Regression and Gradient Descent Methods for Approximating Runge's Function
FYS-STK3155 - Project 1

Eirik Holm, Niklas Vestskogen
University of Oslo
(Dated: October 6, 2025)

Transforming data into knowledge often involves finding models that generalize well to unseen data. Polynomial interpolation suffers from instability when modeling nonlinear functions such as Runge's function. Increasing polynomial degree leads to overfitting and oscillations, known as the Runge phenomenon. We reframed the interpolation task as a regression problem and implemented Ordinary Least Squares, Ridge, and Lasso regression. Gradient descent variants were used for optimization, and model performance was validated using MSE, R^2, bootstrap, and cross-validation. At low regularization strength ($\lambda = 10^{-5}$), Ridge regression performed similarly to OLS but with $\approx$ 23.75% lower R^2. After optimization with momentum, Ridge achieved the lowest test MSE (0.0299), followed by Lasso (0.0305) and OLS (0.0416). Our findings demonstrate that combining regularization with adaptive optimization improves model stability by reducing coefficient fluctuations and mitigating the effects of noise.

I. INTRODUCTION

Transforming data into knowledge has a long history in science, exemplified as early as in the 16th century when Johannes Kepler discovered the empirical laws of planetary motion[1] by using Tycho Brahe's astronomical observations of the planet Mars [2]. Because real-world phenomena rarely can be described by closed-form analytical expressions, we collect data in order to approximate a general relationship between the inputs and outputs. This is the foundational idea of *supervised learning*, where models are trained on known input and output to capture the underlying relationship in order to predict outputs for unseen inputs [3].

A reasonable assumption is that through supervised learning, our model should be able to reproduce our observations exactly, while being able to approximate new, unseen observations. *Interpolation* gives us the framework for estimating $f(x)$ within a dataset. *Polynomial interpolation* seeks to fit a polynomial model $p(x)$ such that

$$p(x_i) = f(x_i), \quad i \in \{0, 1, 2 \ldots, n\},$$

with a polynomial degree $n-1$ or higher, passes through n data in order to estimate intermediate values[4]. The Runge phenomenon, depicted in Figure 1 with the corresponding error in Table I, illustrates the limitations of this method, with the error

$$\max_{x\in[-1,1]} |f(x) - p(x)|,$$

increasing when the number of equally spaced data increases, and subsequently the degree of polynomial, because of the oscillation that occurs at the endpoints. This is reminiscent of overfitting. A solution within interpolation could be *piecewise polynomial interpolation*, where we use a collection of lower degree polynomial interpolants [5],

$$S(x) = \begin{cases} p_0(x), & x_0 \leq x \leq x_1 \\ p_1(x), & x_1 \leq x \leq x_2 \\ \vdots \\ p_{n-1}(x), & x_{n-1} \leq x \leq x_n \end{cases}$$

to smoothen the oscillation at the endpoints. However, unwanted local oscillations can still occur. This raises the issue of overfitting: how can we fit data accurately while preserving the model's ability to generalize to unseen data?

Rather than constraining the model to pass directly through seen data, we should find a model that minimizes the error for unseen data. We will frame the problem as a regression task and apply Ordinary Least Squares, Ridge, and Lasso methods to approximate Runge's function

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

and use the *mean squared error* (MSE) as our cost function. We will use regularization techniques in tandem with hyperparameter tuning for *model selection* and MSE, bootstrap validation, train-test split, and cross validation for *model assessment*[6].

In section II, we will present an overview of regression models such as Ordinary Least Squares, Ridge and Lasso, and how these can be implemented with optimization algorithms, such as *gradient descent* with and without *momentum* and *stochastic gradient decent*, and how we can test the model's performance using resampling methods, such as bootstrap and cross-validation. In section III we will discuss and present our results, as well as pros and cons of the implemented methods and identify areas of improvement.

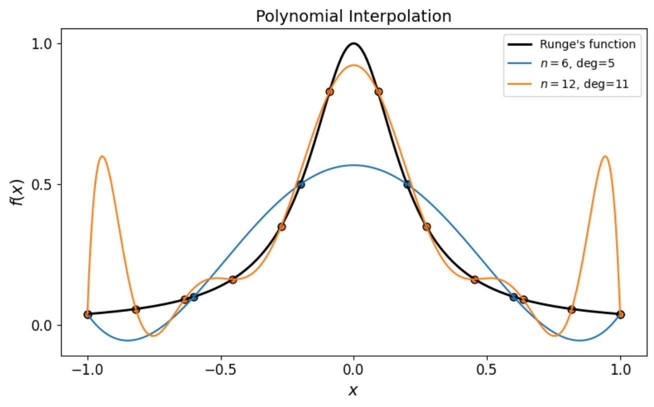


Figure 1: Runge's function and polynomial interpolants $p_5(x)$ and $p_{11}(x)$.

n data	Degree $(n-1)$	$\max\lvert f(x)-p_{n-1}(x)\rvert$
2	1	$9.61e^{-1}$
3	2	$6.46e^{-1}$
4	3	$7.07e^{-1}$
5	4	$4.38e^{-1}$
6	5	$4.33e^{-1}$
7	6	$6.17e^{-1}$
8	7	$2.47e^{-1}$
9	8	$1.05e^{0}$
10	9	$3.00e^{-1}$
11	10	$1.92e^{0}$
12	11	$5.57e^{-1}$
⋮	⋮	⋮
99	98	$1.65e^{2}$

Table I: Maximum interpolation error $\max|f(x)-p(x)|$ for Runge's function using equally spaced data on $[-1,1]$.

II. METHODS

Throughout this article, multiple methods for regression, metrics for evaluating performance, gradient descent and resampling techniques are used, this section is split into four subsections presenting each topic in order. We will first present the relevant regression methods, then go into more detail on how to measure their performance. All of the presented regression models can be optimized using various techniques, as such, the third subsection will tackle gradient descent, before we forth and finally present resampling. All code related to the project can be found in our GitHub repository[7].

A. Regression

In this section, three regression methods are presented *ordinary least squares, Ridge regression* and *Lasso regression.* They will be presented in order and cover both the theoretical basis, algorithmic approach and briefly, our own implementation.

1. *Ordinary Least Squares*

Ordinary least squares (OLS) is one of the most common methods for fitting linear regression models as it is easy to understand and implement compared to other regression models. Unlike polynomial interpolation, OLS regression does not require the model to match every data point or to use a minimum polynomial degree based on the number of data points. Instead, you select the number of predictors based on fit quality - such as R^2 and MSE. This flexibility is desirable when estimating nonlinear functions such as Runge's as we can approximate the function well without overfitting.

Since MSE is a measure of the lack of fit of the model to the data, the least squares method seeks to find the parameter estimates $\hat{\theta}$ that minimize this cost function, defined as

$$C(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2 .$$

Because the cost function is a quadratic function and X^TX is positive-semidefinite, the cost function is convex and must have a minimum. Setting the derivative with respect to θ equal to zero,

$$\frac{\partial C(\theta)}{\partial \theta} = 0,$$

and solving for θ gives us the closed-form solution

$$\hat{\theta} = \left(X^TX\right)^{-1} X^T y.$$

An additional motivation for approximating Runge's function with OLS is it's property of unbiasedness: we can use the sample mean $\hat{\mu}$ to estimate μ [8]. Among all unbiased estimates the OLS has the smallest variance according to the Gauss-Markov theorem [9]. We will revisit these points when we discuss the bias–variance trade-off.

In this project we implemented OLS using a self-written python code, following the structure of Algorithm 1, using Scikit-Learn's train_test_split [10] to split the feature matrix into a test and test dataset. The training set were used to train the model and the test set was used for measuring quality fit on unseen data using MSE and R^2. Because the OLS estimators are scale equivariant, meaning if we scale our estimators we still end up with the same predictor, we don't have to scale our design matrix [8].

Algorithm 1 Ordinary Least Squares

```
1: Given data matrix $X \in \mathbb{R}^n$, p predictors, targets $y \in \mathbb{R}^n$
2: Compute design matrix with polynomial features
       $X \in \mathbb{R}^{n\times p}$
3: Estimate parameters: $\hat{\theta} = (X^TX)^{-1}X^Ty$
4: Predictions: $\hat{y} = X\hat{\theta}$
5: Evaluate quality of fit with MSE and $R^2$
```

2. *Ridge Regression*

The reason we are not satisfied with least squares estimates is because of it's prediction accuracy [11]. OLS gives us low bias and potentially large variance due to how we calculate the estimators $\hat{\theta}$. Recall

$$\hat{\theta}_{OLS} = (X^T X)^{-1} X^T y$$

works only if $X^T X$ can be inverted, meaning $det(X^T X) \neq 0$ and X is linearly independent [11]. If the determinant is close to zero, meaning we can compute it's inverse, there will be high variance in our estimates due to small changes in y can cause large swings in $\hat{\theta}$. If our design matrix X has linearly dependent column vectors, $det(X^T X) = 0$, we will not be able to compute $(X^T X)^{-1}$ and in turn our estimators. This is more likely to happen with high-dimensional matrices, such as when we want to approximate Runge's function to a higher order polynomial. To mitigate this we can use Ridge regression.

Ridge regression solves this by imposing a penalty, also known as L2 regularization, to the least squares method [11],

$$X^T X \to X^T X + \lambda I, \quad \lambda \geq 0,$$

where I is the identity matrix and λ is the regularization strength. Giving us

$$\hat{\theta}_{Ridge} = (X^T X + \lambda I)^{-1} X^T y$$

By increasing λ we shrink the regression coefficients resulting in fewer extreme outliers and stabilizing the estimates by reducing the variance at the cost of bias [12]. We use hyperparameterization to select the regularization strength, exemplified by Figure 2.

As shown in Algorithm 2, Ridge regression extends OLS by *standardizing* the columns of the design matrix and by introducing an L2 penalty to the estimator. We standardize the feature column of the design matrix by subtracting the mean and divide by the standard deviation for each column.

$$X_{scaled} = \frac{X - \mu}{\sigma}$$

The reason for this is that we want the shrinking to be uniform across features. If we did not scale X then large variance features would be shrunk more and the low variance features would be shrunk less resulting in uneven penalization [13].

Our Ridge regression code implementation follows the order of operations given in Algorithm 2. Similar to our OLS implementation we will use Scikit-Learn for our train/test split and use MSE and R^2 as our quality fit measure. As scaling the feature matrix is crucial in Ridge, we used Scikit-Learn's StandardScaler for this. We will compare our results with OLS using the same seed, yielding same inputs and outputs when generating data, so that the results are comparable. In particular, we want to investigate whether the L2 penalty results in a better fit when $X^T X$ is close to zero. Furthermore, we will explore Ridge's dependence on the penalty strength λ and degree of polynomial.

Algorithm 2 Ridge Regression

```
1: Given $X \in \mathbb{R}^n$, p predictors, $y \in \mathbb{R}^n$, penalty $\lambda$
2: Compute design matrix with polynomial features
       $X \in \mathbb{R}^{n \times p}$
3: Standardize each column $X$
4: Estimate parameters:
       $\hat{\theta} = (X^\top X + \lambda I)^{-1} X^\top y$
5: Predictions: $\hat{y} = X\hat{\theta}$
6: Evaluate quality of fit with MSE and $R^2$
```

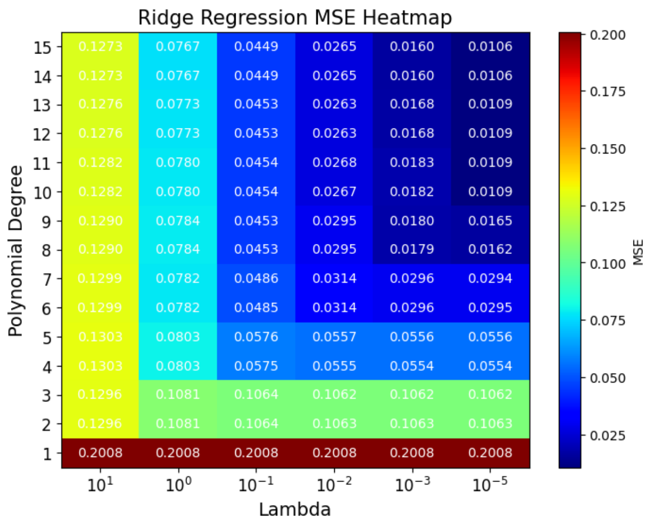


Figure 2: MSE as a function of polynomial degree and λ, with $n = 50$

3. *Lasso Regression*

Lasso regression, *Least Absolute Shrinkage and Selection Operator*, is a very similar method to Ridge regression as it introduces a penalty term into the cost function to mitigate the problem of overfitting at higher model complexities [14]. For Lasso, we use the L1 penalty,

$$||\theta||_1 = \sum_{j=1}^{m} |\theta_j|,$$

which leads to the Lasso cost function,

$$C_{Lasso}(\boldsymbol{X}, \theta) = (y - \boldsymbol{X}\theta)^T (y - \boldsymbol{X}\theta) + \lambda ||\theta||_1.$$

Taking the derivative of the cost function with respect to θ and recalling that the derivative of the absolute value is $\frac{d|\theta|}{d\theta} = sgn(\theta)$, we make two important observations [11][15]:

$$\boldsymbol{X}^T\boldsymbol{X}\theta + \lambda sgn(\theta) = 2\boldsymbol{X}^T y$$

1. We cannot obtain a closed form solution for $\hat{\theta}$ as the function is discontinuous at zero.

2. The L1 penalty is constant and does not scale with the magnitude of the coefficients like the L2 penalty. Consequently, if a coefficient contribution to the cost function is sufficiently small, i.e. less than λ, it will be driven to zero by the penalty term.

The challenge posed by the first observation can be solved using convex optimization algorithms like gradient decent, which we will come back to in the next section. The second observation results in a form of automatic feature selection, where only the coefficients that contribute to the cost function are retained, while those that don't are driven to zero. This is especially relevant for larger datasets with many parameters. Algorithm 3 shows the general structure of Lasso regression.

Algorithm 3 Lasso Regression

```
1: Given $X \in \mathbb{R}^{n\times p}$, $y \in \mathbb{R}^n$, penalty $\lambda$
2: Compute design matrix with polynomial features
3: Standardize each column $X$
4: Compute the Lasso estimator using a convex optimization
   method, e.g. gradient descent, see algorithm 4, 5 and 6:
5: Predictions: $\hat{y} = X\hat{\theta}$
```

B. Evaluation metrics

In *supervised learning* we are interested in finding the machine learning algorithm that makes the best prediction on *unseen data* based on the patterns learned from the *seen data* [16]. Since we already know what the response y is given x we can evaluate how good the prediction $\hat{y}$ is. In regression we split our observed data (x_i, y_i) into two batches: training and testing dataset. Usually the test size is 20-30% of our data. The reason for this is that we want to train a model that makes the best possible prediction, meaning that a too-small train size may fail to capture the underlying pattern. After we have trained our model we can evaluate the performance of the model using our test data. Since our goal is to build a model that can predict future inputs, we evaluate its performance using the test data. To measure performance we use model assessment metrics such as *mean squared error*, R^2 and *bias-variance tradeoff.*

1. *Mean Squared Error*

One way to quantify how good a model is at predicting is the *mean squared error*, defined as

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

where $\hat{y}_i$ is the prediction for the ith observation. The MSE measures how much the collection of predictions deviates from the true response. A small MSE means that the predictions are on average close to the observations, and a large value corresponds to predictions being on average further away. In other words we can say that MSE is a measure of lack of fit of the mode.

The squaring of the error $(y_i - \hat{y}_i)$ ensures that the terms don't cancel each other, making it easier to interpret the result. It also means that large errors get penalized more than small errors, making it more sensitive to outliers. One of the most useful properties is that this leads to a convex quadratic function, which regression models such as *OLS* and *Ridge* optimize to find their optimal estimator $\hat{\theta}$ [17].

Beyond model assessment, MSE can also be used as a metric for model selection. In Figure 3 we can see MSE as a function of polynomial degree. Given that we evaluate performance on the test data and that low MSE is desirable, we can choose the model complexity that yields the lowest test MSE.

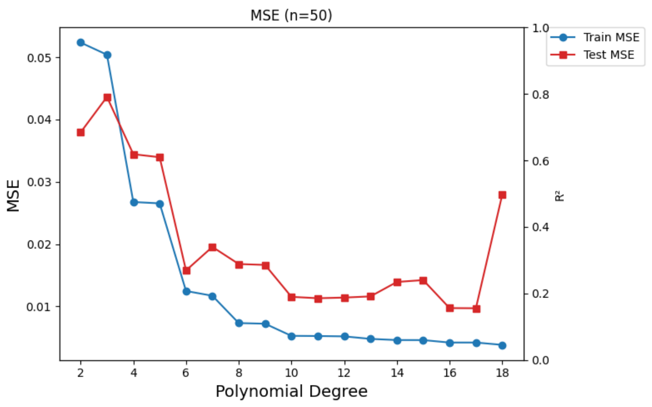


Figure 3: Training and testing MSE as we vary the polynomial degree for $n = 50$ using ordinary least squares.

2. *Bias-variance tradeoff*

Recall that we can consider the test MSE as a measure of the lack of fit of the model to the data using linear algebra analysis. To get a deeper understanding of why that is we will use *bias-variance tradeoff* from statistical analysis and the Figure 3 to get a better intuition.

The characteristic U-curve for the test MSE has in Figure 3 is the result of two competing statistical properties, the *bias*^{2} and *variance*. In the context of machine learning models, *bias* refers to the error introduced by approximating a complex function using a much simpler model. For instance, approximating the polynomial $x^2+3x^4+\sqrt{2}x^5$ using a linear model $\theta_0+\theta_1 x$ will result in bias as it does not produce an accurate estimate. Generally more flexible models, e.g. higher polynomial degree, result in less bias [18] as the predictions $\hat{y}$ is closer to y. On the other hand, *variance* measures the consistency of our predictor $\hat{y}$ if we estimate it using different subsets of the training data set. If small changes in the training data results in large changes in $\hat{y}$ we say that the model has high variance. Using a feature matrix with high polynomial degrees as our predictors will amplify these changes, resulting in higher variance.

Statistically, without deriving it, the expected error gives the the following bias–variance decomposition

$$\mathbb{E}\big[(y-\tilde{y})^2\big] = \underbrace{\mathbb{E}\big[(y-\mathbb{E}[\tilde{y}])^2\big]}_{\text{Bias}[\tilde{y}]} + \underbrace{\mathbb{E}\big[(\tilde{y}-\mathbb{E}[\tilde{y}])^2\big]}_{\text{Var}[\tilde{y}]} + \underbrace{\sigma^2}_{\text{Noise variance}} \tag{1}$$

Now we are better equipped to understand the behavior of the test error in Figure 3, as model complexity is varied. In Figure 4 we have used this decomposition to plot the bias and variance as a function of polynomial degree. Using what we know about bias and variance we can say that bias decreases and variance increases as we increase the model complexity. This means that we cannot decrease the bias and variance as the same time, thus the name *bias-variance tradeoff.*

When we have high bias we say that our model is *underfit* because our model is not complex enough to capture the underlying patterns in the data. In contrast, *overfitting* is when a model is too complex to capture the pattern [19].

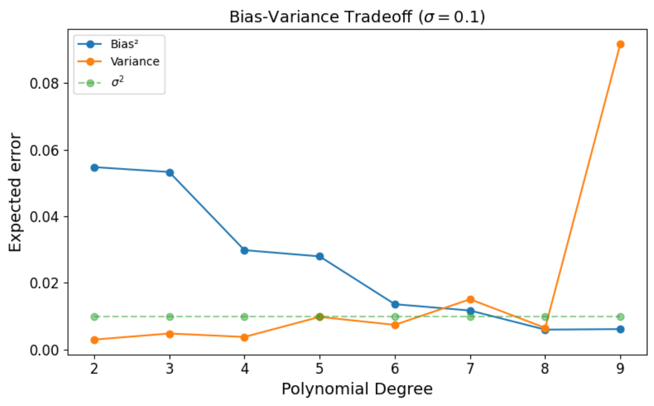


Figure 4: *Bias*^{2}, Variance and noise as we vary the polynomial degree for $n = 50$ using ordinary least squares.

3. R^2 Score

The coefficient of determination (R^2) is an alternative measure of fit. Where the MSE measures how far away our prediction is on average, the R^2 measures the fraction of the response variance that can be explained by the model [20] [21]. To calculate R^2, we use

$$R^2 = 1 - \frac{\sum_{i=0}^{n-1}(y_i-\hat{y}_i)^2}{\sum_{i=0}^{n-1}(y_i-\bar{y})^2} \tag{2}$$

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

where $\bar{y}$ is the mean value of y. The denominator in equation (1) is the *total sum of squares* (TSS), measuring the total variance in our target before regression is performed, and the numerator is the *residual sum of squares* (RSS), measuring the variability that is unexplained after performing the regression [22]. For intuition we can look at equation (2) as

$$R^2 \approx \frac{\text{Variance explained by the regression line}}{\text{Total variance around the mean}}.$$

R^2 takes a value between 0 and 1. An R^2 score of 1 indicates that the regression line explains all of the variability, meaning a perfect fit. Contrarily, an R^2 of 0 means the regression like explains none of the variability.

Similar to MSE, the R^2 can also be used as a metric for model selection. Given that we evaluate performance on the test data and that high R^2 score is desirable, we can choose the model complexity that yields the highest test R^2.

C. Gradient decent

Recall that we found the values of $\hat{\theta}$ that minimize the cost function using closed-form solutions. As pointed out in [23], computing predictor estimates using closed-form solutions can become impractical in high-dimensional settings due to the computational demands of inversion of the feature matrix. Instead, we must use numerical methods to compute the minimum. One of these methods is the *gradient descent.*

Gradient descent is an iterative algorithm that moves on the curve, or surface as seen in Figure 5, in the opposite direction of the gradient

$$\nabla_\theta C(\theta),$$

until a local or global loss minimum is reached. Algorithm 4 describes how we compute the estimators. In each iteration, we take a step in the steepest descent, where the step size is determined by the gradient evaluated at that point and the learning rate, η,

$$\theta^{n+1} = \theta^n - \eta\nabla_\theta(\theta^n).$$

Since the MSE is convex, convergence is guaranteed given enough iterations, the iterations approach the estimator $\hat{\theta}$ where the gradient term vanishes,

$$\nabla_\theta C(\hat{\theta}) = 0.$$

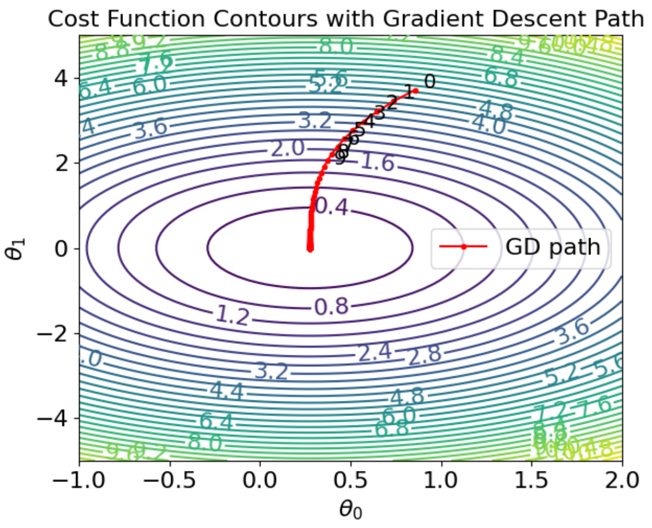


Figure 5: Contour plot of the MSE cost function with GD path from the starting point (θ_0, θ_1) = (1,4).

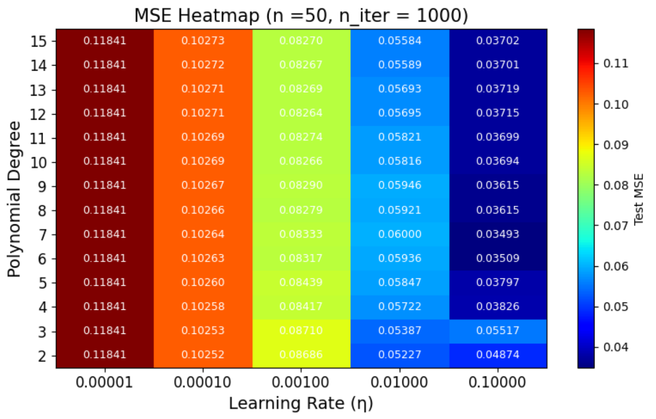
However, this method has some drawbacks as it can lead to slow convergence or even failure to converge as it takes large steps in the direction of steepest descent and small steps where the gradient is shallow. In Figure Figure 6 we can see that we don't converge with a small learning rate given 1000 iterations.

Gradient descent is also computationally heavy for large dataset and higher-dimensional feature matrices [24], as each iteration requires computing the gradient over the entire dataset, making it slow and inefficient at scale [25]. Depending on our initial guess for θ and learning rate, there is a possibility that we converge to a local minima or saddle point when using this algorithm on non-convex cost functions, introducing uncertainty in the final results. To overcome these challenges, variants of gradient descent such as with *momentum* and *stochastic gradient descent* have been developed to accelerate convergence and handle large datasets more efficiently

Algorithm 4 takes us through our implementation of GD. We used Jax [26] to compute the derivative of our cost function. It's not uncommon to stop iterating if $\theta^{n+1} - \theta^n <$ Threshold, given a threshold value, however, we used number of iterations as our only stop parameter.

Algorithm 4 Gradient Descent

```
1: Given learning rate $\eta$, iterations $T$, initial parameters $\theta^{(0)}$
2: for $t = 0, 1, \ldots, T-1$ do
3:     Compute gradient: $\nabla_\theta C(\theta^n)$
4:     Update parameters: $\hat{\theta} \leftarrow \theta^{n+1} = \theta^n - \eta\nabla_\theta(\theta^n)$
5: end for
6: Predictions: $\hat{y} = X\hat{\theta}$
```



Polynomial Degree	0.00001	0.00010	0.00100	0.01000	0.10000
15	0.11841	0.10273	0.08270	0.05584	0.03702
14	0.11841	0.10272	0.08267	0.05589	0.03701
13	0.11841	0.10271	0.08269	0.05693	0.03719
12	0.11841	0.10271	0.08264	0.05695	0.03715
11	0.11841	0.10269	0.08274	0.05821	0.03699
10	0.11841	0.10269	0.08266	0.05816	0.03694
9	0.11841	0.10267	0.08290	0.05946	0.03615
8	0.11841	0.10266	0.08279	0.05921	0.03615
7	0.11841	0.10264	0.08333	0.06000	0.03493
6	0.11841	0.10263	0.08317	0.05936	0.03509
5	0.11841	0.10260	0.08439	0.05847	0.03797
4	0.11841	0.10258	0.08417	0.05722	0.03826
3	0.11841	0.10253	0.08710	0.05387	0.05517
2	0.11841	0.10252	0.08686	0.05227	0.04874

Figure 6: MSE as a function of polynomial degree and η for n=50 using OLS with GD.

1. *Gradient Descent with Momentum*

Gradient descent with momentum can be imagined as rolling a physical ball down a slope—the ball is affected by gravity, thus it accelerates and gains velocity and momentum down the slope. The steeper the slope, the greater the momentum becomes. Even if the slope becomes less steep or changes direction, the ball will "remember" its previous momentum, helping it overcome small local minima and dampen oscillations in narrow valleys.

One of the main drawbacks of vanilla gradient descent is that the step size and direction only depend on the gradient at that point and the learning rate, η. As the gradient approaches a value close to zero, the step size also converges towards zero, even though this minimum might not be the global minimum. This is especially true for datasets that result in noisy or small but consistent gradients [27].

To mitigate this and to accelerate learning, *momentum* or "memory" can be introduced. This method takes all or a selection of previous gradients into account when calculating the step size and direction. We call this the *velocity* (v) of the gradient,

$$v^{(n+1)} = \alpha v^{(n)} - \eta \boldsymbol{g}^{(n)},$$

where α is the momentum parameter, and

$$\boldsymbol{g}^{(n)} = \nabla_\theta C(\theta^{(n)}).$$

From this we can see that the cumulative impact behaves like an exponentially decaying moving average of the previous gradients and is dependent on the momentum parameter α. Typical values for α include 0.5, 0.9 and 0.99 [27], where larger values retain more of the previous gradients. Algorithm 5 outlines how GDW could be implemented.

Algorithm 5 Gradient Descent with Momentum (GDM)

```
1: Given learning rate $\eta$, momentum parameter $\alpha$, initial
   parameters $\theta^{(0)}$, initial velocity $v^{(0)} = 0$
2: while stopping criterion not met do
3:     Compute gradient: $\boldsymbol{g}^{(n)} \leftarrow \nabla_\theta C(\theta^{(n)})$
4:     Compute velocity update: $v^{(n+1)} \leftarrow \alpha v^{(n)} - \eta \boldsymbol{g}^{(n)}$
5:     Update parameters: $\theta^{(n+1)} \leftarrow \theta^{(n)} + v^{(n+1)}$
6: end while
```

Our own implementation follows this algorithm closely - taking the training part of the design matrix and y function as user specified inputs, while using default values for $\alpha = 0.9, \eta = 0.01$, *[number of iterations]=1000 and [θ change tolerance]*=10^{-6}. We utilize numpy's [28] built-in array functionality to for matrix and vector operations and return the optimal θ value found.

2. *Stochastic gradient decent*

The vanilla gradient descent method is sometimes referred to as full batch gradient descent because it iterates through the whole dataset. As we discussed in section II C, running gradient descent on very large datasets can be computationally costly, since we need to reevaluate the whole dataset at each iterations. A widely used alternative is stochastic gradient descent (SGD) with mini-batches, denoted as B_k, which generates randomly sampled subsets of the data to approximate the gradient, at each iteration

$$\nabla_\theta C(\theta) \approx \sum_{i \in B_k} \nabla_\theta C_i(x_i, \theta), \quad k \in \{1, 2, \dots, n/M\}.$$

The number of batches, k, is determined by the number of data data points n and the size of our batches M.

Substituting our new cost function in the vanilla algorithm yields

$$\theta^{n+1} = \theta^n - \eta \sum_{i \in B_k} \nabla_\theta C_i(x_i, \theta),$$

where k is picked at random with equal probability [29]. Each gradient requires M computations, whereas in the gradient descent which we did n computations, saving us $n - M$ calculations per iteration. This is what makes this algorithm efficient. In order to give the full exposure of the dataset we iterate over all the number of batches, also known as an *epoch*. One epoch equates to n/M steps, making one epoch insufficient for reaching the local minima, therefore we rely on iterating multiple epochs of training.

In Figure 7 we extend Figure 5 by adding the trajectory of SGD, to illustrate how the path becomes noisy due to random sampling, which is beneficial for nonlinear cost function as there is a possibility of "wiggling" out of saddle points.

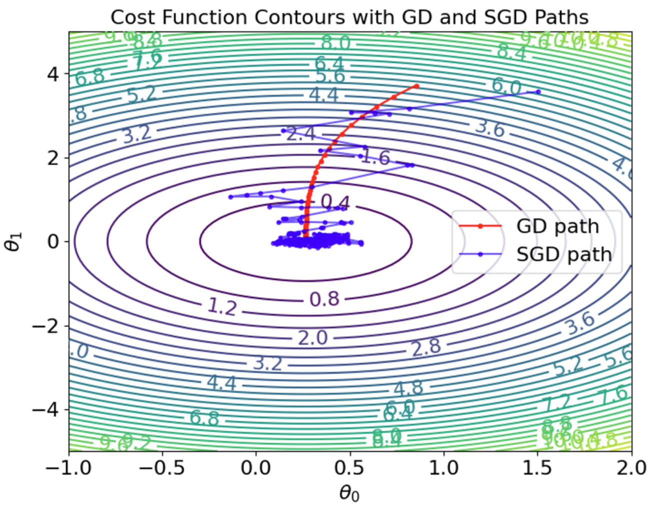


Figure 7: Contour plot of the MSE cost function with GD and SGD path from the starting point $(\theta_0, \theta_1) = (1,4)$.

We used the GD algorithm, Algorithm4, as the skeleton in our SGD implementation, outlined in Algorithm6. We used Jax to compute the derivative for scalability, so that we can parameterize and compute the cost function for future implementations.

Algorithm 6 Stochastic Gradient Descent (SGD)

```
1: Given learning rate $\eta$, number of epochs $E$, mini-batch
   size $M$, initial parameters $\theta^{(0)}$
2: for $e = 1, 2, \dots, E$ do
3:     Shuffle the training data
4:     for each $B_k$ do                          ▷ $n/M$ per epoch
5:         Compute stochastic gradient: $\sum_{i \in B_k} \nabla_\theta C_i(x_i, \theta)$
6:         Update parameters:
               $$\theta^{n+1} = \theta^n - \eta \sum_{i \in B_k} \nabla_\theta C_i(x_i, \theta),$$
7:     end for
8: end for
9: Predictions: $\hat{y} = X\hat{\theta}$
```

D. Resampling

In some cases our data might be limited to few data points or experiments that are hard or time-consuming to reproduce. In these cases *resampling* methods could be used to "generate" new datasets. Using resampling, we repeatedly draw samples from the given dataset and use these samples to obtain additional information about the performance of our model [6] [15] [30].

Here we will focus on two commonly used resampling techniques [6]: *bootstrapping* and *k-fold cross-validation*. We will first apply the bootstrap method to our OLS model to study the *bias-variance tradeoff*. We will then use k-fold cross-validation on OLS, Ridge and Lasso regression to evaluate and compare the performance across models. Both methods are described in more detail in the following subsections.

1. *Bootstrap*

Suppose we have a training dataset $Z = (z_1, z_2, \ldots, z_N)$, and create B number of datasets, each with N samples. For each of these datasets we can compute the estimator $\hat{\theta}^*$ [6] [15].

Given enough bootstrapped datasets, we can apply the central limit theorem (CLT), which states that for a sufficiently large sample size, the distribution of the sample mean approaches a normal distribution. This, in turn, allows us to make an estimation of the best fitting estimator by looking at the mean $\hat{\theta}^*$ across all bootstraps[6]. An example of bootstrapping using a simple dataset can be seen in figure 8.

Our bootstrap algorithm takes advantage of the scikit-learn *resample* function [10] to resample given datasets in a consistent way. We wrap this function into our own function where we specify the number of samples per bootstrap and number of bootstraps, returning an array of arrays containing the bootstrapped datasets.

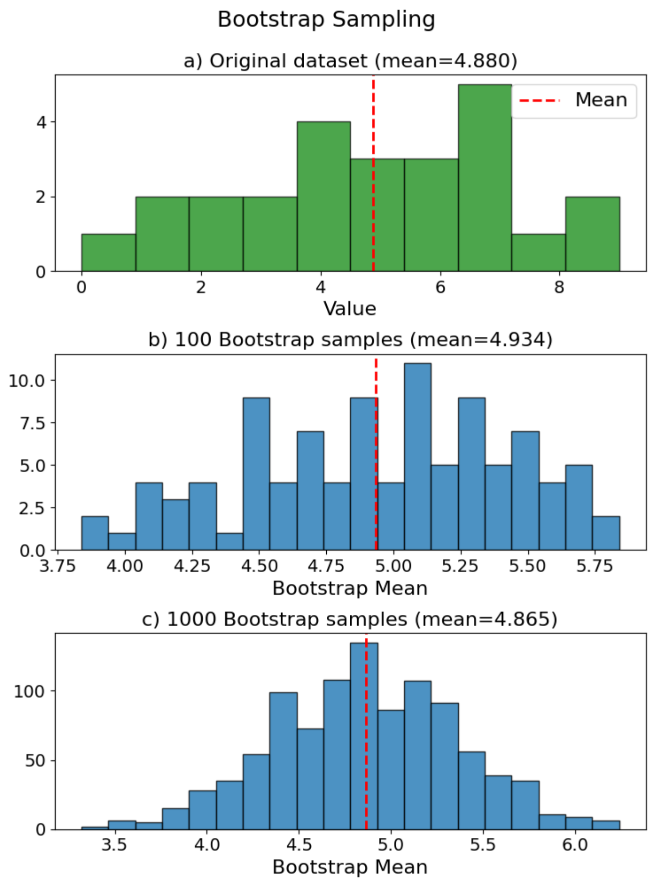


Figure 8: Histograms of a) the original values of the dataset, b) the sampled means of 100 bootstraps, and c) the sampled means of 1000 bootstraps. The original dataset consists of 25 numbers between zero and nine. Each bootstrap samples 25 times with replacement from the original dataset per bootstrap.

2. *Cross-validation*

In some cases, test data is scarce or unavailable and estimating the test MSE becomes difficult [8]. In addition to this, splitting the dataset into training and testing data is generally done randomly. This can, however, lead either subset having an unproportionate amount of outliers which could affect the estimated model [6]. To mitigate both of these challenges, k-fold cross validation splits the data into K *folds*, without replacement [14]. For each fold, we then train on all folds except k'th, before testing out model on the k'th fold, repeating for $k \in \{1, \ldots, K\}$ [30] and combine the estimated error for each step [15]. This means that we can both estimate the test MSE using the training data, as well as smoothing any potential extremes. As discussed in [15] and [14], the typical number of folds is five or ten, which corresponds to 20% and 10% of the data being used for testing respectively, with ten being the optimal number in many situations. If we choose $K = N$, where N is the total number of samples, we get what is called *leave-one-out cross-validation* (LOOCV). In this case we train on $N-1$ data points and test on one remaining [14], completely eliminating the randomness inherit to dividing the data into subsets as each sample gets its turn as the test sample.

While implementing cross-validation from scratch is relatively straight forward, we choose to leverage the existing functionality of scikit-learn [10]. In practice, this means using two functions to define the folds - `KFold(N)` and `LeaveOneOut()`, which generates indexes to split the input data into test and training data using N folds, or $N-1$ folds respectively. The split data is then run through our existing regression method functions before returning the results.

E. Use of AI tools

It is both fitting and amusing that while presenting, implementing and discussing some of the groundwork for machine learning and AI, that we ourselves use these tools to complete this project. Throughout the project period, both ChatGPT[31] and Claude AI[32] have been used. Our main use case has been debugging and altering existing Python and LaTeXcode. However, they have also been used to give suggestions on how to structure our report and as a spellchecker. In the case of Claude AI, all relevant lecture notes and Jupyter notebooks were used as *project knowledge*[33] and formed the knowledge basis for all interactions related to the project - searching this data first, before checking other sources. Links to chat logs can be found in appendix V.

While it is tempting to save time and effort by using these tools to produce the majority of content in this project, we feel like this neither enhances our learning nor necessarily leads to a good report and code. As awesome as these tools can be, they have limitations in occasional hallucinations and misunderstandings, and generally need to be verified against other sources. As such, all text and code used in this project has been produced by us, based on our own understanding of the topics, with available AI tools serving as supporting tools.

III. RESULTS AND DISCUSSION

We defined the data gathering process as

$$y = f(x) + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2),$$

where ϵ is a normally distributed noise with mean zero and variance σ^2. As shown in Equation 1 in Section II B 2, the total expected error contains an irreducible component due to this noise. We can observe the impact this noise has on our metrics, MSE and R^2, in Figure 9. As expected, noise degrades the quality of the fit, by increasing the MSE and reducing the R^2 score. Our data was generated using gaussian noise $\epsilon \sim \mathcal{N}(0,\, 1)$ to reflect real-world measurement error. Keep this in mind when interpreting our results going forward.

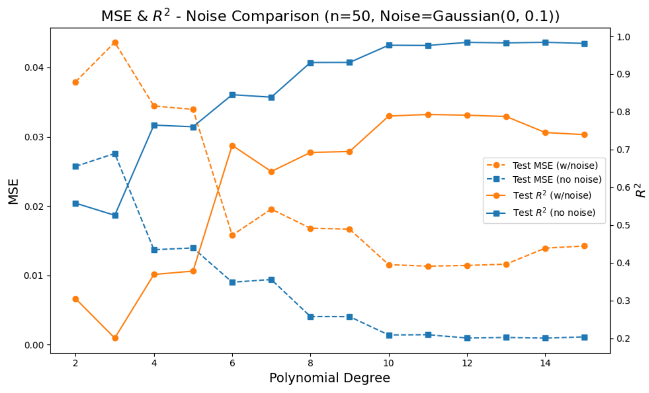


Figure 9: MSE and R^2 for Runge's function using OLS with $\epsilon = 0$ and $\epsilon = 0.1$, $x \in [-1, 1]$.

It should be noted that some of our implementations constrain all methods to using gradient descent, including OLS and Ridge which both have closed-form solutions. This does introduce unnecessary complexity for some analysis of OLS and Ridge. However, this also allows us to directly compare results across all three methods. In addition to this, SGD was only implemented for OLS and Ridge, not for Lasso, due to time-constraints. This is most notable in our discussion and analysis on cross-validation where we limited our analysis to only use vanilla GD and GDM.

We started out by approximating Runge's function using ordinary least squares regression analysis using polynomials as our predictors. In the case of polynomial interpolation, see Figure 1, we saw that the model was dependent on the number of data points, and as a consequence the polynomial degree. Similarly, we should expect that the OLS model's is influenced by these factors to a certain extent.

Through hyperparameterization we got Figure 10. From this plot, we made the following observations: With few data ($n \in [5, 15]$), simpler models are preferable, while larger sample sizes ($n \in [40, 50]$) suggest that increasing polynomial degree improves and stabilizes performance. There is also a high error region for $20 \leq n \leq 35$, indicating that $(X^T X)$ is nearly singular for higher polynomials, which leads to unstable estimators.

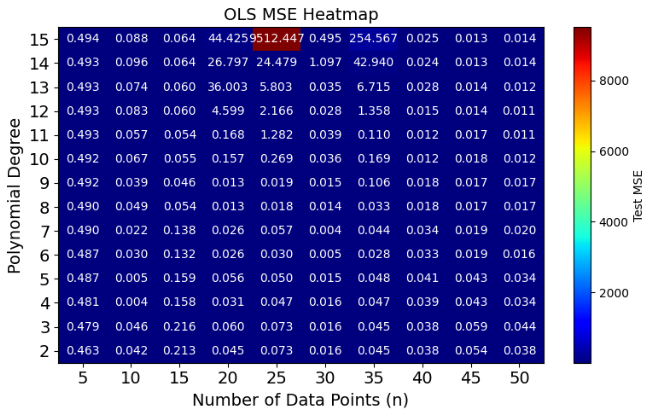


Figure 10: MSE as a function of polynomial degree and number of data points.

Based on the preliminary analysis in Figure 10, we selected the $n = 50$ column for further analysis, as it yielded stable performance. We then evaluated the model's performance using both MSE and R^2. Figure 11 shows both test MSE and test R^2, indicating quality fit for higher degree polynomials. However, beyond polynomial degree 12, we observe diminishing returns with indications of overfitting as both R^2 and MSE performs worse for polynomial degree 15. Given these observations, models with polynomial degrees between 6 and 14 are good candidates for our final model. However, the differences in performance among the candidate models are marginal, making it difficult to distinguish which polynomial degree provides the best overall fit. To gain further insight we turn to an analysis of the OLS coefficients and bias-variance tradeoff.

In Figure 12 we can see the magnitude of the coef-

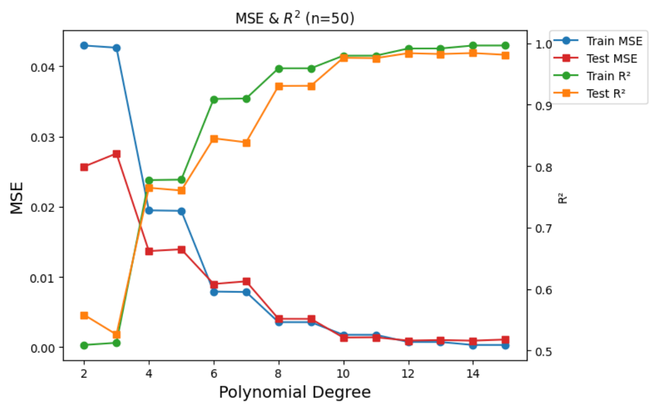


Figure 11: OLS model's performance for $n = 50$ across polynomial degrees, showing both MSE and R^2 for training and test data.

ficients growing rapidly as the degree of the polynomial increases, especially for higher order polynomials. Just as we identified regions of numerical instability in Figure 10, we can see that the coefficients $\theta_8 - \theta_{12}$ for higher order polynomials indicate the same. The bias-variance tradeoff analysis in Figure 4 suggest that polynomial degree of 8 offers the best balance between bias and variance. This observation is supported in the coefficient plot as this polynomial degree is numerically stable for all estimators.

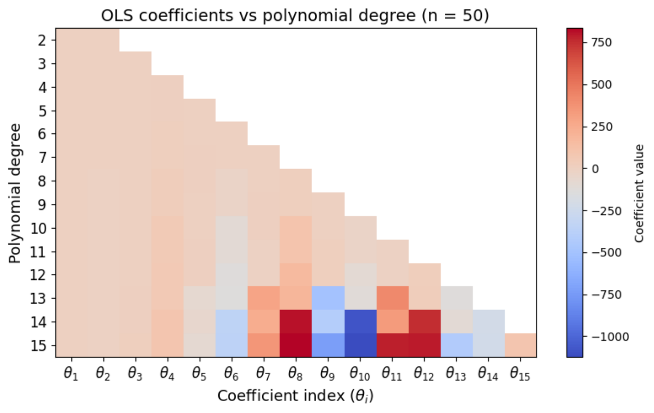


Figure 12: Magnitude of OLS coefficients as a function of polynomial degree for $n = 50$. Increasing coefficient magnitude at higher degrees indicates numerical instability and overfitting.

Numerical instability in OLS, evident in Figure 10 and Figure 12, was our motivation for implementing Ridge. To get comparable results, we used a fixed seed in our code, replicating inputs-output pairs. In Figure 13 and Figure 14 we can see the L2 penalty dampens the regions of error, but doesn't remove them as we hoped. The reason for this is that we used the regularization strength that gave us the column with the lowest MSE in Figure 2, which was $\lambda = 1e^{-5}$. With λ close to zero, meaning Ridge behaves almost identically to OLS, suggests that OLS already captures the underlying relationship well for this dataset.

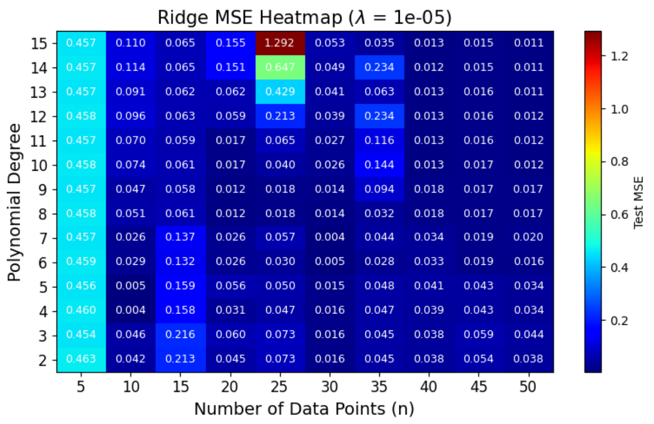


Figure 13: MSE as a function of polynomial degree and number of data points, with $\lambda = 1e^{-5}$.

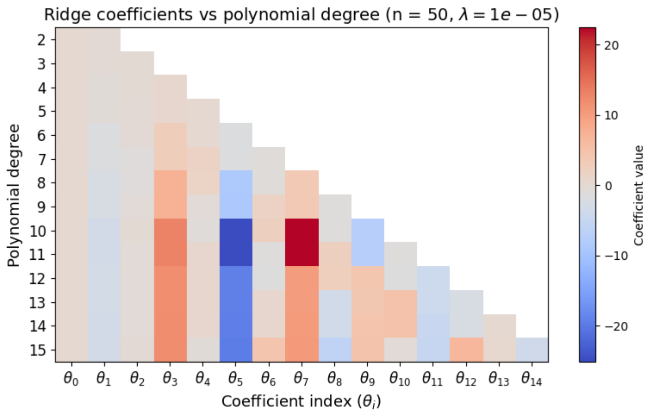


Figure 14: Magnitude of Ridge coefficients as a function of polynomial degree for $n = 50$. Increasing coefficient magnitude at higher degrees indicates numerical instability and overfitting.

By analyzing the MSE and R^2 score in Figure 15 we can see that our Ridge implementation performs worse, with OLS having $\approx 23.75\%(\frac{R^2_{OLS}}{R^2_{Ridge}} = \frac{0.99}{0.8} = 1.2375)$ better R^2 score and overall lower MSE.

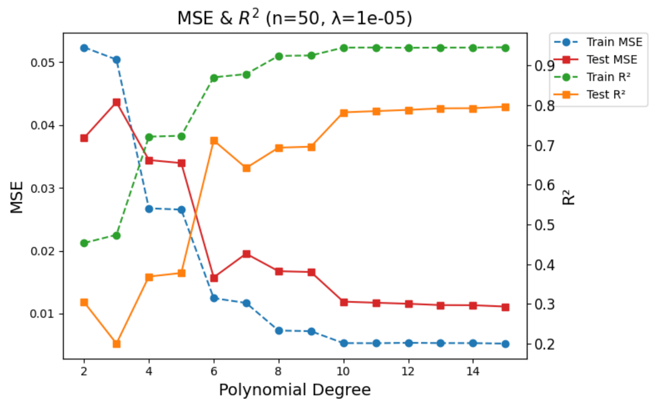


Figure 15: Ridge model's performance for $n = 50$ and $\lambda = 1e^{-5}$ across polynomial degrees, showing both MSE and R^2 for training and test data.

We also investigated the performance of OLS and Ridge by using various gradient descent methods to find our parameters $\hat{\theta}$, In Figure 16 we can see the SGD converges much quicker than it's counterparts, and as such very desirable for very large datasets. In Figure 17 and Figure 18 we can see the performance of the gradient descent methods for OLS and Ridge respectively. The gradient-based parameters yielded similar MSE ($\approx 0.03 - 0.04$) and R^2 ($\approx 0.22 - 0.43$) for OLS. For Ridge regression, the L2 penalty seems to smoothen the fluctuations, resulting in very similar MSE ($\approx$ 0.0125) and R^2 (≈ -1.25) for all methods. Negative R^2 score for Ridge indicate that it performs worse than the mean predictor.

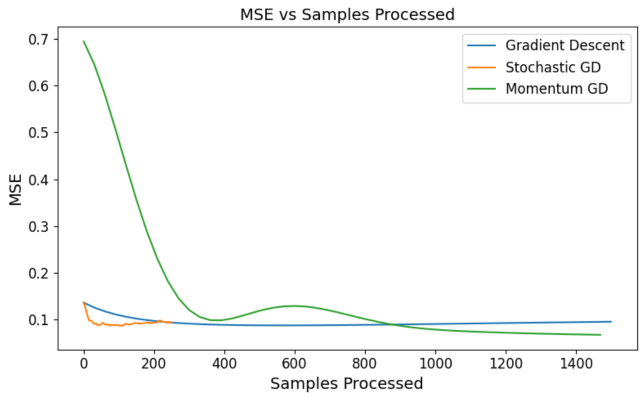


Figure 16: Stochastic Gradient descent converges faster compared to GD and GD w/momentum. n = 30, n_{iter} = 50, epochs = 50, batchsize = 5, eta = 0.05

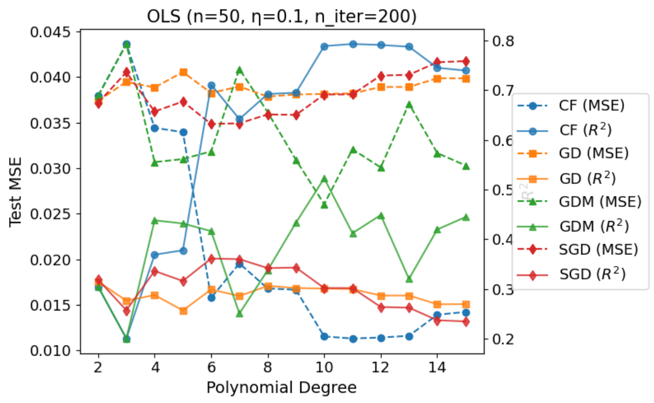


Figure 17: Test R^2 and test MSE performance for OLS using closed form, GD, GDM and SGD to compute $\hat{\theta}$.

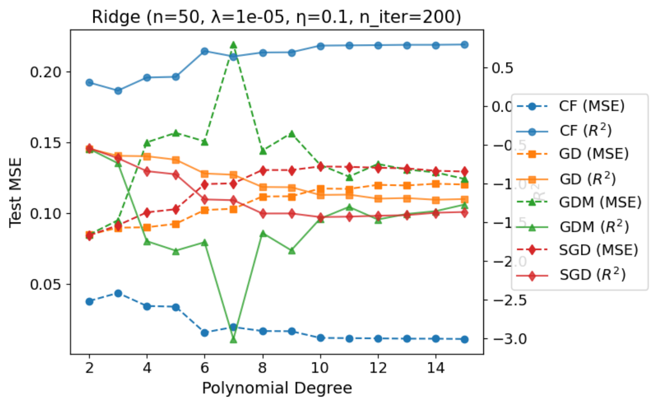


Figure 18: Test R^2 and test MSE performance for Ridge regression using closed form, GD, GDM and SGD to compute $\hat{\theta}$.

Although Ridge addresses the numerical instability, it does not perform any form of feature selection - coefficients might approach zero, but will always remain non-zero. Lasso regression, using the L1 penalty, could drive irrelevant coefficients exactly to zero, which could potentially lead to both stability and model simplification.

Figures 19 and 20 show the result of applying Lasso regression using vanilla gradient descent and GD with momentum respectively. With momentum, the test MSE stabilizes around $0.020 - 0.025$ at polynomial degree six, while without momentum there appears to be no stabilization, as values fluctuate between MSE $0.02 - 0.09$ up to degree 15. These observations are equally valid for the calculated R^2 score in each case. Strictly looking at the minimum value of the MSE, we also see that GDM manages to minimize this value at a lower polynomial degree as compared to vanilla GD.

As the last step of our analysis we apply cross-validation to all three of our methods using both vanilla GD (figure 21) and GDM (22) to compare their

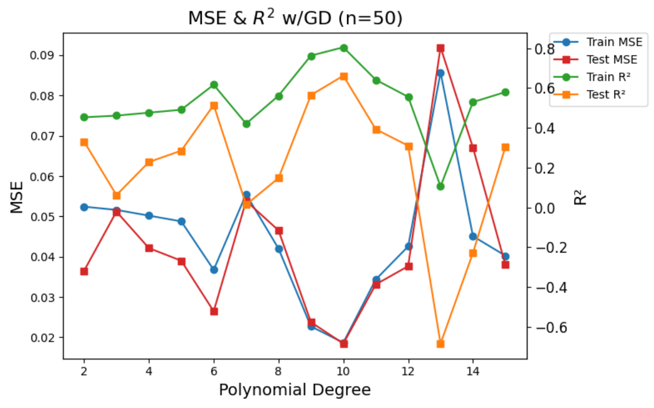


Figure 19: Plot MSE and R^2 score vs polynomial degree for Lasso regression estimate of Runge's function, using vanilla gradient descent to optimize the estimator.

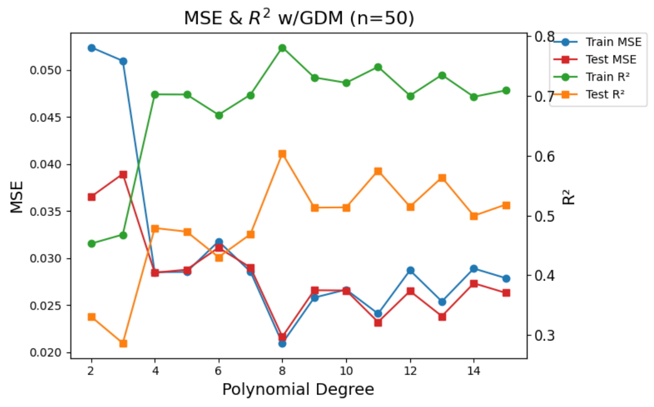


Figure 20: Plot MSE and R^2 score vs polynomial degree for Lasso regression estimate of Runge's function, using gradient descent with momentum to optimize the estimator.

performance using the MSE of their test dataset, The vanilla GD performs slightly worse than the momentum variant, and while there seems to be some oscillations when using momentum, these are within $\approx \pm 0.002$ MSE and are not significant. We can observe that across both methods and number of folds Lasso regression has the best performance with an average MSE across all folds of 0.04148 and 0.03049 for GD and GDM respectively. Ridge gives us an MSE $\approx$twice as high as Lasso using vanilla GD, with an average MSE of 0.08537, while performing slightly better, although almost identically to Lasso using GDM, 0.02993. The standard OLS approach is consistently worst, with avgMSE equal to 0.09066 and 0.04163 respectively.

There could be a variety of reasons for these results. One possible explanation for Lasso having the best overall performance might be due to a high number of irrelevant features, driving them to zero. While these features would also approach zero using Ridge, they would never equal zero. Due to time constraints, this topic was not explored further.
Given that GDM helps dampen oscillations, it is expected that this method performs better than the vanilla GD when estimating Runge's function. However, the significant jump in performance for Ridge regression is unexpected. For cross-validation we have used a fixed learning rate of $\eta = 0.1$ which might perform generally better for Lasso as compared to Ridge. If this is the case, using momentum helps to overcome this challenge, by effectively providing an adaptive learning rate, pushing Ridge to a significantly lower mean MSE across all folds.

Investigating this theory, we ran the same experiment with momentum, which we found was significantly more time efficient, using varying learning rates, $\eta \in [0.1, 0.01.0.001, 0.0001]$. Although not plotted the average MSE for OLS, Ridge and Lasso for each respective learning rate was: $[0.03322, 0.04163, 0.06050, 0.09319]$, $[0.02138, 0.02993, 0.04248, 0.05857]$ and $[0.01954, 0.03049, 0.04220, 0.05856]$
This is a topic that could be explored more in further research, however, due to time-limitations, neither this nor a comparison with stocastic gradient descent will be discuss further. Further research could also address hyperparameter tuning for all methods and sub-methods to find the optimal estimation of Runge's function using OLS, Ridge and Lasso regression.

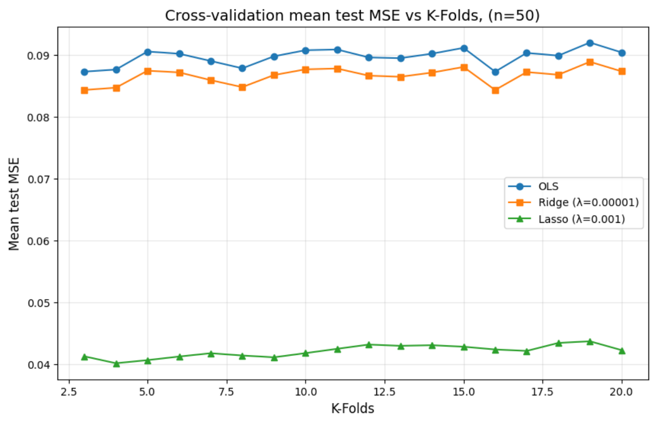


Figure 21: Comparison of test MSE between OLS, Ridge and Lasso regression using vanilla gradient descent trying to estimate Runge's function. Lambda$_{Ridge}$ = 10^{-5}, Lambda$_{Lasso}$ = 10^{-5}, $\alpha = 0.9$, polynomial degree = 8, number of folds = 3-20.

IV. CONCLUSION

While this report isn't expected to discover anything groundbreaking about how ordinary least squares, Ridge regression and Lasso regression are implemented and work, it nevertheless presents some of the important background and basics for future projects within machine learning. Throughout this project we have presented

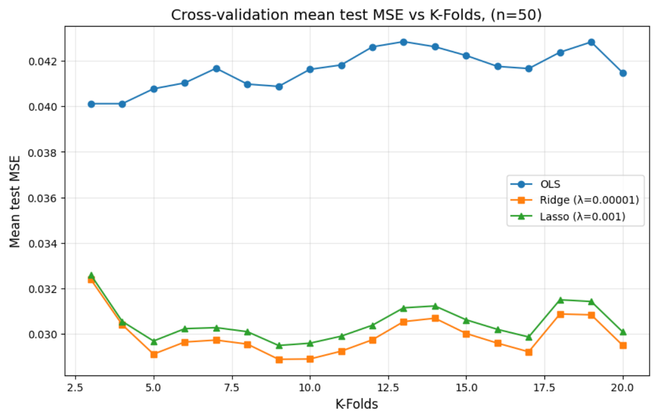


Figure 22: Comparison of test MSE between OLS, Ridge and Lasso regression using gradient descent with momentum trying to estimate Runge's function. Lambda$_{Ridge}$ = 10^{-5}, Lambda$_{Lasso}$ = 10^{-5}, $\alpha = 0.9$, polynomial degree = 8, number of folds = 3-20.

three regression models and discussed various methods for both optimizing and evaluating their respective performance, while comparing the methods to each other. This evaluation and comparison has been done while trying to create a model for Runge's function - a function known to be hard to estimate using interpolation. Our approaches successfully reduced the oscillations seen in traditional polynomial interpolation.

Through our analysis we have seen that noise has a significant impact in estimating functions. They are, however, a natural occurrence in most datasets. Regardless of this, the fit of our models are dependent on their polynomial degrees, where for OLS, a degree of 8 offers the best bias-variance tradeoff. While exploring the optimal hyperparameters for OLS, some numerical instability was observed. Implementing Ridge regression dampened these spikes, although, with $\lambda = 10^{-5}$ Ridge initially performed worse than OLS.

When optimized using cross-validation and gradient descent with momentum Ridge achieved the lowest test MSE (0.0299), with Lasso close behind (0.0305), while OLS performed worst (0.0416). Since our impementation requires all methods to use gradient descent, it should be noted that these findings might reflect convergence challenges rather than model quality. However, for both Ridge and Lasso these results seem to be dependent on the fixed learning rate, η, where Ridge performed better than Lasso at higher learning rates. Further testing is however needed to optimize the hyperparameters for all methods to provide an optimal comparison.

Finally, we would like to address the process of working on a larger project like this. The main takeaway has been gaining insight into regression methods, their optimization and variations, which enhanced our academic learning. But the project has also given us invaluable experience in co-working using tools like Overleaf and GitHub. While it has been a relatively smooth experience, communication, sharing of code and individual workload is always challenging to balance in group projects. However, we are confident that these experiences will enhance our workflow and collaboration in future projects.

[1] C. M. Bishop, *Pattern Recognition and Machine Learning* (Springer Science, Cambridge CB3 0FB, U.K., 2006), URL https://github.com/CompPhysics/MachineLearning/blob/master/doc/Textbooks/Bishop%20-%20Pattern%20Recognition%20and%20Machine.pdf.

[2] C. Yamahata, *Visualizing tycho brahe's astronomical observations of mars* (2024), URL https://observablehq.com/@christophe-yamahata/visualizing-tycho-brahe-s-astronomical-observations-mars.

[3] S. Raschka, Y. Liu, and V. Mirjalili, *Machine Learning with PyTorch and Scikit-Learn* (Packt Publishing Ltd, Birmingham, UK, 2022).

[4] J. Haugan, *Matematikk for ingeniørstudenter* (John Haugan, Røyseplassen 11, Drammen, 2019).

[5] J. Haugan, *Matematikk for ingeniørstudenter* (John Haugan, Røyseplassen 11, Drammen, 2019).

[6] M. Hjorth-Jensen, *Machine Learning Lecture Notes 2025* (Department of Physics, University of Oslo, Norway, 2025), URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter3.html.

[7] N. Vetskogen and E. Holm, *Project 1 in fysstk3155/4155 - data analysis and machine learning* (2025), URL https://github.com/EiHol/Project1_FYSSTK3155-4155/tree/main.

[8] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*, Springer Texts in Statistics (Springer, New York, 2013), ISBN 978-1-4614-7137-0, URL https://www.statlearning.com/.

[9] 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), URL https://link.springer.com/book/10.1007%2F978-0-387-84858-7.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Journal of Machine Learning Research **12**, 2825 (2011).

[11] M. Hjorth-Jensen, *Machine Learning Lecture Notes 2025* (Department of Physics, University of Oslo, Norway, 2025), URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter2.html.

[12] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics (Springer, New York, 2009), 2nd ed., ISBN 978-0-387-84857-0, corrected at the 11th printing, 2016, URL https://link.springer.com/

book/10.1007/b94608.

[13] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*, Springer Texts in Statistics (Springer, New York, 2013), ISBN 978-1-4614-7137-0, URL https://www.statlearning.com/.

[14] S. Raschka, Y. H. Liu, and V. Mirjalili, *Machine Learning with PyTorch and Scikit-Learn* (Packt Publishing Ltd, 2022).

[15] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning* (Springer, 2009).

[16] M. Hjorth-Jensen, *Machine Learning Lecture Notes 2025* (Department of Physics, University of Oslo, Norway, 2025), URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter1.html.

[17] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*, Springer Texts in Statistics (Springer, New York, 2013), ISBN 978-1-4614-7137-0, URL https://www.statlearning.com/.

[18] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*, Springer Texts in Statistics (Springer, New York, 2013), ISBN 978-1-4614-7137-0, URL https://www.statlearning.com/.

[19] S. Raschka, Y. Liu, and V. Mirjalili, *Machine Learning with PyTorch and Scikit-Learn* (Packt Publishing Ltd, Birmingham, UK, 2022).

[20] S. Raschka, Y. Liu, and V. Mirjalili, *Machine Learning with PyTorch and Scikit-Learn* (Packt Publishing Ltd, Birmingham, UK, 2022).

[21] M. Hjorth-Jensen, *Machine Learning Lecture Notes 2025* (Department of Physics, University of Oslo, Norway, 2025), URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter1.html.

[22] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*, Springer Texts in Statistics (Springer, New York, 2013), ISBN 978-1-4614-7137-0, URL https://www.statlearning.com/.

[23] M. Hjorth-Jensen, *Machine Learning Lecture Notes 2025* (Department of Physics, University of Oslo, Norway, 2025), URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week36.html.

[24] M. Hjorth-Jensen, *Machine Learning Lecture Notes 2025* (Department of Physics, University of Oslo, Norway, 2025), URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapteroptimization.html.

[25] S. Raschka, Y. Liu, and V. Mirjalili, *Machine Learning with PyTorch and Scikit-Learn* (Packt Publishing Ltd, Birmingham, UK, 2022).

[26] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, et al., *JAX: composable transformations of Python+NumPy programs* (2018), URL http://github.com/jax-ml/jax.

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

[28] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al., Nature **585**, 357–362 (2020).

[29] M. Hjorth-Jensen, *Machine Learning Lecture Notes 2025* (Department of Physics, University of Oslo, Norway, 2025), URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapteroptimization.html.

[30] K. P. Murphy, *Probabilistic Machine Learning: An introduction* (MIT Press, 2022), URL http://probml.github.io/book1.

[31] OpenAI, *Chatgpt*, https://chatgpt.com (2025), large language model.

[32] Anthropic, *Claude*, https://claude.ai (2025), large language model.

[33] Anthropic, *What are projects?*, https://support.claude.com/en/articles/9517075-what-are-projects (2024), accessed: October 5, 2025.

V. APPENDIX

Links to relevant machine learning/AI tool chats and prompts:

- https://chatgpt.com/share/68e111d0-3a38-8004-aadb-5e4dac01fed0
- https://chatgpt.com/share/68e115ef-8218-8004-b420-444fbe81c8b8
- https://chatgpt.com/share/68e20dea-af44-8004-976e-6adc12ccd1b7
- https://chatgpt.com/share/68e231bf-7394-8004-853d-d2aba131ec26
- https://chatgpt.com/share/68e23cd4-1c8c-8004-9b6f-a76346dec0dc
- https://chatgpt.com/share/68e23ea1-abc8-8004-bd49-8d1f1713cffd
- https://claude.ai/share/c77e4e01-e8b3-491e-ba60-397880794a68
- https://claude.ai/share/f2cbe97b-6bb2-402c-a259-5f782b9d4bd5
- https://claude.ai/share/c36a21a1-e0b5-4b2f-b08b-92737ee480d8
- https://claude.ai/share/5ce8545b-46b0-4f36-ae98-685a4c436ea9