Analyzing Different Regression and Resampling Methods

Stan Daniels, Francesco Minisini, Teresa Ghirlandi, and Carolina Ceccacci
University of Oslo
Data Analysis and Machine Learning (FYS-STK3155/FYS4155)
(Dated: October 6, 2025)

Regression in machine learning is a fundamental technique for predicting outcomes based on input features. It finds relationships between variables so that predictions on unseen data can be made. A major challenge arises as model complexity increases: low-degree models may underfit, while high-degree polynomial regression can become unstable and overfit the data. In this project, we study the Runge function, a well-known function that highlights the difficulties of high-degree polynomial interpolation. We apply Ordinary Least Squares, Ridge, and Lasso regression, complemented by gradient descent and its variants, including momentum, Adagrad, RMSprop, and ADAM. Resampling techniques such as bootstrap and cross-validation are used to evaluate model generalization and analyze the bias-variance trade-off. The dataset was synthetically generated with controlled noise, and all implementations were validated and tested against analytical and scikit-learn references. A computational benchmark was also performed to assess efficiency in terms of execution time, iteration behaviour, and memory usage. The results show that OLS fits become highly unstable for high polynomial degrees, while Ridge and Lasso regularization significantly improve stability and predictive accuracy. Gradient descent methods reproduce the analytical results, though their performance depends strongly on learning-rate strategies. Moreover, the benchmark results confirm that closed-form solvers outperform iterative optimizers by orders of magnitude in speed and memory usage for the chosen problem scale. Overall, the study highlights the importance of regularization and resampling for controlling overfitting and improving the reliability of regression models.

I. INTRODUCTION

Regression is a cornerstone in data analysis, machine learning, and scientific modeling, enabling the approximation of relationships between variables and facilitating predictions. As datasets become larger and larger, choosing the right regression method is crucial, as it directly affects model accuracy, stability, and interpretability[1]. Understanding the performance of different regression techniques is therefore a central problem in statistical learning and computational physics [2]. In this project, various regression methods are investigated, such as Ordinary Least Squares, Ridge Regression, and Lasso Regression. It focuses on fitting polynomials to a specific one-dimensional function, the Runge function:

$$\frac{1}{1+25x^2}, \quad x \in [-1,1]$$

The Runge function serves as a classical example that illustrates the challenges of high-degree polynomial interpolation, such as oscillations near the boundaries, known as Runge's phenomenon[3]. This makes it an ideal test case to compare the performances of the different methods. First, an OLS regression analysis is performed, exploring the dependence on the number of data points and the degree of polynomial. The analysis is then extended to Ridge and Lasso regressions, which add a regularization parameter λ. Gradient descent methods are implemented. The analysis starts with the standard gradient descent method, but then, to improve efficiency and convergence, several variants of the gradient descent have been developed, such as momentum, stochastic gradient descent and adaptive methods, including Adagrad, RMSprop, ADAM. The performance of OLS, Ridge and Lasso is then compared with the gradient descent-based optimization methods. In order to evaluate model performance and investigate bias-variance trade-off, resampling techniques such as bootstrap and cross-validation are applied, highlighting how different choices of model complexity and regularization affect the trade-off between bias and variance. These techniques provide insight into the stability of the models and the reliability of their predictions. Overall, this project aims to illustrate the strengths and the limitations of each method.
The structure of this project is as follows:

- Section II "Methods", describes the preprocessing and scaling of the data, the regression techniques and the optimization algorithms, as well as the resampling methods. It also includes the AI tools used.
- Section III "Results and Discussion", presents the numerical results, compares the computational performance of the different methods and discusses their implications in terms of bias-variance trade-off.
- section IV "Conclusion", summarizes the main results and the insights gained from the methods studied.

II. METHODS

Let $\mathbf{y} \in \mathbb{R}^n$ denote the vector of target values and $\mathbf{X} \in \mathbb{R}^{n \times p}$ the design matrix containing p predictors for n observations. The following linear model is assumed:

$$\mathbf{y} = \tilde{\mathbf{y}} + \boldsymbol{\epsilon}, \quad \text{with} \quad \tilde{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta},$$

where $\tilde{\mathbf{y}}$ represents the predictions of the model, $\boldsymbol{\theta} \in \mathbb{R}^p$ is the vector of unknown coefficients to be estimated and $\boldsymbol{\epsilon}$ is a vector of errors, typically assumed to be independent and identically distributed with zero mean and variance σ^2.

The goal of regression is to find an estimate of the optimal parameter $\boldsymbol{\theta}$ that best explains the observed data according to a chosen criterion.

A detailed description of the methods follows.

A. Scaling Of The Data

Before applying any regression and optimization methods, the dataset was split into a training and testing dataset and scaled appropriately. This was performed using the `split_scale` and `polynomial_features_scaled` functions.
First the training and test sets were seperated. The input feature x was then standardized using statistics computed on the training set: each feature column was centered by subtracting its mean and scaled by dividing by its standard deviation. This process produces features with zero mean and unit variance [4]. The same mean and standard deviation from the training set were applied to scale the test set, avoiding any data leakage.
For polynomial regression, each feature was expanded into polynomial terms up to a maximum degree. After generating the polynomial features, each column (except the intercept) was scaled using the mean and standard deviation of the corresponding column from the training set, via the `polynomial_features_scaled` function. This ensures that all polynomial features are on a comparable scale across training and test sets.

Scaling was necessary for several reasons.
First, high-degree polynomial terms can grow very large, causing numerical instability in the design matrix. Second, regularization methods like Ridge and Lasso assume that all features are on a comparable scale, otherwise the penalty terms distort model behavior. Third, gradient-based optimizers converge faster and more stably when the features are standardized. Finally, applying training-set statistics to scale the test set ensures unbiased evaluation.

B. Ordinary Least Squares

Ordinary Least Squares (OLS) is the classical method for linear regression and it estimates $\boldsymbol{\theta}$ by minimizing the mean squared error[4]. This is the cost function that is going to be optimize:

$$C(\boldsymbol{\theta}) = \frac{1}{n}\left\{(\boldsymbol{y} - \boldsymbol{X\theta})^T (\boldsymbol{y} - \boldsymbol{X\theta})\right\}$$

It means that it's required that the derivative with respect to $\boldsymbol{\theta}$ be set equal to zero:

$$\frac{\partial C(\boldsymbol{\theta})}{\partial \boldsymbol{\theta_j}} = 0 = \mathbf{X}^\top(\mathbf{y} - \mathbf{X\theta})$$

$$\mathbf{X}^\top \mathbf{y} = \mathbf{X}^\top \mathbf{X\theta}$$

For a full-rank design matrix, this has the closed-form solution:

$$\hat{\boldsymbol{\theta}}_{\text{OLS}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

OLS provides a simple solution, suitable when the features are few and not highly correlated.

1. Implementation

The inputs of the OLS function are a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$ and a vector of targets $\mathbf{y} \in \mathbb{R}^n$. The output is the vector of the parameters $\boldsymbol{\theta} \in \mathbb{R}^p$, whose fomula is given above.
The function is implemented in Python, using the NumPy library for efficient numerical computations. The function `np.linalg.pinv` is used to compute the pseudoinverse of the matrix $\mathbf{X}^\top \mathbf{X}$, which is particularly useful when $\mathbf{X}^\top \mathbf{X}$ is not invertible. This ensures numerical stability.

C. Ridge regression

A regularization parameter λ can be introduced by defining a new cost function to be optimized[4], that is:

$$C(\boldsymbol{\theta}) = \frac{1}{n}||\boldsymbol{y} - \boldsymbol{X\theta}||_2^2 + \lambda||\boldsymbol{\theta}||_2^2$$

where the second term represents an L_2 penalty on the size of the coefficients. This leads to the Ridge regression minimization problem where it is required that $|\boldsymbol{\theta}\|_2^2 \leq t$, where t is a finite positive number. One of the main motivations behind Ridge is its ability to resolve the problem of non-invertibility of $\mathbf{X}^\top \mathbf{X}$, which often arises when features are highly correlated. Ridge regression resolves this problem by adding the parameter $\boldsymbol{\lambda}$ to the diagonal of $\mathbf{X}^\top \mathbf{X}$ before inverting it. Taking the derivatives with respect to $\boldsymbol{\theta}$ the optimal parameters are obtained:

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

with $\mathbf{I}$ being a $p \times p$ identity matrix.

1. Implementation

The inputs of the Ridge function are a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$, a vector of targets $\mathbf{y} \in \mathbb{R}^n$, the regularization parameter λ and the intercept. The output is the vector

of the parameters $\boldsymbol{\theta} \in \mathbb{R}^p$, whose fomula is given above. The function uses the same function `np.linalg.pinv` to compute the pseudoinverse of the matrix $\mathbf{X}^\top\mathbf{X} + \boldsymbol{\lambda}\mathbf{I}$, ensuring numerical stability.

D. Lasso regression

Here the regularization term is based on the L_1 norm of the parameters[4]. The cost function is defined as

$$C(\boldsymbol{\theta}) = \frac{1}{n}||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}||_2^2 + \lambda \|\boldsymbol{\theta}\|_1,$$

where $\|\boldsymbol{\theta}\|_1 = \sum_j |\theta_j|$. This formulation leads to the Lasso minimization problem where it is required that $\|\boldsymbol{\theta}\|_1 \leq t$, with t being a finite positive number.

Unlike Ridge regression, taking the derivatives with respect to $\boldsymbol{\theta}$ does not lead to an analytical solution. The optimization problem can however be solved by using iterative gradient descent methods.

The key feature of Lasso lies in its ability to shrink some estimated coefficients $\hat{\theta}_j$ exactly to zero. When this happens, the corresponding predictor is completely removed from the model. In contrast, Ridge regression never eliminates variables: it only shrinks the coefficients $\hat{\theta}_j$ towards zero but keeps all predictors in the model. Typically, Lasso Regression is preferred when the goal is to simplify the model and improve interpretability, especially when there are a lot of features. On the other hand, Ridge regression is better for handling multicollinearity among features.

1. *Implementation*

Lasso regression was implemented using subgradient descent. The routine is very similar to the general gradient descent, but the L_1 regularization term requires the use of subgradient descent. Starting from an initial guess $\boldsymbol{\theta}_0$, the parameters are then updated acording to:

$$\boldsymbol{\theta}^{(n+1)} = \boldsymbol{\theta}^{(n)} - \eta \nabla_{\boldsymbol{\theta}} C_{\text{Lasso}}(\boldsymbol{\theta}^{(n)}) \tag{1}$$

where $\eta > 0$ is the learning rate, and the subgradient of the Lasso cost function is

$$\nabla_{\boldsymbol{\theta}} C_{\text{Lasso}}(\boldsymbol{\theta}) = -\frac{2}{n} X^\top (y - X\boldsymbol{\theta}) + \lambda \,\text{sign}(\boldsymbol{\theta}). \tag{2}$$

After each update, the subgradient is recomputed until convergence.

Algorithm 1 Subgradient Descent for Lasso

1: Initialize $\boldsymbol{\theta}^{(0)}$
2: **for** $n = 0, 1, 2, \ldots$ until convergence **do**
3: Compute subgradient: $\nabla_{\boldsymbol{\theta}} C_{\text{Lasso}}(\boldsymbol{\theta}^{(n)}) = -\frac{2}{n} X^\top (y - X\boldsymbol{\theta}^{(n)}) + \lambda \,\text{sign}(\boldsymbol{\theta}^{(n)})$
4: Update parameters:$\boldsymbol{\theta}^{(n+1)} = \boldsymbol{\theta}^{(n)} - \eta \nabla_{\boldsymbol{\theta}} C_{\text{Lasso}}(\boldsymbol{\theta}^{(n)})$
5: **end for**

E. Gradient descent

Although OLS and Ridge regression have analytical solutions, such solutions are not always available in general, so a numerical approach is often needed to optimize the same cost function.

Consider the cost function $C(\boldsymbol{\theta})$ that has to be minimized with respect to the parameters $\boldsymbol{\theta}$[5]. A second-order Taylor expansion around a point $\boldsymbol{\theta}_n$ is performed:

$$\begin{aligned} C(\boldsymbol{\theta}) \approx C(\boldsymbol{\theta}_n) &+ (\boldsymbol{\theta} - \boldsymbol{\theta}_n)^\top \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) \\ &+ \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_n)^\top \mathbf{H}(\boldsymbol{\theta}_n)(\boldsymbol{\theta} - \boldsymbol{\theta}_n) \end{aligned} \tag{3}$$

where $\mathbf{H}(\boldsymbol{\theta}_n)$ is the Hessian matrix of second derivatives at $\boldsymbol{\theta}_n$.

Neglecting the second-order term (or assuming it is costly to compute), a first-order approximation gives the update rule in the direction of the steepest descent:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n),$$

where $\eta > 0$ is a learning rate controlling the step size.

This iterative procedure moves the parameters towards the minimum of $C(\boldsymbol{\theta})$. For convex functions such as the mean squared error in linear regression, convergence is guaranteed if η is chosen appropriately.

1. *Implementation*

The different gradient descent methods were implemented inside the general routine `Gradient_descent_advanced`, in which a specific method can be selected using, for example, `method='rmsprop'` for RMSprop and so on.
In the vanilla gradient descent, the iteration start from an initial guess $\boldsymbol{\theta}^{(0)}$, the parameters are updated iteratively according to the rule:

$$\boldsymbol{\theta}^{(n+1)} = \boldsymbol{\theta}^{(n)} - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(n)}),$$

where $\eta > 0$ is the learning rate and $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(n)})$ is the gradient of the cost function with respect to $\boldsymbol{\theta}$ at iteration t. After each update, the algorithm recomputes the cost function and its gradient until a convergence criterion is met.

Algorithm 2 Gradient Descent

1: Initialize $\boldsymbol{\theta}^{(0)}$
2: **for** $n = 0, 1, 2, \ldots$ until convergence **do**
3: Compute gradient $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(n)})$
4: Update parameters: $\boldsymbol{\theta}^{(n+1)} = \boldsymbol{\theta}^{(n)} - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(n)})$
5: **end for**

F. Stochastic Gradient Descent (SGD)

Gradient descent can be computationally expensive when applied to large datasets. Stochastic Gradient Descent (SGD) addresses this by updating the model parameters using a single (or small batch of) data point(s) per iteration, rather than the full dataset[6]. This introduces noise in the updates but significantly reduces computation time. The update rule is as follows:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \boldsymbol{\nabla} l_{i_n}(\boldsymbol{\theta_t}),$$

where i_n is a randomly selected data point and η is the learning rate.

While Stochastic Gradient Descent is faster and more memory-efficient than full-batch gradient descent, it introduces noise in the parameter updates, which can make convergence less stable and less precise. However, this stochasticity can be beneficial, as it can help escape poor local minima and explore the optimization landscape more effectively, making it well suited for large datasets. [6]

1. *Implementation*

In the implementation of Stochastic Gradient Descent (SGD), the parameter vector $\boldsymbol{\theta}$ is initialized to zero and updated iteratively using gradients computed on randomly selected data points or mini-batches, rather than the entire dataset. This reduces the computational cost per update and introduces stochasticity into the optimization trajectory, which can help escape shallow local minima. The batch size is a tunable parameter: when set to 1, the algorithm uses a single sample at each iteration; larger batch sizes can be used to balance the trade-off between noise and computational efficiency.

Algorithm 3 Stochastic Gradient Descent

1: Initialize $\boldsymbol{\theta}^{(0)}$
2: **for** $n = 0, 1, 2, \ldots$ until convergence **do**
3: Randomly select a data point or mini-batch i_n
4: Compute stochastic gradient: $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} \ell_{i_n}(\boldsymbol{\theta}^{(n)})$
5: Update parameters: $\boldsymbol{\theta}^{(n+1)} \leftarrow \boldsymbol{\theta}^{(n)} - \eta\, \mathbf{g}$
6: **end for**

G. Momentum

The learning rate η plays a crucial role in the convergence and a limitation of these methods is the fixed learning rate η:

- if η is too large, the updates can overshoot the minimum, causing oscillations or divergence
- if η is too small, convergence is very slow

Moreover, for a function with steep directions and flat directions, a single global η may be inappropriate: steep coordinates require a smaller step size to avoid oscillation and flat coordinates could use a larger step to speed up progress. In order to mitigate this problem, gradient descent with momentum is introduced: it refers to a method that smoothens the optimization trajectory by adding a term that helps the optimizer remember the past gradients[6]. Mathematically, let $\boldsymbol{m}_n$ denote the velocity (or accumulated gradient) at iteration n. The update rules for gradient descent with momentum are:

$$\boldsymbol{m}_{n+1} = \beta\, \boldsymbol{m}_n + (1-\beta) \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n),$$

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta\, \boldsymbol{m}_{n+1}.$$

where $\beta \in [0, 1)$ is the momentum coefficient, which controls how much of the past gradients are remembered in the current update. A value close to 1 means the optimizer will have more inertia while a value closer to 0 means less reliance on past gradients. This mechanism enables the algorithm to suppress oscillations along steep directions while simultaneously accelerating progress across flatter regions of the cost surface. The result is a convergence process that is both faster and more stable than standard gradient descent.
This leads to more advanced optimization methods which use an adaptive learning rate for each parameter that depends on the history of gradients.[6]

1. *Implementation*

In addition to the parameter vector $\boldsymbol{\theta}$, a velocity vector $\mathbf{m}$ of the same size is initialized to zero. At each iteration the velocity is updated as a moving average of past gradients, scaled by the momentum coefficient β, and the parameters are updated using this velocity. The default value $\beta = 0.9$ was used in the experiments, providing a balance between stability and acceleration. Convergence is checked by comparing the change in parameters with a tolerance threshold.

Algorithm 4 Gradient Descent with Momentum

1: Initialize $\boldsymbol{\theta}^{(0)}$, velocity $\mathbf{m}^{(0)} = 0$
2: **for** $n = 0, 1, 2, \ldots$ until convergence **do**
3: Compute gradient $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(n)})$
4: Update velocity: $\mathbf{m} \leftarrow \beta \mathbf{m} + (1-\beta)\mathbf{g}$
5: Update parameters: $\boldsymbol{\theta}^{(n+1)} \leftarrow \boldsymbol{\theta}^{(n)} - \eta \mathbf{m}$
6: **end for**

H. Adagrad

Adagrad adapts the learning rate for each parameter based on the historical squared gradients, giving smaller

updates to frequently updated parameters and larger updates to infrequent ones[6]. This is particularly useful for sparse data.

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \frac{\eta}{\sqrt{\boldsymbol{v}_n} + \epsilon} \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n).$$

where v_n is the sum of squared gradients up to time n and ϵ prevents division by zero.
The accumulation of squared gradients can lead to excessively small learning rates over time, slowing down convergence. [7]

1. Implementation

An accumulator vector $\mathbf{G}$ of the same size as $\boldsymbol{\theta}$ is initialized to zero. At each iteration, the squared gradient is added to $\mathbf{G}$ and used to scale the learning rate for each parameter. To ensure numerical stability, a small constant $\epsilon = 10^{-8}$ is added before taking the square root. This prevents division by zero and is standard in Adagrad implementations.

Algorithm 5 Adagrad

Initialize $\boldsymbol{\theta}^{(0)}$, accumulator $\mathbf{G}^{(0)} = 0$
for $n = 0, 1, 2, \ldots$ until convergence **do**
  Compute gradient $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(n)})$
  Update accumulator: $\mathbf{G} \leftarrow \mathbf{G} + \mathbf{g}^2$
  Update parameters: $\boldsymbol{\theta}^{(n+1)} \leftarrow \boldsymbol{\theta}^{(n)} - \eta\, \mathbf{g}/(\sqrt{\mathbf{G}} + \epsilon)$
end for

I. RMSprop

RMSprop addresses Adagrad's excessive shrinkage of the learning rate by using an exponentially decaying average of past squared gradients instead of a simple sum[6]. This prevents the learning rate from shrinking too much and slowing down. If the running average is

$$v_n = \beta v_{n-1} + (1 - \beta) \left(\nabla C(\theta_n)\right)^2,$$

with decay rate β 0.9 or 0.99,
the new parameter update is

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{v_n + \epsilon}} \nabla C(\theta_n).$$

On the other hand, RMSprop requires tuning of the decay rate β and the base learning rate η. [7]

1. Implementation

Instead of storing the full sum of squared gradients, an exponentially decaying moving average is maintained in a vector $\mathbf{S}$ initialized to zero. At each iteration, $\mathbf{S}$ is updated using the decay rate ρ (set to 0.9 in our experiments), and the gradient is rescaled accordingly. As with Adagrad, a small constant $\epsilon = 10^{-8}$ is added for stability. This prevents excessively small denominators when gradients are close to zero.

Algorithm 6 RMSprop

Initialize $\boldsymbol{\theta}^{(0)}$, accumulator $\mathbf{S}^{(0)} = 0$
for $n = 0, 1, 2, \ldots$ until convergence **do**
  Compute gradient $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(n)})$
  Update moving average: $\mathbf{S} \leftarrow \rho \mathbf{S} + (1 - \rho)\mathbf{g}^2$
  Update parameters: $\boldsymbol{\theta}^{(n+1)} \leftarrow \boldsymbol{\theta}^{(n)} - \eta, \mathbf{g}/(\sqrt{\mathbf{S}} + \epsilon)$
end for

J. Adam

Adam (Adaptive Moment Estimation) combines momentum and RMSprop, maintaining both an exponentially decaying average of past gradients (first moment) and squared gradients (second moment), with bias correction[6]. It adapts the learning rate per parameter, providing stability, while incorporating momentum, accelerating the convergence. So, combining the moving averages of momentum

$$m_n = \beta_1 m_{n-1} + (1 - \beta_1) \nabla_\theta C(\theta_n)$$

and of the squared gradients

$$v_n = \beta_2 v_{n-1} + (1 - \beta_2)(\nabla_\theta C(\theta_n))^2,$$

and correcting bias

$$\hat{m}_n = \frac{m_n}{1 - \beta_1^n}, \quad \hat{v}_n = \frac{v_n}{1 - \beta_2^n},$$

the update rule in Adam becomes

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{\hat{v}_n} + \epsilon} \hat{m}_n,$$

where ϵ prevents division by zero. Adam can sometimes converge to slightly worse minima than SGD in certain tasks, and hyperparameter tuning is still required. [7]

1. Implementation

Two vectors of the same size as $\boldsymbol{\theta}$ are initialized: the first moment estimate $\mathbf{m}$ and the second moment estimate $\mathbf{v}$, both starting at zero. At each iteration, these estimates are updated with exponential moving averages of the gradient and squared gradient. Bias correction is applied using the current iteration counter t. Default hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ were used, consistent with common practice. This implementation therefore combines momentum and RMSprop into a single adaptive optimizer.

Algorithm 7 Adam

Initialize $\boldsymbol{\theta}^{(0)}$, $\mathbf{m}^{(0)} = 0$, $\mathbf{v}^{(0)} = 0$, timestep $t = 0$
for $n = 0, 1, 2, \ldots$ until convergence **do**
  $t \leftarrow t + 1$
  Compute gradient $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}^{(n)})$
  Update moments: $\mathbf{m}, \mathbf{v}$
  Apply bias correction to $\mathbf{m}, \mathbf{v}$
  Update parameters: $\boldsymbol{\theta}^{(n+1)}$
end for

K. Gradient descents comparison

Stochastic Gradient Descent (SGD) is appreciated for its simplicity and memory efficiency, particularly when tackling large datasets. However, it sometimes oscillates and may converge slowly.
Momentum builds upon SGD by accelerating convergence in directions of consistent descent, although it can occasionally bypass the lowest point in regions where the cost function curves steeply.
Adagrad's strength lies in its adaptability to sparse data, but it tends to reduce the learning rate over time, which can stall progress.
RMSprop was introduced to counter Adagrad's diminishing learning rate, making it effective for online and non-stationary tasks where data characteristics frequently change.
Adam is more often the preferred method because it merges the advantages of momentum and adaptive learning rates, generally performing robustly in most deep learning scenarios, though it may require careful parameter tuning to reach optimal performance[7].

L. Bias-variance trade-off and resampling techniques

The bias-variance trade-off is a fundamental concept in machine learning and statistics that describes the relationship between a model's complexity, the accuracy of its predictions, and how well it can make predictions on previously unseen data that were not used to train the model[8]. Bias is the error due to approximating a complex real-world problem with a simplified model:

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

It's the squared difference between the true function f(x) and the average prediction $\mathbb{E}[\tilde{y}]$ of the model over different training sets. High bias leads to underfitting, where the model is too simple (often linear) and cannot capture the underlying patterns, resulting in large errors on both training and test data.
Variance measures the sensitivity of a model to small changes in the training data:

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

It's the expected squared deviation of the model's predictions from its average prediction. High variance leads to overfitting, where the model fits the training data very closely but performs poorly on unseen data, showing large test errors.

More formally, consider a dataset $\{(x_i, y_i)\}_{i=1}^n$, where the true data is generated by a noisy model $y = f(x) + \varepsilon$ and ε is the noise term with zero mean and variance σ^2. The true function $f(x)$ is approximated by a model $\tilde{y}$, which depends on parameters $\boldsymbol{\theta}$ and a design matrix X, determined by minimizing the mean squared error (MSE):

$$\text{C}(\boldsymbol{X}, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \tilde{y}\right)^2 = \mathbb{E}\big[(y - \tilde{y})^2\big]$$

The expected prediction error can be decomposed as

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

where σ^2 is the variance of the errore ϵ.
To derive this equation, it should be recalled that the variance of $\boldsymbol{y}$ and $\boldsymbol{\epsilon}$ are both equal to σ^2. The mean value of $\boldsymbol{\epsilon}$ is by definition equal to zero. Furthermore, the function f is not a stochastics variable, idem for $\tilde{\boldsymbol{y}}$. Using a more compact notation in terms of the expectation value

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

and adding and subtracting $\mathbb{E}\left[\tilde{\boldsymbol{y}}\right]$ one obtains

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

which, using the abovementioned expectation values can be rewritten as

$$\mathbb{E}\left[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2\right] = \mathbb{E}\left[(\boldsymbol{y} - \mathbb{E}\left[\tilde{\boldsymbol{y}}\right])^2\right] + \text{Var}\left[\tilde{\boldsymbol{y}}\right] + \sigma^2 \qquad (4)$$

that is the rewriting in terms of the bias, the variance of the model $\tilde{\boldsymbol{y}}$ and the variance of $\boldsymbol{\epsilon}$. In order to derive this equation, the assumption that is made is that the unknown function $\boldsymbol{f}$ can be replaced by the target data $\boldsymbol{y}$.
This decomposition underlies the bias-variance trade-off: a high bias and low variance situation corresponds to a model that is too simple and underfits the data, while a low bias and high variance situation corresponds to a model that is too complex and overfits the data. The goal is to find a balance, achieving a model that is complex enough to capture the underlying patterns but simple enough to generalize well to new data.

Resampling techniques provide the practical tool for measuring and managing this balance, by reliably estimating the generalization ability of the model. They involve repeatedly drawing samples from a training set

and refitting a model of interest on each sample in order to obtain additional information about the fitted model. Two resampling techniques are discussed here: bootstrap and cross-validation.

Bootstrap is a resampling technique that involves repeatedly drawing samples with replacement from the original dataset to create multiple "bootstrap" samples[8]. Each bootstrap sample is used to fit the model, and the variability of the model's predictions across these samples provides an estimate of the model's variance. This method is particularly useful for estimating the uncertainty of model parameters and assessing the stability of the model.

Cross-validation is a technique used to assess how the results of a statistical analysis will generalize to an independent dataset[8]. The most common form is k-fold cross-validation, where the dataset is divided into k subsets (or "folds"). The model is trained on k-1 folds and validated on the remaining fold. This process is repeated k times, with each fold serving as the validation set once. The average performance across all k trials provides a robust estimate of the model's generalization error. Cross-validation helps in selecting model parameters and in preventing overfitting by ensuring that the model performs well on unseen data.

M. Use of AI tools

In this project, AI tools such as ChatGPT and GitHub Copilot were utilized to improve the efficiency and quality of the work. These tools assisted in finding useful material, generating code snippets, debugging and they were also particularly helpful in drafting sections of the report and suggesting improvements. However, all outputs generated by these AI tools were carefully reviewed and modified.

III. RESULTS AND DISCUSSION

The performance of the regression models are evaluated using both the MSE and the score R^2. The definitions of these are the following:

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

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

where n is the number of data points, y_i are the true values, $\tilde{y}_i$ are the predicted values, and $\bar{y}$ denotes the mean of the true values.

How the reported results were generated (averaging, randomness, and fairness)

Unless otherwise stated, every result displayed in this section is the mean over $N_{\text{runs}} = 30$ independent repetitions. Each run uses a new noisy dataset and a new train/test split, but within a run all models and methods see the same data and splits to enable fair, paired comparisons.

Data generation and scaling. For a given n, we create a uniform mesh of Runge function $f(x) = \frac{1}{1+25x^2}$ that maps the interval $(-1, 1)$. To the mesh we added Gaussian noise

$$y = f(x) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 0.3^2)$$

. We split the data into train/test with a 67%/33% split and standardize features using training statistics only (the intercept, when present, is not scaled). For polynomial models, features are expanded up to degree d and then scaled column-wise using the training means/stds and re-used on the test set.

Reproducibility and fairness. Let s_0 be the global base seed. For each run index $r = 0, \ldots, N_{\text{runs}} - 1$, the run seed is defined as

$$s_r = s_0 + r,$$

as both the NumPy global seed before generating noise and the random_state for the train/test split. This guarantees that, within the same run, all methods share the exact same dataset and splits. Additional randomness is handled as follows:

- **SGD III E:** to make mini-batch selections identical across models within a run and degree, the RNG is reset to a deterministic [9], per-degree seed before each optimizer call (thus the same mini-batch sequence is used by all methods being compared).
- **Bootstrap III F:** for each run, bootstrap resamples of the training set are drawn with sklearn's resample using fixed per-bootstrap seeds [10]; predictions are evaluated on the fixed test set of that run.
- **Cross-Validation III G:** for each run a single KFold is instantiate with `shuffle=True` and `random_state` $= s_r$ [11], so folds are identical across degrees and models within the run.

What is averaged. For all parts of the project, each metric displayed at a given degree d is the mean test metric over N_{runs}. For each average metric displayed is also computed its standard deviation. For part III F, the MSE, bias^2, and variance are computed from bootstrap predictions within each run and then averaged across runs. For part III G, per-run degree-d scores are the mean MSE across $k = 5$ folds.

Validation and testing

To ensure the correctness and robustness of all implemented methods, a testing suite was developed. Each function within the project was validated against scikit-learn's out-of-the-box implementations, ensuring consistency with established numerical baselines. The test suite employs the Pytest framework and compares analytical, gradient-based and resampling functions through deterministic and quantitative checks. All tests were designed to be reproducible, relying on a fixed random seed to guarantee identical data splits and noise realizations across runs. The seed policy used is consistent with what is described in the previous paragraph III.

For the implemented preprocessing routines, the generated design matrices were compared element-wise with those produced by scikit-learn's classes, verifying identical scaling and intercept behavior (results asserted with a numerical tolerance of rtol $= 10^{-12}$).

Metric functions such as the Mean Squared Error (MSE) and the coefficient of determination (R^2) were also cross-checked against scikit-learn's corresponding functions, ensuring numerical equivalence up to machine precision (rtol $= 10^{-12}$, atol $= 10^{-12}$).

Analytical regression solvers like OLS and Ridge Regression were validated by comparing their estimated coefficients and predictions to those obtained from scikit-learn under identical configurations. For Ridge regression, both penalized and unpenalized intercept versions were tested to confirm consistency with scikit-learn's regularization scheme. The detailed numerical tolerances used in these validations are summarized in Table I.

TABLE I: Analytical solver validation results

Model	**Ref.**	**rtol**	**atol**
OLS	`LinearReg.` (`fit_int=False`)	10^{-10}	10^{-12}
Ridge	`Ridge` ($\alpha = \lambda$)	10^{-10}	10^{-10}

The custom gradient descent optimizer was evaluated by verifying that its converged parameters coincide with the analytical OLS solution within a tolerance of 10^{-5}, demonstrating the correctness of both gradient computations and convergence logic. As shown in the following studies, the convergence speed of these methods are sensitive to the chosen hyperparameters and the type of regression problem, for that reason it is worth to mention that the methods were tested with polynomials of degrees $d_1 = 3$ and $d_2 = 4$, with the respective learning rates $\eta_1 = 0.05$ and $\eta_2 = 0.02$, while having a fixed number of iterations of $n_{iter} = 5 \times 10^4$. A full summary of these quantitative checks is reported in Table II.

TABLE II: Gradient Descent validation results

Check	**rtol**	**atol**
Parameters vs. OLS	–	10^{-5}
Predictions vs. OLS	10^{-6}	10^{-8}
MSE gap	–	$\leq 10^{-8}$

For LASSO regression, predictions and coefficient vectors were compared to `sklearn.linear_model.Lasso` under equivalent configurations, confirming numerical agreement up to small regularization-dependent deviations. The tolerances and bounds applied during this validation are reported in Table III.

TABLE III: LASSO regression validation summary

Quantity	**Tolerance**	**Error Bound**
Predictions	$r : 10^{-4}, a : 10^{-5}$	–
Coefficients (L^1 dist.)	–	$\leq 10^{-3}$

The bias–variance decomposition implementation was internally validated by checking the analytical identity 4 using bootstrap-generated predictions, confirming maximum absolute discrepancies across polynomial degrees below 10^{-10}. Cross-validation consistency was also verified by enforcing identical `KFold` partitions across models and degrees, with matching CV scores within rtol $= 10^{-10}$, atol $= 10^{-12}$. These resampling-based validation results are summarized in Table IV.

TABLE IV: Resampling-based validation checks

Procedure	**rtol**	**atol**
Bias–variance identity	–	$\leq 10^{-10}$
Cross-val. scores	10^{-10}	10^{-12}

All assertions passed successfully, confirming that the entire computational pipeline—from data scaling to analytical and iterative solvers—produces results consistent with standard machine learning libraries and theoretical expectations.[12]

A. OLS

The analysis starts with performing a standard ordinary least squares regression using polynomials in x up to order 15. The dependence of the polynomial degree and number of data point has been explored by plotting the mse as a function of polynomial degree for different amount of data points. This can be seen in Figure 1.

As one can see, generally the mse goes down as the polynomial degree increases. But for a small amount of data points like $n = 40$ and $n = 50$, when the polynomial degree goes to high the mse starts to increase

again.
This happens because of overfitting. When the polynomial degree increases the model tries to fit the data too perfectly and will fit the underlying random noise as well and the model will then perform poorly of unseen data. For a high number of data points this is barely noticeable but for a lower amount of data points the effect will become more pronounced.

When plotting the parameters θ as a function of polynomial degree, it can be observed that for low-degree polynomials the coefficients remain relatively stable, whereas for higher-degree polynomials they increase rapidly in magnitude and exhibit pronounced oscillations. This behavior reflects overfitting: at high degrees, the model captures both the true signal and the noise, causing the coefficients to become unstable. Using more data points reduces this effect and stabilizes the parameter estimates.

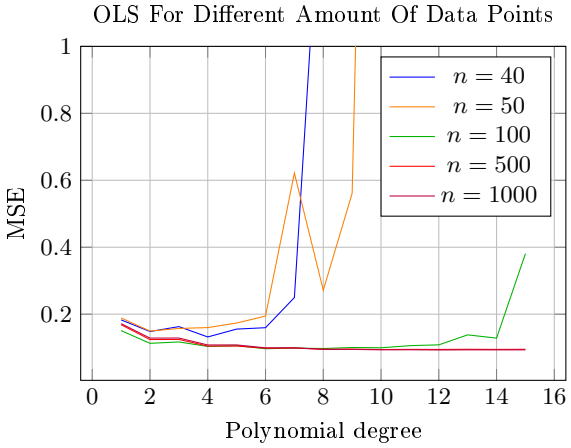

FIG. 1: Test MSE vs. polynomial degree for different dataset sizes n. Small n (40, 50) leads to overfitting at high degrees, while larger n (100, 500, 1000) stabilizes MSE, showing improved generalization.

B. Ridge regression

Now, Ridge regression is applied to the same dataset, exploring the impact of different regularization parameters λ. First, the effect of the regularization parameter λ on the MSE as a function of polynomial degree was analyzed for a fixed number of data points n=100: the results are shown in Figure 2. Small values of λ (e.g., 0.001, 0.01) behave similarly to OLS, with the MSE decreasing as polynomial degree increases. However, as λ increases (e.g., 1, 10, 100), the MSE curve flattens and even increases for high polynomial degrees, indicating that strong regularization prevents overfitting but may lead to underfitting if too large. Among the tested values, λ =0.01 produces the lowest MSE for most polynomial degrees, suggesting an optimal trade-off between bias and variance.

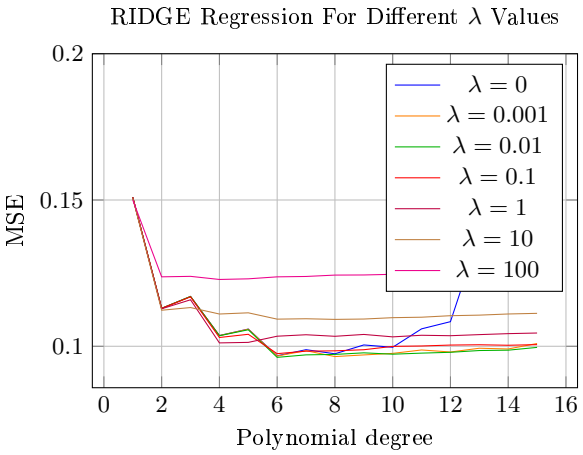

FIG. 2: Test MSE vs. polynomial degree for Ridge regression with varying regularization parameter λ (n = 100).

Next, $\lambda = 0.01$ is selected to plot the MSE as a function of polynomial degree for different numbers of data points, allowing a direct comparison with the standard OLS regression, as shown in Figure 3. The comparison reveals that while both methods achieve low MSE at moderate polynomial degrees, OLS exhibits a sharp increase in MSE at high degrees due to overfitting. In contrast, Ridge regression maintains a lower and more stable MSE across all degrees, demonstrating its effectiveness in controlling model complexity through regularization. This highlights Ridge's advantage in scenarios where overfitting is a concern, particularly with high-degree polynomials. As one can see, the mse decreases as the polynomial degree increases, similar to OLS.

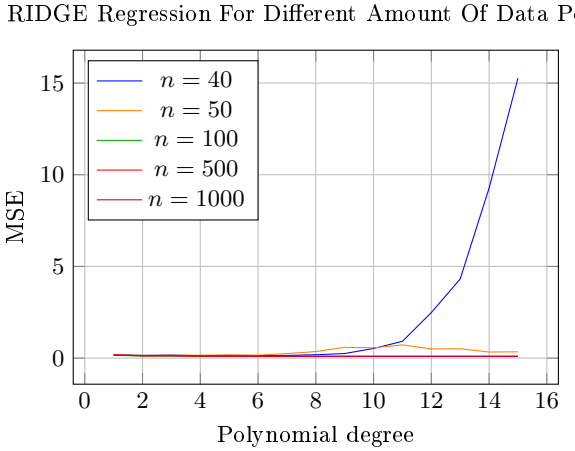

FIG. 3: Test MSE vs. polynomial degree for varying dataset sizes n with regularization parameter $\lambda = 0.01$.

The parameters θ as a function of polynomial degree for Ridge regression are plotted. Compared to OLS, the

coefficients remain more stable even at high polynomial degrees. This stability indicates that Ridge's L_2 regularization effectively constrains the magnitude of the coefficients, preventing them from becoming excessively large and reducing overfitting. The result is a more robust model that generalizes better to unseen data.

C. Gradient Descent Methods

The analytical expression for paramters θ is now compared to the results obtained using gradient descent methods.

First, the standard gradient descent method is applied to both OLS and Ridge regression, exploring the impact of different learning rates η. The results for OLS and Ridge with varying learning rates are shown in the figures below and it is evident that the gradient descent is very sensitive to the choice of the learning rate.
For example, when applying gradient descent to OLS with a learning rate equal to 0.01, as shown in the Figure 4, the method converges to the analytical solution for polynomial degrees up to 15, with the MSE closely matching the analytical results. For smaller learning rates, the algorithm does not reach the analytical solution within the set number of iterations, resulting in a higher MSE. For learning rates larger than 0.01, the algorithm diverges, failing to converge to the analytical solution and causing the MSE to increase dramatically.

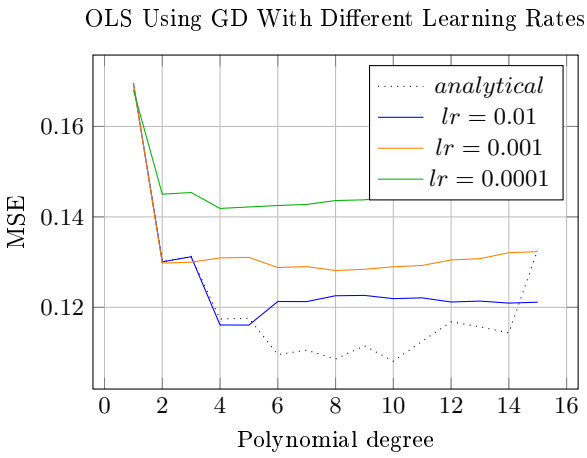

FIG. 4: Test MSE of OLS regression vs. polynomial degree for the analytical solution and gradient descent with step sizes $\eta = 0.01$, 0.001, and 0.0001, showing the effect of learning rate on convergence to the analytical optimum.

Similarly, for Ridge regression with $\lambda = 0.01$, as shown in the Figure 5, a learning rate of 0.01 allows convergence to the analytical solution, while smaller rates lead to slower convergence and larger MSE.

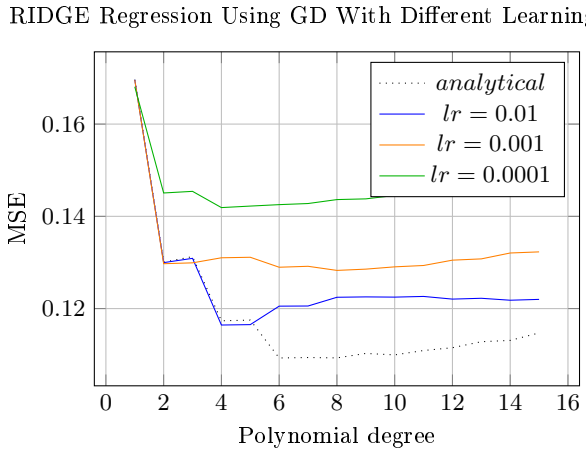

FIG. 5: Test MSE of Ridge regression vs. polynomial degree for the analytical solution and gradient descent with step sizes $\eta = 0.01$, 0.001, and 0.0001, showing the effect of learning rate on convergence to the exact solution.

In both cases, the analytical solution is more accurate than gradient descent, except for a learning rate of 0.01, where the two methods yield nearly identical results.

Next, advanced gradient descent methods are applied to both OLS and Ridge regression.
For OLS, as shown in the Figure 6, Momentum, Adagrad, RMSprop, and Adam all successfully converge to the analytical solution with MSE values closely matching the analytical results across polynomial degrees. RMSprop and Adam show better performance at higher polynomial degrees, because of their adaptive learning rate mechanisms.

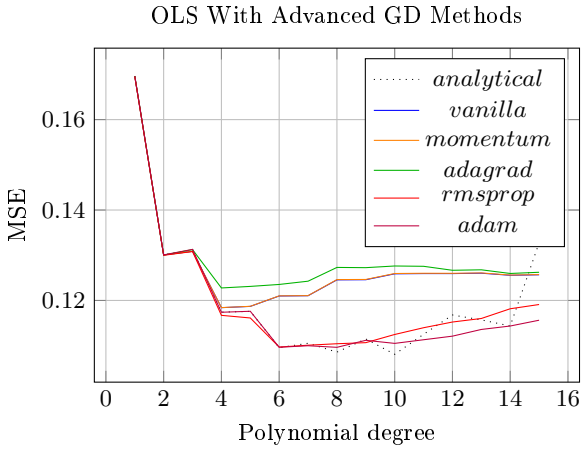

FIG. 6: Test MSE of OLS regression vs. polynomial degree for analytical solution and iterative optimizers (vanilla GD, momentum, Adagrad, RMSProp, Adam, $\eta = 0.01$), illustrating performance differences and convergence behavior.

For Ridge regression with $\lambda = 0.01$, as shown in the

Figure 7, RMSProp and Adam show a better convergence at higher degrees, but still the analytical solution is the best one. The analytical solution is superior because it provides the exact global minimum, whereas iterative methods like RMSProp and Adam are only approximations that may not reach the true optimum.

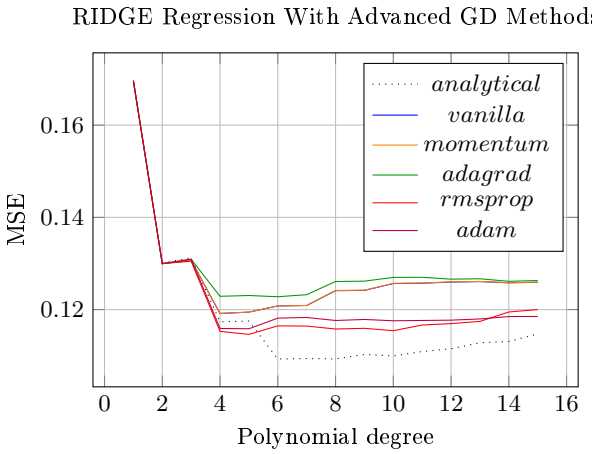

FIG. 7: Test MSE of Ridge regression vs. polynomial degree for analytical solution and iterative optimizers (vanilla GD, momentum, Adagrad, RMSProp, Adam, $\eta = 0.01$), showing differences in convergence and generalization.

D. Lasso regression

Lasso regression does not have an analytical solution, so only gradient descent methods are applied.

First, the standard gradient descent method is applied to Lasso regression, exploring the impact of different learning rates η. The results for Lasso with varying learning rates are shown in the Figure 8. With a learning rate equal to 0.01, the method converges for polynomial degrees up to 15, with the MSE being relatively low. For smaller learning rates, the algorithm converges very slowly, requiring a large number of iterations to approach the minimum.

When using more advanced gradient descent methods for Lasso regression, as shown in the Figure 9, the convergence improves significantly. Ridge and Lasso regression show similar MSE across polynomial degrees. However, the MSE values for Lasso are generally higher than those for Ridge regression, due to the nature of the L_1 penalty. This encourages many coefficients to become exactly zero, which makes the model more interpretable but increases the bias of the coefficients. On the other hand, Ridge regression shrinks coefficients towards zero but does not set them exactly to zero, leading to lower bias and lower MSE. Compared to OLS, which is unbiased but can have high variance, Lasso achieves a lower MSE by reducing the variance, even with the bias it introduces

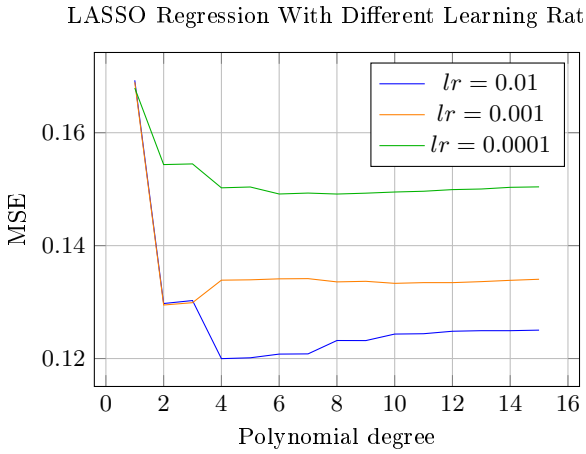

FIG. 8: Test MSE of Lasso regression vs. polynomial degree using vanilla GD for learning rates $\eta = 0.01$, 0.001, and 0.0001, showing the effect of step size on convergence and generalization.

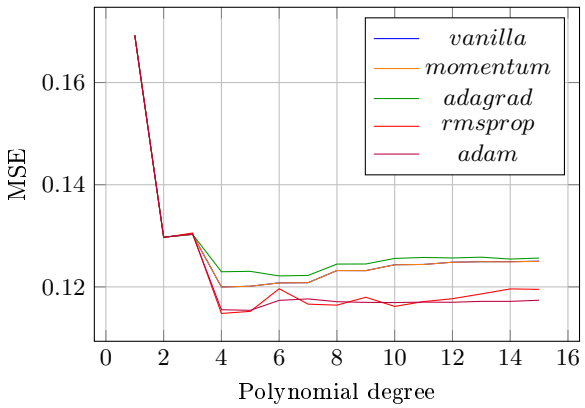

FIG. 9: Test MSE of Lasso regression vs. polynomial degree for vanilla GD, momentum, Adagrad, RMSProp, and Adam ($\eta = 0.01$, $\lambda = 0.01$), illustrating the effect of model complexity on generalization.

E. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is now applied to OLS, Ridge, and Lasso regression. For OLS, as shown in Figure 10, SGD converges to a solution with MSE values that are slightly higher than those obtained with standard gradient descent methods. This difference arises because SGD computes the gradient using individual samples or small mini-batches instead of the entire dataset. While this makes each update faster and more memory-efficient, it also introduces noise into the

updates, leading to a less stable convergence.
Ridge regression with SGD, as shown in Figure ??, exhibits a similar pattern: the MSE is slightly higher compared to batch methods due to the stochastic nature of the updates. However, the L_2 penalty contributes to more stable coefficient shrinkage, so the convergence remains reliable and generalization can even improve compared to OLS.
In the case of Lasso regression with SGD, shown in Figure ??, the optimization becomes more challenging because of the L_1 penalty, which creates flat regions and non-differentiable points at zero. Here, the stochasticity can both complicate and help: while it may cause small fluctuations and slightly higher MSE, it also allows the algorithm to escape plateaus and drive coefficients exactly to zero, reinforcing sparsity in the solution.

Overall, the effect of applying SGD to OLS, Ridge, and Lasso is fundamentally the same: it serves as a stochastic optimization technique that iteratively minimizes each model's objective function, with differences arising primarily from the penalty terms rather than from the optimization process itself.

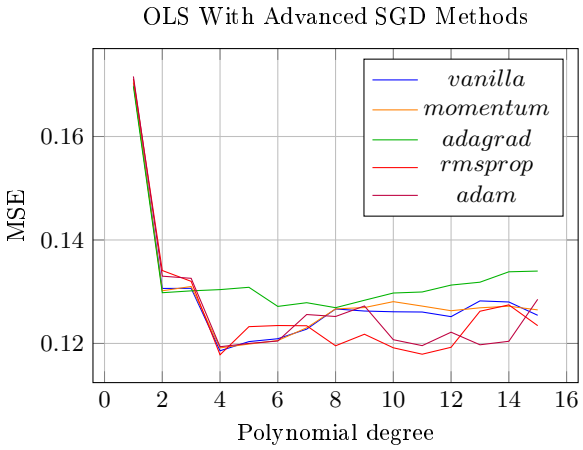

FIG. 10: Test MSE of OLS regression vs. polynomial degree using vanilla GD, momentum, Adagrad, RMSProp, and Adam ($\eta = 0.01$), showing their effect on convergence and generalization as model complexity increases.

F. Bias-Variance trade off

To illustrate the bias–variance trade-off, the Runge function is fitted using an OLS model with varying polynomial degrees. The bias, variance, and mean squared error (MSE) are computed using bootstrap resampling with 50 resamples. The results are shown in Figure 11.

For low polynomial degree the variance is nearly 0, while the bias is slightly above 0. As a result the MSE closely follows the bias, indicating that the model is simple but stable. As the degree of the polynomial increases at a certain point the variance jumps up dramatically, causing a corresponding rise in MSE. This reflects overfitting, where the model starts capturing noise in the training data rather than just the underlying signal.

The optimal model complexity lies between these extremes, balancing bias and variance to minimize prediction error. The bootstrap analysis highlights how the number of data points affects this balance: when the number of data points is low, the variance increases more rapidly with polynomial degree, leading to higher MSE, whil with more data points, the variance is reduced and the MSE stabiliez.

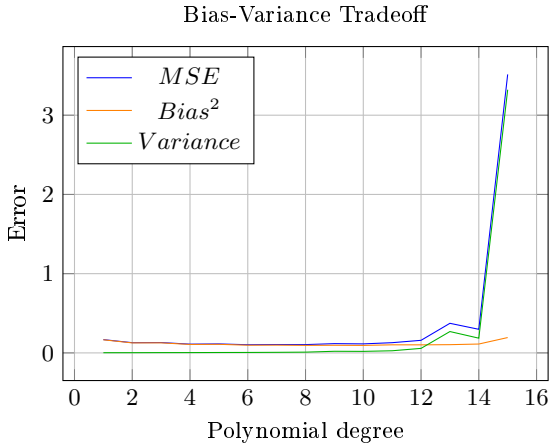

FIG. 11: Bias–variance decomposition of test MSE vs. polynomial degree ($n = 200$), showing that low-degree models have high bias and low variance, while high-degree models have low bias and increased variance, with MSE reflecting their combined effect.

G. Cross Validation

Cross-validation is now applied to evaluate the performance of OLS, Ridge, and Lasso regression models. The MSE is computed using 5-fold cross-validation for polynomial degrees ranging from 1 to 15, as shown in Figure 12. For OLS, the MSE decreases initially with increasing polynomial degree, but then rises sharply at higher degrees due to overfitting, especially with fewer data points. Ridge regression, with a regularization parameter $\lambda = 0.01$, shows a more stable MSE across polynomial degrees, effectively controlling overfitting. Lasso regression, also with $\lambda = 0.01$, exhibits a similar trend, but with slightly lower MSE than Ridge due to the stronger bias introduced by the L_1 penalty.

These results highlight the bias-variance trade-off: while OLS achieves very low bias at high degrees, its

variance becomes large. Ridge and Lasso introduce a small bias but yield more stable models that generalize better.

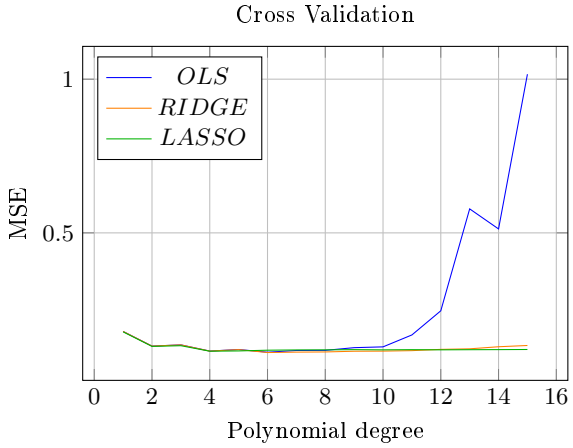

FIG. 12: Cross-validation test MSE vs. polynomial degree for OLS, Ridge, and Lasso regression, showing how regularization in Ridge and Lasso mitigates overfitting compared to OLS.

H. Computational Benchmark and Performance Analysis

In addition to evaluating model accuracy, a computational benchmark was conducted to assess the efficiency of the implemented methods. The benchmark compared analytical solutions and gradient-based variants across multiple polynomial degrees. The analysis focuses on execution time, iteration behavior, floating-point operations, and memory usage. Results were obtained following the procedures in paragraph III, averaging over $N_{runs} = 30$ for polynomials up to degree 14 fitted to a dataset of $n = 100$ points using the following parameters:

TABLE V: Benchmark configuration parameters.

Parameter	Symbol	Value
Regularization factor	λ	0.01
Learning rate	η	0.01
Iterations	N_{iter}	10,000
Tolerance	tol	10^{-8}
Momentum coefficient	β	0.9
Epsilon (for Adam stability)	ϵ	10^{-8}
Mini-batch size	B	32
Bootstrap resamples	N_{boot}	30

Execution time and efficiency. Figure 13 reports the median fit time as a function of polynomial degree. Closed-form methods (OLS and Ridge) were found to be several orders of magnitude faster than iterative optimizers. Ordinary Least Squares achieved a median fit time of approximately 0.1 ms, while Ridge regression required around 0.11 ms, confirming their efficiency for small and medium-scale problems. In contrast, vanilla gradient descent required on average more than 100 ms per fit, corresponding to a computational cost roughly 10^3–10^4 times higher. This difference reflects the analytical $\mathcal{O}(p^3)$ complexity of matrix inversion compared to the iterative $\mathcal{O}(np \times \text{iterations})$ [13, 14] scaling of gradient descent. Despite this, as the results shows, iterative optimization remains preferable when the design matrix is large, sparse, or cannot be stored entirely in memory. Usecases in which iterative gradient descent methods shine are in fact NNs, where a very large nubmber of parameters need to be optimized[7].

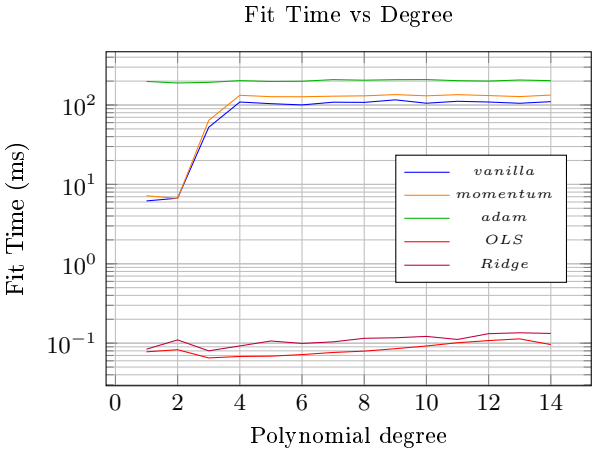

FIG. 13: Memory usage estimate of the implementation of different methods. The plot compares vanilla gradient descent, momentum, Adam.

Computational cost and memory footprint. The estimated number of floating–point operations (FLOPs) confirms the expected scaling trends (Figure 14). In closed–form solvers (OLS, Ridge) the dominant step is solving the normal equations, which scales as $\mathcal{O}(p^3)$ in the number of features p. Within the present range of degrees this remains markedly below the iterative methods: the FLOPs curve for OLS/Ridge stays in the 10^3–10^5 band, while gradient methods rise to 10^7–10^8. For gradient–based optimizers, the per–iteration cost is $\mathcal{O}(np)$, so the total work scales approximately linearly with the number of iterations T and the data size n ($\sim \mathcal{O}(npT)$). Among them, Momentum and Adam exhibit the largest computational load because each step updates extra state vectors[15, 16] (velocity m for Momentum; both m and the second–moment v for Adam), effectively adding one (Momentum) or two (Adam) additional vector operations and element–wise transforms per iteration.

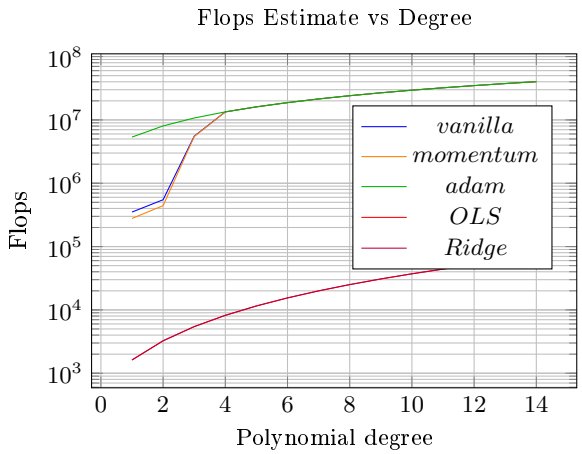

FIG. 14: Flops estimate of the implementation of different methods. The plot compares vanilla gradient descent, momentum, Adam on the number of floating point operations.

Memory usage (Figure 15) grows approximately linearly with polynomial degree, reflecting the linear growth of p and the associated parameter/state vectors. In our setup it remains below 0.02 MB across the displayed degrees for all methods. Closed–form solvers mainly store $\boldsymbol{\theta}$ and the small $p \times p$ normal matrix; adaptive optimizers add constant–factor overhead for their moment buffers (e.g., m, v), which explains the slight vertical offset between curves while preserving linear trends with degree.

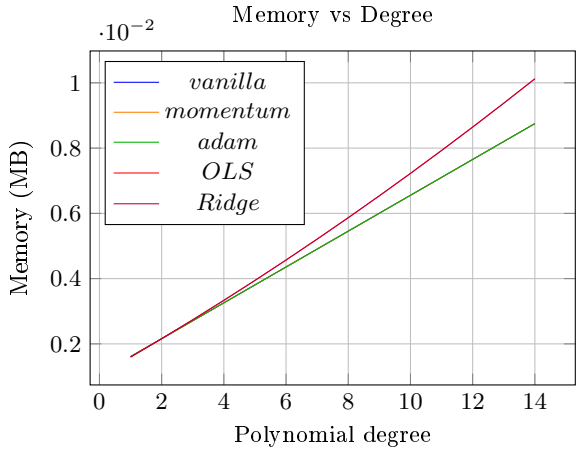

FIG. 15: Memory usage estimate of our implementation of different methods. The plot compares vanilla gradient descent, momentum, Adam ($learningrate = 0.01$).

Accuracy–Speed Pareto analysis. Finally, Figure 16 summarizes the joint accuracy–efficiency trade–off by plotting test R^2 against wall–clock fit time. Two distinct frontiers emerge[14]. Closed–form solvers occupy the leftmost Pareto frontier, achieving $R^2 \approx 0.95$–0.99 at $\mathcal{O}(10^{-1})$ ms. Iterative optimizers form a secondary frontier at $\mathcal{O}(10^{2})$ ms: vanilla GD and momentum cluster around $R^2 \approx 0.5$–0.85, while Adam attains higher accuracy ($R^2 \approx 0.95$) but at greater cost. When the design matrix fits in memory and p is moderate, closed–form OLS and Ridge remain strictly Pareto–superior. Iterative methods become attractive only for large–scale or streaming data, or when analytical solutions are unavailable.

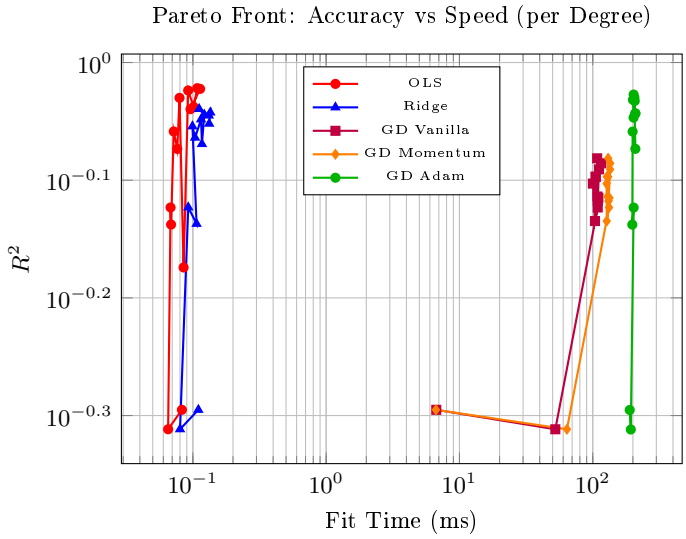

FIG. 16: Pareto analysis of model accuracy (R^2) versus computational cost (fit time in milliseconds). Lines connect successive polynomial degrees per method.

So it is clear from this results that for small and simple regression problems the same accuracy results can be achived both with closed-form and iterative GD methods, but with a higher computational efficiency, making the GD methods not ideal for these usecases.

IV. DISCUSSION AND CONCLUSION

This project analyzed different regression methods such as Ordinary Least Squares(OLS), Ridge and Lasso along with the effects of implementing gradient descent optimizers, the use of bootstrap methods and the exploration of the bias-variance trade-off.

OLS performed well for simple models but overfitted at higher polynomial degrees. Ridge reduced variance through L_2 regularization, while Lasso's L_1 penalty promoted sparsity and produced simpler, more interpretable models.

A major focus was the implementation of gradient descent methods. Standard gradient descent proved to be sensitive to the learning rate, small rates let to slow convergence while large rates caused convergence. With some tuning, the results came close the analytical solutions. Advanced gradient descent methods, such as

Momentum, Adagrad, RMSprop, and Adam improved convergence speed and numerical stability, with Adam providing the most consistent and reliable results across different polynomial degrees. The use of stochastic methods was also implemented, these showed a lower computing time but at the cost of accuracy. For Lasso regression the use of gradient descent was essential as it lacks a analytical solution.

Bootstrap and cross-validation results confirmed the bias–variance trade-off, showing that regularization helps balance underfitting and overfitting. Among the methods, Ridge and Lasso achieved the best generalization performance.

Each approach presented trade-offs: OLS is simple but unstable for complex models; Ridge improves robustness but adds bias; Lasso enhances sparsity but may remove relevant predictors.

The computational benchmark expanded the analysis by quantifying the efficiency and scalability of each method. Closed–form solvers (OLS and Ridge) demonstrated extremely low execution times (on the order of 10^{-1} ms) and minimal memory footprints, remaining several orders of magnitude faster than iterative gradient–based methods for small to medium–scale problems. Overall, the results show that analytical methods are preferable when the design matrix fits entirely in memory and the model size is moderate, while iterative optimizers become indispensable in large–scale or scenarios where analytical inversion is impractical. Therefore, it is clear from these results that for small and simple regression problems, comparable accuracy can be achieved using both closed–form and iterative methods, but with far greater computational efficiency in the analytical case, making gradient descent suboptimal in such contexts.

The study demonstrated the importance of regularization, adaptive optimization, and computational efficiency in modern regression analysis. Each approach entails a trade–off among accuracy, interpretability, and scalability: OLS remains the simplest yet least robust, Ridge offers the best generalization compromise, Lasso improves sparsity and feature selection, while gradient–based methods provide flexibility at a computational cost. Future works could extend these analyses to high–dimensional, real–world datasets and investigate hybrid optimization strategies that balance analytical precision with scalable computation.

[1] P. Mehta and D. J. Schwab, arXiv preprint (2019), pMC, accessed 2025-10-03.

[2] J. M. Aiken *et al.*, Physical Review Physics Education Research **17**, 020104 (2021).

[3] C. Runge, Zeitschrift für Mathematik und Physik **46**, 224 (1901), english translation: "On empirical functions and interpolation between equidistant ordinates".

[4] M. Hjorth-Jensen, Machine learning lecture notes, week 35, `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week35.ipynb` (2025), accessed: 2025-10-03.

[5] M. Hjorth-Jensen, Machine learning lecture notes, week 36, `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week36.ipynb` (2025), accessed: 2025-10-03.

[6] M. Hjorth-Jensen, Machine learning lecture notes, week 37, `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week37.ipynb` (2025), accessed: 2025-10-03.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) Chap. 8.1, 8.2, 8.3, `http://www.deeplearningbook.org`.

[8] M. Hjorth-Jensen, Machine learning lecture notes, week 38, `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week38.ipynb` (2025), accessed: 2025-10-03.

[9] N. Developers, numpy.random.seed, `https://numpy.org/doc/2.2/reference/random/generated/numpy.random.seed.html#numpy.random.seed` (2024), accessed: 2025-10-04.

[10] S. learn developers, Glossary: resample, `https://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html` (2024), accessed: 2025-10-04.

[11] S. learn developers, Cross-validation: evaluating estimator performance, `https://scikit-learn.org/stable/modules/cross_validation.html` (2024), accessed: 2025-10-04.

[12] This testing suite was executed continuously in a GitHub workflow, ensuring the validity of each commit and experiment.

[13] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. (Johns Hopkins University Press, 2013) classic reference for matrix inversion and least-squares computational complexity.

[14] L. Bottou, in *Proceedings of COMPSTAT* (2010) pp. 177–186, introduces computational trade-offs and scalability of SGD.

[15] B. T. Polyak, USSR Computational Mathematics and Mathematical Physics **4**, 1 (1964), original introduction of momentum method (heavy-ball algorithm).

[16] D. P. Kingma and J. Ba, International Conference on Learning Representations (ICLR) (2015), introduces Adam optimizer combining momentum and adaptive learning rates, arXiv:1412.6980 [cs.LG].