Project 1 - Regression, Classification and Resampling Methods

Hannes Kneiding
Claudio Meggio
Verónica Suaste

CompSci PhD Course for Machine Learning

December 15, 2021

Link to GitHub - Repository

Abstract

In this work we present an overview and exemplary application of simple machine learning methods for regression and classification tasks implemented in Python. For regression we use generated data from the Franke function [1] with and without added noise and apply ordinary least squares (OLS), ridge and LASSO regression. For OLS and ridge regression we look at both the closed form solution approaches as well as a cost function driven approach using stochastic gradient descent. A batch stochastic gradient descent (SGD) variant was implemented. For classification we use the Wisconsin breast cancer dataset [2] and apply logistic regression and support vector machines. Additionally, we investigate and use the common resampling methods bootstrap and cross validation.

For polynomial regression on the Franke function we find that polynomials of degree 10 or higher lead to unbiased estimators if no additional noise is added. For ridge and LASSO regression we furthermore find optimal parameters for the regularisation parameter λ. For classification using logistic regression the performance of our own implementation is in accordance with the results from *scikit-learn* [3] with minor deviations which are likely due to the different implementations of stochastic gradient descent solvers. Furthermore, we find that support vector machines lead to slightly better test accuracies compared to logistic regression.

Overall, we come to the conclusion that all the presented methods are easy to use yet powerful approaches for regression/classification tasks. However, the good performance obtained in this work should not be taken at face value since the investigated problems are simple in nature. For more complex data the investigated methods might not be able to produce satisfactory results.

Contents

1 Introduction

Two of the main problem types in machine learning are regression (modelling the relationship between explanatory variables and continuous response variables) and classification (modelling the relationship between explanatory variables and discrete response variables) tasks. While modern neural networks can be used to tackle both of these problem types also less involved approaches exist.

In this work we investigate simple methods for both problem types. More precisely, we are investigating the regression methods

- Ordinary least squares regression (OLS)
- Ridge regression
- LASSO regression

For OLS and Ridge regression we look at both the closed form solution approaches as well as a cost function driven approach using our own implementation of a stochastic gradient descent algorithm. For classification we are investigating

- Logistic regression
- Support vector machines

Furthermore, we look at two common resampling methods, namely

- Bootstrap sampling
- Cross validation

and use these in conjunction with the investigated regression and classification methods.

The main objective of this project is to provide a gentle introduction into the field of machine learning and establish important concepts using toy examples and conceptually simple methods.

This report is structured as follows. First, we briefly review the theoretical background for the different methods used in this project. Then, we discuss the technicalities of our code implementation of these methods. After this we present and discuss our results. Finally, we conclude with a critical evaluation of the used methods and some final remarks.

1.1 Data

For the regression tasks in the first part we use Franke's function [1] in the domain $x, y \in [0, 1]$. Additional normally distributed noise is added in some cases scaled by α to ensure appropriate signal-to-noise ratio.

$$z = F(x, y) + \alpha \mathcal{N}(0, 1) \qquad (1)$$

Explicitly, the x and y data points will be used as inputs to the regression methods while the calculated z values will serve as the targets.

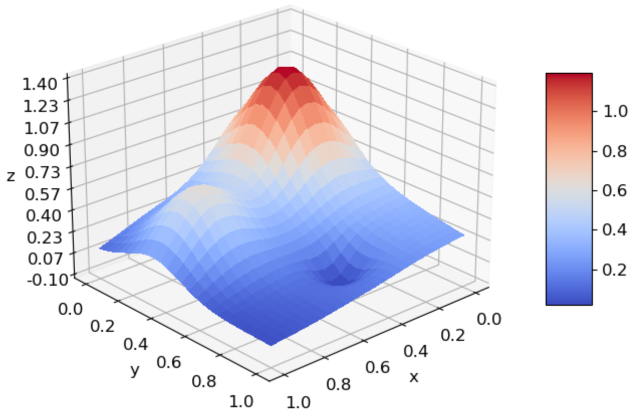


Figure 1: Franke function in the in the domain $x, y \in [0, 1]$ with 25 equidistant grid points in each dimension.

In the case of the classification tasks we use the Wisconsin Breast Cancer dataset [2] which contains 569 data points of 30 pre-selected features that were computed from image data of breast mass cells. For a medical discussion of these features refer to the original publication [2]. The corresponding target labels are binary, either benign or malign and refer to the type of tumor.

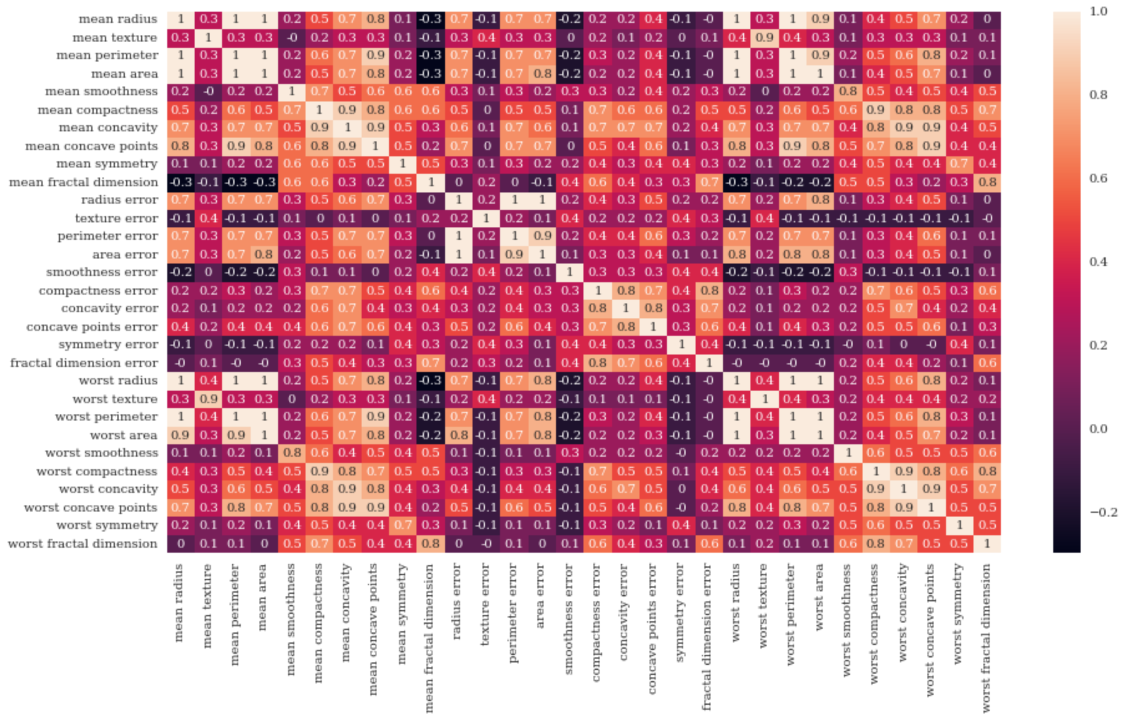


Figure 2: Correlation matrix of the attributes in Wisconsin Breast Cancer Dataset. [4]

2 Methods

2.1 Regression

Regression methods are used in statistics to predict one continuous variable (usually denoted as Y) starting from a $p-$dimensional variable X assuming that there exists a function f such that

$$Y = f(X) + \epsilon \tag{2}$$

where $\epsilon \sim N(0, \sigma)$.
When doing regression we have a dataset $\{x_i, y_i\}_i = 0^{n-1}$ that associates each regressor x_i with a label y_i. The idea is to tune the parameters of a model $h(\cdot)$ basing the choice on an arbitrary loss function that is computed based on the values y_i and $\tilde{y}_i := h(x_i)$.
The most known example is linear regression where the model is a line, its parameters are the slope and the intercept. Usually in this settings the loss function is the quadratic loss function.

2.1.1 Ordinary Least Squares Regression

The OLS estimator is identical to the maximum likelihood estimator (MLE) under the normality assumption for the error terms [5]. For Ordinary Least Square regression (OLS) we have that the loss function is the mean squared error (MSE), that is the sum of the square of the residuals [6]:

$$MSE := \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 \tag{3}$$

Remember that $\tilde{y}_i := h(x_i)$ in this case our model h is a polynomial of degree $p-1$ and its parameters are the coefficients of the polynomial $\beta = \beta_0, \beta_1 \ldots$. If x_i is a scalar then $h(x)$ is

$$h(x) = \sum_{j=0}^{p-1}\beta_j x^j. \tag{4}$$

Note that if we define the vector $\underline{X}_i := 1, x_i, \ldots, x_i^{p-1}$, then equation (4) can be rewritten, using matrix multiplications, as

$$h(x) = \underline{X}_i\beta. \tag{5}$$

Following the idea of transforming these formulas into matrix notation we can define Y as the vector

$$Y := y_0, \ldots, y_{n-1} \text{ and } \tilde{Y} := \tilde{y}_0, \ldots, \tilde{y}_{n-1} \tag{6}$$

Then define the matrix X as the matrix that has the vectors $\underline{X}_i$ as rows. The matrix X is called feature matrix.

With these new definitions equation (3) for the MSE becomes

$$\begin{aligned} MSE &= \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y_i})^2 \\ &= \frac{1}{n}\sum_{i=0}^{n-1}(Y - \tilde{Y})^T(Y - \tilde{Y}) \\ &= \frac{1}{n}\sum_{i=0}^{n-1}(Y - X\beta)^T(Y - X\beta) \end{aligned} \tag{7}$$

The goal of regression is to choose the parameter β that minimises the loss function (MSE). For this model it is possible to obtain a closed form expression for the optimal parameter $\hat{\beta}$. In order to obtain it it is necessary to ontain the first derivative of the MSE with respect to the parameter β:

$$\frac{\partial MSE}{\partial \beta} = -\frac{2}{n}X^T(Y - X\beta) \tag{8}$$

By setting it to zero we obtain

$$\hat{\beta} = (X^TX)^{-1}X^TY \tag{9}$$

Notice that in order to have $\hat{\beta}$ it is necessary that (X^TX) is invertible that is guarantee if X is non singular.
In our case we have that X is always invertible because it is a *Vandermonde* matrix [7] unless we use one observation x_i twice.
In this case is is easy to prove that it is point of minimum since the second derivative is

$$\frac{\partial^2 MSE}{\partial \beta^2} = \frac{2}{n}X^TX \tag{10}$$

which is a positive definite matrix.
If X is singular then it is necessary to use singular value decomposition in order to obtain $\hat{\beta}$.

2.1.2 Ridge Regression

The idea of ridge regression is to add a penalisation term that is proportional to the sum of all squared parameter values. More formally we add a term $\lambda\sum_{j=0}^{p-1}\beta_j^2$, by doing so the cost function becomes

$$C(\beta) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \sum_{j=0}^{p-1}x_{ij}\beta_j^2)^2 + \lambda\sum_{j=0}^{p-1}\beta_j^2, \qquad \lambda > 0 \tag{11}$$

Here λ is called regularisation parameter and it is usually determined by using cross validation, see chapter 2.6.2. Note that setting $\lambda = 0$ would correspond to plain OLS regression. [6]
Practically, this penalisation typically has the effect of flattening the fitted curve. Therefore,

the prediction will be less sensitive to variations of the predictor's x values.
The resulting optimal $\hat{\beta}$ is

$$\hat{\beta} = (X^T X + \lambda I_P)^{-1} X^T Y \tag{12}$$

It is important to note that if $\lambda = 0$, we effectively have no regularisation and we will get the OLS solution. As λ tends to infinity, the coefficients will tend towards 0 and the model will be just a constant function. [6]
Note that in the practical implementation of *scikit-learn* in python the penalisation summation starts from one instead of zero, this is to avoid the intercept to be included in the penalisation counting. [3]

2.1.3 LASSO Regression

LASSO (least absolute shrinkage and selection operator) regression uses the exact same cost function as Ridge regression with the difference that now for the penalisation term we are going to use the L1 norm instead of L2, that is we are going to use the absolute value of the $\beta's$. [6] Therefore the cost function becomes

$$C(\beta) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \sum_{j=0}^{p-1} x_{ij}\beta_j^2)^2 + \lambda \sum_{j=0}^{p-1} |\beta_j|, \qquad \lambda > 0 \tag{13}$$

In this case, because of the absolute value, analytically deriving a close form for the optimal β is difficult. Hence, LASSO regression is done by directly optimising the cost function (13) using some solver from the gradient family. [6]
Much like with ridge, we can vary λ to get models with different levels of regularization with $\lambda = 0$ corresponding to OLS and λ approaching infinity corresponding to a constant function.

2.2 Bias Variance Decomposition

We start by recalling the definition of bias and variance:

$$Bias := \theta - \mathbb{E}[h] \qquad Var(x) := \mathbb{E}[x - \mathbb{E}[x]] \tag{14}$$

Where h is a function to predict θ. So the Bias tell us how much error we expect the prediction h will have. While the variance tell us how much spread we will have among different prediction of the statistic h.
In the following we are going to use the term $\mathbb{E}$ both to indicate the expected value of a random variable and the sample mean of a sample, this means we may write $\mathbb{E}[x] = \frac{1}{n}\sum_i^n (x_i)$.
We have $y = f(x) + \epsilon$ where ϵ is a random variable with mean 0 and variance σ^2 and f is a

deterministic function, hence $\mathbb{E}[y] = \mathbb{E}[f(x) + \epsilon] = f(x)$.

$$
\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 &= \mathbb{E}[(y-\hat{y})^2] \\
&= \mathbb{E}[(f(x)+\epsilon-\hat{y})^2] \\
&= \mathbb{E}[(f(x)+\epsilon-\hat{y})^]2 \\
&= \mathbb{E}[(f(x)+\epsilon-\hat{y}+\mathbb{E}[\hat{y}]-\mathbb{E}[\hat{y}])^2] \\
&= \mathbb{E}[(f(x)-\mathbb{E}[\hat{y}])^2 + 2\epsilon(f(x)-\mathbb{E}[\hat{y}]) + \epsilon^2 + 2\epsilon(\mathbb{E}[\hat{y}]-\hat{y}) + (\mathbb{E}[\hat{y}]-\hat{y})^2 + \\
&\quad + 2(\mathbb{E}[\hat{y}]-\hat{y})(f(x)-\mathbb{E}[\hat{y}])] \\
&= \mathbb{E}[(f(x)-\mathbb{E}[\hat{y}])^2] + 2(f(x)-\mathbb{E}[\hat{y}])\underbrace{\mathbb{E}[\epsilon]}_{=0} + \mathbb{E}[\epsilon^2] + 2\underbrace{\mathbb{E}[\epsilon]}_{=0}\mathbb{E}(\mathbb{E}[\hat{y}]-\hat{y}) + \mathbb{E}[(\mathbb{E}[\hat{y}]-\hat{y})^2] + \\
&\quad + 2\underbrace{(\mathbb{E}[\hat{y}]-\mathbb{E}[\hat{y}])}_{=0}(f(x)-\mathbb{E}[\hat{y}]) \\
&= \mathbb{E}[(f(x)-\mathbb{E}[\hat{y}])^2] + \mathbb{E}[\epsilon^2] + \mathbb{E}[(\mathbb{E}[\hat{y}]-\hat{y})^2] \\
&= \mathbb{E}[(f(x)-\mathbb{E}[\hat{y}])^2] + Var[\epsilon] + Var[\hat{y}] \\
&= \frac{1}{n}\sum_{i=1}^{n}(f(x_i)-\mathbb{E}[\hat{y}_i])^2 + \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i-\mathbb{E}[\hat{y}_i])^2 + \sigma^2
\end{aligned}
\tag{15}
$$

In the resulting expression we have that the first therm is the mean of the square of the bias we have for each single observation squared while the second one is the variance an the third one is the irreducible error that comes from the noise in the data.

In general the ideal result would be to select a model that results in low variance and zero bias. However, this is usually not possible.
Allowing our model to have high variance in practice means we are able to fit very precisely the training set. However this is not always a desirable especially when working with noisy data because we do not want to fit the model to the random noise. This kind of error in the model is often referred as overfitting.
Regarding the bias its interpretation it is easier, it refers to how much, on average, the target data differs from our prediction. Having high bias is often due to underfitting, that means we are oversimplifying our model. [8]

2.3 Stochastic Gradient Descent

Essentially all machine learning methods boil down to being an optimisation task. In linear regression for example, we are trying to find a set of model parameters $\boldsymbol{\beta}$ that minimises the in-sample training error $C_{\boldsymbol{X}}(\boldsymbol{\beta})$ which can be written as

$$\widehat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}\in\mathcal{D}} C_{\boldsymbol{X}}(\boldsymbol{\beta}) \tag{16}$$

where $C_{\boldsymbol{X}}(\boldsymbol{\beta})$ is the error over the training data $\boldsymbol{X}$ using model parameters $\boldsymbol{\beta}$. In the case of the ordinary least squares and ridge regression, closed form expressions for the determination of $\widehat{\boldsymbol{\beta}}$ are available. For more complex models such as neural networks however, closed form expressions do not exist or are unfeasible. Therefore, iterative optimisation schemes are employed.

One of the most widely used optimisation schemes is the gradient descent (GD) method, which is based on the fact that the gradient of a function will always point towards increasing function values. Therefore, in order to minimise a function one needs to move in the opposite (negative) direction of the gradient as shown in equation (17).

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \gamma \nabla_{\boldsymbol{\beta}} C_{\boldsymbol{X}}(\boldsymbol{\beta}_i) \tag{17}$$

Here, γ denotes the learning rate which controls the speed of adaptation and is a hyper-parameter that needs to be specified. The optimisation procedure is initialised at a random point $\boldsymbol{\beta}_0$ and stops when some convergence criterion is satisfied (typically if the magnitude of the $\boldsymbol{\beta}_i$ lies below some threshold).

One important fact to note here is that the gradient is calculated over the whole training set $\boldsymbol{X}$ before making an update to $\boldsymbol{\beta}$. This can become problematic for huge datasets where the calculation of the full gradient takes a lot of time and therefore would significantly slow down the optimisation process. A remedy to this problem can be attained by employing a slight modification to the GD method, the stochastic gradient descent (SGD). The SGD operates in exactly the same way as the GD with the only difference that instead of the full gradient an approximation to it is used in each update step. This approximation is obtained by calculating the gradient only over a randomly sampled subset of the training data. [9] Therefore, the update equation becomes:

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \gamma \nabla_{\boldsymbol{\beta}} C_{\tilde{\boldsymbol{X}}}(\boldsymbol{\beta}_i) \tag{18}$$

where $\tilde{\boldsymbol{X}}$ denotes a randomly sampled subset of the full training data. The strict definition of the SGD requires that the approximated gradient is obtained from a single sample only which is inefficient and typically leads to noisy gradients. More commonly batches of data are used to compute the gradient approximation that ensure smoother convergence.
These batches can either be build deterministically by sampling without-replacement until each of the data points is part of exactly one batch or by sampling data points with-replacement. There is debate over which approach is optimal but empirical evidence seems to suggest that the without-replacement strategy performs better. [10, 11, 12]

Using this approach, the required computation time per iteration can be reduced. Furthermore beneficial is the fact that due to the stochastic nature of this approach jumping out of local minima is possible compared to the standard gradient descent. [9]
The performance of the SGD can be further improved by including a momentum term that additionally uses the update vector of the previous step which can be thought of as a moving average of gradients that is applied in order to keep optimising into direction with steady gradients. This is especially helpful in flat loss function landscapes where gradients is small and therefore optimisation progress with unmodified GD/SGD methods slow.

Many more modifications to the SGD method such as AdaGrad [13] and Adam [14] which can help improve convergence speed have been developed.

2.4 Classification

In this section the two very basic classification methods logistic regression and support vector machines are discussed. Classification methods are used to model the relationship between explanatory variables and discrete response variables.

2.4.1 Logistic Regression

Logistic regression is a generalised linear model used to model a relationship between predictor variables and categorical response variable, which makes it useful in classification problems.
Assuming we have a set of n observations each of them with p features, we will denote this observations with the matrix $\mathbf{X}_{n\times p}$ and the classification of each observation with the vector $\mathbf{Y}_{n\times 1}$. [15]
A generalized linear model has a three part specification [16]:

1. Random component: a probability distribution, from the exponential family, describing the outcome variable $\mathbf{Y}$, with $E(\mathbf{Y}) = \boldsymbol{\mu}$.
2. Systematic component: a linear predictor, with covariates $\mathbf{x_1}, \cdots, \mathbf{x_p}$, given by $\boldsymbol{\eta} = \sum_{j=1}^{p} \mathbf{x_j}\beta_j$.
3. A link function (g), monotic and differentiable, between the random and systematic components, such that: $g(\mu_i) = \eta_i$ or equivalent $g^{-1}(\eta_i) = \mu_i$.

In the case of binary logistic regression, the outcome variable has two classes that without loss of generality we can represent as $\{0, 1\}$ and as we want to model the probability of success for a given set of predictors, the three components for binary logistic regression would be, in the same order as the specification: [15]

1. Random component: For the outcome variable we can assume Binomial distribution with probability $\mathbf{p}$ because we are modeling the probability that $\mathbf{Y}$ is equal to class 1, given $\mathbf{X}$, i.e. $\mathbf{p} = \mathbb{P}(\mathbf{Y} = 1|\mathbf{X})$. Notice here that $\mathbb{P}(\mathbf{Y} = 1|\mathbf{X}) = 1 - \mathbb{P}(\mathbf{Y} = 0|\mathbf{X})$.
2. Systematic component: The parameters $\beta_0, \cdots, \beta_p$ will be estimated using maximum likelihood as described below.
3. The link is a sigmoid function called logit function given by:

   $$g(x) = logit(x) = log(\frac{x}{1-x}), \text{for } 0 \leq x \leq 1$$

   This function has the property that its inverse takes values between $-\infty$ and ∞ and maps them to a value between 0 and 1, which we will take as a probability value.

   $$g^{-1}(x) = \frac{\exp(x)}{1+\exp(x)} = \frac{1}{1+\exp(-x)}$$

Furthermore from the specification we want that $g^{-1}(\eta_i) = p_i$ which implies

$$p_i = \frac{\exp(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_p x_{i,p}))}{1 + \exp(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_p x_{i,p}))}$$

$\implies$

$$p_i = \frac{\exp(x_i^T \boldsymbol{\beta})}{1 + \exp(x_i^T \boldsymbol{\beta}))} \tag{19}$$

$\implies$

$$1 - p_i = 1 - \frac{\exp(x_i^T \boldsymbol{\beta})}{1 + \exp(x_i^T \boldsymbol{\beta}))} \implies 1 - p_i = \frac{1}{1 + \exp(x_i^T \boldsymbol{\beta}))} \tag{20}$$

In order to estimate the parameters $\boldsymbol{\beta}$, the maximum likelihood is used. First, we notice that, assuming $\mathbf{Y}$ as a set of independent and identically distributed random variables, the likelihood is given by:

$$L(\boldsymbol{\beta}) = P(Y|X, \boldsymbol{\beta}) = \prod_{i=1}^{n} p_i^{y_i}(1 - p_i)^{1-y^i}$$

Note that maximizing the likelihood is equivalent of maximizing the log likelihood as the log function is a monotone and increasing function. Therefore we want to estimate $\boldsymbol{\beta}$ as

$$\hat{\boldsymbol{\beta}} = \underset{\beta}{\text{argmax}}\{log(L(\boldsymbol{\beta}))\}$$

Now, instead of maximizing the log likelihood we are going to state the equivalent problem by minimizing the negative of log likelihood:

$$\hat{\boldsymbol{\beta}} = \underset{\beta}{\text{argmin}}\{-log(L(\boldsymbol{\beta}))\} \tag{21}$$

$$-log(L(\boldsymbol{\beta})) = -log(\prod_{i=1}^{n} p_i^{y_i}(1 - p_i)^{1-y^i}) = -\sum_{i=1}^{n} y_i log(p_i) + (1 - y_i) log(1 - p_i)$$

$$= -\sum_{i=1}^{n} y_i[log(p_i) - log(1 - p_i)] + log(1 - p_i) = \sum_{i=1}^{n} y_i log(\frac{p_i}{1 - p_i}) + log(1 - p_i)$$

$$= -\sum_{i=1}^{n} y_i log(\exp(\mathbf{x_i}^T \boldsymbol{\beta})) - log(1 + \exp \mathbf{x_i}^T \boldsymbol{\beta}) \quad \text{by using equations (19) and (20)}$$

Given that we want minimize this function, we take the partial derivatives with respect to the parameters $\boldsymbol{\beta}$. Also note that this function is in fact the cost function in our model and from now on we are going to denote it as $C(\beta)$. [15]
For computing the partial derivatives we have:

$$\frac{\partial C(\beta)}{\partial \beta_j} = -\sum_{i=1}^{n} y_i x_{i,j} - \frac{1}{1 + \exp(\mathbf{x_i}^T \boldsymbol{\beta})} \cdot \exp(\mathbf{x_i}^T \boldsymbol{\beta}) \cdot x_{i,j}$$

$$= -\sum_{i=1}^{n} x_{i,j}(y_i - \frac{\exp(\mathbf{x_i}^T\boldsymbol{\beta})}{1+\exp(\mathbf{x_i}^T\boldsymbol{\beta})}) = -\sum_{i=1}^{n} x_{i,j}(y_i - p_i)$$

Having this result, we can express the gradient of $C(\boldsymbol{\beta})$ vectorwise as:

$$\nabla C(\boldsymbol{\beta}) = -\mathbf{X}^T(\mathbf{Y} - \mathbf{p})$$

Because the partial derivatives don't have a closed-form expression, we use the gradient for minimizing the log likelihood via the stochastic gradient descent algorithm explained in section 2.3.
Once we have $\hat{\boldsymbol{\beta}}$ the binary logistic coefficients estimated, we use them to predict the outcome of a new observation as following: Firs we compute the estimated probability:

$$p(x) = \frac{\exp(\mathbf{x}^T\hat{\boldsymbol{\beta}})}{1+\exp(\mathbf{x}^T\hat{\boldsymbol{\beta}})}$$

And then we associate a class in the binary case accordingly to the obtained probability: [15]

$$f(x) = \begin{cases} 0 & p(x) \leq 0.5, \\ 1 & p(x) > 0.5 \end{cases}$$

2.4.2 Support Vector Machines

Support Vector Machines (SVM) are machine learning approaches mostly used for binary classification tasks. The input to SVMs are data points of the form

$$(\boldsymbol{x}_i, y_i) \tag{22}$$

where $\boldsymbol{x}_i$ are the input vectors of dimension d and $y_i \in \{-1, 1\}$ are the target values that denote the corresponding class of the data point.

The intuition behind this approach is to find a an affine subspace of dimensions $d-1$ (a hyperplane) that linearly separates the data points according to their respective classes. Additionally, one is not just interested in any subspace that separates the data, but in the so called maximum-margin hyperplane. To achieve this we first construct two parallel hyperplanes that separate the data points according to their respective class while maximising the distance between them. This is called the margin. The maximum-margin hyperplane then is the hyperplane in the middle between them. Thereby, one hopes to obtain a model that is robust when trying to classify unseen data. [9]

We constrain the two hyperplanes so that

$$\begin{aligned} \boldsymbol{w}^\intercal\boldsymbol{x}_i - b &\geq 1 \text{ for } y_i = 1 \\ \boldsymbol{w}^\intercal\boldsymbol{x}_i - b &\leq -1 \text{ for } y_i = -1 \end{aligned} \tag{23}$$

or in more compact form as

$$y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i - b) - 1 \geq 0 \tag{24}$$

Note that for $\boldsymbol{x}_i$ values lying on one of the hyperplanes the relationship is sharp.

$$y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i - b) - 1 = 0 \tag{25}$$

From this it follows that their distance is given by $\frac{2}{||\boldsymbol{w}||}$ so that we need to minimise $||\boldsymbol{w}||$ in order to maximise the margin while satisfying equation (24). To achieve this we use Lagrange multipliers to obtain find a suitable constrained optimisation problem.

$$\mathcal{L}(\boldsymbol{w}, b, \lambda) = \frac{1}{2}||\boldsymbol{w}||^2 - \sum \lambda_i (y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i - b) - 1) \tag{26}$$

Note here that minimising $\frac{2}{||\boldsymbol{w}||}$ is equivalent to minimising $\frac{1}{2}||\boldsymbol{w}||^2$ and has been chosen simply for mathematical convenience. Taking derivatives with respect to $\boldsymbol{w}$ and b we obtain

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} &= \boldsymbol{w} - \sum_i \lambda_i y_i \boldsymbol{w}_i = 0 \\ \frac{\partial \mathcal{L}}{\partial b} &= -\sum_i \lambda_i y_i = 0 \end{aligned} \tag{27}$$

which lead to

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2} \sum_{ij}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^\intercal \boldsymbol{x}_j \tag{28}$$

By solving for λ_i we find the support vectors of the problem, that is the data points that lie on the two parallel hyperplanes. The λ_i values are one for data points lying on the hyperplanes and 0 otherwise. Therefore, we can obtain $\boldsymbol{w}$ and b by

$$\begin{aligned} \tilde{\boldsymbol{w}} &= \sum_i \lambda_i y_i \boldsymbol{x}_i \\ \tilde{b} &= \frac{1}{y_i} - \boldsymbol{w}^\intercal \boldsymbol{x} \end{aligned} \tag{29}$$

Predictions can then be made through

$$\tilde{y}_i = \text{sign}(\tilde{\boldsymbol{w}}^\intercal \boldsymbol{x}_i - \tilde{b}) \tag{30}$$

This is a hard classifier in the sense that the resulting maximum-margin hyperplane will exactly separate the data points of the two classes. With noisy data from the real world this obviously can lead to models that generalise badly. To avoid this a so called soft-margin can be used by introducing slack variables that essentially allow for a number of miss-classifications when training the model. [9]

2.4.2.1 Non-linearities

A problem with this standard definition of SVMs is that they can only deal with linearly separable sets of data points. Consider for example a dataset as shown in figure 3. In this simple example the data points x_i can not be linearly separated into the two classes.

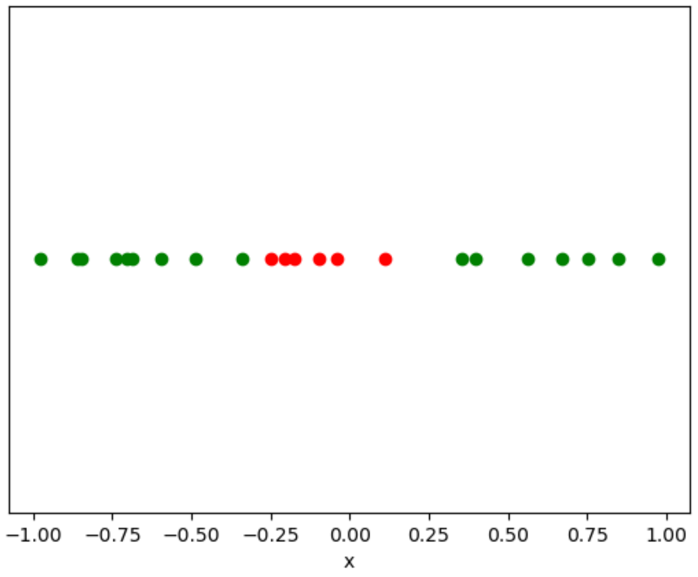


Figure 3: Example of a binary classification problem that is not linearly separable.

In cases like this, a standard SVM will not be able to converge to an appropriate decision rule. Relief to this problem can be obtained by projecting the raw input features into a higher dimensional space in which the problem *is* linearly separable. For this example a simple polynomial expansion up to second order

$$x \mapsto \boldsymbol{z} = \phi(x) = (x, x^2) \tag{31}$$

makes the problem linearly separable as can be seen in figure 4. The type of mapping is completely defined by function $\phi(\cdot)$. [9]
We can simply replace $\boldsymbol{x}_i$ with our new feature vectors $\boldsymbol{z}_i$ in equation (28) to arrive at

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2} \sum_{ij}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{z}_i^{\intercal} \boldsymbol{z}_j \tag{32}$$

which can be solved by a SVM since there clearly exists an affine subspace of dimension $d-1$ that separates the points of the two classes in this expanded feature space.

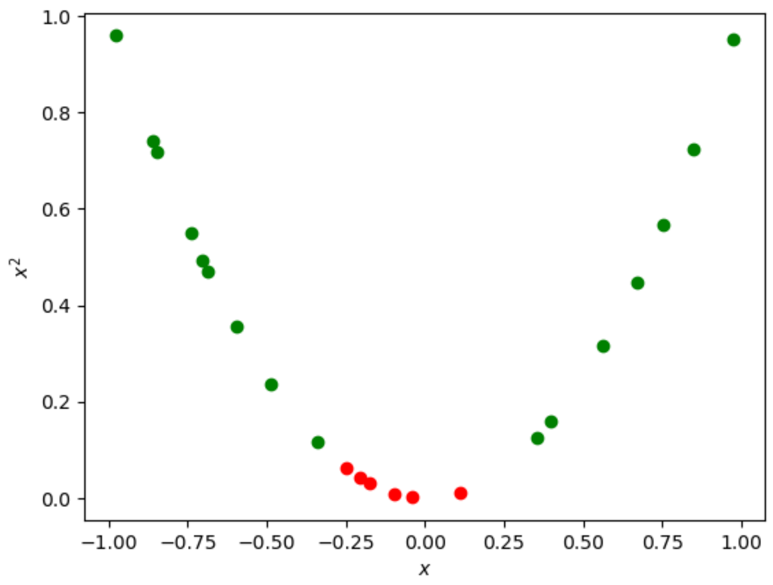


Figure 4: By expanding the problem in terms of a basis function $\phi(\cdot)$ the problem becomes linearly separable.

While this solves the problem of non-linearity this approach a big disadvantage is that we need to perform operations (dot product) on the expanded feature vectors $\boldsymbol{z}_i$. This will lead to large computational costs whenever the dimensionality of the chosen feature space is large. [9]
By virtue of Mercers's theorem [17] we can replace the dot product $\boldsymbol{z}_i^\intercal \boldsymbol{z}_j$ with so called kernel functions $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ which by definition are functions that calculate the dot product of two vectors in a different feature space without actually transforming the data points $\boldsymbol{x}_i$ into the feature space.

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle = \phi(\boldsymbol{x}_i)^\intercal \phi(\boldsymbol{x}_j) \tag{33}$$

In other words this means that for the calculation only the original vectors are used. The existence of this relationship is given by Mercer's theorem for some conditions on the kernel function (symmetry, positive-definiteness).

Using this the Lagrangian becomes

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2} \sum_{ij}^{n} \lambda_i \lambda_j y_i y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{34}$$

where no explicit transformation of vectors $\boldsymbol{x}_i$ into the feature space ϕ is necessary. Commonly used kernels are the polynomial and Gaussian radial basis function kernels. [9]

2.5 Scaling data

When using multidimensional data it is fundamental to scale all the features to be of the same dimension. Otherwise, some features will be given more or less weight depending on the dimension they are represented in. This is because if one feature is lager then the mean square error will be more influenced by it and in turn the fitting will be mainly done on this feature. [8]

Some examples of possible methods to scale the data are:

- min max scaling: essentially normalises all the data between 0 and 1
$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{35}$$
It does not work well with data where outliers are present. [8]

- quantile transformation
It works well with outliers.
It may distort linear correlations between variables measured at the same scale but renders variables measured at different scales more directly comparable (this is why principle component analysis) may not give correct results. [8]

- Z-score normalisation or standard normalisation
$$x' = \frac{x - \bar{x}}{\sigma} \tag{36}$$
which normalises the data so that the mean is zero and variance is 1. [8]

2.6 Resampling Techniques

Resampling is a way of efficiently using data in order to obtain statistical information about a specific model. It typically involves repeatedly fitting a model to slightly different subsets of the whole dataset. The aggregated results of these models can be used to obtain statistical parameters. Often times, resampling methods are used to get reliable estimates of model performance in terms of error and variance in order to make informed decisions for model selection. Since they require repeated refitting of models they come at an increased computational cost.

2.6.1 Bootstrap

The Bootstrap method is the most basic resampling method. Individual samples are simply generated by sampling with replacement from the whole dataset. For each of these sampled datasets the model is fitted and the corresponding test errors are calculated. The collection of these sample errors can be used to obtain a more robust estimation of the error by averaging over them. In the same way also an estimation of the variance can be obtained.
Important to note here is that only the training data is resampled while the test data is the same for each bootstrap sample.

2.6.2 Cross Validation

A somewhat more involved resampling technique is the so called k-fold cross validation (or simply cross validation). The idea here is to split the data into k folds of equal lengths and then perform k iterations where in each iteration exactly one fold is used as test data, and all other folds as training data (see figure 5. In the end an averaged estimated for the error can be obtained.

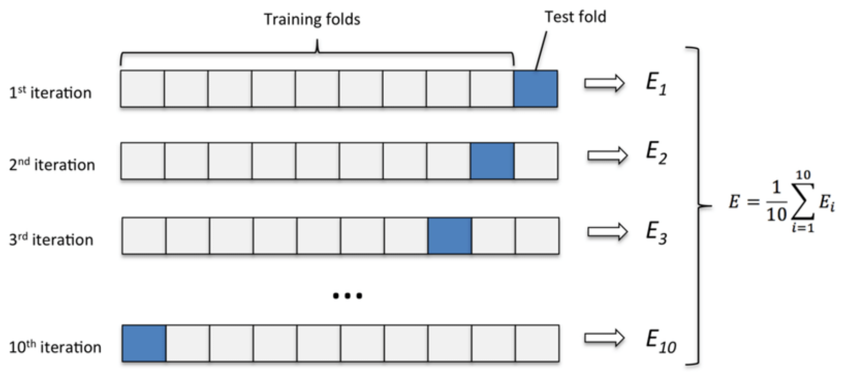


Figure 5: Schematic representation of k-fold cross validation with $k = 10$. Image taken from [18]

A nice effect of this approach is that each data point is used exactly once for testing and $k - 1$ times for training. Therefore, there is no stochastic element that could lead to the introduction of a sampling error as it is the case for bootstrap.

3 Implementation

For this project all functions and analyses have been coded in Python. The code can be found at the GitHub repository:

https://github.com/hkneiding/compsci-course-2021

The code includes functionalities for regression (OLS, Ridge, LASSO) and classification methods (logistic regression). Furthermore, the functions for bootstrap resampling and cross validation are included.

For some functionalities external packages have been used. The (pseudo-)inverse matrix required for ordinary least squares and Ridge regression is obtained using the package *numpy* [19]. Also, for LASSO regression the minimisation function from the *scipy* package [20] is used. For plotting *matplotlib* [21] has been used.

For stochastic gradient descent multiple different variants have been implemented. The one used throughout the project was a batch stochastic gradient descent. That is a gradient descent that goes through the dataset in batches but where each batch is sampled with replacement from the full dataset. After each batch the weights are updated. An epoch is completed after $k = \frac{\text{Number of data points}}{\text{Batch size}}$ batches have been analysed.

The Python package *scikit-learn* [3] has been used for the support vector machine analyses and to compare results with our own implementation.

3.1 Structure

The following is an overview of the different files and their functionalities in the project (found in the `src/` directory).

- **regressors.py** - Contains functions for regressors.
- **resampling.py** - Contains functions for bootstrap resampling and crossvalidation.
- **sgd.py** - Contains multiple different implementations of the stochastic gradient descent method as well as a method to estimate the bias-variance decomposition of the MSE.
- **tools.py** - Toolbox for general functions such as the calculation of R^2 values or cost functions.
- **franke_function.py** - Contains helper functions to get data from the Franke function.
- **regresssor_types.py** - Enum class to distinguish between different regressors.
- **ridge_analysis.py** - Contains all functions used to generate plots and comparisons for ridge regression including hyper-parameters tuning.

The functions are coded to be compatible with a special format of the data as a Python `dict` with the following keys:

- **inputs** - Contains a list of one-dimensional *numpy* arrays. Each array holds the feature values for the individual data points.
- **targets** - Contains a one-dimensional *numpy* array that holds the target values for the individual data points.

The functions used to performed the different analyses can be found in the **run.py** file in the root directory. The different functions can be called separately by modifying the `__main__` function. Furthermore, we included some tests to compare our implementation of the different regression methods with the implementation from *scikit-learn* (found in the `test/` directory). The tests can be run by executing `pytest` in the root directory and should all pass.

Within the root directory we also redistribute a CSV file with the data from the Wisconsin breast cancer dataset [2] used for the classification tasks.

4 Results and Discussion

All results presented in this chapter are obtained using either bootstrapping or cross validation if not specified otherwise. In the case of bootstrapping a training ratio of 80% was used.

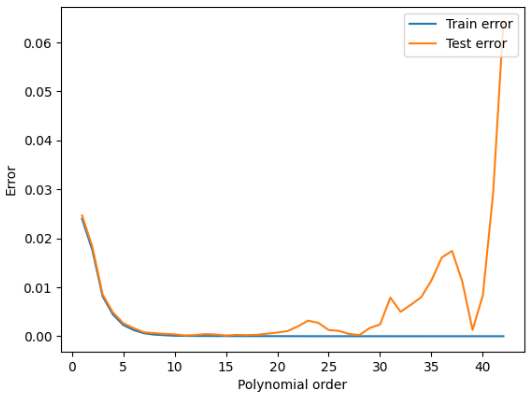
4.1 Regression

In this section we investigate regression problems using ordinary least squares, ridge and LASSO regression on data generated from the Franke function as described in chapter 1.1. For this data additional scaling of input features was omitted since the values are bounded between 0 and 1 by definition. For a discussion of scaling and an overview of different techniques refer to chapter 2.

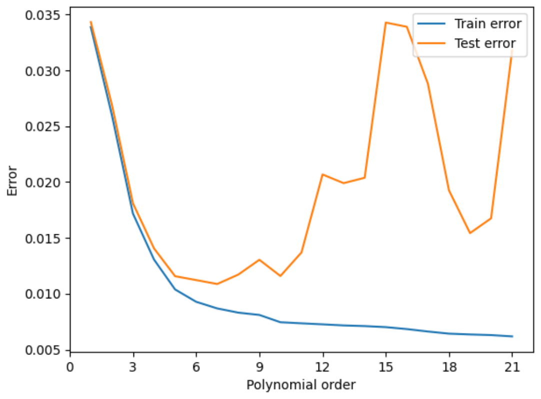
4.1.1 Ordinary least squares

For ordinary least square we first investigate the relationship of the test and train errors and the polynomial order. We expect the typical picture of decreasing train error with increasing model complexity while the test error first decreases and then increases due to overfitting to the training data. We have a clear example of this in figure 6 where in the first case the data have no noise and the test error does not increase until we have a very high polynomial degree (bigger than 30). While in the second case, when we add noise to our data one can immediately notice that we start having a situation of overfitting already when the polynomial's degree is bigger than 8.

We can observe underfitting in the first part of figure 6 where the polynomial used for the regression has degree 4 or lower. If we think at the extreme case when we use degree zero, the explanation of underfitting becomes immediate. Indeed we are trying to fit the Franke function with an horizontal plane that means we simply use the average value of the function as approximation, which can not be a satisfying result.



(a) Without noise.



(b) With Gaussian noise $\sigma = 0.1$.

Figure 6: Error rate of the OLS regression for Franke function against the degree of the polynomial. Used cross validation using a grid 20 $\times$ 20 as dataset.

Worth of a discussion is also the behaviour of the algorithm when we are using dataset of different dimensions 7. Indeed when the training dataset is small, when modelling the function OLS algorithm has a limited amount of information and hence it tends to model the error too resulting in overfitting. On the other hand if the dataset is large the generalisation capability of the model is much higher. The results in this case is a larger training error in favour of lower test error.

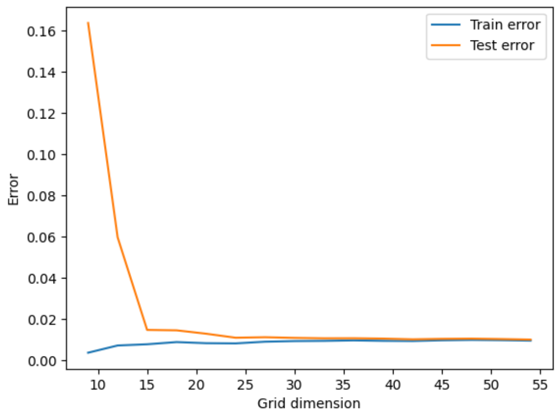
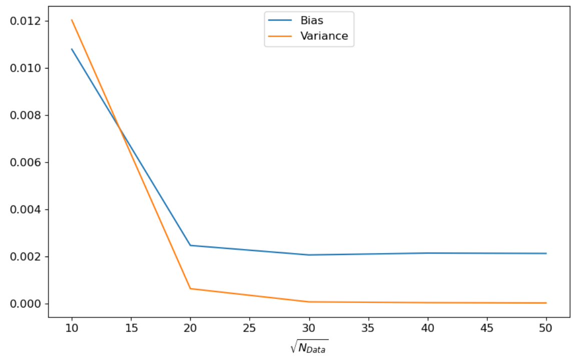


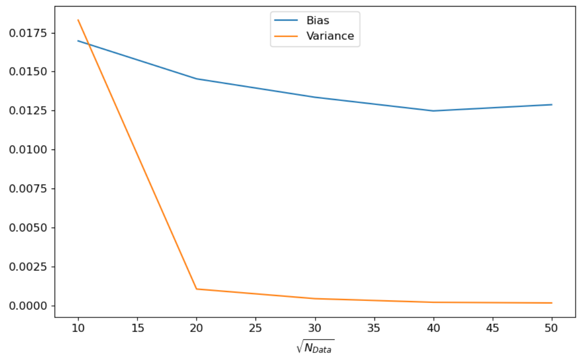
Figure 7: Performance of OLS on Franke function with noise $\sim N(0, 0.1)$. On the x axes is reported the dimension of the grid $n \times n$ used as dataset. For this simulations bootstrap resampling and polynomial of degree 11 has been used.

4.1.1.1 Bias-Variance decomposition

To further investigate both these relationships we can also look at the decomposition of the mean squared error into bias and variance. In figures 8 and 9 the bias and variance contributions in relation to the number of data points and model complexity are shown, respectively.



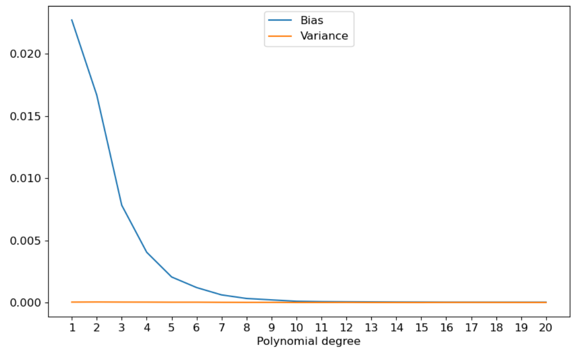
(a) Without noise.



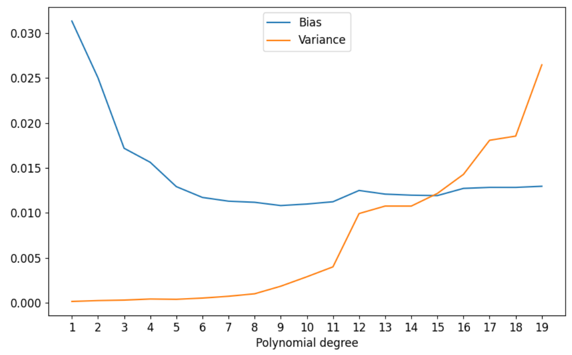
(b) With Gaussian noise $\sigma = 0.1$.

Figure 8: Bias-variance decomposition for OLS in relation to the number of data points used.

From figure 8 it can be seen that the bias completely converges for 30 grid points and more.



(a) Without noise.



(b) With Gaussian noise $\sigma = 0.1$.

Figure 9: Bias-variance decomposition for OLS in relation to the polynomial degree used. 30 grid points in each dimension.

In figure 9 we can observe an analogous picture to figure 6. When adding no noise we observe that the bias converges to 0 at around polynomials of degree 10. From this we can deduce that polynomials of degree 10 or higher are required to properly (without introducing bias) describe the data. In other words, when using polynomials of degree less than 10 we are underfitting, meaning that our model is not complex enough to reproduce the real data. When adding noise the bias converges to a non-zero value which is due to the fact that the reported quantity is the bias of the test error. At the same time we can also see that we obtain larger variance which is an overfitting effect due to the fact that the model start fitting to the random noise.

4.1.1.2 Resampling

We further investigate the effect of resampling methods using cross validation (see section 2). For this we ran OLS with cross validation for 5 up to 10 folds and report the average fold error as well as the variance of fold errors in table 1. Additionally, we compare the results with bootstrap using 100 samples.

Table 1: Averaged errors and variances of estimated errors for OLS using bootstrap and cross validation with different numbers of folds. 50 grid points in each dimension. Polynomial of degree 10. No noise.

	Bootstrap[†]	Cross validation				
		5	7	8	9	10
E[ϵ]	$1.1 \cdot 10^{-4}$	$1.13 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	$1.09 \cdot 10^{-4}$
Var(ϵ)	$2.4 \cdot 10^{-11}$	$6 \cdot 10^{-10}$	$4.2 \cdot 10^{-10}$	$3.6 \cdot 10^{-10}$	$2.4 \cdot 10^{-10}$	$4.4 \cdot 10^{-10}$

$k = 6$ omitted for clarity. Results were identical to $k = 7$.

[†] Using 100 samples.

The averaged error values for bootstrap and cross validation are very similar. For cross validation with $k = 10$ a marginally smaller error has been obtained which most likely is a stochastic effect. Similarly, for the variance all cross validation runs yielded errors with variances of the same magnitude. For bootstrap a slightly lower variance has been obtained (one order of magnitude). This is expected since with a hundred bootstrap samples, the statistical basis for the averaged errors is much more reliable than with 5 to 10 folds in cross validation.

4.1.1.3 Parameter estimation with SGD

In this part of the work we analyse OLS regression not implemented with its closed form for matrix inversion but with SGD algorithm. We compare the model performance, measuring the accuracy, in terms of different learning rates. For this experiment we select 10 values in the range $[0.000001, 0.001]$ spaced evenly and we compute the accuracy along 200 epochs. The graphic results are shown in Figure 10. In this image we see firstly that a learning rate of value 0.01 is larger enough to make the gradient increase and decrease along epochs which results in accuracy also increasing and decreasing along epochs. We can also note that a learning rate smaller than 0.0001 makes the convergence of accuracy slow and values between 0.001 and 0.0005 seem to give the best performance. In order to compare our results, we executed similar experiment using *scikit-learn* library. We used SGDRegressor, with the following parameters in order to simulate better our own algorithm:

- loss: 'squared_error'
- max_iter: 400
- learning_rate: 'constant'
- fit_intercept: False

- penalty: 'l2'
- alpha: 0
- eta0: each of ten in range $[0.000001, 0.001]$ spaced evenly.

Figure 11 displays the results, in which we can observe close outcomes as the ones obtained with our algorithm for SGD. In this image we can see the same behaviour with learning rate value 0.01, the accuracy increases and decreases along the epoch although smoother than the one observed in Figure 10. Another difference worth to noting is that convergence happens slightly earlier with our algorithm of SGD than the one from *scikit-learn*.

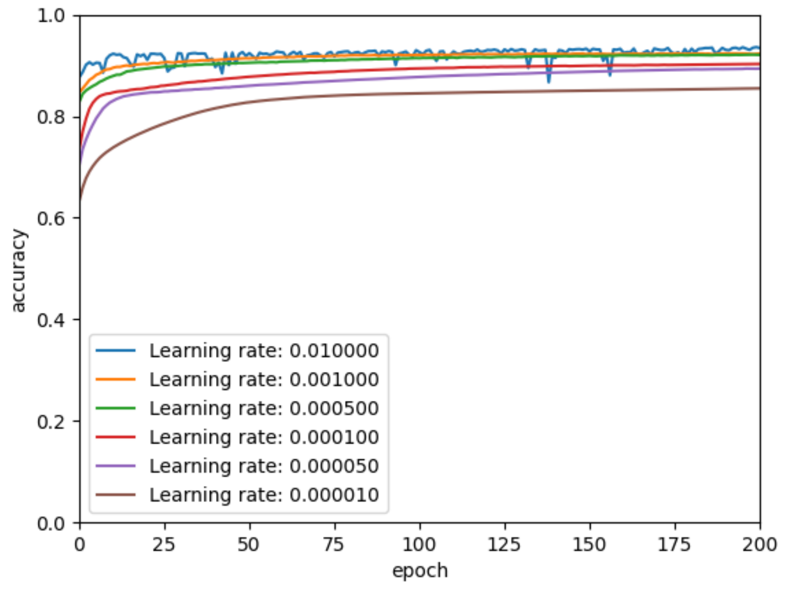


Figure 10: Accuracy convergence in terms of learning rates along epochs.

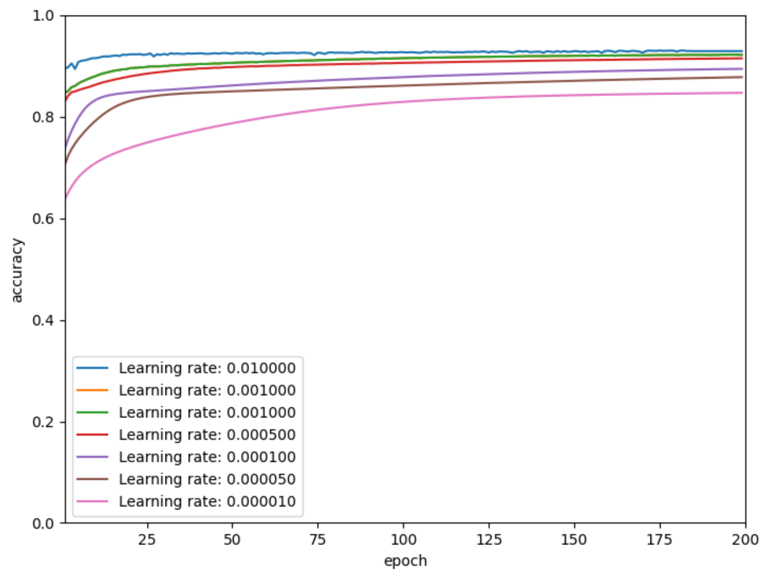


Figure 11: Accuracy convergence in terms of learning rates along epochs, using SGDRegressor from *scikit-learn* library

4.1.2 Ridge

In order to explore the behaviour of the regularisation parameter on the Ridge regression for the Frank function, we conducted an experiment where we took 100 values of λ evenly spaced between [-7, 3] in a log10 scale and used cross-validation(10-fold) resampling method to compute the mean squared error. The results are shown in Figure 12, where we also display a comparison with *scikit-learn* implementation of Ridge regression. In both implementations the closed-form solution was used for matrix inversion. The polynomial degree used is 10 and noise with variance of 0.1 was added to the data.
Results from this experiments suggest that the smaller the regularisation parameter the smaller the MSE. Values smaller than 10^{-4} seem to give better performance in terms of MSE.
As for the comparison with the implementation of *scikit-learn* is clearly no much difference, we can note how both curves practically overlap.

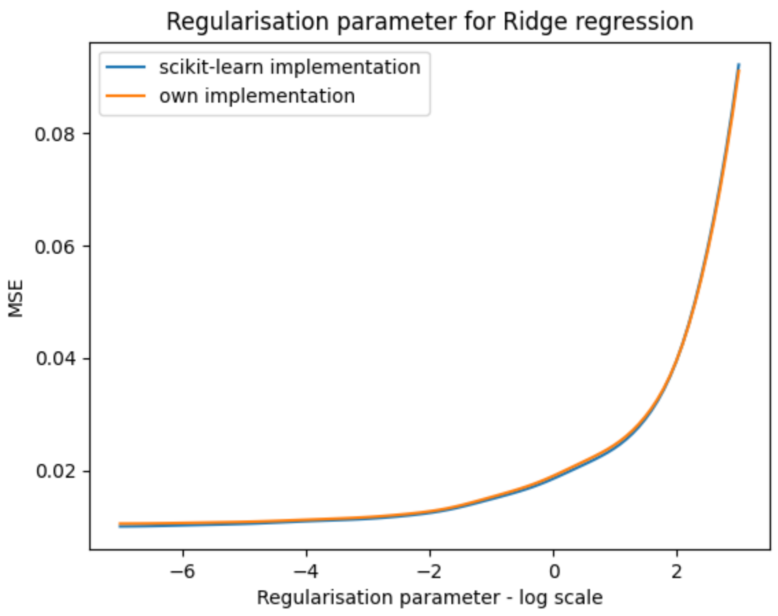


Figure 12: MSE for different values of regularisation parameter in the training data. Comparison with *scikit-learn* implementation

4.1.2.1 Bias-Variance decomposition

When comparing the behaviour of the variance in OLS (figure 9) with the same graph for ridge regression, figure 13, one can immediately notice that ridge regression is much more robust to overfitting. This is because the penalisation term force the algorithm to have a smoother behaviour. The decrease of the variance however comes at the expense of the bias [22].

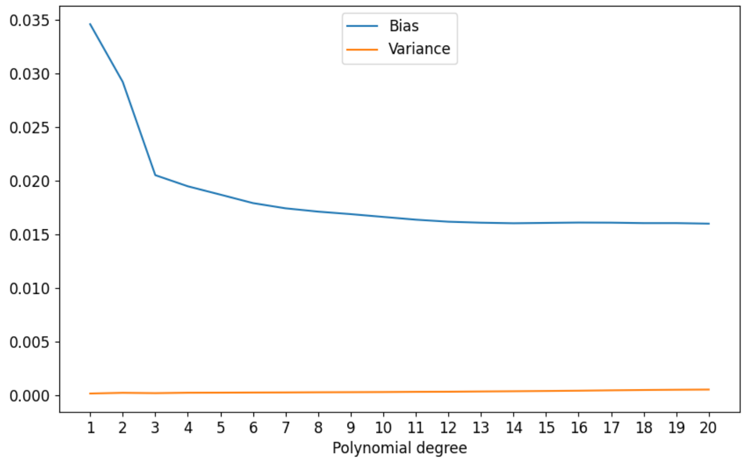


Figure 13: Bias variance decomposition of the MSE for ridge regression with parameter $\lambda = 0.05$ and a dataset of $30\times$ 30 on the Franke function with added noise $N(0, 0.1)$

In figure 14 it is reported how the bias variance changes in relation of the hyperparameter λ. Note that in the figure bias and variance have been separated, this has been done in order to better highlight the Bias Variance trade-of that happens as the hyperparameter λ increase. Indeed as λ increase the bias increases as well, but the variance decrease.
This happens because having a large parameter λ implies more focus on the penalisation parameter and hence more emphasis on a smoother prediction sacrificing precision on individual observations.

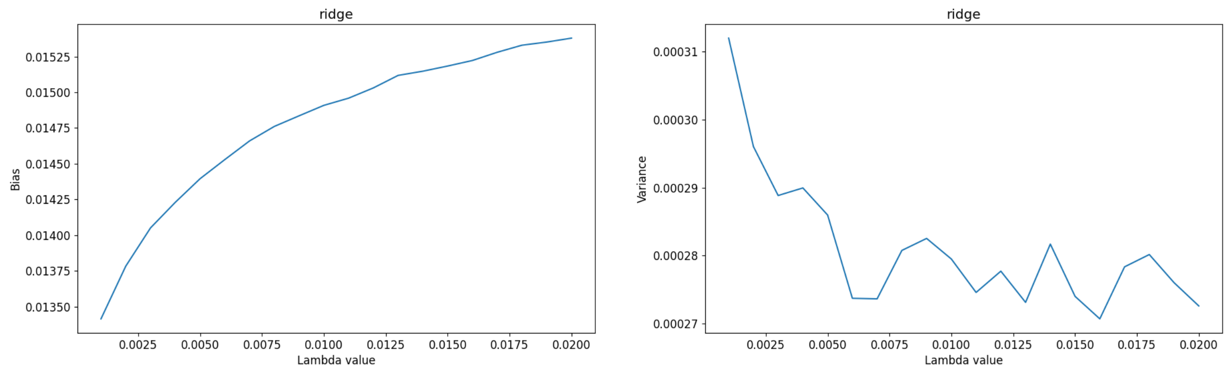


Figure 14: Bias variance decomposition of the MSE for ridge regression with polynomial degree 8 and a dataset of $30\times$ 30. Plots for bias and variance are split in order to represent the different scales.

4.1.2.2 Resampling

For ridge regression we also compare bootstrapping with cross validation with different k values, but this time with added noise. The results are shown in table 2. Similar to the results we obtained for OLS (compare figure 1) the obtained error values are all very similar. The main difference to OLS is that the errors are greater by two orders of magnitude which can be attributed to the fact that this time noise was added to the Franke function output. Also for the variances we observe the same behaviour as for OLS: bootstrapping leads to a variance that is about one magnitude less compared to cross validation. The variance values are larger by about three orders of magnitude compared with OLS which is also an effect of the added noise.

Table 2: Averaged errors and variances of estimated errors for ridge regression using bootstrap and cross validation with different numbers of folds. 50 grid points in each dimension. Polynomial of degree 10. Gaussian noise $\sigma = 0.1$.

	Bootstrap[†]	Cross validation				
		5	7	8	9	10
$\mathrm{E}[\epsilon]$	$1.09 \cdot 10^{-2}$	$1.03 \cdot 10^{-2}$	$1.03 \cdot 10^{-2}$	$1.03 \cdot 10^{-2}$	$1.03 \cdot 10^{-2}$	$1.02 \cdot 10^{-2}$
$\mathrm{Var}(\epsilon)$	$2.6 \cdot 10^{-8}$	$2.5 \cdot 10^{-7}$	$4.4 \cdot 10^{-7}$	$3.4 \cdot 10^{-7}$	$6.1 \cdot 10^{-7}$	$9.5 \cdot 10^{-7}$

$k = 6$ omitted for clarity. Results were identical to $k = 7$.
[†] Using 100 samples.

4.1.2.3 Parameter estimation with SGD

In this section we analyse Ridge regression, but in this case we use, for matrix inversion, our own implementation for SGD. The experiment consists more precisely on doing Ridge regression on the Frank function. We first generate a grid data of 60 $\times$ 60 and we split this data set in 80% for training and 20% for testing. In sections above we have seen that by using our model with polynomial degree greater equal than five we get better results, therefore we use degree 6 in this analysis. We also use the variation of SGD using stochastic mini-batches.

First, a well known problem when using SGD is the selection of the learning rate parameter. For finding the learning rate parameter we executed a set of experiments in which we analyse how the loss function in the Ridge regression give by $2 * \mathbf{X^T}(\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}) + 2 * \lambda * \boldsymbol{\beta}$ in its vector form converges with different learning rate parameters.

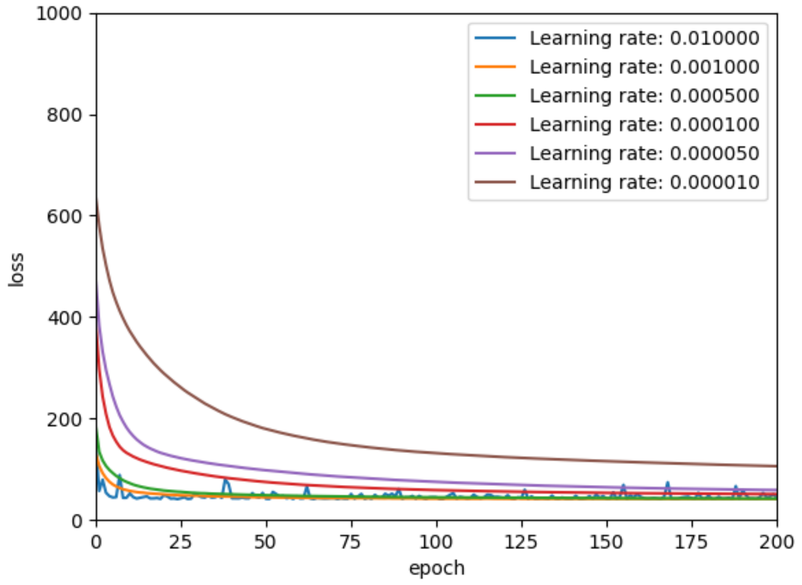


Figure 15: Loss convergence in terms of learning rates along epochs

In image 15 we observe that the smaller the learning rate the longer it takes to converge, it is worth to note that in this case we have fixed the regularisation parameter to 0.1, and size of batch to 32, later we will also vary these other parameters. We can also conclude that 0.01 for learning rate is already too large as in the plot we observe the jumps that the gradient is doing.

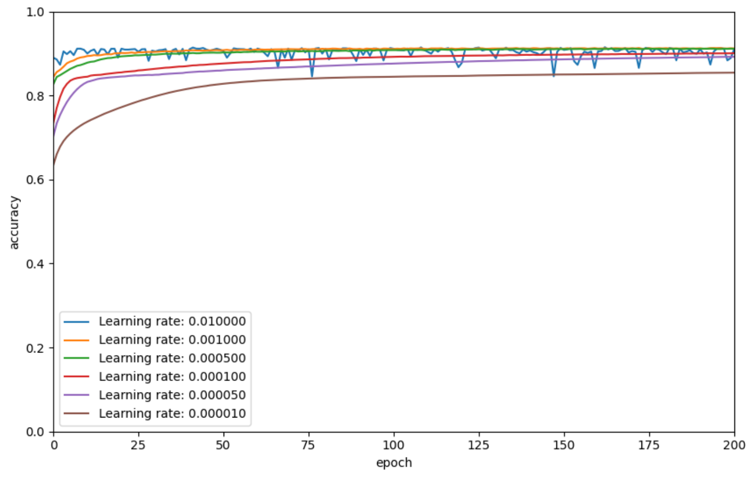


Figure 16: Accuracy in terms of learning rates along epochs

In terms of accuracy, in image 16 we see similar results, with learning rate 0.01 the accuracy increase and decrease along the epochs, nevertheless with learning rates like 0.001 and 0.0005 the convergence is increasing smoothly.

For comparison purposes we display the results of similar experiment using *scikit-learn* library in Figure 17. In this image we can observe the same behaviour although the convergence of accuracy happens for almost all learning rates used in an earlier epoch. The plot shows that in epoch between 50 and 75 with each one of the learning rates considered, except for value of 0.00001, the accuracy achieves the convergence value.
We can also note that with learning rate of value 0.01 the accuracy increase and decrease, this is clearer in earlier epochs.

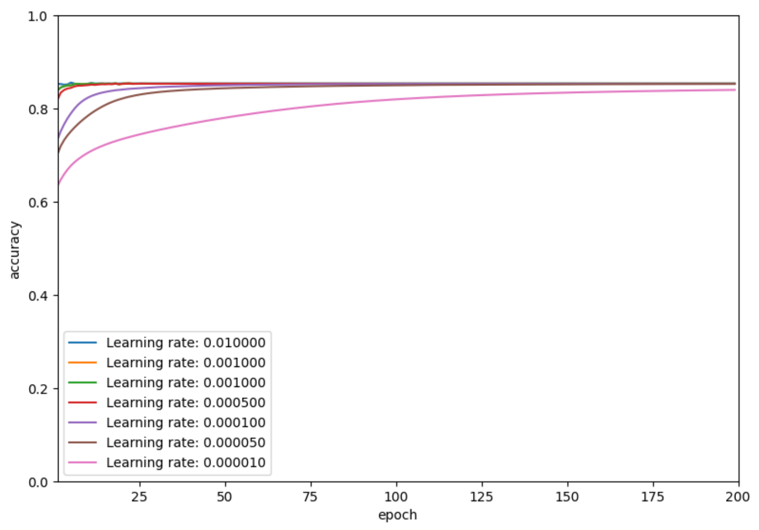


Figure 17: Accuracy in terms of learning rates along epochs with *scikit-learn* library

The experiments we have conducted until now have been with a constant value of learning rate, but in practice, it is necessary to gradually decrease the learning rate over time [23]. According to Goodfellow et al. [23] one of the option for doing so it is to decay the learning rate linearly:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau$$

Where ϵ_k denotes the learning rate a epoch k, $\alpha = \frac{k}{\tau}$ and also learning rate remain constant after epoch τ. For selection of the parameters involved, usually τ is set to the number of iterations required to make a few hundred passes through the training set and ϵ_τ should be set to roughly to 1% the value of ϵ_0 [23].
We set up the experiment for liner decay as explained above with different starting learning rates and $\tau = 200$. The results are shown in Figure 18 where compared with Figure 15 where we have constant learning rate, in this case value of learning rate 0.01 makes the loss function converge without doing the jumps and this is expected because decaying learning rate along epochs makes that the steps in the gradient descent algorithm go slower when close to the minimum.

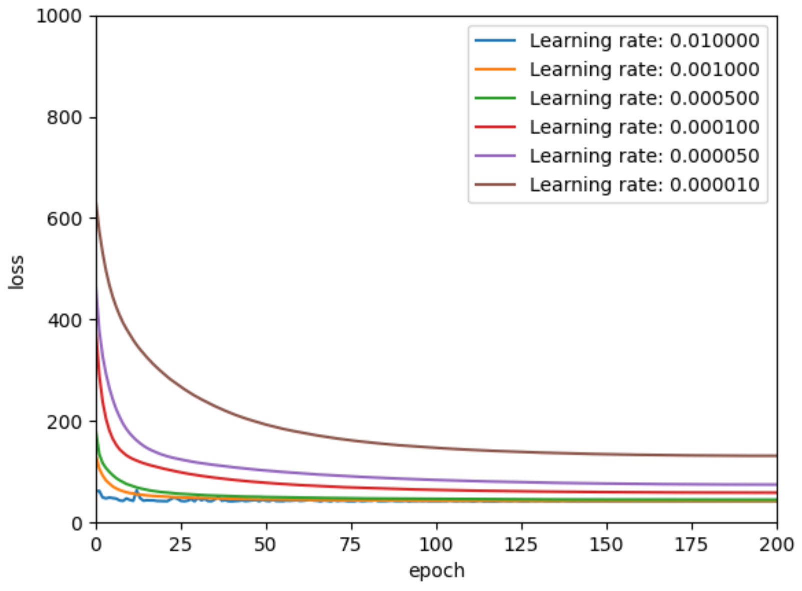


Figure 18: Loss function in terms of learning rate with linear decay along epochs.

Another option for scaling the learning the rate is the momentum algorithm, which accumulates an exponentially decaying moving average of past gradients and continues to move in their direction and common values for this parameter in practice are 0.5, 0.9, 0.99 [23]. We run an experiment with different learning rates and momentum of 0.5. Figure 19 display the results, in which we can also observe that starting with learning rate of 0.01 the jumps in the gradient are also smoother than the ones observed in Figure 15 without any momentum, nevertheless is clear that the jumps in the gradient disappear along epochs only in the Figure 18 with linear decay. Furthermore, in this image we also note that the convergence, for almost all other learning rates, becomes faster, we can corroborate that if we look at epoch 150, where in the plot with momentum the curves practically overlap.

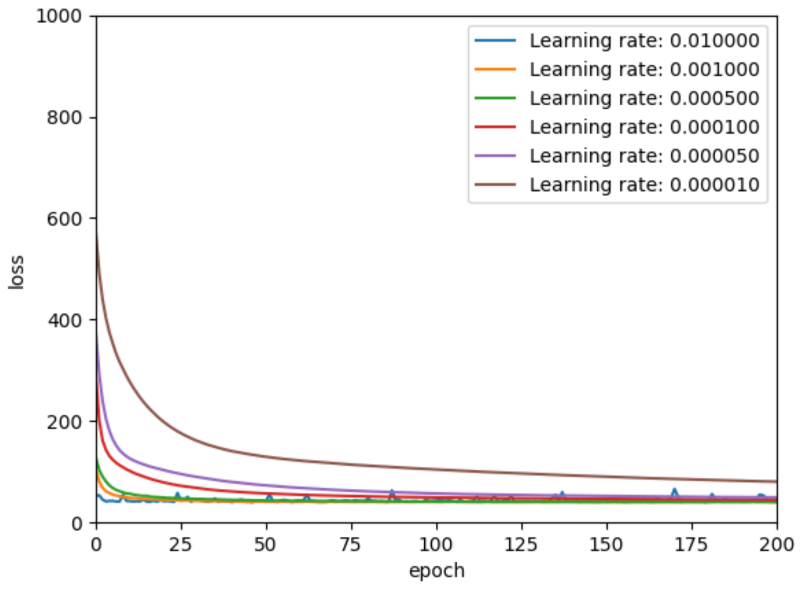


Figure 19: Loss function in terms of learning rate with momentum along epochs.

So far we have seen how the learning rate behaves in the training model with fixed regularisation parameter and fixed batch size, the insights taken from this will help us to make a deeper tune of hyper-parameters on this Ridge regression problem.

In order to objectively search for values of hyper-parameters, that will give us better performance in the machine learning model, we use a grid search approach. In a grid search we fix a set of values for each one of the hyper-parameters and with all combination we compute the performance, the best combination in terms of performance will be the hyper-parameter values for our model. In this case we will consider learning rates in the range $[0.000001, 0.001]$, and regularisation parameters in the range $[0.000001, 2]$ spaced evenly on a log scale. Finally we will measure the performance in terms of the accuracy in the test set.
In Figure 20 we display the results of this experiment, the heat map tell us that the closer to white, the better the performance of the model. For regularisation parameter we can see that values less than 0.1 give us greater performance in combination with learning rates with values larger than 0.0002. A good selection for the hyper-parameters would be learning rate of 0.001 and regularisation parameter between 0.00001 and 0.01, we select the largest learning rate possible because we know that the larger the learning rate the fastest the convergence of accuracy as shown in Figure 16.

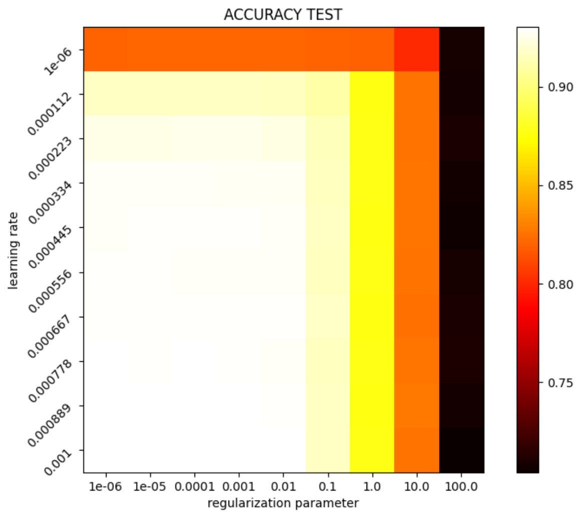


Figure 20: Accuracy in terms of learning rates and regularisation parameter

In order to compare our grid search results we executed similar experiments with *scikit-learn* python library. We used SGDRegressor, with the following parameters in order to simulate better our own algorithm:

- loss: 'squared_error'
- max_iter: 400
- learning_rate: 'constant'
- fit_intercept: False
- penalty: 'l2'
- alpha: each of nine in range $[0.000001, 2]$ spaced evenly on a log scale.
- eta0: each of ten in range $[0.000001, 0.001]$ spaced evenly.

Figure 21 displays the results, in which we can observe similar results as in Figure 20 using our own code. In this grid we can also conclude that learning rates from 0.0002 to 0.001 are the ones that lead to better performance in combination with regularisation parameter values starting from 0.001 while with our code the value 0.01 can be also considered as a good selection. For larger regularisation parameters starting from 1.0 we observe different results for accuracy although with the same tendency of significantly decrease the performance.

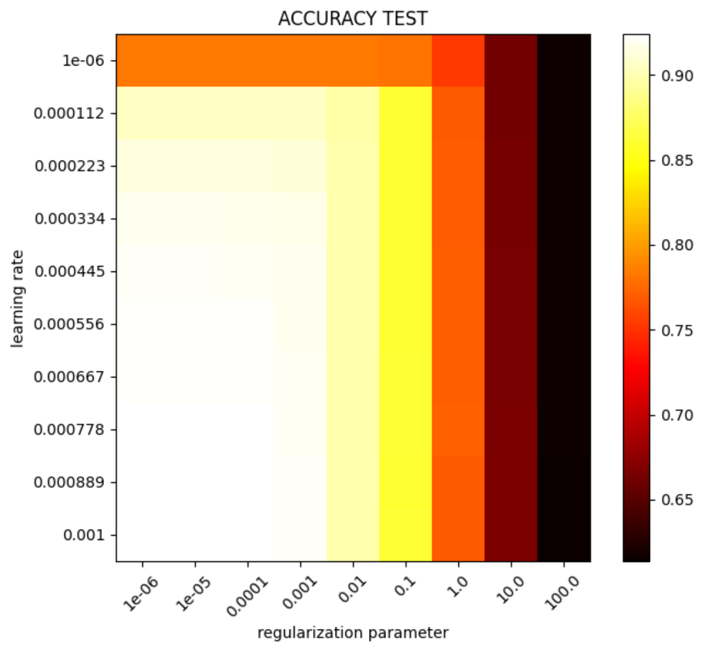


Figure 21: Accuracy in terms of learning rates and regularisation parameter using SGDRegressor from *scikit-learn*

Another parameter to look at is the batch size in the SGD algorithm, for this we fixed learning rate value to 0.001 and regularisation parameter to 0.05, we computed the loss function for different batch sizes along epochs. The results of this experiment can be seen in Figure 22. In this image we can note that batch size is not actually changing loss function along epochs.

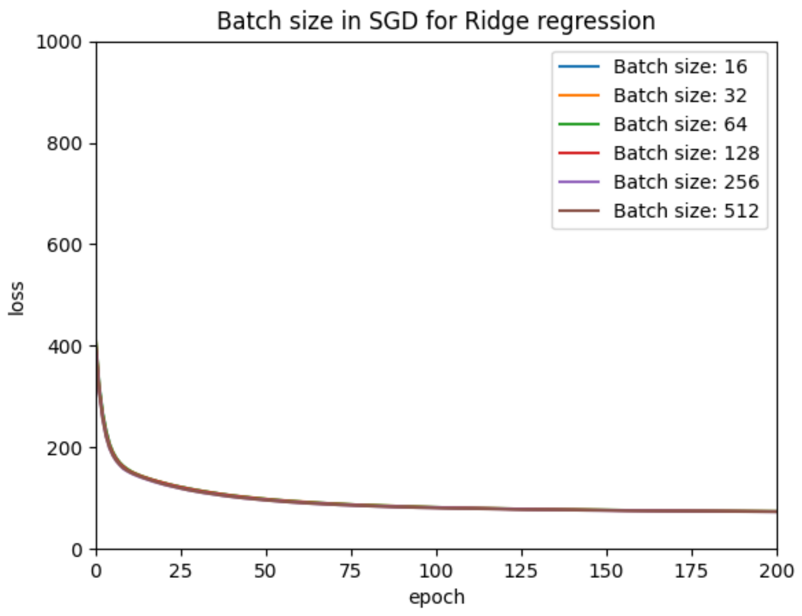


Figure 22: Accuracy in terms of batch size along epochs

4.1.3 LASSO

In order to explore the behaviour of the regularisation parameter on the LASSO regression for the Frank function, we conducted an experiment where we took values of λ of the order of 10^{-5} and used cross-validation(8-fold) resampling method to compute the mean squared error. The results are shown in figure 23. The polynomial degree used is 8 and noise with variance of 0.1 was added to the data.
Results from this experiments suggest that the smaller the regularisation parameter the smaller the MSE. In order to see an inversion of this trending we had to set $\lambda \sim 10^{-7}$. However in this case the difference in the resulting MSE for different λ is of the order of 10^{-4} which is very small value for a function with values between 0 and 1.

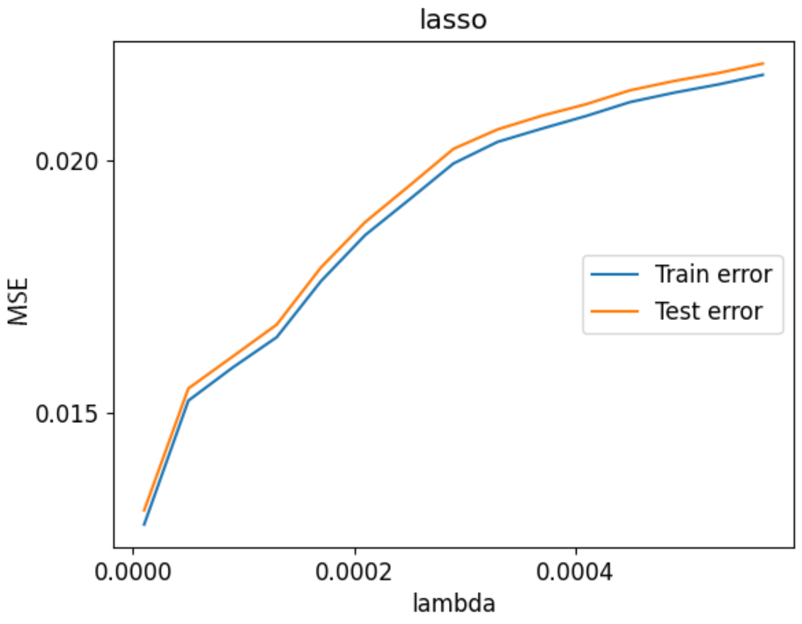


Figure 23: MSE for different values of regularisation parameter in the training and test data coming from the Franke function with noise $N(0, 0.1)$.

For the complexity analysis we used a dataset of 50×50, and we used Cross validation with 8 folds. In order to be able to differ the results from OLS, the penalisation parameter has been set to 0.1.
As we saw for ridge, the introduction of a penalisation parameter brings a much more stability when it comes to overfitting. However it is interesting to note that both the test and train errors do not have the same smooth behaviour we observed with the previous algorithms. This is just a qualitative observation and it may be due to the smaller scale of the error.

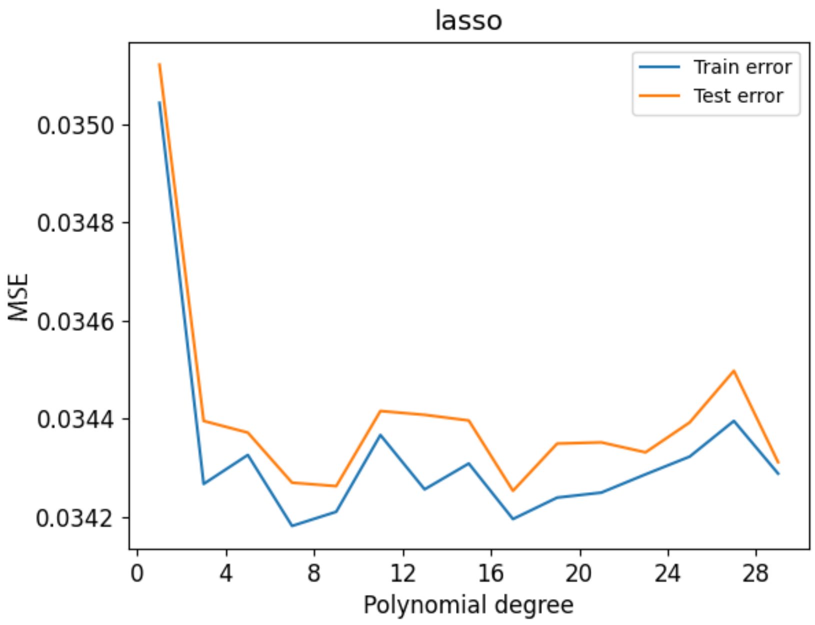


Figure 24: MSE for polynomials of different degree in the training and test data coming from the Franke function with noise $N(0, 0.1)$.

4.1.3.1 Bias-Variance decomposition

Much like in ridge regression, in figure 25 when the bias variance trade off is analysed it is clear that as the regularisation parameter λ increases, the variance decreases, at cost of an increase of the bias. This again is expected behaviour since a larger value of λ means that the regularisation term has more weight in the overall cost function. Thereby, a bias is added to the estimator that counteracts overfitting.

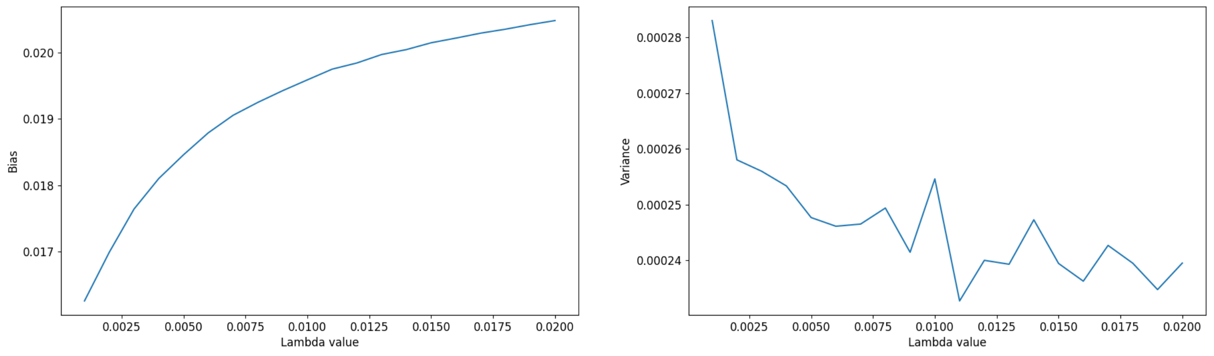


Figure 25: Bias variance decomposition of the MSE for LASSO regression with polynomial degree 4 and a dataset of 30× 30. Plots for bias and variance are split in order to represent the different scales.

4.1.3.2 Resampling

Again we compare the errors and variances for cross validation with different k values. The results are shown in table 3 and are similar to those obtained for ridge regression (compare figure 2). Errors and variances are roughly of the same order of magnitude with slightly higher error values. A peculiar result is that cross validation with 10 folds actually returns the highest variance (one order of magnitude larger than the other cross validation runs) which is likely a stochastic effect.

Table 3: Averaged errors and variances of estimated errors for LASSO regression using bootstrap and cross validation with different numbers of folds. 50 grid points in each dimension. Polynomial of degree 10. Gaussian noise $\sigma = 0.1$.

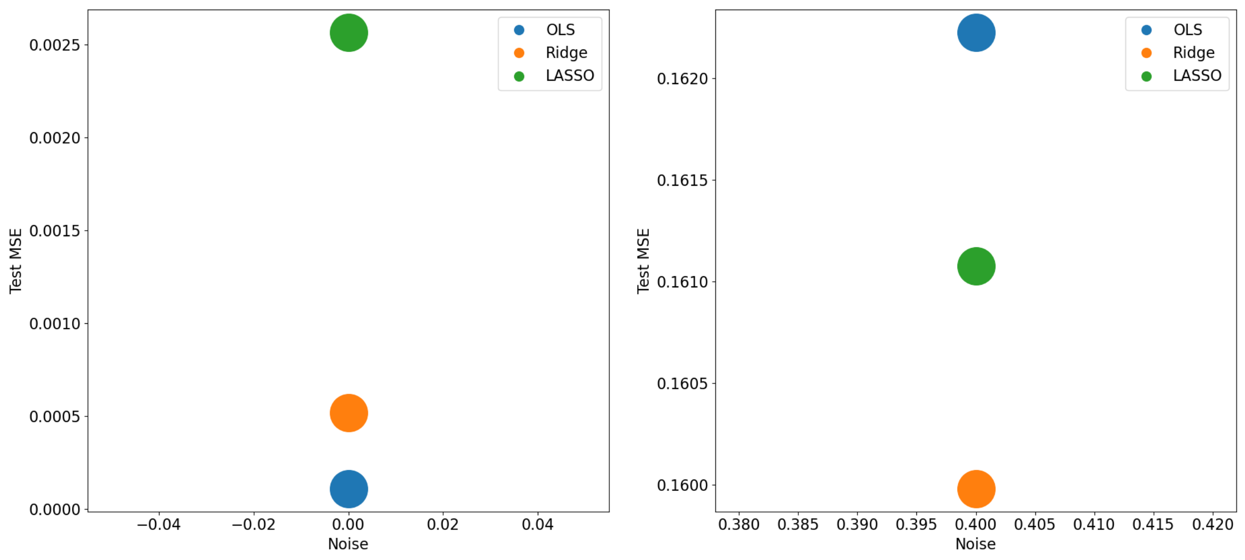
	Bootstrap[†]	Cross validation				
		5	7	8	9	10
E[ϵ]	$1.24 \cdot 10^{-2}$	$1.25 \cdot 10^{-2}$	$1.25 \cdot 10^{-2}$	$1.22 \cdot 10^{-2}$	$1.25 \cdot 10^{-2}$	$1.24 \cdot 10^{-2}$
Var(ϵ)	$1.1 \cdot 10^{-7}$	$1.6 \cdot 10^{-7}$	$4.0 \cdot 10^{-7}$	$5.1 \cdot 10^{-7}$	$8.0 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$

$k = 6$ omitted for clarity. Results were identical to $k = 7$.

[†] Using 100 samples.

4.1.4 Comparison of methods

In figure 26 we report the average test errors for the three investigated regression methods run on data with and without added Gaussian noise. What can be observed is, that if no noise is added to the data OLS outperforms the two regularised methods. Conversely, if noise is added OLS shows the worst performance of the three methods.
These results are in accordance with previous results we obtained for OLS (6) where we could observe that without noise our model needs to be very complex (polynomials of order 20 and higher) in order to produce overfitting effects. This is mainly due to the fact that the Franke function without added noise is smooth. Therefore, the regularised methods do not offer improvement to the test error since overfitting is not a source of error. As we add noise however, we previously saw that the OLS regressor starts overfitting much sooner, fitting to the noise. Here, the regularised regressor lead to a lower test error because they reduce the variability in the output.



(a) Without noise.

(b) With Gaussian noise $\sigma = 0.4$.

Figure 26: Test MSE of OLS, Ridge and LASSO regression for the same dataset with and without Gaussian noise. 50 grid points in each dimension. Polynomial degree 10. Error values estimated using 6-fold cross validation. $\lambda_{Ridge} = 10^{-6}$. $\lambda_{LASSO} = 10^{-5}$.

In real world applications where data will always be noisy LASSO and Ridge regression are better suited compared to OLS which here only performed better due to the simplicity of the investigated data. Ridge and LASSO are comparable in what they do (reducing variance at the cost of added bias) with the difference that LASSO can set parameter values to zero while ridge only dampens their magnitude. This can be of importance if one is interested in promoting sparsity in the parameters. Practically, LASSO is a bit harder to implement since there is no simple derivative of the cost function available so that typically some (sub-)gradient methods have to be used in order to find the LASSO parameters. [6]

4.2 Classification

In this section we investigate classification problems using logistic regression and support vector machines on the Wisconsin breast cancer [2] dataset described in chapter 1.1. For this dataset all features were scaled using min-max-scaling.

4.2.1 Logistic regression

For logistic regression all analyses have been conducted using k-fold cross validation. As a first step a good value for k was determined. To do this cross validation with different values of k was performed and the corresponding variances of the obtained test errors plotted. This is shown in figure 27.

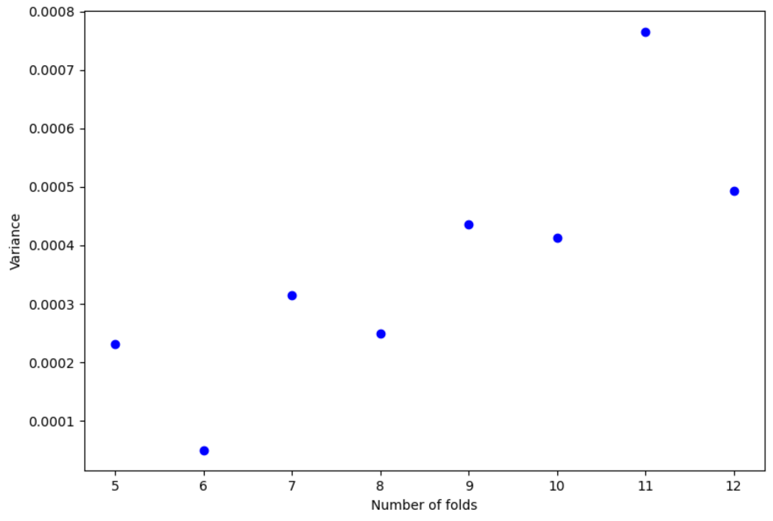


Figure 27: Variance in of fold test errors in dependence on the number of folds.

For $k \geq 7$ there seems to be an increasing trend in fold test error variance. A minimum could be found for $k = 6$. In general however, all values lead to small variances so that the impact of different values of k within the investigated range might not be significant anyway. Nonetheless, for all following calculations cross validation has been used with 6 folds.

Next, a parameter scan for the learning rate was conducted. This was done first using a very coarse grid from 10^{-4} to 1 in 5 steps to get an idea of the magnitude of the optimal learning rate. The results are shown in figure 28.

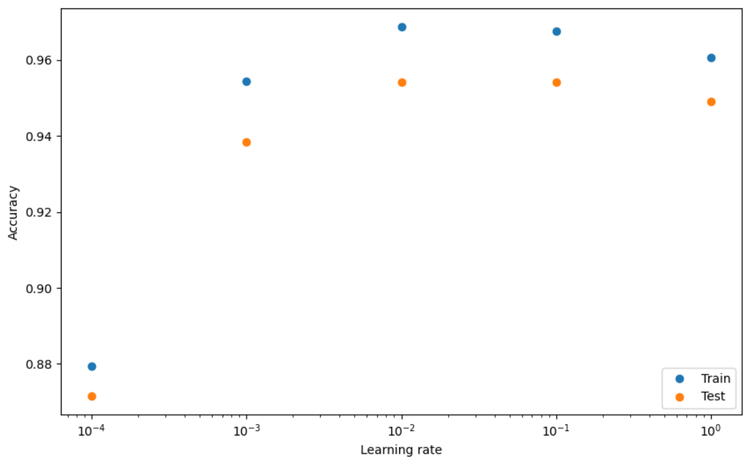


Figure 28: Hyperparameter scan for the learning rate from 10^{-4} to 1 (logarithmic scale).

The results suggest that the optimal value for the learning rate lies somewhere between 10^{-3} to 10^{-1}. Interesting to note is the significantly worse performance for a learning rate of 10^{-4}.

The reason for this is that the gradient descent algorithm reached the maximum number of allowed iterations before converging.

To get a more precise estimate of the optimal learning rate a finer scan in the range from 10^{-3} to $5 \cdot 10^{-1}$ in 28 steps was conducted. The results are shown in figure 29.

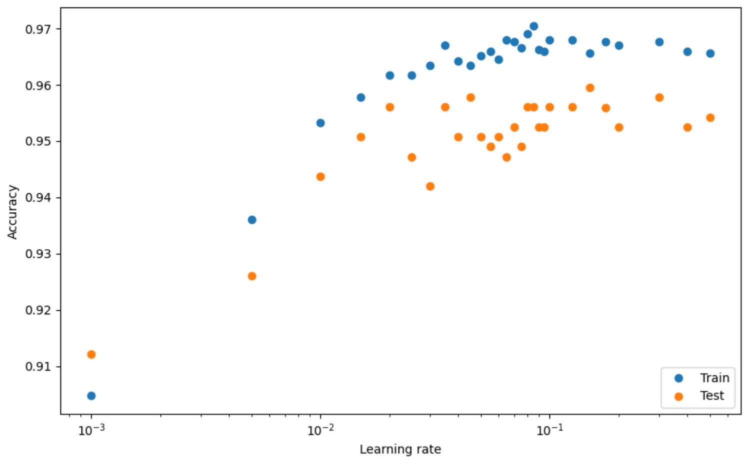


Figure 29: Hyperparameter scan for the learning rate from 10^{-3} to $5 \cdot 10^{-1}$ (logarithmic scale).

This data seems to suggest that the optimal value for the learning rate lies around 0.085 (maximum in training and among the best values in testing). Worth noting is the fact that there is quite some noise in the values of the test accuracies. All following runs have been conducted using the obtained optimal value for the learning rate 0.085.

Also a scan for the momentum parameter was conducted. Again first in a coarse manner with values from 10^{-4} to 10^{-1} in 4 steps. The results are shown in 30.

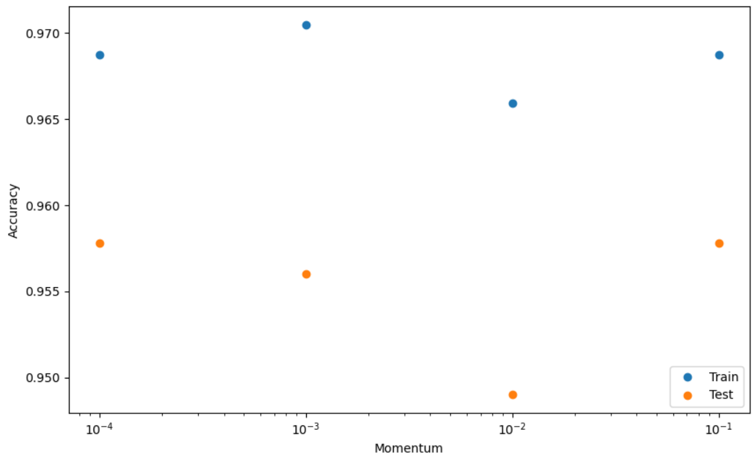


Figure 30: Hyperparameter scan for the momentum with a set learning rate of 0.085. Values from 10^{-4} to 1 (logarithmic scale).

The results indicate that the optimal value for the momentum lies around 10^{-3}. Therefore, a finer scan in the range from 10^{-4} to 10^{-2} in 15 steps has been conducted. The results are shown in figure 31

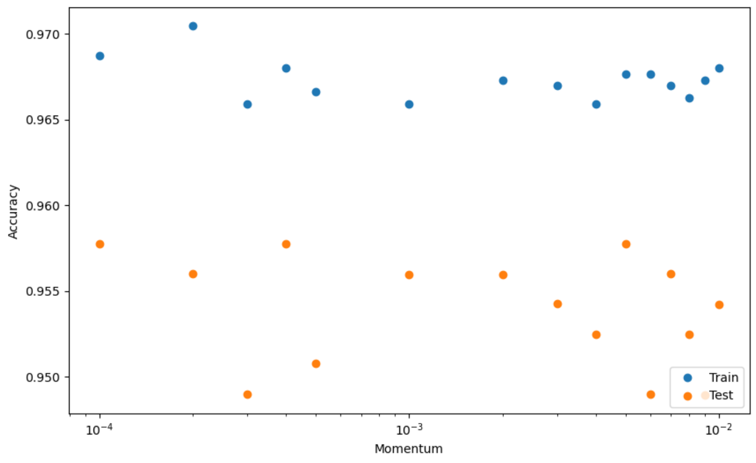


Figure 31: Hyperparameter scan for the momentum with a set learning rate of 0.085. Values from 10^{-4} to 10^{-2} (logarithmic scale).

All values for the momentum in this range seem to lead to similar performances. The best value is obtained for a momentum of 0.0002 which leads to the same accuracy obtained for a learning rate of 0.085 and no momentum. The reason for this could be that the solution space is sufficiently well-behaved in the sense that there are no patches of the solution landscape where the gradient is very small so that momentum cannot offer much improvement over a simple implementation.

Next, also the introduction of a l_2 regularisation term has been investigated. Again, a scan for the regularisation parameter λ has been performed for values in the range from 0 to 1. The results are shown in figure 32.

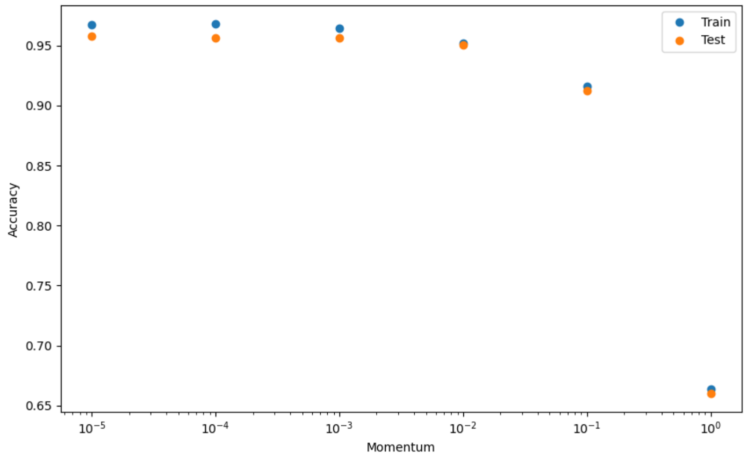


Figure 32: Hyperparameter scan for the regularisation parameter with a set learning rate of 0.085 and no momentum. Values from 10^{-5} to 10^{-1} (logarithmic scale).

From these results it seems that the optimal value should lie somewhere between 10^{-5} and 10^{-4}. An appropriate finer scan has been conducted and its results are shown in figure 33. Note that the accuracy for a value of 1 is very low. This is due to the fact that with a value of one the contribution of the penalty term will be so high, that it dominates the overall cost. Therefore, setting all parameters to zero or very low values will lead to the lowest cost. Ultimately this means that the predictive power of the model is heavily impaired.

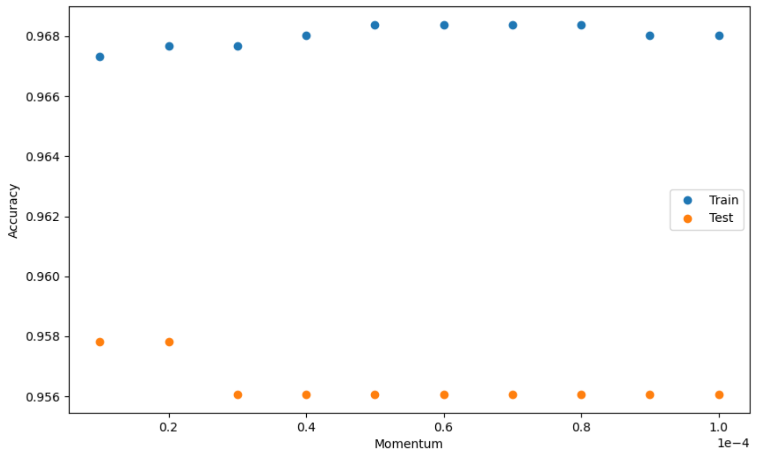


Figure 33: Hyperparameter scan for the regularisation parameter with a set learning rate of 0.085 and no momentum. Values from 10^{-5} to 10^{-4}.

In this range all values are again very similar. The highest training accuracy is obtained with λ values in the range from $5 \cdot 10^{-5}$ to $8 \cdot 10^{-5}$. The highest test accuracy is obtained for $\lambda = 2 \cdot 10^{-5}$. The identical accuracies for some of the settings are due to the discrete nature of classification problems.

Finally, a comparison of our own implementation with *scikit-learn* has been conducted. For twenty random shuffles of the dataset analyses with both methods using the aforementioned hyperparameter settings and 6-fold cross validation have been conducted. For *scikit-learn* we chose the stochastic average gradient (SAG) solver. The experiment-wise results are shown in figure 34.

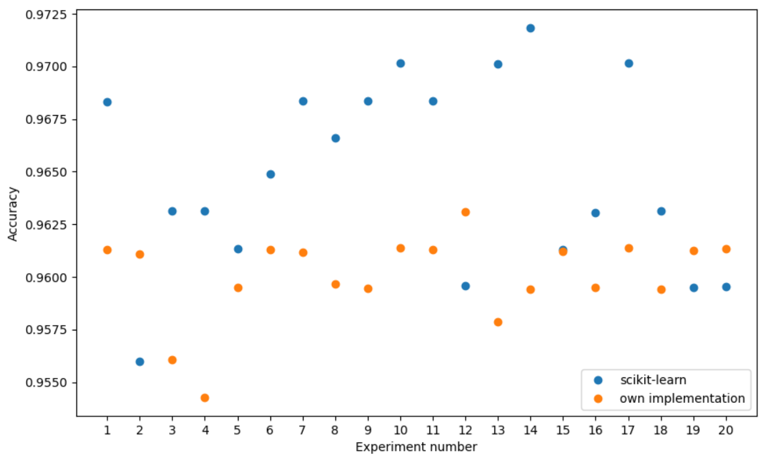


Figure 34: Comparison of *scikit-learn* and our own implementation of logistic regression for the Wisconsin dataset. All values obtained through 6-fold cross validation with learning rate of 0.085 and l_2 regularisation parameter of $8 \cdot 10^{-5}$.

From the obtained data it is apparent that both methods yield very similar results, always providing results with accuracies close to 96%. However, it seems that *scikit-learn* tends to outperform our own implementation by a very small margin. Averaging of the individual errors gives the following results shown in table 4.

Table 4: Average accuracies over 20 random shuffles of the data.

	scikit-learn[†]	own implementation
Accuracy	0.965	0.960

[†] Using the SAG solver.

This shows that on average *scikit-learn* provides better accuracies by an absolute value of 0.5%. These small differences are probably due to slight differences in the gradient solvers of *scikit-learn* and our own implementation used for the fitting of parameters in logistic regression. *scikit-learn* uses the SAG solver in which the gradient is only calculated with respect to a single training point at each iteration. [24] We on the other hand have implemented a batch stochastic gradient descent as described in chapter 3.
Furthermore, *scikit-learn* uses a separate function to calculate the learning rate, so that a learning rate can not be set manually. Therefore, we had to use different learning rates for *scikit-learn* and our own implementation which is another factor that might contributed to the observed deviations. Also, the *scikit-learn* implementation of the SAG solver does not include a momentum term that we use in our own implementation.

4.2.2 Support vector machines

As a first step we employ a standard support vector machine (SVM) approach with a linear kernel for the dataset. We apply the same scaling (min-max) as for logestic regression and again use 6-fold cross validation. Averaging of 20 independent runs we obtain an accuracy of 97.6% which means that a linear SVM outperforms the logistic regression by an absolute value of about 1%. (compare table 4)
The *scikit-learn* implementation uses a regularisation parameter C that controls the "hardness" of the margin. High values of C correspond to a harder margin whereas lower values correspond to a softer margin. With the standard settings C is set to 1. In analogy to the experiments for logistic regression we look into the dependence of the accuracy on this regularisation parameter. We try values in the range from 0.001 to 1000. The results are shown in figure 35.

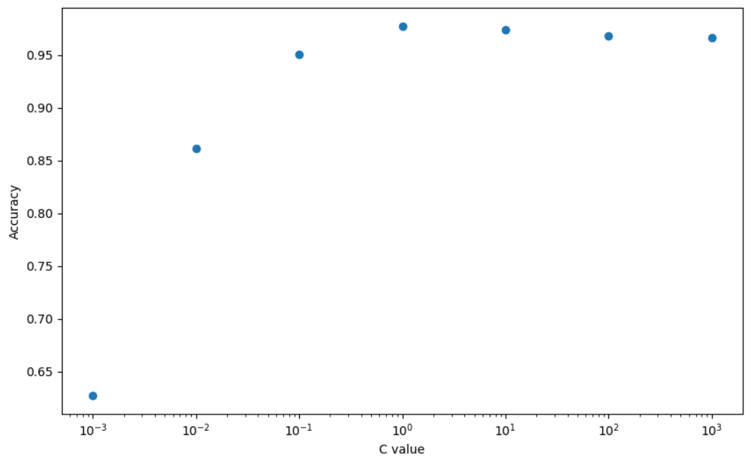


Figure 35: SVM hyperparameter scan for the value of C in the range from 0.001 to 1000 using a linear kernel (logarithmic scale). Test errors obtained through 6-fold cross validation.

These results indicate that a value of $C = 1$ is actually the optimal setting for this dataset. Furthermore, we can observe that for very low values of C (i.e. soft margins) the average accuracy greatly decreases. This is expected behaviour for the SVM because if we allow too many miss classifications in the fitting of training data the resulting affine subspace separating the positive and negative cases will be inaccurate leading to large errors on the test set. This effect is comparable to using large λ values in regularised regression which typically also leads to bad performance.
On the other hand increasing C by the same number of magnitudes does not lead to a comparable deterioration of accuracy. This is due to the fact that the Wisconsin breast cancer set itself is completely linearly separable in the 30 features [2] which means that even a completely "hard" classifier should return good results.

Additionally we also looked at the impact of using a polynomial kernel compared to a linear one. For this we used the parameterisation $C = 1$ which was found to be optimal, at least for the linear kernel. The results for polynomial kernels up to fifth degree are shown in figure 36.

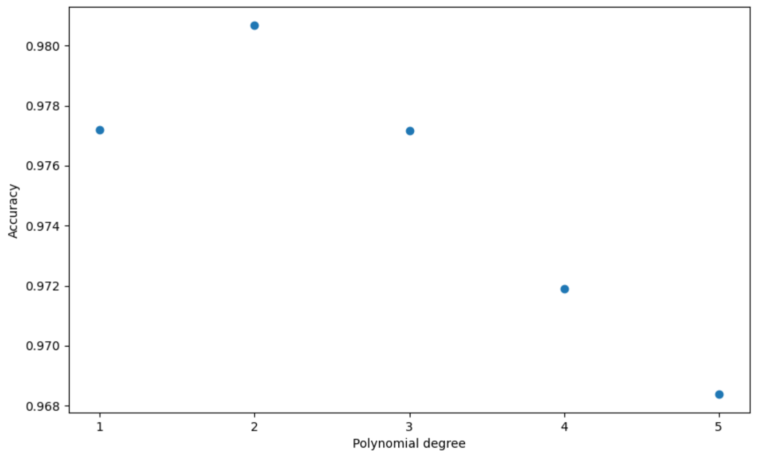


Figure 36: SVM hyperparameter scan for a polynomial kernel using degrees in 1 to 5. Using $C = 1$. Test errors obtained through 6-fold cross validation.

From this it can be seen that a polynomial kernel of second degree actually increases the performance by about 0.5% compared to a linear kernel. Going to higher degree polynomials however, leads to a reduction in test accuracies which is simply an overfitting effect similar to what we have seen for regression.

5 Conclusion

Overall, for both regression and classification we could achieve good results with the investigated methods. In the case of regression we found that plain OLS regression outperforms ridge and LASSO regression, at least if no additional noise is added to the targets.
Generally we noticed that the algorithms that implement a penalisation term such as LASSO and ridge regression are more robust to overfitting, however this comes with an associated cost. On the one hand for data where overfitting is unlikely and not a problem, the results will be worse compared to OLS regression as we have seen for the Franke function. On the other hand this also means introducing another hyperparameter that has to be tuned in order to obtain good results.
The comparably bad performance of the LASSO regressor might be explained by the fact that for LASSO no simple closed form expression for the beta values is available. Thus, numerical optimisation using (sub-)gradient methods has to be used which might be a source of error. In our case this was *scipy*'s `minimize` function.
Despite the generally good performance of OLS and ridge regression, these methods might not be viable for problems with many data points where the corresponding model matrices are large and thus obtaining the (pseudo-)inverse can be computationally unfeasible. For situations like this the alternative of using a cost driven approach with a stochastic gradient descent is well suited. Nevertheless when using this kind of algorithms we should consider that more hyper-parameters tuning is needed. In stochastic gradient descent, the selection of learning rate is important and because there is not a deterministic way of finding the appropriated value in general, therefore we should explore the performance of different values in combination with the hyper-parameters of our model. In our case, when analysing ridge regression with SGD, we found that using grid search for tuning regularisation parameter and learning rate was necessary in order to find the good combinations that makes the model perform better.
One thing this study highlighted is that, when using numerical approximations ridge regression tends to converge earlier compared with plain OLS.

For classification we obtained very good accuracies on the Wisconsin breast cancer dataset using both investigated methods ($\geq 96\%$). For logistic regression we found a value of 0.085 for the learning rate to be optimal while momentum and the regularisation parameter λ did not seem to have large impact on the overall performance. In comparison with *scikit-learn* our own implementation was slightly worse, most probably due to differences in the used gradient optimisation schemes. With support vector machines we observed even better average accuracies and by using a quadratic kernel obtained an average accuracy of 98%.
These highly accurate results could be achieved with methods that are both fairly easy to implement. Furthermore, the extension to multiclass cases for both methods is possible by using the softmax function in logistic regression [15] and by reduction to multiple binary classification problems in support vector machines [25].[1] Therefore, these methods are also

[1]This splitting approach in support vector machines for multiple cases also motivates the boosting ensemble learning approach in which many weak learners are combined to form one strong learner that makes predictions. [26]

fit for more complex, multi-classification tasks.

Resampling methods proved to be useful for getting better error estimates of specific models and getting an idea of the variance in its predictions. The former is especially useful for problems where data is sparse and the latter can be useful for troubleshooting developed models. All this without being very involved algorithmically. In particular bootstrapping proved to be very simple to implement and could be used to estimate bias and variance of the investigated regressors. A disadvantage of bootstrapping however is, that it requires somewhat high computational effort in order to obtain enough samples to produce meaningful error estimates. Especially for large datasets where training is especially expensive this might be prohibitive. For cross validation this is less of a problem since the value for k can be set to rather small values. Also its property that every sample is used exactly once for testing and $k-1$ times for training can be advantageous. Our results of multiple cross validation runs with different k values showed that the influence on the average error is marginal, at least for this dataset and the used fold numbers (5 to 10).

With all this in mind the presented results should still be taken with some caution since the investigated problems are rather simple and "well-behaved". The Franke function is a smooth function and the Wisconsin breast cancer dataset is linearly separable in the 30 features [2] and only binary in the targets. Another interesting fact about the Wisconsin breast cancer dataset is that there is a $40:60$ ratio of malignant and benign data points. But even without weighing data points to account for the imbalance in the dataset very good accuracies could be achieved, which also shows that it is an "easy" dataset to work with.
For more complex problems the investigated methods might not be enough in order to achieve satisfactory results and more involved approaches such as neural networks might be required. Also other pre-processing steps such as reweighing data points and dimension reduction (for example through autoencoders [27]) might become crucial.

List of Figures

List of Tables

References

[1] Richard Franke. A critical comparison of some methods for interpolation of scattered data. Technical report, NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1979.

[2] W Nick Street, William H Wolberg, and Olvi L Mangasarian. Nuclear feature extraction for breast tumor diagnosis. In *Biomedical image processing and biomedical visualization*, volume 1905, pages 861–870. International Society for Optics and Photonics, 1993.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[4] Morten Hjorth-Jensen. Applied data analysis and machine learning, 2021.

[5] Fumino Hayashi. Econometrics. *Princeton University Press*, 2000.

[6] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[7] L. R. Turner. Inverse of the vandermonde matrix with applications, 1966.

[8] Joel Grus. *Data science from scratch: first principles with python*. O'Reilly Media, 2019.

[9] Stephen Marsland. *Machine learning: an algorithmic perspective*. Chapman and Hall/CRC, 2011.

[10] Benjamin Recht and Christopher Ré. Beneath the valley of the noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences. *arXiv preprint arXiv:1202.4184*, 2012.

[11] Léon Bottou. Curiously fast convergence of some stochastic gradient descent algorithms. In *Proceedings of the symposium on learning and data science, Paris*, volume 8, pages 2624–2633, 2009.

[12] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-rdbms analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 325–336, 2012.

[13] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[15] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[16] P. McCullagh and J.A. Nelder FRS. *Generalized Linear Models*. Chapman and Hall, 1989.

[17] James Mercer. Xvi. functions of positive and negative type, and their connection the theory of integral equations. *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character*, 209(441-458):415–446, 1909.

[18] Johar Ashfaque and Amer Iqbal. Introduction to support vector machines and kernel methods. 04 2019.

[19] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[21] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[22] Marco Taboga. ”ridge regression”, lectures on probability theory and mathematical statistics. *Kindle Direct Publishing*, 2019.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[24] Nicolas Le Roux, Mark Schmidt, and Francis Bach. A stochastic gradient method with an exponential convergence rate for finite training sets. *arXiv preprint arXiv:1202.6258*, 2012.

[25] Kai bo Duan and S. Sathiya Keerthi. Which is the best multiclass svm method? an empirical study. In *Proceedings of the Sixth International Workshop on Multiple Classifier Systems*, pages 278–285, 2005.

[26] Robert E Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[27] Mark A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, 37(2):233–243, 1991.