBs, Adam dominates for $\gamma_0 = 0.1$, and 0.01 followed by AdaGrad and RMS.

The number of required epochs to converge varies a lot, being the determinant factor γ_0. For 1 MB, only AdaGrad converges for $\gamma_0 = 0.1$ and no one for $\gamma_0 = 0.01$. This means that this last value is to small if we use only 1 MB. Required epochs to converge decrease as we increase the MB. However, for the smallest γ_0 we find the same problem, it is to small. Only Adam with momentum is able to converge for MB=10 and both Adam for MB = 20 and MB = 50. The best convergence speed is achieved by $\gamma_0 = 0.1$ for MB = 20 and 50, where all the algorithms converge. Adam is the fastest for MB=50. With respect to the case of $\gamma_0 = 1$, algorithms do not converge. Another important point is the general tendecy to improve with momentum. The versions with momentum $\delta = 0.6$ are faster than the versions

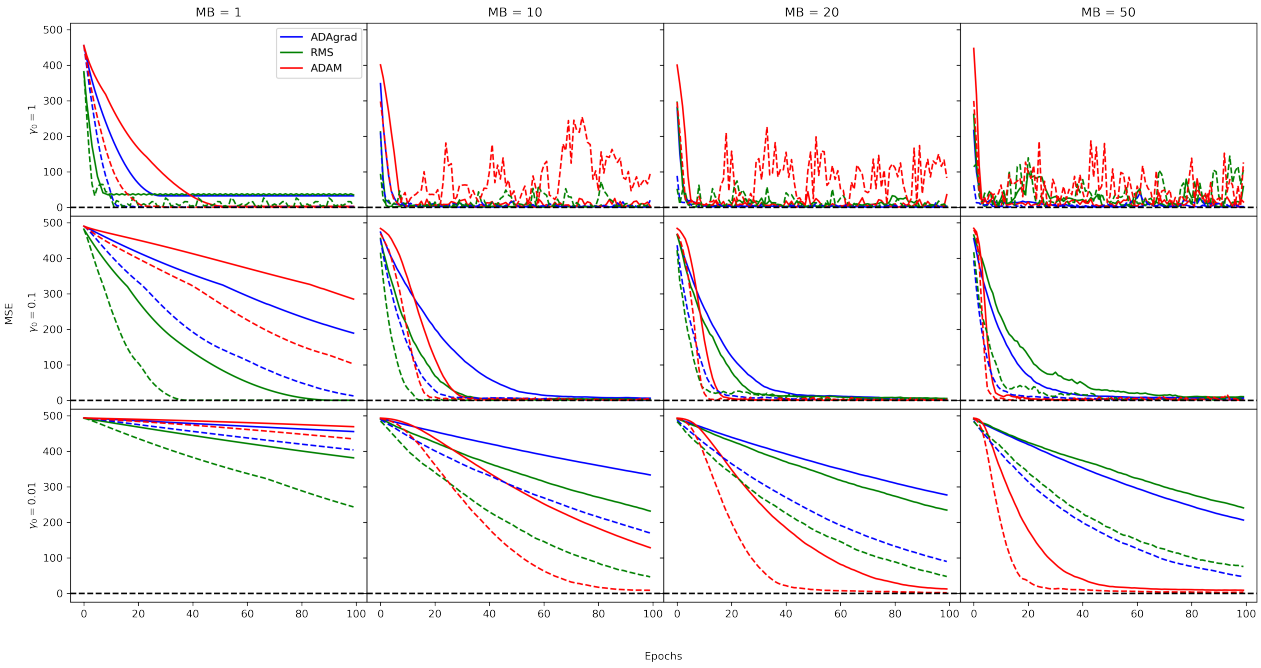

Fig. 12. MSE between $\boldsymbol{\beta}$ computed by 10 and SGD with different initial γ as γ_0, MB sizes and momentum. Solid lines represents no momentum or $\delta = 0$ and dashed liens represents a momentum $\delta = 0.6$. For more information, read explanation in text

Algorithm 8 Adam algorithm

Require: Global γ, decay rate ρ_1 and ρ_2
Require: Exponential decay rates for moment estimates,ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: initial β_0
Require: Small constant δ, usually 10^{-8}
Initialize 1st and 2nd moment variables $s = 0, r = 0$
Initialize time step $t = 0$
while stopping criterion not met **do**
  Sample a MB of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
  Compute gradient $\boldsymbol{g}(\boldsymbol{\beta})$
  $t \leftarrow t + 1$
  Update first momentum: $s \leftarrow \rho_1 s + (1 - \rho_1)\boldsymbol{g}$
  Update second momentum: $r \leftarrow \rho_2 r + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
  Correct bias in first momentum: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1{}^t}$
  Correct bias in second momentum: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2{}^t}$
  Compute parameter: $\Delta\beta = -\gamma \frac{\hat{\boldsymbol{s}}}{\delta + \sqrt{\hat{\boldsymbol{r}}}}$
  Apply update: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \Delta\boldsymbol{\beta}$
end while

with $\delta = 0$. The biggest difference are found for MB=10,20 and 50 and $\gamma_0 = 0.01$, where all the times are really improved with the implementation of momentum (see difference between solid and doted lines).

In conclusion, for a proper γ and little MB, RMS domains. However, when we need to use more MB due the ammount of data, Adam beats RMS by far.

6. Logistic regresion

In this section, we will work in a classification problem. The objective of these problems is, given a certain number of parameters, of features, belonging to the same sample, to classify this sample into different target classes. If we only have one possible case, as will be our case, then we have a binary classification problem. Given the input with its features, the model has to predict whether the input belongs to the target class or not. For example, we will work with cancer breast data. Each patient, i.e, each sample, has different tumour features, such as the mean radius, mean texture or mean symmetry. Therefore, we want the model to predict if the patient has cancer or not. To do this, we will use *breast cancer data* from *Scikit-learn* (Scikit) (Pedregosa et al. 2011a). This dataset contains, in total, 569 patients with 30 features per patient. But, first, let's look at how to do the classification. By convention, we wil call the detections, i.e cancer patients, positives (target = 1), and healthy patients, negative (target = 0).

6.1. Sigmoid function and Cross-entropy.

In the previous sections, we were interested in training a model that replicates a function. Hence, we expected outputs within the range of the function. However, we are now interested in an output $\in [0, 1]$, which indicates the probability of belonging to the class. So we have to change the range of the output from $\mathbb{R}$ to $[0, 1]$. In binary problems, this is usually done with the *Sigmoid function*:

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}, \ \mathbb{R} \rightarrow [0, 1]. \tag{25}$$

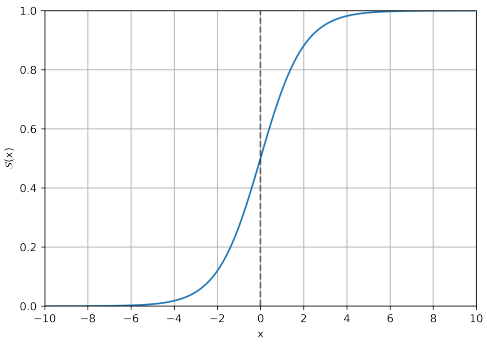

Fig. 13. Representation of the Sigmoid function.

It is represented in the Fig. 25. This function collapses any number into [0, 1], being 0 the negative numbers lower than -6 and 1 the positive numbers larger than 6 (Zhang et al. 2021).

Then, we apply this function to obtain a number that we can interpret as a probability, but which is not a real probability. Nonetheless, we wont dive into this topic because is not part of the project. For more information, read Guo et al. (2017). If we use a model of complexity 1, our output would be:

$$\mathcal{S}(x) = \frac{e^{\beta_0+\beta_1 x}}{1+e^{\beta_0+\beta_1}}. \tag{26}$$

On the other hand, as loss function, we will use the so-called *Cross entropy*, defined as:

$$H(y,\hat{y}) = -\sum_i y_i log\hat{y}, \tag{27}$$

where y_i represents the i_{th} target and $\hat{y}_i$ represents the output of our model. This function computes "how similar" are the probability distributions $\hat{y}_i$ and y_i, given a set of inputs x_i following a probability distribution y_i. For more detail, check chapter 3 from Goodfellow et al. (2016). This loss function is widely used in classification problems. The name *Logistic Regression* is due to the inputs x of Eq. 25, known as *logits* and for the *log* transformation to derive 27 from a binomial distribution and not from the Information Theory. Anyway, it is a classification problem with a different name.

6.2. F1 score, accuracy

In classification problems, additional metrics or scores are often used in addition to the loss function to asses the model. Let's define the following quantities:

- True Positives (TP): inputs we classify as positives and are positives.
- False Positives (FP): inputs we classify as positives and are negatives.
- True Negatives (TN): inputs we classify as negatives and are negatives.
- False Negatives (FN): inputs we classify as negatives and are positives.

With these definitions, we can define:

- Precision = $\frac{\text{TP}}{\text{TP+FP}}$
- Recall = $\frac{\text{TP}}{\text{TP+FN}}$
- Accuracy = $\frac{\text{TP+TN}}{\text{TP+FN+TN+FP}}$
- F1 = H(Precision, Recall) = $\frac{\text{TP}}{\text{TP+0.5(FP+FN)}}$

All these quantities are between 0 and 1. Precision is a measure of, given all the events we classify as positives, how many really are. A precision of 1 means that all the events we have classified as positives are positive. Recall quantifies how many positives we have correctly detected out of all positives. Accuracy is the ratio between the correct classifications, both positive and negative, out of all the cases. A common desired result is when recall and precision are balanced. This means, detect with precision as many events as possible. This is computed with the F1, commonly known as F1-Score, which is the harmonic mean between precision and recall. Evaluating all these metrics can give us a better picture of our model. An important point is that, to compute the metrics above, we need an output being 0 or 1, but the range from Eq. 26 is between 0 and 1, both include. We need to *collapse* the values to 0 or 1 using a threshold. When working with real probabilities, we use 0.5 as threshold. This means that lower values mean that negative is more likely than positive and vice versa for values above 0.5. To achieve this behavior in our model, we need to calibrate it, but as this is beyond the scope of this project, we will not do so. For more information, see Guo et al. (2017). We will assume, for practicality, the threshold in 0.5, considering positive the outputs ≥ 0.5. An important aspect of the metrics presented is that they depend on the balance of the data, i.e. the ratio between the number of detections and the number of non-detections. Therefore, they are not reliable for comparing different models trained and tested on different data. In our case, we always use the same data, so we do not have this problem.

6.3. Breast Cancer Classification

Once we have defined the metrics we will use, is time to train the model. To summarise, we will use as cost function:

$$C(\boldsymbol{\beta}) = -\sum_i y_i log\hat{y}, \text{ where } \hat{yi} = S(x_i) = \frac{e^{\beta_0+\beta_1 x_i}}{1+e^{\beta_0+\beta_1 x_i}}. \tag{28}$$

As we don't have a large amount of data, we can use plain GD, described in Alg. 4. We split the data as before, 70% for train set and 30% for test set. We have run 100 iterations with the momentums $\delta \in [0, 1]$ and learning rates $\gamma \in [0.001, 0.1]$. For each case, we computed all Precision, Recall, Accuracy, and F1 for the test data. The results are shown in the Fig. 14. The overall result is that the values do not change significantly, except in the bottom right of the panels, with no values for F1, Recall and Precision and with a value of 0.409 for the Accuracy. These values are produced because there are not TP, so the model didn't converge. This is characterized by a learning rate or momentum that is too large. The largest variation between results is observed in the F1, where the best one is achieved for $\delta = 0.9$ and $\gamma = 0.001$, with F1=0.976. This maximum is also achieved for Accuracy and Precision by the same parameters combination, with Accuracy=0.971 and Precision=0.953. Eventually, for the Recall, the maximum value is achieved by different combinations, including the maximizer combination in the other metrics. Then, the proper values to train the model are $\gamma = 0.001$ and $\delta = 0.9$.

7. Conclusions

In the second section, we study the main Least Squares Regression methods for the same case, obtaining best performance for

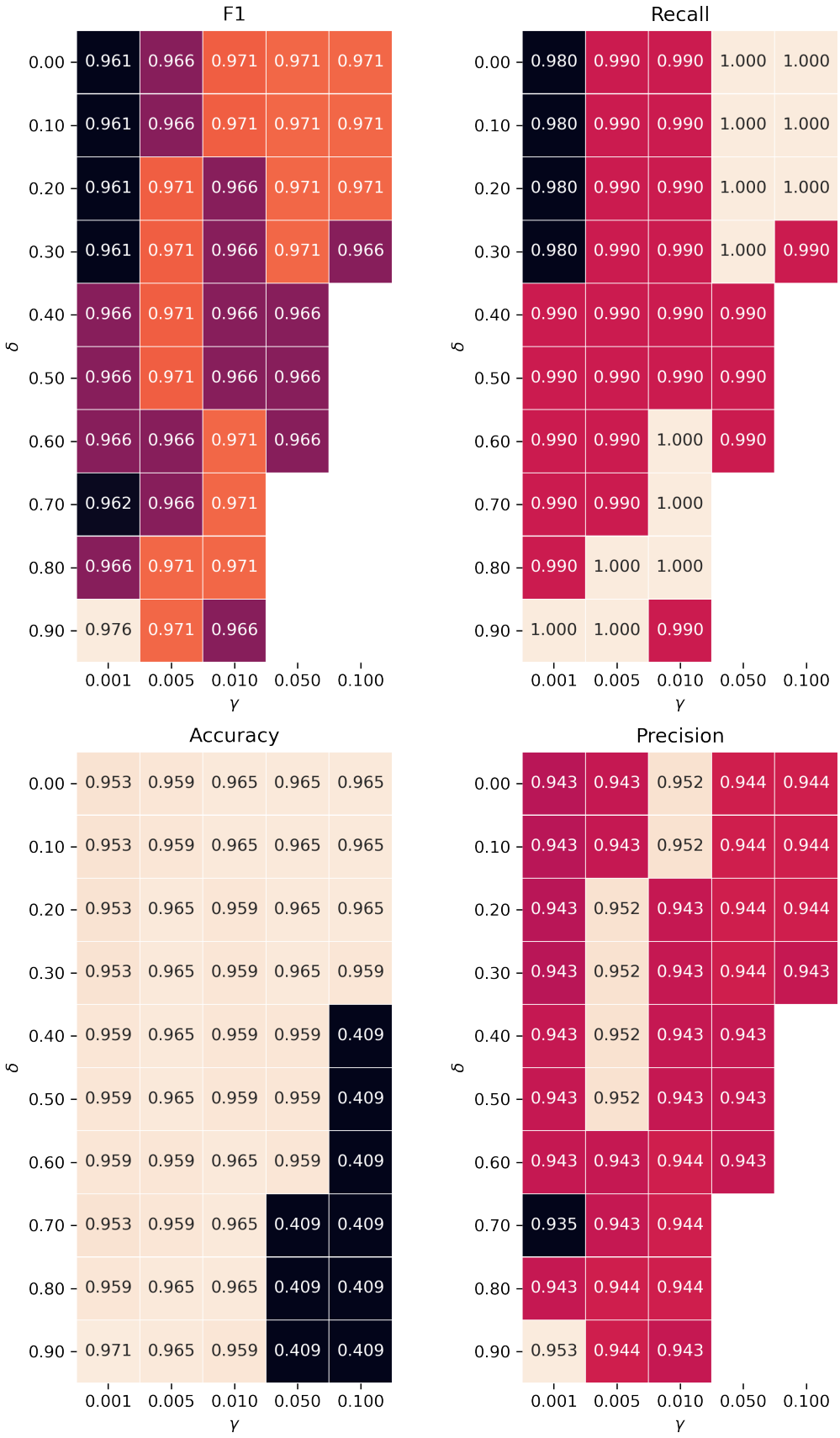

Fig. 14. F1, Recall, Accuracy and Precision scores for the trained logistic regression model. They are computed using the test set with different gamma and sigma values.

OLS and Ridge cases, although the difference with Lasso is minimal. This difference virtually disappears for the first resampling method, the Bootstrap technique, but increases a bit for the CV. Although both methods show very similar results, CV allows us to better explore how the model generalizes, due the different test sets. These method allows us to extract more information from our limited amount of data, becoming more important as we have a smaller amount of data. In this last case, the computational cost to compute the regression is not a problem, being possible to do it with both inverse or pseudo-inverse matric and plain GD descent methods. But, in realistic cases, where we have a very large models or big data, those ways are not affordable. Herein lies the great importance of the GDS. We compared three different methods, Adam, AdaGrad and RMSProp, showing that Adam dominates in the cases where we have more MBs and AdaGrad when we have less MBs. Then, for big size data sets, Adam is the optimal algorithm. Finally, we used a Logistic regression to solve a binary classification problem. We used the F1 score, recall, accuracy and precision and, for all of them, we obtained optimal results for a given parameters δ and γ. Thus, for the case used, a linear regression is sufficient. However, for more complicated cases, we would have to resort to more complex models, such as NN.

8. Code used

In this section we will explain, in a general way, the code developed and used from scratch for this project. For more detailed information, see the explanations in the code itself. These codes can be found in the following GitHub repository: `https://github.com/Ignasi17/CompSci`. We have developed the codes with *Python* v3.8.5 (Van Rossum & Drake 2009). We have assiduously used the Python packages *NumPy* v1.19.2 (Harris et al. 2020), *Scikit-learn* v.0.23.2 (Pedregosa et al. 2011b) and *Matplotlib* v3.3.2 (Hunter 2007). Based on how Scikit is programmed and the fact that Python is an Object-Oriented language, we developed the codes with the same philosophy, making use of classes. We developed two kind of classes, one to data management and, the other ones, represent models that are trainable. These are found in the **modelos.py** file.

class Dataframe:

Class oriented to manage the sets of data used. Each instance of the class works as a one independent set of data. This class allows to manage 1D and 2D data, grid them if it is 2D, split them into train and data sets, create the features matrices and scale the data. To split data and create features matrices, we have used functionalities from Scikit.

class Ridge:

Class to compute the Ridge Regression.

class OLS: Class oriented to compute OLS regression directly or using gradient descent methods. Available methods are the ones presented in the previous sections: GD, GD with momentum, and SGD . Also allows to set the studied learning rate schedules: Adam, Adagrad, and RMSprop. This class has two main methods, *fit* and *predict*. With fit method, the model computes, using the method and scheduler indicated, the optimal parameters for the model. With predict method, it predicts the value for a given feature matrix.

class LogisticRegression: This class is similar to OLS but for Logistic Regression. The differences are internal. See the code for more information.

The code where the project is done using the classes defined in ***modelos.py*** are in the following jupyter files:

- Part a) to e) included, OLS, Ridge and Lasso: **project1_part1.ipynb**.
- Part f), GD methods: **project1_part2.ipynb**.
- Part g), Logistic Regression: **project1_part3.ipynb**.

References

Duchi, J., Hazan, E., & Singer, Y. 2011, Journal of machine learning research, 12

Efron, B. 1979, View Article PubMed/NCBI Google Scholar, 24

Gautschi, W. 2011, Numerical analysis (Springer Science & Business Media)

Goodfellow, I., Bengio, Y., & Courville, A. 2016, Deep Learning (MIT Press)

Guo, C., Pleiss, G., Sun, Y., & Weinberger, K. Q. 2017, in International conference on machine learning, PMLR, 1321–1330

Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, Nature, 585, 357–362

Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. 2009, The elements of statistical learning: data mining, inference, and prediction, Vol. 2 (Springer)

Hinton, G., Srivastava, N., & Swersky, K. 2012, Cited on, 14, 2

Hunter, J. D. 2007, Computing in Science & Engineering, 9, 90

Kingma, D. P., & Ba, J. 2014, arXiv preprint arXiv:1412.6980

Mosteller, F., & Tukey, J. W. 1968, Handbook of social psychology, 2, 80

Pedregosa, F., Varoquaux, G., Gramfort, A., et al. 2011a, Journal of Machine Learning Research, 12, 2825

—. 2011b, Journal of machine learning research, 12, 2825

Polyak, B. 1964, USSR Computational Mathematics and Mathematical Physics, 4, 1

Van Rossum, G., & Drake, F. L. 2009, Python 3 Reference Manual (Scotts Valley, CA: CreateSpace)

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. 2021, arXiv preprint arXiv:2106.11342