Regression analysis on Runge function: gradient descent and resampling techniques

FYS-STK3155/4155 - Project 1

Viktor Bilstad, Frederik Callin Østern, Heine Elias Husdal
Department of Physics, University of Oslo
(Dated: October 6, 2025)

Abstract: We investigate the performance of different regression techniques and central machine learning methods on polynomial fits to the Runge function. We employ the Ordinary Least Squares (OLS), Ridge and Lasso techniques and compare their accuracy using the mean squared error (MSE) and R^2 statistic. Furthermore, we use bias-variance tradeoff and k-fold cross validation to analyze model complexity and overfitting, and gradient descent in order to analyze the training of the models. We find that OLS regression gives more accurate predictions compared to Ridge and Lasso, with Lasso sometimes even giving R^2 scores larger than 1. However, with only $k = 10$ folds, our cross validation results suggest that both Ridge and Lasso greatly reduce overfitting, and that Lasso gives the biggest reduction in the average MSE. We also find that the range of optimal polynomial degrees with a minimum bias and variance depends significantly on the number of datapoints and the splitting into training and testing data. Our gradient descent results suggest that the stochastic Adagrad algorithm performs well, but this conclusion probably depends on the dataset used. Our methods can be used on more complex data, and they are an essential stepping stone to tackle more advanced designs, such as neural networks.

I. INTRODUCTION

Regression analysis is a central supervised learning technique in machine learning and statistics, which enables us to make a qualified numerical prediction based on a set of input features. In general, such machine learning predictions can be useful in a wide range of different applications, such as bioinformatics, weather forecasting and marketing [1]. Some of the central regression techniques are Ordinary Least Squares (OLS), Ridge and Lasso. The latter two are extensions of the former that add some penalty terms to the cost function, which reduces the effect of certain features.

One may face some difficulties when one is performing a regression analysis. This includes the issue of overfitting, which signifies the need to optimize the model complexity in order to obtain accurate predictions on unseen data. Moreover, for complex models, one may need to use numerical techniques to train it. In machine learning, the gradient descent (GD) methods are a particularly versatile class of techniques that can be used to tune the model parameters. This includes the Adam technique, which has experienced a particularly high popularity in the recent decade [2].

In this work, we analyze the performance of the OLS, Ridge and Lasso regression techniques on a simple dataset taken from the one dimensional Runge function. The goal was to get a deeper intuition of central methods in the field, without the hurdle of having to deal with complex data. We evaluated model performance for different polynomial function fits and regression techniques by using the Mean Squared Error (MSE) and R^2 metrics, comparing both analytical formulas and numerical results obtained from gradient descent methods. In particular, we tested the non-stochastic and stochastic versions of the ordinary, momentum, Adagrad, RMSProp and Adam gradient descent algorithms. Finally, we introduced resampling techniques, including bias-variance tradeoff and k-fold cross validation, in order to investigate the problem of overfitting. These methods allow us to find the correct balance between model complexity and the accuracy of its predictions.

In section II, we present a detailed review of the theory underlying the methods we use, as well as a description of how we implemented them in Python. We also include a short description of the dataset. In section III, we go through our main results and discuss their implications. Finally, in section IV, we end with a conclusion of the discussion and some ideas for future work.

II. METHODS

A. Regression techniques

Consider a dataset consisting of a vector $\boldsymbol{x}$ with n input data values and a corresponding vector $\boldsymbol{y}$ of output data. Suppose the data can be modelled by the relation

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{\epsilon} \tag{1}$$

for some non-stochastic continuous function $\boldsymbol{f}$. Here, $\boldsymbol{\epsilon}$ is a vector of independent and identically normally distributed error terms with zero mean and some standard deviation $\sigma > 0$. In regression, we use the dataset to fit a model to the output values [3]. For linear regression, in particular, the model approximation $\tilde{\boldsymbol{y}}$ of the output values is given by a linear function of the input,

$$\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\theta} \tag{2}$$

Here, $\boldsymbol{X}$ is the *design matrix* corresponding to the input data [3]. The columns in this matrix is some function

of the input values in $\boldsymbol{x}$. For a polynomial fit (which we focused on in our work), the j-th column in $\boldsymbol{X}$ consists of the j-th powers of the input values in $\boldsymbol{x}$. In other words, $X_{i,j} = {x_i}^j$.

The vector $\boldsymbol{\theta}$ consists of a certain number p of parameters. These parameters must be trained in order to minimize the error between the predicted and actual output values. Different regression techniques quantify the error using different cost functions.

1. OLS regression

In the Ordinary Least Squares (OLS) regression method, the cost function of the model parameters that must be minimized is the usual mean squared error (MSE) given by

$$C_{OLS}(\boldsymbol{\theta}) = \frac{1}{n}(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2 = \frac{1}{n}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^2 \tag{3}$$

with the gradient [3]

$$\frac{\partial C_{OLS}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^T} = -\frac{2}{n}\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) \tag{4}$$

The mean squared error quantifies the (squared) average error between the true and predicted output. In our work, it will therefore be a central metric to quantify the performance of our models.

There is an analytical expression for the optimal parameters where $C(\boldsymbol{\theta})$ has its minimum. It is given by

$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{5}$$

2. Ridge regression

In Ridge regression, one adds an extra l_2 penalty to the cost function, so that it takes the form

$$C_{Ridge}(\theta) = \frac{1}{n}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^2 + \lambda\sum_{i=1}^{p}\theta_i^2 \tag{6}$$

with the gradient

$$\frac{\partial C_{Ridge}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^T} = -\frac{2}{n}\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) + 2\lambda\boldsymbol{\theta} \tag{7}$$

The corresponding analytical solution to the optimal parameters is [3]

$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X} + n\lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{8}$$

There are, in principle, two main advantages with this technique. First, for high-dimensional design matrices, some of its columns may be close to being linearly dependent (which is called collinearity), making the matrix $\boldsymbol{X}^T\boldsymbol{X}$ nearly singular. Thus, the extra term $\lambda\boldsymbol{I}$ ensures that Eq. 8 does not diverge. Moreover, Ridge regression generally suppresses some of the less sensitive degrees of freedom of the model [3] [4], which may improve generalizability.

On the other hand, due to the extra term in the cost function, the Ridge regression technique may not give a smaller MSE on the *testing data* compared to OLS regression (while it is guaranteed to give a bigger MSE on the *training data*). Thus, to find the best regression technique on our dataset, we will compare the mean squared error on the testing data of the three techniques, including a thorough analysis using cross validation and bias-variance tradeoff.

3. Lasso regression

Finally, for Lasso regression, we add an l_1 penalty to the MSE, giving the cost function

$$C_{Lasso}(\theta) = \frac{1}{n}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^2 + \lambda\sum_{i=1}^{p}|\theta_i| \tag{9}$$

The gradient of this cost function is [3]

$$\frac{\partial C_{Lasso}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^T} = -\frac{2}{n}\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) + \lambda\text{sgn}(\boldsymbol{\theta}) \tag{10}$$

where $\text{sgn}(x) = 1$ if $x > 0$, $\text{sgn}(x) = 0$ if $x = 0$ and $\text{sgn}(x) = -1$ if $x < 0$.

The effect of Lasso regression may be to select certain features, setting the other parameters exactly equal to zero [4]. There is no analytical solution to the Lasso problem. However, it can be solved numerically using gradient descent, which we will therefore explain next.

B. Gradient descent

In general, in order to minimize the cost function and hence update the model parameters numerically, the gradient descent (GD) class of methods is particularly useful and popular. The essential idea is to iteratively update the parameters by moving in the direction of steepest local descent in the cost function. The main equations can be derived in a few ways, including by calculating the steepest descent or by using Newton's method [5]. In our work, we implemented first-order algorithms, meaning that we only used first order derivatives (gradients) of the cost function.

For the linear OLS and Ridge regression techniques, we mentioned above that there are analytical formulas for the optimal parameters. However, we still implement

the gradient descent techniques explained below, so that we can compare the convergence, noise, instability etc. of different training algorithms. Moreover, we can then implement and compare with the Lasso regression technique as well.

1. Ordinary GD

In the ordinary gradient descent algorithm, we move a fixed step size (or *learning rate*) in the opposite direction of the gradient of the cost function in each iteration. Thus, if $\boldsymbol{\theta}^{(i)}$ is the vector of parameter values in iteration i, then [3]

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(i)}) \tag{11}$$

where η is the constant learning rate. The idea is straightforward; since the negative gradient is the direction of the first order steepest descent in a multivariable function, we keep moving a fixed step size in this direction to minimize the cost function. After a certain number of iterations, the hope is that the parameters have converged towards the optimal values at the (global) minimum.

If the learning rate is too large, then the parameters may not converge towards the minimum, but will instead overshoot the minimum and grow exponentially. For a cost function with a constant Hessian matrix, the limit is [6]

$$\eta < \frac{2}{\lambda_{\max}} \tag{12}$$

where $\lambda_{\max}$ is the maximum eigenvalue of the Hessian. We will compare this theoretical limit with our numerical results on the OLS and Ridge regression methods. For those, the Hessian matrices are

$$\begin{aligned} \frac{\partial^2 C_{OLS}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta} \boldsymbol{\theta}^T} &= \frac{2}{n} \boldsymbol{X}^T \boldsymbol{X} \\ \frac{\partial^2 C_{Ridge}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta} \boldsymbol{\theta}^T} &= \frac{2}{n} \boldsymbol{X}^T \boldsymbol{X} + 2\lambda \boldsymbol{I} \end{aligned} \tag{13}$$

2. Momentum GD

In the ordinary gradient descent algorithm, the learning rate is constant for each iteration, which may lead to slow convergence. The idea of momentum-based GD is to speed up convergence by having the parameters "accelerate" towards the minimum of the cost function [5]. There is a physical analogy with a ball rolling down a curved surface under the influence of gravity and velocity dependent friction; the cost function then presents the shape of the surface (proportional to the potential energy). The friction term is taken care of by an extra term in the GD algorithm. Formally, we define a p-dimensional velocity vector $\boldsymbol{v}^{(i)}$ for iteration i. We have $v^{(0)} = \mathbf{0}$ and

$$\begin{aligned} \boldsymbol{v}^{(i+1)} &= \alpha \boldsymbol{v}^{(i)} + \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(i)}) \\ \boldsymbol{\theta}^{(i+1)} &= \boldsymbol{\theta}^{(i)} - \boldsymbol{v}^{(i)} \end{aligned} \tag{14}$$

Here, η is the (constant) learning rate and α is a constant between 0 and 1 that is responsible for the "friction" term. We see in the equation above that it contributes an exponential decrease in the velocity $\boldsymbol{v}$.

3. Adagrad GD

Another approach to obtain better convergence, is to use *adaptive learning rates* [3]. The idea is to adapt the learning rate to the gradients of the cost function. In particular, if there have been large gradients in previous iterations (so that the parameters have already changed a lot), one wishes to use smaller learning rates, and vice versa.

One of the simplest adaptive techniques is Adagrad. The idea is to add up the accumulated gradients, and to use it to correspondingly scale down the learning rate in the next iteration [3]. More precisely, we define a p-dimensional vector $\boldsymbol{r}^{(i)}$ in iteration i. We set $\boldsymbol{r}^{(0)} = \mathbf{0}$ and

$$\begin{aligned} \boldsymbol{r}^{(i+1)} &= \boldsymbol{r}^{(i)} + (\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(i)}))^2 \\ \boldsymbol{\theta}^{(i+1)} &= \boldsymbol{\theta}^{(i)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(i+1)}} + \boldsymbol{\epsilon}} \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(i)}) \end{aligned} \tag{15}$$

where $\eta > 0$ is a constant. The "effective" learning rate is given by $\frac{\eta}{\sqrt{\boldsymbol{r}^{(i)}} + \boldsymbol{\epsilon}}$ in ireation i, and hence is scaled down by the accumulated gradient $\boldsymbol{r}^{(i)}$. Note that in the equation above, elementary functions (like square roots, divisions and exponential powers) are understood to be performed on each component in the vectors separately, so that the result is a vector with the resulting values. Moreover, the vector $\boldsymbol{\epsilon}$ of small tolerance values is there to ensure that the fraction in Eq. (10) does not diverge. The components of $\boldsymbol{\epsilon}$ are set equal to 10^{-8} in our work.

4. RMSProp GD

The RMSProp algorithm is a modification of Adagrad, where one includes an exponential decay of the accumulated gradient. This ensures that it deletes memory of the gradients in the far past [5], which makes the effective learning rate adapt to the newest gradients. This keeps the iterative change in parameters large for many more iterations. More precisely, one introduces a "decay rate" ρ, which is a number between 0 and 1. Then, with $\boldsymbol{r}^{(0)} = \mathbf{0}$, one updates according to [3]

$$\begin{aligned} \boldsymbol{r}^{(i+1)} &= \rho \boldsymbol{r}^{(i)} + (1-\rho)(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(i)}))^2 \\ \boldsymbol{\theta}^{(i+1)} &= \boldsymbol{\theta}^{(i)} - \frac{\eta}{\sqrt{\boldsymbol{r}^{(i+1)}} + \boldsymbol{\epsilon}} \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(i)}) \end{aligned} \tag{16}$$

(with the same choice of tolerance as for the Adagrad algorithm).

5. Adam GD

Finally, the Adam adaptive algorithm combines both momentum and the adaptive learning rates of RMSProp [3]. Specifically, momentum is included as an exponential weighting of the gradient [5], and the learning rate is adapted according to the exponentially weighted accumulated sum of squared gradients of the RMSProp algorithm. Moreover, Adam includes a rescaling of the accumulated gradients, in order to ensure there is no bias for the first iterations [3]. More precisely, we introduce vectors $\boldsymbol{m}^{(i)}$ and $\boldsymbol{r}^{(i)}$ with $\boldsymbol{m}^{(0)} = \boldsymbol{0}$ and $\boldsymbol{r}^{(0)} = \boldsymbol{0}$, and update according to

$$\begin{aligned} \boldsymbol{m}^{(i+1)} &= \beta_1 \boldsymbol{m}^{(i)} + (1-\beta_1)\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(i)}) \\ \boldsymbol{r}^{(i+1)} &= \beta_2 \boldsymbol{r}^{(i)} + (1-\beta_2)(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(i)}))^2 \\ \hat{\boldsymbol{m}}^{(i+1)} &= \frac{\boldsymbol{m}^{(i+1)}}{1-\beta_1^{i+1}} \\ \hat{\boldsymbol{r}}^{(i+1)} &= \frac{\boldsymbol{r}^{(i+1)}}{1-\beta_2^{i+1}} \\ \boldsymbol{\theta}^{(i+1)} &= \boldsymbol{\theta}^{(i)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{r}}^{(i+1)}} + \epsilon} \hat{\boldsymbol{m}}^{(i+1)} \end{aligned} \tag{17}$$

6. Stochastic gradient descent

In our work, we also tested the (minibatch) *stochastic* version of all the previously mentioned gradient descent algorithms. The idea is that for each iteration (or *epoch*), we randomly group together all of the testing datapoints into a certain number $n_{batches}$ of minibatches. Then we run through each batch, calculate the gradient of the cost function corresponding to the data in the batch, and then update the parameters according to the algorithms explained above. In our implementation, we picked the batches without replacement. This ensured that all of the data was used in each epoch.

There are several advantages of using stochastic gradient descent. We may expect faster convergence, since we update the parameters more frequently [3]. For more complicated models, it may also be useful that that we save memory storage by using a much smaller sample of the data each time we update the parameters.

Since batches are chosen at random, we may expect the development of the parameters to be noisy. For some models, this may be advantageous to ensure the parameters do not get stuck in a local minimum [3]. However, for our simple convex cost functions, we expect a single global minimum, and we wish that the parameters settle there. Thus, to reduce noise over time, we lowered the learning rate η according to a learning schedule. In [5], it is explained that there are some conditions on the learning schedule to ensure convergence. A simple function satisfying these conditions, which we used in our work, is the following (adapted from [3]):

$$\eta(t) = \frac{t_0}{t + t_1}\eta_0 \tag{18}$$

Here, $\eta(t)$ is the learning rate used in Eqs. (11) and (14)-(17) in time t (defined as $t = i \cdot n_{batches} + j$, where $j = 1, 2, ..., n_{batches}$ is the minibatch number within epoch i), and η_0 is the initial learning rate. In our work, we defined $t_0 = t_1 = 70 \cdot n_{batches}$. This implies that to get the learning rate, you divide the initial rate by a number that increases by 1 for for every 70th epoch. We found that this gave a sufficient rate of convergence.

C. Resampling techniques

1. Bias-variance tradeoff and bootstrapping

When we choose a model complexity (given by the degree of the polynomial fit in our case), there are two main considerations: first, if it is too low, then it may not be able to capture the complexity of the data, leading to a large mean squared error between the prediction and true output for both the training data used to train the model, and the testing data used to verify its predictions. As the model complexity increases, we may expect this error to decrease to a minimum. However, if the model is too complex, then it may *overfit* on the training data, leading to low performance and a high mean squared error on testing data. Thus, the MSE on the testing data rises again for a sufficiently complex model. In Figure 1, we show this behavior by plotting the MSE of training and testing data as a function of the degree of the polynomial fit.

One may use bias-variance tradeoff to analyze this behavior. The *bias* quantifies the expected (mean squared) error between a model's expected predictions and the true output of testing data, whereas the *variance* represents the expected variability of model predictions from different training data.

We can split the expected mean squared error $\mathbb{E}\left[(y-\tilde{y})^2\right]$ between the true and predicted testing data output into a sum of the bias and variance, as follows. Using Eq. (1), we write

$$\mathbb{E}\left[(y-\tilde{y})^2\right] = \mathbb{E}\left[((f(x) - \mathbb{E}(\tilde{y})) + (\mathbb{E}(\tilde{y}) - \tilde{y}) + \varepsilon)^2\right] =$$

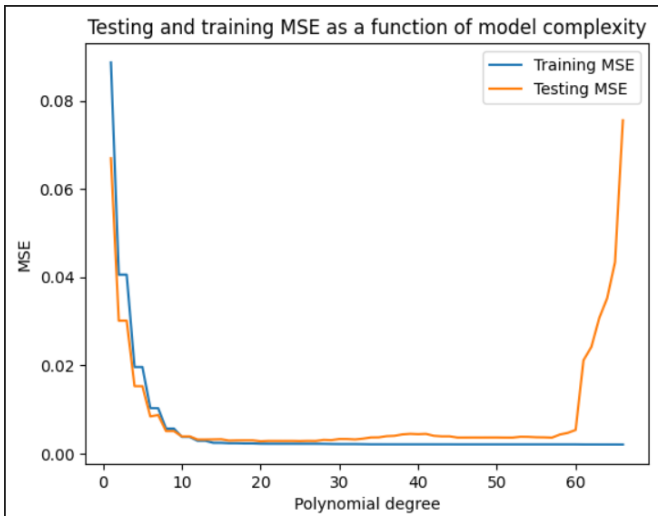

Figure 1. *MSE of training and testing data as a function of polynomial degree (resconstruction of Figure 2.11 in [4]). At first, the MSE of both the training and testing data drop as the polynomial degree increases. However, eventually the testing MSE rises again, indicating that the model overfits during training.*

$$\mathbb{E}\left[(f(x)-\mathbb{E}(\tilde{y}))^2\right]+\mathbb{E}\left[(\mathbb{E}(\tilde{y})-\tilde{y})^2\right]+\mathbb{E}\left[\varepsilon^2\right]$$

$$+2\mathbb{E}\left[(f(x)-\mathbb{E}(\tilde{y}))(\mathbb{E}(\tilde{y})-\tilde{y})\right]+2\mathbb{E}\left[(f(x)-\mathbb{E}(\tilde{y}))\varepsilon\right]$$

$$+2\mathbb{E}\left[(\mathbb{E}(\tilde{y})-\tilde{y})\varepsilon\right]$$

Note that when an expression contains two expectation operators $\mathbb{E}$, the first signifies an expectation/average over the population of *testing data*, whereas the second one indicates an expectation over the population of *training data*.

We know that $\tilde{y}$ and ε are independent random variables. The reason is that ε is the random error of the output of the testing data, and $\tilde{y}$ is the model prediction after it is trained on the training data. Thus, we get

$$\mathbb{E}\left[(\mathbb{E}(\tilde{y})-\tilde{y})\varepsilon\right]=\mathbb{E}\left[(\mathbb{E}(\tilde{y})-\tilde{y})\right]\mathbb{E}\left[\varepsilon\right]=0$$

Furthermore, each value of $f(x)-\mathbb{E}(\tilde{y})$ is a given, non-stochastic value, and hence the expectation value of the product of $f(x)-\mathbb{E}(\tilde{y})$ with either ε or $(\mathbb{E}(\tilde{y})-\tilde{y})$ is also zero (since the expectation value of both ε and $(\mathbb{E}(\tilde{y})-\tilde{y})$ are zero).

Thus, defining the bias [3]

$$\text{Bias}[\tilde{y}]=\mathbb{E}\left[(f(x)-\mathbb{E}(\tilde{y}))^2\right]$$

and the variance

$$\text{var}[\tilde{y}]=\mathbb{E}\left[(\tilde{y}-\mathbb{E}\left[\tilde{y}\right])^2\right]$$

and using that the variance of ε is σ^2, we finally get

$$\mathbb{E}\left[(y-\tilde{y})^2\right]=\text{Bias}[\tilde{y}]+\text{var}[\tilde{y}]+\sigma^2 \tag{19}$$

Note that when we compute the bias numerically in this work, we use the known value of y instead of $f(x)$. Formally, if we replace $f(x)$ with y in the definition of the bias, then the σ^2 term will not be present in Eq. (19).

For low polynomial degrees, we expect both the variance and bias to drop as the degree of the model increases. However, when overfitting occurs, then the variance tends to rise, signifying a large average error between the expected model predictions and the predictions of a specific overfitted model. Since we wish to minimize the MSE of a model, we see from Eq. (19) that we should minimize both the bias and the variance.

In order to estimate the expectation value over testing data, we compute the mean over many pairs of x and y values from the dataset. In order to estimate the expectation over training data, however, we use the *bootstrapping* technique. The idea is to effectively create many training samples from a single collection of training data, and to estimate expectation values by taking the mean over the training samples. More precisely, to create one independent bootstrapping sample from a collection of k training data pairs, we pick k elements from the collection at random, with replacement. This can be repeated in order to get any desired number of samples B.

2. *k-fold cross-validation*

Another method to resample data and test for overfitting is cross-validation. First, we split the dataset into k equally sized *folds*. Then, we designate one fold as the test set in the current iteration and train the model on the data in the remaining $k-1$ folds. After training, we evaluate the model's performance (such as the mean squared error) on the test fold. This process is repeated for k iterations, such that each fold is used as the test set exactly once. The final performance can then be computed as the average over the iterations.

A notable variant of this method, called *leave-one-out cross-validation*, sets k equal to the number of samples in the dataset [3]. This effectively tests the model on every individual datapoint. That may be computationally expensive for large datasets. In particular, since the model must be trained k times, cross-validation increases computational cost by a factor of roughly k. In practice, k is typically set between 5 and 20 instead in order to get a balance between computational cost and accuracy.

Compared to bootstrapping, where new training sets are generated by sampling with replacement, cross-

validation usually provides a less biased estimate of prediction error. The reason is that each data point is guaranteed to be in both training and test sets across different folds.

D. Model performance metrics

In order to compute the performance of model predictions, we use two main metrics. The first is the mean squared error or MSE (which we briefly introduced earlier) between the predictions and actual output,

$$\text{MSE}(\boldsymbol{y}, \tilde{\boldsymbol{y}}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 \tag{20}$$

for predicted model outputs $\tilde{y}_i$ $(i = 1, 2, ..., n)$ and actual outputs y_i. The other important metric is the R^2 score,

$$R^2(\boldsymbol{y}, \tilde{\boldsymbol{y}}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{21}$$

where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ is the average of the true outputs. The R^2 score measures the portion of the variability in the testing data that can be explained by the model. Unlike the MSE, it is not affected by a scaling of the data outputs, and hence its values are more meaningful. We also found it useful to consider $1 - R^2$ in our work, which will be the portion of the variability not explained by the model. It can also be considered a normalized version of the MSE.

E. Our implementation

We implemented the methods on datasets taken from the one-dimensional Runge function, with a normally-distributed noise. See Figure 2. More precisely, the ouput values were given by

$$y = f(x) + \epsilon$$

with the Runge function

$$f(x) = \frac{1}{1 + 25x^2}$$

and noise ϵ distributed according to the standard normal distribution, with an amplitude of 0.05. The input x-values were taken from the interval $x \in [-1, 1]$ with a fixed step size between consecutive values.

In order to create training and testing data from a dataset, we used the `train_test_split` function from Python's scikit-learn [7]. The testing data size was set to

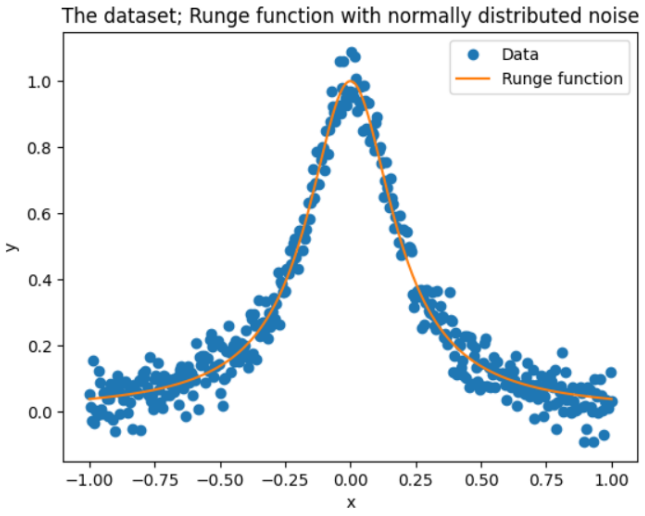

Figure 2. *The dataset, consisting of the Runge function with the addition of standard normally distributed noise with amplitude* 0.05. *Here we have 400 datapoints.*

20 % of the total data for all our analysis, except for the gradient descent results, where the testing data size was 30 % (though the difference is small). We always trained our models on the training data, and computed statistics (such as the MSE and R^2 metric) based on the model's predictions on the testing data.

Before training the models, we used the `StandardScaler` class by scikit-learn [7] to perform a *standard scaling* of the columns in the design matrices of the training data. That is, we subtracted the mean and divided by the standard deviation in each column. Moreover, we did not include the first intercept column. One of the reasons for normalizing the data is to ensure numerical stability. The components of the features columns corresponding to large polynomial degrees will have very small values, which could lead to floating point errors, as well as a nearly-singular matrix $\boldsymbol{X}^T\boldsymbol{X}$. But more importantly, for Ridge and Lasso regression, we want to have a uniform penalty for all parameters (except for the intercept, which we remove in order to not penalize a net offset in the data). That requires having the same normalization for all features. Standard scaling could be problematic in the presence of outliers [7], but this is not a big issue, since the input is confined to the interval $[-1, 1]$.

We also used funtionality from scikit-learn for other purposes. In particular, in order to resample from the training data in the implementation of bootstrapping, we used the `resample` function from scikit-learn [7]. Moreover, in order to quantify the performance of a cross validation (in particular, the mean testing MSE over the folds), we used the `cross_val_score` function. We also found it useful to use scikit-learn to find the optimal parameters of Lasso regression using the class `linear_model.Lasso`, so that we could compare to our

gradient descent results.

III. RESULTS AND DISCUSSION

1. *Analytical results for OLS and Ridge regression*

In order to find an appropriate number of datapoints in the dataset, we visualized the MSE as a function of the number of datapoints for different polynomial fits to the data. We used OLS regression and the analytical expression in Eq. (5) to train the model. The results are presented in the supplementary material on GitHub. We found that the MSE fluctuated, but that it converged for large numbers of datapoints. For several polynomial fits, it appeared that using several hundreds of datapoints would be a good compromise between having a MSE with relatively small fluctations and having relatively short runtimes. Thus, we settled on using $n = 400$ datapoints throughout.

In our analysis, we first used the analytical solutions in Eqs. (5) and (8) to compute the MSE and R^2 metrics as a function of polynomial degree for OLS and Ridge regression. We used polynomial degrees from 1 to 15, and λ-values between 10^{-5} and 10^3 for Ridge regression. See Figure (3) for the OLS result and Figure (4) for the Ridge result.

As the polynomial degree increases, we see that the MSE decreases and R^2 increases. Hence, the model performance improves. In particular, for a degree of 15, the R^2 value for OLS regression is approximately 0.95, meaning that the model explains 95% of the variability in the data.

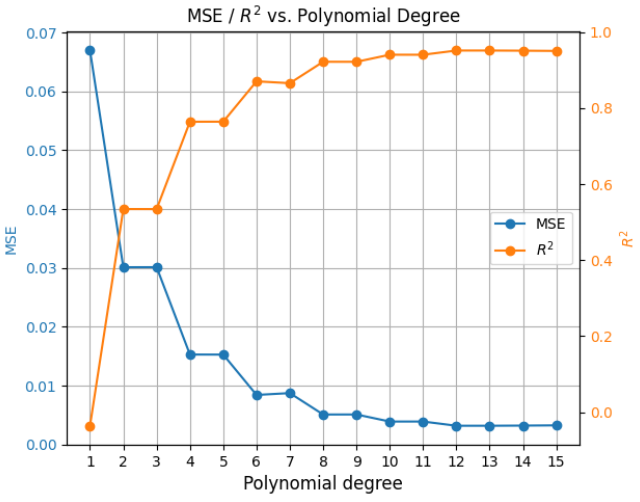

Figure 3. *MSE and R^2 as a function of polynomial degree, for OLS regression. The MSE shrinks and R^2 rises as the polynomial degree increases.*

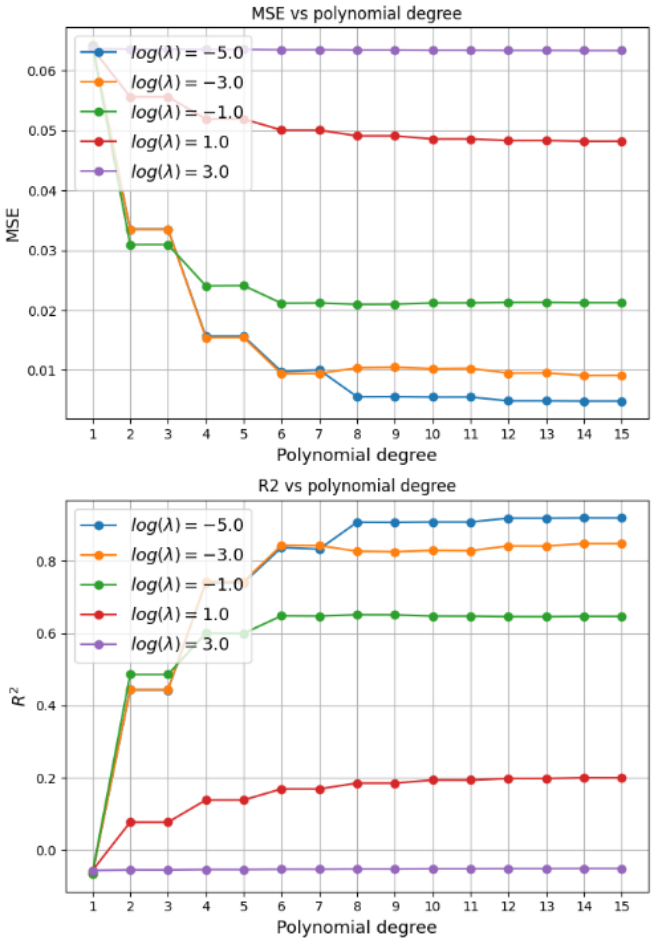

Figure 4. *MSE and R^2 as a function of polynomial degree, for Ridge regression. The values of $log_{10}(\lambda)$ that are used is indicated.* Top: *MSE.* Bottom: R^2.

In the case of Ridge regression, we notice that both the MSE rises and the R^2 score decreases for higher values of λ. In particular, when λ is very small ($\sim 10^{-5}$), one may notice by comparing Figure 3 and 4 that the performance metrics of Ridge regression are close to that of the OLS results. This may be expected, since we expect to recover OLS when $\lambda \to 0$. Furthermore, when λ is very large ($\sim 10^3$), we see that performance no longer improves for higher polynomial degrees. This indicates that λ is large enough that the optimal parameters have gone to 0, as suggested by Eq. (8).

Since model performance declines for higher values of λ, it seems that as far as analytical results are concerned, the OLS regression method gives better results.

For the OLS results, we may also note that for even polynomial degrees (such as 2, 4 and 6), we see that the performance metrics for OLS regression are approximately the same as for models with one higher degree. This may be due to the fact that the Runge function is an even function. Hence, the optimal parameters for the odd terms in the polynomial should be very small, meaning that adding such a term should not improve

performance.

In Figure 5, we show a plot the first few optimal parameters of OLS regression, as a function of the degree of the polynomial fit. We notice that both θ_1 and θ_3 are small relative to the even parameters, which is evidence for our suggestion above. Moreover, one finds that for high polynomial degrees, the (even) parameters eventually converge (as can be seen for θ_2, though θ_4 has not converged yet in the figure). With that said, for higher order features, the parameters rise very quickly, and it takes much longer before they converge. By then, the model may already overfit. Thus, it appears that the running of the parameters is not a good indication of model performance.

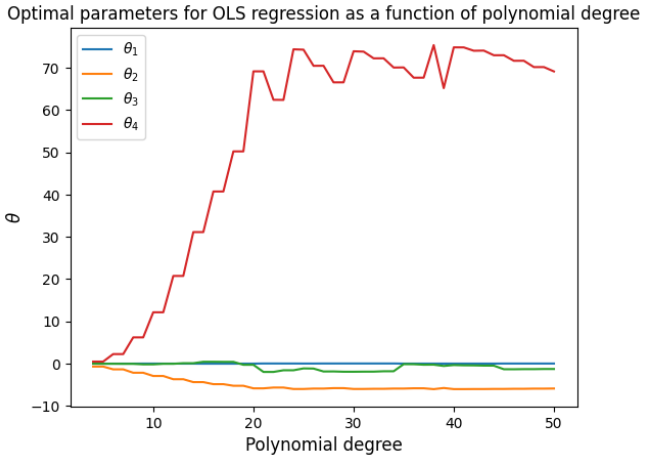

Figure 5. *Optimal parameters for OLS regression as a function of polynomial degree. The subscript on the parameters indicates the order of x that they correspond to in the polynomial fit.*

2. Gradient descent results

Next, we analyzed the gradient descent techniques on the three regression methods. We chose to use a six dimensional polynomial as a model, and for Ridge and Lasso regression, we used $\lambda = 10$. These choices were partially made from inspecting Figure 4, where it appears that Ridge regression gives significantly different performance relative to OLS for $\lambda \sim 10$, and that relative to the simplest polynomial fits, a polynomial of degree six has good performance.

The model parameters were iterated over time using Eqs. (11) and (14)-(17) with the gradients in Eqs. (4), (7) and (10). We calculated $1 - R^2$ (ie. the amount of data not explained by the model) as a function of the number of iterations. The hyperparameters that were chosen were $\alpha = 0.7$ (for momentum GD), $\rho = 0.9$ (for RMSProp GD), as well as $\beta_1 = 0.9$ and $\beta_2 = 0.999$ (for Adam GD), as suggested by [3]. The initial parameters $\boldsymbol{\theta}^{(0)}$ were chosen from a uniform distribution in the interval $[-1, 1]$, but the same initial values were used for all methods. The results are shown in Figures 6, 7 and 8. The gradient descent results were compared with the analytical value of $1 - R^2$ found from predicting on the optimal parameters of OLS and Ridge in Eqs. (5) and (8), as well as scikit-learn's solution for Lasso regression.

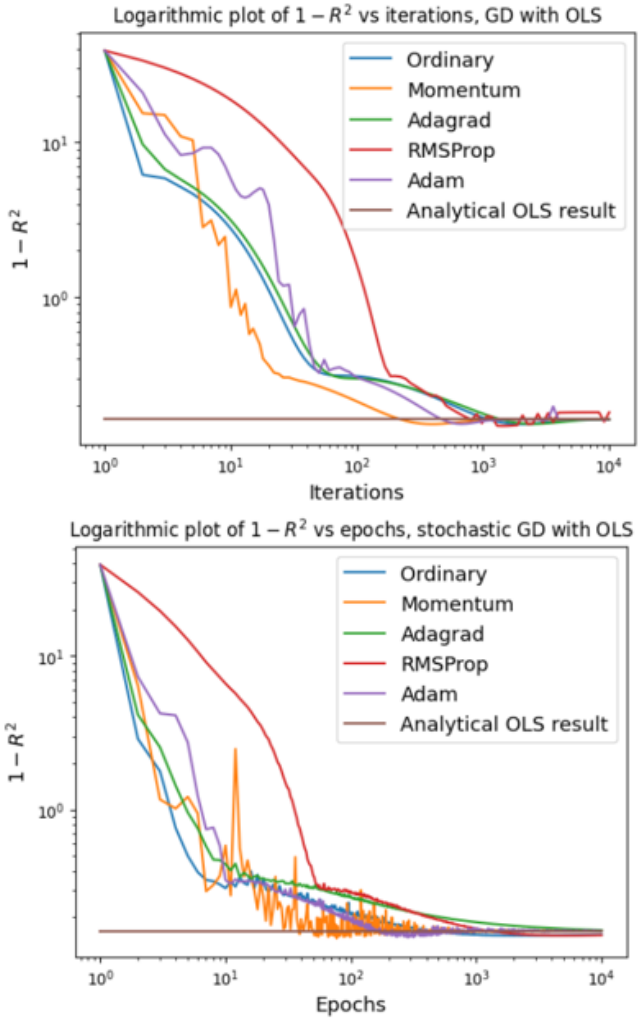

Figure 6. $1 - R^2$ *score vs iterations (or epochs) for gradient descent with OLS regression. We compare with the score for the analytical solution.* Top: *Results for non-stochastic gradient descent.* Bottom: *Results for stochastic gradient descent, with 8 minibatches.*

Table I. *Learning rates η used for each combination of GD algorithm and regression technique, for both the stochastic and non-stochastic GD results.*

	OLS	**Ridge**	**Lasso**
Ordinary	0.2	0.04	0.001
Momentum	0.3	0.002	0.001
Adagrad	0.3	0.4	0.2
RMSProp	0.008	0.005	0.016
Adam	0.1	0.05	0.02

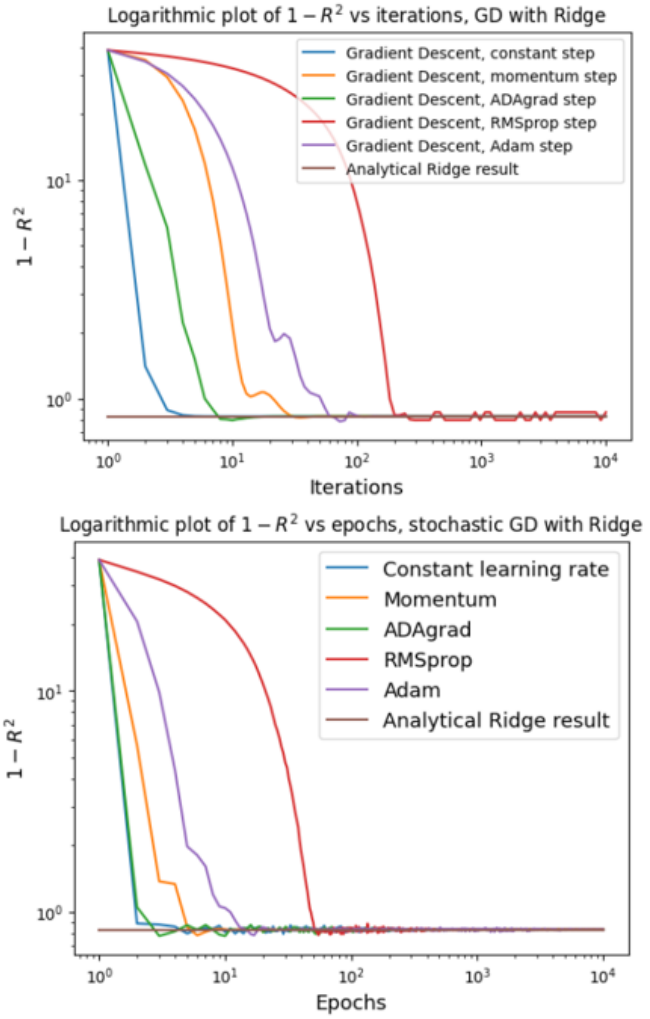

Figure 7. $1-R^2$ *score vs iterations (or epochs) for gradient descent with Ridge regression. We compare with the score for the analytical solution.* Top: *Results for non-stochastic gradient descent.* Bottom: *Results for stochastic gradient descent, with 8 minibatches.*

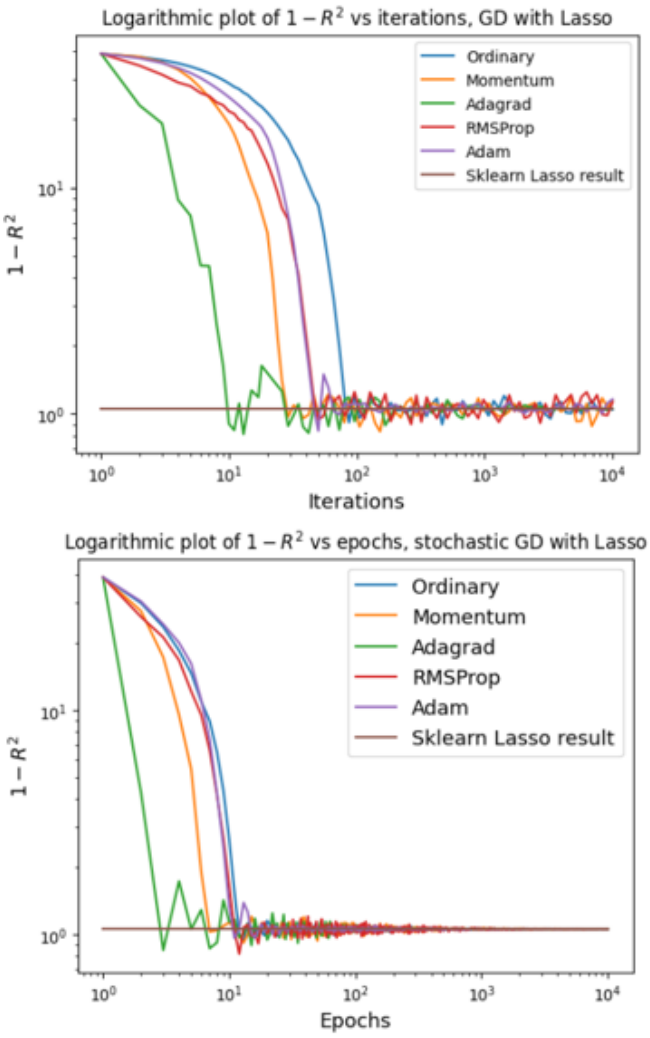

Figure 8. $1-R^2$ *score vs iterations (or epochs) for gradient descent with Lasso regression. We compare with the score for the solution from scikit-learn.* Top: *Results for non-stochastic gradient descent.* Bottom: *Results for stochastic gradient descent, with 8 minibatches.*

In this project, we were interested in getting a first impression of the performance of the GD algorithms, and so we did not need to carefully fine tune the learning rates η. However, in order to get interpretable results, we found it useful to nonetheless choose rates that were appropriate for each method, rather than using a single value on all algorithms. Thus, we landed on a middle ground; we did a manual grid search by plugging in different learning rates and numbers of iterations and visualizing the development of $1-R^2$ as a function of these iterations. We first did a broad search to find a range of values that could be useful, before doing a finer search in these ranges. We made two main considerations; first, we wished to use learning rates that caused convergence to the minimum after the least number of iterations, which favors higher learning rates. However, we also wanted to minimize fluctuations around the minimum, which favors lower learning rates. Thus, we did a rough search for the learning rates where each of the algorithms had small and comparable fluctuations in $1-R^2$ around the minimum. If this search led to a range of possible learning rates, all of which converged after the same number of iterations, then we typically chose a value that gave the smallest fluctuations *before* the minimum was reached. We then ended up with the values shown in Table I.

In Figures 6, 7 and 8, we also show the stochastic gradient descent results. Again, we visualized $1-R^2$ as a function of the number of *epochs*, with the same initial learning rates η_0 as the learning rates of the corresponding non-stochastic algorithms. We used $n_{batches} = 8$ minibatches in each epoch, corresponding to a batch size of 50.

From the plots, we can make some important observations:

- All of the gradient descent algorithms converge towards either the analytical or scikit-learn solutions for the three regression techniques. This is a desired consistency that gives confidence in our implementation of gradient descent. However, one finds that for OLS and Ridge regression, the analytical value of $1 - R^2$ at the minimum in the cost function is 0.163 and 0.831 respectively, whereas it is 1.06 for Lasso regression. Thus, with OLS regression, the model only does not explain 16.3% of the output variability, whereas Ridge regression does not explain as much as 83.1%, and Lasso regression explains *none* (or, in fact worse than none) of the variability. Thus, the analytical OLS result is clearly preferred for our simple dataset. If one had used the average of the training data instead of the testing data in Eq. (21), then one would get a value of exactly 1.0 for $1 - R^2$ for Lasso regression. Considering that the output is scaled, the reason for this is presumably that the value of λ is large enough that *all* parameters are set to exactly zero.

- From Table I and the plots, we see that for both OLS and Ridge regression, the RMSProp learning rate must be very small ($\sim 10^{-2}$ to 10^{-1} times smaller) relative to the Adagrad and Adam algorithms in order to ensure comparably tiny fluctuations around the minimum. In figure 6 and 7, one can even see that with such a small learning rate, there are still some fluctuations left. We see that this low learning rate causes the slowest convergence (although, probably owing to the fact that the vector $\boldsymbol{r}$ in Eq. (16) adapts to a slowly decreasing gradient, we see that $1 - R^2$ picks up pace towards the minimum after the first few iterations). The reason such a small learning rate is required may be that in the RMSProp algorithm, the change in the parameters between iterations is adapted to the small gradient around the minimum, so that the parameters do not settle there as easily.

- From Table I, it is clear that Ridge and Lasso prefer much lower learning rates ($\sim 10^1$ or 10^2 times smaller) for ordinary and momentum GD compared to OLS regression. This is probably a clear consequence of the fact that the extra penalty term in the cost function of Ridge and Lasso regression provide much steeper gradients, which therefore amplifies the speed of convergence. Thus, a smaller learning rate is required to give both fast convergence and small oscillations around the minimum.

- During our manual grid search for the learning rates for Lasso regression, we noticed an interesting phenomenon. Regardless of which values we chose, we were not able to eliminate significant fluctuations around the minimum for most GD algorithms. These fluctuations can clearly be seen by contrasting the non-stochastic results in Figure 8 with Figures 6 and 7. There is a likely explanation for this; in Eq. (10), we see that the Lasso gradient includes a discontinuous contribution that, when combined with gradient descent, tends to push the parameters towards 0. Since we argued that the optimal parameters were exactly zero in our case, this implies that that this term acts as a kind of constant "restoring" force that pushes the parameters in the direction towards the minimum. Thus, similar to a harmonic oscillator in physics, the parameters will tend to oscillate. However, in the stochastic GD results, we see that these oscillations attenuate over time. Thus, it appears that the learning schedule works as it should.

- By comparing the stochastic with the non-stochastic GD results for the three regression techniques, we see a general pattern where the stochastic algorithms tend to converge significantly faster than the corresponding non-stochastic ones, especially for Lasso and Ridge regression. For instance, for Ridge regression, the Adam and RMSprop methods converge after roughly 100 and 200 iterations for non-stochastic GD, respectively, but after only roughly 20 and 60 epochs with stochastic GD. This confirms the theoretical idea that stochastic GD gives faster convergence.

- Surprisingly, we did not find significant noise in the parameters for most GD algorithms when we used stochastic GD. This persisted when we tried more than 8 minibatches. However, in Figure 6, we see that momentum GD with OLS regression is a notable counterexample. It has significant fluctuations that eventually attenuate due to the learning schedule. This result may be reasonable, as the velocity in the momentum GD algorithm could, over time, accumulate the noise in the gradient of the minibatch cost function.

In general, the behavior of the various algorithms was not significantly different from one another with the simple dataset and cost functions used in our work. For Ridge regression, it even appears that the simplest ordinary GD algorithm had the fastest convergence. With that said, however, it appears that the Adagrad algorithm, particularly with stochastic GD, has significant benefits. It was one of the algorithms with the fastest convergence for all three regression methods (even being *the* fastest in Lasso regression). Moreover, in the non-stochastic Lasso case, then unlike the other methods, its fluctuations around the minimum attenuated over time. Finally, we should note that, in our manual grid search, we noticed that, in the case of Ridge and OLS regression, the Adagrad algorithm was robust for a large range of choices for learning rates; in particular, for learning rates larger than roughly 0.3 and

0.4 with OLS and Ridge regression, respectively, the parameters converged after roughly the same number of iterations.

One might have naively expected that Adam would have been better, considering its current popularity, but this shows that the appropriate methods depend on the dataset and models used.

We end our discussion of gradient descent with Figure (9). During our manual grid search on ordinary non-stochastic GD, we noticed that for specific values of the learning rates, the development of $1-R^2$ as a function of the number of iterations became unstable; in particular, for learning rates slightly above these values, the value of $1-R^2$ increased exponentially. In Figure (9), we compare the development for some learning rates with values right under these values. We see that close to a certain limit, there are large oscillations in $1-R^2$, and there is a very late convergence to the minimum.

We can compare these limits with the theoretical limit in Eq. (12), using the Hessian matrices in Eq. (13). We then find limits of 0.3436008 and 0.07746 for OLS and Ridge regression, respectively. We see that these values are very close to the learning rates of 0.3436 and 0.0774 in Figure (9), showing close agreement between theory and observations.

3. *Resampling results*

Next, in our analysis of the bias-variance tradeoff for the OLS regression method, we tried combinations of different training and testing data (by using different seeds in the `train_test_split` function) and numbers of datapoints. For each combination, we visualized the bias and variance as a function of degree, and found the range in polynomial degrees where the bias and variance were at the minimum. In particular, after the initial drop in the bias and variance, they typically stayed at a minimum for a range of polynomial degrees before the variance started rising again. In Table II, we show these ranges, and in Figure 10, we show an example of this behavior. In our implementation, we used $B = 500$ resamples.

Table II. *Ranges of polynomial degrees where the bias and variance is at the minimum, for OLS regression. We show the results for each combination of seed-value for the splitting into testing and training data of the* `train_test_split` *function, as well as the number n of datapoints.*

n \ Seed	**40**	**44**	**48**
300	10-16	10-16	8-30
400	12-20	12-15	10-17
500	12-60	10-32	10-23

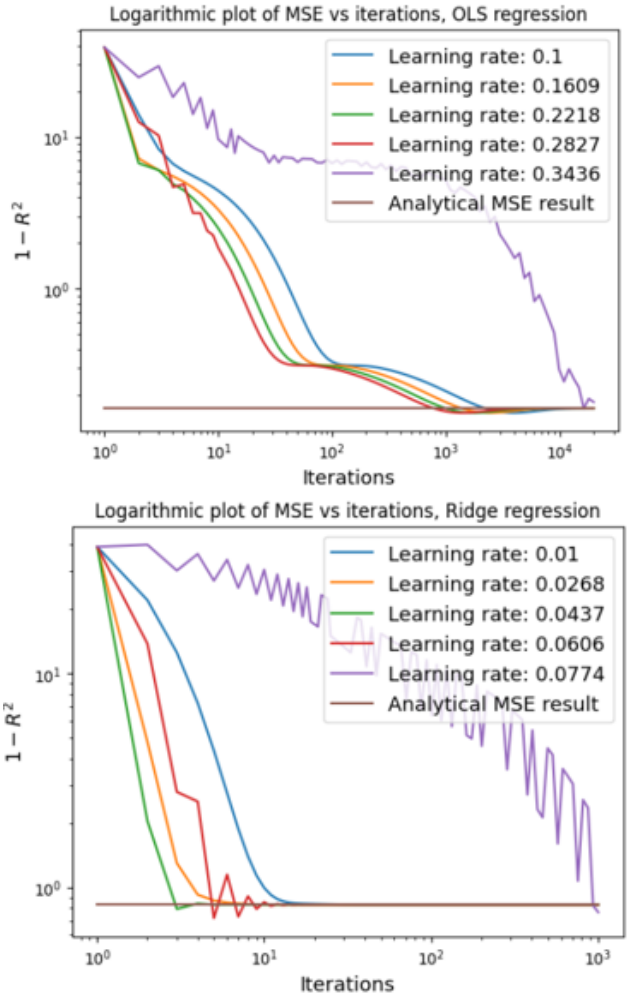

Figure 9. $1-R^2$ *as a function of of the number of iterations for different learning rates. One sees that at a specific learning rate, the development becomes unstable.* Top: *OLS regression.* Bottom: *Ridge regression, with* $\lambda = 10$.

We see that the range of optimal model complexity varies significantly, depending on the number of data points and the data we use to train and test the model. In particular, the polynomial degree at which overfitting starts to occur varies a lot. However, we also see that the minimum in variance is reached with a polynomial degree of 12 in all cases, indicating that this is a good choice to achieve a minimally complex model with good performance.

To further explore resampling techniques and the impact of model complexity, we compared the MSE obtained from bootstrap and 10-fold cross-validation for OLS, Ridge, and Lasso regression, as shown in Figure 11. In this analysis, we used a smaller dataset of 100 data points, instead of the usual 400, in order to emphasize the effects of overfitting. For the bootstrapping procedure, we generated $n_{bootstraps} = 100$ resamples, while for cross-

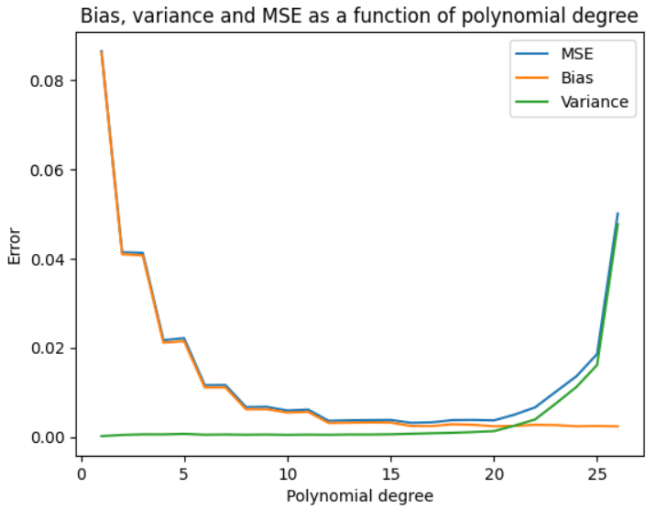

Figure 10. *Example plot of bias, variance and MSE as a function of polynomial degree. Here, we use 400 datapoints and a random state of 40 to split into training and testing data. We see that the bias and variance is at a minimum for polynomial degrees between 12 and 20 (compare with Table II).*

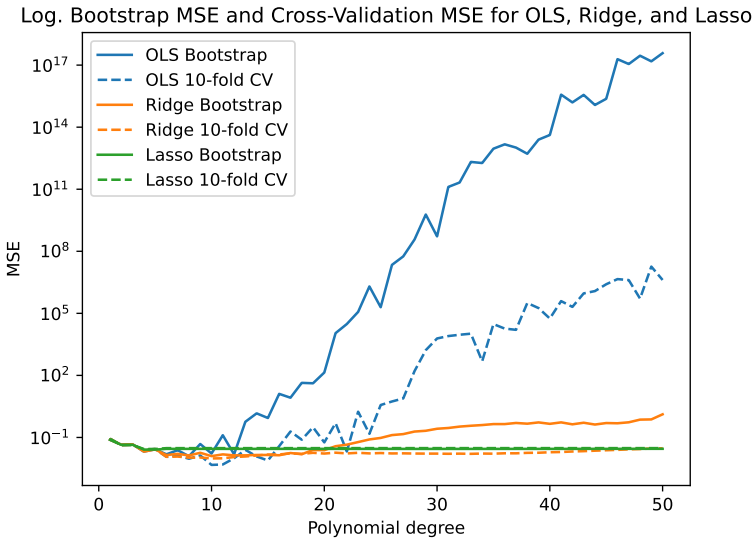

Figure 11. *Plot comparing the MSE obtained from bootstrap resampling and 10-fold cross-validation for OLS, Ridge, and Lasso regression.*

validation we used $k = 10$ folds. The Ridge and Lasso regularization parameters were fixed at $\lambda_{Ridge} = 10^{-2}$ and $\lambda_{Lasso} = 10^{-2}$, and were chosen to ensure some regularization, while still showing the results of overfitting.

The results reveal that for OLS regression, the MSE grows rapidly for polynomial degrees above roughly 15, indicating strong overfitting as the model complexity increases. Ridge and Lasso regression, on the other hand, maintain substantially lower and more stable MSE values across the whole range of polynomial degrees. Moreover, Lasso regression appears to give an even smaller MSE than Ridge regression. This confirms that regularization effectively suppresses overfitting by constraining the model coefficients. Cross-validation tends to yield slightly lower MSE than bootstrap, which is consistent with the results.

These findings are in strong agreement with the earlier discussion on Ridge and Lasso regression. The inclusion of a penalty term stabilizes the optimization and reduces sensitivity to collinearity. While OLS quickly overfits in high-complexity models, regularized models like Ridge and Lasso remain robust and generalize well to unseen data. This behavior aligns with the results in [4], where Ridge and Lasso are shown to substantially reduce variance and stabilize model performance as complexity increases.

IV. CONCLUSION

In this project, we have analyzed the performance of the OLS, Ridge and Lasso regression techniques for polynomial fits to the Runge function, as well as the behavior of various stochastic and non-stochastic gradient descent algorithms used to train the models. We found that the OLS technique generally gave the most accurate predictions, with a higher R^2 score than the corresponding values of Ridge and Lasso. Using a bias-variance tradeoff analysis, we found that for OLS regression, polynomials with a degree around 12 gave the best performance, having both a small bias and variance. However, from our results on k-fold cross validation, we also found that even with a relatively "small" value of $\lambda = 10^{-2}$ for the regularization parameter, Ridge and Lasso significantly suppressed overfitting. This means that their performance could be sustained for larger polynomial degrees.

We found that stochastic gradient descent generally gave faster convergence compared to the corresponding non-stochastic method. While it was harder to compare the different gradient descent algorithms, we found that Adagrad generally performed well for all regression techniques. It had relatively fast convergence, a damping of oscillations around the minimum of the cost function (including in the case of Lasso regression), and its convergence was robust with respect to the choice of learning rate.

Our work was limited by the choice of dataset, which did not make it possible to explore the full scope of the capabilities allowed by the regression techniques and gradient descent algorithms. Moreover, we sometimes employed manual "trial and error" searches for parameter values, including in the search after learning rates for gradient descent. This could raise suspicion for some of our results, in particular for the numerical values that we employed.

Hence, in a future study, it would be advantageous to investigate a more complex, high dimensional dataset. Despite the limitations, our implementation of the central methods are sufficiently generalized that they should be easy to make use of with other data. One could also implement a more quantitative grid search algorithm for optimal parameter values. In the case of learning rates, one could vary both the number of iterations and learning rates and use an acceptable tolerance for the amplitude of oscillations around the minimum. With these improvements, one might obtain more robust comparisons between different methods and regression techniques.

[1] S. Angra and S. Ahuja, Machine learning and its applications: A review, in *2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)* (2017) pp. 57–60.

[2] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization (2017), arXiv:1412.6980 [cs.LG].

[3] M. Hjorth-Jensen, Applied data analysis and machine learning (2025).

[4] T. Hastie, R.Tibshirani, and J.Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition. Springer Series in Statistics* (Springer, New York, 2009) pp. 389–414.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) `http://www.deeplearningbook.org`.

[6] R. Tappenden and M. Takáč, Gradient descent and the power method: Exploiting their connection to find the leftmost eigen-pair and escape saddle points (2022), arXiv:2211.00866 [math.OC].

[7] 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 (2011).