Project 1 (FYS-STK4155)

Yoan Tutunarov & Vladislav Foss & Sebastian Koranda

October 2025

Abstract

This project investigates three regression methods—Ordinary Least Squares (OLS), Ridge, and LASSO—applied to polynomial fitting of the Runge function. We implement both closed-form solutions and gradient descent approaches, and compare their performance, noting small deviations that are expected due to numerical optimization. To analyze model complexity, we study the bias–variance trade-off using bootstrap resampling, which provides estimates of bias, variance, and noise. The results show that OLS overfits at higher polynomial degrees, Ridge reduces variance by shrinking coefficients, and LASSO promotes sparsity by eliminating irrelevant terms. Overall, the study illustrates how regularization and resampling techniques improve predictive performance and provide deeper insight into the generalization properties of regression models.

Acknowledgments and AI Disclaimer

We would like to thank our supervisor for valuable feedback and discussions throughout the project. This project has also made limited use of AI tools, such as ChatGPT, as a support mechanism for improving clarity, refining explanations, and troubleshooting code. The primary analysis, problem solving, and discussion were carried out independently by the project members. AI assistance was used only to enhance language in the report, verify understanding, and used for polishing figure names, ensuring that all team members fully comprehend the underlying concepts and methods.

Introduction

In this project, we study different regression methods with a particular focus on Ordinary Least Squares (OLS), Ridge regression, and Lasso regression.

These methods form the foundation of many statistical and machine learning approaches, and they provide useful insights into the trade-off between model complexity and predictive performance. As a test case, we apply these methods to the Runge function, a one-dimensional function that is well known to illustrate the challenges of polynomial interpolation, especially at higher polynomial degrees (Runge's phenomenon). By fitting polynomials of varying complexity to this function, we investigate how different regression techniques handle overfitting, variance, and bias. The project is based on weekly exercises, lecture material, and additional references. All numerical experiments are implemented in Python, with the code provided in a separate file and linked through GitHub. This report serves as documentation of our implementation, results, and analysis.

Theory, Methods and Concepts

Mean Squared Error (MSE)

Mean Squared Error (MSE) is one of the most fundamental measures of model performance in statistics and machine learning. It quantifies how well a predictive model approximates the true data by measuring the average squared difference between the actual (target) values and the predicted values [2].

Formally, for a dataset with n samples, the MSE is defined as

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

where

- y_i denotes the true (actual) value of the response variable for observation i,
- $\hat{y}_i$ denotes the predicted value from the model for the same observation,
- n is the total number of observations.

The term $(y_i - \hat{y}_i)$ is called the residual, which represents the error between the actual and predicted values for observation i. Squaring the residual ensures that negative and positive deviations are treated equally and emphasizes larger errors. Averaging over all n samples provides a single metric that reflects the model's overall predictive accuracy.

A lower MSE indicates that the model predictions are closer to the actual values, while a higher MSE suggests larger discrepancies. Therefore, in regression problems, the objective is typically to minimize the MSE, ideally approaching zero.

$\mathbf{R^2}$ score

The R^2 score (often called the coefficient of determination) is a standard metric in regression analysis that quantifies how well the variation in the independent variables explains the variation in the dependent variable. In simpler terms, R^2 measures the proportion of the variance in the target y that is captured by the regression model, relative to a baseline model that always predicts the mean of y [10].

Formally, let y_i be the actual observed values, $\hat{y}_i$ the predicted values, and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ the sample mean. Then

$$\text{SSE} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2, \quad \text{SST} = \sum_{i=1}^{n} (y_i - \bar{y})^2,$$

and

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}} = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}.$$

Here:

- SSE ("Sum of Squared Errors") is the sum of squared deviations between predictions and actual values.
- SST ("Total Sum of Squares") is the sum of squared deviations of actual values around their mean.
- R^2 ranges (in theory) from $-\infty$ to 1. An $R^2 = 1$ indicates perfect prediction (all points lie exactly on the regression line). An $R^2 = 0$ indicates that the model is no better than the constant mean predictor. Negative R^2 occurs when the model performs worse than simply predicting the mean.

A higher R^2 implies a better-fitting model in that it reduces residual error relative to the variance of the data itself.

Regression

Regression is a statistical method used to study the relationship between a dependent variable (outcome) and one or more independent variables (predictors). It estimates how changes in predictors are associated with changes in the outcome. In its simplest form, linear regression fits a straight line through data to best explain this relationship. More complex forms include multiple regression (many predictors) and nonlinear regression (curved relationships). Regression is widely used for prediction, trend analysis, and decision-making in fields like economics, finance, and machine learning [1].

Ordinary Least Squaress (OLS)

Ordinary Least Squares (OLS) is a method used to estimate the coefficients of a linear regression model. It finds the values of the coefficients that minimize the total squared distance between the observed data points and the values predicted by the model. Mathematically, OLS solves:

$$\min_{\beta} (Y - X\beta)'(Y - X\beta)$$

where Y is the vector of observed values, X is the matrix of predictors, and β is the vector of coefficients. OLS relies on certain conditions: the errors should be independent, have constant variance, and ideally follow a normal distribution. The method can struggle when predictors are highly correlated or when the dataset is too small compared to the number of variables. Once estimated, OLS coefficients let us assess model quality (e.g., using R^2) and make predictions [3]. In Task A, we implemented our own OLS function to compute the optimal $\hat{\theta}$. From the lecture notes in Week 35, we know that the solution to the regression problem is given as $y = X\theta$, where X is the design matrix. In this assignment we used a polynomial of degree 15, meaning that $\hat{\theta}$ is a one-dimensional array with 15 parameters.

The solution for OLS is obtained by differentiating the cost function and solving for its minimum:

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

and given that $X^\top X$ is invertible, the solution becomes

$$\hat{\theta} = (X^\top X)^{-1} X^\top y.$$

Here y is given by Runge's function $y = \frac{1}{1+25x^2}$ with added noise. The noise was generated using NumPy's `np.random.normal()`, with parameters specified in the project description.

We then constructed the design matrix X and split the data into training and test sets using scikit-learn's built-in `train_test_split()` function.

Ridge regression

Ridge Regression (also called L2 regularization) is a variation of linear regression that introduces a penalty on large coefficient values in order to reduce model variance and improve stability in the presence of multicollinearity. It adjusts the ordinary least squares estimator by adding a term λI (where $\lambda \geq 0$ is the regularization parameter and I is the identity matrix) such that the coefficient estimates are given by

$$\hat{\beta}_\lambda = \left(X^\top X + \lambda I\right)^{-1} X^\top y$$

This estimator shrinks the coefficients toward zero, but unlike some other methods, does not set any coefficients exactly to zero. The choice of λ controls the bias-variance tradeoff: a larger λ increases bias but lowers variance and may improve prediction for new data. Care must be taken: if λ is too large, the model may underfit; if too small, it may behave similarly to OLS and still overfit [4].

LASSO Regression

LASSO (Least Absolute Shrinkage and Selection Operator) regression is an extension of linear regression that introduces both regularization and variable selection. It modifies the ordinary least squares (OLS) objective by adding an L_1 penalty term, which constrains the absolute values of the model coefficients. The LASSO objective function is defined as

$$\min_{\boldsymbol{\beta}} \left\{ \frac{1}{2n} \sum_{i=1}^{n} \left(y_i - \mathbf{x}_i^\top \boldsymbol{\beta}\right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\},$$

where $\lambda \geq 0$ is a regularization parameter controlling the strength of the penalty.

The L_1 regularization term encourages sparsity in the coefficient vector $\boldsymbol{\beta}$ by shrinking some coefficients exactly to zero, thereby performing feature selection. This simplifies the model and helps manage the bias–variance trade-off: as λ increases, model variance decreases while bias increases. However, if λ is too large, the model may underfit and fail to capture important data patterns.

Unlike OLS and Ridge regression, LASSO does not admit a closed-form analytical solution because the L_1 norm introduces a non-differentiable point at zero. Consequently, iterative optimization algorithms such as coordinate descent or gradient descent are required to estimate the parameters [5].

Gradient Descent Variants

In machine learning, we typically have a dataset X and a model defined by parameters θ (sometimes also denoted β). The objective is to determine the parameter values that minimize a chosen cost function $C(\theta)$. Gradient descent is an iterative optimization algorithm designed to solve this problem.

The main idea is the following: if we have a function $F(x)$, with $x = (x_1, x_2, \ldots, x_n)$, then the gradient $\nabla F(x)$ points in the direction of the steepest ascent of the function. By moving in the opposite direction of the gradient, the algorithm approaches a minimum of the cost function after taking enough steps. The general update rule for gradient descent is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta_k \nabla F(\mathbf{x}_k),$$

where the parameter η is the *learning rate*. This can be thought of as the step size taken in the chosen direction. If η is too small, convergence towards the minimum is very slow, while if η is too large, the algorithm may diverge. Choosing a suitable learning rate is therefore of great importance.

In our project, the goal is to minimize the Mean Squared Error (MSE) cost function for OLS, and MSE with an additional penalty term for Ridge regression. Minimization is performed by iteratively adjusting the model parameters θ, thereby reducing the error between the predicted values and the true outcomes [6]. We start by initializing all coefficients to zero. At each iteration, we compute the gradient of the cost function, which tells us how the error changes with respect to each parameter. The gradient indicates the slope of the loss surface: a positive gradient means the parameter should be decreased, while a negative gradient means it should be increased. Using a fixed learning rate η, we then update the coefficients according to

$$\theta \;\leftarrow\; \theta - \eta \nabla_\theta C(\theta).$$

For OLS, the gradient is

$$\nabla_\theta C_{\text{OLS}}(\theta) = \frac{2}{n} X^\top (X\theta - y),$$

while for Ridge it becomes

$$\nabla_\theta C_{\text{Ridge}}(\theta) = \frac{2}{n} X^\top (X\theta - y) + 2\lambda\theta,$$

where λ is the regularization parameter [7].

Stochastic Gradient Descent (SGD)

In the most basic version of the gradient descent, we usually include the whole dataset to compute the the gradient descent at each stop. This is in contrast to **SGD**, where we would calculte the gradient on a subset of data called mini-batches. If our whole dataset is **n** datapoints, and we want to have a mini-batch of size **M**, then the number of batches would be $\frac{\mathbf{n}}{\mathbf{M}}$.

Compared to the full batch (whole dataset) gradient descent, SGD typically will converge faster per iteration, but will however experience larger variance due to noise and the random nature of the method. A trade off with the SGD is the size of each batch: larger batches will be computationally more expensive but less noisy, and smaller batches will be very cost efficient, however will have a lot more noise in the trajectory they are calculating [9].

The gradient step of the SGD is calculated as:

$$\theta_{t+1} = \theta_t - \eta_t \nabla C_{B_k}(\theta_t),$$

Here k represents the randomly chosen mini-batch from the dataset at each iteration. One complete pass over all mini-batches is referred to as an *epoch* [9]. Calculating the gradient on subsets of the data reduces computational cost compared to full-batch gradient descent, and the inherent randomness can also help the algorithm escape shallow local minima and saddle points.

Since the learning rate plays a critical role in the performance of SGD, a variety of adaptive learning rate algorithms have been developed. These methods adjust the step size dynamically based on information from previous updates. In this project, we focus on four variants: **Momentum**, **AdaGrad**, **RMSProp**, and **Adam**, each of which introduces specific improvements to the basic GD and SGD functions.

Momentum based GD

SGD generally today is implemented with momentum. The general idea behind momentum based GD / SGD is that if the landscape we are traversing with our gradient is a canyon, the vanilla gradients will have large oscillations instead of gliding towards the minima. The implementation of momentum build on the idea that if the direction of the gradient descent aligns with the previous point, then we gain "momentum", and if not we dampen it. This way we dampen the oscillations that we might experience. This is mathematically described as:

$$\theta_{t+1} = \theta_t - v_t, \text{ where}$$
$$v_t = \gamma v_{t-1} + \eta_t \nabla E(\theta_t)$$

The function $E(\theta)$ represents our function we are trying to find the minimum of [9]. Here we can see that we will get a smoothed path where we accelerate if the gradient is still pointing in the same direction. We also introduce a new parameter γ, which represents the momentum, and we usually choose between $0 \leq \gamma \leq 1$ [9].

Adagrad

The adaptive learn rate of Adagrad drops the parameter γ represented in momentum completely and has a different approach. For this algorithm, we first define a vector:

$$r_t = r_{t-1} + (\nabla C(\theta))^2$$

Calculating the AdaGrad at step t will then become

$$\theta_{t+1} = \theta_t - \eta H_t^{-1/2} \nabla C(\theta)$$

, where we have that the matrix H_t is defined as $H_t = \text{diag}(r_t)$, where the entries in the matrtix will diagonally be entries of r_t over time. The update of θ will be:

$$\theta_{t+1,j} = \theta_{t,j} - \frac{\eta}{\sqrt{\epsilon + r_{t,j}}} \nabla C(\theta_{t,j})$$

Here we introduce the parameter ϵ as a small constant to avoid division by zero if $r_{t,j}$ is 0. From the mathematical expressions, we can see that there we have removed the momentum (γ) as implemented in momentum gradient descent described in a section above. We see that if a gradient is very large the step size we take ($\frac{\eta}{\sqrt{\epsilon + r_{t,j}}}$) will automatically become very small, meaning we have less large jumps, while if the gradient is very small the step size becomes larger and speeds up [9].

Unfortunately, a key limitation of AdaGrad is that the accumulated squared gradients in H_t grow without bound. This causes the algorithm to take very large steps in the beginning, but as the vector $r_{t,j}$ grows, the effective learning rate

$$-\frac{\eta}{\sqrt{r_{t,j} + \epsilon}}$$

becomes smaller and smaller [9]. As a result, the algorithm slows down excessively, and the step sizes eventually become infinitesimally small, effectively stopping further learning. This limitation is the main motivation behind the development of RMSProp and Adam, which modify AdaGrad to maintain a more stable learning rate [9].

RMSProp

Subsequently, the root-mean-square propagation (RMSProp) algorithm was developed to address AdaGrad's limitation of diminishing learning rates that eventually lead to extremely small step sizes. The key idea is to use an exponentially decaying moving average of past squared gradients rather than accumulating all of them. This acts as a weighted average that gives more importance to recent gradients than to older ones, preventing the learning rate from decaying towards zero over many iterations [8, 9].

The exponentially decaying average is defined as

$$v_t = \rho v_{t-1} + (1-\rho)(\nabla C(\theta_t))^2,$$

where ρ is the decay rate, typically chosen in the range $0.9 \leq \rho \leq 0.99$.

The update rule for the parameters then becomes

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla C(\theta_t),$$

where ϵ is a small constant included to avoid division by zero.

From these expressions, we see that recent gradients have greater influence than earlier ones, ensuring that the effective learning rate remains stable. This modification makes RMSProp more robust than AdaGrad and it has become a widely used optimizer in practice, as well as an important component of the Adam algorithm [8, 9].

Adam

The adaptive moment estimation (Adam) algorithm combines the ideas of RMSProp, with its adaptive learning rate, and momentum, which smooths the optimization path. Adam maintains running averages of both the first and the second moments of the gradients [9].

The first moment is defined as

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla C(\theta_t),$$

and the second moment as

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla C(\theta_t))^2,$$

where the parameters are typically chosen as $\beta_1 = 0.9$ and $\beta_2 = 0.999$, with initial values $m_0 = 0$ and $v_0 = 0$ [9].

When we first calculate the first and second moments (m_t, v_t), both are biased towards zero in the early iterations. This is because the running averages favor the most recent gradients, and thus require time to build up from zero. To correct for this bias, Adam introduces normalized versions of the moments. The bias-corrected terms are given as [9]:

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

The parameter update rule then becomes

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t,$$

where η is the learning rate and ϵ is a small constant to avoid division by zero.

From these expressions we see that in the first few iterations, when t is small, the denominators $(1-\beta_i^t)$ are close to zero, leading to significantly larger values of $\hat{m}_t$ and $\hat{v}_t$ compared to the uncorrected terms. This correction is critical for stability in the early iterations of the algorithm [9]. As the number of iterations increases, the bias-corrected estimates converge towards the uncorrected values.

By combining the exponential moving averages from RMSProp with the smoothing effect of momentum, Adam achieves fast convergence with relatively little tuning. It is also robust when dealing with sparse gradients, making it one of the most widely used optimization algorithms in modern machine learning [9].

Bias-Variance Trade-Off

An important concept in machine learning and statistical analysis is the bias-variance trade-off. We can decompose the expected prediction error of a model into three components: bias, variance, and irreducible error (noise). This means that the expected squared error between the true values and the model's predictions can be expressed as the sum of these three components.

Knowing this, and the definition of mean squared error (MSE), we have:

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

where

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

and

$$\mathrm{var}[\tilde{y}] = \mathbb{E}\left[(\tilde{\boldsymbol{y}} - \mathbb{E}\left[\tilde{\boldsymbol{y}}\right])^2\right] = \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}\left[\tilde{\boldsymbol{y}}\right])^2.$$

We can derive this decomposition as follows. Assume the data is generated according to the model:

$$y_i = f(x_i) + \varepsilon_i, \qquad \mathbb{E}[\varepsilon_i] = 0, \quad \mathrm{Var}(\varepsilon_i) = \sigma^2,$$

and let $\tilde{y}_i$ be the predictions of the model

$$\mathbb{E}\big[(y_i - \tilde{y}_i)^2\big] = \mathbb{E}\big[(f(x_i) + \varepsilon_i - \tilde{y}_i)^2\big]\,.$$

Add and subtract $\mathbb{E}[\tilde{y}_i]$:

$$\begin{aligned}\mathbb{E}\big[(y_i - \tilde{y}_i)^2\big] &= \mathbb{E}\Big[\big(f(x_i) - \mathbb{E}[\tilde{y}_i] + \mathbb{E}[\tilde{y}_i] - \tilde{y}_i + \varepsilon_i\big)^2\Big] \\ &= \underbrace{\big(f(x_i) - \mathbb{E}[\tilde{y}_i]\big)^2}_{\text{bias}^2} + \underbrace{\mathbb{E}\big[(\tilde{y}_i - \mathbb{E}[\tilde{y}_i])^2\big]}_{\mathrm{Var}(\tilde{y}_i)} + \underbrace{\mathbb{E}[\varepsilon_i^2]}_{\sigma^2} \\ &\quad + \underbrace{2\,(f(x_i) - \mathbb{E}[\tilde{y}_i])\,\mathbb{E}[\tilde{y}_i - \mathbb{E}[\tilde{y}_i]]}_{=0} + \underbrace{2\,\mathbb{E}[(\tilde{y}_i - \mathbb{E}[\tilde{y}_i])\,\varepsilon_i]}_{=0}.\end{aligned}$$

The cross-terms vanish because $\mathbb{E}[\tilde{y}_i - \mathbb{E}[\tilde{y}_i]] = 0$ and, at test time, $\tilde{y}_i$ is independent of ε_i with $\mathbb{E}[\varepsilon_i] = 0$. Therefore,

$$\boxed{\mathbb{E}\big[(y_i - \tilde{y}_i)^2\big] = \big(f(x_i) - \mathbb{E}[\tilde{y}_i]\big)^2 \; + \; \mathrm{Var}(\tilde{y}_i) \; + \; \sigma^2}$$

Averaging over all points $i = 1, \ldots, n$ yields the dataset-level decomposition:

$$\boxed{\frac{1}{n}\sum_{i=1}^{n} \mathbb{E}\big[(y_i - \tilde{y}_i)^2\big] = \frac{1}{n}\sum_{i=1}^{n} \big(f(x_i) - \mathbb{E}[\tilde{y}_i]\big)^2 \; + \; \frac{1}{n}\sum_{i=1}^{n} \mathrm{Var}(\tilde{y}_i) \; + \; \sigma^2}$$

Implementation

Measurements for error

Mean Squared Error (MSE)

For calculating the Mean Squared Error (MSE), we make use of the scikit-learn library's built-in function `mean_squared_error`. This ensures both efficiency in our implementations

R^2 score

The R^2 score is implemented from its theoretical definition. The numerator and denominator are calculated separately, and then combined into the final return, as shown in Code 2. This metric will be used to evaluate the performance of our models later.

Scaling

On the topic of scaling, we usually scale our data before performing regression. This ensures that the different features θ are on comparable scales, so that no single feature dominates or is weighted less simply because of the magnitude of the feature. As a general practice in our implementations (see Listings 6, 1, and 3), we include an explicit scaling step.

The standard way we scale the data is to call the `StandardScaler` function from the Scikit-learn library [11], and create it as an instance called *scaler*. We then scale the training data and apply the same fitted scaler to the test data to avoid data contamination between the train and test split.

Regression Models

OLS for the Runge function

We begin by generating a dataset of 100 equally spaced points $x \in [-1, 1]$. For each point, the response value y is calculated from the Runge function, with added random noise to simulate measurement errors. To prepare the data for regression, we implemented the function `polynomial_features`. This function constructs the design matrix for a chosen polynomial degree p. Each row in the design matrix contains the powers of x, i.e. $[1, x, x^2, \ldots, x^p]$, with the option of including or excluding an intercept term. The regression parameters are estimated with our function `OLS_parameters`, which implements the ordinary least squares (OLS) solution. Model performance is measured using two standard metrics: the mean squared error (MSE) and the coefficient of determination (R^2), both coded directly from their mathematical definitions (`R2_score`). In the main script, the design matrix is created and the dataset is split into training (80%) and test (20%) sets. The polynomial degree is varied from 1 to 15. For each degree, the following steps are carried out:

1. Construct the design matrix with `polynomial_features`.

2. Split the dataset into training and test sets.

3. Compute the OLS coefficients from the training data.

4. Use the coefficients to predict values on the test set.
5. Compute MSE and R^2 for the predictions.
6. Store the values for later comparison.

Finally, we create an array of polynomial degrees and plot both MSE and R^2 as functions of the polynomial degree. These plots provide a visual overview of how model complexity affects prediction accuracy. This implementation directly reflects the theoretical framework from the lectures: polynomial features control model complexity, the OLS solution provides a baseline estimator, and MSE and R^2 serve as performance measures. Gradient-based optimization methods (introduced later in the course) could be applied as an alternative, but here we focus on the analytical OLS approach.

Ridge

Based on the theoretical definition, we implement Ridge regression as a function `ridge_regression(X, y, λ)`. The parameters are estimated by

$$\theta = (X^\top X + \lambda I)^{-1} X^\top y.$$

To be able to study the effect of the regularization parameter λ, we selected multiple values of λ using `train_test_split` to divide data into training and test sets. For each λ, the model was fitted on training data and evaluated on the test set using mean squared error (MSE) and R^2. The implementation is provided in Code Listing 3. We then subsequently plot the **MSE** values along with the R^2 values with the different values for λ.

We also implemented a way to gauge the difference between the **MSE** and R^2 values between the Ridge and OLS implementations, with differing λ values. The way this as done was by computing the mean of the **MSE** values for each of the λ values, and then plotting them against eachother, as we can see in the code 4

Gradient Descent Variants

After running the algorithm for a sufficient number of iterations, the coefficients converge close to the closed-form solutions we derived in Parts A and B. Comparing the two approaches shows that gradient descent gives nearly the same coefficients, with small numerical differences due to the finite number of iterations and the choice of learning rate. Ridge regression, as expected, shrinks the coefficients compared to OLS.

A key observation from experimenting with the learning rate η is that it strongly influences convergence: if η is too small, the algorithm converges very slowly; if it is too large, the updates overshoot and the algorithm can diverge. Choosing a moderate value allows stable and efficient convergence. [8].

Momentum GD

We implement the momentum gradient descent as a standalone function to allow flexibility and reuse in other parts of the project. The algorithm is initialized with $\theta = 0$, and a variable `change` is introduced to store the previous update direction. At each iteration, the gradient is computed depending on whether we solve the OLS, Ridge, or Lasso case. For our momentum, we select a parameter of our choosing. We also track the mean squared error (MSE) at each iteration to monitor convergence, as shown in code listing 6.

AdaGrad

We implement the Adagrad also as a standalone function to allow flexibility and the ability to reuse in other parts of the project. We initialize $\theta = 0$ and also $\mathbf{r} = 0$. For gradient calculation we can easily choose OLS or Ridge in our function. Then we square the gradient and add it to the previous value of $\mathbf{r}$. We calculate the square of the gradient, the adaptive step and the **MSE** value, and return the θ, **MSE**. The code for the Adagrad implementation is seen in 6.

RMSProp

RMSProp was also implemented as a standalone function as well, allowing for reussability and flexibility. We start by initializing the parameters $\theta, \mathbf{v} = 0$, followed by setting up a matrix for storing our **MSE** values and calculating the decaying average of gradients. We then calculate the parameter $\mathbf{v}$, and update the θ value.The code for the RMSProp implementation can be seen in 6.

ADAM

We also implement ADAM as a standalone function, where we first initialize the parameters $\theta, \mathbf{v}, \mathbf{m} = 0$. We then loop over the iterations and calculate the 1st and 2nd moment, but also creating parameters $\hat{\mathbf{m}}, \hat{\mathbf{v}}$ for the bias corrections, and using these in the final update for θ. The code for the ADAM implementation can be seen in 6.

Stochastic Gradient Descend (SGD)

We imeplented a Stochastic gradient descent (SGD) as a standalong function with different adaptive learning rates that we tested, along with **OLS** and **Ridge** regression methods. The implementation follows the principles laid out in the section about the SGD method, further above.

At each epoch, the data is shuffled randomly to make sure the batches are not correlated between the updates. Then we have the minibatches being created with the chosen length of the batches. For every minibatch, the gradient is computed based on the chosen style, and further, a learn rate optimizer is being used. The ones that are in question are the **Momentum** , **AdaGrad** , **RMSProp** and **ADAM**. The function then inside of these optimizers caluclated the θ update, and also calculates the **MSE** values for the minibatch. This way we can check convergence later in plotting.

The learn rate was set to $\eta = 0.03$, with a momentum factor of 0.3, as well as the λ being able to be specified. The reasons for setting these parameters was that they during testing provided most stable plots and convergences. The full implementation of the SGD is shown in code listing 8

Resampling

Bootstrap setup

To perform bootstrap resampling, we call the `resample` utility function of scikit-learn iteratively on the training data for the number of bootstraps given as argument. This function samples the dataset, with replacement, meaning it returns a dataset of the same size, consisting of elements randomly selected from the original dataset while allowing duplicates. Essentially, we get variations of the original data with some points repeated and others omitted. Repeating this process multiple times generates a distribution of datasets that reflect the variability inherent in the original data.

For each iteration, we fit the model using the resampled data and perform a prediction using each resulting fit. After completing all bootstrap iterations, we compute the mean and standard deviation of the predictions across all bootstrap samples. This provides an estimate of the uncertainty in our model's predictions, as the variability across the bootstrap samples reflects the sensitivity of the model to changes in the training data.

Bootstrap resampling is implemented as a method of the OLS class in Appendix B 9

Cross-validation setup

For cross-validation, we use the `KFold` function from scikit-learn to split the data into k folds. For each fold, we fit the model on $k-1$ folds and validate it on the remaining fold. We repeat this process for all folds and compute the average performance metrics (MSE and R^2) across all folds to assess the model's generalization ability.

K-folds cross validation is also implemented for the OLS class 9

Results

Part A: Ordinary Least Square (OLS) for the Runge function

We fitted the noisy Runge function $f(x) = 1/(1+25x^2)$ using OLS with polynomial features up to degree 15. We tested both the scaled an unscaled versions of the feature matrix to evaluate the effect of scaling the data. The results are from the code listed in 1. The results can be seen in figure 1.

Part B: Adding Ridge regression for the Runge function

We implemented the **Ridge** regression model and fitted it to the noisy Runge function $f(x) = 1/(1+25x^2)$ using different values of the regularization parameter λ. The tested values were

$$\lambda = [0.0001,\ 0.01,\ 0.1,\ 0.5].$$

The data were scaled prior to fitting, and the full implementation and plotting code are provided in Code Listing 3. The resulting fits are shown in Figure 2.

We also plotted the MSE values of OLS vs Ridge by taking the average of the values for the dataset, and then subtracting the differences between them for each of the λ parameters. The results can be seen in figure 3, and the code for implementation can be found in code listing 4.

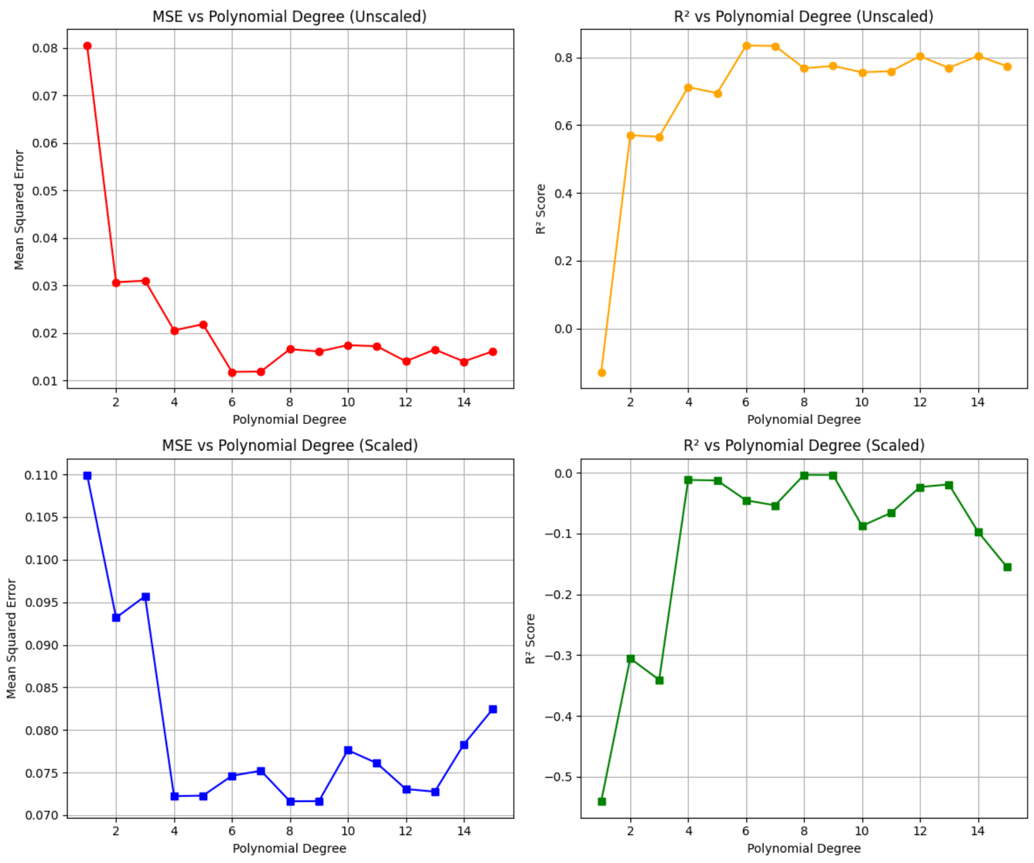

Figure 1: OLS and Ridge regressions. Scaled and unscaled versions

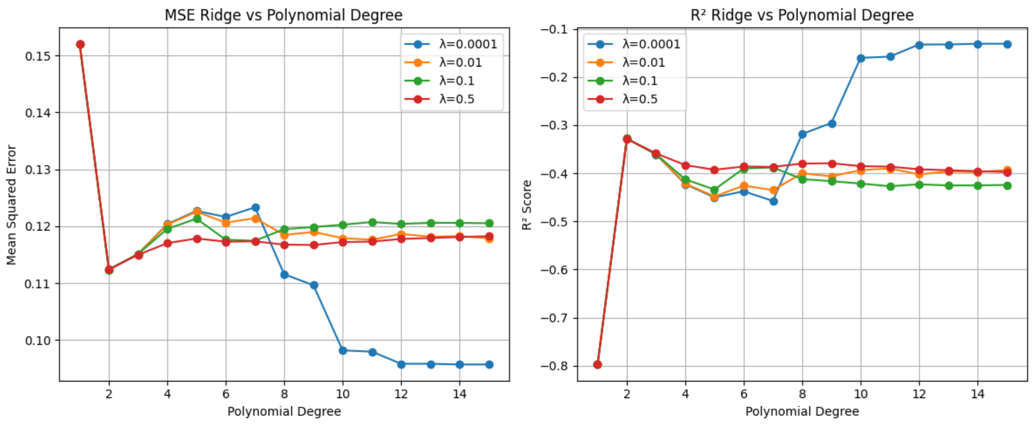

Figure 2: Ridge Analysis for different polynomial degrees

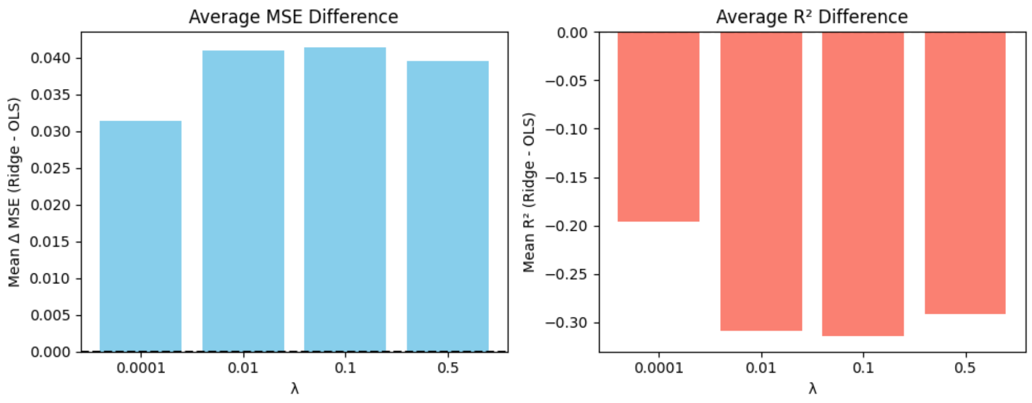

Figure 3: OLS vs Ridge comparisson in MSE

Part C: Writing our own Gradient descent code

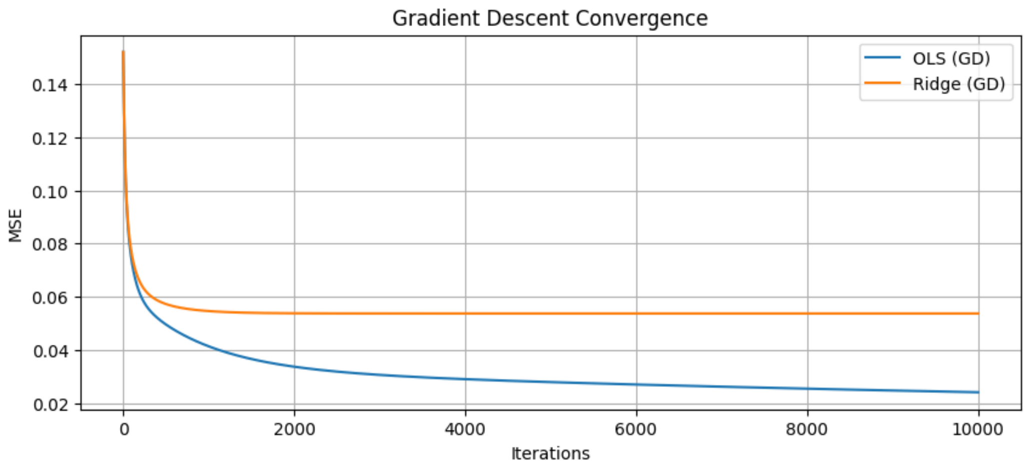

Figure 4: GD Convergence

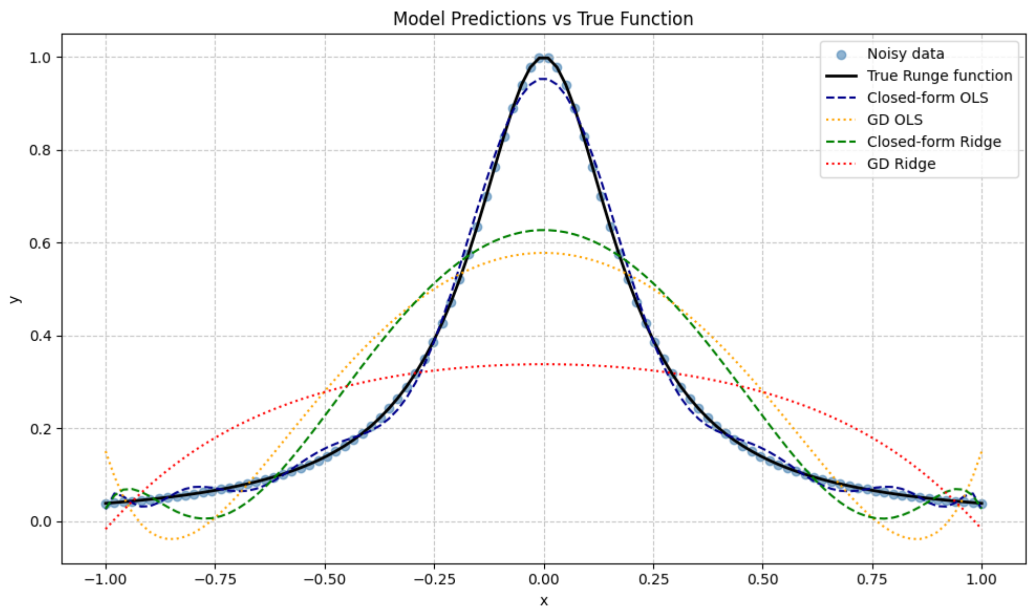

Figure 5: GD: Model prediction vs True function

Part D: Including momentum and more advanced ways to update the learning the rate

For this assignment, we applied the different adaptive gradient descent optimizers (**Momentum**, **AdaGrad**, **RMSProp**, and **Adam**) to both **OLS** and **Ridge** regression on the noisy Runge function $f(x) = 1/(1+25x^2)$ using polynomial features of degree 6. In testing this was proven to give the best fit. The models were trained on the training data (80/20 split), and the fitted parameters were then applied to the entire dataset to visualize the final approximation to the true function.

The learning rate was set to $\eta = 0.03$ with a momentum factor of 0.4, and the parameter λ for Ridge regression was $\lambda = 1 \times 10^{-4}$. For each optimizer, we also recorded the mean squared error (MSE) at each iteration to study convergence.

The complete implementation is provided in Code Listing 6, and the resulting fits and convergence curves are shown in Figure 6.

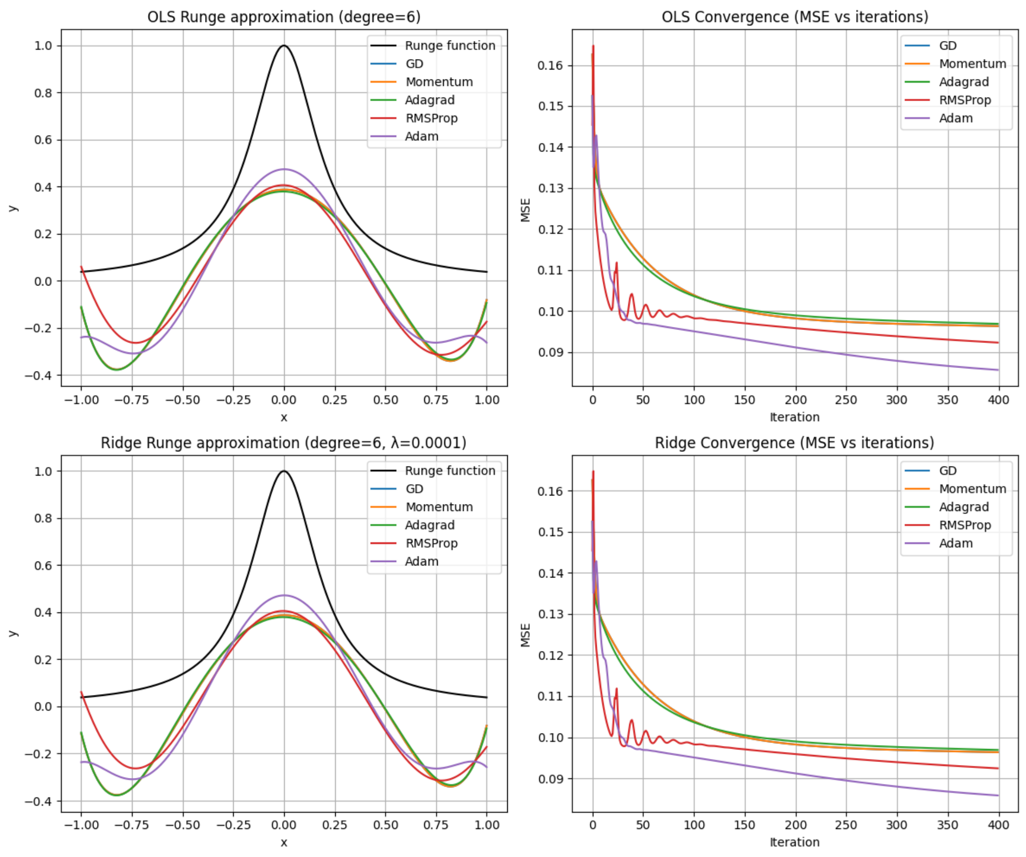

Figure 6: Optimizers fitted on the Runge Function

Part E: Writing own code for Lasso regression

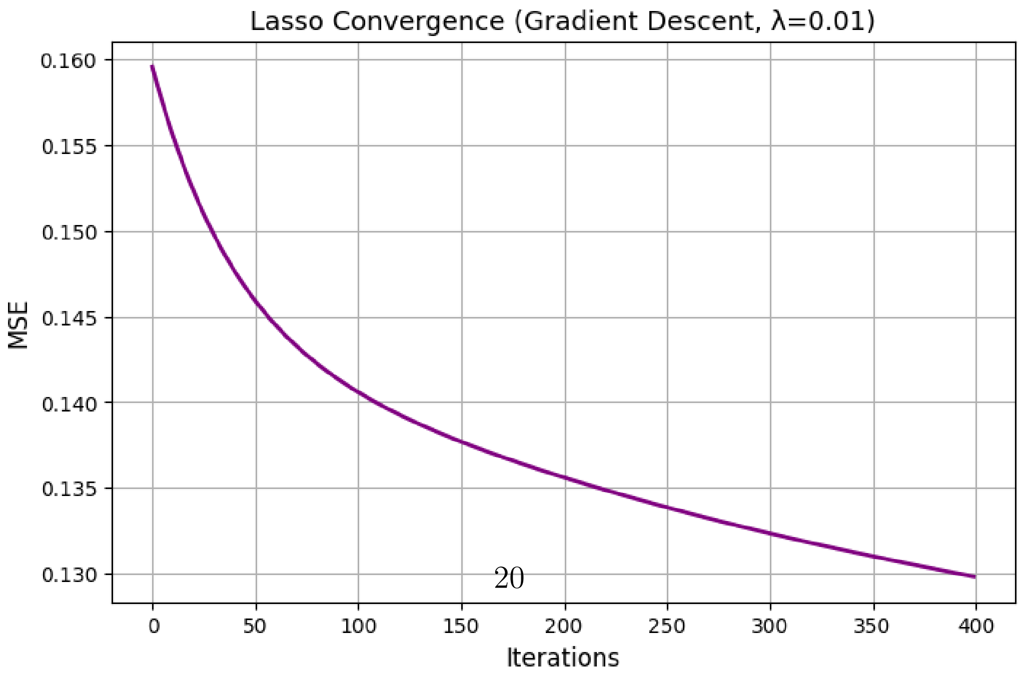

Figure 7: Lasso Convergence

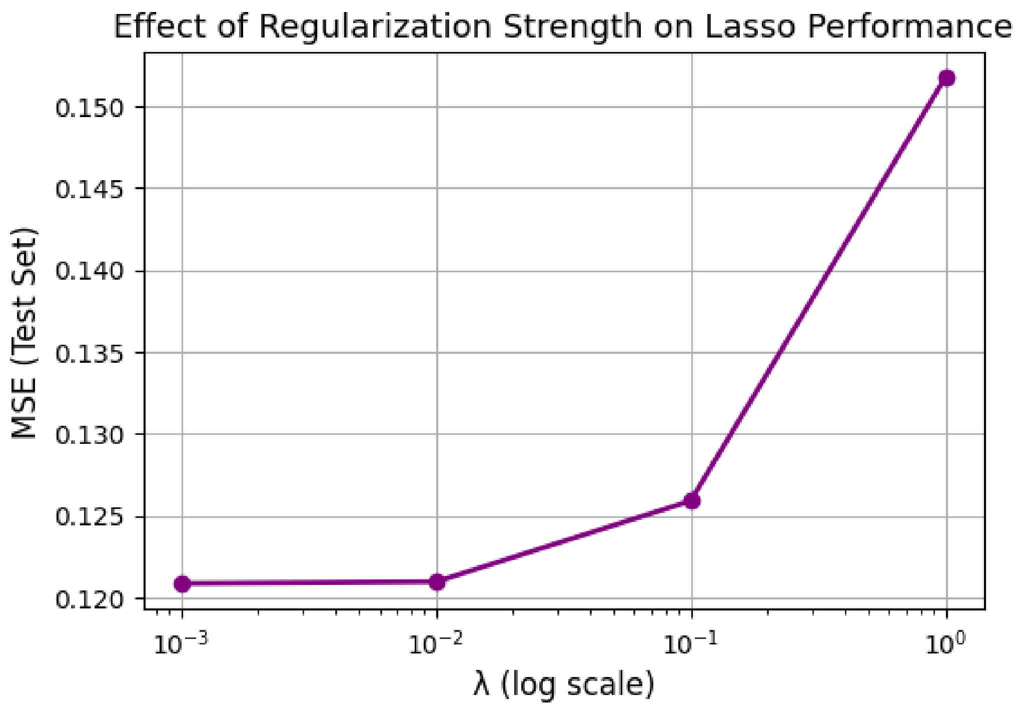

Figure 8: Effect of regularization strength on Lasso

Part F: Stochastic Gradient Descent

We implemented **SGD** and tested the adaptive optimizers **Momentum**, **AdaGrad**, **RMSProp**, and **Adam** for both **OLS** and **Ridge** regression on the Runge function. The models were trained using mini-batches with an 80/20 train–test split, a polynomial degree of 10, and a learning rate of $\eta = 0.03$. The results were compared with the baseline **vanilla gradient descent (GD)**.

For each optimizer, we recorded the mean squared error (MSE) at each iteration to study convergence behaviour. The implementation of the SGD procedure is provided in Code Listing 8, and the resulting convergence curves are shown in Figure 9.

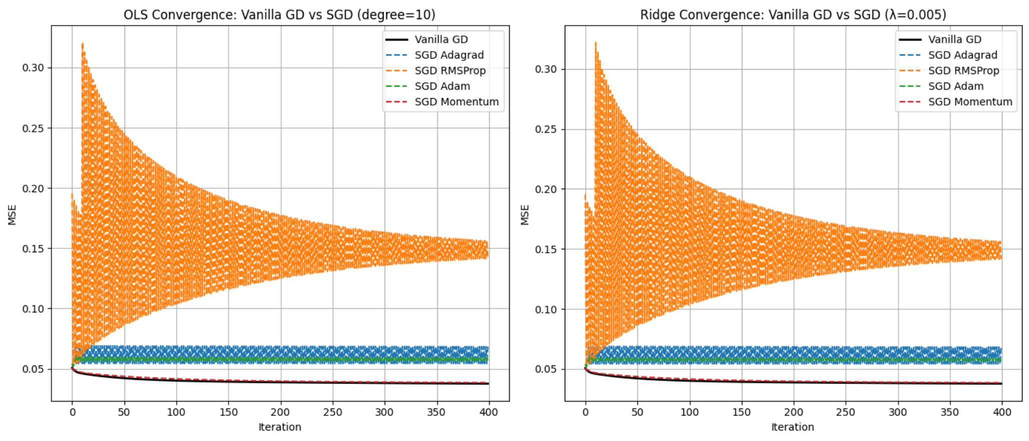

Figure 9: SGD convergence for OLS (left) and Ridge (right) regressions compared with vanilla gradient descent.

Part G: Bias-Variance Trade-off with Bootstrap Resampling

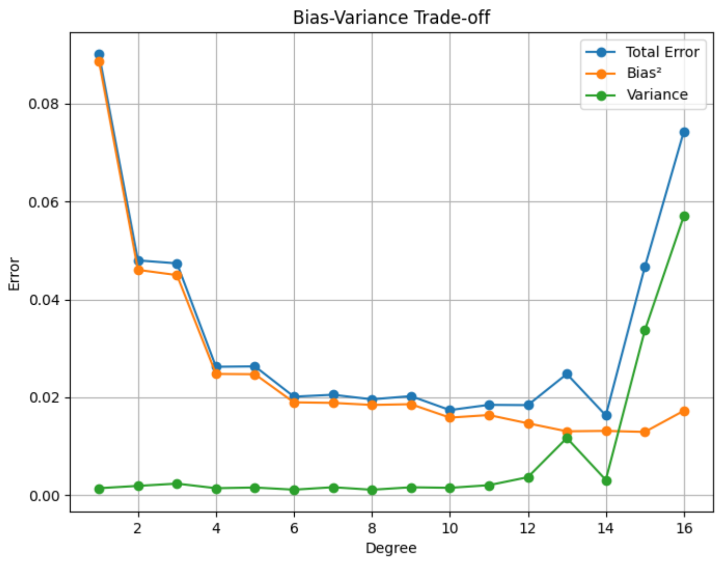

Figure 10: Total Error, Bias, and Variance, with 100 bootstrap resamplings for an Ordinary Least Squares (OLS) model on 100 data points.

Part H: Cross-Validation

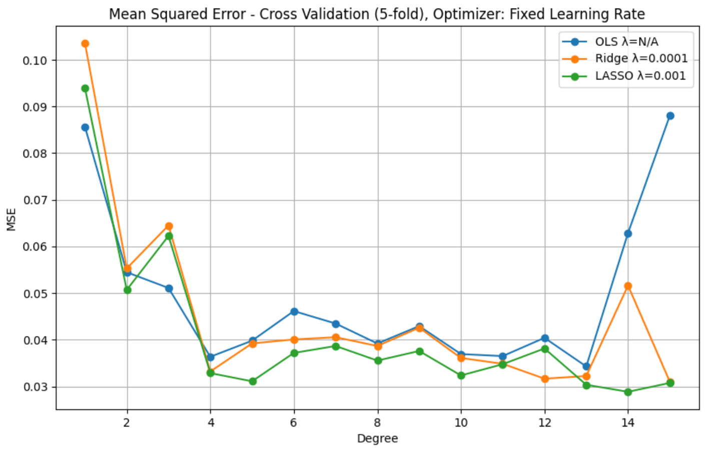

Figure 11: 5-fold cross validation on our three linear regression models, optimized using fixed learning rate, full-batch gradient descent for 50 data points. Values of regularization hyperparameters specified in legend.

Discussion

Part A

As we can see from the results in figure 1, we observe that generally the **MSE** values decrease as and the R^2 values increase in the unscaled version with increasing polynomial degrees. This should not perplex us given that a more complicated model can acount for more of the variation in the dataset than a more simple one. We can see that the lowest values for the error are given with a degree 6 and onward in the unscaled version of the **OLS**.

When observing the scaled data (2 bottom images): we can observe that at first glance we see more of a variation, but if we notice the scale of the data we can actually see that the variation of the data is quite significantly lower than in the unscaled versions. The scaling of the data magnifies the differences between the **MSE** values, making our plot less smooth, however we can see that having the scaled data makes makes our variance much lower. This demonstrates the effect of scaling on numerical stability and ties directly to the bias–variance trade-off: scaling reduces variance and stabilizes coefficient estimates without changing the underlying model complexity.

Scaling ensures that all polynomial features contribute proportionally to the optimization process. Without scaling, higher-order terms dominate the regression due to their larger magnitudes, which leads to ill-conditioned matrices and unstable coefficient estimates. Standardizing the features (using `StandardScaler`) centers the data which improves numerical stability and convergence of the optimization. This stabilization reduces variance in the parameters but does not change the underlying model complexity as said above.

Part B

As we can see from the results in figure 2 we can see the scaled data for the ridge regression fitted for several different λ. As discussed in the section for **Ridge** regression, a lower value of λ will generally give us a lower **MSE** values due to penalization of the large coefficients being less. From the image we can see that the lowest error comes from the value $\lambda = 0.0001$. We can also see that in likeness to Part A, a more complex model will generally map out the parameters θ more optimally than a simpler model, however overfitting can happen.

To analyze further the effect of regularization, we computed the difference in MSE between Ridge and the corresponding OLS models across the dataset. The results, shown in Figure 3, confirm that larger λ values increase the overall error due to stronger coefficient shrinkage, while smaller λ values preserve model flexibility and yield lower MSE. This behavior reflects the fundamental bias–variance trade-off: as λ increases, variance decreases and bias increases. Hence, Ridge regression provides a mechanism to control model complexity and improve generalization compared to unregularized OLS. However as we see the scale of the λ values above 0.01 do not give any significant effect.

Part C

The graph in Figure 4 illustrates how Gradient Descent (GD) for OLS and Ridge regression behave differently in terms of Mean Squared Error (MSE) minimization. The optimization was performed with a fixed learning rate $\eta = 0.01$ over 10,000 iterations (as shown on the x-axis).

Both curves display a rapid initial decrease in MSE, indicating that the gradient descent algorithm quickly approaches a region close to the optimal solution.

The blue curve represents GD for OLS. It continues to decrease slowly toward very small MSE values, reflecting the fact that OLS lacks a regularization term and thus aims to minimize the training error as much as possible. However, this tendency to fully minimize the error may cause the model to overfit the training data, resulting in high variance.

In contrast, the orange curve represents GD for Ridge regression. Here, the MSE converges more quickly and stabilizes at a slightly higher value. This behavior arises from the inclusion of the regularization parameter λ, which penalizes large coefficients and effectively shrinks them toward zero. As a result, Ridge regression accepts a slightly higher bias (poorer fit on the training data) in exchange for lower variance and improved generalization performance on unseen data.

The graph in Figure 4 illustrates how Gradient Descent (GD) for OLS and Ridge regression behave differently in terms of Mean Squared Error (MSE) minimization. The optimization was performed with a fixed learning rate $\eta = 0.01$ over 10,000 iterations (as shown on the x-axis).

Both curves display a rapid initial decrease in MSE, indicating that the gradient descent algorithm quickly approaches a region close to the optimal solution.

The blue curve represents GD for OLS. It continues to decrease slowly toward very small MSE values, reflecting the fact that OLS lacks a regularization term and thus aims to minimize the training error as much as possible. However, this tendency to fully minimize the error may cause the model to overfit the training data, resulting in high variance.

In contrast, the orange curve represents GD for Ridge regression. Here, the MSE converges more quickly and stabilizes at a slightly higher value. This behavior arises from the inclusion of the regularization parameter λ, which penalizes large coefficients and effectively shrinks them toward zero. As a result, Ridge regression accepts a slightly higher bias (poorer fit on the training data) in exchange for lower variance and improved generalization performance on unseen data.

Part D

The implementation of our adaptive learning rate and momentum methods can be found in code listing 6. From the results in figure 6 we can observe clear differences in the different methods.All optimizers were trained using a fixed learning rate of $\eta = 0.03$ and momentum factor 0.4, with $\lambda = 1 \times 10^{-4}$ for Ridge regression. The chosen polynomial degree was 6 since this proved to be the best one tested up to a degree of 15, where the program became slower and much more computationally expensive. Based on the results, the adaptive methods that mapped the Runge function best in order of performance were:

1)ADAM
2)RMSProp
3)Mometum
4)AdaGrad
5)Vanilla GD

The superiority of **Adam** is consistent with the theoretical expectations expressed in the section about gradient descent variants. By combining momentum and adaptive learning rate adjustments, the algorithm provides fast and stable convergence, as well as the best approximation.
RMSProp also performs well due to its exponentially weighted moving average of squared gradients, which helps maintain a stable learning rate. The algorithm is heavily influenced by the choice of hyperparameters chosen, and therefore might be better is it was tuned separatly.
Momentum is an improvment above the vanilla gradient descent, as well as the AdaGrad, where it smoothes the path of the gradient descent and accelerates convergence if the direction is consistent.
AdaGrad, while stable, performs worse due to its continuously decaying learning rate, which eventually becomes too small for further updates. The lack of performance is probably due to the iteration number chosen in this task (400 iterations).

The overall results highlight how the adaptive learning rate algorithms improve the convergence and numerical stability. We have chosen to overlay the actual function with those created by utilizing the algorithms so we can visually get a sense of how well they have performed. Adam and RMSProp provide the best performance, while AdaGrad provides the worst performance because of it's decaying step size limit. These findings are in line with the theoretical behaviors described in the [9].

0.1 Part E

Figure 7 shows the convergence of the LASSO regression model using gradient descent for $\lambda = 0.01$. The mean squared error (MSE) decreases steadily with each iteration, demonstrating that the optimization algorithm is functioning correctly and approaching a minimum. The convergence is smooth and monotonic, indicating that the chosen learning rate and number of iterations are appropriate for stable optimization. The final MSE stabilizes around 0.13, suggesting that the model reaches a satisfactory trade-off between fitting the data and applying regularization.

Figure 8 illustrates how the test MSE varies with the regularization parameter λ. For small λ values (10^{-3}–10^{-2}), the MSE remains low, implying that mild regularization helps control variance without significantly increasing bias. As λ increases beyond 0.1, the MSE rises sharply, indicating that excessive regularization causes underfitting by shrinking relevant coefficients too strongly towards zero. This behaviour aligns with the expected bias–variance trade-off inherent in LASSO regression.

Overall, the results confirm that LASSO effectively regularizes the model by penalizing large coefficients and promoting sparsity. However, for the smooth Runge function, strong LASSO regularization can remove useful higher-order polynomial terms, resulting in poorer generalization performance. The optimal performance is achieved for small λ values, where the model balances bias and variance effectively.

Compared to the gradient descent results for OLS and Ridge regression, the LASSO method exhibits slower convergence and higher final MSE values. This behavior arises from the non-smooth L_1 penalty, which makes the optimization problem more difficult and less stable near zero coefficients. While Ridge regression benefits from the smooth L_2 penalty that improves conditioning and stabilizes convergence, LASSO must rely on subgradient updates, resulting in a more gradual descent. Furthermore, the regularization path of LASSO shows that increasing λ causes several coefficients to shrink exactly to zero, producing a sparse model. Although this sparsity can be beneficial for feature selection, it introduces additional bias and leads to underfitting for the smooth Runge function, where many polynomial terms contribute meaningfully. In contrast, Ridge achieves a better bias–variance balance, whereas LASSO prioritizes simplicity at the expense of accuracy.

Part F

In this final part, we extended our gradient descent implementations by introducing **stochastic gradient descent (SGD)** using the same adaptive learning rate methods as in Parts C–E. The implementation is shown in Code Listing 8, and the results are presented in Figure 9. As is the practice throughout this project: the train-test split was 80/20. We compare the convergence behaviour of SGD with that of the full-batch gradient descent used previously. In our SGD we introduce the mini batches and sampling of the dataset, which leads to higher variance in our dataset. Based on the results, the adaptive methods that had the least **MSE** values in order were:

1)Vanilla GD
2)Momentum SGD
3)ADAM SGD
4)AdaGrad SGD
5)RMSProp SGD

Most interestingly, the vanilla GD achieved the lowest final MSE. This is perhaps because the full-batch gradient computes the exact gradient at each step, giving us a more stable and smoother conversion. We don't actually see any of the SGD implementations outperforming the vanilla GD with the varying adaptive learning rates. Perhaps this is a result of the parameters chosen, but also perhaps this is a flaw of our code.

Evidently from the results in Figure 9, the **Momentum** optimizer performed closest to the baseline **Vanilla GD**, achieving the best performance among the adaptive methods. **Adam** followed closely in third place, while **AdaGrad** ranked fourth with a noticeably higher variance in its MSE curve. This behaviour is consistent with theoretical expectations, as AdaGrad's learning rate decays over time, eventually becoming too small for significant updates.

Most surprisingly, the **RMSProp** implementation performed the worst, displaying severe oscillations that far exceeded those of the other methods. This likely stems from the sensitivity of RMSProp to hyperparameter tuning—particularly the decay factor ρ and the learning rate η. Even after testing multiple parameter configurations, the instability persisted, suggesting that the combination of stochastic mini-batches and our chosen parameters amplified oscillatory behaviour.

Overall, in our case, **SGD did not outperform vanilla GD** in terms of final convergence. While stochastic updates can improve per-iteration efficiency, they also introduce gradient noise that can hinder convergence to the true minimum, especially for smaller datasets or when the learning rate is not finely tuned. Momentum and Adam helped mitigate this noise, but the fully deterministic gradient descent ultimately produced the lowest final MSE and smoothest convergence.

Part G

Figure 10 illustrates the bias–variance trade-off for an Ordinary Least Squares (OLS) model fitted to the noisy Runge function using 100 bootstrap resamplings. The plot shows the total error, bias, and variance as functions of polynomial degree. As the polynomial degree increases from 1 to 15, we observe the following trends:

- **Bias**: The bias decreases consistently with increasing polynomial degree. This is expected, as more complex models can better capture the underlying function, reducing systematic error.
- **Variance**: The variance increases with polynomial degree. Simpler models (low degree) are stable and have low variance, while complex models (high degree) are sensitive to fluctuations in the training data, leading to high variance.
- **Total Error**: The total error, which is the sum of bias and variance, exhibits a U-shaped curve. It decreases initially as bias reduction dominates, reaching a minimum between 6 and 12, before increasing again as variance becomes the dominant factor.

These results illustrate the classic bias–variance trade-off: simple models (low degree) have high bias but low variance, while complex models (high degree) have low bias but high variance. The optimal polynomial degree balances these two sources of error, minimizing the total prediction error. In this case, a polynomial degree of around 6 achieves the best trade-off, yielding the lowest total error on average across the bootstrap samples.

Part H

Figure 11 presents the results of 5-fold cross-validation for three linear regression models: Ordinary Least Squares (OLS), Ridge, and Lasso. Each model was optimized using full-batch gradient descent with a fixed learning rate (selected based on the largest eigenvalue of the Hessian matrix of the cost function) over 50 data points sampled from the noisy Runge function. The regularization hyperparameters for Ridge ($\lambda = 0.0001$) and Lasso ($\lambda = 0.001$) were selected based on prior experiments and because of the small dataset. The plot shows the mean squared error (MSE) on the test folds as a function of polynomial degree from 1 to 15. The results illustrate several key points:

- **OLS**: The OLS model exhibits a U-shaped curve, with low MSE at low polynomial degrees (between 4 and 13) due to high bias. As the degree increases, the MSE rises sharply due to overfitting and high variance.
- **Ridge**: The Ridge regression model shows a slightly more stable MSE curve. The regularization helps control variance, allowing the model to maintain lower MSE at higher polynomial degrees compared to OLS.
- **Lasso**: The Lasso model also demonstrates improved stability over OLS, with the most consistently low MSE, even as the polynomial degree increases.

These effects are more pronounced with fewer data points, where the risk of overfitting is higher.

Overall, the cross-validation results confirm that regularization techniques like Ridge and Lasso effectively mitigate overfitting compared to unregularized OLS. In this case, LASSO, in particular seems the most effective for this setup.

If we compare these results to those from the bootstrap analysis in Part G, we see consistent trends. Both methods highlight the bias–variance trade-off, with OLS suffering from high variance at complex polynomial degrees. Ridge and Lasso regularization reduce variance, leading to more stable performance across folds and bootstrap samples.

Conclusion

This project provided valuable insight into the fundamental principles of machine learning and regression analysis. By using the Runge function as a test case, we demonstrated how increasing polynomial complexity can lead to overfitting when data points are evenly spaced. Although high-degree polynomials appear capable of capturing complex patterns, they tend to generalize poorly due to excessive sensitivity to training data. Through this, we gained a deeper understanding of the bias–variance tradeoff, a central concept in machine learning that highlights the balance between a model's accuracy and its stability. Regularization techniques such as Ridge and LASSO proved effective in mitigating overfitting by penalizing large coefficients and controlling model complexity. Comparing different optimization methods, we observed that Ridge regression achieved the best generalization performance, while Adam provided the fastest and most stable convergence among the gradient-based algorithms. These findings emphasize the importance of choosing both an appropriate regularization scheme and optimization strategy for achieving reliable predictive models. However, our analysis was limited to one-dimensional data and polynomial features. Future work could extend this framework to multivariate datasets or alternative basis functions, such as splines or kernel expansions, to better assess generalization in higher dimensions. Overall, this project deepened our practical and theoretical understanding of how regression methods, regularization, and optimization interact to control overfitting and improve predictive performance—concepts that are foundational across modern machine learning.

Source Code

All source code is available from our GitHub repository: `https://github.com/komandoyoko/Yoan_FYS_STK4155/tree/main/Project1`

Sources

References

[1] Brian Beers, *Regression: Definition, Analysis, Calculation, and Example*, Investopedia, 2025. Available: `https://www.investopedia.com/terms/r/regression.asp` [Accessed: 14-Sep-2025].

[2] GeeksforGeeks, "Mean Squared Error (MSE)," `https://www.geeksforgeeks.org/maths/mean-squared-error/`, accessed October 2025.

[3] XLSTAT, *Ordinary Least Squares Regression (OLS)*, Addinsoft, 2025. Available: `https://www.xlstat.com/solutions/features/ordinary-least-squares-regression-ols`. [Accessed: 14-Sep-2025].

[4] GeeksForGeeks, *What is Ridge Regression?*, GeeksForGeeks, 2025. Available: `https://www.geeksforgeeks.org/machine-learning/what-is-ridge-regression/` [Accessed: 14-Sep-2025].

[5] GeeksforGeeks, *What is Lasso Regression?*, available at: `https://www.geeksforgeeks.org/machine-learning/what-is-lasso-regression/`, accessed October 2025.

[6] IBM Corporation, "Gradient Descent," IBM THINK, `https://www.ibm.com/think/topics/gradient-descent`, accessed September 2025.

[7] CompPhysics, *MachineLearning: Lecture Notes, Week 36*, GitHub repository, https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week36.ipynb, accessed September 2025.

[8] M. Hjorth-Jensen, “ Optimization, the central part of any Machine Learning algorithm” University of Oslo, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapteroptimization.html, accessed September 2025.

[9] M. Hjorth-Jensen, “Stochastic Gradient Descent vs. Full Batch Gradient Descent,” in *Machine Learning Lecture Notes, Week 37*, University of Oslo, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week37.html#sgd-vs-full-batch-gd-convergence-speed-and-memory-comparison, accessed September 2025.

[10] GeeksforGeeks, “R-squared (Coefficient of Determination),” https://www.geeksforgeeks.org/maths/r-squared/, accessed October 2025.

[11] Scikit-learn Developers, “StandardScaler — Scikit-learn Documentation,” https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html#plot-all-scaling-standard-scaler-section, accessed October 2025.

A Code Listings

```
#choosing datapoints between 1 and -1
n = 100
x = np.linspace (-1,1, n)


#From the Runge function: for every x, compute the
    correspponding y
y = 1/(1 + 25 * x**2)
#Adding noise (Gaussian) with mean 0 and variance 1: N(0,1)
noise = np.random.normal(0,1,n)
y_noisy = y + 0.1 * noise


#Creating feature matrix with shape [1,x,x^2,x^3,x^4 ...]
def polynomial_features(x, p, intercept = True):
    n = len(x)
```

```python
    if intercept == True: # adding functionality for intercept
        X = np.zeros((n, p + 1))
        X[:, 0] = 1  # intercept column first
        for i in range(1, p + 1): # from 1 to p inclusive
            X[:, i] = x ** i
    else: #if intercept is False
            X = np.zeros((n, p)) #no intercept column
            for i in range(p): #from 0 to p-1
                X[:, i] = x ** (i + 1)

    return X

def OLS_parameters(X, y):
    X_T_X = np.linalg.pinv(X.T @ X)
    beta_OLS = X_T_X @ X.T @ y
    return beta_OLS

def R2_score(actual , predicted):
     numerator = np.sum((actual - predicted) ** 2) #the numerator of the r2 formula
     actual_mean = np.mean(actual) #our mean value of the actual values
     denominator = np.sum((actual - actual_mean) ** 2) # denominator of the r2 formula
     R2 = 1 - (numerator / denominator) #calculating the r2
     return R2


import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

mse_values_OLS_unscaled , R2_values_OLS_unscaled = np.zeros(15) ,np.zeros(15)  #storing the mse values for each polynomial degree - unscaled
mse_values_OLS_scaled , R2_values_OLS_scaled = np.zeros(15) , np.zeros(15) #storing the r2 values for each polynomial - Scaled

for i in range(1 , 16):
    #unscaled version
    X = polynomial_features(x , i)
    X_train , X_test , y_train , y_test = train_test_split(X , y_noisy , test_size = 0.2 , random_state=67) #splitting into train_test_split
    beta = OLS_parameters(X_train , y_train) #finding optimal beta
    y_pred = X_test @ beta #predicting using the training model
    mse_values_OLS_unscaled[i-1] = mean_squared_error(y_test , y_pred)
    R2_values_OLS_unscaled[i-1] = R2_score(y_test , y_pred)

    #scaled version
    '''
    Important note: We split, and then introduce the scaler because we do not want to contaminate the dataset of train with the information from test!!!
    '''
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled  = scaler.transform(X_test)

    beta_scaled = OLS_parameters(X_train_scaled , y_train)
    y_pred_scaled = X_test_scaled @ beta_scaled

    mse_values_OLS_scaled[i-1] = mean_squared_error(y_test , y_pred_scaled)
    R2_values_OLS_scaled[i-1] = R2_score(y_test , y_pred_scaled)





#now we create degrees array as well
degrees = np.arange(1 , 16)
plt.figure(figsize = (12 , 10))

# Unscaled MSE
plt.subplot(2 , 2 , 1)
plt.plot(degrees , mse_values_OLS_unscaled , marker = 'o', color='red')
plt.xlabel('Polynomial Degree')
plt.ylabel('Mean Squared Error')
plt.title('MSE vs Polynomial Degree (Unscaled)')
plt.grid()

# Unscaled $R^2$
plt.subplot(2 , 2 , 2)
plt.plot(degrees , R2_values_OLS_unscaled , marker = 'o', color='orange')
plt.xlabel('Polynomial Degree')
plt.ylabel('R^2 Score')
plt.title('R^2 vs Polynomial Degree (Unscaled)')
plt.grid()

# Scaled MSE
```

```
plt.subplot(2 , 2 , 3)
plt.plot(degrees , mse_values_OLS_scaled , marker = 's',
    color='blue')
plt.xlabel('Polynomial Degree')
plt.ylabel('Mean Squared Error')
plt.title('MSE vs Polynomial Degree (Scaled)')
plt.grid()

# Scaled R^2
plt.subplot(2 , 2 , 4)
plt.plot(degrees , R2_values_OLS_scaled , marker = 's', color
    ='green')
plt.xlabel('Polynomial Degree')
plt.ylabel('R^2 Score')
plt.title('R^2 vs Polynomial Degree (Scaled)')
plt.grid()

plt.tight_layout()
plt.show()
```

Listing 1: Implementation of OLS regression

```
def R2_score(actual , predicted):
     numerator = np.sum((actual - predicted) ** 2) #the
    numerator of the r2 formula
     actual_mean = np.mean(actual) #our mean value of the
    actual values
     denominator = np.sum((actual - actual_mean) ** 2) #
    denominator of the r2 formula
     R2 = 1 - (numerator / denominator) #calculating the r2
     return R2
```

Listing 2: R^2 Implementation

```
def ridge_regression(X , y , lamb):
def ridge_regression(X , y , lamb):
    I = np.identity(X.shape[1]) # need to have the identity
    matrix
    return (np.linalg.inv(X.T @ X + lamb * I) @ X.T @ y) #the
     actual expression for optimal beta

#setting differing values for lambda
lambd = [0.0001 , 0.01 , 0.1 , 0.5]

mse_values_ridge = np.zeros((15 , len(lambd))) #defining the
    mse values matrix
R2_values_ridge = np.zeros((15, len(lambd))) #R2 matrix for
    storage

for l in range(len(lambd)): #we want to test for all lambdas
```

```python
    for i in range(1 , 16): #we want to test for teh different polynomials
        X = polynomial_features(x , i)
        X_train , X_test , y_train , y_test = train_test_split(X , y_noisy , test_size = 0.2 , random_state= 60) #split the data

        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        beta_ridge = ridge_regression(X_train , y_train , lambd[l]) #do the ridge regression
        y_pred_ridge = X_test @ beta_ridge #do the prediction

        mse_values_ridge[i-1, l ] = mean_squared_error(y_test , y_pred_ridge) #calculate the mse values
        R2_values_ridge[i-1 , l] = R2_score(y_test , y_pred_ridge) #calculate the r2 values




plt.figure(figsize=(12, 5))

# ---- MSE plot ----
plt.subplot(1, 2, 1)
for l in range(len(lambd)):
    plt.plot(degrees, mse_values_ridge[:, l], marker='o', label=f"\lambda={lambd[l]}")
plt.xlabel('Polynomial Degree')
plt.ylabel('Mean Squared Error')
plt.title('MSE Ridge vs Polynomial Degree')
plt.legend()
plt.grid()

# ---- R^2 plot ----
plt.subplot(1, 2, 2)
for l in range(len(lambd)):
    plt.plot(degrees, R2_values_ridge[:, l], marker='o', label=f"\lambda={lambd[l]}")
plt.xlabel('Polynomial Degree')
plt.ylabel('R^2 Score')
plt.title('R^2 Ridge vs Polynomial Degree')
plt.legend()
plt.grid()

plt.tight_layout()
plt.show()
```

Listing 3: Ridge Implementation

```python
difference_mse = np.zeros((15 , 4))
difference_r2 = np.zeros((15 , 4))
for j in range(len(lambd)):
    for i in range(1 , 16):
        difference_mse[i-1, j] = mse_values_ridge[i-1, j] - mse_values_OLS_scaled[i-1]
        difference_r2[i-1 , j] = R2_values_ridge[i-1 , j] - R2_values_OLS_scaled[i-1]


mean_diff_mse = np.mean(difference_mse, axis=0)    # one value per \lambda
mean_diff_r2  = np.mean(difference_r2, axis=0)     # one value per \lambda

plt.figure(figsize=(10,4))

# ---- Mean difference in MSE ----
plt.subplot(1,2,1)
plt.bar([str(l) for l in lambd], mean_diff_mse, color='skyblue')
plt.axhline(0, color='black', linestyle='--')
plt.xlabel("\lambda")
plt.ylabel("Mean \Delta MSE (Ridge - OLS)")
plt.title("Average MSE Difference")

# ---- Mean difference in R^2 ----
plt.subplot(1,2,2)
plt.bar([str(l) for l in lambd], mean_diff_r2, color='salmon')
plt.axhline(0, color='black', linestyle='--')
plt.xlabel("\lambda")
plt.ylabel("Mean R^2 (Ridge - OLS)")
plt.title("Average R^2 Difference")

plt.tight_layout()
plt.show()
```

Listing 4: Capturing difference between OLS and Ridge using MSE and R^2

```python
eta = 0.01
num_iters = 10000
lam = 0.1

```

```python
n_samples, n_features = X.shape

# Initialize parameters to zero
theta_gdOLS = np.zeros(n_features)
theta_gdRidge = np.zeros(n_features)

def ols_gradient(X, y, theta):
    return (2/n_samples) * X.T @ (X @ theta - y)

def ridge_gradient(X, y, theta, lam):
    return (2/n_samples) * X.T @ (X @ theta - y) + 2 * lam *
   theta

# Track MSE
ols_loss = []
ridge_loss = []

for t in range(num_iters):
    grad_OLS = ols_gradient(X, y, theta_gdOLS)
    grad_Ridge = ridge_gradient(X, y, theta_gdRidge, lam)

    theta_gdOLS   -= eta * grad_OLS
    theta_gdRidge -= eta * grad_Ridge

    ols_loss.append(np.mean((y - X @ theta_gdOLS) ** 2))
    ridge_loss.append(np.mean((y - X @ theta_gdRidge) ** 2))

print("Gradient Descent OLS coefficients:", theta_gdOLS)
print("Gradient Descent Ridge coefficients:", theta_gdRidge)

#Closed-form solution
def OLS_parameters(X, y):
    return np.linalg.inv(X.T @ X) @ X.T @ y

def Ridge_parameters(X, y, lam):
    I = np.eye(X.shape[1])
    return np.linalg.inv(X.T @ X + lam * I) @ X.T @ y

theta_closed_formOLS   = OLS_parameters(X, y)
theta_closed_formRidge = Ridge_parameters(X, y, lam)

print("Closed-form OLS:", theta_closed_formOLS)
print("Closed-form Ridge:", theta_closed_formRidge)

#plot GD Convergience
plt.figure(figsize=(10, 4))
plt.plot(range(num_iters), ols_loss, label='OLS (GD)')
plt.plot(range(num_iters), ridge_loss, label='Ridge (GD)')
plt.xlabel('Iterations')
plt.ylabel('MSE')
plt.title('Gradient Descent Convergence')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()


# Plot Model Predictions vs True Function
y_pred_OLS_gd        = X @ theta_gdOLS
y_pred_Ridge_gd      = X @ theta_gdRidge
y_pred_OLS_closed    = X @ theta_closed_formOLS
y_pred_Ridge_closed = X @ theta_closed_formRidge

plt.figure(figsize=(10, 6))
plt.scatter(x, y, color='steelblue', alpha=0.6, label='Noisy
    data')
plt.plot(x, y, color='black', linewidth=2, label='True Runge
    function')
plt.plot(x, y_pred_OLS_closed, '--', color='darkblue', label=
    'Closed-form OLS')
plt.plot(x, y_pred_OLS_gd, ':', color='orange', label='GD OLS
    ')
plt.plot(x, y_pred_Ridge_closed, '--', color='green', label='
    Closed-form Ridge')
plt.plot(x, y_pred_Ridge_gd, ':', color='red', label='GD
    Ridge')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Model Predictions vs True Function')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

Listing 5: Gradient descent for OLS and Ridge

```python
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures,
    StandardScaler
import matplotlib.pyplot as plt


n_samples = 200 #number of samples we will be using


def ols_gradient(X, y, theta):
    return (2/n_samples) * X.T @ (X @ theta - y)

def ridge_gradient(X, y, theta, lam):
    return (2/n_samples) * X.T @ (X @ theta - y) + 2 * lam  * theta





def momentum_gd(X , y , iterations, momentum , n_steps , func , lam , theta_init = None):
    if theta_init is None:
        theta = np.zeros(X.shape[1])
    else:
        theta = theta_init.copy() #we start by implementing the starting point of the theta

    change = np.zeros_like(theta) #we initialize the change

    mse_val = np.zeros(iterations) # We also initialize mse_val as the measure of how far away from our goal we are
    for i in range(iterations): #here we want to iterate until we reach our point
        if func == "OLS": #we split between OLS and ridge gradients
            grad = ols_gradient(X , y , theta)
        elif func == "Ridge":
            grad = ridge_gradient(X , y , theta , lam)

        change = momentum * change + n_steps * grad #the change is based on the momentum * change + direction we are headed
        theta -= change #the final result is theta is changed based on the previous change + the new one.

        mse_val[i] =  mean_squared_error(y , X @ theta) #we want to calculate the error away from the true values of the function
    return theta , mse_val





def ADAgrad(X , y , iterations , n_steps , func , lam  , eps = 1e-6 , theta_init = None):
    if theta_init is None: #if we do not have an initial
```

```python
theta we set it to 0
        theta = np.zeros(X.shape[1])
    else:
        theta = theta_init.copy() #initialize the parameters we need from the function call
    r = np.zeros_like(theta)

    mse_val = np.zeros(iterations)
    for i in range(iterations):
        if func == "OLS":
            grad = ols_gradient(X , y , theta)
        elif func == "Ridge":
            grad = ridge_gradient(X , y , theta , lam)

        gtgt = grad ** 2 #here we are taking the square of the gradient
        r= r+ gtgt #we apply the adaptive step

        theta = theta - (n_steps / (np.sqrt(r) + eps)) * grad #we calculate the new theta here based on the adaptive step

        mse_val[i] = mean_squared_error(y , X @ theta)  #and we also calculate the MSE
    return theta , mse_val




def RMSProp(X , y , iterations , n_steps , func , lam  , p = 0.9 , eps = 1e-5, theta_init = None):
    if theta_init is None:
        theta = np.zeros(X.shape[1])
    else:
        theta = theta_init.copy() #initalize the first steps
    v = np.zeros_like(theta)

    mse_val = np.zeros(iterations)
    for i in range(iterations):
        if func == "OLS":
            grad = ols_gradient(X , y , theta)
        elif func == "Ridge":
            grad = ridge_gradient(X , y , theta , lam)

        v = p*v + (1-p) * grad**2 # Here is our step

        theta = theta - (n_steps/np.sqrt(v + eps)) * grad # the new theta calculated from the RMSprop algorythm

        mse_val[i] = mean_squared_error(y , X @ theta) #as always we calculate the MSE
    return theta , mse_val



def ADAM(X , y , iterations , n_steps , func , lam  , b1 = 0.9 , b2 = 0.999 , eps = 1e-5, theta_init = None):
    if theta_init is None:
        theta = np.zeros(X.shape[1])
    else:
        theta = theta_init.copy() # we initialize the different things we need
    v = np.zeros_like(theta)
    m = np.zeros_like(theta)

    mse_val = np.zeros(iterations)
    for i in range(1 , iterations + 1): #we cannot start with 0 in this loop because of the bias corrections need to raised to the power of the iteration.
        if func == "OLS":
            grad = ols_gradient(X , y , theta)
        elif func == "Ridge":
            grad = ridge_gradient(X , y , theta , lam)

        m = b1*m + (1-b1) * grad #we define our unbiased terms
        v = b2*v + (1-b2)* grad ** 2

        m_correct = m / (1-b1**i) #here we apply the bias correction terms
        v_correct = v / (1-b2**i)

        theta = theta - ( n_steps / (np.sqrt(v_correct) + eps) ) * m_correct #calculate the theta according to the algorythm

        mse_val[i-1] = mean_squared_error(y , X @ theta)  # calculkate the MSE as always
    return theta , mse_val









```

```python



'''
In this block of the code we define the Runge function, create a design matrix and try to extract the optimal theta values using the
gradient descents we have implemented. This will be plotted against the actual runge function to visually see how close we are, and we will
also implement a plot of the MSE vs iterations of the different methods
'''
x = np.linspace(-1, 1, 200)  #we define x
y_true = 1 / (1 + 25 * x**2) #the true runge function
lam = 1e-4
degree = 6 #we choose a polynomial of 4th degree


# setup
poly = PolynomialFeatures(degree)
X = poly.fit_transform(x.reshape(-1, 1))

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)   # mean/std learned here

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_true, test_size=0.2) #We then split the data







'''
Here we define a function called run_optimizer that will take all the different functions and go thourgh them creating different gradient descent methods
and extracting the mse_val and theta values
'''


def run_optimizer_OLS(method, X, y , lambd):
    iterations = 400
    n_steps = 0.03
    momentum = 0.4
    lam = lambd
    if method == "GD":
        # Standard gradient descent (no momentum)
        return momentum_gd(X, y, iterations, momentum, n_steps, "OLS" , lam)
    elif method == "Momentum":
        return momentum_gd(X, y, iterations, momentum, n_steps, "OLS" , lam)
    elif method == "Adagrad":
        return ADAgrad(X, y, iterations, n_steps, "OLS" , lam)
    elif method == "RMSProp":
        return RMSProp(X, y, iterations, n_steps, "OLS" , lam)
    elif method == "Adam":
        return ADAM(X, y, iterations, n_steps, "OLS" , lam)
    else:
        raise ValueError("Unknown method")

def run_optimizer_ridge(method, X, y, lamb):
    iterations = 400
    n_steps = 0.03
    momentum = 0.4
    lam = lamb
    if method == "GD":
        return momentum_gd(X, y, iterations, momentum, n_steps, "Ridge", lam)
    elif method == "Momentum":
        return momentum_gd(X, y, iterations, momentum, n_steps, "Ridge", lam)
    elif method == "Adagrad":
        return ADAgrad(X, y, iterations, n_steps, "Ridge", lam)
    elif method == "RMSProp":
        return RMSProp(X, y, iterations, n_steps, "Ridge", lam)
    elif method == "Adam":
        return ADAM(X, y, iterations, n_steps, "Ridge", lam)
    else:
        raise ValueError("Unknown method")
```

```python
if __name__ == "__main__" :

    plt.figure(figsize=(12, 10))

    # --- OLS fit ---
    plt.subplot(2, 2, 1)
    plt.plot(x, y_true, 'k', label="Runge function")
    for method in ["GD", "Momentum", "Adagrad", "RMSProp", "Adam"]:
        theta, mse_val = run_optimizer_OLS(method, X_train, y_train, lam)

        # Use the same poly + scaler from above
        X_pred = poly.transform(x.reshape(-1, 1))
        X_pred = scaler.transform(X_pred)   # <--- FIX: use transform, not fit_transform
        y_pred = X_pred @ theta

        plt.plot(x, y_pred, label=method)
    plt.legend()
    plt.title(f"OLS Runge approximation (degree={degree})")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid(True)




    # OLS convergence
    plt.subplot(2, 2, 2)
    for method in ["GD", "Momentum", "Adagrad", "RMSProp", "Adam"]:
        _, mse_val = run_optimizer_OLS(method, X_train, y_train, lam)
        plt.plot(mse_val, label=method)
    plt.legend()
    plt.title("OLS Convergence (MSE vs iterations)")
    plt.xlabel("Iteration")
    plt.ylabel("MSE")
    plt.grid(True)





    # Ridge fit
    plt.subplot(2, 2, 3)
    plt.plot(x, y_true, 'k', label="Runge function")
    for method in ["GD", "Momentum", "Adagrad", "RMSProp", "Adam"]:
        theta, mse_val = run_optimizer_ridge(method, X_train, y_train, lam)

        X_pred = poly.transform(x.reshape(-1, 1))
        X_pred = scaler.transform(X_pred)   # <--- same scaler here too
        y_pred = X_pred @ theta

        plt.plot(x, y_pred, label=method)
    plt.legend()
    plt.title(f"Ridge Runge approximation (degree={degree}, \lambda={lam})")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid(True)

    # Ridge convergence
    plt.subplot(2, 2, 4)
    for method in ["GD", "Momentum", "Adagrad", "RMSProp", "Adam"]:
        _, mse_val = run_optimizer_ridge(method, X_train, y_train, lam)
        plt.plot(mse_val, label=method)
    plt.legend()
    plt.title("Ridge Convergence (MSE vs iterations)")
    plt.xlabel("Iteration")
    plt.ylabel("MSE")
    plt.grid(True)

    plt.tight_layout()
    plt.show()
```

Listing 6: Different adaptive learning rates

```python
import numpy as np
from sklearn.metrics import mean_squared_error
from project1_d import ADAgrad, RMSProp, ADAM

def lasso_gradient(X, y, theta, lam):
    n_samples = X.shape[0]
    grad = (2 / n_samples) * X.T @ (X @ theta - y)
    subgrad = lam * np.sign(theta)
    subgrad[0] = 0.0
    return grad + subgrad

def run_optimizer_lasso(method, X, y, lam=0.01):
    iterations = 400
    n_steps = 0.001
    momentum = 0.9

    if method in ["GD", "Momentum"]:
        theta = np.zeros(X.shape[1])
        change = np.zeros_like(theta)
        mse_val = np.zeros(iterations)
        mom = 0 if method == "GD" else momentum

        for i in range(iterations):
            grad = lasso_gradient(X, y, theta, lam)
            change = mom * change - n_steps * grad
            theta += change
            mse_val[i] = mean_squared_error(y, X @ theta)

        return theta, mse_val

    elif method == "Adagrad":
        return ADAgrad(X, y, iterations, n_steps, "Lasso",
    lam)
    elif method == "RMSProp":
        return RMSProp(X, y, iterations, n_steps, "Lasso",
    lam)
    elif method == "Adam":
        return ADAM(X, y, iterations, n_steps, "Lasso", lam)
    else:
        raise ValueError("Unknown method")

# Part E : plotting
#regularization strengths to test
lambdas = [0.001, 0.01, 0.1, 1]

# Store results for each
results = {}

print("=== Lasso Regression (Gradient Descent) ===")
for lam in lambdas:
    theta, mse_val = run_optimizer_lasso("GD", X_train,
    y_train, lam=lam)
    y_pred = X_test @ theta

    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    results[lam] = (mse_val, mse, r2)

    print(f"  ={lam:<6}: MSE={mse:.4f}, R  ={r2:.4f}")

# Plot MSE convergence for one example
example_lambda = 0.01
plt.figure(figsize=(8, 5))
plt.plot(results[example_lambda][0], color='purple',
    linewidth=2)
plt.xlabel('Iterations', fontsize=12)
plt.ylabel('MSE', fontsize=12)
plt.title(f'Lasso Convergence (Gradient Descent,   ={
    example_lambda})', fontsize=13)
plt.grid(True)
plt.show()

# Plot MSE vs
mse_values = [results[lam][1] for lam in lambdas]

plt.figure(figsize=(6, 4))
plt.plot(lambdas, mse_values, marker='o', color='purple',
    linewidth=2)
plt.xscale('log')
plt.xlabel('   (log scale)', fontsize=12)
plt.ylabel('MSE (Test Set)', fontsize=12)
plt.title('Effect of Regularization Strength on Lasso
    Performance', fontsize=13)
plt.grid(True)
plt.show()
```

Listing 7: Lasso implementation and plots

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures,
    StandardScaler
from project1_d import  ADAgrad, RMSProp, ADAM,
    momentum_gd



def Stochastic_gradient(X, y, lambd, method, optimizer,
    size_batch, epochs, total_updates=400):
    datapoint = len(y)
    n_steps = 0.03   # learning rate
    momentum = 0.3   # momentum factor
    lam = lambd
    theta = np.zeros(X.shape[1])

    mse_vals = []
    updates = 0   # keep track of total updates

    while updates < total_updates:   # run until we have the same number of updates as vanilla GD
        indices = np.random.permutation(datapoint) #running a random on the datapoints to shuffle
        X_shuffled, y_shuffled = X[indices], y[indices] # createing the new shuffled versions

        for i in range(0, datapoint, size_batch):
            if updates >= total_updates:
                break  # stop exactly at total_updates

            X_batch = X_shuffled[i:i+size_batch] #creating the minibatches
            y_batch = y_shuffled[i:i+size_batch]

            if method == "OLS" and optimizer == "Adagrad":
                theta, mse_val = ADAgrad(X_batch, y_batch, iterations=1, n_steps=n_steps, func="OLS", lam=lam, theta_init=theta)
            elif method == "OLS" and optimizer == "RMSProp":
                theta, mse_val = RMSProp(X_batch, y_batch, iterations=1, n_steps=n_steps, func="OLS", lam=lam, theta_init=theta)
            elif method == "OLS" and optimizer == "Adam":
                theta, mse_val = ADAM(X_batch, y_batch, iterations=1, n_steps=n_steps, func="OLS", lam=lam, theta_init=theta)
            elif method == "OLS" and optimizer == "Momentum":
                theta, mse_val = momentum_gd(X_batch, y_batch, iterations=1, momentum=momentum, n_steps=n_steps, func="OLS", lam=lam, theta_init=theta)

            elif method == "Ridge" and optimizer == "Adagrad":
                theta, mse_val = ADAgrad(X_batch, y_batch, iterations=1, n_steps=n_steps, func="Ridge", lam=lam, theta_init=theta)
            elif method == "Ridge" and optimizer == "RMSProp":
                theta, mse_val = RMSProp(X_batch, y_batch, iterations=1, n_steps=n_steps, func="Ridge", lam=lam, theta_init=theta)
            elif method == "Ridge" and optimizer == "Adam":
                theta, mse_val = ADAM(X_batch, y_batch,
iterations=1, n_steps=n_steps, func="Ridge", lam=lam, theta_init=theta)
            elif method == "Ridge" and optimizer == "Momentum":
                theta, mse_val = momentum_gd(X_batch, y_batch, iterations=1, momentum=momentum, n_steps=n_steps, func="Ridge", lam=lam, theta_init=theta)

            # store MSE
            mse_vals.append(mse_val[-1] if isinstance(mse_val, (list, np.ndarray)) else mse_val) #Here we append the MSE values, and create a robust append method

            updates += 1

    return theta, np.array(mse_vals)




x = np.linspace(-3, 3, 80)
y = 1/(1+25*x**2)

lam = 0.005
degree = 10

X = PolynomialFeatures(degree).fit_transform(x.reshape(-1, 1)) #here we create the polynomial features
X = StandardScaler().fit_transform(X) #scale the data




from project1_d import run_optimizer_OLS, run_optimizer_ridge #import the optimizers ran in previous assignment



optimizers = [ "Adagrad", "RMSProp", "Adam", "Momentum"] # here we call our optimizers for the adaptive learning step

fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# --- OLS ---
theta_gd, mse_gd = run_optimizer_OLS("GD", X, y, lambd=0)
axes[0].plot(mse_gd, label="Vanilla GD", color='black', linewidth=2)

```

```python
for opt in optimizers:
    theta, mse_val = Stochastic_gradient(
        X, y,
        lambd=0,
        method="OLS",
        optimizer=opt,
        size_batch=80,
        epochs=400
    )
    axes[0].plot(mse_val, label=f"SGD {opt}", linestyle='--')

axes[0].set_title(f"OLS Convergence: Vanilla GD vs SGD (
    degree={degree})")
axes[0].set_xlabel("Iteration")
axes[0].set_ylabel("MSE")
axes[0].legend()
axes[0].grid(True)

# --- Ridge ---
theta_gd, mse_gd = run_optimizer_ridge("GD", X, y, lamb=lam)
axes[1].plot(mse_gd, label="Vanilla GD", color='black',
    linewidth=2)

for opt in optimizers:
    theta, mse_val = Stochastic_gradient(
        X, y,
        lambd=lam,
        method="Ridge",
        optimizer=opt,
        size_batch=80,
        epochs=400
    )
    axes[1].plot(mse_val, label=f"SGD {opt}", linestyle='--')

axes[1].set_title(f"Ridge Convergence: Vanilla GD vs SGD (\
    lambda={lam})")
axes[1].set_xlabel("Iteration")
axes[1].set_ylabel("MSE")
axes[1].legend()
axes[1].grid(True)

plt.tight_layout()
plt.show()
```

Listing 8: Stochastic Gradient Descent

B Object-oriented approach

To improve code organization and reusability, we experimented with encapsulating the regression methods within a class structure. This proved to be useful for managing the added complexity of the project, especially when dealing with multiple regression techniques and optimization algorithms. We used this approach when implementing bootstrap resampling and cross-validation in the last part of the project.

The proposed structure is based on a base OLS class, with Ridge and Lasso classes inheriting from it. They share common methods for fitting the model and making predictions, while allowing for specific implementations of regularization techniques.

The optimization algorithms are implemented as separate objects that can be passed to the fit function of the regression classes, promoting modularity and flexibility. The fit function of the model class calls the step function of the optimizer object, which return updated model parameters based on the given optimization algorithm, passing the gradient and the current parameters for each iteration.

```
import numpy as np
from sklearn.model_selection import train_test_split, KFold
from sklearn.utils import resample
from .optimization import Optimizer

class OLS:
    """
    Ordinary Least Squares Regression Model.
    """
    def __init__(self, x: np.ndarray, y: np.ndarray,
                 degree: int = 1,
                 test_size: float = 0.2) -> None:
        """
        Initialize the OLS model with data.
        Args:
            x (np.ndarray): Input data of shape (n_samples, n_features).
            y (np.ndarray): Output data of shape (n_samples,).
            degree (int): The polynomial degree for the model.
        """
        self.x_train, self.x_test, self.y_train, self.y_test = \
            train_test_split(x, y, test_size=test_size)
        self.degree = degree
        self.theta: np.ndarray = None
```

```python

    def name(self) -> str:
        return "OLS"

    def set_degree(self, degree: int) -> None:
        """
        Set the polynomial degree for the model.
        Args:
            degree (int): The polynomial degree for the model.
        """
        self.degree = degree

    def design_matrix(self, x: np.ndarray | None = None) -> np.ndarray:
        """
        Create the design matrix for polynomial features.
        Returns:
            np.ndarray: The design matrix.
        """
        if x is None:
            x = self.x_train
        return np.vander(x, N=self.degree + 1, increasing=True)

    def fit(self, optimizer: Optimizer | None = None, batch_size: int | None = None) -> None:
        """
        Fit the model to the data.
        Args:
            optimizer (Optimizer | None): The optimization method to use. If None, use analytical solution.
            batch_size (int | None): The batch size for stochastic gradient descent. If None, use full batch gradient descent.
        """
        if optimizer is None:
            X = self.design_matrix()
            self.theta = np.linalg.inv(X.T @ X) @ X.T @ self.y_train # Analytical solution
        else:
            self.gradient_descent(optimizer=optimizer, batch_size=batch_size)


    def gradient(self, X: np.ndarray) -> np.ndarray:
        """
        Compute the gradient of the loss function.
        Args:
            X (np.ndarray): The design matrix.
        Returns:
            np.ndarray: The gradient vector.
        """
        return -2/len(self.y_train) * X.T @ (self.y_train - X @ self.theta)

    def hessian(self, X: np.ndarray) -> np.ndarray:
        """
        Compute the Hessian matrix of the loss function.
        Args:
            X (np.ndarray): The design matrix.
        Returns:
            np.ndarray: The Hessian matrix.
        """
        return 2/len(self.y_train) * X.T @ X

    def update_parameters(self, theta: np.ndarray, eta: float) -> None:
        """
        Update the model parameters.
        Args:
            theta (np.ndarray): The new parameter values.
        """
        self.theta = theta

    def gradient_descent(self, optimizer: Optimizer, batch_size: int | None = None) -> None:
        """
        Fit the model using Gradient Descent.
        Args:
            optimizer (Optimizer): The optimization method to use.
            batch_size (int | None): The batch size for stochastic gradient descent. If None, use full batch gradient descent.
        """
        X = self.design_matrix()
        self.theta = np.zeros_like(X[0])  # Initialize parameters

        H = self.hessian(X)         # Hessian matrix
        eig = np.linalg.eigvals(H) # Eigenvalues of the Hessian
        eta = 1.0 / np.max(eig)     # Learning rate based on Hessian's largest eigenvalue

        optimizer.reset()           # Reset optimizer state for new parameter size
        while optimizer.iterate():
            # Stochastic Gradient Descent with mini-batches
            if batch_size is not None:
                indices = np.random.choice(len(self.y_train), batch_size, replace=False)
                X_batch = X[indices]
                y_batch = self.y_train[indices]
                grad = -2/batch_size * X_batch.T @ (y_batch - X_batch @ self.theta)
            # Full Batch Gradient Descent
            else:
                grad = self.gradient(X)

            # Update parameters using the optimizer
            step = optimizer.step(self.theta, grad, eta)
            self.update_parameters(step, eta)

    def predict(self) -> np.ndarray:
        """
        Predict using the fitted model.
        Returns:
            np.ndarray: Predicted values of shape (n_samples,).
        """
        if self.theta is None:
            raise ValueError("Model is not fitted yet. Call 'fit' before 'predict'.")
        X = self.design_matrix(self.x_test)
        return X @ self.theta

    def mse(self) -> float:
        """
        Calculate the Mean Squared Error of the model.
        Returns:
            float: Mean Squared Error.
        """
        return np.mean((self.y_test - self.predict()) ** 2)

    def r2_score(self) -> float:
        """
        Calculate the R  score of the model.
        Returns:
            float: R  score.
        """
        return 1 - np.sum((self.y_test - self.predict()) ** 2) \
                 / np.sum((self.y_test - np.mean(self.y_test)) ** 2)

```

```python
    def bootstrap(self, optimizer: Optimizer | None = None, n_bootstraps: int = 100) -> tuple[float, float, float]:
        """
        Perform bootstrap resampling to estimate bias and variance.
        Args:
            n_bootstraps (int): Number of bootstrap samples.
        Returns:
            tuple[np.ndarray, np.ndarray, np.ndarray]: Tuple containing
                - error (np.ndarray): Total error of the model.
                - bias (np.ndarray): Bias of the model.
                - variance (np.ndarray): Variance of the model.
        """
        x, y = self.x_train, self.y_train # Save original data
        y_test = self.y_test.reshape(-1, 1)
        y_pred = np.zeros((self.y_test.shape[0], n_bootstraps))

        for i in range(n_bootstraps):
            self.x_train, self.y_train = resample(x, y) # Bootstrap resample
            self.fit(optimizer=optimizer)                # Fit model
            y_pred[:, i] = self.predict().ravel()

        # Calculate error, bias, and variance
        error: float = np.mean( (y_test - y_pred)**2 )
        bias: float = np.mean( (y_test - np.mean(y_pred, axis=1, keepdims=True))**2 )
        variance: float = np.mean( np.var(y_pred, axis=1, keepdims=True))

        self.x_train, self.y_train = x, y  # Restore original data

        return error, bias, variance

    def kfold_cross_validation(self, optimizer: Optimizer | None = None, k: int = 5) -> float:
        """
        Perform k-fold cross-validation to estimate the model's performance.
        Args:
            k (int): Number of folds.
        Returns:
            float: Average Mean Squared Error across all folds.
        """

        kf = KFold(n_splits=k, shuffle=True)
        mse_list = []

        x, y = self.x_train, self.y_train  # Use training data for cross-validation

        for train_index, val_index in kf.split(x):
            self.x_train, x_val = x[train_index], x[val_index]
            self.y_train, y_val = y[train_index], y[val_index]

            self.fit(optimizer=optimizer)

            X_val = self.design_matrix(x_val)
            y_val_pred = X_val @ self.theta
            mse_fold = np.mean((y_val - y_val_pred) ** 2)
            mse_list.append(mse_fold)

        self.x_train, self.y_train = x, y  # Restore original training data

        return np.mean(mse_list)
```

Listing 9: models/ols.py

```
from typing import override
import numpy as np

from .optimization import Optimizer
from .ols import OLS

class Ridge(OLS):
    def __init__(self,
                 x: np.ndarray,
                 y: np.ndarray,
                 degree: int = 1,
                 reg_lambda: float = 1.0,
                 test_size: float = 0.2) -> None:
        """
        Initialize the Ridge regression model with data.
        Args:
            x (np.ndarray): Input data of shape (n_samples, n_features).
            y (np.ndarray): Output data of shape (n_samples,)
```

```python
.
            degree (int): The polynomial degree for the model.
            reg_lambda (float): Regularization strength.
            test_size (float): Proportion of the dataset to include in the test split.
        """
        super().__init__(x, y, degree, test_size)
        self.reg_lambda = reg_lambda

    @override
    def name(self):
        return "Ridge"

    def set_lambda(self, reg_lambda: float) -> None:
        """
        Set the regularization strength for the model.
        Args:
            reg_lambda (float): Regularization strength.
        """
        self.reg_lambda = reg_lambda

    @override
    def fit(self, optimizer: Optimizer | None = None, batch_size: int | None = None) -> None:
        """
        Fit the model to the data (analytical).
        """
        if optimizer is not None:
            self.gradient_descent(optimizer=optimizer, batch_size=batch_size)
        else:
            X = self.design_matrix()
            self.theta = np.linalg.inv(X.T @ X + self.reg_lambda * np.eye(X.shape[1])) @ X.T @ self.y_train

    @override
    def gradient(self, X):
        return super().gradient(X) + 2 * self.reg_lambda * self.theta
```
Listing 10: models/ridge.py

```python
from typing import override
import numpy as np
from .ols import OLS
from .optimization import Optimizer, FixedLearningRate # default optimizer if none provided

```

```python
class Lasso(OLS):
    """
    LASSO (Least Absolute Shrinkage and Selection Operator)
   Regression Model.
    """

    def __init__(self,
                 x: np.ndarray,
                 y: np.ndarray,
                 degree: int = 1,
                 reg_lambda: float = 0.01,
                 test_size: float = 0.2) -> None:
        """
        Initialize the LASSO regression model with data.
        Args:
            x (np.ndarray): Input data of shape (n_samples,
   n_features).
            y (np.ndarray): Output data of shape (n_samples,)
   .
            degree (int): The polynomial degree for the model
   .
            reg_lambda (float): L1 regularization strength (
   lambda).
            test_size (float): Proportion of the dataset to
   include in the test split.
        """
        super().__init__(x, y, degree, test_size)
        self.reg_lambda = reg_lambda

    @override
    def name(self):
        return "LASSO"

    def set_lambda(self, reg_lambda: float) -> None:
        """
        Set the regularization strength for the model.
        Args:
            reg_lambda (float): L1 regularization strength.
        """
        self.reg_lambda = reg_lambda

    @override
    def fit(self, optimizer: Optimizer | None = None,
   batch_size: int | None = None) -> None:
        """
        Fit the LASSO model using gradient descent.
        LASSO cannot be solved analytically, so it requires
   an optimizer.
        Args:
            optimizer (Optimizer | None): The optimization method to use. Required for LASSO.
            batch_size (int | None): The batch size for stochastic gradient descent.
        """
        if optimizer is None:
            optimizer = FixedLearningRate()

        self.gradient_descent(optimizer=optimizer, batch_size=batch_size)

    @override
    def update_parameters(self, theta: np.ndarray, eta: float) -> None:
        """
        Update the model parameters.
        Args:
            theta (np.ndarray): The new parameter values.
        """
        self.theta = np.sign(theta) * np.maximum(0, np.abs(theta) - self.reg_lambda * eta)
```

Listing 11: models/lasso.py

```python
from typing import override
import numpy as np

MAX_ITER_DEFAULT: int = 1000

class Optimizer:
    """
    Base class for optimization methods.
    """

    def __init__(self, max_iter: int = MAX_ITER_DEFAULT):
        self.max_iter = max_iter
        self.iterations = max_iter

    def iterate(self) -> bool:
        """
        Check if more iterations are allowed.
        Returns:
            bool: True if more iterations are allowed, False otherwise.
        """
        self.iterations -= 1
        return self.iterations > 0

    def name(self) -> str:
        return self.__class__.__name__

    def step(self, theta: np.ndarray, gradient: np.ndarray, eta: float = 1.0) -> np.ndarray:
        """
        Perform a single optimization step.
        Args:
            theta (np.ndarray): Current parameters.
            gradient (np.ndarray): Current gradient.
            eta (float): Learning rate.
        Returns:
            np.ndarray: Updated parameters.
        """
        raise NotImplementedError("This method should be overridden by subclasses.")

    def reset(self) -> None:
        """Reset optimizer state. Override in stateful optimizers."""
        self.iterations = self.max_iter

class FixedLearningRate(Optimizer):
    """
    Fixed Learning Rate optimization method.
    """

    @override
    def name(self) -> str:
        return "Fixed Learning Rate"

    @override
    def step(self, theta: np.ndarray, gradient: np.ndarray, eta: float = 1.0) -> np.ndarray:
        return theta - eta * gradient

class AdaGrad(Optimizer):
    """
    AdaGrad optimization method.
    """

    def __init__(self, max_iter: int = MAX_ITER_DEFAULT, epsilon: float = 1e-8) -> None:
        super().__init__(max_iter)
        self.epsilon = epsilon
        self.G = None

    @override
    def name(self) -> str:
        return "AdaGrad"

    @override
    def reset(self) -> None:
        super().reset()
        self.G = None

    @override
    def step(self, theta: np.ndarray, gradient: np.ndarray, eta: float = 1.0) -> np.ndarray:
        if self.G is None:
            self.G = np.zeros_like(gradient)
        self.G += gradient**2
        lr = eta / (np.sqrt(self.G) + self.epsilon)
        return theta - lr * gradient

class Adam(Optimizer):
    """
    Adam optimization method.
    """

    def __init__(self, max_iter: int = MAX_ITER_DEFAULT, beta1: float = 0.9, beta2: float = 0.999, epsilon: float = 1e-8) -> None:
        super().__init__(max_iter)
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = None
        self.v = None
        self.t = 0

    @override
    def name(self) -> str:
        return "Adam"

    @override
    def reset(self) -> None:
        super().reset()
        self.m = None
        self.v = None
        self.t = 0

    @override
    def step(self, theta: np.ndarray, gradient: np.ndarray, eta: float = 1.0) -> np.ndarray:
        if self.m is None:
            self.m = np.zeros_like(gradient)
        if self.v is None:
            self.v = np.zeros_like(gradient)

        self.t += 1
        self.m = self.beta1 * self.m + (1 - self.beta1) * gradient
        self.v = self.beta2 * self.v + (1 - self.beta2) * (gradient ** 2)

        m_hat = self.m / (1 - self.beta1 ** self.t)
        v_hat = self.v / (1 - self.beta2 ** self.t)

        return theta - (eta * m_hat) / (np.sqrt(v_hat) + self.epsilon)

class Momentum(Optimizer):
    """
    Momentum optimization method.
    """

    def __init__(self, max_iter: int = MAX_ITER_DEFAULT, momentum: float = 0.9) -> None:
        super().__init__(max_iter)
        self.momentum = momentum
        self.velocity = None

    @override
    def name(self) -> str:
        return "Momentum"

    @override
    def reset(self) -> None:
        super().reset()
        self.velocity = None

    @override
    def step(self, theta: np.ndarray, gradient: np.ndarray, eta: float = 1.0) -> np.ndarray:
        if self.velocity is None:
            self.velocity = np.zeros_like(gradient)
        self.velocity = self.momentum * self.velocity - eta * gradient
        return theta + self.velocity

class RMSProp(Optimizer):
    """
    RMSProp optimization method.
    """

    def __init__(self, max_iter: int = MAX_ITER_DEFAULT, decay_rate: float = 0.9, epsilon: float = 1e-8) -> None:
        super().__init__(max_iter)
        self.decay_rate = decay_rate
        self.epsilon = epsilon
        self.cache = None

    @override
    def name(self) -> str:
        return "RMSProp"

    @override
    def reset(self) -> None:
        super().reset()
        self.cache = None

    @override
    def step(self, theta: np.ndarray, gradient: np.ndarray, eta: float = 1.0) -> np.ndarray:
        if self.cache is None:
            self.cache = np.zeros_like(gradient)
        self.cache = self.decay_rate * self.cache + (1 - self.decay_rate) * (gradient ** 2)
        lr = eta / (np.sqrt(self.cache) + self.epsilon)
        return theta - lr * gradient
```

Listing 12: models/optimizer.py

```python
import numpy as np

START = -1.0
STOP = 1.0

def runge(x: np.ndarray, noise: bool = False) -> np.ndarray:
    """
    Compute the Runge function with optional Gaussian noise.
    Args:
        x (np.ndarray): Input data of shape (n_samples,).
        noise (bool): If True, add Gaussian noise to the output.

    Returns:
        np.ndarray: Output data of shape (n_samples,).
    """
    y = 1 / (1 + 25 * x**2)
    if noise:
        y += 0.1 * np.random.normal(0, 1, x.shape)
    return y

def x(n_samples: int = 100, random_state: int | None = None) -> np.ndarray:
    """
    Generate uniformly spaced input data in the range [start, stop].
    Args:
        start (float): Start of the range.
        stop (float): End of the range.
        n_samples (int): Number of samples to generate.
        random_state (int | None): Seed for the random number generator.

    Returns:
        np.ndarray: Input data of shape (n_samples,).
    """
    if random_state is not None:
        np.random.seed(random_state)
    return np.linspace(START, STOP, n_samples)

def generate_data(n_samples: int = 1000, noise: bool = True, random_state: int | None = None) -> tuple[np.ndarray, np.ndarray]:
    """
    Generate input and output data using the Runge function.
    Args:
        n_samples (int): Number of samples to generate.
        noise (bool): If True, add Gaussian noise to the output.
        random_state (int | None): Seed for the random number generator.

    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple containing input data (x) and output data (y).
    """
    x_data = x(n_samples=n_samples, random_state=random_state)
    y_data = runge(x_data, noise=noise)
    return x_data, y_data
```

Listing 13: data.py

```python
import numpy as np
from matplotlib import pyplot as plt

from data import generate_data
from models.ols import OLS
from models.ridge import Ridge
from models.lasso import Lasso
from models.optimization import (
    Optimizer,
    FixedLearningRate,
    AdaGrad,
    Adam,
    RMSProp
)

def visualize_data(x: np.ndarray, y: np.ndarray,
                   model: OLS,
                   optimizer: Optimizer | None = None,
                   batch_size: int | None = None) -> None:
    """
    Visualize the input data and a simple OLS fit.
    Args:
        x (np.ndarray): Input data of shape (n_samples,).
        y (np.ndarray): Output data of shape (n_samples,).
        model (OLS): The OLS model to fit and visualize.
        optimizer (Optimizer | None): The optimization method
 to use. If None, use analytical solution.
        batch_size (int | None): The batch size for
stochastic gradient descent. If None,
    Returns:
        None
    """
    name = model.name()
    if optimizer is None:
        name += "(Analytical Solution)"
    else:
        name += f"(Gradient Descent with {optimizer.name()})"
        if batch_size is not None:
            name += f", batch_size={batch_size}"

    # fit model and predict
    model.fit(optimizer=optimizer, batch_size=batch_size)
    y_pred = model.predict()

    # Create figure or use existing one
    plt.figure(figsize=(10, 6))
    # Plot data and prediction points
    plt.scatter(x, y, alpha=0.6, color='lightgray', s=20,
label='Data', zorder=1)
    plt.scatter(model.x_test, y_pred, label='Prediction',
color='blue', zorder=2)

    # Create smooth prediction line
    x_smooth = np.linspace(x.min(), x.max(), 300)
    X_smooth = np.vander(x_smooth, N=model.degree + 1,
increasing=True)
    y_smooth = X_smooth @ model.theta
    plt.plot(x_smooth, y_smooth, color='red', linewidth=2,
    label='Prediction', zorder=2)

    # Final plot settings
    plt.title(f"{name}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()

def analyze_bias_variance(x: np.ndarray, y: np.ndarray,
                          max_degree: int = 15,
                          n_bootstraps: int = 100) -> None:
    """
    Analyze bias-variance trade-off using bootstrap
    resampling.
    Args:
        x (np.ndarray): Input data of shape (n_samples,).
        y (np.ndarray): Output data of shape (n_samples,).
        degree (int): Polynomial degree for the model.
        n_bootstraps (int): Number of bootstrap samples.
    """
    from models.ols import OLS
    model: OLS = OLS(x, y)

    degrees:  np.ndarray = np.arange(1, max_degree + 1)
    error:    np.ndarray = np.zeros(max_degree)
    bias:     np.ndarray = np.zeros(max_degree)
    variance: np.ndarray = np.zeros(max_degree)

    for degree in degrees:
        model.set_degree(degree)
        error[degree - 1], bias[degree - 1], variance[degree
    - 1] = \
            model.bootstrap(n_bootstraps=n_bootstraps)

    plt.figure(figsize=(8, 6))
    plt.plot(degrees, error, marker='o', label='Total Error')
    plt.plot(degrees, bias, marker='o', label='Bias ')
    plt.plot(degrees, variance, marker='o', label='Variance')
    plt.title("Bias-Variance Trade-off")
    plt.xlabel("Degree")
    plt.ylabel("Error")
    plt.legend()
    plt.grid()
    plt.show()

def analyze_cross_validation(x: np.ndarray, y: np.ndarray,
                                 models: list[OLS],
                                 max_degree: int = 15,
                                 k_folds: int = 5,
                                 optimizer: Optimizer | None = None) -> None:
    """
    Analyze model performance using k-fold cross-validation.
    Args:
        x (np.ndarray): Input data of shape (n_samples,).
        y (np.ndarray): Output data of shape (n_samples,).
        models (list[OLS]): List of models to evaluate.
        max_degree (int): Maximum polynomial degree to evaluate.
        k_folds (int): Number of folds for cross-validation.
    """

    degrees: np.ndarray = np.arange(1, max_degree + 1)
    mse_cv:  np.ndarray = np.zeros((len(models), max_degree))

    for i, model in enumerate(models):
        for degree in degrees:
            model.set_degree(degree)
            mse_cv[i, degree - 1] = model.kfold_cross_validation(optimizer=optimizer, k=k_folds)

    plt.figure(figsize=(10, 6))
    for i, model in enumerate(models):
        plt.plot(degrees, mse_cv[i], marker='o', label=f"{model.name()}   ={getattr(model, 'reg_lambda', 'N/A')}")
    plt.title(f"Mean Squared Error - Cross Validation ({k_folds}-fold), Optimizer: {optimizer.name() if optimizer else 'Analytical'}")
    plt.xlabel("Degree")
    plt.ylabel("MSE")
    plt.grid()
    plt.legend()
    plt.show()

if __name__ == "__main__":
    # Generate data
    n = 100
    x, y = generate_data(n_samples=n, noise=True, random_state=42)
    optimizer = FixedLearningRate()

    # Visualize data with OLS fit
    ols = OLS(x, y, degree=8)
    visualize_data(x, y, ols, optimizer=optimizer)
```

```python
    # Visualize data with Ridge fit
    ridge = Ridge(x, y, degree=8, reg_lambda=0.0001)
    visualize_data(x, y, ridge, optimizer=optimizer)

    # Visualize data with Lasso fit
    lasso = Lasso(x, y, degree=8, reg_lambda=0.001)
    visualize_data(x, y, lasso, optimizer=optimizer)

    # Cross-Validation
    models: list[OLS] = [ols, ridge, lasso]
    analyze_cross_validation(x, y, models=models, max_degree
=20, k_folds=5, optimizer=optimizer)

    # Analyze bias-variance trade-off
    analyze_bias_variance(x, y, max_degree=16, n_bootstraps
=100)
```

Listing 14: main.py