CompSci

Faculty of Mathematics and Natural Sciences at the University of Oslo

Project 1

Amandine Kaiser, Anthony Val Compasano, Harish P. Jain

December 2021

Abstract

Two dimensional polynomials of various orders are fit to noisy Franke function using linear regression methods. The regression methods are ordinary least squares (OLS) and shrinkage methods like ridge and lasso regression. The parameters are either found through analytical methods, scikit functions, or by optimizing a cost function using stochastic gradient descent. Next, logistic regression and support vector machines are used to learn a classification model to predict whether a tumor is benign or malignant based on Wisconsin Breast Cancer Dataset. Various statistics are utilized to determine the performance of the regression and classification models.

Contents

1 Introduction

The focus of this report is to discuss some commonly used supervised learning techniques.

We start with introducing the linear regression model. It is a model that predicts a response given an input vector and introduces a learnable parameter that is linear with respect to the input variable. Even though it is called linear, it can be used to model nonlinear functions. A combination of fixed nonlinear functions of input variables serves as a basis function to extend linear regression capability. To estimate the statistical model, we use two commonly used techniques. The first technique uses an exact equation for the learnable parameter obtained by minimizing the mean-squared error. This can be computationally costly since it involves calculating the matrix inverse. The second approach, commonly used for large data sets, uses stochastic gradient descent. We also explore both methods using shrinkage methods with L1 and L2 regularization. Ultimately, we aim to obtain a statistical model that can map a new input variable to a response with acceptable accuracy.

We explore the power of linear regression using a typical test function such as the Franke function. In addition, we use two sampling techniques to analyze the various linear regression model. For bias and variance analysis, we use the bootstrap sampling method. To evaluate the MSE of the linear regression models, we use k-fold cross-validation.

Another type of supervised learning is classification models. For this case, we explore logistic regression and support vector machines. As data, we use the Wisconsin breast cancer data. The optimization or training uses stochastic gradient descent with mini-batches for logistic regression. The classification models were trained on the cancer data using various hyperparameters such as minibatch size, number of epochs, and learning rate. For classification of the continuous-valued output, the logistic regression uses a decision boundary. In the case of SVM classification, we utilize the built-in functionality of scikit.

For each type of supervised learning in the following section, we start with a theoretical discussion, followed by results and a discussion of them.

The code can be found in this GitHub repository.

2 Linear Regression

2.1 Method

Consider that we have a data set of inputs and their corresponding output. Regression is a process of arriving at a relation between the independent input

variables and the dependent output variables. In the field of statistical learning, regression is considered to be a supervised learning method. This is because the model parameters generated in regression are learned from labeled data which is data that comprises both inputs and their corresponding output, i.e the labels. In this report, we assume that there are multiple input variables and a single output variable for each member of the data set.

The dataset with n members is defined as $\mathcal{D} := [(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$. Where x_i and y_i are the i^{th} input and output respectively. Let, m be the number of components of the input such that $x_i := (x_{i1}, x_{i2}, ..., x_{im})$. In linear regression we assume that the data is of the form

$$y_i = 1 \cdot \beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + ... + x_{im}\beta_m + \epsilon_i \tag{1}$$

where $\beta_0, \beta_1, \ldots, \beta_m$ are unknown fixed regression parameters and ϵ_i is a random error or noise process consisting of independent identically (iid) normally distributed variables with mean zero and variance σ^2 [7]. Equation 1 can be rewritten as $y = \beta\phi(x) + \epsilon$ with

$$\phi(x) := [1, x_{i,1}, x_{i,2}, \ldots, x_{i,m}]. \tag{2}$$

Moreover, it can be shown that $\mathbb{E}[y] = \beta\phi(x)$ and $var(y) = \sigma^2$. Therefore, $y \sim \mathcal{N}(\beta\phi(x), \sigma^2)$ [6].

In linear regression, we seek to fit a best-fit line between input, or some transformation of the input, and the output. This best-fit line is chosen such that it minimizes an error or a cost function. The best fit line is a function of input variables and its value is called the predicted value or the fit. We denote this with $\hat{y}$. The linear regression model has the form:

$$\hat{y}_i = 1 \cdot \beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + ... + x_{im}\beta_m \tag{3}$$

The β's are chosen such that they minimize a cost function. If this cost function is chosen to be the mean squared error (MSE) then we call the regression *least squares* regression. The mean squared error is defined as

$$\mathcal{C} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4}$$

By extending the input such that $x_i = (1_i, x_{i1}, x_{i2}, ..., x_{im})$, i.e. the first entry of x_i is a vector which only contains 1, and denoting the parameter vector by $\beta = (\beta_0, \beta_1, ..., \beta_m)$, we can express the predicted value as an inner product such that

$$\hat{y}_i = x_i \cdot \beta = {x_i}^T\beta.$$

Moreover, we can express the set of predicted outputs as a vector which allows us to write the previous equation as a matrix-vector product:

$$\hat{y} = X\beta \tag{5}$$

where, $\hat{y}$, X and β have dimensions $n \times 1$, $n \times (m+1)$ and $(m+1) \times 1$. X is called design matrix. However, such a description is only useful if the output is a linear or near-linear function of the input variables. We can make the model more "complex" by allowing higher powers of input variables and their products to be the features. One possibility is to consider polynomial functions of the input variables. For simplicity, we consider input vectors where $m = 2$ as they are the only ones relevant for this project. Then, we can express the predicted output $\hat{y}$ as a polynomial function of order p such that

$$\hat{y}_i = \beta_0 + \sum_{k=1}^{k=p} \sum_{l=0}^{l=p} \beta_{\frac{k(k+1)}{2}+l} x_{i1}^{p-l} x_{i2}^{l} \tag{6}$$

which is equivalent to

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{n \times 1} = \underbrace{\begin{bmatrix} 1 & x_{11} & x_{12} & x_{11}^2 & \dots & x_{11}^p & x_{11}^{p-1}x_{12} & \dots & x_{11}x_{12}^{p-1} & x_{12}^p \\ 1 & x_{21} & x_{22} & x_{21}^2 & \dots & x_{21}^p & x_{21}^{p-1}x_{22} & \dots & x_{21}x_{22}^{p-1} & x_{22}^p \\ \vdots & & & & & \ddots & & & & \vdots \\ 1 & x_{n1} & x_{n2} & x_{n1}^2 & \dots & x_{n1}^p & x_{n1}^{p-1}x_{n2} & \dots & x_{n1}x_{n2}^{p-1} & x_{n2}^p \end{bmatrix}}_{n \times \frac{p^2+5p}{2}} \underbrace{\begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{\frac{p^2+3p}{2}} \end{bmatrix}}_{\frac{p^2+5p}{2} \times 1}. \tag{7}$$

For clarity if $p = 2$

$$\hat{y}_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2}.$$

We see that the output vector is linear in the vector of parameters even though it is nonlinear in the features. Note that p determines the model complexity and therefore the number of columns in the design matrix and the number of parameters. The output vector though depends only on the number of inputs. By changing p we are changing the basis with which we describe y_i.

As mentioned before our aim is to find β such that the cost function is minimized. We start by rewriting the equation 4:

$$\mathcal{C}(\beta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i(\beta))^2 = \frac{1}{n} (y - \hat{y})^T (y - \hat{y}) = \frac{1}{n} (y - X\beta)^T (y - X\beta). \tag{8}$$

The cost function is at its minimum when its gradient with respect to all parameters is 0, i.e.

$$\nabla_\beta \mathcal{C}(\beta) = X^T (y - X\beta) = 0.$$

Assuming that X has full column rank, and hence $X^T X$ is positive definite, solving the above equation provides us with the optimal parameters as follows

$$\beta = (X^T X)^{-1} X^T y. \tag{9}$$

The cost function for a linear regression problem is convex. Hence, there is only one global extremum. This is not proven in this report.

In the case where $X^T X$ is near singular or singular calculating the pseudoinverse (specifically the Moore-Penrose inverse) is a very useful and well-known method. It calculates the inverse by using the Singular Value Decomposition algorithm. Moreover, this approach is more efficient than computing the regular inverse [3].

An important step when calculating the best suited model for given input and output data is the splitting of the dataset $\mathcal{D}$ into a training dataset $\mathcal{D}_{train} = \{\boldsymbol{x}_{train}, y_{train}\}$ and a test dataset $\mathcal{D}_{test} = \{\boldsymbol{x}_{test}, y_{test}\}$. The ratio of the number of data points in the $\mathcal{D}_{test}$ to the number of data points in the $\mathcal{D}_{train}$ is called the test ratio r. The model parameters are learned on the training dataset in the training phase. While, the performance of the model is assessed by various indicators that compare the output of fitting the model, on the input of the test set, with the output of the test set.

Let the mean squared error(MSE) be

$$MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{10}$$

the squaring is important so that the positive and negative errors do not cancel each other. Certainly, the lower the value of MSE the closer is our prediction($\hat{y}$) to the ground truth(y). The training MSE is given as $MSE_{train} = MSE(y_{train}, \hat{y}_{train})$ and the testing MSE as $MSE_{test} = MSE(y_{test}, \hat{y}_{test})$.
The MSE can be decomposed into bias and variance as

$$MSE = \underbrace{\mathbb{E}[(y - \mathbb{E}[\hat{y}])^2]}_{\text{bias}} + \underbrace{\mathbb{E}[(\hat{y} - \mathbb{E}[\hat{y}])^2]}_{\text{variance}} + \sigma^2 \tag{11}$$

Let the R^2 score be

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y})^2}{\sum_{i=1}^{n}(y_i - \mathbb{E}[\hat{y}])^2}. \tag{12}$$

It measures the ability of the model to capture the true variance relative to the actual variance. When the model fits the data perfectly $R^2 = 1$. While, the least possible value of R^2_{train} is 0, but R^2_{test} can be negative.
The difference in the MSE values for the training and testing set determines the degree of generalization of the model from trained data to new unseen data. The average generalization error is [5]

$$|MSE_{train} - MSE_{test}| = 2\sigma^2\frac{m}{n}.$$

If $m >> n$ then the model is not generalising, i.e. learning. In addition, the error can be large if the intrinsic noise σ^2 in the data is large. To counter this problem, regularisation is performed. In regularisation, the parameter values are penalized. In ridge regression, the L^2 norm of the parameter vector is penalized

and in lasso regression, the L^1 norm of the parameter vector is penalized. This is done by modifying the cost functions as follows

$$\mathcal{C}_{ridge}(\beta) = \frac{1}{n}(y - X\beta)^T(y - X\beta) + \lambda||\beta||_2^2, \tag{13}$$

$$\mathcal{C}_{lasso}(\beta) = \frac{1}{n}(y - X\beta)^T(y - X\beta) + \lambda||\beta||_1. \tag{14}$$

2.1.1 Feature Scaling

In general, machine learning algorithms do not perform well if the scales of the data vary a lot. There exist different scaling methods. We focus on min-max scaling, also known as min-max normalization. In this approach, the values are scaled and shifted such that they range from 0 to 1, i.e.

$$x^{'} = \frac{x - min(x)}{max(x) - min(x)}.$$

2.1.2 Need for resampling

Typically, data contains outliers. These outliers can be erroneously obtained data. They can also be the true data that sometimes deviates from the established patterns of the system due to intrinsic noise. The presence of outliers in both the training and the testing set can either influence the fitting of parameters or the statistics. Therefore, it is essential to subdue the effects of outliers using methods that usually involve multiple iterations of training and testing on different subsets of the data. Moreover, the statistics obtained for a single fit are not necessarily the true indicator of the model performance. There is intrinsic variance in those statistics. To ameliorate the effect of variance we utilize resampling techniques. In resampling, we repeatedly learn the model with a different training set and find their statistical performance on the different test sets. In the end, the mean of the statistics of individual iterations is taken to be the final statistic.

In the following sections we discuss two of the most commonly used methods: the bootstrap algorithm and cross-validation.

2.1.3 Bootstrap Algorithm

The bootstrap algorithm is a resampling technique that uses sampling with replacement. There are multiple variants of the bootstrap method. We use one that is distinct from the lecture notes.
The steps of the algorithm are as follows:

1. Let the number of bootstrap iterations on given data be: n_{boots}
2. For each bootstrap iteration n_{bi}:

(a) Randomly divide the input data consisting of n samples into two categories. The two categories are the training and test set. The training set has $(1-r)n$ number of samples which are randomly selected with replacement. The rest of the samples then make up the testing set. Therefore the number of samples in the testing set is greater than or equal to rn. Here, r is the testing ratio.

(b) Train parameters for the training set and fit them to the testing set.

(c) Compute statistics on the training set: MSE_i, R_i^2

3. The final statistics of the bootstrap algorithm are the mean of the statistics of all the bootstrap iterations.

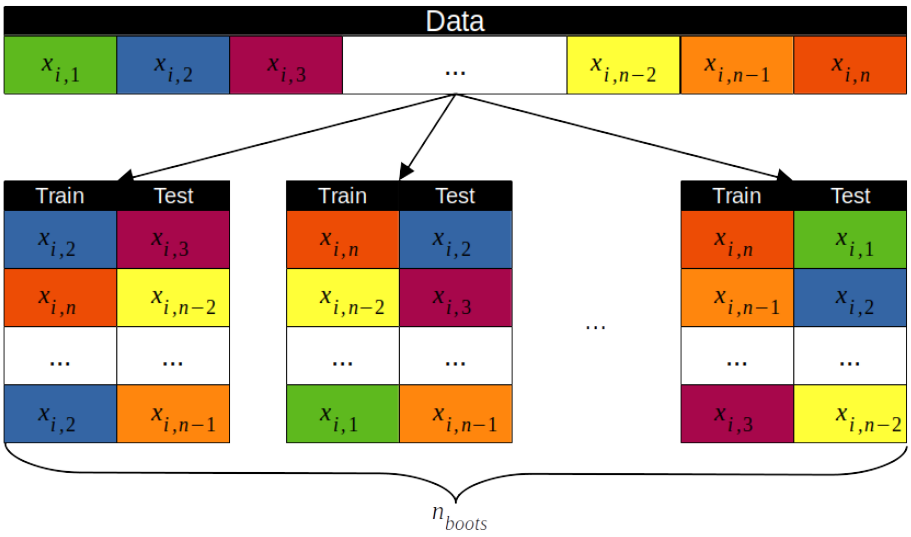


Figure 1: Flowchart of the bootstrap algorithm.

2.1.4 K-fold Cross-Validation

K-fold cross-validation is an ideal method to circumvent the problem that data is often scarce and there is usually not enough data to set aside a validation set. To avoid this problem the data into k parts of the same size [2].
The steps of the algorithm are as follows:

1. The data is divided into a finite number k_{folds} of equal-sized sets called folds.

2. For each fold k_{fi}:

   (a) Take the fold as the testing set while the rest of the folds comprise the training set.

(b) Train parameters for the training set and fit them to the testing set.

(c) Compute statistics on the training set: MSE_i, R_i^2

3. The final statistics of the cross-validation algorithm are the mean of the statistics obtained by considering each fold as the testing set.

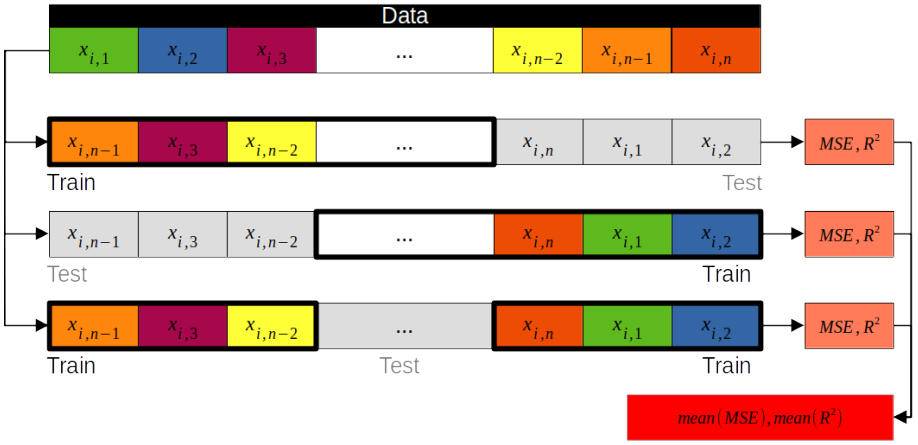


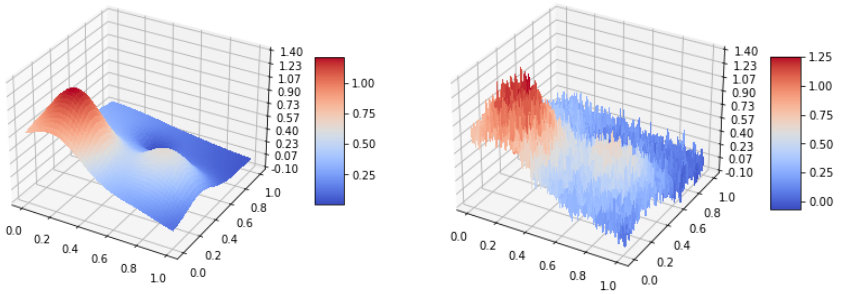
Figure 2: Flowchart of the k-fold cross-validation algorithm with $k_{folds} = 3$ and $n = 9$.

2.2 Data

In this project, we are fitting $2D$ polynomials to the Franke function with gaussian noise. The Franke function is defined as

$$\begin{aligned} f(x_1, x_2) := &\frac{3}{4}\exp\left(-\frac{(9x_1-2)^2}{4} - \frac{(9x_2-2)^2}{4}\right) \\ &+\frac{3}{4}\exp\left(-\frac{(9x_1+1)^2}{49} - \frac{(9x_2+1)}{10}\right) \\ &+\frac{1}{2}\exp\left(-\frac{(9x_1-7)^2}{4} - \frac{(9x_2-3)^2}{4}\right) \\ &-\frac{1}{5}\exp\left(-(9x_1-4)^2 - (9x_2-7)^2\right) \end{aligned} \tag{15}$$

To this Franke function, we add noise sampled from a Gaussian distribution $\mathcal{N}(0, var)$ to get a noisy Franke function: $\tilde{f}(x_1, x_2)$. Figure 3 shows a plot of the Franke function with and without noise.



(a) (b)

Figure 3: Plot of the Franke function with noise (b) and without noise (a). The variance of the added noise is 0.1.

2.3 Results

This jupyter notebook consists of examples to apply different regression techniques. It also consists of widgets where the precalculated statistical performance of different regression methods can be viewed for over 10^5 parameter combinations. The data for these experiments is included in the repository. At the end of the jupyter notebook, one can find code that will allow for simulating more parameter combinations if needed. This will overwrite the existing data and also would take a significant time to run. For 10^5 parameter combinations, it took us about 2 hours to obtain the data.

2.3.1 Terminology of symbols and variable names

Here, we list the names of the variables that are the options in the widgets of the jupyter notebook. The corresponding symbols if applicable are also shown. These symbols relate the variable names to their corresponding mathematical notation used in this report. The general template is the following

Symbol: variable name $\rightarrow$ explanation

The list of widget variables are:

- p : polynomial order $\rightarrow$ the order of 2D polynomial used for fitting
- var : noise_var $\rightarrow$ the variance of the zero mean Gaussian noise added to the Franke data
- r : ratio $\rightarrow$ testing ratio. For example, if the ratio is 0.1 then the test set comprises 10 percent of the dataset.
- n : num $\rightarrow$ the number of points taken along each direction of the input vector. If num = 20, then the total number of data points is 20*20 = 400

- stat → Displays the chosen statistic
  - MSE_{train} : training MSE
  - MSE_{test} : testing MSE
  - R^2_{train} : training R2 score
  - R^2_{test} : testing R2 score
  - $bias_{test}$: bias in test data
  - var_{test} : variance in test prediction
- method → to choose the regression type
  - direct solution and stochastic gradient descent solution available
    - OLS
    - OLS with bootstrap resampling
    - OLS with cross-validation resampling
    - Ridge
    - Ridge with bootstrap resampling
    - Ridge with cross-validation resampling
  - direct Solution only (using scikit learn)
    - Lasso
    - Lasso with bootstrap resampling
    - Lasso with cross-validation resampling
- n_b : n_boot → number of times bootstrap sampling is performed in the bootstrap method. Changing this for methods that don't involve bootstrap sampling will not have any effect
- k_f : k_fold → number of folds in the cross-validation method. Changing this for methods that don't involve cross-validation will not have any effect
- λ_r : ridge_lambda → regularisation parameter for ridge regression
- λ_l : lasso_lambda → regularisation parameter for lasso regression
- η : learn_rates → parameter that controls the descent jump size in gradient descent methods
- n_e : epoch → parameter that controls the number of times the algorithm works through the entire dataset in gradient descent methods
- n_{batch} : number of batches → parameter that controls the mini-batch size in gradient descent methods

Here, we present selected results of linear regression that we find particularly interesting.

2.3.2 Ordinary Least Squares (OLS)

Foremost, we look at the MSE_{train} of OLS (see figure 4a). We see that the MSE_{train} reduces as the model complexity or polynomial order(p) increases. In other words, with increasing complexity, our model becomes more flexible. As the input data is an exponential function that can be expressed as a series of polynomials, increasing the terms in the fitted polynomial should decrease the MSE_{train}. We also see that for large values of test ratio, i.e. $r = 0.4$, and low number of data points, $n = 100$, which amounts to about 60 data points in the training set, the MSE_{train} is significantly lower for high p.

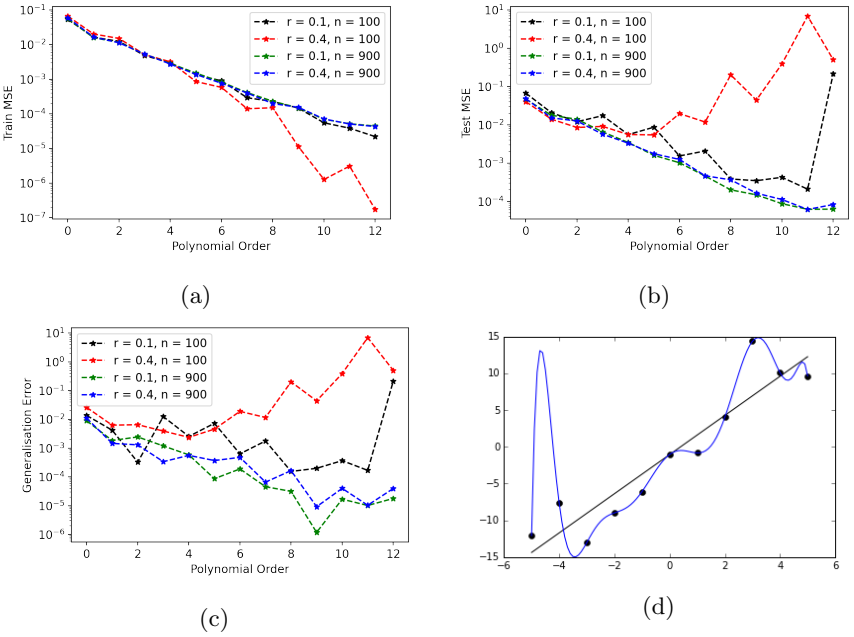
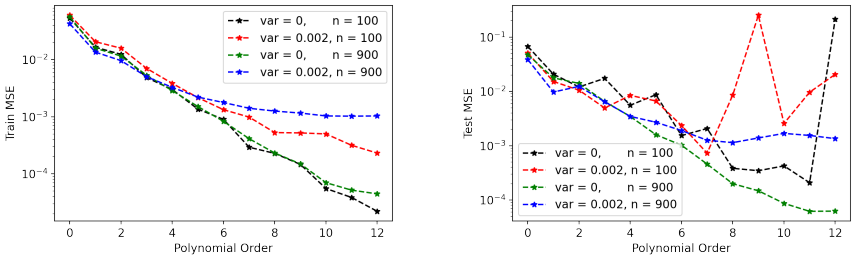


Figure 4: OLS for noiseless Franke data: (a) MSE_{train}, (b) MSE_{test} and (c) Generalization error plotted as functions of model complexity. (d) Schematic describing overfitting of data due to higher model complexity (Source: Ghiles, Wikipedia).

Now, if we shift attention to the MSE_{test}, we see that for r=0.4, n=100, the generalization is poor (see figure 4b). This is the result of overfitting. This also happens for r=0.1, n=100 albeit not as strongly. This reveals that even for low values of MSE_{train}, the model can perform poorly. We recognize that if the number of datapoints is higher in the training set, then there is no overfitting. This is also evident by looking at the generalization error(see figure 4c). The model 'sees' more variety of data during the training phase which prevents it from overfitting. Provided we don't overfit, the MSE_{test} reduces as p increases. It should be noted that irrespective of the number of data points used for train-

ing, there exist p's where we will experience overfitting. The figure 4d shows a 1D polynomial with order equal to the number of points which will perform much worse than a linear polynomial for unseen data. If n_{train} is the number of training data points then to prevent overfitting in many situations, we need $n_{train} >> p$.

Next, we see that the introduction of noise into the dataset increases MSE_{train} (see figure 5a). We also see that introducing noise affects the MSE_{test} of a dataset with a larger number of points significantly when model complexity is high (see figure 5b). MSE_{test} decreases initially with model complexity but later plateaus. This portrays that the higher degree of freedom, introduced by additional polynomial terms, doesn't sufficiently capture the noise. When the model complexity is low, the prediction is unaffected by the presence of noise.



(a) (b)

Figure 5: OLS for $r = 0.1$ and $n = 900$: (a) MSE_{train} and (b) MSE_{test} plotted as functions of model complexity

If we now look at the bias and variance, they are both roughly constant as the model complexity increases (see figure 6). This might be a discrepancy in our calculation and it prevents us from understanding the bias-variance tradeoff. This behavior is observed even when other parameters are changed. As variance measures, the spread of our prediction, and the plots show that our model always has a finite spread. As bias is a measure of the mean deviation of our prediction from the truth, the constant values raise questions on our calculations. We know that parameters like r, var, p and n have an impact on the MSE_{test}, but as the bias seems to be roughly the same then that shows that irrespective of the parameters, our model has approximately the same mean value of the predicted output. Either our model is capturing the mean well almost always or our calculations of bias and variance are incorrect.

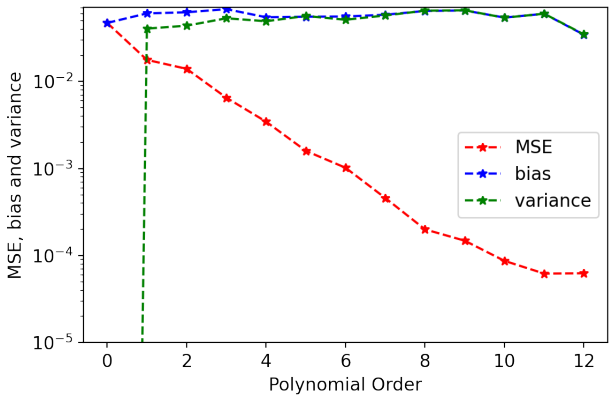
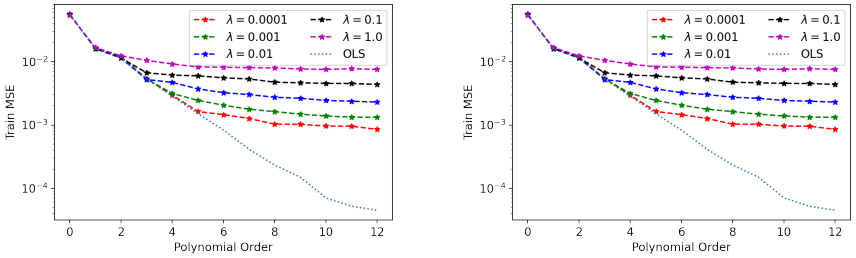


Figure 6: OLS for $r = 0.1$, $var = 0$ and $n = 900$. Bias and variance plotted as functions of model complexity

2.3.3 Ridge and lasso Regression

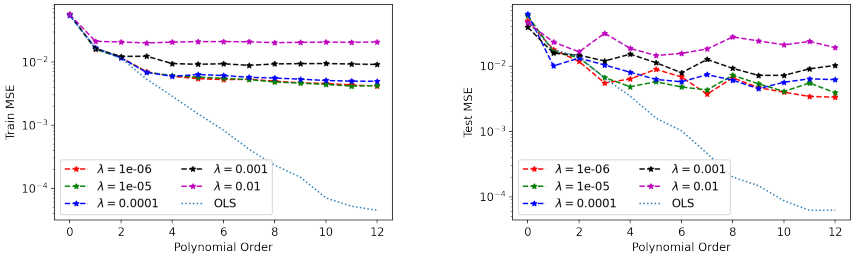
In ridge regression, we use the L_2 norm of the parameter vector $\boldsymbol{\beta}$ as a penalty term. As we are not tuning the size of the elements of the design matrix, the interrelations between the different columns should remain the same. This is the case when the design matrix is orthogonal. As we use polynomials as basis functions, the columns of the design matrix are likely to be independent. They will not be independent if there is a correlation between x_{i1} and x_{i2} which is not true in our dataset. So, the parameters in ridge regression would be rescaled by a factor of $\frac{1}{1+\lambda_r}$ compared to the OLS parameters [5]. However, since that there might be errors in calculating the pseudoinverse, the scaling might not always be by the factor of $\frac{1}{1+\lambda_r}$. So, not all parameters are scaled the same. This would deviate the predictions from that of the OLS. We can see this in the figure 7a and 7b where the regularisation parameter affects both the training and prediction phase. The training and testing error both worsen as λ_r increases. The deviation from the OLS errors kicks in at a lower model complexity for higher values of λ_r. For lower model complexity, the ridge regression performs similarly to OLS. But, as the complexity increases, ridge regression has a poorer prediction. It is unclear why this happens, but the most likely reason might be numeric underflow or overflow.
Meanwhile, in lasso regression, we use the L_1 norm of the parameter vector $\boldsymbol{\beta}$ as a penalty term. Visually, this is equivalent to optimizing the parameters until we hit the edges of a constraint hypercuboid. While, in ridge regression, we hit the surface of a constraint hyperellipsoid. See chapter 3 of [2] for more details. This means that in lasso regression when the optimum parameters are reached, the components of some of the parameter vector $\boldsymbol{\beta}$ are 0. These features were not important enough in affecting the combined optimization of the least square error and the L_1 norm of the parameter vector $\boldsymbol{\beta}$. We see in the figure 8a that



(a) (b)

Figure 7: Ridge regression for $r = 0.1$, $var = 0$ and $n = 900$: (a) MSE_{train} and (b) MSE_{test} plotted as functions of model complexity for different regularisation strength

both the training error and testing error worsen as the λ_l increases. But, the performance is much worse than for ridge regression and is several orders of magnitude worse than OLS. As λ_l increases, more components of $\boldsymbol{\beta}$ turn out to be 0.



(a) (b)

Figure 8: lasso regression for $r = 0.1$, $var = 0$ and $n = 900$: (a) MSE_{train} and (b) MSE_{test} plotted as functions of model complexity for different regularisation strength

The figure 9a and 9b show the relative performance in training and testing for OLS, ridge, and lasso regression. Here, we show the best-case scenario for Ridge and lasso among the parameters screened by us. OLS outperforms ridge regression by an order of magnitude while it is 2 orders of magnitude better than lasso. This is explained by the fact that in OLS, the optimization involves lowering the squared error but in the other methods, there is an additional cost incurred for the size of components of parameter vectors. The figure 9c shows the R^2_{test} error which is a measure of the variance captured by our model in relation to the actual variance. We see that as the model complexity increases, the R^2_{test} tends to reach 1 for both OLS and ridge regression. While in lasso

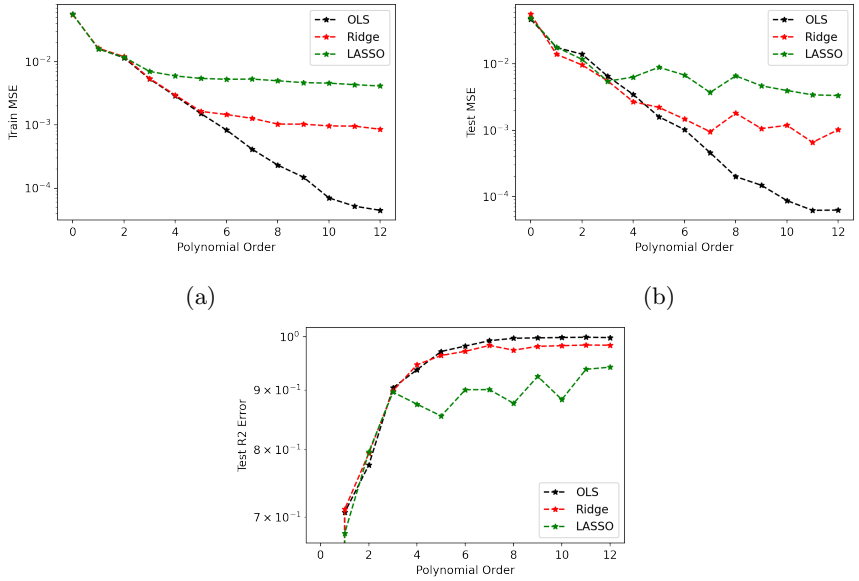


Figure 9: OLS, Ridge($\lambda_r = 1 \times 10^{-4}$) and lasso($\lambda_l = 1 \times 10^{-6}$) regression for $r = 0.1$, $var = 0$ and $n = 900$: (a) MSE_{train}, (b) MSE_{test} and (c) R^2_{test} plotted as functions of model complexity

regression, the variance capture is much poorer. Certainly, the results here indicate that OLS performs much better than Ridge and lasso. Why would these two methods remain relevant? This is because there can be situations where the size of components of $\boldsymbol{\beta}$ can cause a numerical overflow. Additionally, shrinkage methods like Ridge and lasso reduce the prediction variability in many situations [2].

2.3.4 Regression with Bootstrap and Cross-validation

The figure 10 shows the comparison between OLS, Ridge, and lasso methods with bootstrap and cross-validation resampling. Resampling is primarily performed to reduce the uncertainty in statistics like the mean squared error. Often, statistics of a single fit might not be the true indicator of the performance of the model due to the inherent stochasticity while splitting data into train and test sets, and in the noise. However, we see that in the case of OLS, the introduction of resampling has almost no effect. This can be attributed to the fact that due to $n = 900$, the model 'sees' a variety of data during the training phase. If the number of data points is reduced for example to $n = 100$ (see figure 11), then the effect of resampling is more prominent. We notice that for $n = 100$, the training is equally well for OLS and OLS with cross-validation resampling.

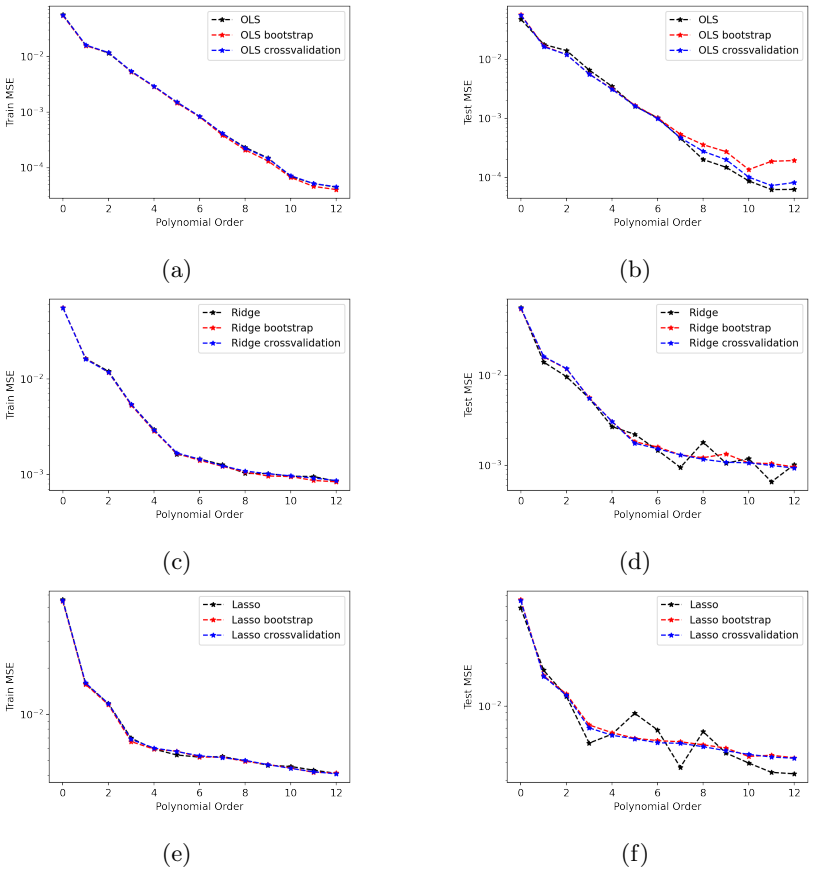


Figure 10: For $r = 0.1$, $var = 0$ and $n = 900$: (a), (c), (e) MSE_{train} and (b), (d), (f) MSE_{test} plotted as functions of model complexity

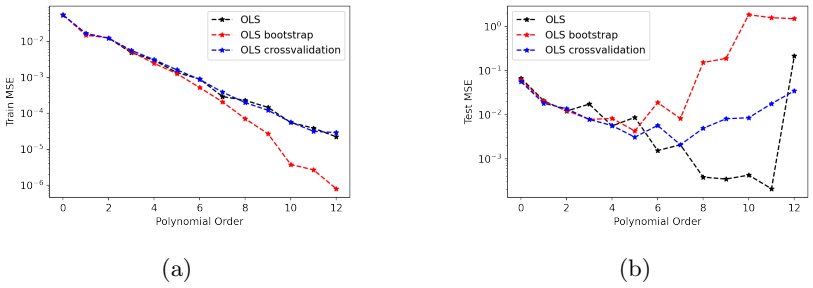


Figure 11: For $r = 0.1$, $var = 0$ and $n = 100$: (a) MSE_{train} and (b) MSE_{test} plotted as functions of model complexity

But, MSE_{train} is lower for bootstrap for high values of p. This is an indicator of overfitting where, in bootstrap resampling, some of the data is 'seen' again due to sampling with replacement. This reduces the effective training set size and makes the model more prone to overfitting. This is also reflected in poor MSE_{test} for OLS with bootstrap resampling. We also think that OLS with cross-validation resampling provides more reliable statistics than OLS without resampling as statistics of OLS with cross-validation can be thought of as a numerical average of statistics of multiple instances of OLS without resampling. The introduction of resampling in Ridge and lasso has no effect on training (figures 10c and 10e) but they do smoothen out the irregularities of the MSE_{test} (figures 10d and 10f).

2.3.5 OLS with Stochastic Gradient Descent

For stochastic gradient descent (SGD) we have three additional parameters: the epoch e, learning rate η, and the number of mini-batches n_{batch}. See section 3.1.1 for a detailed description of the SGD algorithm. The resource [4] showed that using small batch sizes achieves the best training stability and test performance across a wide range of experiments. The best results were obtained for batch sizes equal to or smaller than 32. Therefore, we focus in this discussion on batch sizes 1, 2, 5, 10, 32. For a batch size of 1, we have a classical stochastic gradient descent. Otherwise, the algorithm is called mini-batch gradient descent. Usually, the number of epochs is large to sufficiently minimize the error by the learning algorithm. Therefore, we compare the output with epochs of size 10, 100, 500, 1000. Lastly, the learning rate is a small positive number and traditionally between 0 and 1. We analyze the output with the following learning rates 0.001, 0.01, 0.1, 1.0.

While the MSE for the training dataset hardly varies for different learn rates, number of epochs, or batch sizes the MSE for the test dataset improves slightly if the parameters are chosen correctly and the order of the fitted polynomial is high (see figure 12). The best results for MSE_{test} were obtained for $n_{batch} = 10, n_e = 32$ and $\eta = 0.1$. But even for the size of the mini-batches, which displays the "biggest" variance in the MSE_{test}, the difference is only between roughly 10^{-3} and 10^{-4} (see figure 12d). To conclude, the choice of the SGD parameters, which we tested, does not seem to play an important part for the performance of the fitting in this case.

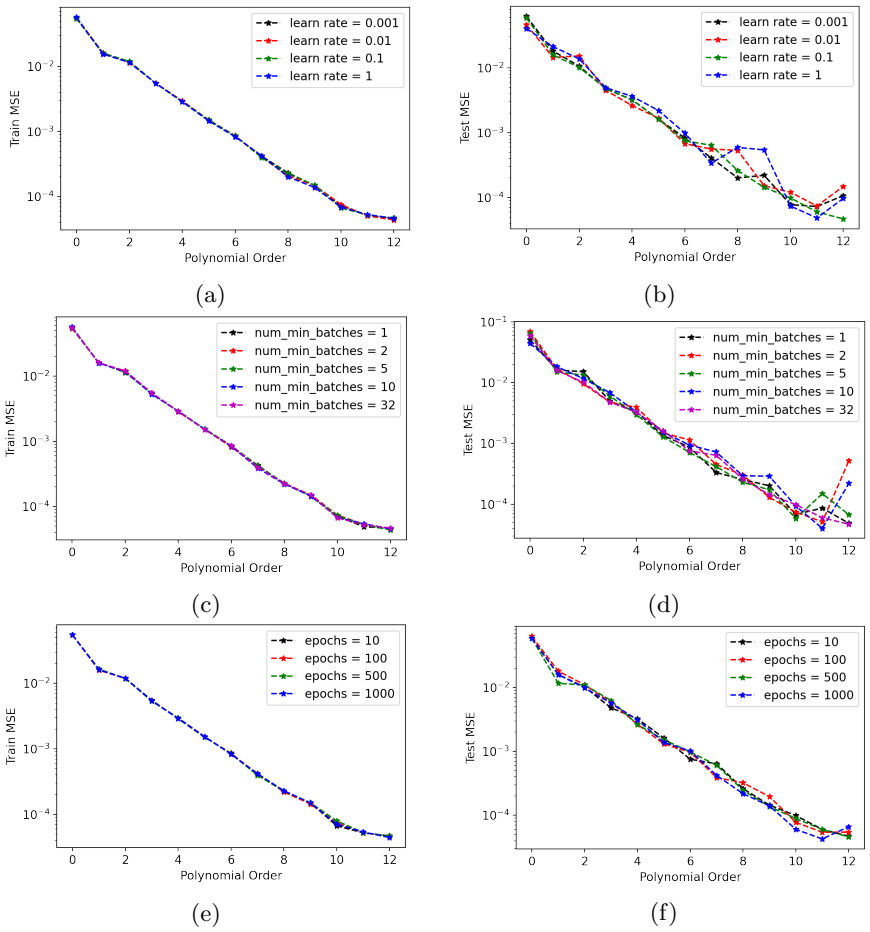
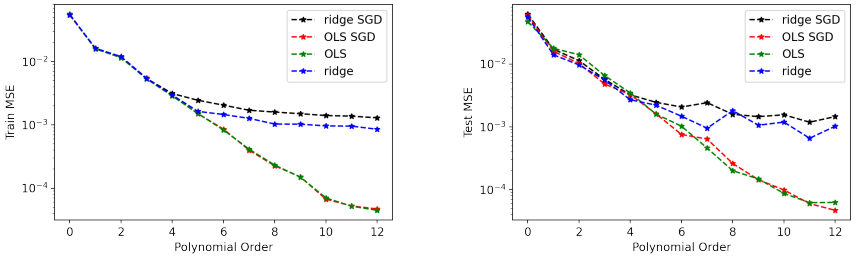


Figure 12: For $r = 0.1$, $var = 0$, $n = 900$: MSE_{train} (left column) and MSE_{test} (right column) plotted as functions of model complexity. a), b): $n_{batch} = 10, n_e = 32$, c), d): $\eta = 0.1, n_e = 10$, e), f): $\eta = 0.1, n_{batch} = 32$

Similar results were also obtained for optimizing the cost function for ridge regression. When we compare the test and train MSE for ridge and OLS regression with and without SGD (see figure 13) we clearly see that for OLS there is nearly no difference if the mini-batch stochastic gradient descent is applied or not. In contrary, when we look at ridge regression both the train and test MSE are better without applying the mini-batch stochastic gradient descent. For this simulation we used: $n_{batch} = 10, n_e = 32, \eta = 0.1$ and $\lambda_{ridge} = 0.0001$.



(a) (b)

Figure 13: For $r = 0.1$, $var = 0$, $n = 900$: MSE_{train} and MSE_{test} plotted as functions of model complexity.

3 Classification

3.1 Method

The goal of classification is to assign an input vector $\mathbf{x}$ to one of K classes $\mathcal{C}_k$ where $k = 0, 1, ..., K$. Linear regression can be extended to binary classification by two changes [6]. First is the change of the Gaussian distribution for y with a Bernoulli distribution, for the case of binary response, $y_i \in \{0, 1\}$.

$$p(y \mid \mathbf{x}, \boldsymbol{\beta}) = Ber(y \mid p(y = 1 \mid \boldsymbol{x}, \boldsymbol{\beta}))) \tag{16}$$

$$= p^y \left(y = 1 \mid \mathbf{x}, \boldsymbol{\beta}\right) \left(1 - p(y = 1 \mid \mathbf{x}, \boldsymbol{\beta})\right)^{1-y} \tag{17}$$

Second is to pass the linear combination of the inputs and parameters to a function that outputs in the range $[0, 1]$. This is achieved by using a logit function also known as sigmoid function, $p(y = 1 \mid \mathbf{x}, \boldsymbol{\beta}) = sigm(\boldsymbol{\beta}^T \mathbf{x})$.

We define sigmoid function as,

$$sigm(z) = \frac{1}{1 + e^{-z}} \tag{18}$$

Putting these two steps together we get,

$$p(y \mid \mathbf{x}, \boldsymbol{\beta}) = Ber(y|sigm(\boldsymbol{\beta}^T \mathbf{x}))) \tag{19}$$

For a multivariate set of data $\mathcal{D} = \{y_i, \mathbf{x_i}\}$ with y_i being a binary response $y_i \in \{0, 1\}$ and $\mathbf{x_i}$ is a vector of size $1 \times n$ where n is the number of features, the likelihood for all possible outcome is,

$$P(\mathcal{D} \mid \boldsymbol{\beta}) = \Pi_i p(y_i \mid \mathbf{x}_i, \boldsymbol{\beta}) \tag{20}$$

From maximum likelihood estimation principle, the most probable observed data can be obtained from the maximum log-likelihood function where we maximizes with respect to $\boldsymbol{\beta}$. The log-likelihood for equation (20) is given by,

$$\mathrm{LL}(\boldsymbol{\beta}) = \sum_{i=1}^{N} \left[y_i \log p(y = 1 \mid \mathbf{x}, \boldsymbol{\beta}) + (1 - y_i) \log \left(1 - p(y = 1 \mid \mathbf{x}, \boldsymbol{\beta})\right) \right] \tag{21}$$

For logistic regression, the cost function is just the negative log-likelihood. We want a cost function to be minimized hence a negative sign on the log-likelihood.

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{N} \left[y_i \log p(y = 1 \mid \mathbf{x}, \boldsymbol{\beta}) + (1 - y_i) \log \left(1 - p(y = 1 \mid \mathbf{x}, \boldsymbol{\beta})\right) \right] \tag{22}$$

This is also known as the cross-entropy error function. Unlike linear regression, the maximum likelihood estimate (MLE) for the parameters is no longer in closed form, so we need an optimization algorithm to compute it. One famous simple algorithm for unconstrained optimization is gradient descent, also known as steepest descent. It is given as follows,

$$g_k = -\mathbf{X}^T(\mathbf{y} - \mu) \tag{23}$$

$$\beta_{k+1} = \beta_k - \eta_k g_k \tag{24}$$

where η_k is the learning rate, and g_k is the gradient of the loss function with respect to β. Several hyperparameters can be utilized in controlling the training. We use the following tuning parameters in our code: learning rate, number of epochs, and number of batches. Learning rate is just the step size on each iteration as it approaches the minimum of the cost function. The number of batches is the number of samples processed before updating the learnable parameter. And the number of the epoch is a complete cycle over the training data. After completing one epoch, the optimal parameter $\boldsymbol{\beta}$ can be obtained. This can be used to calculate a continuous response $t = \boldsymbol{X}^T\boldsymbol{\beta}$. If we set a threshold value for t we can create a decision rule. In our case, we use the following decision rule:

$$\hat{y} = \begin{cases} 1 & \text{if } t \geq 0 \\ 0 & \text{if } t < 0 \end{cases} \tag{25}$$

There are limitations of gradient descent. It can get stuck in local minima, it is sensitive to the initial condition, and for a large dataset, it is computationally expensive [5]. It is alleviated through stochasticity, for instance, by only taking the gradient of a subset of data called mini-batches.

3.1.1 Stochastic Gradient descent with mini-batches

If there are n samples in total, and the mini-batch size set is M, there are n/M possible mini-batches B_k where $k \in (1, 2, ..n/M)$. In this case, the gradient

descent is approximated in each cycle over the mini-batches using a single mini-batch. The approximated gradient descent over a single batch , $c_i(\mathbf{x_i}, \beta)$, is the gradient used to update the parameter. We denote the approximated gradient by $\nabla_\beta C^{MB}(\beta)$ written as,

$$\nabla_\beta C^{MB}(\beta) = \sum_{i \in B_k} \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{26}$$

The SGD with mini-batches equation is given by,

$$g_k = \nabla_\beta C^{MB}(\beta) \tag{27}$$

$$\beta_{k+1} = \beta_k - \eta_k g_k \tag{28}$$

3.1.2 Support Vector Machine

Fundamentally support vector machine (SVM) is a two-class classifier. Since the problem at hand consists of two classes, benign and malignant, the Wisconsin breast cancer data is a good candidate to test SVM. SVM classifier determines a decision boundary by choosing parameters that maximize the margin. This constraint optimization problem is solved using Lagrange multipliers that follow Karush-Kuhn-Tucker conditions [1]. A dual representation of the Lagrangian form is expressed in terms of the kernel and obtained by taking the gradient of the Lagrangian form with respect to the parameters and setting this gradient to zero. The new data points are classified using the trained model by evaluating the signs of the output. To test SVM, we use scikit's built-in functionality.

3.2 Data

For logistic regression and Support Vector Machines, we use the Wisconsin Breast Cancer Dataset [1]. This dataset contains measurements for breast cancer cases. There are two types of cancer in the dataset benign and malignant. An overview of the dataset if given in the jupyter notebook logistic_regression_analysis which can be found in the GitHub repository corresponding to this report. Based on this dataset we want to find a model which predicts the diagnosis, i.e. either benign or malignant. For the design matrix, we drop the column id and diagnosis from the data. The id is not important for making predictions and the diagnosis is what we want to predict.

3.3 Results

Using scikit's grid search functionality it can be shown that both a rbf and a linear kernel with $C = 1$ are a good choice for the support vector machine. Therefore, we will use a linear kernel and set $C = 1$.

[1] https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

Classification Method	TP (train)	TN (train)	TP (test)	TN (test)
logistic sgd	0.94	0.99	0.90	0.97
logistic sgd cv	0.95	0.98	0.93	0.99
logistic scikit	0.94	0.99	0.90	1
logistic scikit cv	0.92	0.99	0.90	1
svm	0.96	1	0.94	0.97
svm cv	0.96	1	0.93	1

Table 1: Performance of each classification method in terms of the true positive (TP) and true negative (TN) percentage for training and test data. We use the following hyperparameters: $\lambda = 0.001$, learn rate=0.001, batchsize=1, epoch=1000, test ratio=0.1, k-fold=5.

From table 1 it shows that SVM is the best performing classification algorithm. SVM is known to be capable of separating overlapping class distribution [1]. Logistic regression, on the other hand, requires too many hyperparameters that finding the best set of hyperparameters is another task aside from optimizing the learnable parameters. We can say that all methods perform well in classifying benign classes and perform average on classifying cancer, i.e. malignant tumors.

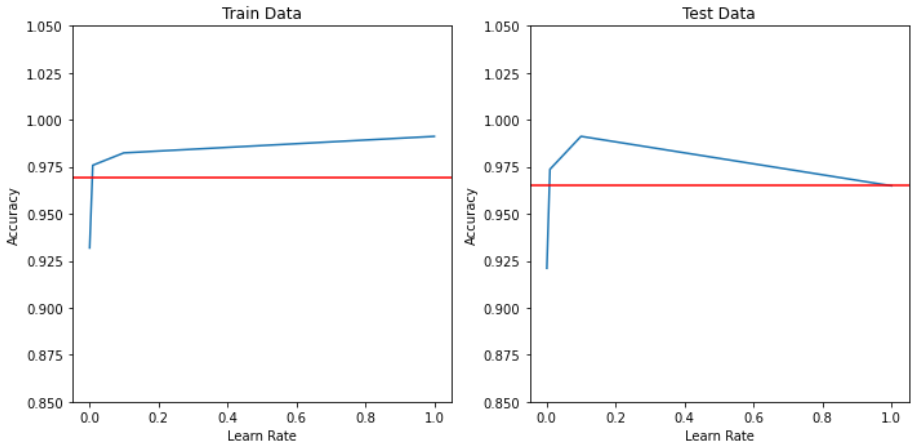


Figure 14: Accuracy variation as a function of the learning rate for the case with the following hyperparameter: $\lambda = 0.001$, batch size=32, epoch=100, test ratio=0.2

To study the effect of tuning parameters in the performance of the model we examine the following cases: effect of the number of epoch and effect of the number of batches.
In figure 14 we see that the model starts to overfit as the learning rate increases, it is evident in the result of the testing accuracy. It is a result of a small number

of iteration. One can improve the model by increasing the number of iterations, as seen in 15, which starts performing well as a function of the learning rate.

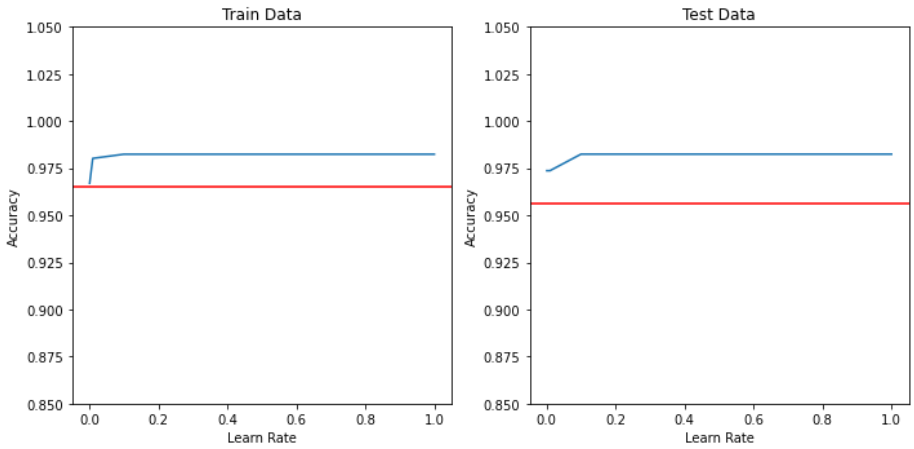


Figure 15: Accuracy variation as a function of the learning rate for the case with the following hyperparameter: $\lambda = 0.001$, batch size=32, epoch=1000, test ratio=0.2

We explore also the effect of mini-batches size. The number of mini-batches determines the speed of the gradient search and thus helps minimize the chances of being stuck in a local minimum. This can be seen in figure 16-17 in both mini-batch cases. At standard stochastic gradient descent case (batch size=1) the accuracy increase is slow, as shown in the test accuracy of figure 16, compared that to the second case where the mini-batch size is 32 shown in the test accuracy of figure 17, the change of accuracy improves drastically. Basically, the batch size affects how quickly the model optimizes the learnable parameters.

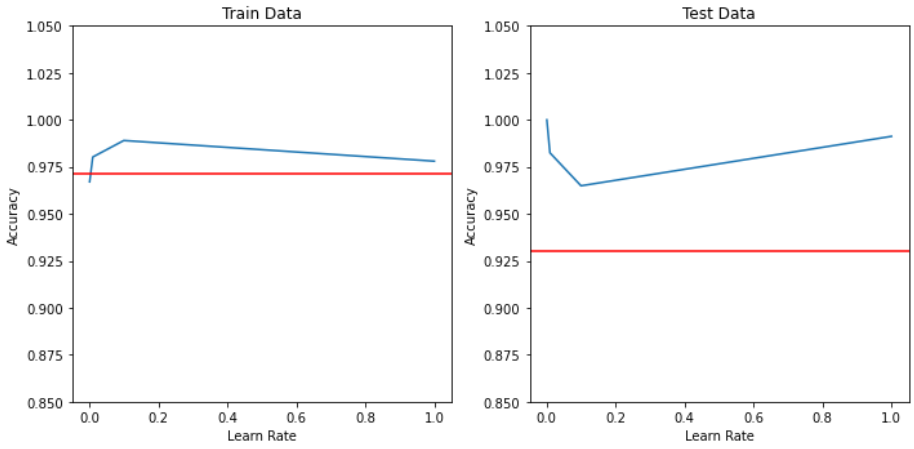


Figure 16: Accuracy variation as a function of the learning rate for the case with the following hyperparameter: $\lambda = 0.001$, batch size=1, epoch=1000, test ratio=0.2

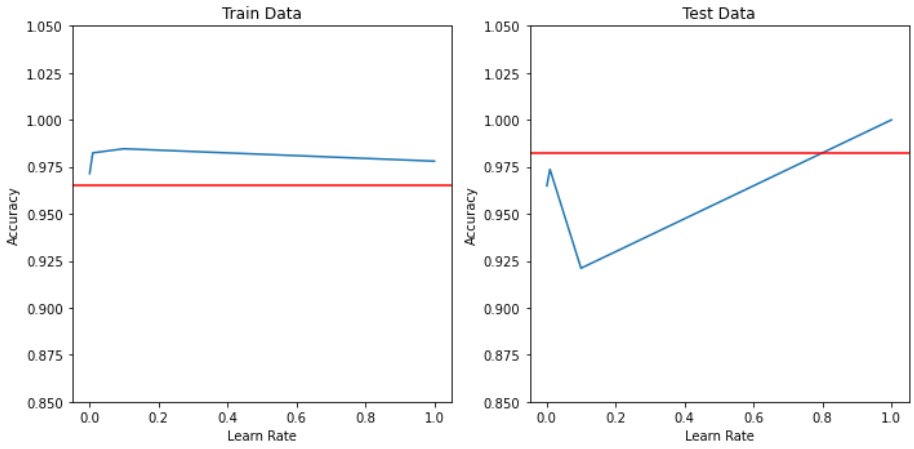


Figure 17: Accuracy variation as a function of the learning rate for the case with the following hyperparameter: $\lambda = 0.001$, batch size=32, epoch=1000, test ratio=0.2

4 Summary

We trained various linear regression models like OLS, ridge, and lasso to fit polynomials to the Franke function with noise. Thereafter, we used the model to predict the output for unseen test data. Various statistics like MSE, R^2 Error, and generalization error were used to probe the model performance. We

also applied resampling techniques to reduce the uncertainty in the statistics. Along with finding optimal parameters directly using closed-form solutions, we also utilized stochastic gradient descent to find optimal parameters. We discovered that OLS is better at predicting than lasso and ridge for the sample data considered. When the lasso and ridge models tend to more OLS like behavior i.e. λ_l and λ_r tend to 0, the prediction improves. OLS performed several orders of magnitude better than lasso regression and a few orders of magnitude better than ridge in terms of the prediction error. We also experienced the problem of overfitting when the training data was less numerous and particularly during bootstrap resampling where the effective data that the model could 'see' during the training phase was reduced due to sampling with replacement.
In addition, we used logistic regression and Support Vector Machines to fit a model to the Wisconsin Breast Cancer data. Overall, we showed that SVM with a linear kernel is the best performing classification algorithm for this setup. Nonetheless, all tested methods performed well when classifying benign classes. While the performance could be improved for malignant ones.

5 Acknowledgment

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 Skłodowska-Curie Action (MSCA) - Co-funding of Regional, National and International Programmes (COFUND).

References

[1] Christopher M Bishop. Pattern recognition. *Machine learning*, 128(9), 2006.

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

[3] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., 2 edition, 2019.

[4] Dominic Masters and Carlo Luschi. Revisiting Small Batch Training for Deep Neural Networks. *arXiv:1804.07612 [cs, stat]*, April 2018. arXiv: 1804.07612.

[5] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre GR Day, Clint Richardson, Charles K Fisher, and David J Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics reports*, 810:1–124, 2019.

[6] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press, Cambridge, MA, 2012.

[7] Robert H. Shumway and David S. Stoffer. *Time Series Analysis and Its Applications*. Springer International Publishing, 2017.