Project 1 (FYS-STK4155)

Yoan Tutunarov & Vladislav Foss & Sebastian Koranda

September 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.

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

Methods

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 [2].

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 [3].

Lasso regression

Lasso Regression (Least Absolute Shrinkage and Selection Operator) is a version of linear regression that incorporates an additional penalty term on the absolute values of the model coefficients in order to both regularise the model and perform feature selection. The cost function it minimises is:

$$\min_{\beta} \left\{ (Y - X\beta)'(Y - X\beta) \; + \; \lambda \sum_{j=1}^{p} |\beta_j| \right\}$$

Here $\lambda \geq 0$ is a tuning parameter: larger λ penalises large coefficients more strongly, which tends to shrink some β_j exactly to zero, removing those features. This introduces a bias-variance trade-off: more regularisation (larger λ) reduces variance but increases bias, and choosing λ carefully (often via cross-validation) is essential. Lasso is particularly beneficial when dealing with many predictors and when model interpretability is important since it can produce sparse solutions by eliminating weak predictors [4].

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.

Gradient descent is the concept of that if a function $F(x), x \equiv (x_1, x_2, x_3, ..., x_n)$, then we can pratially derivate the function, and getthe direction where the function grows fastest. If we instead go the opposite way, we will be approaching a minimum of the cost function after taking enough steps. The general function for calculating the next step in a gradient descent is given as:

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

. Here, the parameter η is the learning rate. This can be though of as the step length we take in the direction we want to go, in our case towards the

minimum, and therefore we have the negative term. If the parameter η is too small, the convergence towards the minimum will happends very slowly, and on the other side: if the parameter is too large, the algorithm may diverge. Choosing a learn rate η accordingly is then of great importance. [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 $\mathbf{n}$ datapoints, and we want to have a mini-batch of size $\mathbf{M}$, 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 [8].

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* [8]. 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 [8]. 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 $0 \leq \gamma \leq 1$ [8].

Adagrad

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

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

RMSProp

Adam

Part A

For part A of the project, we are creating our own implementation of the OLS function, giving us the optimal $\hat{\theta}$. From the lecture notes in Week 35, we know that the solution to our regression is given as $y = \mathbf{X}\theta$, where $\mathbf{X}$ is the design matrix. In this assignment we are doing a regression with a polynomial of degree 15. This means that our $\hat{\theta}$ is a 1D array with 15 parameters.
The solution to the OLS is gotten from the lecture notes in week 35, where it is found by derivating the cost function, and solving for the global minima of the function:

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

and given taht $X^T X$ is invertible, the solution becomes:

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

, where y is given from the assignment as Runge's function $y = \frac{1}{1+25x^2}$ + noise. The noise was generated using the numpy library's *random.normal()*, where we select parameters using the information given in the assignment.

We then create the design matrix $\mathbf{X}$ and split the data into train and test using scikit-learn's inbuilt *train test split*() as we can see in the code.'

Part C

In this part, we write our own gradient descent algorithm to estimate the parameters of the regression model. We already encountered a similar implementation in the week 37 exercises, so the code here is a small adjustment of that idea. The goal of gradient descent is to minimize a chosen cost function (in our case the Mean Squared Error for OLS, and MSE with an additional penalty term for Ridge). Minimization is done by iteratively adjusting the model parameters θ. In practice, this means reducing the errors between the predicted values and the true outcomes by updating the coefficients step by step [5]. 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, and 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[6]. 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.

Part E

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

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

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

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

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

[7] 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.

[8] 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.

A Code Listings

```

def polynomial_features(x, p, intercept=True):
    """Build design matrix with polynomial features up to degree p."""
    n = len(x)
    if intercept:
        X = np.zeros((n, p + 1))
        X[:, 0] = 1
        for i in range(1, p + 1):
            X[:, i] = x ** i
    else:
        X = np.zeros((n, p))
        for i in range(p):
            X[:, i] = x ** (i + 1)
    return X

def OLS_parameters(X, y):
    """Compute OLS coefficients using pseudo-inverse."""
    return np.linalg.pinv(X.T @ X) @ X.T @ y

def R2_score(actual, predicted):
    """Compute R^2 score."""
    ss_res = np.sum((actual - predicted) ** 2)
    ss_tot = np.sum((actual - np.mean(actual)) ** 2)
    return 1 - ss_res / ss_tot

# Construct design matrix and split into train/test sets
X = polynomial_features(x, 15)
X_train, X_test, y_train, y_test = train_test_split(X, y_noisy, test_size=0.2)

# Fit OLS and evaluate
beta = OLS_parameters(X_train, y_train)
y_pred = X_test @ beta
```

```
33 mse_y_pred = mean_squared_error(y_test, y_pred)
34 R2 = R2_score(y_test, y_pred)
```

Listing 1: Implementation of OLS regression