The Great Regression

FYS-STK3155: Project 1

Bror Johannes Tidemand Ruud (640394), Egil Furnes (693784) & Ådne Rolstad (693783)
University of Oslo

(Dated: 06.10.2025)

Approximating the Runge function $(1/(1+25x^2))$ with linear regression is important because it illustrates how regression models handle overfitting and model complexity. The Runge function is known to cause oscillations for high-degree polynomials, making it ideal for studying model complexity and regularization. In this report, Ordinary Least Squares (OLS), Ridge regression, and LASSO regression are implemented using both analytical and gradient descent (GD) methods. Bootstrapping and cross-validation (CV) were applied to estimate bias and variance. Ridge regression achieved the lowest mean squared error (MSE = 0.0019) with a polynomial of degree 13 and regularization $\lambda = 10^{-5}$, while OLS and LASSO produced slightly higher MSEs of 0.0106 and 0.0028, respectively. GD with ADAM produced the best GD performance for Ridge, OLS and LASSO regression. Our results show that linear regression can approximate the Runge function effectively for moderate complexity, but higher-degree models tend to overfit. Regularization helps reduce variance and improve model stability.

I. INTRODUCTION

Linear regression models are widely used to approximate continuous relationships between input variables and a target variable. Despite their simplicity, they offer interpretable coefficients and can perform surprisingly well, even for non-linear data, when combined with polynomial features and regularization [1].

In this study, we approximate Runge's function $1/(1+25x^2)$, a classical test function in numerical analysis known for its strong oscillations when fitted with high-degree polynomials. The Runge function is particularly interesting because it highlights fundamental challenges in regression analysis, such as over-fitting, numerical instability, and the bias–variance trade-off. Approximating this function therefore provides a clear framework for studying how model complexity and regularization affect performance.

We implement and compare Ordinary Least Squares (OLS), Ridge, and LASSO regression models, using both analytical solutions and gradient descent (GD) methods. To evaluate model performance, we use metrics such as mean squared error (MSE) and R^2, before using resampling methods bootstrap and cross-validation (CV) to analyse the bias-variance trade-off. This allows us to explore how different regression models balance accuracy and generalization when approximating the Runge function [2].

First, we describe the methods and algorithms used in the report in section II, before presenting and discussing results in section III, and finally summarize and conclude the report in section IV. Furthermore, section V contain additional figures and algorithms used in the report.

II. METHODS

We introduce performance metrics MSE and R^2, the theoretical foundation for the standard regression models OLS, Ridge, and LASSO in section II A 2, before describing variations of the standard GD method and stochastic gradient descent (SDG) in section II A 3. We conclude with deducing the bias-variance trade-off in section II A 4.

A. Method

Before presenting models, we introduce the evaluation metrics used throughout the report, MSE and R^2. MSE quantifies the average squared deviation between predictions $\tilde{\mathbf{y}}$ and true values $\mathbf{y}$, shown in eq. (1). R^2 score measures the proportion of variance in the data explained by the model, where eq. (2), $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ is the sample mean. As such, a 'better' model should have lower MSE and higher R^2.

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

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

1. Splitting and scaling the data

Before training machine learning models, we split and scale the data appropriately. We perform an 80/20 train-test split to evaluate model performance on unseen data. Since we train without an intercept term, we scale the entire feature matrix using the function `StandardScaler` from the `Python` package `sklearn`. It standardizes features by removing the mean and scaling to unit variance, with the standardization formula shown in eq. (3).

$$z = \frac{x - \mu}{\sigma} \tag{3}$$

2. Standard regression methods

First, we derive the standard regression methods [3]. Consider a linear model in matrix form $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$ where $\mathbf{y} \in \mathbb{R}^n$ is the response vector, $\mathbf{X} \in \mathbb{R}^{n\times p}$ is the design matrix, $\boldsymbol{\beta} \in \mathbb{R}^p$ contains the model coefficients, and $\boldsymbol{\varepsilon}$ is random noise with $\mathbb{E}[\boldsymbol{\varepsilon}] = \mathbf{0}$ and $\mathrm{Var}[\boldsymbol{\varepsilon}] = \sigma^2\mathbf{I}$. In OLS, the estimator minimizes the residual sum of squares in eq. (4), which yields a closed-form solution when $\mathbf{X}^\top\mathbf{X}$ is invertible in eq. (5):

$$\hat{\boldsymbol{\beta}}_{\mathrm{OLS}} = \arg\min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 \tag{4}$$

$$\hat{\boldsymbol{\beta}}_{\mathrm{OLS}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} \tag{5}$$

Ridge regression introduces an L_2 penalty that shrinks coefficients toward zero. The estimator minimizes the penalized cost function with regularization parameter $\lambda \geq 0$ in eq. (6). The closed-form solution is shown in eq. (7):

$$\hat{\boldsymbol{\beta}}_{\mathrm{Ridge}} = \arg\min_{\boldsymbol{\beta}} \left(\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_2^2\right) \tag{6}$$

$$\hat{\boldsymbol{\beta}}_{\mathrm{Ridge}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y} \tag{7}$$

LASSO regression replaces the L_2 penalty with an L_1 penalty, where $\|\boldsymbol{\beta}\|_1 = \sum_{j=1}^p |\beta_j|$. Unlike OLS and Ridge, LASSO has no closed-form solution and requires iterative optimization methods such as coordinate descent. The L_1 penalty in eq. (8) encourages sparsity, driving some coefficients exactly to zero.

$$\hat{\boldsymbol{\beta}}_{\mathrm{LASSO}} = \arg\min_{\boldsymbol{\beta}} \left(\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1\right) \tag{8}$$

3. GD methods

In linear regression the goal is to minimize a cost function that depends on the model parameters. For OLS and Ridge, optimal parameters can be found analytically, however when more complex models and cost functions make this impossible, numerical optimization methods like GD can be used to approximate optimal parameters.

The main idea behind GD is to start with an initial guess for the parameters, compute the gradient of the cost function, and update parameters by taking steps in the direction opposite to the gradient, where η determines the step size. This is repeated iteratively until the algorithm either converges or a predefined maximum number of iterations is reached.

To use GD methods, we calculate gradients for the OLS, Ridge and LASSO regression. The gradient of the cost function in OLS regression is shown in eq. (9), Ridge in eq. (10), and the cost function in LASSO is defined in eq. (11).

$$\nabla_{\boldsymbol{\theta}} C_{OLS}(\boldsymbol{\theta}) = -\frac{2}{n}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T\boldsymbol{X} \tag{9}$$

$$\nabla_{\boldsymbol{\theta}} C_{Ridge}(\boldsymbol{\theta}) = 2((\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})^T\boldsymbol{X} + \lambda\boldsymbol{\theta}^T) \tag{10}$$

$$C_{LASSO}(\boldsymbol{\theta}) = \frac{1}{n}\|\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{\theta}\|_1 \tag{11}$$

In eq. (11) the L_1 penalty $\|\boldsymbol{\theta}\|_1 = \sum_i |\theta_i|$ encourages sparsity in the model parameters. Because the absolute value is not differentiable at $\theta_i = 0$, the gradient is not defined everywhere, and standard GD cannot be applied directly. Instead, methods such as proximal GD are used, where the soft threshold operator in eq. (12) is applied during each update step to shrink coefficients towards zero and set small ones exactly to zero.

$$S_\lambda(z) = \mathrm{sign}(z)\ \max(|z| - \lambda, 0) \tag{12}$$

Going further than standard GD, we implemented GD with momentum, ADAGrad, RMSProp and ADAM. The algorithm for standard GD is described in algorithm 1 inspired by [4], while the algorithms for the aforementioned variations on GD if found in the appendix in section V D.

Algorithm 1 Standard GD

```
1: Input: learning rate $\eta$, number of iterations $T$, feature matrix $\boldsymbol{X}$, target values $\boldsymbol{y}$
2: Initialize parameters $\boldsymbol{\theta}^{(0)}$
3: for $t = 1$ to $T$ do
4:     Compute gradient $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$
5:     Update parameters:
           $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \eta\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$
6: end for
7: Output: optimized parameters $\boldsymbol{\theta}^{(T)}$
```

Standard GD can be slow to converge and may get trapped in local minima [5]. GD with momentum helps accelerate convergence and smooth out updates by incorporating information from previous iterations, allowing the algorithm to escape shallow local minima. The algorithm for GD with momentum is described in algorithm 5.

GD with momentum improves convergence by accumulating information from previous gradients. However, it still uses a single global learning rate for all parameters. The AdaGrad algorithm adapts the learning rate individually for each parameter based on the historical magnitude of its gradients, allowing larger updates for infrequently changing parameters and smaller updates for frequently changing ones. The algorithm for AdaGrad is described in algorithm 6.

While AdaGrad adapts the learning rate for each parameter, its accumulated squared gradients can grow indefinitely, causing the effective learning rate to decrease too quickly. RMSProp addresses this by using an exponentially decaying average of past squared gradients, which allows the learning rate to adapt dynamically without diminishing over time. The algorithm for RMSProp is described in algorithm 7.

RMSProp adapts the learning rate using an exponentially decaying average of past squared gradients, and it does not take the direction of the gradient updates into account. ADAM combines the benefits of both RMSProp and momentum by maintaining exponentially decaying averages of both past gradients and their squared values, along with bias correction. This makes ADAM robust and efficient for a wide range of optimization problems. The algorithm for ADAM is described in algorithm 8.

In practice, GD is often implemented in a stochastic or mini-batch setting, where only a subset of the training data is used to compute the gradient at each iteration [6]. This approach reduces computational cost and can improve convergence speed by introducing noise that helps the algorithm escape local minima. All the gradient-based methods described above (standard GD, GD with momentum, AdaGrad, RMSProp and Adam) can be implemented in this way by updating the parameters using gradients computed on mini-batches rather than the full dataset.

Algorithm 2 Mini-batch (Stochastic) GD

```
1: Input: learning rate $\eta$, batch size $m$, number of iterations $T$, feature matrix $\boldsymbol{X}$, target values $\boldsymbol{y}$
2: Initialize parameters $\boldsymbol{\theta}^{(0)}$
3: for $t = 1$ to $T$ do
4:     Randomly sample a mini-batch $(\boldsymbol{X}_b, \boldsymbol{y}_b)$ of size $m$ from $(\boldsymbol{X}, \boldsymbol{y})$
5:     Compute gradient on the mini-batch:
           $\nabla_{\boldsymbol{\theta}} C_b(\boldsymbol{\theta}^{(t-1)}) = \frac{2}{m}\boldsymbol{X}_b^\top(\boldsymbol{X}_b\boldsymbol{\theta}^{(t-1)} - \boldsymbol{y}_b)$
6:     Update parameters $\boldsymbol{\theta}^{(t)}$ according to the chosen optimization rule (e.g., GD, Momentum, AdaGrad, RMSProp, Adam)
7: end for
8: Output: optimized parameters $\boldsymbol{\theta}^{(T)}$
```

SGD methods can achieve faster convergence and require less computational effort compared to standard GD techniques. However, the inherent randomness in these methods can cause the convergence process to become irregular or unstable.

In order to analyse how well the GD methods approximate the closed-form solution, for Ridge and OLS, or the solution from Scikit-Learn, for LASSO, we fit models using the parameters found by the GD methods. In section III we compare MSE and R^2 and study whether analytical and numerical methods differ, and if so, how many iterations are needed to converge. We also show how convergence rates depend on model complexity.

4. Bias-variance trade-off

Next, we deduce the bias-variance trade-off. Bias is the difference from prediction to correct value and variance is the variability in model prediction for a given data point [7]. In short, a too simple model might have higher bias, and a more complex model might have higher variance. Seeking to optimize this, we want to minimize the sum of bias and variance, shown in eq. (13) and eq. (14). In particular we want to show that total error can be split into bias and variance as shown in eq. (15) and eq. (16).

$$\text{Total Error} = \text{Bias}^2 + \text{Var} + \text{Irreducible Error} \tag{13}$$

$$\mathbb{E}\big[(y-\tilde{y})^2\big] = \text{Bias}[\tilde{y}]^2 + \text{Var}[\tilde{y}] + \sigma^2 \tag{14}$$

$$\text{Bias}[\tilde{y}]^2 = \Big(\mathbb{E}[\tilde{y}] - y\Big)^2 \tag{15}$$

$$\text{Var}[\tilde{y}] = \mathbb{E}\Big[\big(\tilde{y} - \mathbb{E}[\tilde{y}]\big)^2\Big] \tag{16}$$

We decompose the expected prediction error $\mathbb{E}[(y-\tilde{y})^2]$, by taking squared error between the true observation y and our prediction $\tilde{y}$. The true relationship is $y = f(x) + \varepsilon$ with function $f(x)$ and irreducible noise ε.

$$\begin{aligned}\mathbb{E}[(y-\tilde{y})^2] &= \mathbb{E}[(f(x)+\varepsilon-\tilde{y})^2] \\ &= \mathbb{E}[(f(x)-\tilde{y})^2 + 2(f(x)-\tilde{y})\varepsilon + \varepsilon^2] \\ &= \mathbb{E}[(f(x)-\tilde{y})^2] + 2\mathbb{E}[\varepsilon]\mathbb{E}[f(x)-\tilde{y}] + \mathbb{E}[\varepsilon^2]\end{aligned}$$

Since the noise has zero mean $\mathbb{E}[\varepsilon] = 0$ the middle term $2\mathbb{E}[\varepsilon]\mathbb{E}[f(x)-\tilde{y}]$ vanishes. Furthermore, the variance of the noise is $\mathbb{E}[\varepsilon^2] = \sigma^2$ which gives us the following:

$$\begin{aligned}&= \mathbb{E}[(f(x)-\tilde{y})^2] + \sigma^2 \\ &= \mathbb{E}[(f(x)-\mathbb{E}[\tilde{y}]+\mathbb{E}[\tilde{y}]-\tilde{y})^2] + \sigma^2 \\ &= \mathbb{E}[\{(f(x)-\mathbb{E}[\tilde{y}]) + (\mathbb{E}[\tilde{y}]-\tilde{y})\}^2] + \sigma^2 \\ &= \mathbb{E}[(f(x)-\mathbb{E}[\tilde{y}])^2] \\ &\quad + 2\mathbb{E}[(f(x)-\mathbb{E}[\tilde{y}])(\mathbb{E}[\tilde{y}]-\tilde{y})] \\ &\quad + \mathbb{E}[(\mathbb{E}[\tilde{y}]-\tilde{y})^2] + \sigma^2 \\ &= \mathbb{E}[(f(x)-\mathbb{E}[\tilde{y}])^2] \\ &\quad + 2\mathbb{E}[(f(x)-\mathbb{E}[\tilde{y}])]\mathbb{E}[(\mathbb{E}[\tilde{y}]-\tilde{y})] \\ &\quad + \mathbb{E}[(\mathbb{E}[\tilde{y}]-\tilde{y})^2] + \sigma^2 \\ &= \mathbb{E}[(f(x)-\mathbb{E}[\tilde{y}])^2] + \mathbb{E}[(\mathbb{E}[\tilde{y}]-\tilde{y})^2] + \sigma^2 \\ &= \text{Bias}[\tilde{y}]^2 + \text{Var}[\tilde{y}] + \sigma^2\end{aligned}$$

We used the trick of adding/subtracting $\mathbb{E}[\tilde{y}]$ to $\mathbb{E}[(f(x)-\tilde{y})^2]$ thus expanding the square. Cross term disappear from definition of the expectation $\mathbb{E}[\tilde{y} - \mathbb{E}[\tilde{y}]] = 0$.

As such, we are left with the squared bias ($\text{Bias}[\tilde{y}]^2$), the variance $\text{Var}[\tilde{y}]$, and the irreducible error σ^2.

We highlight this trade-off in figure 1, where both models show higher bias and lower variance for low polynomial degrees, and where the test data show higher variance with lower bias as model complexity increases.

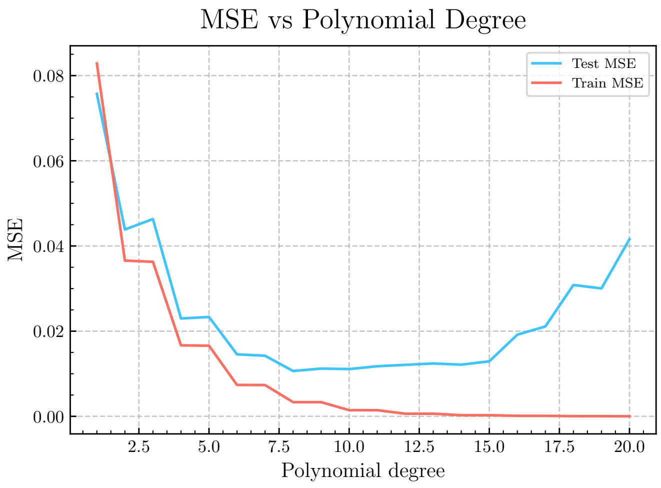

Figure 1: Training and test MSE versus polynomial degree. Test MSE minimizes around degree 8, while training MSE decreases monotonically to near-zero, indicating over-fitting at high degrees. The divergence illustrates the bias-variance trade-off.

5. *Resampling methods*

Resampling methods allow us to assess model performance by repeatedly train and evaluate the model on different subsets of the train data.

The bootstrap method explained in algorithm 3 generate multiple sets of train data by taking a random sample with replacement, each having the same size as the original train data. As such, some observations appear multiple times while others are excluded. This approach is useful for estimating uncertainty in model parameters and predictions [8].

Algorithm 3 Bootstrap Resampling

```
1: **Input:** data $x = (x_1, \dots, x_n)$, iterations $B$
2: **for** $b = 1$ to $B$ **do**
3:     Sample with replacement $n$ indices $i_1, \dots, i_n$ from $\{1, \dots, n\}$
4:     Set $x^* = (x_{i_1}, \dots, x_{i_n})$
5:     Compute statistic $\hat{\theta}^{(b)}$ from $x^*$
6: **end for**
7: **Output:** bootstrap estimates $\{\hat{\theta}^{(b)}\}_{b=1}^{B}$
```

CV in algorithm 4 partition train data into k disjoint folds, train the model on $k-1$ folds and test on the remaining fold. Rotating through all k folds, each observation serves as a test point exactly *once*. This approach provides a less variable estimate of model performance compared to a single train-test split [9].

B. Implementation

The coding in this project was implemented using `Python` and written in the `VSCode` and `Cursor IDE` using

Algorithm 4 k-fold CV

```
1: **Input:** dataset $\mathcal{D}$, number of folds $k$
2: Randomly shuffle $\mathcal{D}$ and partition into $\{\mathcal{D}_1, \dots, \mathcal{D}_k\}$
3: **for** $i = 1$ to $k$ **do**
4:     $\mathcal{D}_{\text{test}} \leftarrow \mathcal{D}_i$
5:     $\mathcal{D}_{\text{train}} \leftarrow \bigcup_{j \neq i} \mathcal{D}_j$
6:     Train model on $\mathcal{D}_{\text{train}}$ to obtain $\hat{f}^{(i)}$
7:     Evaluate performance $s_i$ on $\mathcal{D}_{\text{test}}$
8: **end for**
9: **Output:** performance scores (e.g., mean $\bar{s} = \frac{1}{k}\sum_{i=1}^{k} s_i$)
```

a collaborated `GitHub` [10], and the report is co-written in LaTeXusing OverLeaf. Whenever running code in our project, for reproducibility, we used a global `set.seed()` of `6114`, the sum of our day-month birthdays.

Writing `Python` we used a plethora of packages, among those `NumPy` [11] for efficient array operations and linear algebra, `Pandas` for for data manipulation and analysis [12], `scikit.learn` for training/evaluation [13], and `matplotlib.pyplot` for generating plots [14].

Seeking to write a project using modular and reusable code in `GitHub`, we divided general functions like `OLS.py`, `Ridge.py`, `gradient_descent.py`, and `stochastic_gradient_descent.py`. We then imported these modules when solving the project, split into one `Jupyter` notebook per part, like `Exa.ipynb`.

Solving the project chronologically, we found this approach to work well for a while. Although, when introducing the alternative GD methods in `Part: d` this led to a large number of functions. A simpler and more elegant approach would have been to instead use classes and object-oriented programming in `Python`. This would be better, especially with many mutations of three regression models per GD alternative. However, the use of functions are intuitive, and implementing each function gave us better insight on how the GD methods functions.

Throughout the project we used tests to validate that our implementations produced correct results. For instance, we systematically compared analytical Ridge and numerical GD results in Listing 1, confirming whether numerical GD parameters equal Ridge within a given tolerance. Similar tests were made for all GD implementations.

```python
# Train model using gradient descent
beta, t = gradient_descent_Ridge(X_train_s,
    y_train, lam=lam, eta=eta, num_iters
    =100000)

print("GD parameters:", beta)
print(f"Number of iterations: {t}")

# Compute closed-form solution
beta_closed = Ridge_parameters(X_train_s,
    y_train, lam)

print("Ridge coefficients:", beta_closed)

# Verify convergence
```

```
tol = 1e-3
is_close = np.isclose(beta, beta_closed, rtol
    =tol, atol=tol)

print(f"GD equal to closed-form: {is_close}")
print(f"Tolerance = {tol}")
```

Listing 1: Ridge regression: Gradient descent validation against closed-form solution

C. Use of AI tools

In this project we have leveraged AI tools in assisting both coding and writing. For coding, we have found it useful to use AI-models as partners in writing, debugging, and talking about the tasks at hand. For writing the report, it has been quite useful to leverage these tools for help on formatting and phrasing.

For coding, we are all frequent users of the models published by OpenAI, namely `ChatGPT 5` [15]. Some of us use the `Cursor IDE` from Anyspehere as our standard IDE [16], which integrates Antrophic's `Claude AI` and their latest model `Sonnet 4.5` [17].

Of course, writing the project in `GitHub`, include assistance from `GitHub Copilot` [18]. Some of us even used `Grok 4` from xAI [19]. For instance, here we used `Grok 4` to suggest a good 'docstring' for our OLS function in `Python` [20].

These AI tools have been quite useful when writing the report too, for instance we got help with formatting figure captions from `Sonnet 4.5` [17], or formatting LaTeXtables [21] and references in the `.bib`-file from `ChatGPT 5` [22].

In [23], we used ChatGPT 5 to combine plots and create a more consistent and efficient layout for the report figures. It also assisted in developing a styling function to ensure that all plots followed a uniform LaTeX-style appearance, and in troubleshooting LaTeX compilation errors encountered when embedding plots generated in Python.

In [24], we used ChatGPT 5 to debug the bootstrap implementation of the bias–variance trade-off and tune parameters to achieve a clearer decomposition. It was also used to implement Scikit-Learn pipelines for exploring OLS, Ridge, and LASSO performance as model complexity increased. Finally, ChatGPT 5 assisted in generating boxplots to visualize the variation in MSE across different values of k in the k-fold CV experiments.

In [25], we use `ChatGPT 5` to explain the standard GD method we implemented, to better understand the algorithm. `ChatGPT 5` then suggested a docstring for the function.

III. RESULTS AND DISCUSSION

This section presents the results of applying OLS, Ridge, and LASSO regression to Runge's function. We first compare model performance as a function of model complexity and regularization strength. Next, we explore optimization using GD methods and finally examine the bias–variance trade-off with resampling techniques such as bootstrapping and CV. The optimal fit for each regression model is presented in table I, where we find that the Ridge with polynomial degree 13 and $\lambda = 10^{-5}$ yield the lowest MSE and highest R^2, followed by a LASSO with degree 10 and $\lambda = 10^{-5}$. The standard OLS with degree 8 give the worst MSE (0.0106) and R^2 (0.8444) out of the standard regression models.

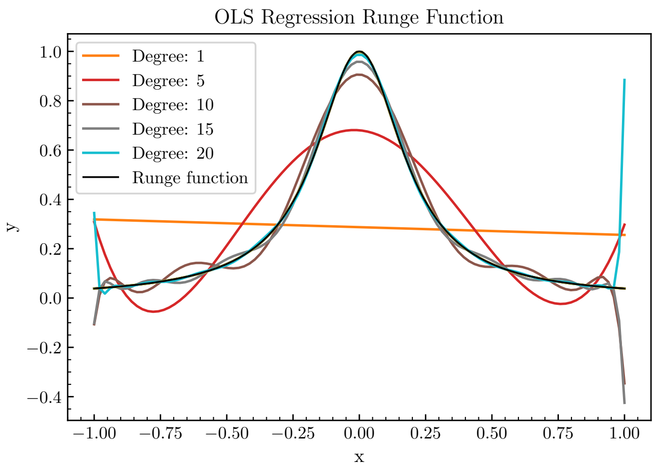

Figure 2: Comparison of OLS polynomial regression fits of varying degrees (1, 5, 10, 15, and 20) against the true Runge function.

Table I: Best Standard Regression Models

Model	Poly. degree	λ	MSE	R^2
OLS	8	—	0.0106	0.8444
Ridge	13	10^{-5}	0.0019	0.9730
LASSO	10	10^{-5}	0.0028	0.9593

A. Regression models

1. OLS

Figure 2 show the fitted polynomials for degrees $[1, 5, 10, 15, 20]$ with the true Runge function. Increasing polynomial degree from 1 to 20, we see a progression from severe under-fitting to an apparent good fit, and finally to extreme over-fitting near the boundaries for Degree: 20.

In Figure 3, we compare the MSE obtained when fitting a model to a clean target (the true Runge function) with the MSE from fitting the same model to a noisy dataset. As expected, the MSE increases in the presence of noise, particularly for models with higher complexity. The added noise degrades the quality of the fit, leading to overfitting at higher polynomial degrees. The optimal OLS fit is achieved at polynomial degree 8, with an MSE of 0.0106

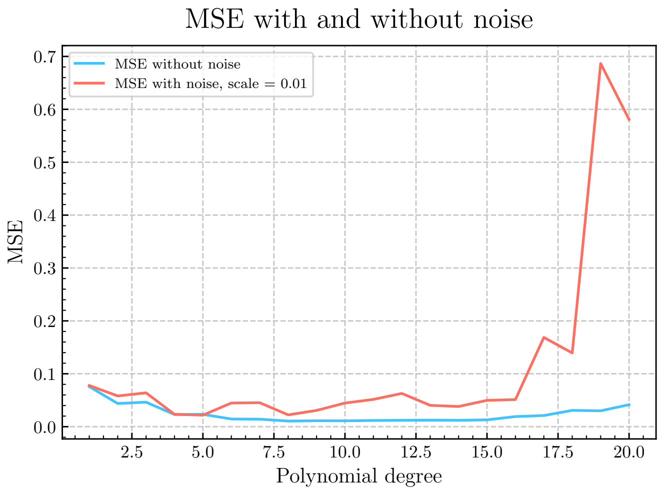

Figure 3: Comparing MSE when fitting a model on the true Runge function, and with added stochastic noise.

In figure 4 we increase the model complexity and we see that the β parameters 'blow up' with increasing complexity (higher polynomial degree). This exponential increase in coefficient values is a direct consequence of the ill-conditioning of the design matrix $\mathbf{X}^T\mathbf{X}$ at high polynomial degrees, even after standardization. This pattern reflects the increasing correlation between polynomial basis functions as degree increases: fitting x^{20} in the presence of $x^{19}, x^{18}, \ldots$ requires enormous coefficient adjustments to capture subtle differences between these highly correlated features. This exponential increase in parameters explain the boundary oscillations discussed earlier.

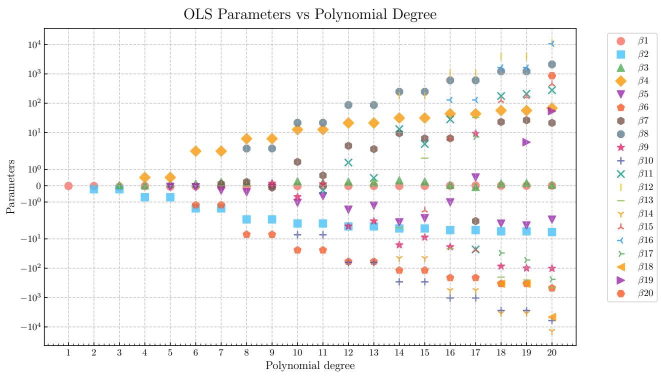

Figure 4: Evolution of OLS regression coefficients (β_1 through β_{20}) as polynomial degree increases when fitting the Runge function. Coefficients remain stable below degree 8, then explode, producing the boundary oscillations in Figure 2.

2. Ridge

Ridge regression introduces an L2 penalty term $\lambda\|\boldsymbol{\beta}\|_2^2$ to the OLS cost function, explicitly constraining coefficient magnitudes to combat the numerical instabilities observed in Figure 4.

Table II in appendix V A 2 show the MSE for different values of λ and polynomial degrees. Based on these values, we see that the best fit with Ridge regression is obtained at polynomial degree 13, with a small regularization penalty $\lambda = 10^{-5}$. Based on this, we will study polynomials of degree 13 in our analysis of Ridge regression. The plot of the Ridge model with the optimal parameters can be found in figure 19 in appendix V A 2.

Figure 5 show how the MSE depends on the regularization parameter λ for polynomial degree 13. The figure reveals that too strong regularization increases the MSE, while a small regularization can yield a low MSE. One interpretation of this is that large regularization introduces a large bias, while reducing variance, which leads to a larger MSE when the regularization becomes too large. A small regularization may dampen the large parameters in a high-complexity-model, and thus reduce the oscillations at the end-points of the interval. This, in turn, reduces the variance of the fit. Since the optimal

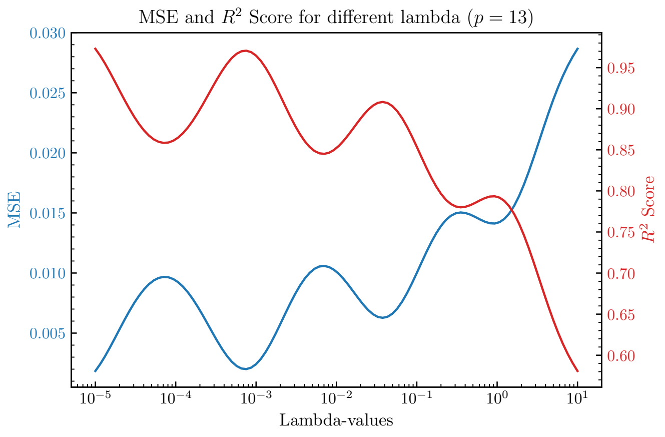

Figure 5: MSE as a function of the regularization parameter λ, for polynomial degree 13, with Ridge regression.

polynomial degree was 8 for OLS, it is interesting to see how the Ridge penalty affects the MSE and R2-score for this degree. Figure 16 in appendix V A 2 shows the MSE and R2-score as a function of the regularization for this degree. Without going into too much detail, the Ridge regularization improves the MSE for degree 8, with a minimal MSE obtained with $\lambda \approx 0.005$.

Figure 6 illustrate the shrinkage of coefficients toward zero as the penalty strength λ increases (more shrinkage to the left in the figure, or larger λ). At minimal regularization ($\lambda = 10^{-5}$), the coefficients closely resemble the OLS solution, with the highest-degree terms (β_{12}, β_{13}) exhibiting magnitudes around $\pm 10^2$. As λ increases, all coefficients undergo monotonic decay, but they do so at different rates: higher-order coefficients shrink more rapidly than lower-order terms.

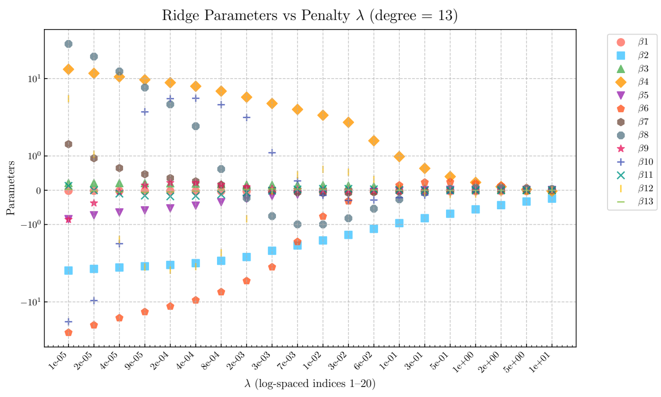

Figure 6: Ridge coefficients as a function of λ (degree 13).

3. LASSO

We used Scikit-learn to find the optimal parameters for LASSO regression. The MSE values for different regularization parameters λ and different polynomial degrees can be found in table III in the appendix. Based on these calculations, we found that the optimal LASSO fit was obtained at polynomial degree 10, with a regularization $\lambda = 10^{-5}$. The plot of this fit can be found in figure 20 in the appendix. The discussion of LASSO regression continues in section III B, where we compute LASSO parameters using our own GD implementations.

B. Gradient descent methods

1. Standard Gradient Descent

Figure 7 show how the MSE for standard GD diverges from the MSE obtained by using the closed-form solution for OLS, with a fixed learning rate $\eta = 0.01$. The R^2 comparison in Figure 22, appendix V B, reveal the same story, the GD method yields the same MSE as the closed-form solution up to degree 6 before diverging.

By implementing a stopping criterion in the gradient descent methods, the method will stop when the solution converges to the true solution, within a given threshold. When the model complexity increases, the cost function becomes more complex, and the standard gradient descent method needs more iterations in order to approximate the optimal parameters. Figure 8 shows this phenomenon for OLS: Increasing the polynomial degree yields a higher number of iterations. When the model becomes too complex, the standard gradient descent method fails to approximate the true parameters. This leads to a higher MSE, as shown in figure 7.

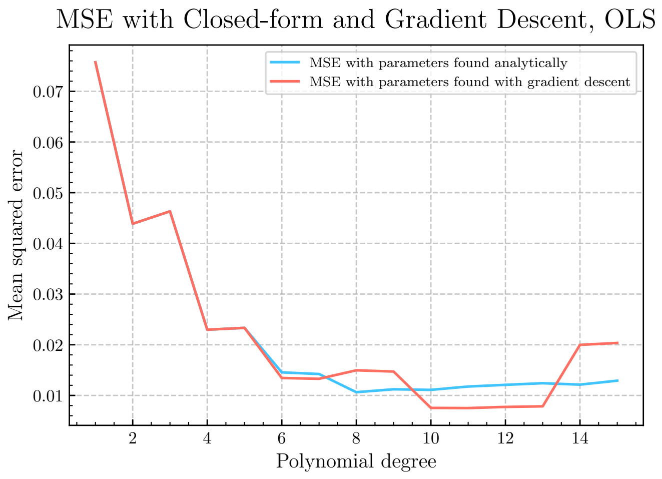

Figure 7: Comparison of MSE between closed-form and GD solutions.

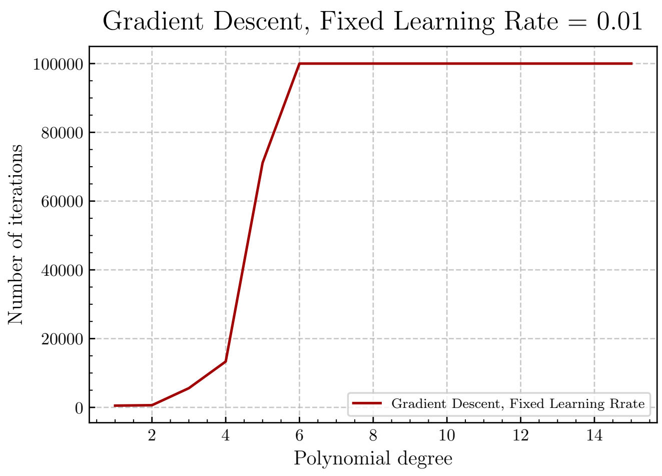

Figure 8: Iterations to convergence for gradient descent (fixed $\eta = 1$, tolerance 10^{-8}) versus polynomial degree. Low-degree models converge in $< 20{,}000$ iterations, while degrees ≥ 6 fail to converge within 100,000 iterations, revealing the ill-conditioning of high-degree polynomial regression that prevents standard GD from finding solutions.

Standard gradient descent with Ridge and LASSO yields similar results. When the cost function becomes complex, the method fails to converge. Figure 23 in the appendix shows that increasing the Ridge regularization yields more accurate parameters for standard gradient descent.

The regularization penalty in Ridge regression produces a less complicated cost function, which makes the standard GD method converge in fewer iterations, as shown in figure 21.

The results for standard gradient descent with LASSO regression is included in section III B 2.

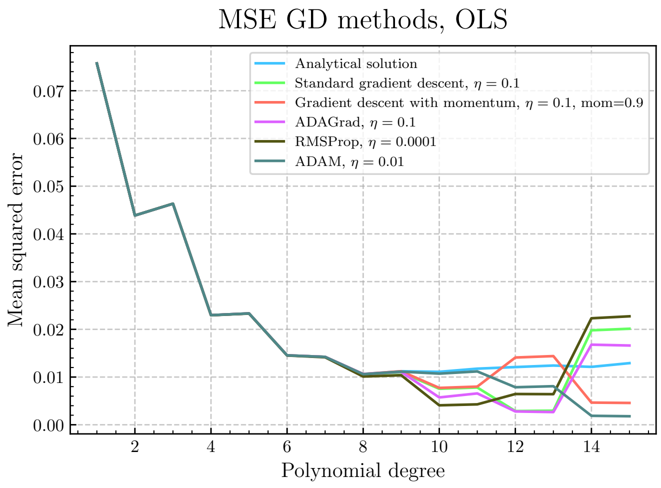

Figure 9: Gradient descent with OLS: Most methods diverge from the analytical MSE when the polynomial degree is greater than 9, while ADAM follows the analytical MSE up to degree 11.

2. *Gradient Descent with momentum, ADAGrad, RMSProp and ADAM*

We now introduce GD with momentum, ADAGrad, RMSProp and ADAM, in order to speed up convergence and handle more complex cost functions. For OLS, the performance of the different methods is given in figure 9. As before, we compare the MSE in order to understand the performance of the methods.

The figure shows that ADAM outperforms the other methods for GD. However, the results are extremely sensitive to the learning rate η. For instance, this figure shows that standard GD converges up to degree 9. In the last section we used a different learning rate, and got a worse result. The number of iterations needed for the different methods is shown in figure 10

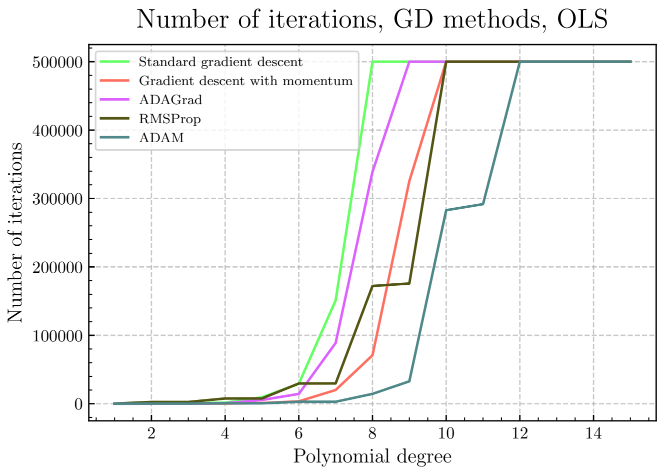

Figure 10: Number of iterations needed as a function of model complexity, OLS regression.

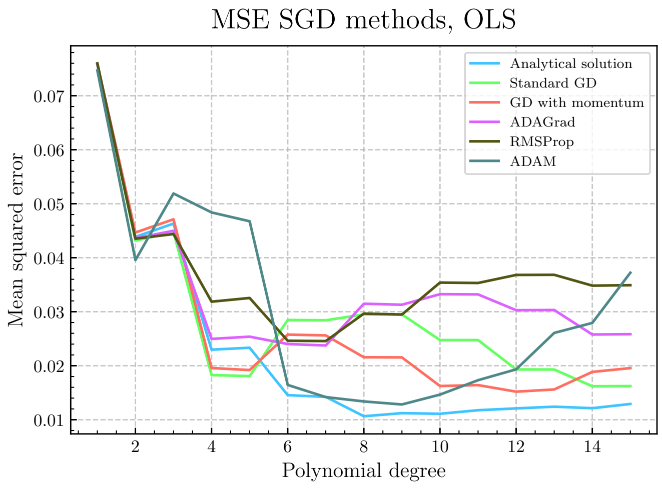

Figure 11: SDG with OLS: MSE comparison across polynomial degrees

We see that GD with ADAM is the fastest to converge, while GD with RMSProp and GD with momentum are also relatively fast. The plot also illustrates that GD with ADAM can handle a higher model complexity than the other methods.

For Ridge and LASSO regression, we get similar results. Figure 29 and 31 in the appendix show how the MSE from the GD methods compares to the GD from the closed-form solutions.

We see that ADAM outperforms the other methods, and that the regularization increases the accuracy of the parameters that are found numerically. For LASSO regression, both ADAGrad and ADAM gives results that are identical to the results obtained from Scikit-Learn, for all regularizations λ. The corresponding plots of the R^2-scores can be found in figure 28 and 30 in appendix V B The number of iterations until convergence is shown in figure 25 and 26 in the appendix.

We see that in both LASSO and Ridge regression, the number of iterations needed decreases with increased regularization. Again, the methods are sensitive to the learning rate, which could be the reason why RMSProp does not converge well in Ridge regression with high regularization.

3. *Stochastic Gradient Descent*

In order to reduce the computational cost of the GD methods, we implemented stochastic mini-batch gradient descent (SGD) for the methods we implemented earlier. Figure 11 show the MSEs found with the SGD methods for OLS regression. Figure 34 and 36 in the appendix show the similar results for Ridge and LASSO regression.

The plots of the corresponding R^2-scores can be found in figure 32, 33 and 35 in appendix V C.

While the stochastic methods for GD reduces the computational cost, we see that the parameters does not necessarily correspond to the ones obtained by closed-form solutions or Scikit-Learn. This is expected, as the randomness in the stochastic methods introduce some noise. The convergence and behaviuor of stochastic GD methods can be erratic, as we see in the figure.

C. Resampling methods

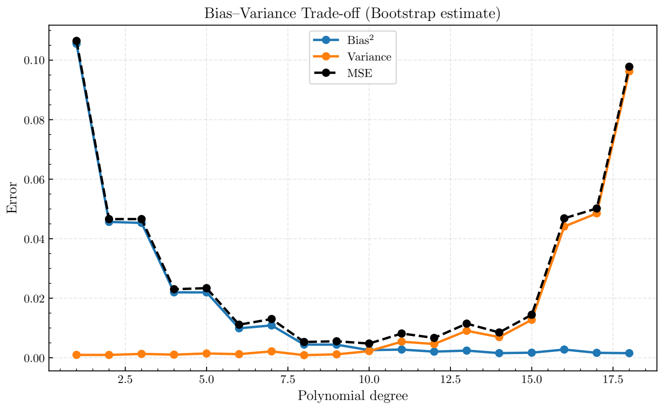

Figure 12: Bias Variance trade-off shown through bootstrapping. With bias-variance decomposition from MSE, fitted by OLS

We decompose the MSE into bias and variance in fig. 12. For low polynomial degrees, the MSE is dominated by bias, as the model is too simple to capture the true underlying relationship. As the polynomial degree increases, variance becomes the dominant contributor. This occurs because low degree models underfit and they produce similar predictions regardless of the training data. In contrast, high-degree models are overly flexible: even small changes in the training data lead to large variations in the fitted model, resulting in high variance.

Figure 13 shows the MSE obtained using k-fold CV with $k = 2$. We observe that all three models—OLS, Ridge, and LASSO—initially decrease in MSE as the polynomial degree increases. However, for OLS, the MSE starts to increase sharply beyond degree 10, while Ridge and LASSO remain stable. This behaviour arises because OLS becomes ill-conditioned at higher polynomial degrees when trained on small data partitions. With only two folds, each training set contains limited data, and the design matrix $X^\top X$ becomes nearly singular. This leads to unstable parameter estimates and inflated variance, causing the MSE to grow with model complexity.

In contrast, Ridge and LASSO include regularization terms that stabilize the solution and prevent over-fitting even when the data are limited. The ill-conditioning seen in OLS is reduced when we increase the number of folds. For example, with $k = 5$ (see Appendix 37), OLS produces a much smoother curve with an MSE comparable to Ridge and LASSO. Increasing k improves the conditioning of the training matrices and yields a more reliable estimate

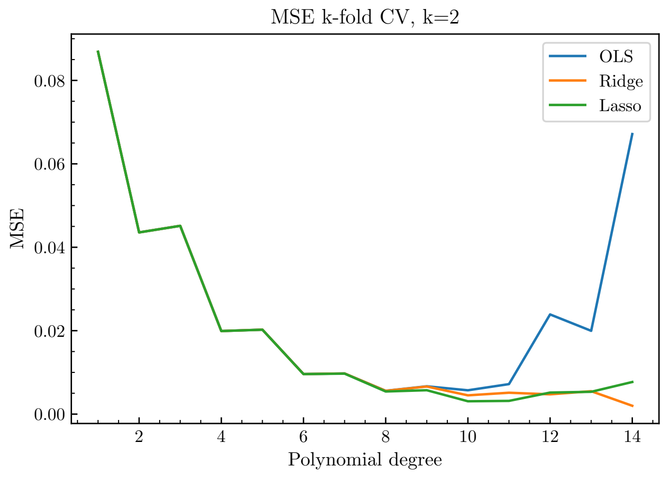

Figure 13: MSE with k-fold CV using $k = 2$, fitted with OLS, Ridge, and LASSO. With $\lambda = 10^{-5}$ for Ridge and Lasso. Which were found to be the optimal parameters III II

of model performance.

We also investigated how the choice of k influences the spread of MSE values across folds. For small k, such as $k = 2$, the test sets are large and the MSE estimates vary less between folds (see Figure 38). When k is increased to larger values, such as $k = 20$ (Figure 38), the test sets become small, leading to greater variation in the estimated MSE across folds, as reflected in the wider interquartile ranges of the boxplots.

Comparing the bootstrap and CV results, we observe that both methods reveal similar variance patterns for OLS. In the $k = 2$ case, OLS shows a sharp increase in MSE for higher polynomial degrees, resembling the high variance seen in the bootstrap results. When k increases, the OLS MSE becomes more stable and approaches the smoother behavior of Ridge and LASSO. This suggests that a higher number of folds improves the conditioning of the training data and reduces the variance in OLS estimates, while regularized models remain stable across all k values.

IV. CONCLUSION

A. Main Findings and Interpretations

In this project we have done a deep-dive into linear regression methods fit to Runge's function. We started by using the closed-form parameters for Ridge and OLS regression. This analysis showed that the optimal OLS fit was obtained at polynomial degree 8 (MSE=0.0106), while the optimal Ridge fit was obtained at polynomial degree 13 with a small regularization $\lambda = 10^{-5}$. This turned out to be the fit with the smallest MSE across all models (MSE = 0.0019). Our interpretation is that the small regularization in the Ridge fit reduced the oscillations near the endpoints of the interval, and thus limiting the undesired Runge phenomenon. For LASSO regression, we found the optimal parameters by using the functionality of Scikit-Learn. The smallest MSE for LASSO regression was obtained at polynomial degree 10, with a regularization $\lambda = 10^{-5}$ (MSE=0.0028).

We then expanded the analysis by introducing GD methods, namely the standard GD method, with momentum, ADAgrad, RMSprop and ADAM, before also introducing a SGD method. These methods were applied to LASSO, Ridge and OLS regression, and our analysis showed that a higher model complexity made the GD methods converge more slowly. For Ridge and LASSO regression, increasing the regularization made the GD methods converge more quickly, because the regularization penalized large coefficients, and thus making the cost function more simple. The GD method with ADAM performed best out of the GD methods. For OLS, the parameters from ADAM did not diverge from the closed-form parameters until polynomial degree reached 9.

SGD methods were implemented in order to speed up convergence. However, these methods introduced a random noise. This lead to some error in the parameters computed with SGD, compared to the closed-form parameters (OLS and Ridge) and the parameters from Scikit-Learn.

Finally, we performed a bias-variance analysis using the resampling methods bootstrapping and cross validation. Our results illustrate the bias-variance trade-off: overfitting causes high variance, while underfitting causes high bias.

1. *Pros and cons of this project and possible improvements*

OLS, Ridge, and LASSO regression each have distinct strengths and weaknesses. OLS provides an exact analytical benchmark and performs well for simple models, but it is highly sensitive to overfitting as model complexity increases. Ridge regression reduces this effect by penalizing large coefficients, resulting in more stable models at the cost of slightly higher bias. LASSO adds the advantage of feature selection by driving some coefficients to zero, but it can be unstable when predictors are strongly correlated.

GD methods enable flexible numerical optimization where closed-form solutions are not available, but they depend strongly on hyperparameter choices such as the learning rate and may suffer from slow or noisy convergence. In our results, we saw that the learning rate affected the convergence for standard GD with OLS, and this is also the case for other GD methods and with other linear regression models.

2. *Limitations of our conclusion*

This study focuses on a single one-dimensional test function, the Runge function, which limits the generality of the conclusions. The controlled setting provides clear insight into model behavior, but it does not fully capture the complexity of higher-dimensional or noisy real-world data. Furthermore, the analysis primarily compares polynomial models, so the results may not directly transfer to other basis functions or model types.

3. *Perspectives on future work*

Future work could extend this analysis to multidimensional regression problems or functions with varying smoothness to test model robustness under more realistic conditions. Additional experiments could explore adaptive or hybrid GD algorithms to improve convergence speed and stability. Furthermore, the GD algorithms implemented in this project could be made more efficient and modular by using object-oriented programming principles, for example by organizing the methods into classes to reduce redundancy and improve scalability. It would also be interesting to examine how different regularization strategies affect model interpretability and performance when applied to real datasets.

References

[1] 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`.

[2] Morten Hjorth-Jensen. *Computational Physics Lecture Notes 2015.* Department of Physics, University of Oslo, Norway, 2015. URL `https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf`.

[3] Wikipedia contributors. Ordinary Least Squares — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Ordinary_least_squares`, 2025. Accessed: 2025-10-04.

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

[5] Wikipedia contributors. Gradient descent — Wikipedia, The Free Encyclopedia, 2025. URL `https://en.wikipedia.org/wiki/Gradient_descent`. Accessed: 2025-10-04.

[6] Wikipedia contributors. Stochastic gradient descent — Wikipedia, The Free Encyclopedia, 2025. URL `https://en.wikipedia.org/wiki/Stochastic_gradient_descent`. Accessed: 2025-10-04.

[7] GeeksforGeeks. Bias-variance trade off - machine learning, August 2025. URL `https://www.geeksforgeeks.org/machine-learning/ml-bias-variance-trade-off/`. Accessed: 2025-10-04.

[8] Josef Waples. What is Bootstrapping in Statistics? A Deep Dive, 2024. URL `https://www.datacamp.com/tutorial/bootstrapping`. Published: 23 September 2024. Accessed: 2025-10-04. DataCamp.

[9] scikit-learn developers. 3.1. Cross-validation: evaluating estimator performance, 2025. URL `https://scikit-learn.org/stable/modules/cross_validation.html`. Accessed: 2025-10-04. © 2007–2025, scikit-learn developers (BSD License).

[10] Ådne Rolstad Bror Ruud, Egil Furnes. Fys-stk3155: Applied data analysis and machine learning projects. `https://github.com/egil10/FYSSTK3155`, 2025. GitHub repository. Department of Physics, University of Oslo.

[11] 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, 2020. doi: 10.1038/s41586-020-2649-2.

[12] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010.

[13] 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.

[14] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[15] OpenAI. Chatgpt. `https://chat.openai.com/`, 2025. Large language model developed by OpenAI. Accessed October 2025.

[16] Cursor AI. Cursor: The ai code editor. `https://cursor.com/`, 2025. AI-assisted development environment for programming. Accessed October 2025.

[17] Anthropic. Conversation with claude ai (sonnet 4.5). `https://claude.ai/share/035710da-2469-4a38-928d-cfbc75897de9`, 2025. Accessed via Claude AI, Sonnet 4.5 model at `https://claude.ai/`.

[18] GitHub. Github copilot. `https://github.com/features/copilot`, 2025. AI pair programmer developed by GitHub in collaboration with OpenAI. Accessed October 2025.

[19] xAI. Grok ai. `https://grok.com/`, 2025. Conversational AI model developed by xAI. Accessed October 2025.

[20] xAI. Grok Model Conversation. `https://grok.com/share/bGVnYWN5_8c0bdfaf-e62a-467e-9677-ee6c8456a0b2`, 2025. Accessed via `https://grok.com/`.

[21] Anthropic. Claude table formatting help. `https://claude.ai/share/0cdd65aa-9e6a-4988-911f-311532615f83`, 2025. Accessed: 2025-10-06.

[22] OpenAI. ChatGPT-5 Model Conversation. `https://chatgpt.com/share/68e26f5e-fb40-8005-8a45-2089ebd087cb`, 2025. Accessed via `https://chatgpt.com/`.

[23] OpenAI ChatGPT. Chatgpt: "combine plots with y-axes" (shared conversation). Online ChatGPT conversation, Oct 2025. URL `https://chatgpt.com/share/68e4f548-8518-800e-a98e-748b5037f1bb`. `https://chatgpt.com/share/68e4f548-8518-800e-a98e-748b5037f1bb`.

[24] OpenAI ChatGPT. Chatgpt: "second shared conversation". Online ChatGPT conversation, Oct 2025. URL `https://chatgpt.com/share/68e4f5d7-f17c-800e-8a47-c6ebc3c0dae3`. `https://chatgpt.com/share/68e4f5d7-f17c-800e-8a47-c6ebc3c0dae3`.

[25] OpenAI ChatGPT. Chatgpt: "gradient descent ols review" (shared conversation). Online ChatGPT conversation, Oct 2025. URL `https://chatgpt.com/share/68e4f80c-1e7c-800a-a13e-78166dc49f27`. `https://chatgpt.com/share/68e4f80c-1e7c-800a-a13e-78166dc49f27`.

Link to GitHub: `github.com/egil10/FYSSTK3155`

V. APPENDIX

A. Regression models

1. OLS

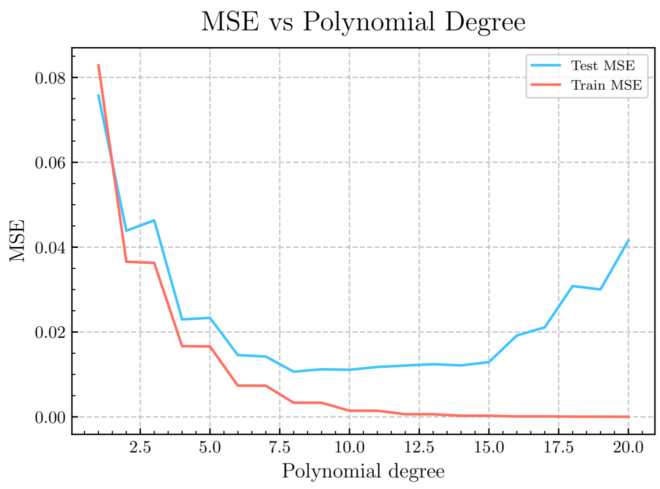

Figure 14: Training and test MSE versus polynomial degree. Test MSE minimizes around degree 8, while training MSE decreases monotonically to near-zero, indicating over-fitting at high degrees. The divergence illustrates the bias-variance trade-off.

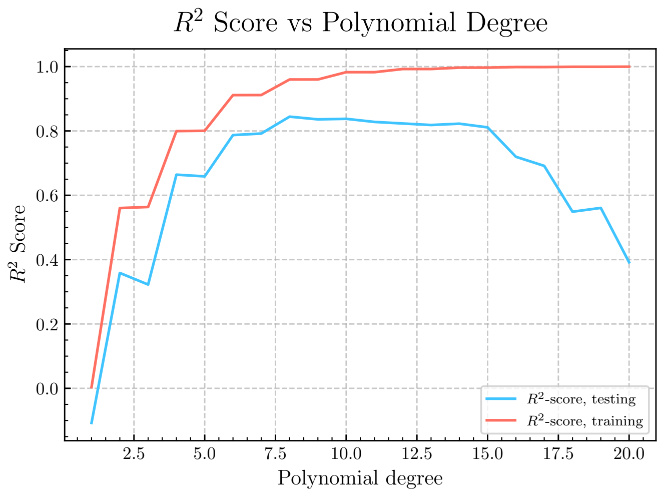

Figure 15: Training and test R^2 versus polynomial degree. Test R^2 peaks around degree 8 before declining due to over-fitting, while training R^2 approaches 1.0, mirroring the pattern observed in MSE (fig. 14).

2. Ridge Regression

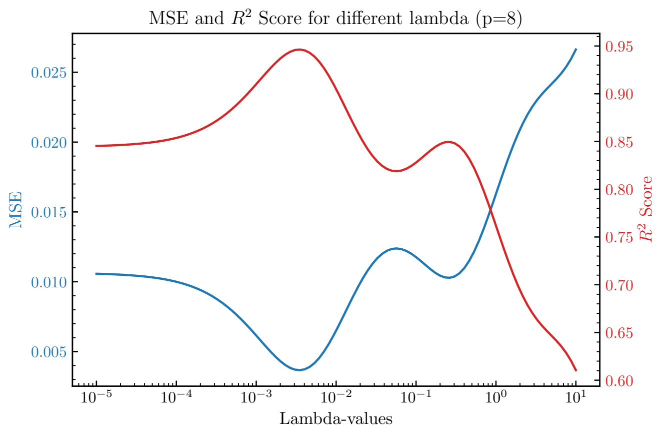

Figure 16: Ridge regression: MSE and R^2 for polynomial degree 8.

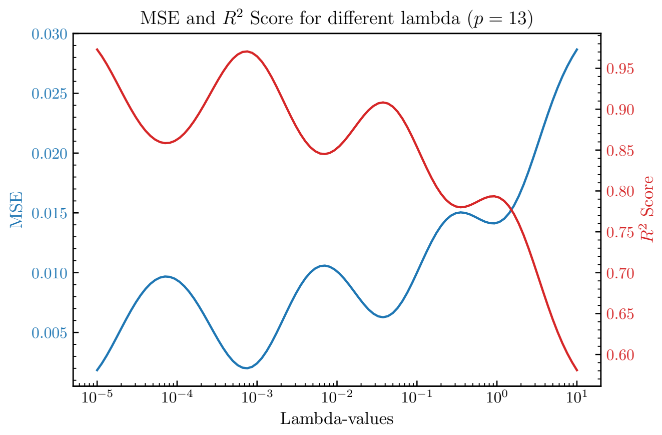

Figure 17: Ridge regression: MSE and R^2 for polynomial degree 13.

Table II: Mean squared error as a function of regularization parameter λ and polynomial degree for Ridge regression

Polynomial Degree	**Regularization Parameter λ**							
	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}
1	0.073961	0.073961	0.073961	0.073961	0.073961	0.073957	0.073919	0.073586
2	0.044121	0.044121	0.044121	0.044120	0.044115	0.044063	0.043561	0.039889
3	0.046265	0.046265	0.046264	0.046263	0.046253	0.046148	0.045205	0.040157
4	0.022012	0.022012	0.022011	0.021999	0.021882	0.020808	0.016504	0.032569
5	0.022425	0.022425	0.022423	0.022409	0.022269	0.021031	0.016482	0.032857
6	0.016000	0.015997	0.015971	0.015717	0.013556	0.008532	0.018819	0.026696
7	0.015683	0.015680	0.015656	0.015417	0.013372	0.008566	0.018794	0.026807
8	0.010057	0.009994	0.009456	0.005914	0.007315	0.012687	0.015677	0.025826
9	0.010613	0.010543	0.009941	0.006134	0.007272	0.012625	0.015671	0.025864
10	0.011960	0.009893	0.002381	0.007410	0.005532	0.013825	0.013597	0.026701
11	0.012643	0.010377	0.002425	0.007652	0.005642	0.013750	0.013614	0.026713
12	0.011376	0.002212	0.010002	0.002514	0.009629	0.010818	0.014277	0.027560
13	0.011686	0.002194	0.010192	0.002521	0.009872	0.010794	0.014314	0.027562
14	0.013253	0.004932	0.007447	0.005603	0.010727	0.008276	0.016370	0.027886
15	0.014056	0.005167	0.007436	0.005654	0.010931	0.008276	0.016447	0.027887

Table III: Mean squared error as a function of regularization parameter λ and polynomial degree for LASSO regression

Polynomial Degree	**Regularization Parameter λ**						
	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}
1	0.075719	0.075700	0.075512	0.073769	0.072478	0.072478	0.072478
2	0.043856	0.043819	0.043454	0.040080	0.037244	0.072478	0.072478
3	0.046310	0.046193	0.045064	0.040080	0.037244	0.072478	0.072478
4	0.022947	0.022732	0.020734	0.016442	0.037244	0.072478	0.072478
5	0.023292	0.022957	0.020740	0.016442	0.037244	0.072478	0.072478
6	0.014393	0.013039	0.007103	0.019969	0.037244	0.072478	0.072478
7	0.014148	0.013174	0.007103	0.019969	0.037244	0.072478	0.072478
8	0.009453	0.003526	0.010362	0.019969	0.037244	0.072478	0.072478
9	0.009924	0.003610	0.010568	0.019969	0.037244	0.072478	0.072478
10	0.002783	0.006515	0.010568	0.019969	0.037244	0.072478	0.072478
11	0.003300	0.007241	0.010718	0.019969	0.037244	0.072478	0.072478
12	0.006725	0.007241	0.010718	0.019969	0.037244	0.072478	0.072478
13	0.007620	0.007773	0.010854	0.019969	0.037244	0.072478	0.072478
14	0.007620	0.005310	0.010854	0.019969	0.037244	0.072478	0.072478
15	0.008173	0.005797	0.010993	0.019969	0.037244	0.072478	0.072478

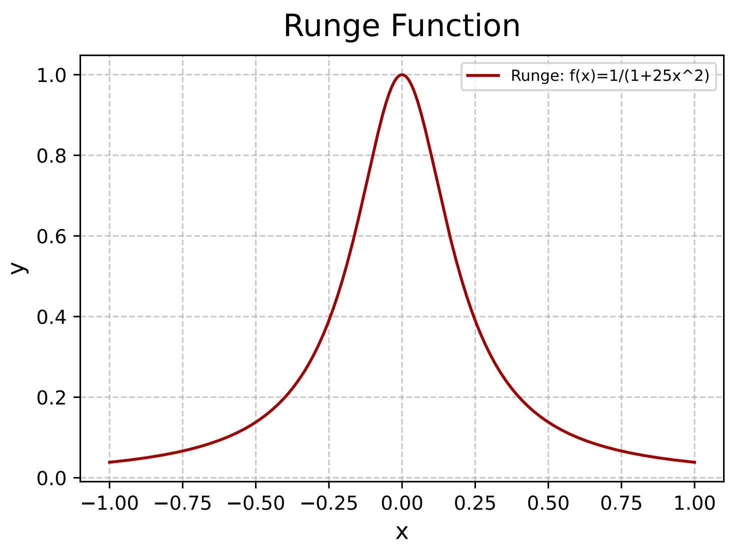

Figure 18: Runge's Function $f(x) = 1/(1 + 25x^2)$

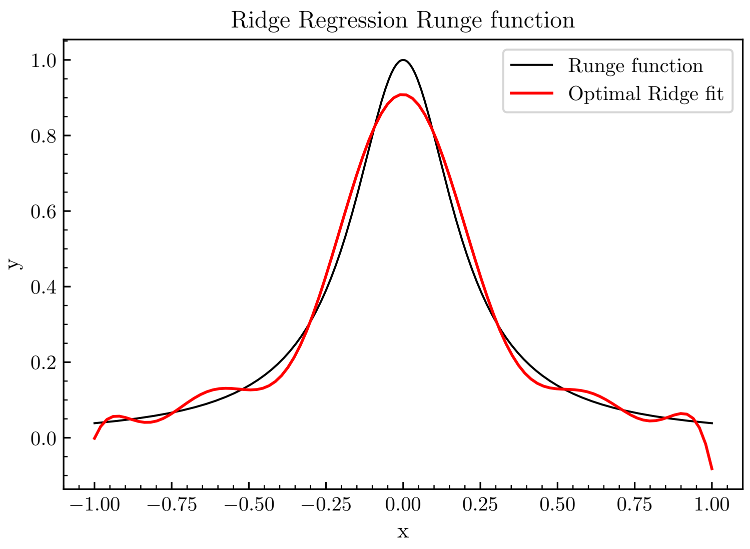

Figure 19: Ridge regression model with optimal parameters: Polynomial degree = 13, regularization parameter $\lambda = 10^{-5}$

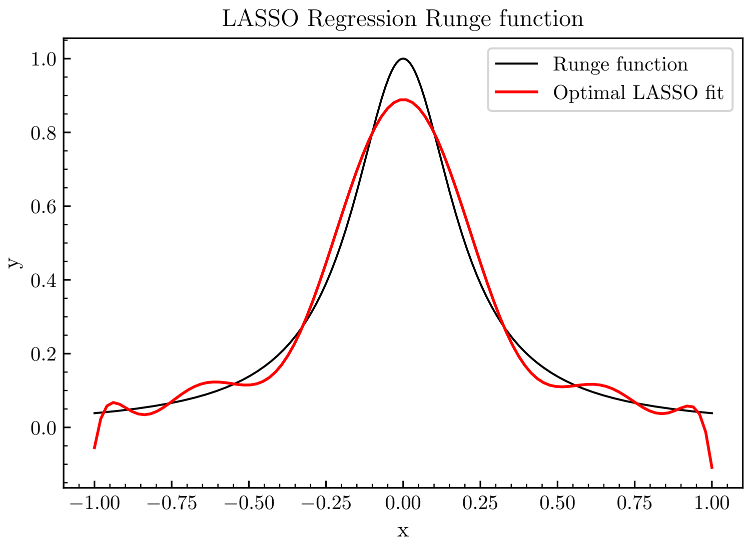

Figure 20: LASSO regression model with optimal parameters: Polynomial degree = 10, regularization parameter $\lambda = 10^{-5}$

B. Gradient descent

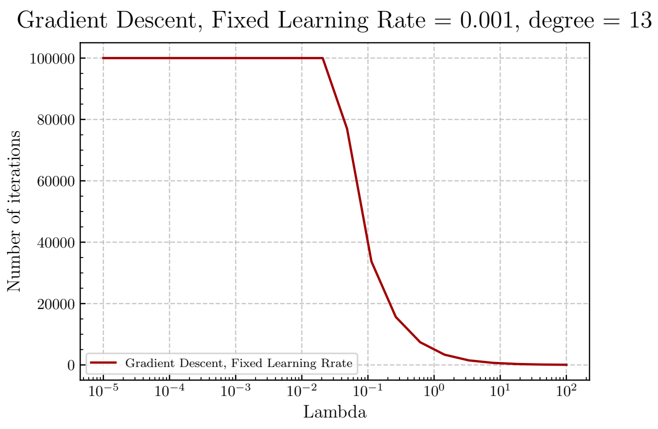

Figure 21: Number of iterations until convergence, standard gradient descent, Ridge regression

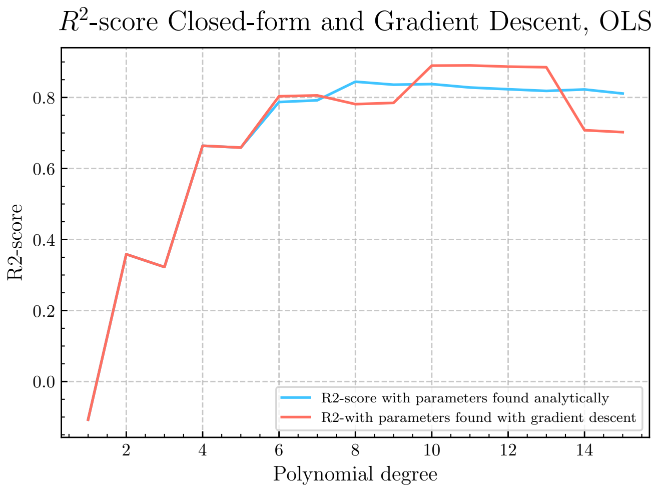

Figure 22: R^2 comparison between closed-form and standard GD for OLS.

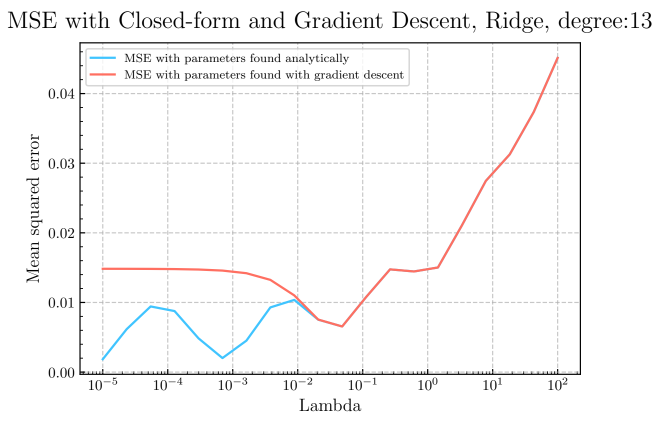

Figure 23: Comparison of MSE from closed-form and GD solutions, Ridge

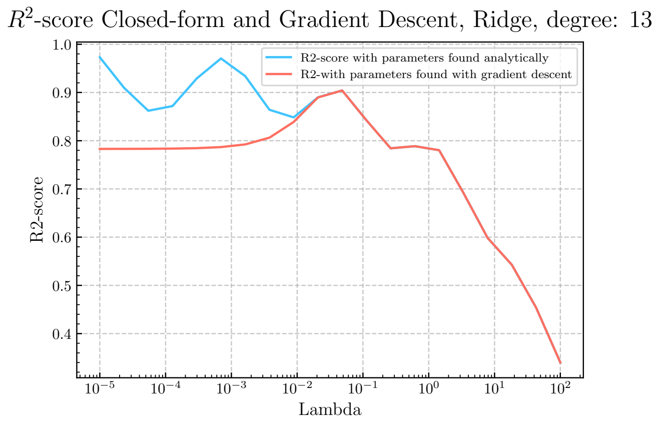

Figure 24: R^2 comparison between closed-form and standard GD method for Ridge.

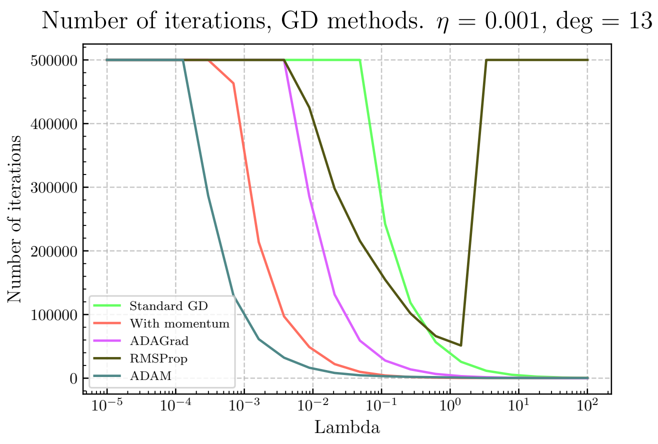

Figure 25: Number of iterations needed as a function of model complexity, Ridge regression.

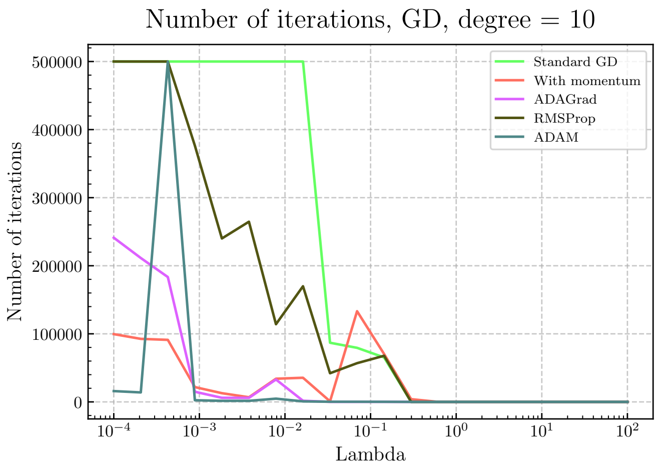

Figure 26: Number of iterations needed as a function of model complexity, LASSO regression.

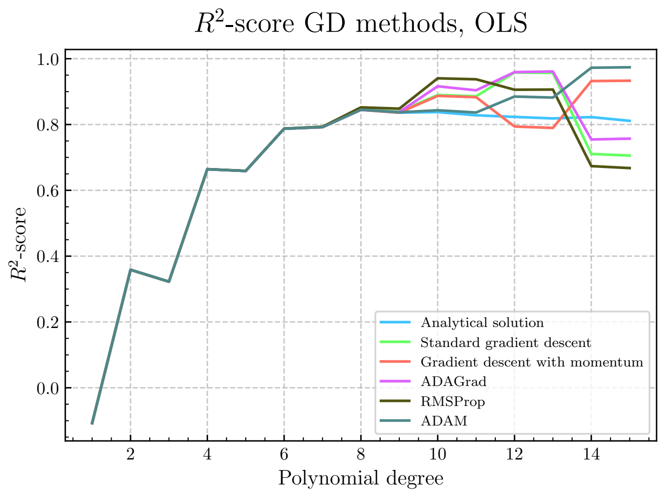

Figure 27: R^2 comparison between closed-form and GD methods for OLS.

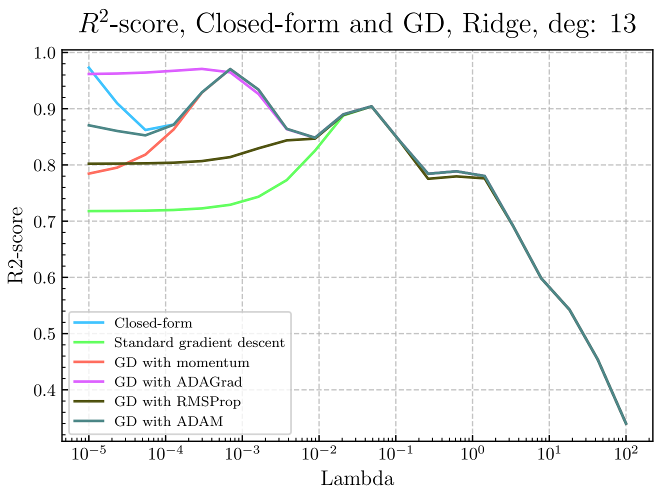

Figure 28: R^2 comparison between closed-form and GD methods for Ridge.

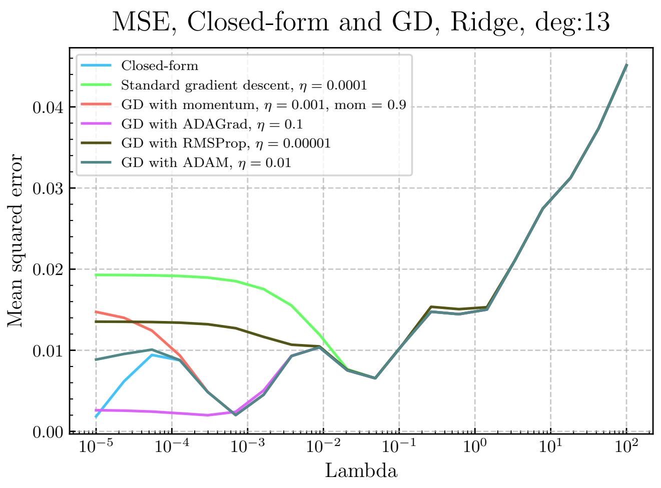

Figure 29: Gradient descent with Ridge: Increasing the regularization yields more accurate numerical approximations with gradient descent methods.

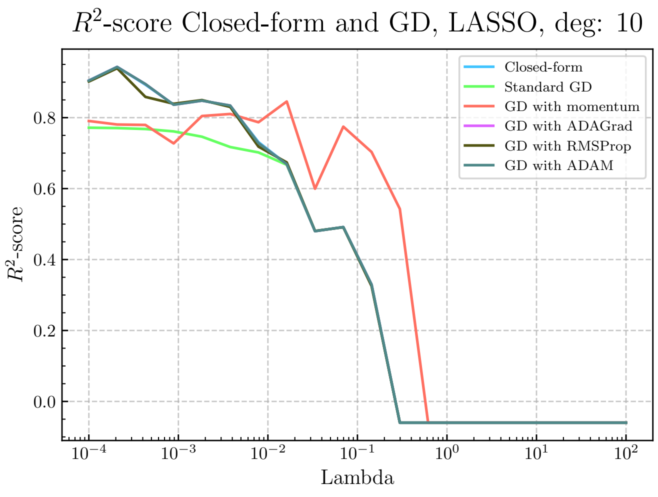

Figure 30: Gradient descent with LASSO: R^2 comparison between Scikit-Learn and GD methods

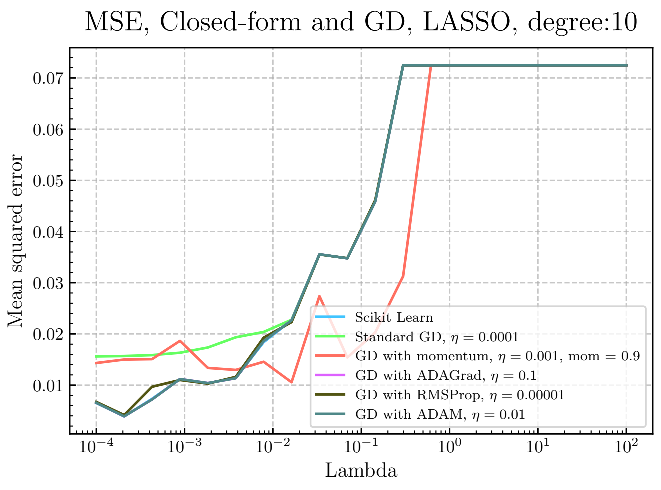

Figure 31: Gradient descent with LASSO: comparing MSEs for different regularizations.

C. Stochastic Gradient Descent

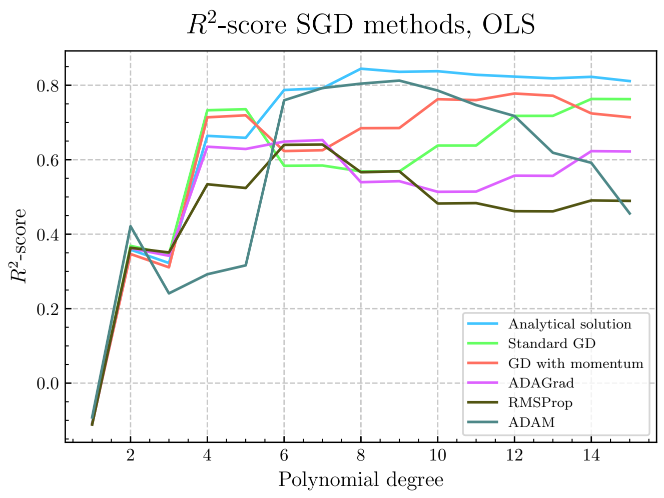

Figure 32: R^2 comparison between closed-form parameters and SGD parameters, OLS

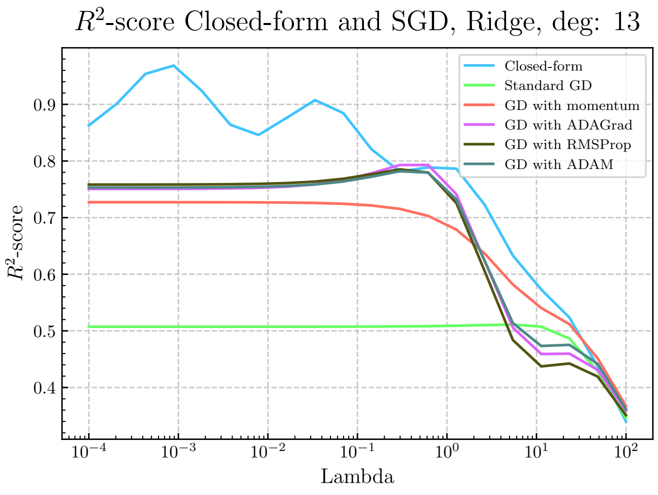

Figure 33: R^2 comparison between closed-form parameters and SGD parameters, Ridge

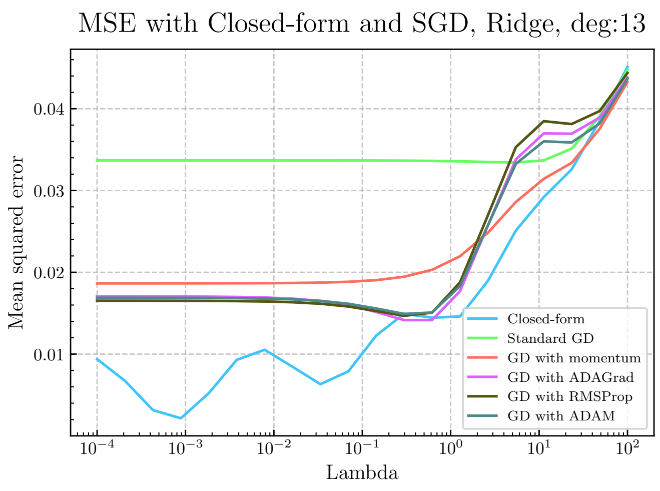

Figure 34: Stochastic gradient descent with Ridge: MSE comparison across polynomial degrees

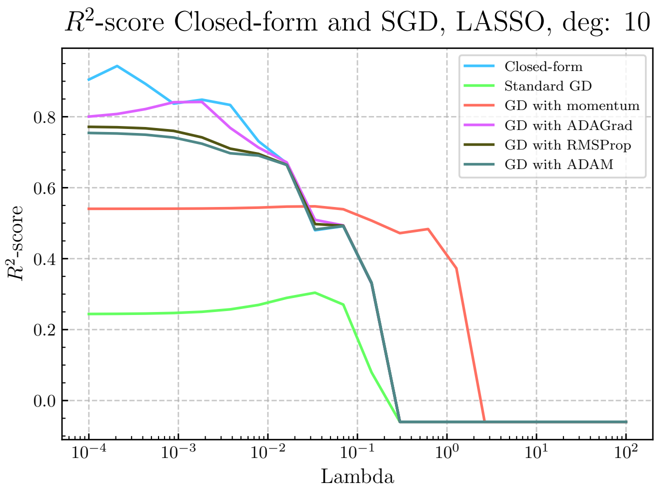

Figure 35: R^2 comparison between closed-form parameters and SGD parameters, LASSO

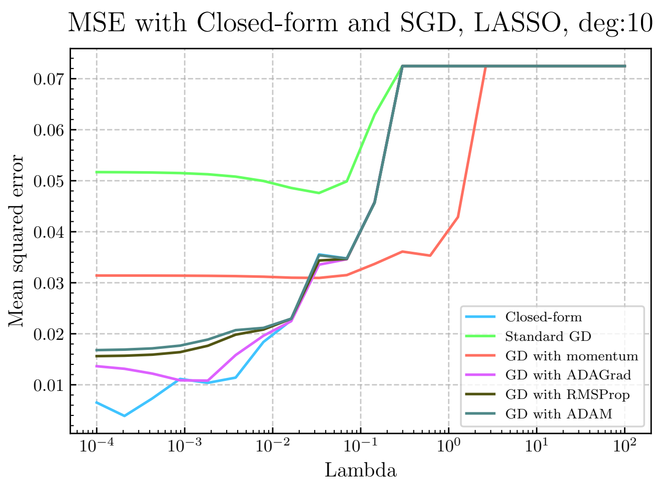

Figure 36: Stochastic gradient descent with LASSO: MSE comparison across polynomial degrees

D. Pseudo-code Algorithms

Algorithm 5 Gradient Descent with Momentum

```
1: Input: learning rate $\eta$, momentum parameter $\beta$, number of iterations $T$, feature matrix $\boldsymbol{X}$, target values $\boldsymbol{y}$
2: Initialize parameters $\boldsymbol{\theta}^{(0)}$ and velocity vector $\boldsymbol{v}^{(0)} = \mathbf{0}$
3: for $t = 1$ to $T$ do
4:     Compute gradient $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$
5:     Update velocity:
           $$\boldsymbol{v}^{(t)} = \beta \boldsymbol{v}^{(t-1)} + \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$$
6:     Update parameters:
           $$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \boldsymbol{v}^{(t)}$$
7: end for
8: Output: optimized parameters $\boldsymbol{\theta}^{(T)}$
```

Algorithm 6 AdaGrad Gradient Descent

```
1: Input: learning rate $\eta$, number of iterations $T$, feature matrix $\boldsymbol{X}$, target values $\boldsymbol{y}$
2: Initialize parameters $\boldsymbol{\theta}^{(0)}$ and accumulator $\boldsymbol{r}^{(0)} = \mathbf{0}$
3: Set small constant $\epsilon = 10^{-7}$ to avoid division by zero
4: for $t = 1$ to $T$ do
5:     Compute gradient $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$
6:     Update accumulated squared gradients:
           $$\boldsymbol{r}^{(t)} = \boldsymbol{r}^{(t-1)} + \left(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})\right)^2$$
7:     Compute adaptive learning rate:
           $$\boldsymbol{\eta}^{(t)} = \frac{\eta}{\sqrt{\boldsymbol{r}^{(t)}} + \epsilon}$$
8:     Update parameters:
           $$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \boldsymbol{\eta}^{(t)} \odot \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$$
                                   ▷ $\odot$ denotes element-wise multiplication.
9: end for
10: Output: optimized parameters $\boldsymbol{\theta}^{(T)}$
```

Algorithm 7 RMSProp Gradient Descent

```
1: Input: learning rate $\eta$, decay rate $\rho$, number of iterations $T$, feature matrix $\boldsymbol{X}$, target values $\boldsymbol{y}$
2: Initialize parameters $\boldsymbol{\theta}^{(0)}$ and running average $\boldsymbol{v}^{(0)} = \mathbf{0}$
3: Set small constant $\epsilon = 10^{-8}$ to avoid division by zero
4: for $t = 1$ to $T$ do
5:     Compute gradient $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$
6:     Update running average of squared gradients:
           $$\boldsymbol{v}^{(t)} = \rho \boldsymbol{v}^{(t-1)} + (1-\rho)\left(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})\right)^2$$
7:     Compute adaptive step:
           $$\boldsymbol{s}^{(t)} = \frac{\eta}{\sqrt{\boldsymbol{v}^{(t)}} + \epsilon} \odot \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$$
8:     Update parameters:
           $$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \boldsymbol{s}^{(t)}$$
                                   ▷ $\odot$ denotes element-wise multiplication.
9: end for
10: Output: optimized parameters $\boldsymbol{\theta}^{(T)}$
```

Algorithm 8 ADAM Gradient Descent

```
1: Input: learning rate $\eta$, decay rates $\rho_1$, $\rho_2$, number of iterations $T$, feature matrix $\boldsymbol{X}$, target values $\boldsymbol{y}$
2: Initialize parameters $\boldsymbol{\theta}^{(0)}$, first moment vector $\boldsymbol{s}^{(0)} = \mathbf{0}$, second moment vector $\boldsymbol{r}^{(0)} = \mathbf{0}$
3: Set small constant $\epsilon = 10^{-8}$ to avoid division by zero
4: for $t = 1$ to $T$ do
5:     Compute gradient $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$
6:     Update biased first moment estimate:
           $$\boldsymbol{s}^{(t)} = \rho_1 \boldsymbol{s}^{(t-1)} + (1-\rho_1)\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})$$
7:     Update biased second moment estimate:
           $$\boldsymbol{r}^{(t)} = \rho_2 \boldsymbol{r}^{(t-1)} + (1-\rho_2)\left(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(t-1)})\right)^2$$
8:     Compute bias-corrected moment estimates:
           $$\hat{\boldsymbol{s}}^{(t)} = \frac{\boldsymbol{s}^{(t)}}{1-\rho_1^t}, \qquad \hat{\boldsymbol{r}}^{(t)} = \frac{\boldsymbol{r}^{(t)}}{1-\rho_2^t}$$
9:     Update parameters:
           $$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \eta \frac{\hat{\boldsymbol{s}}^{(t)}}{\sqrt{\hat{\boldsymbol{r}}^{(t)}} + \epsilon}$$
10: end for
11: Output: optimized parameters $\boldsymbol{\theta}^{(T)}$
```

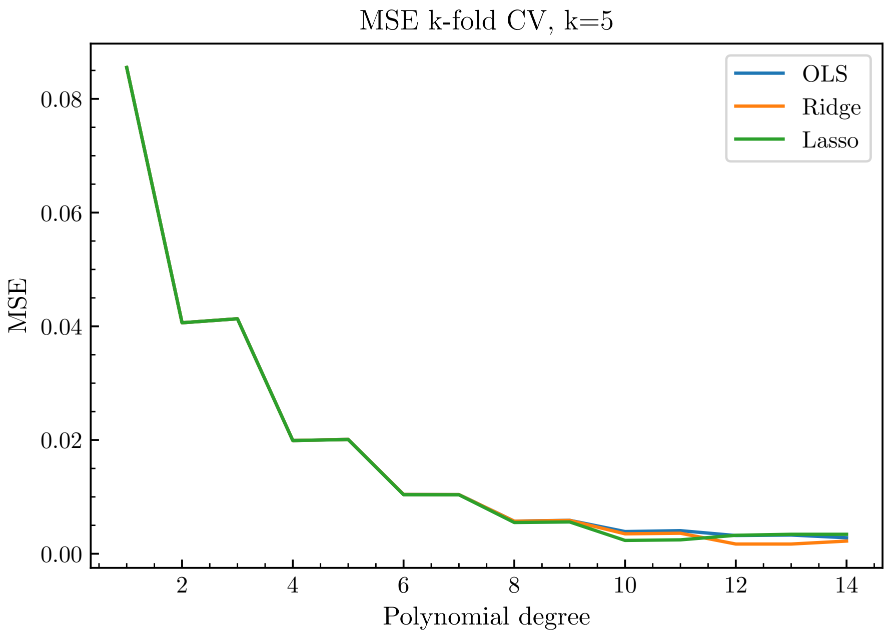

Figure 37: Mean squared error across different model complexities using 5-fold cross-validation. The plot shows the training and validation error trends, with the optimal model complexity indicated at the minimum validation error.

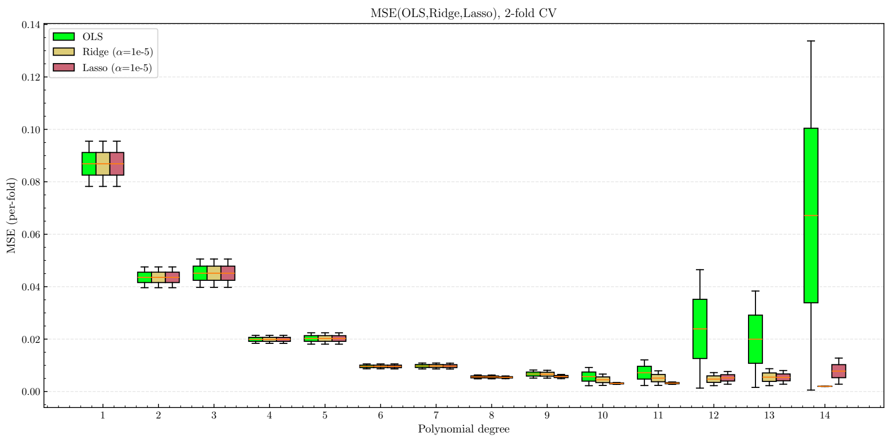

Figure 38: Distribution of mean squared errors for low model complexity ($k = 2$). Box plots compare the variability of training and validation errors across cross-validation folds, illustrating potential under-fitting behaviour with high bias.

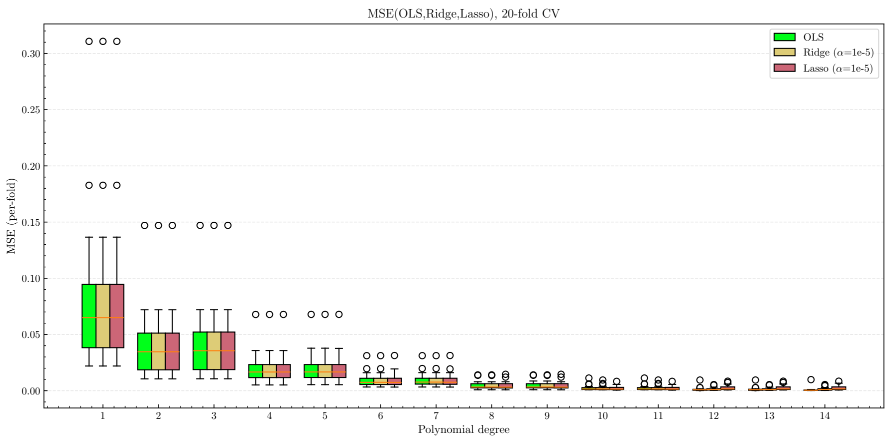

Figure 39: Distribution of mean squared errors for high model complexity ($k = 20$). Box plots reveal the increased variability in validation error compared to training error, demonstrating over-fitting characteristics with high variance.