Regression analysis and resampling methods
FYS-STK4155 - Project 1

Jenny Guldvog, Ingvild Olden Bjerkelund, Sverre Johansen & Kjersti Stangeland
University of Oslo
(Dated: October 6, 2025)

In the field of machine learning and statistical modeling, understanding the limitations of various regression techniques is crucial to obtain accurate predictions. The Runge function, known for its sensitivity to modeling errors, presents a significant challenge in this context. This project investigates the application of linear regression methods to the Runge function and compares their abilities in accurately predicting the function. First, the Ordinary Least Squares regression, Ridge regression, and Lasso regression are tested and discussed. Then, gradient descent methods are employed to identify if they will provide better models. To improve the statistical performance of the models, we employ two different resampling techniques, bootstrapping and cross-validation. We find that Ordinary Least Squares and Ridge regression both capture the Runge function well, with Ordinary Least Squares slightly outperforming Ridge at higher polynomial degrees. Lasso regression yields higher error due to additional bias. Gradient descent methods converged more slowly and performed worse than analytical solutions, with Adam and RMSProp giving the most reliable optimization among the adaptive schemes. The resampling methods produced consistent error estimates and confirmed that bias dominates over variance in our setup. These findings highlight how model complexity, sample size, and regularization jointly determine predictive accuracy.

I. INTRODUCTION

The field of machine learning and statistical modeling continues to evolve, and understanding the limitations of various regression techniques is crucial to making accurate predictions. A prime example of such challenges can be seen in the Runge function, expressed as:

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

which is known for its sensitivity to modeling errors [1]. High-degree polynomials used to interpolate this function often lead to oscillations near the boundaries, failing to converge to the true function [1]. Because of this, the function serves as a valuable case study for evaluating regression methods. The goal of these methods, often optimized using gradient descent, is to approximate the true function with minimal error, though each has important limitations.

Regression analysis has a long history in statistics and machine learning, beginning with the method of least squares introduced by Gauss in the early 19th century [2]. Over time, regression has become a fundamental tool for modeling relationships between variables, with applications ranging from the natural sciences to economics. Modern extensions such as Ridge regression and Lasso [3, 4] were developed to address overfitting, a problem that often arise in high-dimensional data. With the growth of machine learning, regression methods are now frequently coupled with optimization techniques such as gradient descent, which allow models to find optimal parameters without an analytical solution [5].

In this work, we investigate different linear regression techniques applied to the Runge function, specifically focusing on Ordinary Least Squares, Ridge regression, and Lasso regression. These methods have been chosen for their wide usage and distinct properties that affect their ability to handle various modeling challenges. We also employ gradient descent methods to find the best-fitting model and to discuss their implications on predictive accuracy. This will include a comparison between analytical solutions and gradient descent methods, as well as comparing different gradient descent methods.

To further enhance the statistical performance of our models, we apply two resampling techniques: bootstrapping and k-fold cross-validation. These approaches help mitigate overfitting and improve the reliability of our predictions. Our goal is to shed light on the strengths and weaknesses of each regression method in relation to the Runge function, providing insights that can inform future modeling choices in machine learning.

In Section II we describe the mathematical background of the different regression methods, and resampling techniques. The results of employing the different regression schemes and resampling methods for the Runge function are presented in V and are further discussed in Section VI. Lastly, we present a brief conclusion and recommendations for future work in Section VII. Additional results are found in the Appendix A, section VII. All code developed for this report is available via our GitHub repository. [1]

[1] https://github.com/kjes4pres/Project_1_FYSSTK

II. THEORY AND METHODOLOGY

A. Regression analysis

Linear regression is a statistical method that is used to model the relationship between a dependent variable and one or more independent variables [6]. The aim of the method is to describe the dependent variable as a linear combination of the independent variables. More explicitly, it assumes that an output y can be described in terms of a function $f(x)$, such that

$$f(x) = \beta_0 + \sum_{i=1}^{n} x_i \beta_i \tag{2}$$

where β_0 is the intercept and $\beta_1, \beta_2, \dots$ are coefficients associated with each term. The model is thus linear in the parameters [6].

There are numerous methods for estimating the coefficients β_i, and we will assess a selection of these and the way they perform in predicting the Runge function (Equation (1)).

1. *Ordinary Least Squares*

One of the simplest ways to determine the coefficients in linear regression is *Ordinary Least Squares* (OLS). The observed data, $\mathbf{y}$, is assumed to be described by some continuous function $f(\mathbf{x})$ with some stochastic noise $\boldsymbol{\epsilon} \sim N(0, \sigma^2)$ [5]:

$$\mathbf{y} = f(\mathbf{x}) + \boldsymbol{\epsilon}. \tag{3}$$

We model the data using a linear model

$$\tilde{\mathbf{y}} = \boldsymbol{X}\boldsymbol{\theta}, \tag{4}$$

where X is the design matrix, which contains the independent variables, and $\boldsymbol{\theta}$ are the model parameters.

In OLS, the optimal parameters $\hat{\boldsymbol{\theta}}$ are found by minimizing the mean squared error between the model and the observed data:

$$\hat{\boldsymbol{\theta}} = \min_{\boldsymbol{\theta} \in \mathbb{R}^p} \frac{1}{n} (\mathbf{y} - \boldsymbol{X}\boldsymbol{\theta})^T (\mathbf{y} - \boldsymbol{X}\boldsymbol{\theta}). \tag{5}$$

To minimize and find the optimal parameters, we take the derivative of (5) with respect to θ set it equal to zero, which leads to the analytical solution:

$$\hat{\boldsymbol{\theta}}_{OLS} = (\boldsymbol{X^T X})^{-1} \boldsymbol{X^T y} \tag{6}$$

Thus, the final predictive model is:

$$\tilde{\mathbf{y}} = \boldsymbol{X}\hat{\boldsymbol{\theta}}. \tag{7}$$

2. *Ridge regression*

Ridge regression is an extension of OLS that includes a regularization term which penalizes large coefficients. This can be useful to combat issues with overfitting and cases where features are non-orthogonal and highly correlated [3]. In the case of correlated variables in linear regression, coefficients can be poorly determined and result in high variance [6]. A high variance can result in fitting the training data noise, rather than the underlying trend.

Ridge regression thus try to combat this, by adding a small positive constant to the diagonal of $\boldsymbol{X^T X}$, making it nonsingular [3]. The model remains the same as in Equation (4), but the optimal parameters are now found by minimizing the cost function with an added penalization term (L2 norm):

$$\min_{\theta \in \mathbb{R}^p} \frac{1}{n} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2 + \lambda \|\boldsymbol{\theta}\|_2^2. \tag{8}$$

Where λ is the hyperparameter/regularization term that controls how strongly the coefficients are suppressed. This yields:

$$\hat{\boldsymbol{\theta}}_{\text{Ridge}} = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y}. \tag{9}$$

If $\lambda = 0$, Equation (9) is simply the solution to OLS. Hence, Ridge regression is merely OLS with an added diagonal term to the matrix $\boldsymbol{X^T X}$ [5]. It should be noted that the intercept, θ_0, is usually excluded from the penalization [6]. Moreover, as Ridge penalizes features, the features must be scaled in order for the penalization to happen fairly [6].

3. *Lasso regression*

Lasso is an acronym for Least absolute shrinkage and selection operator. Like ridge regression, its a shrinkage method that introduces a penalty term on the parameters, but unlike ridge, it forces some of the parameters to zero through an L1-norm penalty.

$$\min_{\theta \in \mathbb{R}^p} \frac{1}{n} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2 + \lambda \|\boldsymbol{\theta}\|_1 \tag{10}$$

The optimal parameters has no analytical solution as with OLS or ridge, as the function is discontinuous because of the L1-norm $\lambda\|\boldsymbol{\theta}\|_1$. Taking the derivative of the cost function we find:

$$\frac{\delta C(\boldsymbol{X}, \boldsymbol{\theta})}{\delta \boldsymbol{\theta}} = \frac{2}{n}(y - \boldsymbol{X}\boldsymbol{\theta}) + \lambda sgn(\theta) \tag{11}$$

where the last term obviously is discontinuous.

B. Evaluation metrics

The metrics used to evaluate the prediction will be the Mean Squared Error (MSE) and the R^2. MSE takes the length between the true values and the predictions and squares the distance for all point pairs. The sum of all the lengths is then divided by the number of samples, as shown in equation 12. [6]

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y}_i)^2 \tag{12}$$

R-squared is metric that evaluates how much of the variance in target data, the predicted data explains. Equation 13 shows the R^2, where the numerator in the fraction is the residual sum of squares, meaning the distance between the predicted point and the true value. The denominator is the total variance of the target dataset.

$$\mathrm{R} = 1 - \frac{\sum_{i=0}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n}(y_i - \overline{y})^2} \tag{13}$$

The MSE is a sum over all the points and therefore increase with the number of samples, while the R^2 is a fraction and the important range lies between $-1 \leq R^2 \leq 1$. [6]

C. Gradient descent

When optimizing regression models, the second derivative (or the Hessian matrix) of the loss function must be computed. An analytical form of the Hessian matrix may not be easily computed, and in such cases we can use the gradient descent methods, where we set the second derivative to be constant, namely the learning rate.

1. *Basic gradient descent*

The basic gradient descent method is an iterative optimization algorithm that is used to minimize the cost function [5]. It involves taking steps against the gradient of the function in the current location. This will eventually lead to finding the minima of the cost function, i.e. the location where the mean squared error between the model and the 'true' values are smallest. This method is sensitive to the landscape of the cost function and therefore also to the initial values of the parameters θ. Depending on ones knowledge of the cost function, whether its convex or non-convex, one can start with setting the initial theta to zero, or a random value, and as we expect a global minima (convex problem), the initial theta is set to zero. For basic gradient descent, the parameter update for each iteration is given by:

$$\theta_{t+1} = \theta_t - \eta\nabla f(\theta_t) \tag{14}$$

where $\nabla f(\theta)$ is the gradient of the objective function $f(\theta)$, and η is the learning rate. The gradient $\nabla f(\theta)$ will change for each regression method (OLS, Ridge, and Lasso), as well as the cost function (the mean squared error between the model and the 'true' values).

Using the gradient descent method, one can reduce the time and computational power needed to perform the regression. However, the performance is sensitive to the choice of the learning rate, and by choosing a too small learning rate, the convergence can take a long time. Moreover, if the learning rate is too high, it might overshoot the local minima and potentially lead to divergence. Another important factor when using the basic gradient descent is that it can get 'stuck' in a local minima, while there exist even lower global minima. However, in our case, we expect a global minima (convex problem).

2. *Momentum gradient descent*

Momentum gradient descent helps with the problem of being stuck in a local minima [7]. One can imagine a ball being dropped down a slope, with too little momentum, it will be stuck in the small bumps, while with more momentum it can escape local minima and 'fall' into the global minimum. The parameter update for each iteration is given by:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla f(\theta_t) \tag{15}$$

$$\theta_{t+1} = \theta_t - \eta v_t \tag{16}$$

where v_t represents the velocity (momentum) and β controls the contribution of past gradients. As we see from the equation, past gradients are saved and their contribution is defined by the β-term. This means that if the past gradient was steep, it will contribute to a higher 'speed' down the hill towards the minimum, and it can help shoot out of local minima and into a global minimum.

3. *AdaGrad gradient descent*

AdaGrad, or Adaptive Gradient Algorithm, adjusts the previous learning rates based on the historical accumulated gradients [8]. This ensures that parameters receiving updates less frequently get larger learning rates. This can help if some parameters contribute only for some time or if some parameters contain more information than others. The parameter update for each iteration is given by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}}\nabla f(\theta_t), \tag{17}$$

where ϵ is a small constant for numerical stability and G_t is the diagonal matrix of squared gradients accumulated up to time t:

$$G_t = G_{t-1} + \nabla f(\theta_t) \odot \nabla f(\theta_t). \tag{18}$$

Here, $\odot$ symbolizes the Hadamard product or element-wise multiplication. Information about previous gradients is accumulated in the squared gradient term, and a larger variance in the gradient term results in a smaller effective learning rate. This can help with saddle points, where there is a small gradient down the saddle, but a steeper gradient on the sides. Normal gradient descent would then follow the saddle, and could get 'stuck' in the saddle point. While the AdaGrad, would adjust the learning rate to follow down from the saddle point.

Even though AdaGrad can handle saddle points and work well if some parameters contain more information than others, it has some disadvantages. It does not have any momentum, and it can therefore also get 'stuck' in local minima, and it is thus not suited for non-convex problems, as it does not explore the cost-landscape. It also adjusts the learning rates, and as it accumulates the squared gradients, learning rate will decrease over time. This can lead to a longer iteration number before convergence, and it can decrease the learning rate too much, essentially leaving the difference between the gradients in the iterations so small that the tolerance is hit, and it stops before reaching the minimum.

4. RMSProp gradient descent

The RMSProp, or Root Mean Square Propagation, is a modification from AdaGrad to make the learning rate decay more gracefully [9]. The parameter update for each iteration is given by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla f(\theta_t). \tag{19}$$

Here ϵ is still a small constant for numerical stability. However, now G_t is:

$$G_t = \beta G_{t-1} + (1-\beta)\nabla f(\theta_t)^2. \tag{20}$$

With a β-term dictating how much previous gradients influence. This helps mitigate the diminishing learning rate discussed previously in AdaGrad, by adding the decay rate term β, it controls the influence of previous gradients. A larger β emphasizes the past gradients more, which helps avoiding the learning rates to be ineffectively small.

RMSProp have the benefit from AdaGrad to help optimize the converge through different regions, with the added effect of not letting the learning rate become too small. By letting the learning rate adjust, but not get too small, RMSProp is less sensitive to the choice of initial-learning rate, however, a too high learning rate could also here lead to divergence. We also note that both AdaGrad and RMSProp lack momentum, so they both still could get stuck in local minima.

5. Adam gradient descent

Adam, or Adaptive Moment Estimation, combines the advantages of Momentum and RMSProp [10]. The parameter update for each iteration is given by:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(\theta_t) \tag{21}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla f(\theta_t)^2. \tag{22}$$

The bias-corrected estimates for the moments are given by:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{23}$$

This helps mitigate a bias towards zero, if the initialization is set to zero, and not a random theta.

And the final update is given by:

$$\theta_{t+1} = \theta_t - \frac{\eta m_t}{\sqrt{v_t + \epsilon}}. \tag{24}$$

The first moment is the mean (m_t), and the second moment is the variance (v_t) of the gradients. As before, the β term dictate how much influence the previous gradients have on the learning rate. By using the two moments of mean and variance, Adam allows for optimizing the training trough momentum (from the mean) and an adaptive learning rate (from the variance). It essentially mixes the momentum and RMSProp gradient descent methods. This therefore also avoids too small learning rates, as well as adding momentum, so it will not get 'stuck' in local minima. By adjusting the learning rate, it adapts to changing dynamics in the cost landscape. As Adam has the best of both worlds, it is highly used in machine learning.

6. Stochastic gradient descent

Basic gradient descent methods handles the whole dataset each iteration, which can be computationally costly and induce memory errors. Stochastic gradient descent (SGD) has a great advantage, as it only handles a subset of the data per iteration, making it a more memory efficient method [5]. The concept of SGD lies in the assumption that the observed data can be written as a sum over n data points, and that the cost function and the derivative of it, consequently can also be written as a sum [5].

$$\nabla_\theta C(\theta) = \sum_{i=0}^{n-1} \nabla_\theta c_i(\boldsymbol{x_i}, \theta) \tag{25}$$

Randomness is introduced by choosing random subsets of size M from a dataset of n samples. The number of subsets is then $\frac{n}{M}$, where each subset is drawn with replacement.

$$\theta_{j+1} = \theta_j - \eta_j \sum_{i \in B_k}^{n} \nabla_\theta c_i(\boldsymbol{x_i}, \theta) \tag{26}$$

where k is picked randomly.

D. Bias-Variance tradeoff (squared loss)

In the fields of machine learning and statistics, a central concept is the bias–variance tradeoff, which characterizes the interchange between model complexity and predictive performance. A model with high bias tends to underfit the data by failing to capture relevant structures, whereas a model with high variance is prone to overfitting, as it adapts to random fluctuations in the training data (noise). With increasing complexity, bias typically decreases whereas variance increases, and their interaction often leads to a U-shaped generalization error. However, this shape is not guaranteed. For example, a limited degree range and a high number of samples can lead to a dominating bias, which will make the test error dependent on bias and be more or less continously decreasing with complexity, while variance remains relatively flat. The challenge lies in identifying an appropriate balance that allows the model to represent the underlying data-generating process while maintaining the ability to generalize to unseen observations. The expected generalization error can generally be decomposed into three components: the bias, the variance, and an irreducible error due to noise [5, 11].

To formalize this tradeoff, we consider the following standard regression framework:

$$y = f(x) + \varepsilon,\ \mathbb{E}[\varepsilon] = 0,\ Var(\varepsilon) = \sigma^2$$

where ε is assumed to be independent of the training data used to construct the predictor $\tilde{y}$. Let

$$\mu(x) = \mathbb{E}[\tilde{y}(x)]$$

denote the expected prediction at a point x. The prediction error can then be written as

$$y - \tilde{y} = f(x) + \varepsilon - \tilde{y}.$$

By adding and subtracting $\mu(x)$, this expression becomes

$$y - \tilde{y} = (f(x) - \mu(x)) + (\mu(x) - \tilde{y}) - \varepsilon.$$

Squaring and expanding yields

$$(y - \tilde{y})^2 =$$

$$(f-\mu)^2 + (\mu-\tilde{y})^2 - \varepsilon^2 + 2(f-\mu)(\mu-\tilde{y}) + 2(f-\mu)\varepsilon + 2(\mu-\tilde{y})\varepsilon.$$

Taking expectations, and using the fact that $\mathbb{E}[\tilde{y}-\mu] = 0, \mathbb{E}[\varepsilon] = 0$, and independence between ε and $\tilde{y}$, the cross terms vanish. Thus,

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

This decomposition then identifies the three distinct contributions to the expected prediction error as mentioned earlier:

- $(f - \mu)^2$, the bias,
- $\mathbb{E}[(\tilde{y} - \mu)^2]$, the variance of the predictor,
- $\mathbb{E}[\varepsilon^2]$, the irreducible error due to noise in the data.

We therefore obtain the bias-variance decomposition, which provides a theoretical explanation for the tradeoff between bias and variance introduced at the beginning of this section.

E. Resampling techniques

In supervised learning, we ideally split our data into a training set for fitting the model and a test set for evaluating generalization. When data are limited, however, a single hold out split can be unstable. Resampling models could be a solution here, by repeatedly reusing the available data to obtain more reliable error estimates, and in this case, to decompose the error into bias and variance.

1. *Bootstrapping*

From the training set we draw B new datasets by sampling with replacement (so some observations may appear multiple times and others not at all). For each bootstrap replicate we refit the model and predict on a fixed evaluation set (in this project: the test set). For the choice of B, we follow [6, § 7.11] and choose B = 200. Aggregating the B sets of predictions at each test point gives bias, variance and MSE as derived in section II D. (See also [12, Chs. 6–7] for formal derivations.)

2. *K-fold cross-validation*

With k-fold cross-validation, we divide the the training set into K equally big datasets (or folds). For each $K-1$ set, we refit the model and evaluate on the held-out fold. We then calculate the MSE, and take the average error.

By averaging the error across the K folds, we can use this to estimate the best choice for the hyper-parameters (here: degree p and λ). We then use these parameters to retrain the model on the entire training set *x_train*, and then evaluate on the previously untouched test set *x_test* [6]. (See also [13] for formal derivations.)

There is then the natural question of which number of K folds is the optimal choice. For a K = N, we obtain a *leave-one-out cross-validation*, where each model model is trained on N-1 points and validated on a single point. This gives almost unbiased estimates, but with high variance since each error is based on just one observation. Using fewer folds, e.g. $K = 5$, reduces variance because validation errors are averages over more points, at the cost of higher bias since models are trained on smaller subsets. Following [6, § 7.10] recommendation, we use K = 10 as a balance between bias and variance [6].

III. IMPLEMENTATION

A. Linear regression

To investigate the linear regression methods, we generated a synthetic dataset from the Runge function with $x \in [-1, 1]$ using NumPy [14]. To mimic real-world data, Gaussian noise with $\mu = 0, \sigma^2 = 0.1$ was added. The data was scaled with Scikit-learns ‘StandardScaler()‘ [15], using the argument ‘with_std=False‘. This choice avoided division by very small standard deviations, since our dataset already has low variance. In principle, scaling is unnecessary here because the features are dimensionless and on the same scale. However, we applied it both for training practice and as a matter of good style. In real-world applications, scaling is essential for methods such as Ridge and Lasso, which penalize large feature values. Proper scaling ensures that the regularization parameter affects all features equally rather than being dominated by outliers in the dataset.

After scaling, the dataset was split into training and test sets (with test sample size of 20%), using Scikit-learns ‘train_test_split‘ [15]. To make predictions, we constructed a design matrix. For this, we used both custom implementations using NumPy [14], and built-in Scikit-learn utilities [15]. We also employed custom implementations when finding the optimal parameters for model predictions using NumPy [14], as well as Scikit-learn functions [15].Results were stored in Pandas [16] DataFrames before visualization with Matplotlib [17].

B. Gradient descent

Methods for implementation were made by hand, both for basic and stochastic gradient descent.

C. Implementation - bootstrap

1. Split the available data once into training *x_train* and testing data *x_test* (using Scikitlearn model selection).

2. For each degree d, and b = 1, · · · , B:

   (a) Draw a bootstrap sample from the training data, x_train^b by sampling n points with replacement.

   (b) Fit OLS on x_train^b.

   (c) Predict on the fixed test *x_test* to obtain $\hat{y}^b$.

3. Aggregate across b on the test points to compute bias, variance and MSE for each p. We finally plot bias, var and MSE versus d to visualize the trade-off.

D. Implementation k-fold cross-validation

For each (method, degree d, λ):

1. Split the available data once into training *x_train* and testing data *x_test* (using Scikitlearn model selection).

2. Randomly split *x_train* into k folds (using Scikitlearn model selection).

3. For each fold j = 1, · · · , k:

   (a) Fit the model on the ols using pipeline

   (b) Evaluate the MSE on the held-out fold fold$_j$.

4. Report the mean across folds:

$$\text{CV\_MSE}(p, \lambda) \;=\; \frac{1}{k}\sum_{j=1}^{k} \text{MSE}^{(j)}.$$

We then scan $d \in \mathcal{D}$ and, for Ridge/Lasso, $\lambda \in \Lambda$, and choose

$$(p^\star, \lambda^\star) \;=\; \arg\min_{p\in\mathcal{P},\ \lambda\in\Lambda} \text{CV\_MSE}(p, \lambda).$$

IV. USE OF AI TOOLS

Artificial Intelligence (AI) tools, namely ChatGPT [18] and GPT UiO [19], has been used for parts of this project. The usage has been limited and primarily used for debugging code, formatting of figures, making descriptive documentation strings for functions, and grammatical help when writing the report. All usage have been quality checked by the authors.

V. RESULTS

A. OLS and Ridge regression

As a first approach, we studied how adding Gaussian noise to the Runge function (equation (1)) affected the model results of OLS. With $\epsilon \sim N(0, 0.1)$ and a sample size of $n = 700$, we found that the added noise worsened the MSE score of the predictions made on the test data. To quantify this effect, we computed the relative percentage difference between the test MSE with and without noise as:

$$Diff_{\%} = \frac{MSE_{test}^{clean} - MSE_{test}^{noisy}}{MSE_{test}^{clean}} * 100, \tag{27}$$

The results are summarized in Table I. For low-degree polynomials ($P = 2$–4), the increase in test error was moderate, with relative differences of about -17% to -48%. However, as the polynomial degree increased, the effect of noise became increasingly pronounced. For example, at $P = 10$, the test error was nearly six times larger in the noisy case, and for $P = 15$, the MSE increased by more than twenty-five times compared to the clean data. This indicates that higher-degree polynomial models are far more sensitive to noise. A more detailed view of how the MSE evolves with polynomial degree in the presence of noise is provided in Figure 15 (Appendix A, section VII).

P	$Diff_{\%}$
2	-17
3	-17
4	-48
5	-50
6	-130
7	-134
8	-285
9	-284
10	-595
11	-593
12	-1218
13	-1217
14	-2576
15	-2575

TABLE I. Percentage difference between the MSE of OLS with and without Gaussian noise added, $N(0, 0.1)$, for polynomial degrees $2 - 15$. The percentages have been rounded up to closest whole number.

Based on these findings, we included Gaussian noise in the following experiments to obtain results that better reflect realistic data conditions.

The best model complexity of OLS with $n = 700$ samples was found to be a polynomial of degree $p = 14$, determined by the lowest MSE of the test prediction. Moreover, we also investigated how the number of samples affect the model performance, and found that OLS with polynomial degree $p = 14$, a sample size larger than $n > 1000$ performs sufficient in terms of MSE and R^2 scores. This can be seen in figure 16 in appendix A, section VII.

A similar analysis was performed for Ridge regression. With $n = 700$ samples and regularization parameter $\lambda = 0.1$, the best model complexity in terms of MSE on the test data was polynomial degree $p = 8$ (figure 18. By performing Ridge regression with $p = 8$, $\lambda = 0.1$, and varying number of samples, the lowest MSE was found for $n = 1910$. However, the best R^2 score was found for $n = 1310$. Overall, both scores showed improvement with larger sample sizes (figure 19. Lastly, Ridge regression was performed for $n = 1500$ samples, polynomial degree $p = 8$ and with varying values of the hyperparameter λ. The best model performance in terms of MSE here was $\lambda = 10^{-10}$. However, we see in figure 20 that little change in MSE are found for $\lambda < 10^{-3}$. Moreover, MSE was found to increase with increasing value of λ. The features of the Ridge regression with varying λ are plotted in figure 21, where features become less in magnitude with increasing λ.

Figure 1 shows a comparison of OLS and Ridge regression in terms of MSE and R^2. It is obvious that the test predictions of both methods perform worse than the training predictions. For lower degree polynomials, Ridge performs better than OLS for both test and train, whereas OLS performs better for polynomial order larger than 9.

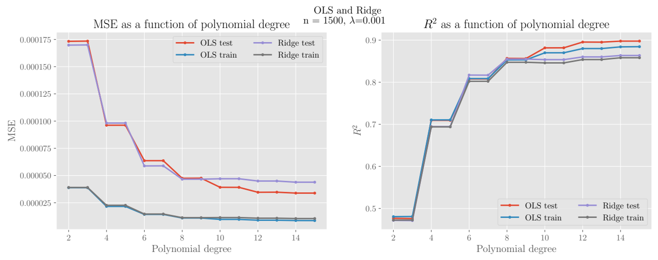


FIG. 1. The left panel shows the MSE for OLS and Ridge regression on both training and test data. The right panel shows the corresponding R^2 values.

Comparing the features of OLS and Ridge, for $n = 1500$ samples and $\lambda = 0.001$, as in figure 2, show that Ridge penalizes features strongly keeping them small and with lower variance compared to OLS.

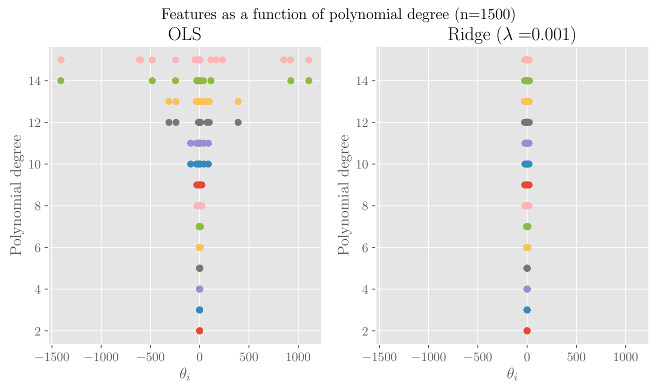


FIG. 2. The left panel shows OLS features as a function of polynomial degree. The right panel shows Ridge features as a function of polynomial degree.

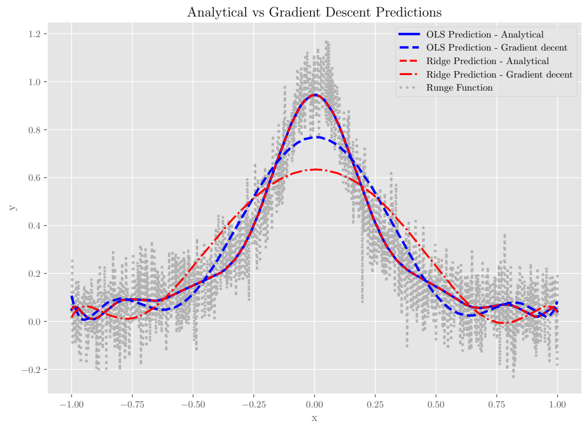


FIG. 3. Comparison between the analytical solutions to the Hessian matrix for OLS (blue) and Ridge (red), with lambda=0.001, and the gradient descent method, with learning rate = 0.1. The Runge function with noise is shown in the gray dotted line.

B. Gradient descent methods

Having analyzed the analytical solutions for OLS and Ridge, we next tested iterative optimization approaches, both to validate the analytical results and to prepare for methods like Lasso that lack closed-form solutions. In the gradient descent section, we keep some of the hyperparameters found in the first section. The number of samples will be kept at 1500, the polynomial degrees will be kept at 14 (the polynomial degree resulting in the best MSE for OLS), and lambda will be kept at 0.001, through the gradient descent experiments.

1. *Comparison between analytical and gradient descent*

Because OLS and Ridge have analytical solutions to their Hessian matrix, the initial experiment was to test how the gradient descent method performed compared to the analytical solution. From figure 3 we see that the analytical solutions also struggle with oscillations when fitting the Runge function, as is expected [1]. However, the analytical solutions perform significantly better compared to the gradient descent method. There is no significant difference between the analytical OLS and Ridge. For the gradient descent methods, the OLS gradient descent perform slightly better than the Ridge gradient descent.

We also tested a range from 0.001 to 0.1. From figure 22 we see that for the OLS gradient descent, the learning rates influences the performance of the model significantly, where the smallest learning rates did not lead to a convergence. For the Ridge gradient descent, it did not matter as much, as the convergence happened at an earlier stage, and all learning rates achieved convergence.

2. *Gradient descent and learning rate*

In addition to the OLS and Ridge regression methods, which have analytical solutions to their Hessian matrix, we also investigated the Lasso regression method. This does not have an analytical solution to compare and we therefore wanted to investigate the influence of the learning rate.

To assess the influence of the learning rate, we looked at how the learning rate affected the models by fitting 5 different models with varying learning rates for the three different regression methods. Each model iterates through 100000 updates while tracking the cost history before its applied to the test set for a final prediction evaluation.

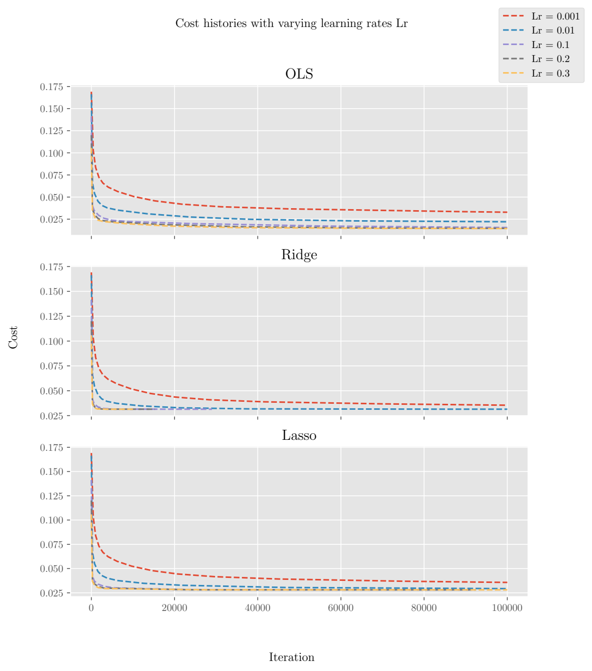


FIG. 4. How different learning rates affect the OLS, Ridge, and Lasso regression. Learning rates set in the model hyperparameters are listed in the legend.

The results in Figure 4 show that through the 100000 iterations, multiple ridge and lasso models converge, while the OLS models keeps getting better through all iterations. The MSE scores when applying the theta values to the test set is shown in Figure 5 and OLS gives the lowest MSE scores on the unseen data.

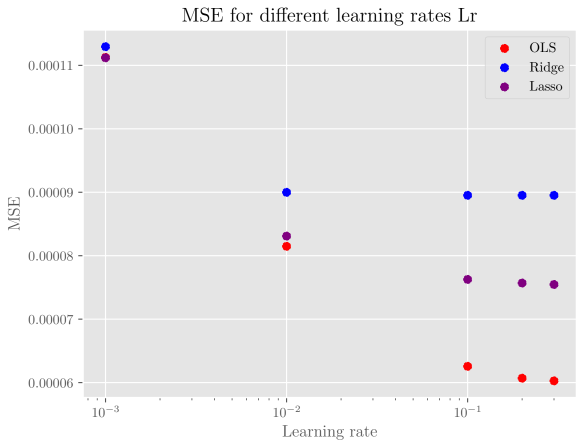


FIG. 5. The resulting MSE scores at different learning rates when applied to the test set.

3. *Gradient descent methods with updated learning rates*

To see the convergence rate of the different methods, we set the learning rate to a smaller number, 0.001, and tolerance 1e-10. We see from figure 23 that the AdaGrad method converge the slowest, and ends with the highest cost. The second worst method is the gradient descent method with no learning rate updates it starts with a higher convergence rate, but flattens out faster than AdaGrad. Momentum perform slightly better than the two, with a faster initial convergence rate however, it flattens even faster than the other two. Adam and RMSProp behaves very similar, and it is hard to find a significant difference. It looks like RMSProp decrease faster initially, however, Adam catches up, and except for ridge, it reaches convergence faster than RMSProp. Adam and RMSProp are the only two methods that reach convergence with these parameters.

If we look at a better performing learning rate and tolerance parameters ($\mathrm{lr} = 0.1, \mathrm{tol} = 1e - 15$), we see that the performance for each model in figure 24 does not compare to the convergence rate. The full cost history for these parameters can be seen in figure 23. Here we see that the RMSProp flattens out at a higher cost compared to all other methods. We also see that the predictions form RMSProp have a different feature than the rest of the models, with low x-values. This is more visual from the Ridge regression results, where all other gradient descent methods reach the same cost, and performance, while RMSProp have a different look.

We also tested the convergence for different beta values for momentum, RMSProp and Adam in figure 25. For simplicity, even though Adam have two beta, they where essentially kept the same in this test. For momentum the beta value does change the convergence rate significantly, however, for RMSProp it seems to not affect the convergence rate. For Adam, a higher beta made it reach convergence faster, however, the convergence rate did not change. It could differ, if one had tested two separate betas.

An overview of how the regression methods performed when applied to the test set is provided in Table II, where OLS performed best in most cases and Lasso outperforming OLS when using AdaGrad. Ridge regression performed generally worst.

	OLS	Ridge	Lasso
GD	0.000111	0.000113	0.000111
Momentum	0.000081	0.000090	0.000083
AdaGrad	0.000153	0.000153	0.000152
RMSProp	0.000054	0.000090	0.000075
Adam	0.000045	0.000090	0.000075

TABLE II. MSE scores when applying the learned model with different optimization methods to the test set.

4. *Stochastic gradient descent*

In this section we apply the earlier methods to a stochastic gradient descent model which iterates through

300 epochs and with a batch size of 20 samples. We start by checking how the learning rate affects the different methods when applied to a stochastic method as shown in Figure 6. We see that for the higher learning rates, the optimization methods becomes noisy, especially for RMSProp and Adam. The right side of the figure shows how the noise is negated when using Ridge or Lasso regression.

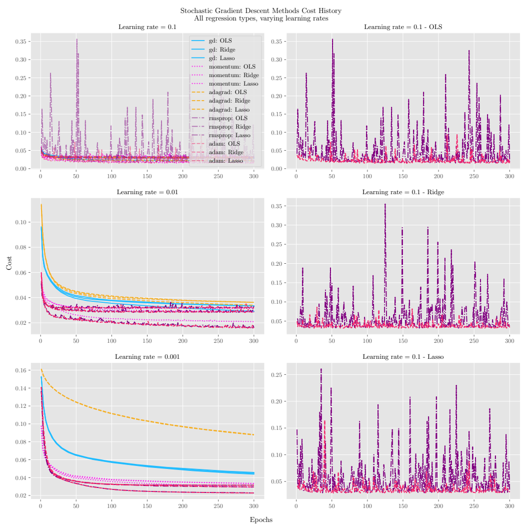


FIG. 6. Left plots) How different learning rates affect the cost history. Upper left has the highest learning rate. Right plots) RMSProp and Adam optimizer has an erratic movement, but decreases when using regularization methods.

The learning rate is set to 0.001 and the epochs are set to 1000 for the next experiments where we look at the performance of the different optimization methods with stochastic gradient descent. Figure 7 shows AdaGrad as an outlier in performance, essentially doubling the MSE of the next worse method. None of the optimization methods converged in the 1000 epoch model.

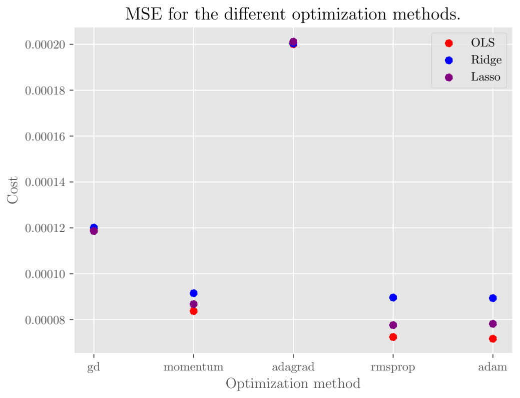


FIG. 7. MSE when applied to the test set, for the different optimization methods.

Bootstrap

In this section, we present the OLS results, focusing on the training and testing errors as well as the bias–variance analysis.

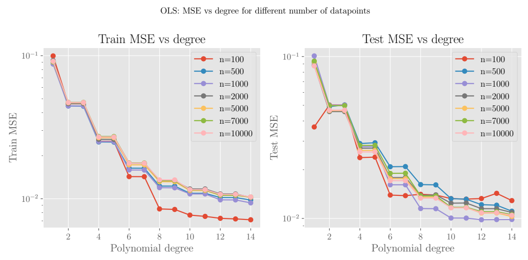


FIG. 8. OLS: MSE as a function of polynomial degree for different choices of n (training and testing data).

Figure 8 shows that both the training and test MSE monotonously decreases with increasing polynomial degree for every n. With increasing n, the level of test-MSE decreases, but the differences are small starting from $n = 500$. We are not observing the classic U-shape on the testing error in the interval of these polynomial degrees, which indicates that the bias-reduction dominates over the increasing of variance.

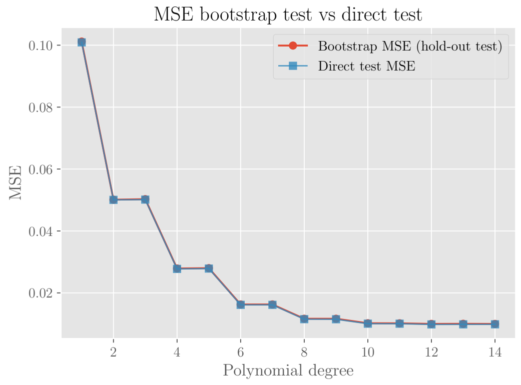


FIG. 9. OLS: Bootstrap-MSE on hold-out test vs. direct test-MSE ($n = 1000$).

As illustrated in figure 9, the bootstrap-estimates closely matches the direct test-MSE over different polynomial degrees, indicating stable estimates for generalization error.

The details for the decomposition of bias-variance is in the appendix, Fig. 26–27. We see that bias markedly drops with degree, while the variance is low and slightly increasing for high degrees of small datasets (especially $n = 100$). The MSE is therefore largely reflecting the bias in our setup, indicating a bias dominance.

K-fold cross-validation

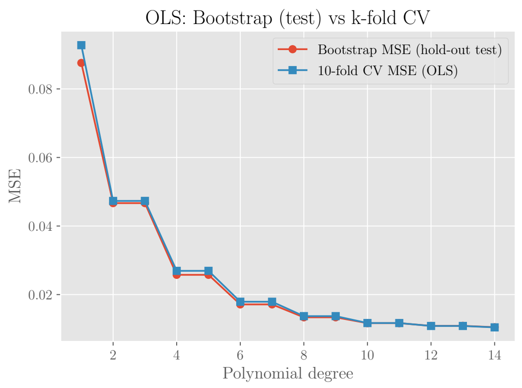


FIG. 10. OLS: Bootstrap-MSE (test) vs. 10-fold CV-MSE (training), $n = 10000$.

Figure 10 shows that bootstrap and 10-fold cross-validation (CV) produce very similar error curves as a function of degree, even though they estimate error in different ways (test resampling vs. training resampling). This supports that our conclusions are robust to the choice of resampling scheme.

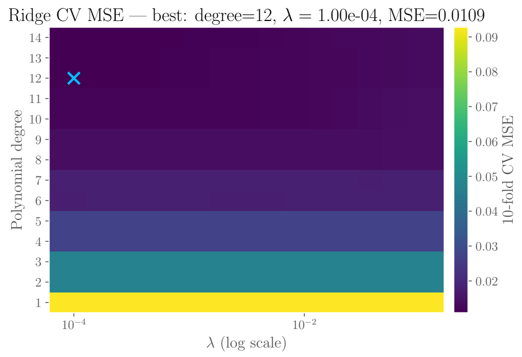


FIG. 11. Ridge: 10-fold CV-MSE as a function of degree d and λ. The cross marks the best $d = 12$, $\lambda = 10^{-4}$ (MSE = 0.0109).

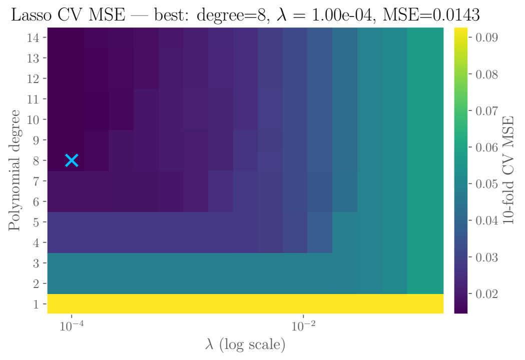


FIG. 12. Lasso: 10-fold CV-MSE as a function of polynomial degree d and λ. The cross marks the best $d = 18$, $\lambda = 10^{-4}$ (MSE = 0.0143).

Figures 11–12 show that both methods favor low regularization (small λ), but at different polynomial degrees: Ridge hits its minimum at a high degree, whereas Lasso prefers a moderate degree.

For Ridge, the CV–MSE surface is relatively flat in λ once the degree is high, indicating limited sensitivity to small changes in λ when the basis is rich (high polynomial degree). For Lasso, stronger L1 penalization introduces additional bias and effectively prunes higher-order terms, so the optimum shifts to a lower degree compared to Ridge/OLS; larger λ quickly increases the error due to excessive shrinkage. An additional view of the best lambda per degree is shown in the Appendix in figure 29. The white dots marks the 'best' λ, i.e the λ that minimizes the CV MSE for each degree. This illustration confirms the pattern seen above: Ridge consistently chooses smaller λ as the degrees increase, while Lasso always chooses a very small λ.

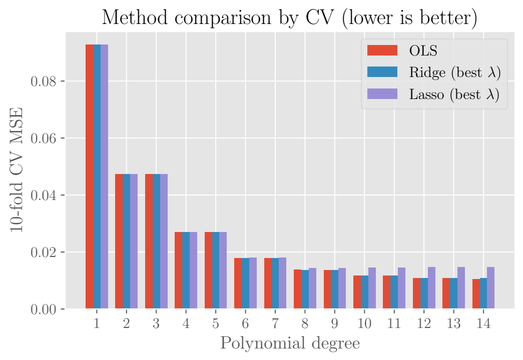


FIG. 13. 10-fold CV-MSE: OLS, Ridge (best λ) and Lasso (best λ) for every degree.

The method comparison in Figure 13 shows that the CV MSE decreases with degree for all three methods. In our setup, the curves are similar up to $d \approx 7$. At higher degrees, OLS attains the lowest CV error, with Ridge close behind, while Lasso yields somewhat higher errors—consistent with the additional bias introduced by the L1 penalty when many coefficients should be small but not exactly zero.

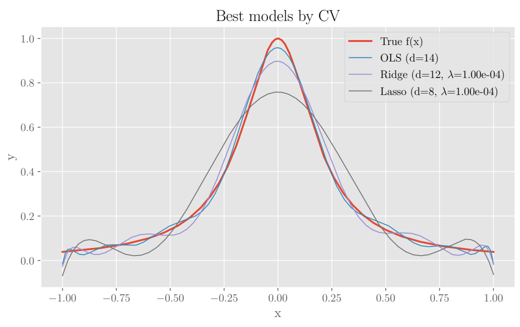


FIG. 14. Predictions from the beset models (per CV) compared with true function $f(x)$.

Figure 14 illustrates predictions from the CV-selected models compared to the true function $f(x)$. All methods capture the main shape; OLS with high degree tracks the peak most closely, while Ridge and Lasso gives somewhat higher oscillations at the end points due to regularization.

VI. DISCUSSION

1. *Analytical OLS and Ridge*

As described in section V, the increase in MSE when stochastic noise is added, suggests that higher order polynomials are more sensitive to noise. This result was expected, as higher polynomial degrees tend to overfit the noise in the training sample, while failing to generalize to the test data [6].

OLS and Ridge are fairly simple linear regression methods that allow analytical solutions. Both methods showed improved performance with increasing model complexity, as higher order polynomials can capture more features of the underlying dataset [6]. However, as observed in figure 1, Ridge performs similar to OLS for lower polynomial degrees, whereas OLS outperforms Ridge at higher polynomial degrees. This behavior can possibly be explained by the penalization in Ridge regression: for low-degree polynomials, the regularization term reduces overfitting and stabilizes the model, leading to better generalization. For high-degree polynomials, however, the penalization can suppress coefficients that would otherwise fit meaningful signal, making Ridge slightly worse than OLS.

The effect of polynomial degree on the model coefficients, as illustrated in figure 2, shows that OLS coefficients fluctuate more strongly with higher degrees, reflecting sensitivity to noise. In contrast, Ridge shrinks the coefficients, reducing the variance of the coefficients. The reduction in variance however, introduces a bias for Ridge [3].

We saw that sample size also plays a crucial role in model performance. As the number of data points increased, both OLS and Ridge showed improved MSE and R^2 scores (figure 16 and 19). With smaller sample sizes, model estimates are more prone to variability due to random fluctuations in the data, which can give a misleading impression of performance. In real-world applications, increasing n is usually not a choice one has, as datasets come with a fixed number of data points. Nonetheless, these results highlight the importance of sufficient data when fitting complex models.

2. *Comparing analytical solutions to gradient descent*

We see from figure 3, that both the OLS and Ridge regression methods perform better with analytical solutions to the Hessian matrix, compared to the gradient descent methods. This is expected, as the gradient descent methods is an approximation. We also see that OLS performs better than Ridge, even when the lambda parameter is low. It could seem that the lambda parameter stops the optimization, hence leaves OLS achieving a lower MSE and cost. From figure 22 we observed that higher learning rates did not affect the convergence of Ridge, and it seems that it is more sensitive to the choice of the lambda parameter. For OLS on the other hand, a higher learning rate continued to improve the performance.

3. *Gradient descent methods with adaptive learning rates*

To visualize the convergence rate of each method, the learning rate was set to 0.001 (figure 23). In this case AdaGrad converged the slowest, and ended with the highest cost, performing even worse than gradient descent without adaptive learning rates. AdaGrad accumulates the squared gradients, and over time the learning rate will decrease. This slows down the convergence, and in our case, as we only have one global minima, it performs the worst. For more complex cost landscapes, AdaGrad could help with the problems of getting stuck in local minima. Momentum performed slightly better than basic gradient descent, this is also expected, as it will remember the past gradients, and speed up or essentially increase the learning rate for steeper gradients. Both Adam and RMSProp showed the best results, however RMSProp converged faster than Adam. Nevertheless, Adam would reach convergence faster, as RMSProp flattened out before Adam. Compared to AdaGrad, RMSProp controls the influence of previous gradients, and it therefore does not have the same problem with a decreasing learning rate. Adam combines RMSProp with momentum, which could explain why RMSProp flattens out earlier.

In the same figure (23) we observe that for a higher learning rate, the difference between the methods initially is not as big. However, RMSProp does not reach as low cost compared to the other methods. We also see for Ridge that it converges at a higher cost, leaving AdaGrad to essentially outperform RMSProp. In figure 24, we see that this leads to RMSProp displaying the worst model prediction for OLS. For Ridge, we observe the same result, where RSMprop have a distinct feature for low x-values, where the maximum predicted value lies. For Ridge, we see the same results as with different learning rates, that all models, except RMSProp, performs equally good, which could imply that it is not as sensitive to the gradient descent method, as the lambda parameter. For OLS the results are a bit different. We see that different gradient descent methods does achieve different results. Adam achieves the best results, and if we do not include RMSProp, basic gradient descent displays the worst result. Momentum and AdaGrad have almost identical results.

In figure 25, we saw that the choice of beta value only affected the momentum gradient descent, and it will lead to bigger momentum, and a faster convergence. For RMSProp the beta value controls the influence of previous gradients, and this seems to not be as important in our case. For Adam on the other hand, it does have two beta values, and as we treated them as one, it could show a different results. However, since Adam combines momentum and RMSProp, it seems likely that the beta controlling the variance of the gradients is not as affected by the choice of beta, as in the RMSProp, while the beta controlling the momentum, would lead to a faster convergence, similar to the momentum method.

Table II showed us that OLS generally performed the best and Ridge generally worst, for most of the optimization methods. Lasso performed better than Ridge in all cases. This could be a result of the penalties induced by the regularization terms. The L1-term shrinks the bigger parameters less and the smaller parameters more, even to zero in some cases. So in a case like ours where we have 15 parameters, Lasso regression simplifies the model, making for a smoother fit, while Ridge only dampens the parameters, resulting in a worse generalization.

4. *Stochastic gradient descent vs. basic gradient descent*

The turbulent iterations we see from the stochastic gradient descent models with higher learning rates is expected as we draw gradients from small batches that might differ heavily from the statistics of the whole dataset, and each other. When using Ridge and Lasso regression, the turbulence is reduced as the gradient are dependent on the parameters, which are penalized through the regularization term and shrinking the gradient towards zero. Convergence did not occur with any of the optimization methods, even though the graphs hover around the same MSE for quite a while. This could be a result of a combination between the learning rate being too high and the tolerance of the break method being too low. It could also simply be that it needs more epochs to converge.
Comparing the MSE of stochastic versus the non-stochastic gradient descent methods, we can see that the overall scores favor the non-stochastic models. From figure 5 and 7 we see that the best models from stochastic gradient descent showing a score of around $8{*}10^{-5}$ (SGD: OLS) and $6 * 10^{-5}$ (GD: OLS) from the basic gradient descent models, after 100000 iterations and 300 epochs respectively, when applied to the test set. 300 epochs equals 22500 iterations when using batch size of 20, as well as only using a fraction of the samples per iteration. This really showcases the power of the SGD to make less precise predictions at a fraction of the memory and time, an important discussion of which to favor when applied to huge datasets. Since the stochastic gradient descent model is meant to also partially solve the basic gradient descent's problem of local minima, applying it to the convex function we have in our case does not showcase the full power of the SGD.

5. *Bootstrap and k-fold cross-validation*

We find that the testing error decreases monotonously with degree (Fig 8), and we observe no U-shape in this interval. This indicates that the bias-reduction dominates. Bootstrap and 10-fold CV gives very similar error curves (Fig 9), which strengths the conclusions independent of resampling choices. CV places the best points in the area with moderate to high degrees and low λ (Fig 11 and 12). Thus, the regularization effect is relatively weak, and OLS is the best choice at the highest degrees, with Ridge close behind.

The bias-variance-decomposition (appendix Fig 26-28) shows that the bias is quickly decreasing with degree, while the variance stays low (especially for $n \geq 500$). The sum (MSE) therefore decreases for the entire degree interval, explaining the absence of a U-curve. A clear U-shape could probably occur in a more demanding setting with higher degree area (i.e 30-40), even smaller choice of n or more noise. With these criteria the variance would presumably increase quicker and give more to regularization, leading to the classical overfitting and the U-shaped curve.

We standardize after the polynomial extension, so that the L1/L2-penalizing work similar on every column (fair penalizing). Ridge has its minima at a high degree and small λ: L2 shrinks the coefficients uniformly and exploits a rich basis without exploding the variance. Lasso has its minima at a moderat degree and small λ: L1 zeroes out some coefficients, which increases bias and reduces the effective degree.

VII. CONCLUSION

The analysis of OLS and ridge regression have shown that the model complexity, number of samples and choice of regularization parameter highly influence model performance. Higher-degree polynomials can capture more complex patterns but are very sensitive to noise, especially with smaller datasets. Ridge regression helps reduce overfitting by shrinking the model's features, making it more stable for simpler models. However, for more complex models, this regularization can hold back the model from fully capturing important patterns, allowing OLS to perform better in those cases. We also saw that having more data improves performance for both OLS and Ridge and helps the models generalize better. These findings show the importance of carefully choosing the right model complexity and regularization while considering the amount of available data, to build models that perform well on new, unseen data.

Analytical solution for OLS and Ridge outperformed the gradient descent methods, and for the different gradient descent methods Adam performed the best, while RMSProp performed the worst. In general there was no significant difference in the convergence behavior of the different gradient descent methods for different regression methods, however, it seems that OLS is more sensitive to the choice of learning rate and gradient descent method. The choice of beta seemed to influence the momentum, with a higher beta leading to faster convergence, while not affecting the variance of the gradients significantly.

In our set-up, we clearly see that OLS is marginally better at a higher degree; Ridge follows closely behind, while Lasso gives somewhat higher error due to extra bias. Our results are robust for resampling (bootstrap $\sim$ CV). In more demanding settings we would expect the regularization to be better than OLS due to higher variance.

While this study provides useful insights using the Runge functions and various regression methods, its findings are limited as we used synthetic data with simple Gaussian noise. Synthetic data used here may work well under controlled experiments like these. We do however not expect these results to necessarily transfer to other applications where the data could be more noisy. Future work should address these limitations by applying the methods to real-world datasets, to explore more realistic noise and outlier handling.

[1] Wikipedia contributors. Runge's phenomenon, August 2025. Page Version ID: 1303928185.

[2] H. A. David and A. W. F. Edwards. *Annotated Readings in the History of Statistics*. Springer Series in Statistics. Springer New York, New York, NY, 2001.

[3] Arthur E. Hoerl and Robert W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, February 1970.

[4] Robert Tibshirani. Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288, January 1996.

[5] Morten Hjorth-Jensen. *Computational Physics Lecture Notes 2015*. Department of Physics, University of Oslo, Norway, 2015.

[6] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York, New York, NY, 2009.

[7] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, January 1999.

[8] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, July 2011. Publisher: JMLR.org.

[9] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Neural Networks for Machine Learning - Lecture 6a Overview of mini-batch gradient descent*. University of Toronto.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014. Version Number: 9.

[11] Wikipedia contributors. Bias–variance tradeoff, September 2025. Page Version ID: 1311585014.

[12] Bradley Efron and Robert John Tibshirani. *An introduction to the bootstrap*. Number 57 in Monographs on statistics and applied probability. Chapman & Hall, New York, 1993.

[13] M. Stone. Cross-Validatory Choice and Assessment of Statistical Predictions. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 36(2):111–133, January 1974.

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

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

[16] Wes McKinney. Data Structures for Statistical Computing in Python. pages 56–61, Austin, Texas, 2010.

[17] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[18] OpenAI. ChatGPT, 2025.
[19] University of Oslo. GPT UiO, 2025.

APPENDIX A

A. OLS and Ridge regression

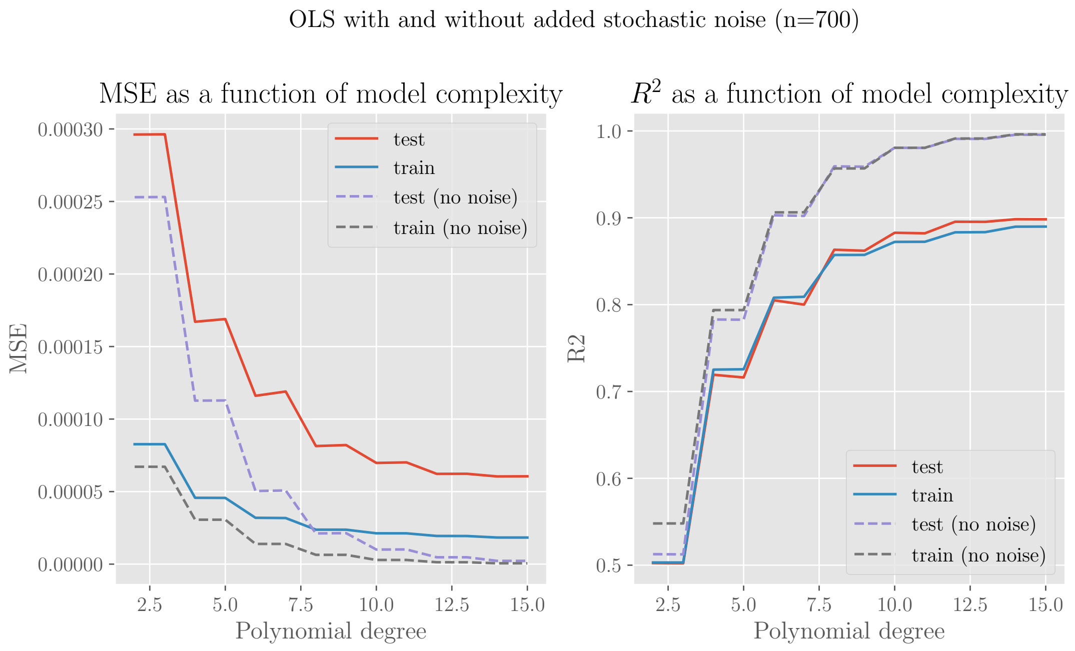


FIG. 15. OLS for the Runge function assessed with MSE and R^2 scores as a function of polynomial degree, on both the training and test data sets. Stipled lines represent the function without added stochastic noise, full lines include stochastic noise. Total number of samples is 700.

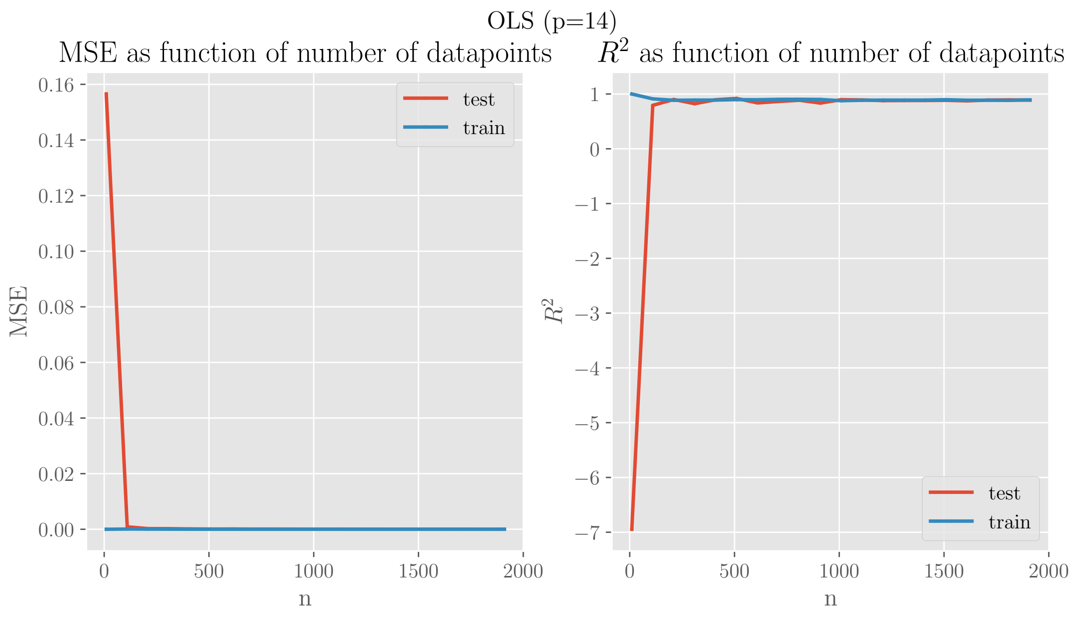


FIG. 16. MSE and R^2 as a function of number of samples for training and test data sets of OLS on the Runge function with polynomial degree 14.

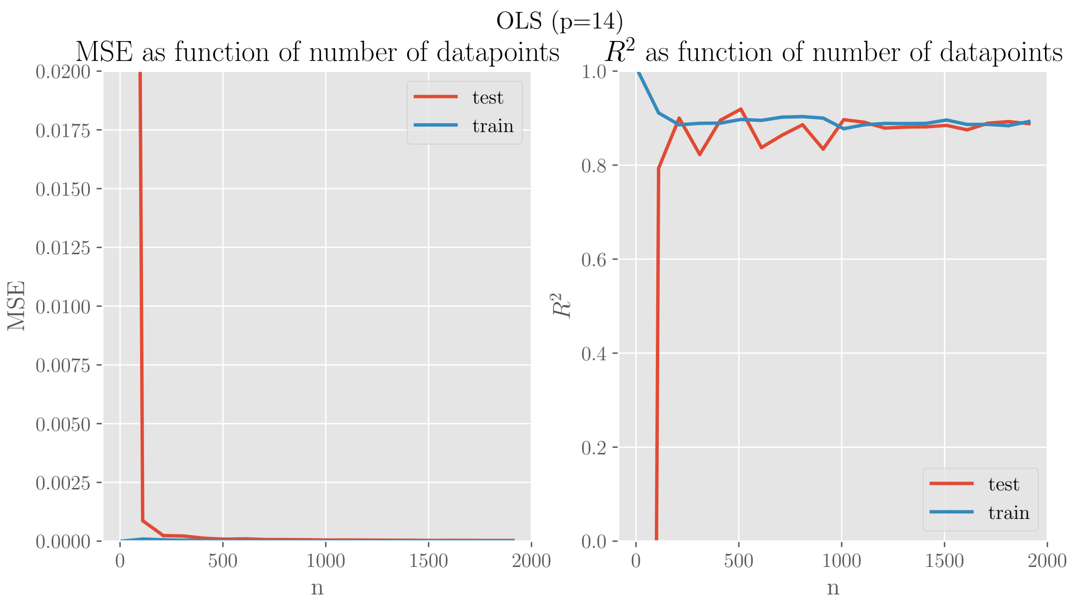


FIG. 17. This figure shows the same as figure 16, but here we have applied $(0, 0.02)$ as limits to the y-axes.

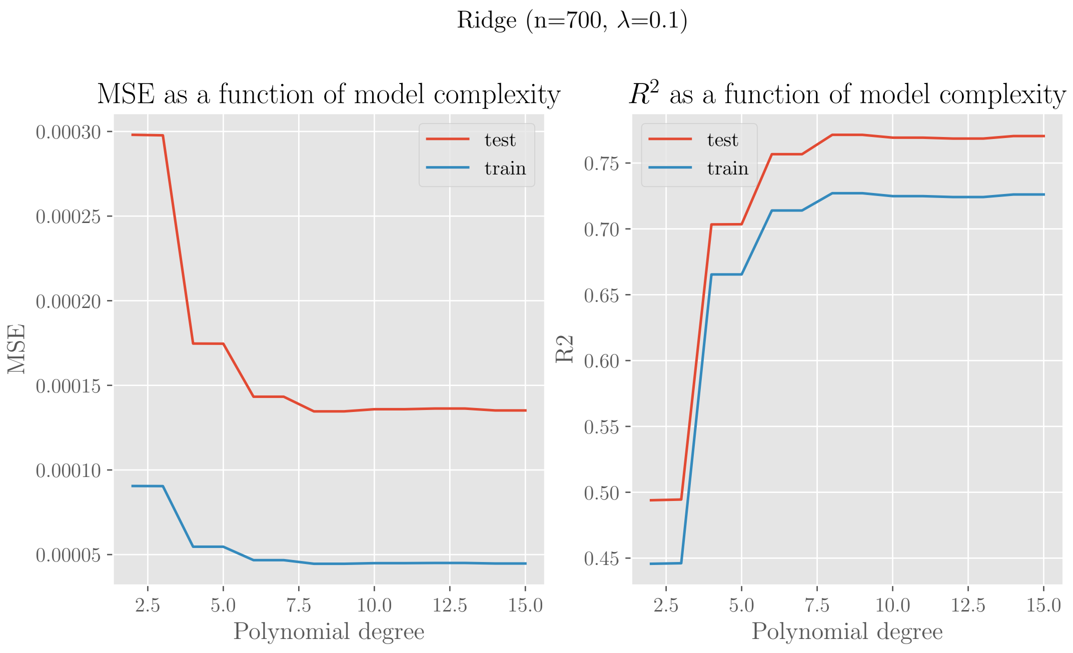


FIG. 18. MSE and R^2 as function of polynomial degree for Ridge regression applied on the Runge function with 700 samples and $\lambda = 0.1$.

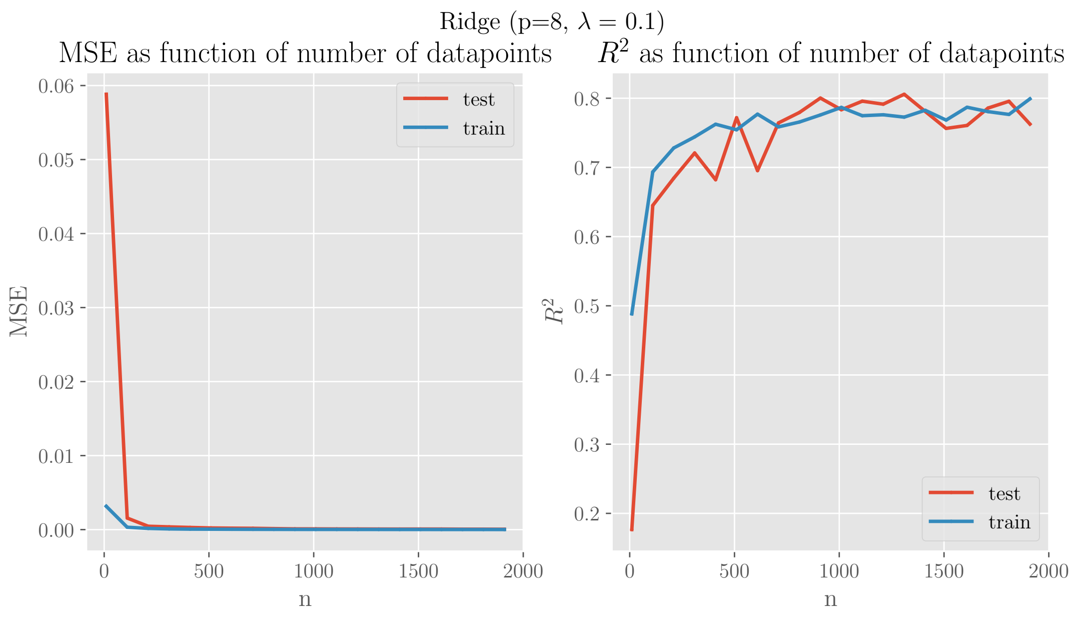


FIG. 19. MSE and R^2 as function of number of samples for Ridge regression applied on the Runge function with polynomial degree 8 and $\lambda = 0.1$.

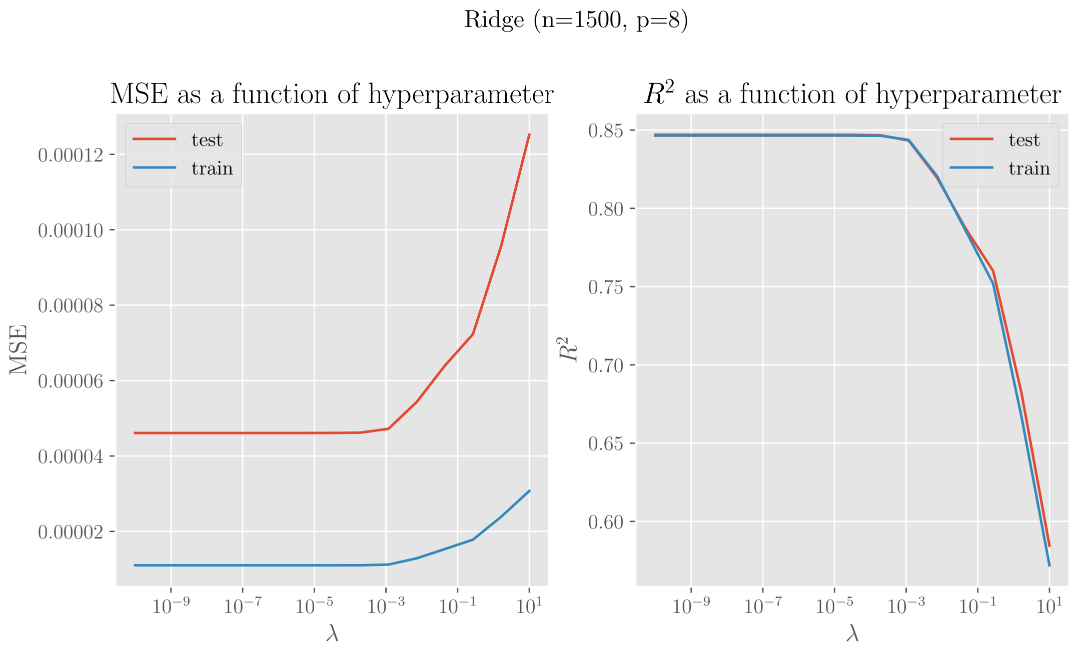


FIG. 20. MSE and R^2 as function of hyperparameter, λ, for Ridge regression applied on the Runge function with polynomial degree 8 and 1500 samples.

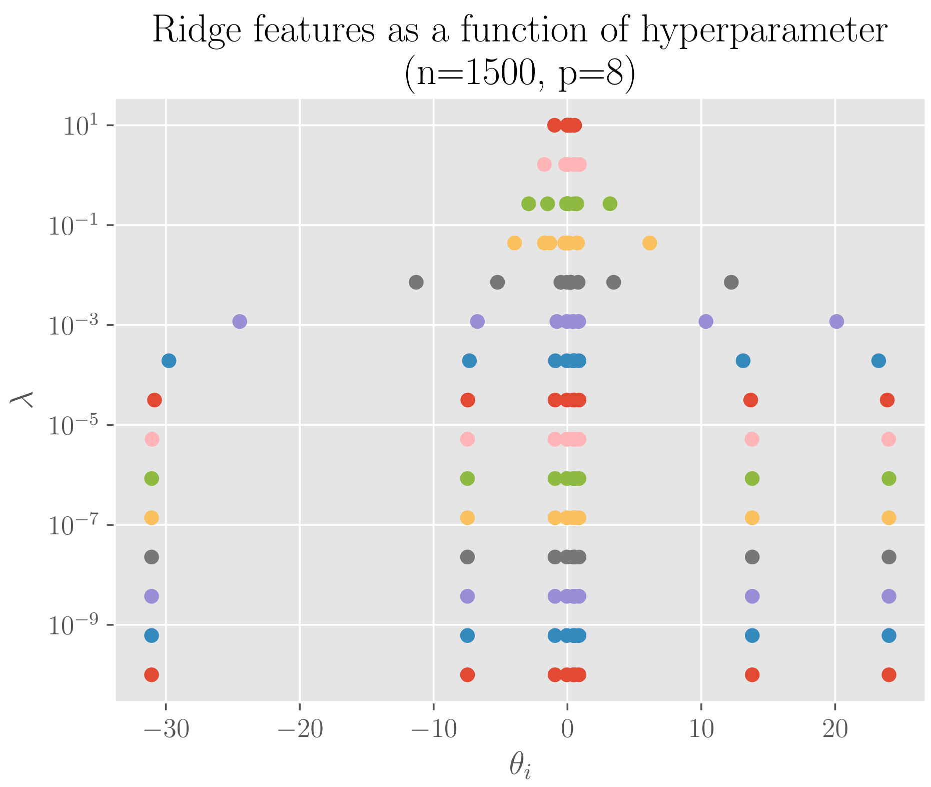
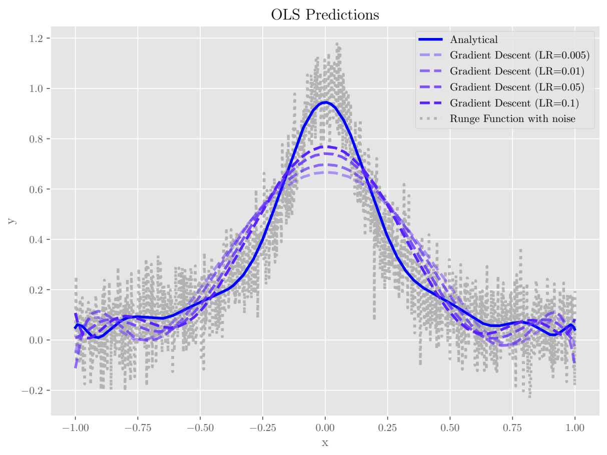
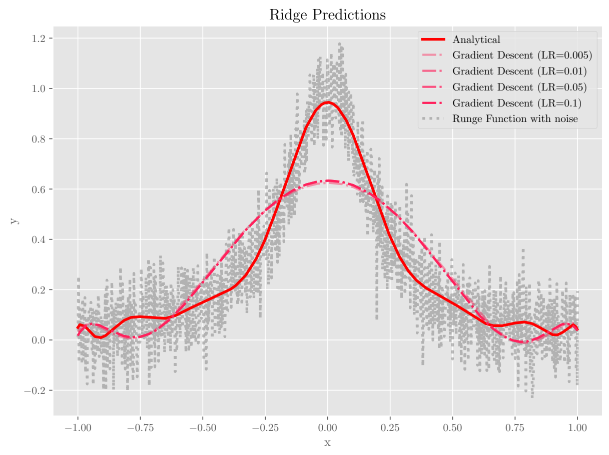


FIG. 21. Ridge features as a function of hyperparameter, λ. Ridge regression was performed on the Runge function with 1500 samples and polynomial degree 8.

B. Gradient descent with different learning rates for OLS and Ridge



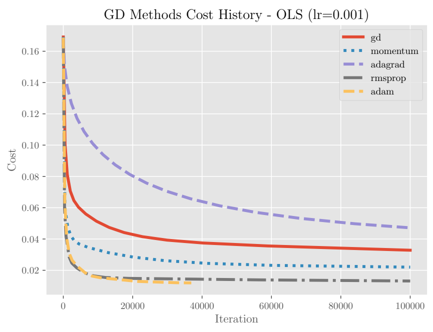
(a) Comparison between the analytical solutions to the Hessian matrix for OLS, and the gradient descent method, with different learning rates. The Runge function with noise is shown in the gray dotted line.



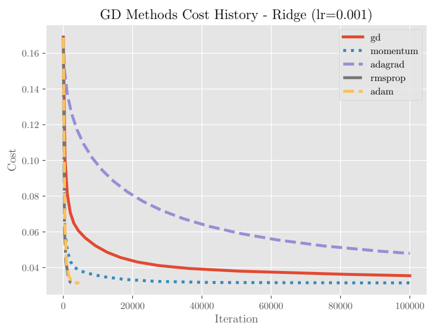
(b) Comparison between the analytical solutions to the Hessian matrix for Ridge, with lambda=0.001, and the gradient descent method, with different learning rates. The Runge function with noise is shown in the gray dotted line.

FIG. 22. Comparison of Gradient Descent vs Analytical Solutions for OLS and Ridge

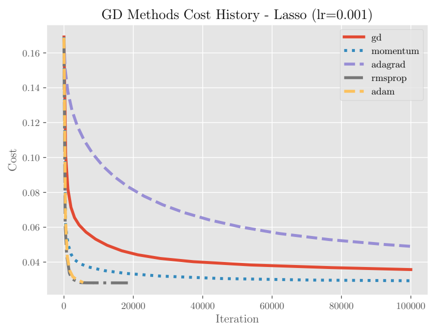
C. Cost history for different Gradient descent methods



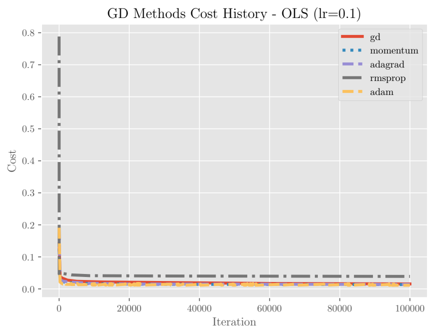
(a) Cost history for OLS with learning rate = 0.001.



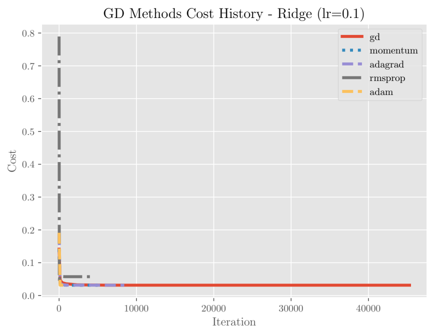
(b) Cost history for Ridge, with lambda=0.001 with learning rate = 0.001.



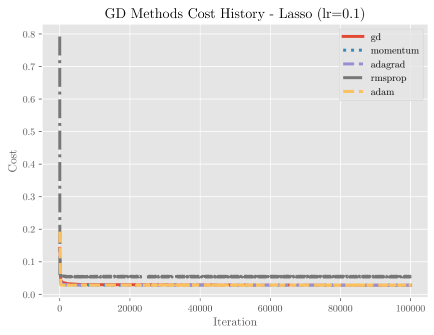
(c) Cost history for Lasso, with lambda=0.001 with learning rate = 0.001



(d) Cost history for OLS with learning rate = 0.1.



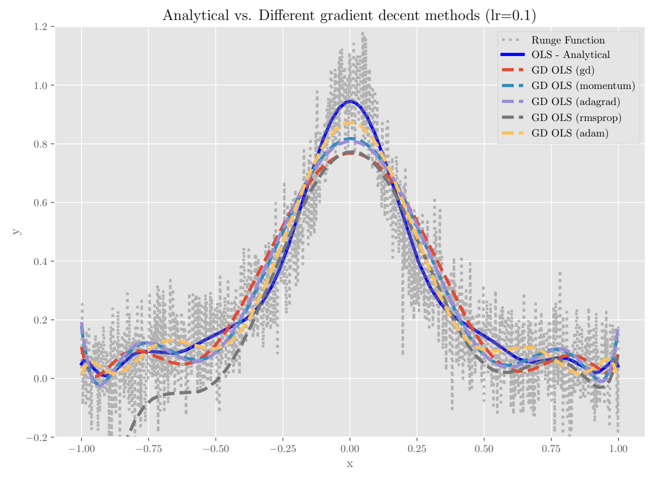
(e) Cost history for Ridge, with lambda=0.1 with learning rate = 0.001.



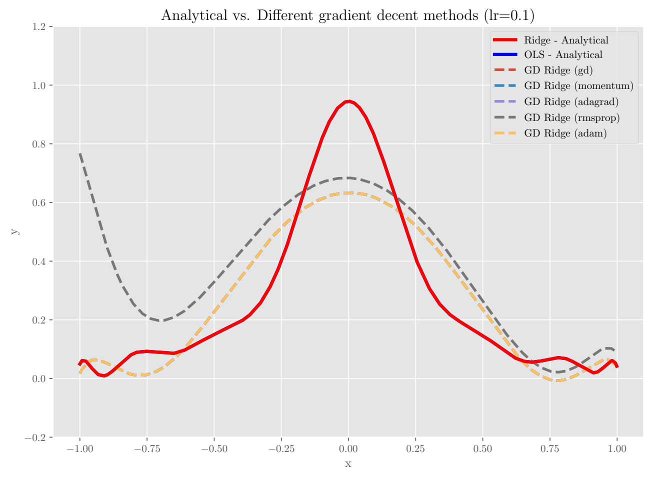
(f) Cost history for Lasso, with lambda=0.1 with learning rate = 0.001

FIG. 23. Comparison of cost history for different Gradient descent methods.

D. Model performance for different Gradient descent methods



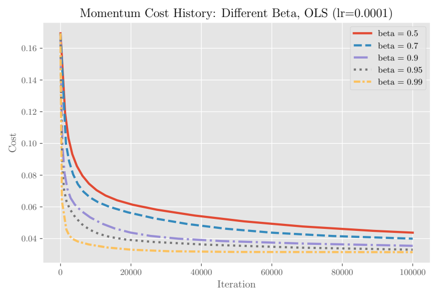
(a) Model predictions for OLS with learning rate = 0.1.



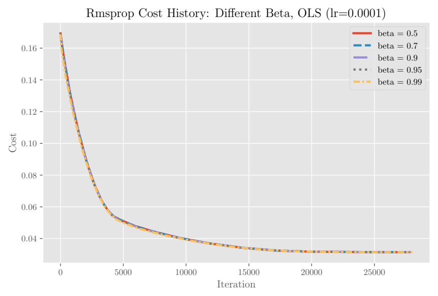
(b) Model predictions for Ridge, with lambda=0.001 with learning rate = 0.1.

FIG. 24. Comparison of model performance for different Gradient descent methods vs. analytical solutions.

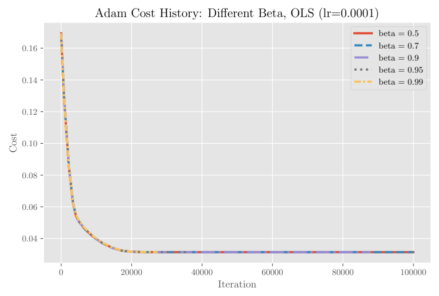
E. Model performance for different beta values



(a) Cost history for momentum gradient descent, with different beta values.



(b) Cost history for RMSProp gradient descent, with different beta values.



(c) Cost history for Adam gradient descent, with different beta values.

FIG. 25. Comparison of cost history for different beta values.

F. Bootstrap - MSE, Bias and Variance

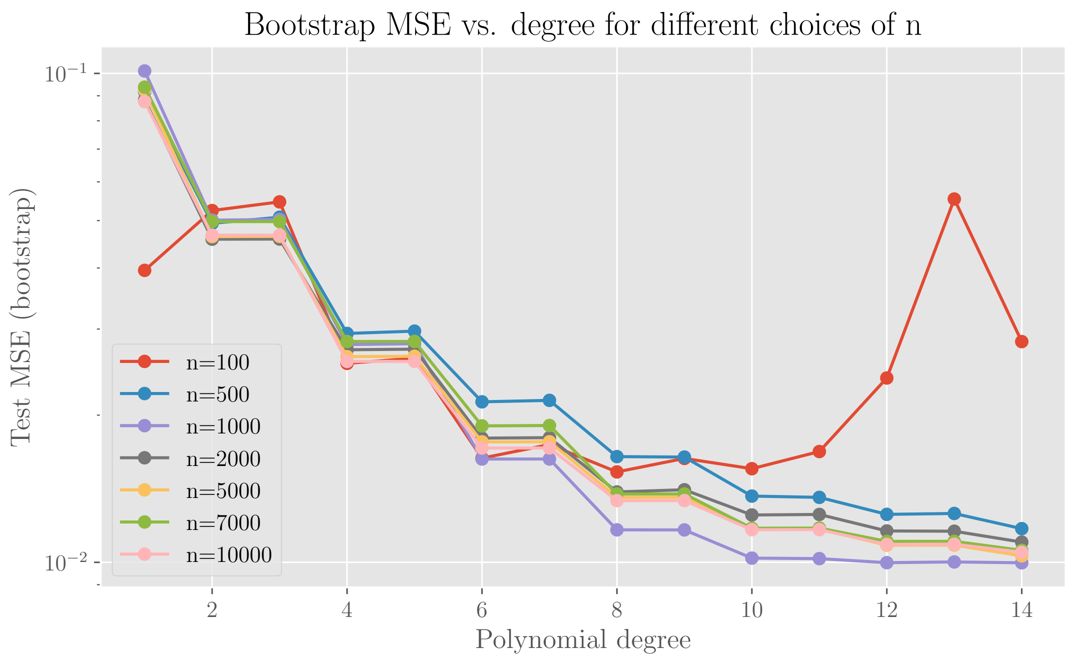


FIG. 26. Bootstrap-estimated test-MSE (OLS) as a function of polynomial degree for different choices of n. Every curve is a mean of B bootstrap-draws.

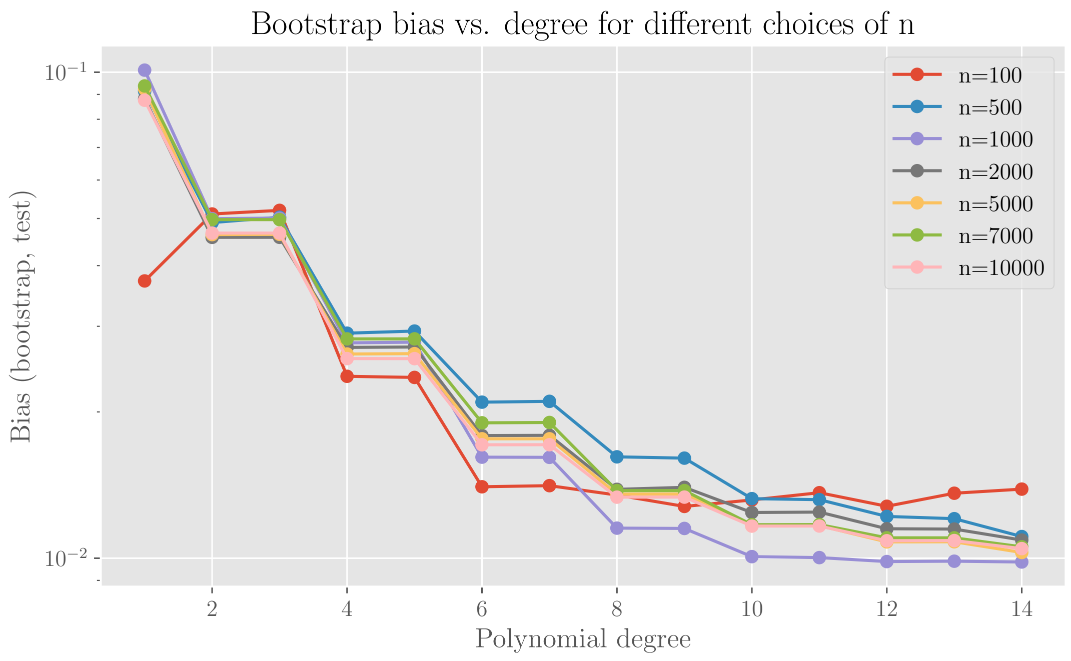


FIG. 27. Bias (bootstrap, OLS) on hold-out test as a function og polynomial degree for different choices of n.

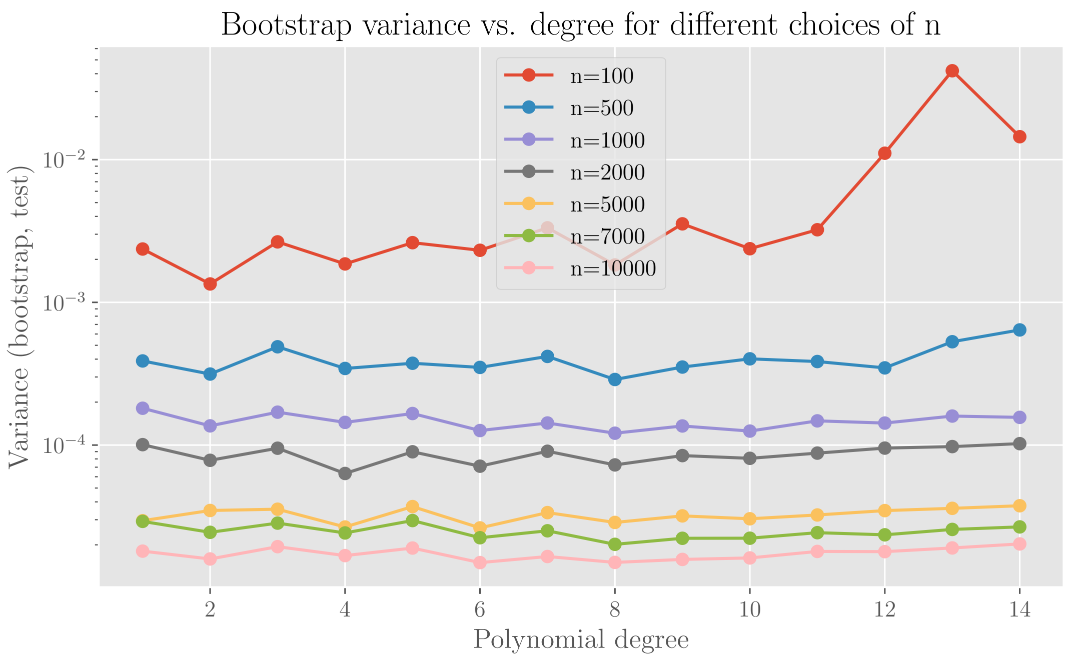


FIG. 28. Variance (bootstrap, OLS) on hold-out test as a function of polynomial degree for different choices of n.

G. K-fold CV

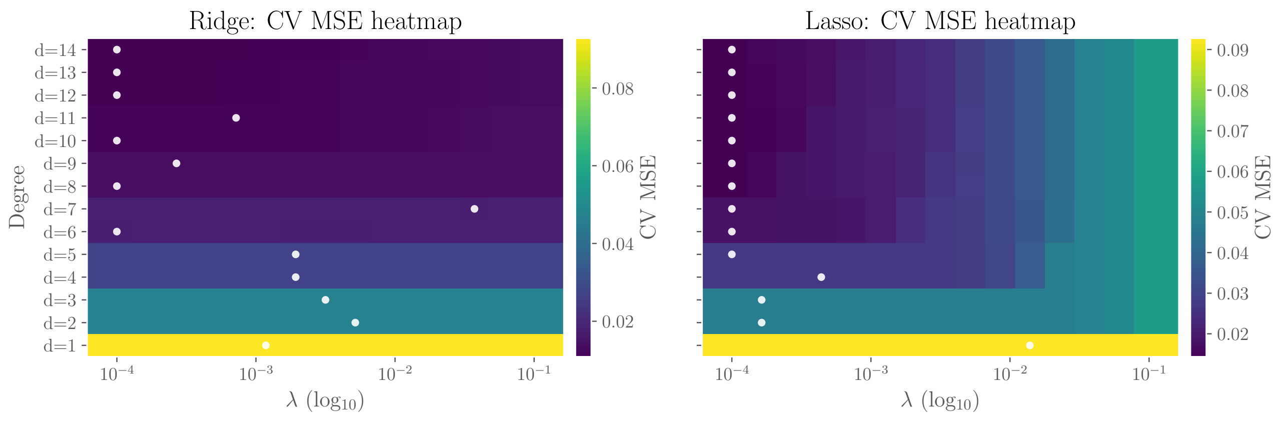


FIG. 29. Best λ (white dots) per degree, overlaid on 10-fold CV–MSE heatmaps for Ridge (left) and Lasso (right). Darker colors indicate lower CV MSE. The design matrix is standardized after the polynomial expansion.