Regression analysis and resampling methods

Claudia Fabris[a], Chiara Giraudo[b], Agnieszka Seremak[a],Ana Anzulović[c]

[a]Department of Chemistry, University of Oslo, Norway
[b]Department of Mathematics, University of Oslo, Norway
[c]Department of Geosciences, University of Oslo, Norway

Abstract

We studied numerous regression methods, in order to master the supervised Machine Learning. Our algorithms were used for in depth analysis of Ordinary Least Squares method, Ridge and Lasso Regression as well as few optimization techniques. Additionally, Logistic Regression was applied in classification problem. Linear regression approaches were also coupled with common resampling methods. This study equipped us with better understanding of simple ML algorithms, helped us recognize their weaknesses and strengths. We identify and report here the most accurate approaches for each problem studied.

1 Introduction

One of the most common machine learning (ML) methods is regression analysis. It is a set of statistical approaches used to perform predictive analysis, *i.e.* estimating the relationship between a dependent variable (called *'outcome'*) and one or more independent variables (called *'predictor'*). Regression and classification are the two primary applications of the supervised type of machine learning. Regression is used for predicting the outcome and classification is essentially the categorization of the objects based on learned features. Important component of both approaches is the labeling of the input and output training data since it enables the model to gain better understanding of their relationship. By analyzing this correlation the model can accurately predict the outcome of unseen data. Linear regression is the most commonly used and straightforward regression technique. In this approach the model identifies the linear relationship between the predictor and outcome and trains it to better predict the outcome of a new predictor by finding the most optimal regression parameters. It is important to acknowledge that, despite its name, linear regression can be applied for nonlinear functions. Obtaining the best fit to a regression line can be achieved with various methods. Firstly, by minimizing the cost function, which in linear regression is the mean-squared error between the predicted and true output values. However, this can be computationally expensive approach, as it involves the calculation of matrix inverse. In order to improve the model performance it is prevalent to use resampling methods. Two of them are the most recognized and accepted: bootstrap and cross-validation. A second sets of methods, called optimization algorithms, can overcome this issue and are usually applied for large datasets. The strengths of these tools is the possibility to continuously update the regression parameters whilst minimizing the cost function. The most popular algorithms are Gradient Descent, Stochastic Gradient Descent, Adagrad, RMSProp and ADAM. Additionally, all the above mentioned methods could be improved with regularization techniques, like Ridge and Lasso Regression. Another algorithm used in supervised ML is logistic regression, a type of classification technique. As the name indicates, it employs a logistic function to model the dependent variable. This technique is applied for binary data. To quantify the error between the predicted outcome and true values, the cost function is calculated. Usually, it is represented by cross-entropy. With this algorithm, optimization methods (*i.e.* (Stochastic) Gradient Descent) can be implemented. In this work we present an investigation for the most optimal regression analysis method. We tested the linear regression method and expanded it with resampling methods and regularization techniques. Additionally, we compared numerous optimization methods and applied some of them in logistic regression. In the linear regression study we used Franke's bivariate function to train and test our model. For optimization methods we chose a polynomial of third order to compare among various techniques. Finally, in classification problem, where we tested logistic regression, a famous Wisconsin breast cancer dataset was employed. The methodology section is based on the general theory mainly presented in the lecture notes [1]. For the linear regression part, we referred to [1]. Regarding the classification part, we referenced [2, Chapter 2]. Our results show the differences between these approaches as well as provide an answer to which of these methods is the most suitable for the chosen examples. The link to the codes can be found in the Supplementary Files.

2 Methodologies

2.1 Linear Regression

In this study we have p characteristics of n samples. The data set consists of inputs, denoted $\boldsymbol{X}$ and called *design matrix* and the corresponding outcome, denoted $\boldsymbol{y}$. The goal of regression analysis is to find a functional relationship between y and X. Having a vector of inputs $X^T = (X_1, X_2, ..., X_p)$ and assuming that this relationship is linear we can define the *linear regression model* of the form:

$$\hat{y} = \hat{\beta_0} + \sum_{j=1}^{p} X_j \hat{\beta_j} \tag{1}$$

where $\hat{\beta_0}$ is the intercept and $\hat{\beta_j}$ are unknown *regression parameters*.
In order to simplify the form of the model, it is useful to include the intercept β_0 in the vector of regression parameters and therefore also include it in X. Writing this model in the vector form gives us:

$$\hat{y} = \mathbf{X}^T \hat{\beta} \tag{2}$$

with $y^{\,n\times 1}$, $\mathbf{X}^{\,n\times p}$, and $\hat{\beta}^{\,p\times 1}$. Since we are looking for a function $f(\mathbf{X})$ for predicting y given the input X, it is essential to include a *cost function* C in order to analyze the correctness of the model. In this work we use the *mean squared error* (MSE) to compare the predicted values with the actual values.

$$C(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i^T\beta)^2 \tag{3}$$

The overall aim is to find parameters $\hat{\beta}$ that minimize the MSE: $\hat{\beta} = \arg\min C(\beta)$ Rewriting Equation (3) in matrix notation, we get:

$$C(\beta) = \frac{1}{n}\{(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta)\} \tag{4}$$

To find global minimum of the cost function it is necessary to differentiate w.r.t. β:

$$\frac{\partial C(\beta)}{\partial \beta} = \mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0 \tag{5}$$

When $\mathbf{X}^T\mathbf{X}$ is non-singular (positive definite), then the solution for optimal regression parameters is:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{6}$$

However, if $\mathbf{X}^T\mathbf{X}$ is singular or near singular then the $\hat{\beta}$ coefficients are not uniquely defined. In order to avoid this problem we calculate the pseudo-inverse of the matrix, i.e. the Moore-Penrose inverse. In this approach the generalized inverse of a matrix is computed using the *singular value decomposition* (SVD) algorithm. In order to validate our model, an important step is data splitting. A given dataset is divided into two disjoint sets: a *training* set and a *test* set. The machine learning model is then fitted on the training set and validated using the test set. By separating the test dataset from the training, we can evaluate and compare the predictive performance of different models without worrying about possible overfitting on the training set. In our study we use 80% of data as training data and remaining 20% for validation. Another element to consider is data scaling. Assuming that the data differ, the machine learning algorithm would perform poorly. In order to avoid this issue, many scaling functions exist. In our work we applied the **StandardScaler** function from **Scikit-Learn** library [3], which ensures that for each studied feature, the mean value is zero and the variance is one.

Ridge and Lasso regression Linear regression does not perform well with complex models and, furthermore, the models might be prone to overfitting. A common way to overcome these limitations is using regularized regression techniques, *i.e.* Ridge and Lasso regression. They allow for the regularization of β coefficients by imposing a regularization hyper-parameter called λ. Essentially, the estimated β coefficients are minimized and pushed towards 0. The Ridge hyper-parameter controls the regularization strength. Here the L^2-norm of the β coefficient vector is penalized. As a result a modified cost function is obtained of the form:

$$C_{Ridge}(\beta) = \frac{1}{n}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\|\beta\|_2^2 \tag{7}$$

The Lasso hyper-parameter, in contrast, reduces the number of features upon which the given solution is dependent, *i.e.* it sets the desired sparseness of the results. It solves the minimization of the least squares penalty with regularization term λ and L_1-norm of the β coefficient. The cost function with the Lasso hyper-parameter is as follows:

$$C_{Ridge}(\beta) = \frac{1}{n}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\|\beta\|_1 \tag{8}$$

Resampling Resampling methods are used to gain additional information about a fitted model. First the starting dataset is split into training and testing sets. Data is then repetitively drawn from the training sets to train the model which is then refitted by only using the test datasets [1]. They are useful for dealing with imbalanced datasets or reducing the variance of a model. Common resampling methods include bootstrapping, cross-validation, and the jackknife method.

Bootstrap Bootstrapping is used to estimate the accuracy of a model on unseen data. Bootstrapping works by resampling a given dataset with replacement, creating multiple datasets of the same size as the original. Each of these datasets is then used to train and test a model, and the results are averaged over all of the datasets. This process can be repeated multiple times to further improve the accuracy of the estimates. Bootstrapping is a powerful method for reducing the variance of accuracy estimates, enabling models to be evaluated more accurately, while allowing to approximate a sampling distribution from just one sample [4]. The pseudo-code of this resampling technique is shown in Algorithm 1.

Algorithm 1 Bootstrap

Require: $n_{bootstrap}$: number of bootstraps
Require: *degree* : degree of the polynomial
Require: x : input data
Require: y : output data
  $x_{train}, x_{test}, y_{train}, y_{test} \leftarrow$ split the data (train_test_split from **Scikit-Learn**)
  $\tilde{y} \leftarrow 0$ (initialise empty prediction matrix of dimension: $size(y_{test}) \times n_{bootstrap}$)
  for all the numbers in the range of $n_{boostraps}$ **do**
    $x_{resample}, y_{resample} \leftarrow$ resample x_{train} and y_{train}
    $X_{train} \leftarrow$ compute the design matrix for resampled training data
    $\hat{\beta} \leftarrow$ compute the optimal β
    $X_{test} \leftarrow$ compute the design matrix for resampled test data
    $\tilde{y} \leftarrow$ compute the predicted test output
  end for
  return $\tilde{y}$

Cross-validation Cross-validation involves randomly splitting the dataset into training and testing sets for every iteration (*k-fold*). The selected model is then trained on the training set and its performance is evaluated on the testing set. This process is repeated multiple times with different splits of the dataset, to get a better estimate of the model's performance. A graphical representation of this resampling method is shown in Figure 1. Cross-validation is a powerful tool for reducing the bias of a model and for providing a good estimate of its generalization performance on unseen data.

2.2 Optimization algorithms

Optimization algorithms are key for deep learning. They allow to continuously update the model parameters and to minimize the loss function value. The tuning of the hyper-parameters allows to improve the performance of deep learning models. It should be noted that the goal of the optimization and deep learning are different since the first aims at minimizing a certain target, whereas the second focuses on finding a suitable model, starting from a finite amount of data.

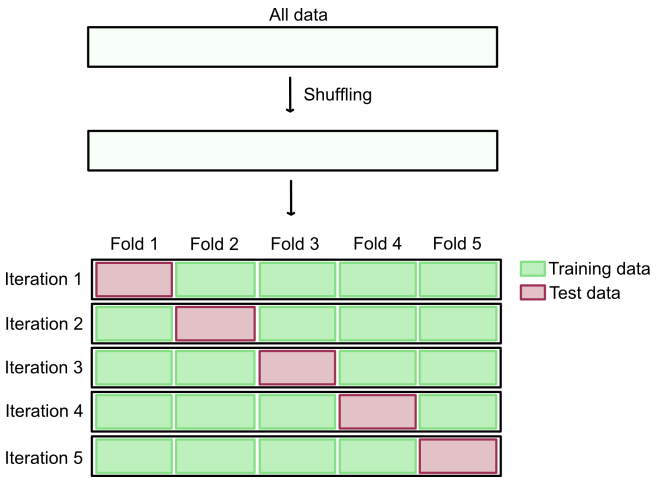


Figure 1: *Graphical representation of the cross validation resampling method: the starting dataset is split into a set of training and test sets which change with every iteration.*

Gradient Descent Optimization is a process of iteratively improving the accuracy of a machine learning model. Gradient descent is an algorithm used for optimization or minimization of the cost function, and to tune it we used the learning rate (η), number of iterations (also known as number of epochs) and number of batches. We selected three values for the learning rate, which represents the step size of iteration process to reach a minimum. We analyze the output with learning rates 0.08, 0.1 and 0.2. Momentum is an extension to the GD optimization method that builds inactivity in a search direction to overcome local minima. Momentum adds history to the parameter updates of origin problems and that substantially accelerate the optimization. This "history" amount is determined with the hyper-parameter δ, with value ranging from 0 to 1. A higher δ indicates that more gradients from the past are considered.

Stochastic gradient descent with minibatches (SGD) is an iterative method for optimizing a differentiable objective function, which is a function that can be minimized or maximized. SGD operates by calculating the gradient of the objective function with respect to the parameters and then updating the parameters in the opposite direction of the gradient with a small step size. In Stochastic Gradient Descent only one example is considered at a time to take a single step. Therefore, SGD is recommended for a large dataset, as it can significantly reduce the computational costs. In SGD with minibatches, a small subsets (batches) of a fixed number of training examples which are smaller than the actual dataset, are used.

AdaGrad Features in a model may occur infrequently and the parameters associated with them receive only meaningful updates. With a decreasing learning rate, the parameters related to common features quickly reach their optimal values, whereas the uncommon features require a longer time as they must be observed frequently enough to determine the optimal values. To try and solve this issue, one can count the number of times a certain feature is observed and use it as a timer to update the learning rates. AdaGrad by Duchi et. al. [5] solves this by replacing the counter with an aggregate of the squares of the gradients which were previously observed to adjust the learning rates. This allows to automatically scale the magnitude of the gradients and it is no longer required to decide when a gradient is large enough. The algorithm permits to scale down the coordinates that correspond to large gradients, whereas those with small gradients have a milder scaling. The algorithm is reported in Equation (9), where the variable s_t allows to accumulate the variance of the past gradient.

$$\begin{aligned} \mathbf{g}_t &= \partial l(y_t, f(x_t, \mathbf{w})), \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot \mathbf{g}_t \end{aligned} \tag{9}$$

RMSProp The coordinatewise-adaptivity of AdaGrad is a very desirable pre-conditioner, but the defined schedule of the learning rate is only appropriate for convex problems. This is not adequate for non-convex ones, which are generally found in deep learning. The RMSProp algorithm (10), proposed by Tieleman and Hinton [6] decoupled the scheduling rate from the coordinate-adaptive learning rates. AdaGrad collected the square of the gradient $\mathbf{g}_t$ in a state vector $\mathbf{s}_t$, which continuously grows without limit due to the lack of normalization. This problem is solved by using a leaky average which contains a parameter $\gamma > 0$.

$$\begin{aligned} \mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot \mathbf{g}_t \end{aligned} \tag{10}$$

The constant value $\epsilon > 0$ is generally set to 10^{-6} to avoid any division by zero or large step sizes. Thanks to this expansion, the learning rate η is independent of the scaling of the per-coordinates.

ADAM Stochastic gradient descent is a more effective optimization algorithm than Gradient descent thanks to its resilience to repeating data. The addition of minibatches makes the algorithm more efficient, whereas the presence of the momentum adds a memory term, so the history of the past gradients is aggregated to accelerate convergence. Finally, AdaGrad added a computationally efficient pre-conditioner, whereas RMSProp decoupled the learning rate adjustment from the per-coordinate scaling. The ADAM algorithm [7] combines these techniques to obtain an efficient learning algorithm. The estimate of the momentum and the second moment of the gradient are estimated by using weighted moving averages as follows:

$$\begin{aligned} \mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2\, \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \end{aligned} \tag{11}$$

β_1 and β_2 are non-negative weighting parameters with values usually equal to 0.9 and 0.999, respectively so that the variance moves at a much slower rate than the momentum term. These terms are normalized since starting with $v_t, s_t = 0$ would mean a high bias towards smaller values initially. The normalized variables are:

$$\begin{aligned} \hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1-\beta_1^t}, \\ \hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1-\beta_2^t} \end{aligned} \tag{12}$$

Similarly to the rescaling of the gradient of the RMSProp algorithm, we write:

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon} \tag{13}$$

The update is given by the momentum instead of the gradient itself, like in RMSProp and it can be written as:

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t' \tag{14}$$

It should be noted that thanks to the explicitation of the learning rate η, the step length can be controlled to address the issues related to convergence.

2.3 Classification

In the previous sections, we described methods for approximating continuous functions, where the output is a continuous interval in $\mathbb{R}$. However, in classification problems, the goal is to group the features into a finite number of classes, meaning that the outcome is discrete. In our work, we focus on binary classification, where the output variable y can take on the values of either 1 (true) or 0 (false). Also in this case, we use $\mathbf{x} \in \mathbb{R}^{d \times 1}$ to denote an observation and y to represent its corresponding outcome. The regression model is now given by a distribution of probability $p(\mathbf{x})$, $0 \leq p(\mathbf{x}) \leq 1$. The choice of the probability function affects the final results. Since we have no a priori information about the data, for simplicity we choose the sigmoid function:

$$p(\mathbf{x}) = \frac{1}{1 + \exp^{-(\beta_0 + \mathbf{x}^T \boldsymbol{\beta})}}, \tag{15}$$

where $\beta_0 \in \mathbb{R}$ is the intercept and $\boldsymbol{\beta} \in \mathbb{R}^d$ is the coefficient vector related to the features of the problem. The vector $(\beta_0, \boldsymbol{\beta})$ is the regression unknown parameter. The function $p(\mathbf{x})$ represents the probability of $\mathbf{x}$ being classified in class 1, while $(1 - p(\mathbf{x}))$ represents the probability of $\mathbf{x}$ being in class 0. A threshold parameter, called decision boundary, is required to convert the predicted probability into a class label. If $p(\mathbf{x})$ is above the threshold, it indicates one class, while if it is below the threshold, it indicates the other class. The decision rule is then given by:

$$\begin{cases} y = 0 & \text{if } p(\mathbf{x}) < k \\ y = 1 & \text{if } p(\mathbf{x}) \geq k\,, \end{cases} \tag{16}$$

where $k \in (0, 1)$ is the threshold. In order to use logistic regression, the following assumptions must be satisfied:

- the output (dependent variable) is discrete;
- the samples or observations (inputs) are independent;
- the independent variables (inputs) are not highly correlated;
- the relationship between the independent variables is linear and the probability is logarithmic.

As with linear regression, in logistic regression we need to minimize a cost function C to tune the regression parameters $(\beta_0, \boldsymbol{\beta})$ and find its optimal values. In this work, the cost function is represented by the cross-entropy:

$$C(\beta_0, \boldsymbol{\beta}) = -\frac{1}{n} \sum_{i=1}^{n} \left(y_i \log \tilde{y}_i + (1 - \tilde{y}_i) \log (1 - \tilde{y}_i) \right), \tag{17}$$

where $y_i \in \{0, 1\}$ is the given output label and $\tilde{y}_i = p(\mathbf{x}_i)$ is the predicted probability output, with $i = 1, .., n$. First of all, we compute the cross entropy gradient with respect to the regression parameters $(\beta_0, \boldsymbol{\beta})$:

$$\nabla_{(\beta_0, \boldsymbol{\beta})} C = \left(\frac{\partial C}{\partial \beta_0}, \frac{\partial C}{\partial \beta_1}, .., \frac{\partial C}{\partial \beta_d} \right) \tag{18}$$

where

$$\frac{\partial C}{\partial \beta_0} = \frac{1}{n} \sum_{i=1}^{n} (\tilde{y}_i - y_i)\,, \tag{19}$$

$$\frac{\partial C}{\partial \beta_j} = \frac{1}{n} \sum_{i=1}^{n} x_i^{(j)} (\tilde{y}_i - y_i)\,, \qquad j = 1, .., d. \tag{20}$$

If we define the design matrix taking into account the intercept, i.e.:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d)} \\ 1 & x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d)} \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & x_n^{(1)} & x_n^{(2)} & \cdots & x_n^{(d)} \end{bmatrix}, \tag{21}$$

then the cross entropy gradient (18) can be rewritten in the following matrix form:

$$\nabla_{(\beta_0, \boldsymbol{\beta})} C = \frac{1}{n} \mathbf{X}^T (\tilde{\mathbf{y}} - \mathbf{y})\,, \tag{22}$$

where

$$\tilde{\mathbf{y}} = \begin{bmatrix} p(\mathbf{x}_1) \\ p(\mathbf{x}_2) \\ \vdots \\ p(\mathbf{x}_n) \end{bmatrix} \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

To clarify the notation used in Equation (21), consider a dataset with n samples and d features. In this context, each $\mathbf{x}_i$ corresponds to the i^{th} observation in the dataset, while $x_i^{(j)}$ denotes the value of the j^{th} feature for that observation, where $j = 1, \ldots, d$. Equation (22) shows that there is no closed-form solution for the optimal weights $(\hat{\beta}_0, \hat{\boldsymbol{\beta}})$ such that

$$\nabla_{(\beta_0, \boldsymbol{\beta})} C(\hat{\beta}_0, \hat{\boldsymbol{\beta}}) = 0\,.$$

Therefore, we use gradient descent and stochastic gradient descent with minibatches as alternative optimization methods. Let η be the learning rate, the $(k+1)^{th}$ iteration of gradient descent with fixed learning rate, is given by

$$(\beta_0, \boldsymbol{\beta})^{k+1} = (\beta_0, \boldsymbol{\beta})^k - \eta_k \mathbf{X}^T (\tilde{\mathbf{y}} - \mathbf{y})\,. \tag{23}$$

When using stochastic gradient descent, the $(k+1)^{th}$ iteration is still described by Equation (23). However, in this case, the learning rate changes at each iteration. As with linear regression, the overfitting problem in logistic regression can be mitigated by adding a regularization term λ to the cost function. For example, using L^2 regularization modifies the cost function as:

$$\begin{aligned} C(\beta_0, \boldsymbol{\beta}) = &-\frac{1}{n} \sum_{i=1}^{n} \left(y_i \log \tilde{y}_i + (1 - \tilde{y}_i) \log (1 - \tilde{y}_i) \right) \\ &+ \frac{\lambda}{2n} \sum_{j=0}^{d} \beta_j^2\,, \end{aligned} \tag{24}$$

and as a result, the gradient is given by:

$$\nabla_{(\beta_0, \boldsymbol{\beta})} C = \frac{1}{n} \left[\mathbf{X}^T (\tilde{\mathbf{y}} - \mathbf{y}) + \lambda (\beta_0, \boldsymbol{\beta}) \right]. \tag{25}$$

To find the optimal λ, the cross-validation technique can be used. The λ found is optimal in the sense that, by iteratively trying different parameter values, the one with the best performance across all subsets is selected.

After training the model, it is necessary to assess its performance. As highlighted in [2], the choice

of appropriate performance measures may vary depending on the classification task and the available data. In the following, we list the most commonly used measures.

Accuracy. Accuracy measures the proportion of correct predictions out of the total predictions. A higher accuracy generally indicates a better model, but accuracy alone is not sufficient for evaluating model performance. For example, when one class is much more prevalent than the other, a model that always predicts the majority class can still achieve high accuracy, despite not being useful. Therefore, other measures, such as confusion matrix, precision, recall, and F1-score, should also be taken into account.

Confusion matrix. In binary classification, the class associated with the value 1 is commonly referred to as the *positive* class, while the one associated with the value 0 is referred to as the *negative* class. With this in mind, the confusion matrix takes the following form:

$$\begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix}, \tag{26}$$

where the first column is relative to the negative class, and the second column is relative to the positive class. Specifically,

- TN (True Negative) represents the samples correctly classified as 0;
- FP (False Positive) represents the samples wrongly classified as 1;
- FN (False Negative) represents the samples wrongly classified as 0;
- TP (True Positive) represents the samples correctly classified as 1.

Although the confusion matrix provides detailed information, precision and recall offer a more concise way to evaluate performance.

Precision and Recall. Precision measures the accuracy of positive predictions, while recall represents the proportion of correct predictions made by the model for samples belonging to the positive class. They are mathematically defined as:

$$\text{precision} = \frac{TP}{TP + FP}, \tag{27}$$

$$\text{recall} = \frac{TP}{TP + FN}. \tag{28}$$

It is important to compute both precision and recall, as one alone does not provide enough information. Precision and recall are inversely proportional; increasing precision tends to reduce recall, and vice versa. For example, a model that classifies correctly only one positive sample will have precision equal to 1, but a very small recall (close to 0). In general, the precision/recall trade-off analysis can be used to choose the threshold hyper-parameter introduced in Equation (16). However, in our work, we fixed the value of this hyper-parameter without performing the precision/recall trade-off analysis.

F1 score. The F1 score is a single measure that combines precision and recall. It can be computed using the following formula:

$$\text{F1 score} = \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}}. \tag{29}$$

The F1 score favors classifiers with similar precision and recall values. However, it is not always the best choice to balance precision and recall, as it depends on the problem at hand. For instance, in our study of the Wisconsin breast cancer dataset, the objective is to determine whether a tumor is malignant (positive class) or benign (negative class). In this scenario, it is preferable to have a false positive than to misclassify a malignant tumor as benign. Therefore, we may favor models with slightly higher recall than precision.

3 Dataset Description

3.1 Franke function

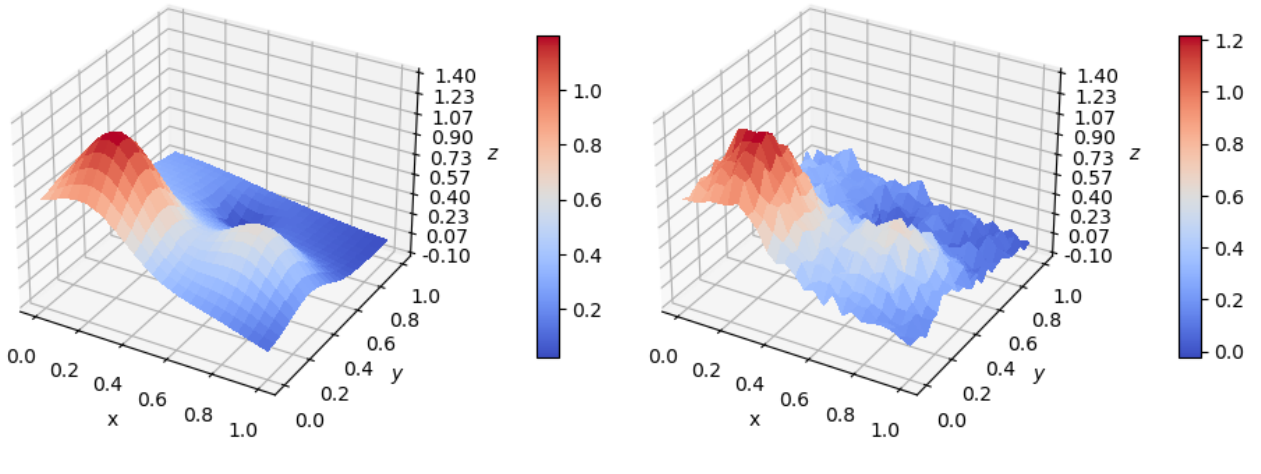
The aim of the first part of this work was to fit a 2D polynomial to the Franke function with stochastic Gaussian noise. The function is defined as:

$$\begin{aligned} f(x_1, x_2) := &\frac{3}{4}\exp\left(-\frac{(9x_1-2)^2}{4} - \frac{(9x_2-2)^2}{4}\right) \\ &+\frac{3}{4}\exp\left(-\frac{(9x_1+1)^2}{49} - \frac{(9x_2+1)}{10}\right) \\ &+\frac{1}{2}\exp\left(-\frac{(9x_1-7)^2}{4} - \frac{(9x_2-3)^2}{4}\right) \\ &-\frac{1}{5}\exp\left(-(9x_1-4)^2 - (9x_2-7)^2\right) \end{aligned} \tag{30}$$

In order to better represent the real data collection, we introduced random noise. It was sampled from a Gaussian distribution $\mathcal{N}(0, var)$ and added to the above mentioned Franke function. The plots of the Franke function with and without noise are shown in Figure 2a and Figure 2b, respectively. The dataset consists of 50×50 data-points and the noise coefficient was set to 0.05.

3.2 Wisconsin breast cancer data set

The Wisconsin breast cancer data [8] was used to study a classification problem with logistic regression. Each attribute had a diagnosis (M = malignant, B = benign) and list of 30 features for each cell nucleus including radius, texture, smoothness, and



(a) **(b)**

Figure 2: *Graphical representation of the Franke function a) without and b) with noise.*

symmetry. According to the information provided with the data set, the class distribution is 357 benign and 212 malignant tumors.

In order to better visualize the statistical data used in this part of the work, the **seaborn** package was used [9].

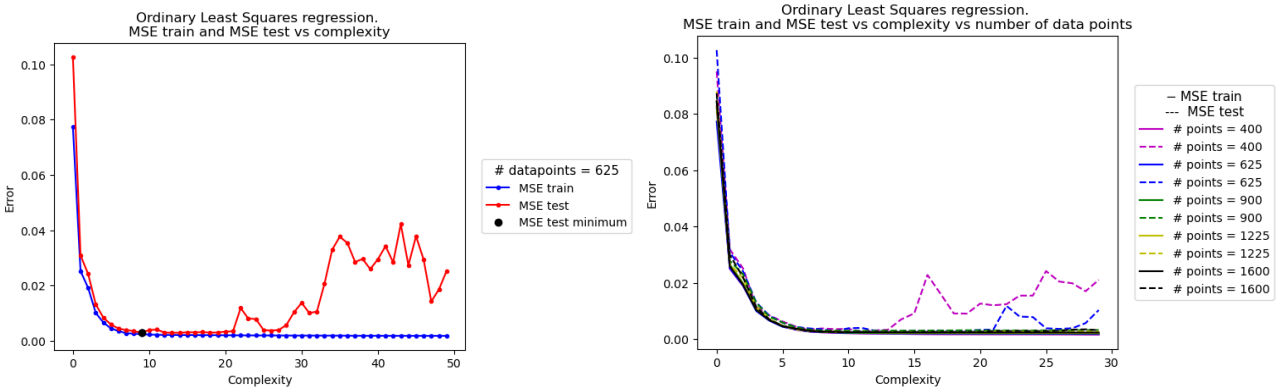
4 Results and Discussion

4.1 OLS and resampling techniques

In order to determine the model that best fit the given function, the model complexity was varied and the MSE was studied while keeping the number of data points fixed. The minimum MSE for the test dataset was obtained with a polynomial of degree 9, as reported in Figure 3a. Moreover, the optimal number of data points was also evaluated by maintaining the model complexity to the optimal value previously found. As Figure 3b shows, the smallest MSE was achieved with 625 data points, a polynomial degree of 9, and was equal to $2.87 * 10^{-3}$. With the same conditions we also evaluated the R^2 score which was equal to $9.71 * 10^{-1}$. It should also be noted that no major differences in the MSE value were highlighted for a higher number of data points. To study the bias-variance trade-off, the dataset was split using the bootstrap resampling technique into 500 bootstrap sets and it was studied with respect to the number of data points, the model complexity and the number of data points. By varying the number of data points, it was observed that no big difference was present after 900 data points. On the other hand, it was noted that by increasing the number of data points from 400 to 625, a large improvement in the mean MSE value was observed. Moreover, the higher model complexity did not reduce the MSE but, as expected, lead to overfitting (Figure 4). Similarly to what was done for the bootstrap method, the cross-validation technique was applied to various amounts of data points. The number of k-folds was set to 7 for all the subsequent analyses. From 625 data points used, no major difference in the mean MSE could be discerned. Figure 5 shows the mean MSE, with respect to the polynomial degree, evaluated when the cross-validation resampling technique was used. Even though the difference in the mean MSE was not major, above a polynomial degree of 5, the MSE was always below 0.01 until the model complexity of 40. In order to understand the difference between the two resampling techniques, the mean MSE derived from both methods was plot with respect to the polynomial degree. Figure 6 shows that cross-validation yields a lower mean MSE along a larger range of model complexity with respect to the bootstrap technique, which showed an increase in the mean MSE value above a polynomial degree of 10.

4.2 Ridge regression

In Ridge regression we introduced the parameter λ which serves as a penalty to the traditional least squares method. Such parameter must be tuned and for this reason various λ values were selected and Ridge regression was performed. Figure 7 shows the different trials carried out to find the regularization parameter which lead to the minimum MSE values of the test dataset also as a function of the model complexity. The number of data points was set to 625, as it was previously confirmed that this value was a good trade-off between optimal MSE and calculation speed. The smallest test mean squared error was $3.63 * 10^{-3}$ with a polynomial degree of 16 and



(a) **(b)**

Figure 3: *Mean squared error comparison for OLS with respect to a) model complexity, and b) number of data points*

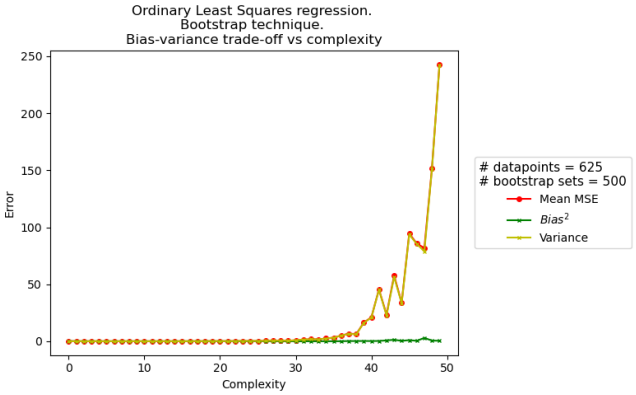


Figure 4: *Graphical representation of the bias-variance trade-off by implementation of the bootstrap resampling technique with respect to the model complexity.*

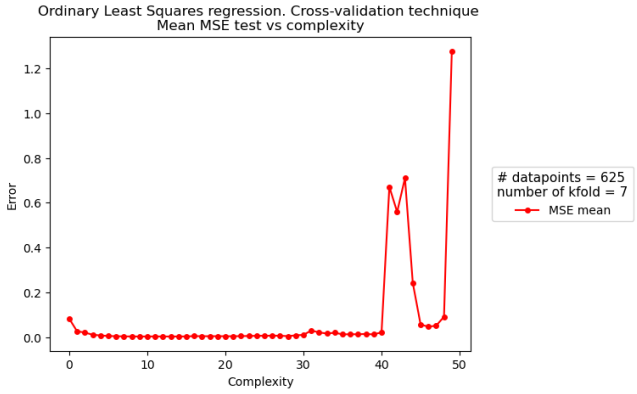


Figure 5: *Graphical representation of the variation of the MSE with respect to the model complexity.*

a regularization term λ equal to 0.0001. It was also highlighted that as the regularization parameter increased, the MSE increased for both the training and testing datasets, regardless of the model complexity. The R^2 score was also evaluated and the maximum value was obtained for the same conditions stated above and had a maximum value of $9.63 * 10^{-1}$. Ridge regression was then performed with respect to the regularization parameter λ by fixing the model complexity to 16 and 625 data points were considered. When fixing the degree of the polynomial, the smallest MSE was $3.56 * 10^{-3}$ and was obtained with a regularization term of $4.43 * 10^{-5}$. Similarly to what was previously done, the R^2 score was evaluated and the maximum value was obtained at the same conditions with a value of $9.64 * 10^{-1}$. The bootstrap technique was also applied to Ridge regression with 500 bootstrap sets and 625 data points. The effect of the regularization parameter λ and the model complexity were investigated. Figure 9 shows how the regularization parameter affects the baias-variance trade-off using the boostrap resampling technique.

It can be observed that lower values of λ minimize the mean MSE for a model complexity between 5 and 20, and a large increase can be observed above this threshold. On the other hand, higher values of the λ hyper-parameter lead to a larger mean MSE in the same polynomial degree range, but showed the same large increase in its value at higher model complexities (above 40) with respect to the aforementioned case. The same analysis was also performed using the cross-validation resampling technique as reported in Figure 10. The same trend was observed regarding the hyper-parameter λ which showed the best values of the mean MSE for the lowest value ($\lambda = 0.0001$).

4.3 Lasso regression

Similarly to what was performed for Ridge regression, Lasso regression presents a regularization parameter λ which had to be tuned. The degree of the polynomial and the number of data points in this analysis were also fixed to 16 and 625, respectively. In Figure 11, the MSE of the train and test datasets were plotted against the regularization parameter.

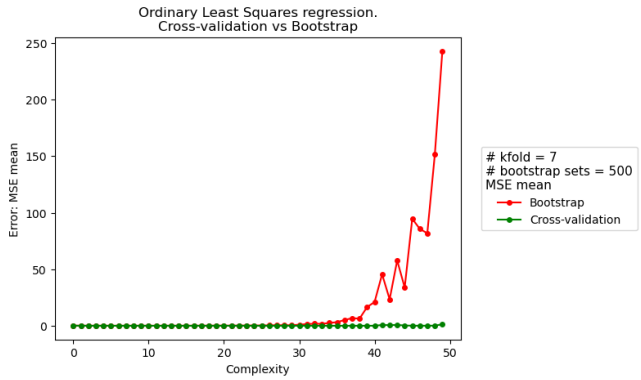


Figure 6: *Comparison between the resampling techniques used in this work: bootstrap and cross-validation.*

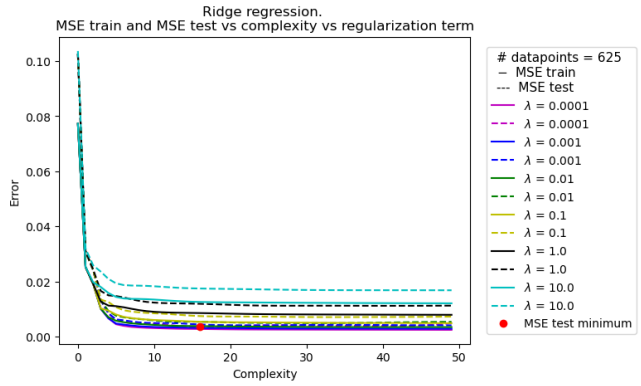


Figure 7: *Mean squared error comparison for Ridge regression with respect to the model complexity and the regularization parameter λ.*

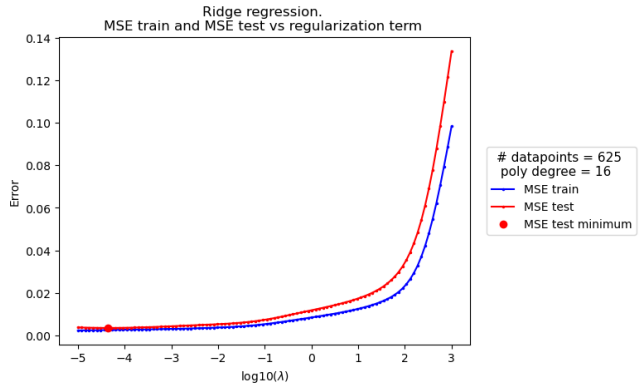


Figure 8: *Mean squared error calculated from Ridge regression with respect to the regularization parameter λ using 625 data points and a model complexity of 16.*

The minimum of the MSE of the test dataset was also highlighted in the plot. With 625 data points and polynomial degree of 16, the smallest mean squared error was $4.846 * 10^{-3}$ and it was obtained with regularization term λ equal to $1 * 10^{-5}$, while the greatest R^2 is $9.508 * 10^{-1}$ and it was obtained with regularization term λ equal to $1 * 10^{-5}$. The resampling techniques were applied also in this analysis. The

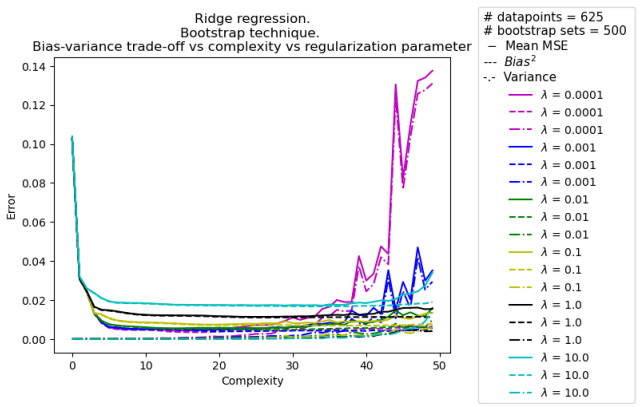


Figure 9: *Bias-variance trade-off of Ridge regression using the boostrap resampling method with respect to the model complexity and the regularization parameter λ using 625 data points.*

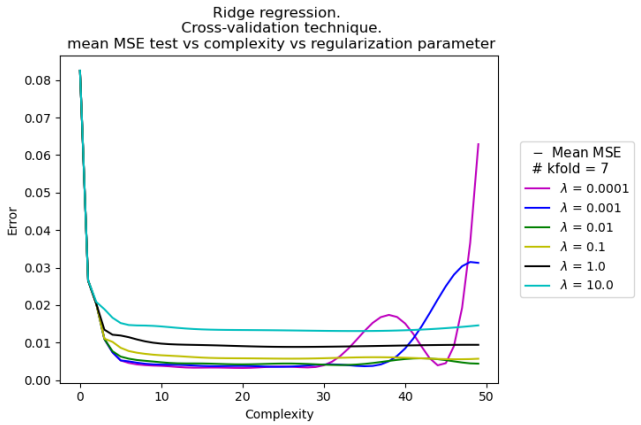


Figure 10: *Mean MSE of Ridge regression using the cross-validation method with respect to the model complexity and the regularization parameter λ using 7 k-folds.*

bootstrap analysis was used first and the different MSEs, including the square of the bias and the variance, were calculated for different values of the polynomial degree and the regularization parameter λ, as shown in Figure 12. Above a model complexity of 5, the MSE changes are negligible. Additionally, the lower the regularization parameter λ, the lower the MSE with a minimum value of approximately 0.01. Such precision come with a trade-off: the computation time increases considerably as the value of λ decreases. Specifically, for the lowest tested value of λ (0.0001), almost 22 hours were necessary to obtain the results. The cross-validation method was also employed to analyze our data using 7 k-folds. The value of the mean MSE was plotted once again with respect to the model complexity, while modifying the regularization parameter λ, as shown in Figure 13. The behaviour of the mean MSE using the cross-validation algorithm shows a similar behaviour to the boostrap method. Although the mean MSE values are similar to those from the previous method,

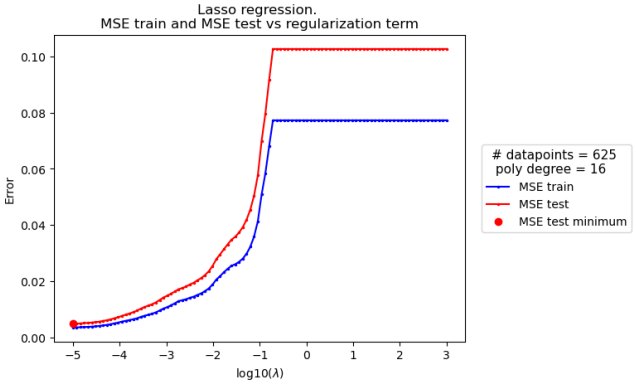


Figure 11: *Mean MSE of Lasso regression of both train and test datasets.*

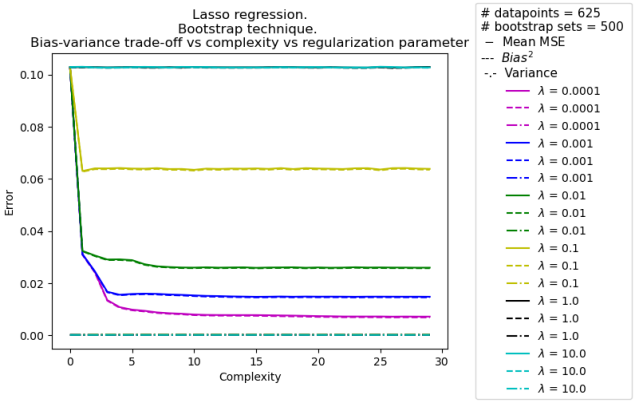


Figure 12: *Mean MSE of Lasso regression calculated with the bootstrap resampling technique. The analysis was performed using 500 bootstrap sets and 625 data points for different model complexities and values of the regularization parameter λ.*

they are reached with slightly lower values of the model complexity (4 instead of 5). The lower the values of λ, the higher the precision, similarly to what observed before. The highest tested λ, 10, yielded higher values of the mean MSE with respect to the bootstrap technique. It should also be noted that the time requested to calculate the mean MSE for the lowest λ (0.0001) was much more efficient than in the previous method as the calculations were done in approximately 40 minutes.
The three different regression methods studied so far were compared by plotting the MSE of the test datasets using 625 data points, as shown in Figure 14. The regularization parameter for Ridge and Lasso regression was fixed for both to 0.0001. The three methods show relatively similar trends up to a value of model complexity of 20. Above this value, the OLS method showed an increase in the values of the MSE, whereas both Ridge and Lasso regression maintain low values of the MSE. The lowest MSE was yielded by Ridge regression.

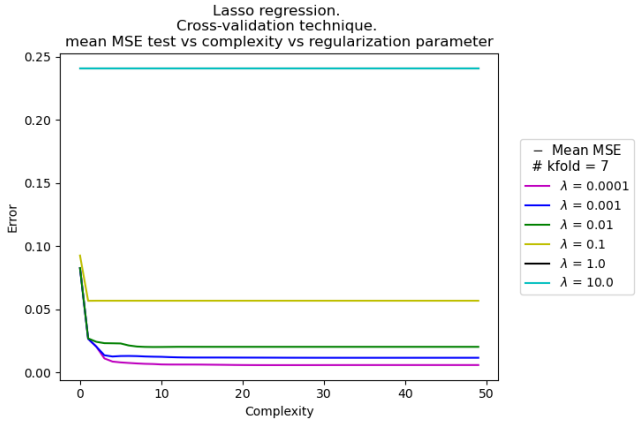


Figure 13: *Mean MSE of Lasso regression calculated with the cross-validation resampling technique. The analysis was performed using 7 k-folds for different model complexities and values of the regularization parameter λ.*

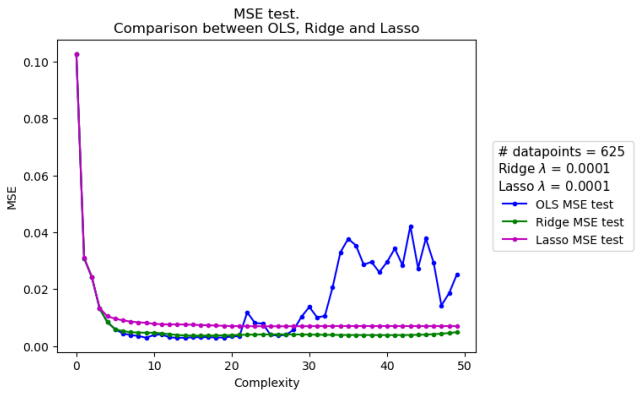


Figure 14: *Comparison between MSE of the test dataset for the three different linear regression methods: OLS, Ridge, and Lasso.*

4.4 Optimization algorithms

Gradient Descent with and without momentum To test our algorithm we firstly generated our training dataset. It consists of 1000 data points from the interval $[-3, 3]$ and our function to fit is:

$$y = -6 - 11x + 3x^2 + 2x^3 + \epsilon \tag{31}$$

where ϵ is a random noise, sampled from Gaussian distribution. For testing our model we choose 430 data points. In order to apply Gradient Descent (GD) we need an additional hyper-parameter: the learning rate η, which determines the step size at each iteration. The first step was the data scaling. We performed the standardization process using the **Scikit-learn** object **StandardScaler** [3]. It is important to realize the consequence of skipping the preprocessing data. Figure 15 present fitting the test data to the function introduced in Equation (31) on the non-scaled and scaled data with number of iteration steps set to 5000 and 50000 and with stopping criterion of 10^{-9}. The OLS (also named Matrix inversion) are the results for analytical calculations using

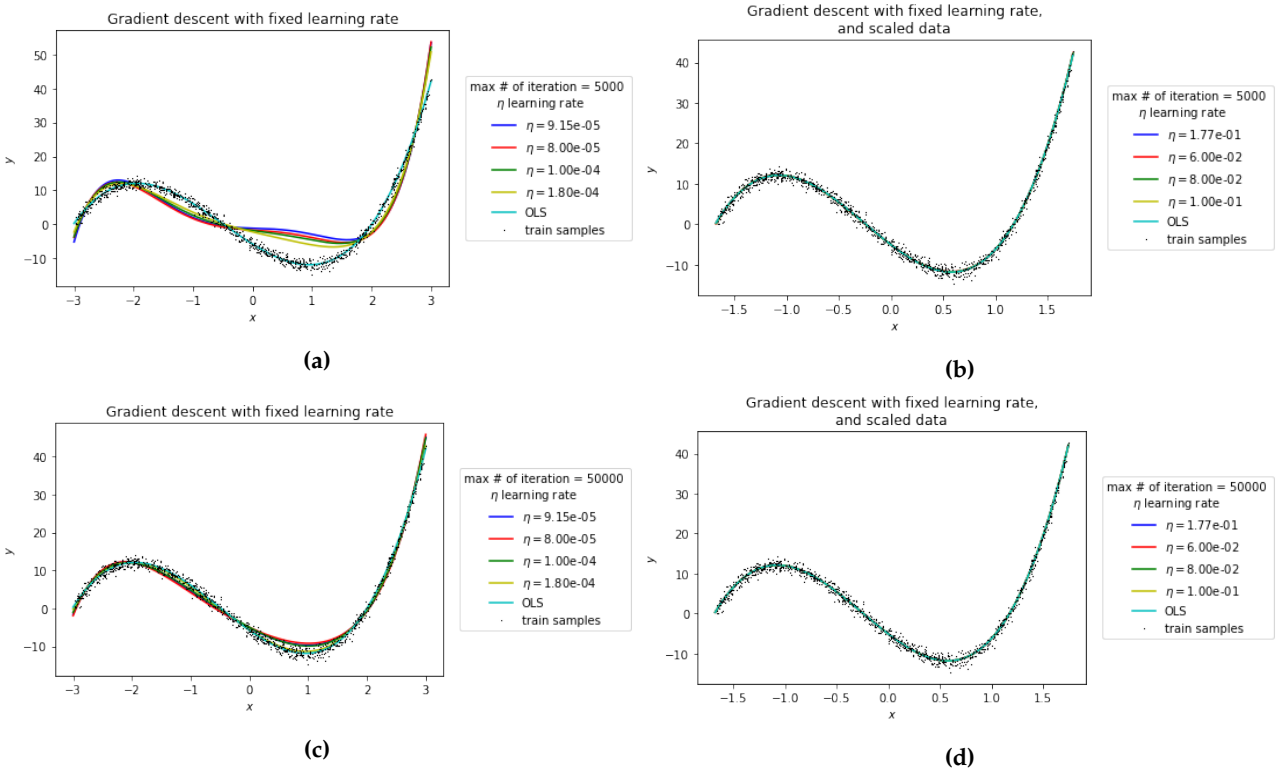


Figure 15: *Graphical representations of fitting our model to the data set with Gradient Descent approach with 5000* ***(a, b)****, and 50000 iteration steps* ***(c, d)*** *on non-scaled and scaled data* ***(a, c)*** *and* ***(b, d)*** *respectively.*

exact inverse matrix, other outputs are approximated with the GD method. With the lower iteration steps count, non-scaled data fit to the test function is rather poor. Increasing the iteration steps 10 times provides a better fit (Figure 15c) but it comes with higher computational costs. Furthermore, it should be noted, that the MSE is dependent on the choice of learning rate (Table 1). However, regardless of the number of iteration points with the pre-processed data our code delivers satisfying fit. The stopping criterion also shows that in case of maximum iteration number set to 50000 the actual iteration number was significantly lower, *i.e.* for learning rate = $1.77 * 10^{-1}$ the actual iteration number was 12632 with norm 2 of the gradient. Table 2 shows the values of MSE for train and test data with different learning rates and calculated form the exact matrix inverse. Relatively low and identical results were obtained for all chosen learning rates and from analytical calculations. Additionally, it can be observed that for the scaled data, η is of three orders of magnitude lower than in case of non-scaled data. Our function to fit is a polynomial of third order, therefore GD algorithm works well for our data test and it provides results within seconds. However, GD has limitations related to its computational cost. When performing the same calculations with 5 million data-points the GD optimization takes two minutes to deliver the results. If the function to

GRADIENT DESCENT without SCALED DATA		
Learning rate	MSE train	MSE test
9.15e-05	2.499	2.731
8.00e-05	3.575	3.889
1.00e-04	2.084	2.274
1.80e-04	1.090	1.144
Matrix inversion	0.969	0.979

Table 1: *Mean squared error for non-scaled data with maximum iteration number of 50000 treated with Gradient Descent*

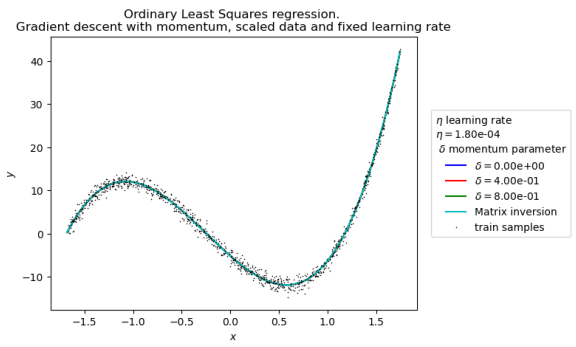


Figure 16: *Graphical representation of model fitting to the dataset with varied δ hyper-parameter with Momentum Gradient Descent approach.*

fit was more complicated it is reasonable to assume that the computation would take significantly longer time. One way to overcome this obstacle is to intro-

GRADIENT DESCENT with SCALED DATA		
Learning rate	MSE train	MSE test
1.77e-01	0.969	0.979
6.00e-02	0.969	0.979
8.00e-02	0.969	0.979
1.00e-01	0.969	0.979
Matrix inversion	0.969	0.979

Table 2: *Mean squared error for scaled data with maximum iteration number of 50000 treated with Gradient Descent*

duce a "memory" hyper-parameter: δ. We applied the method, known as Momentum Gradient Descent (described in the Methodologies section), to compare it with the standard GD. The results are presented in a graphical (Figure 16) and numerical (Table 3) form. In the case of our function to fit there is no observable improvement when applying Momentum GD.

GRADIENT DESCENT WITH MOMENTUM		
Momentum parameter	MSE train	MSE test
0.0	0.969	0.979
0.4	0.969	0.979
0.8	0.969	0.979
Matrix inversion	0.969	0.979

Table 3: *Mean squared error for scaled data with maximum iteration number of 50000 treated with Momentum Gradient Descent with fixed learning rate = 0.1*

GD and Momentum GD with Ridge regression Application of regularization parameter in linear regression provided better fit of the model and minimization of over-fitting problem. We decided to test whether addition of Ridge regression to GD with and without momentum could improve the fit to the model and provide us with smaller MSE. To test it we tried different values not only of hyper-parameters but also of learning rate. The results for Gradient Descent method are presented in Figure17 as plots of model fitting (**a**, **b**, **c**) and with results of MSE for both train and test data (**d**, **e**). Analyzing the plots with different λ it can be observed, that for small enough value of hyper-parameter, *i.e.* up to $3.16 * 10^{-4}$ the application of Ridge regression doesn't affect the accuracy of the model. It is specifically noticeable when comparing the MSE results with Table 2. However, if the λ is too high, then we observe the negative impact of employing Ridge. The MSE increased over 12 times. These results shows that using Ridge regression with GD method does not have a positive outcome and furthermore, if the λ parameter chose carelessly, the overall fit of the model is weakened. The same practice was applied to Gradient Descent with momentum. We learned that Ridge is not necessary when applying GD optimization method, albeit it can be useful for Linear Regression models. We tested our system with different hyper-parameters, *i.e.* Ridge λ and momentum δ to check whether adding a penalty term can be practical and if so, how much attention has to be given in deciding on these parameters. We kept the learning rate constant, $\eta = 0.1$. The results we obtained are presented in Figure18. Identical to GD with Ridge, when the "memory" hyper-parameter is included in the code, the most successful model fitting is with $\lambda \leq 3.16 * 10^{-4}$. Nevertheless, the results are corresponding to the ones without the penalty term. Moreover, regardless of the choice for δ value, if the λ is too high again the overall fit of the model is quite poor. Therefore, we remark the lack of benefits when using Ridge with Gradient Descent methods and based on our results it is not recommended to apply this method with these optimization techniques.

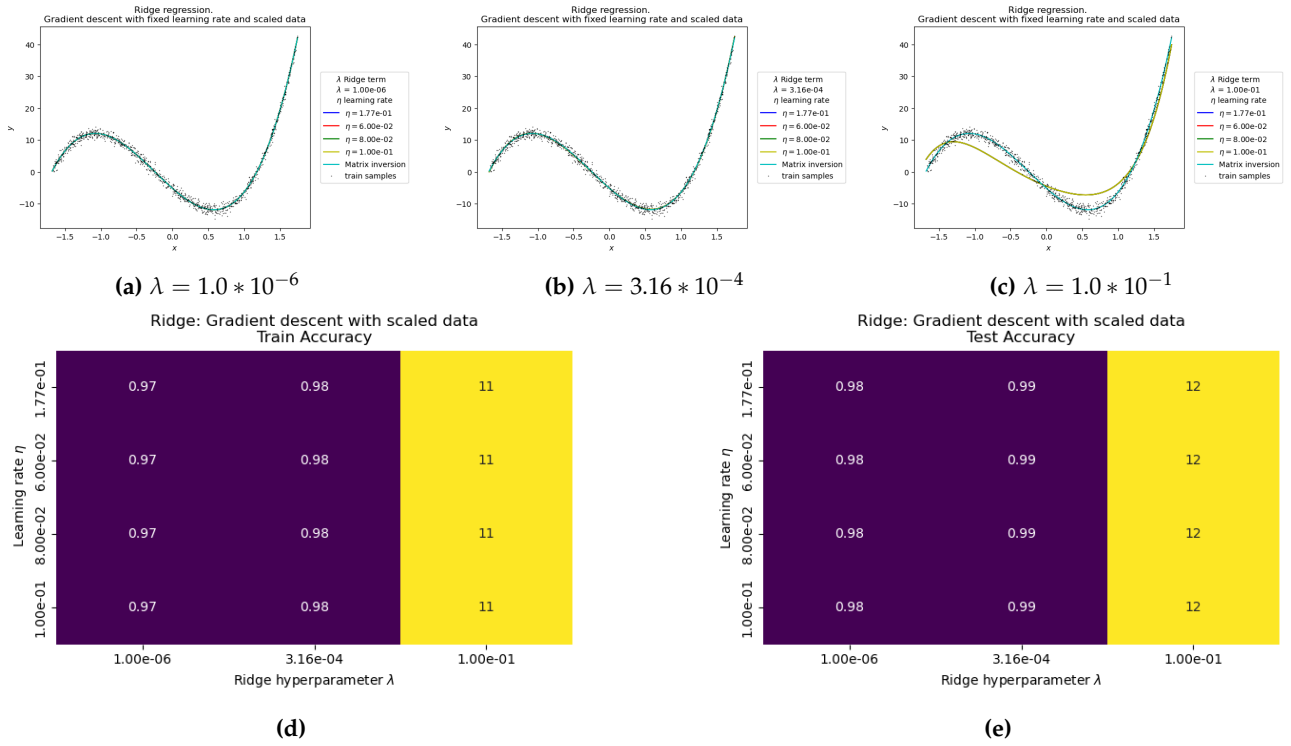
Stochastic Gradient Descent with minibatches Another iterative method applied in order to minimize the error function and find accurate model is SGD. Here, we introduce the subsets of training samples, called minibatches. For our study we decided to test the minibatches of the sizes: {2, 4, 8, 16} and with constant number of epochs, $n_{epochs} = 150$. The choice on the size of minibatches to test was made as a rule of thumb. The number of each minibatch was calculated as the ratio between number of datapoints (1000) and size of a minibatch. Therefore, the size was calculated to be {500, 250, 125, 62.5}, respectively. We applied the time decay approach from the Equation (32), where we set t_0 to 10, and $t_1 = 100$ in order to let the step length γ be flexible, *i.e.* to become small after time and providing the gradient does not move at all.

$$\gamma = \frac{t_0}{t + t_1} \tag{32}$$

Additionally, we introduced the Ridge hyper-parameter and examined the error function for $\lambda_1 = 3.16 * 10^{-6}$, $\lambda_2 = 1.00 * 10^{-3}$, $\lambda_3 = 1.00 * 10^{-8}$ with the same settings as mentioned above.

Plotted results for Stochastic Gradient Descent with minibatches are presented in Figure19a and the MSE values are reported in second column of Table 4. It is evident that the lowest error of model fitting is with the system of the smallest size of the minibatch. It is reasonable to assume that this is related to the simplicity of our function to test. The fitting is the lowest for size of $M = 16$, resulting with almost 30% higher MSE compared to $M = 2$. Therefore, it is discouraged to choose the size of minibatch $M \geq 4$ for a third degree polynomial function.

Introduction of Ridge hyper-parameter to the system does not improve the performance of the code. On the contrary, it results in worsening of the error,



(a) $\lambda = 1.0 * 10^{-6}$ **(b)** $\lambda = 3.16 * 10^{-4}$ **(c)** $\lambda = 1.0 * 10^{-1}$

(d) **(e)**

Figure 17: *Top: Graphical representations of fitting our model to the dataset with Gradient Descent approach with Ridge penalty parameter. Bottom: A heat-map of MSE for* ***(d)*** *train and* ***(e)*** *test accuracy on dataset treated with GD with Ridge regression with different values of penalty parameter* λ*.*

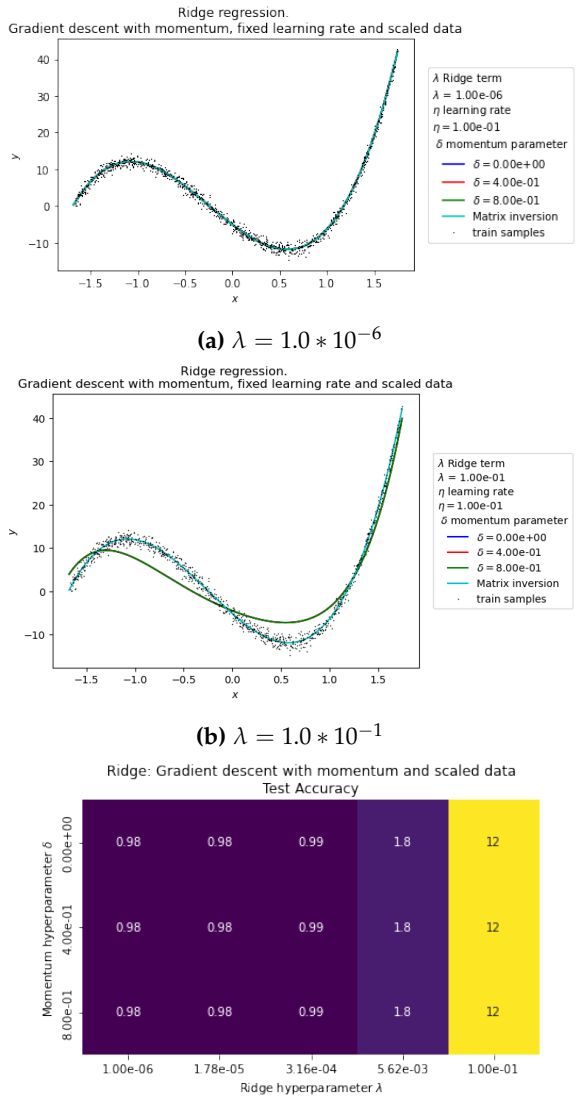
as can be seen in Table 4. For $\lambda_1 = 3.16 * 10^{-6}$ the best fit of the model is for the size of minibatch equal 4. For a smaller penalty term ($\lambda_2 = 1.00 * 10^{-3}$) it is evident that either size of 2 and 4 of the minibatches present the best performance. In any case, similarly to the SGD without Ridge, the least accurate results are in the case of the biggest size of minibatches.

Lastly, comparing SGD with minibatches with GD method, we can observe that SGD converges much faster compared to GD but the error function is not as well minimized as in the case of GD. Furthermore, the effect of adding Ridge hyper-parameter on GD when too high had more pronounced negative effect on MSE than when applied in SGD method.

MSE for Stochastic Gradient Descent			
Size of minibatch	OLS	λ_1	λ_2
2	1.65	1.77	1.70
4	1.68	1.70	1.89
8	1.93	1.84	2.12
16	2.12	2.04	2.12

Table 4: *Mean squared error results for SGD optimization algorithm with different size of minibatches, where* $\lambda_1 = 3.16 * 10^{-6}$ *and* $\lambda_2 = 1.0 * 10^{-3}$*.*

Optimizers: Adagrad, RMSProp and ADAM Three other widely acknowledged optimization methods were implemented in our study. New parameters had to be added to the code to allow for their application. For Adagrad, RMSProp and ADAM methods we decided to test the algorithm with number of epochs: $n_{epochs} = 100$, size of minibatches: $M = 5$ number of minibatches: $n_{minibatch} = 200$ and parameter used for obtaining numerical stability: $\epsilon = 10^{-8}$. In RMSProp approach an additional parameter γ was introduced and its role was to discount for the old gradients. We used the default value, $\gamma = 0.9$. For ADAM algorithm, two extra parameters were essential: $\beta 1$ and $\beta 2$. Such parameters are exponential decay rates for the first an second moment estimates, respectively, and were given the default values: 0.9 and 0.999, respectively. We then run our code with the aforementioned settings and compared the error functions for all three algorithms. We ran our codes for non-scaled and scaled data and the first difference between the three optimizers and GD or SGD present similar error results, regardless if the data were pre-processed or not, meaning that the accuracy of these methods is not limited by lack of data scaling. The output values for MSE are collected in Table 5. From the values of MSE it is coherent that for our function the most accurate algorithm was RMSProp, where the error for test sample was 0.978, comparable with results provided with GD method. The highest error function was obtained with Adagrad, however after minibatches sizes optimization, we found that

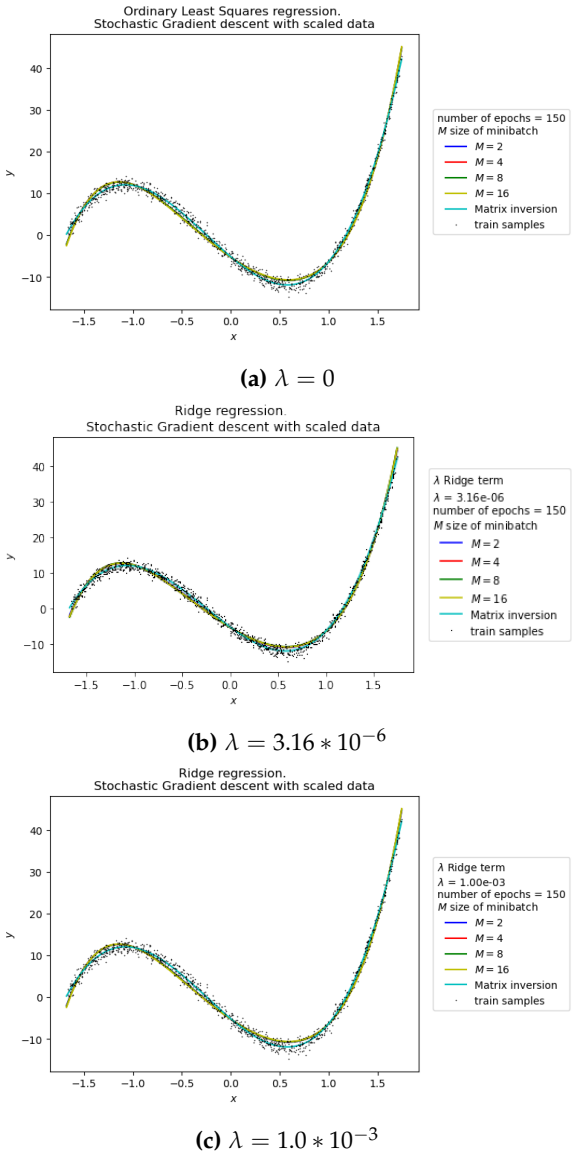


(a) $\lambda = 1.0 * 10^{-6}$

(b) $\lambda = 1.0 * 10^{-1}$

(c)

Figure 18: *Graphical representations of fitting our model to the dataset with Momentum Gradient Descent approach with Ridge penalty parameter.* ***(a)*** *The best fit with the function,* ***(b)*** *the least accurate fit,* ***(c)*** *a heatmap representation of MSE results for different momentum and Ridge hyper-parameters.*



(a) $\lambda = 0$

(b) $\lambda = 3.16 * 10^{-6}$

(c) $\lambda = 1.0 * 10^{-3}$

Figure 19: *Graphical representations of fitting our model with* ***(a)*** *Stochastic Gradient Descent, and* ***(b,c)*** *Stochastic Gradient Descent with Ridge regression penalty term.*

for $M = 16$, the MSE for both training and test data was 0.98. This shows the importance of tuning the size of hyper-parameters. We report the graphical presentment of these three methods in Figure 20. To

MSE	OPTIMIZATION METHOD		
	Adagrad	RMSProp	ADAM
train	1.011	0.984	0.996
test	1.044	0.978	1.028

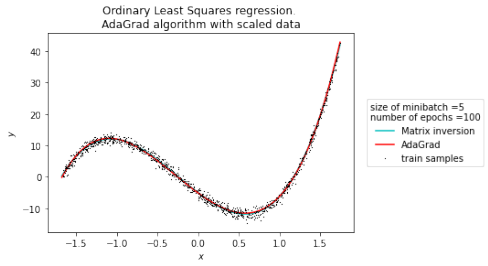
Table 5: *Mean squared error results for SGD optimization algorithm with different size of minibatches*

summarize, from all studied optimization methods in the case of third order polynomial function, the most accurate fit of the model is with Gradient Descent method, with the condition of pre-processed dataset.

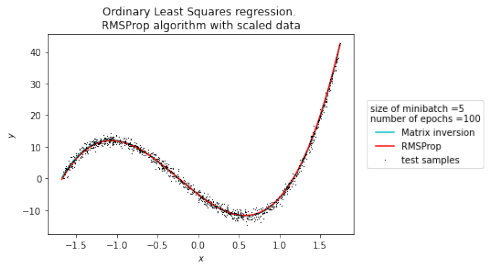
4.5 Logistic Regression

We utilized logistic regression to investigate a binary classification problem using the Wisconsin breast cancer dataset. This dataset includes measurements of cells and their corresponding diagnosis (malignant or benign). Our goal is to develop a classification model capable of predicting whether a cell is malignant or not based solely on its measurements.

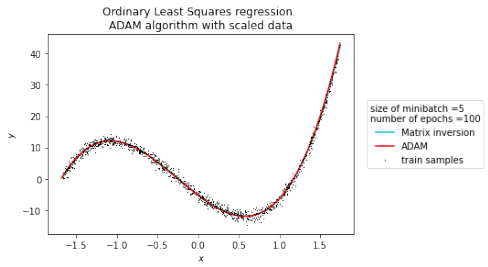
To begin our analysis, we downloaded the Wisconsin breast cancer dataset from this Kaggle page. We then utilized the Python data analysis library, pandas [10], to import and analyse the data. Pre-processing



(a) *Adagrad algorithm*



(b) *RMSProp algorithm*



(c) *ADAM algorithm*

Figure 20: *Comparison of the fitting performed with a) AdaGrad, b) RMSprop, and c) ADAM algorithms with the matrix inversion method.*

the data is crucial in classification problems, as features may not be numerical, have different units, or some samples may lack the value of some features. The dataset includes 569 samples and 33 variables, out of which 31 are of type float64, while the remaining variables, ID and diagnosis, are of type int64 and object, respectively. We can drop the ID variable since it is independent of the diagnosis. The diagnosis is the output we want to predict, and since we are implementing a logistic regression, we converted the target variables to 1*s* when they were classified as malignant ("M") and 0*s* when they were classified as benign ("B"). Further analysis showed that all variables were non-null for all samples, except for the last variable (Unnamed: 32), which had NaN (Not a Number) as its value. Consequently, we dropped the "Unnamed: 32" variable from the features as it is irrelevant to the classification problem. This left us with 30 features and 2 target classes. According to Figure 21, the dataset is composed of roughly 60% benign samples (class 0) and 40% malignant ones (class 1). This diagnosis distribution indicates

Figure 21: *Graphical visualization of the distribution of the diagnosis.*

that both classes are well-represented, allowing us to proceed with the classification task. However, since the distribution is not perfectly balanced, it is important to use both accuracy and precision/recall measurements when evaluating the performance of the classification model.

Once the data was cleaned and deemed suitable for classification, we separated the target variable, called y, from the predictors matrix $X \in \mathbb{R}^{569\times 30}$. We then split the data into training and test sets using the **train_test_split** function from the **Scikit-Learn** library [3]. To ensure reproducible results, we set

$$\textbf{random\_state} = 19\,. \tag{33}$$

We opted to include 30% of the dataset in the test split, resulting in 398 samples in the training set and 171 samples in the test set.

Since the values of the features have different scale, we needed to apply feature scaling to our data. We use the **StandardScaler** class from **Scikit-Learn** library [3] to standardize the data by subtracting the mean and dividing by the variance for each feature. As remarked in [2, Chapter 2], the scaler should only be fit to the training set. Subsequently, both the training and test sets (and any new data) must be transformed using the fitted scaler.

As a final step, we needed to set the decision boundary k, a parameter introduced in Equation (16). However, as we did not perform a precision/recall trade-off analysis to tune it, we maintained this hyperparameter at a fixed value of

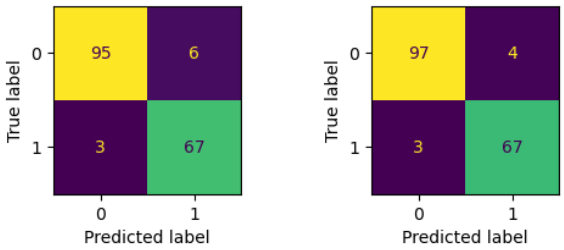
$$k = 0.5 \tag{34}$$

throughout the entire code.

Scikit-Learn logistic regression. We proceeded to define and train the models, starting with a logistic regression classifier using the **LogisticRegression** class from the **Scikit-Learn** library [3]. As parameters of the function we set

- **penalty** = None or 'l2',
- **solver** = 'lbfgs',
- **max_iter** =100000,
- **random_state** = 42.

The solver 'lbfgs' is the default one and it is an optimization algorithm that approximates the Broyden-Fletcher-Goldfarb-Shanno algorithm (quasi-Newton methods). Figure 22 displays the confusion matrices obtained when no penalty term is added (Figure 22a) and when an L^2 penalty term is added (Figure 22b). The addition of a penalty term reduces overfitting, as indicated by the performance measurements in Table 6. Specifically, the train accuracy decreases, while the test accuracy increases. Additionally, precision is higher, while the recall remains the same when a penalty term is added. This trend can also be observed in the confusion matrix, as the number of false negatives (samples wrongly classified as 0) remains unchanged, while the number of false positives (samples wrongly classified as 1) decreases from 6 (no penalty term) to 4 (L^2 penalty term).



(a) *No penalty term* **(b)** L^2 *penalty term*

Figure 22: *Confusion matrix of the classifier obtained using the LogisticRegression class from the Scikit-Learn library.* ***(a)*** *Model without penalty term.* ***(b)*** *Model with* L^2 *penalty term.*

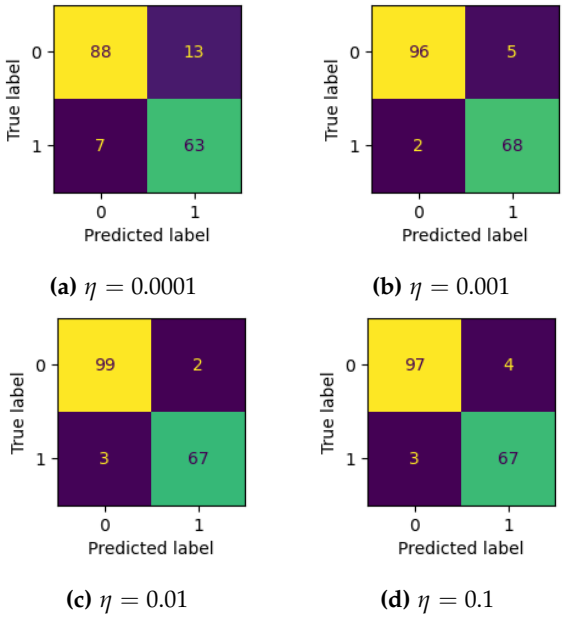
SciKit-learn. Performance measurements		
penalty term	None	'l2'
train accuracy	1.00	0.98
test accuracy	0.94	0.95
precision	0.91	0.94
recall	0.95	0.95
F1 score	0.93	0.95

Table 6: *Performance of the SciKit-learn classifier on the test set, with a focus on its ability to correctly classify the positive (malignant) class. The table compares the performance of the model with and without an* L^2 *penalty term.*

Gradient descent logistic regression. We implemented a gradient descent method for logistic regression and used it to train a classifier. Our goal was to investigate the effect of different learning rates on the classifier's performance. To this end, we set the hyper-parameters with the following values:

- **Niterations** = 10000,
- $\boldsymbol{\eta} = [0.0001, 0.001, 0.01, 0.1]$,
- **tol** = $10e^{-8}$
- $\boldsymbol{\lambda} = 0$.

Here, Niterations represents the maximum number of iterations, η is the vector of learning rates, tol is the stopping criterion, and λ is the regularization term.



(a) $\eta = 0.0001$ **(b)** $\eta = 0.001$

(c) $\eta = 0.01$ **(d)** $\eta = 0.1$

Figure 23: *Confusion matrix of the classifier obtained using Gradient Descent without a regularization term and varying the learning rate* η. ***(a)*** $\eta = 0.0001$ ***(b)*** $\eta = 0.0001$ ***(c)*** $\eta = 0.0001$ ***(d)*** $\eta = 0.0001$

GD. Performance measurements				
η	0.0001	0.001	0.01	0.1
train accuracy	0.85	0.96	0.98	0.98
test accuracy	0.88	0.95	0.97	0.95
precision	0.82	0.93	0.97	0.94
recall	0.90	0.97	0.95	0.95
F1 score	0.86	0.95	0.96	0.95

Table 7: *Performance of the Gradient Descent classifier on the test set, with a focus on its ability to correctly classify the positive (malignant) class. The table compares the performance of the model under different values for the learning rate* η.

Based on the confusion matrices shown in Figure 23, it is evident that the model with the smallest learning rate, $\eta = 0.0001$, performs poorly. The other models show similar performance, with the model using $\eta = 0.01$ having the smallest number of misclassified elements (5). However, one may prefer the model with $\eta = 0.001$ as it has the highest recall value. This preference is due to the specific classification problem we are considering, where it is more acceptable to wrongly classify a benign tumor than

a malignant one. The performance measurements of the models with varying learning rates can be found in Table 7.

We also experimented with adding an L^2 regularization parameter ($\lambda = 0.001$ /0.01 /0.1 and 1), but this did not significantly change the results. Therefore, we have chosen not to include those results in the report.

Stochastic gradient descent logistic regression with minibatches. As final model, we implemented a stochastic gradient descent method for logistic regression and used it to train a classifier. In this case we investigated the effect of the initial learning rate and the regularization parameter on the classifier's performance. We set the hyper-parameters with the following values

- **n_epochs** = 500,
- **M** = 4,
- **m** = int(n/M),
- $\boldsymbol{\lambda}$ = np.logspace(-5, 1, 7) ,
- $\mathbf{t_0}$ = 5 * np.logspace(-5, 1, 7) ,
- $\mathbf{t_1} = 50$.

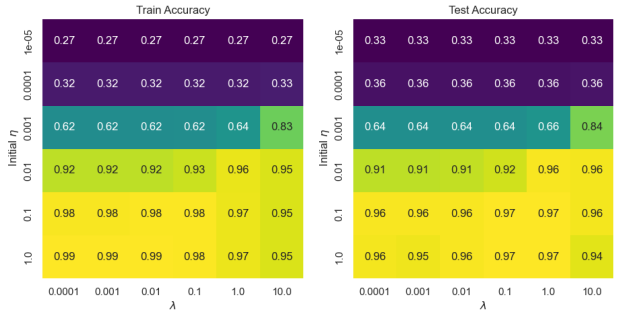
Here n_epochs is the number of epochs, M is the size of each minibatch, implying that m is the number of minibatches, where n is the number of samples in the set. The regularization term λ takes the values $[10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10]$, while the initial learning rates can be computed from the hyper-parameters t_0 and t_1 using the formula $\eta = t_0/t_1$. The parameters λ and t_0 are scaled logarithmically to ensure a wide range of values.

As a first step, we investigated the influence of the initial learning rate (η) and the regularization term (λ) on the train and test accuracy. As illustrated in Figure 24, we obtained higher test accuracy values for $\eta = 0.1$ and $\eta = 1$, while keeping λ at 0.01, 0.1, or 1. However, accuracy alone is not sufficient to select the best model. Therefore, we chose the models with the highest test accuracy and performed a more detailed performance analysis, resulting in the selection of models with:

- $\eta = 0.1$ and $\eta = 1$, with $t_0 = 5$ and $t_1 = 50$;
- $\lambda = 0.01$, 0.1, and 1.

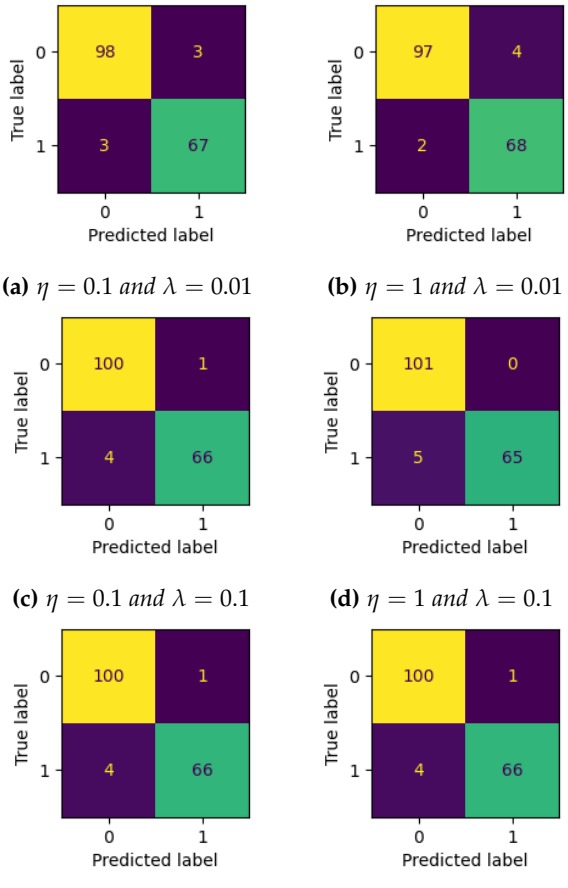
Based on the performance measurements presented in Table 8 and Table 9, we observed that increasing the regularization term led to higher precision but lower recall. Even if all the models had similar results, we preferred the one with $\eta = 1$ and $\lambda = 0.001$ because it misclassified only two elements of the positive (malignant) class, as shown in Figure 25b.

In conclusion, to gain a more comprehensive understanding of the SGD algorithm's behavior, it may be beneficial to explore the effects of other hyperparameters such as minibatch size and the number of epochs.



(a) *Train accuracy* **(b)** *Test accuracy*

Figure 24: *Train and test Accuracy of the SGD classifier varying the initial learning rate η (ordinate) and the regularization term λ (abscissa). The hyper-parameter are fixed as follows: n_epochs = 500, M = 4, t_0 = 5 * np.logspace(-5, 1, 7) , t_1 = 50, λ =np.logspace(-5, 1, 7).* ***(a)*** *Train accuracy* ***(b)*** *Test accuracy.*



(a) $\eta = 0.1$ *and* $\lambda = 0.01$ **(b)** $\eta = 1$ *and* $\lambda = 0.01$

(c) $\eta = 0.1$ *and* $\lambda = 0.1$ **(d)** $\eta = 1$ *and* $\lambda = 0.1$

(e) $\eta = 0.1$ *and* $\lambda = 1$ **(f)** $\eta = 1$ *and* $\lambda = 1$

Figure 25: *Confusion matrix of the SGD classifier varying the initial learning rate η and the regularization term. The hyper-parameter are fixed as follows: n_epochs = 500, M = 4.* ***(a)*** t_0 = 5, t_1 = 50, $\lambda = 0.01$ ***(b)*** t_0 = 50, t_1 = 50, $\lambda = 0.01$ ***(c)*** t_0 = 5, t_1 = 50, $\lambda = 0.1$ ***(d)*** t_0 = 50, t_1 = 50, $\lambda = 0.1$ ***(e)*** t_0 = 5, t_1 = 50, $\lambda = 1$ ***(f)*** t_0 = 50, t_1 = 50, $\lambda = 1$.

SGD. Performance measurements Initial learning rate $\eta = 0.1$			
λ	0.01	0.1	1
train accuracy	0.98	0.98	0.96
test accuracy	0.96	0.97	0.97
precision	0.95	0.98	0.98
recall	0.95	0.94	0.94
F1 score	0.95	0.96	0.96

Table 8: *Performance of the Stochastic Gradient Descent classifier on the test set, with a focus on its ability to correctly classify the positive (malignant) class. The table compares the performance of the model under different values of the regularization term. The hyper-parameter are fixed as follows: n_epochs = 500, M = 4, t_0 = 5, t_1 = 50.*

SGD. Performance measurements Initial learning rate $\eta = 1$			
λ	0.01	0.1	1
train accuracy	0.98	0.98	0.96
test accuracy	0.96	0.97	0.97
precision	0.94	1.0	0.98
recall	0.97	0.92	0.94
F1 score	0.95	0.96	0.96

Table 9: *Performance of the Stochastic Gradient Descent classifier on the test set, with a focus on its ability to correctly classify the positive (malignant) class. The table compares the performance of the model under different values of the regularization term. The hyper-parameter are fixed as follows: n_epochs = 500, M = 4, t_0 = 50, t_1 = 50.*

5 Conclusions

This work aimed at identifying the optimal linear regression method to model the Franke function, including the use of resampling methods and regularization techniques.

Resampling techniques were used to obtain further information about our fitted model. When applied to the Ordinary Least Sqeared method, cross-validation yielded a lower mean MSE over a larger range of model complexities with respect to the bootstrap technique.

The OLS method showed a decrease in the MSE up to a minimum followed by a rapid increase in its value (overfitting). This issue was overcome by introducing a regularization term. The lowest MSE was yielded by Ridge regression.

Since OLS requires the calculation of the pseudo-inverse matrix, it is very computationally expensive when dealing with large datasets. For this reason various optimization methods were tested.

Comparing SGD with minibatches with GD method, we can observe that SGD is a faster algorithm with respect to GD, but the error function is not as well minimized as in the case of GD. Furthermore, the addition of the L^2 regularization hyper-parameter unexpectedly increased the MSE, with a worse effect on SGD. This could be due to the simplicity of the function chosen to fit the data for which the problem of overfitting seems to be negligible.

We also studied our model with three particularly popular optimization methods: AdaGrad, RMSProp and ADAM. With the same settings, the best accuracy was found with RMSProp algorithm, with MSE analogous to the one obtained with the Gradient Descent method.

It would be tempting to state that the optimization methods proved to be computationally cheaper although less accurate with respect to the linear regression ones. However, these results are not comparable because of the difference in the chosen functions and output intervals. For the future work, it would be interesting to apply these optimization methods to the Franke function.

In the binary classification problem, Scikit-learn, GD and SGD classifiers all achieved similar results. This could be attributed to the small size of the dataset used in this study. As a precaution when classifying tumors, we preferred models with higher recall than high precision, regardless of the test accuracy. Our experiments revealed that these methods were more effective without or with a small regularization term, as a larger regularization term favors precision over recall. An improvement to the current work would be to perform the precision-recall trade-off analysis to better tune the decision boundary parameter.

Additionally, we need to highlight an important observation, which although was not obvious from the beginning right now appears to be true. This study was a direct proof that AI can one day gain control of human life, as it took ours for 4 months.

Acknowledgements

All contributors to this report are part of the CompSci doctoral program which is managed by the Faculty of Mathematics and Natural Sciences at the University of Oslo (UiO). The program is partly funded by the EU Horizon 2020 under the Marie Sklodowska-Curie Action (MSCA) - Co-funding of Regional, National and International Programmes (COFUND).

We would also like to acknowledge the UCI Machine Learning Repository for assistance received by using their repository.

Competing interests

The authors have no competing interests to declare.

Supplementary Files

The code can be found at this GitHub repository.

References

[1] "Lecture notes - Chapter 5: Resampling methods". In: (2023). URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter3.html.

[2] A Gereon. "Hands-on Machine Learning with Scikit-Learn and Tensor Flow". In: *O'Reily Media Inc., USA* (2018).

[3] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[4] "Bootstrap Methods and Permutation Tests". In: *Companion chapter 18 to The Practice of Business Statistics* (2003). URL: https://www.researchgate.net/publication/265399426_Bootstrap_Methods_and_Permutation_Tests.

[5] J. Duchi, E. Hazan, and Y. Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* (2011), pp. 2121–2159. URL: https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[6] T. Tieleman and G. Hinton. "Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* (2012), pp. 26–31.

[7] P. Kingma Diederik and Lei Ba Jimmy. "ADAM: A method for stochastic optimization". In: *Published as a conference paper at ICLR 2015* (2014). URL: https://doi.org/10.48550/arXiv.1412.6980.

[8] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[9] Michael L. Waskom. "seaborn: statistical data visualization". In: *Journal of Open Source Software* 6.60 (2021), p. 3021. DOI: 10.21105/joss.03021. URL: https://doi.org/10.21105/joss.03021.

[10] The pandas development team. *pandas-dev/pandas: Pandas*. Version (v1.5.2). Nov. 2022. DOI: 10.5281/zenodo.7344967. URL: https://doi.org/10.5281/zenodo.7344967.