Polynomial Regression on the Runge Function
FYS-STK4155

Selma Beate Øvland
University of Oslo
(Dated: October 13, 2025)

We study polynomial regression on the noisy Runge function with added Gaussian noise, and compare Ordinary Least Squares (OLS), Ridge, and Lasso across closed-form and gradient-based solvers. Here we analyze: (i) model complexity (polynomial degree), (ii) explicit regularization (Ridge/LASSO) vs implicit regularization from early stopping in gradient descent (GD), (iii) optimizers (Momentum, AdaGrad, RMSProp, Adam), and (iv) stochastic mini-batch training. We evaluate performance using test MSE and R^2, as well as convergence and divergence, and the bias–variance trade-off via the bootstrap.

Our main findings: (1) explicit ℓ_2 shrinkage (Ridge) gives the strongest generalization at high degrees; (2) for OLS, early-stopped GD can improve on the closed-form solution because of implicit regularization; (3) for Ridge, the closed-form solution remains best unless we tune GD and run the model long enough; (4) adaptive optimizers mainly speed up convergence rather than change final test error on these convex problems; (5) mini-batch training shows the expected noisier curves per epoch with modest test-MSE gaps that shrink with more epochs or tuned steps. The bootstrap study reproduces the classic U-shape, where bias falls and variance rises with the degree, and Ridge/LASSO reduces the variance.

I. INTRODUCTION

A challenge in machine learning methods for regression is to establish a reliable input-output relationship from the data while simultaneously controlling model complexity to prevent poor generalization. While polynomials can approximate smooth functions well, high degree models are prone to numerical instability and overfitting when data is noisy or features are not properly scaled. For effective regression models we want to balance simplicity and flexibility to minimize the expected prediction error. The bias-variance trade-off summarizes this critical tension. [1].

A model with high bias suffers from underfitting, failing to capture the complexity of the true function. A model with high variance suffers from overfitting, meaning it has learnt the noise of the training data, leading to poor generalization on unseen data [2], [3]. Our goal in this project is to explore methods that balance this trade-off, as shown in figure 8. To obtain models that are both stable and accurate, we use regularization and resampling techniques. Especially for multicollinearity issues, we benefit from Ridge and Lasso regression, as they introduce a penalty term to the MSE cost function, forcing the model parameters towards smaller magnitudes, which reduces variance.

In the project, we will use the Runge function.

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

To create a synthetic dataset of evenly spaced data points. We use this function to study how fitting it with high-polynomials illustrates the issue with interpolation instability and overfitting, which is a known phenomenon useful for studying regularization methods [4].

We first examine the fitting of OLS up to degree 15 on the Runge function in Section II B 2, where we evaluate the accuracy using MSE and R^2 as a function of degree. In section II B 3, we study the penalty term λ and its relationship to the polynomial degree again using MSE and R^2 analysis. In section II B 4, we replace the equations with gradient-based solvers and examine convergence under different step sizes. We also add optimizers to our GD models to see how this impacts computability and convergence in section II B 5. In section II B 6, we implement the bootstrap method to separate the error into bias and variance, exploring the effects of the polynomial degree and sample size. Section II B 7 implements k-fold cross-validation and compares these test errors to the bootstrap estimates. Ultimately, we will have a method for diagnosing underfitting and overfitting, as well as selecting polynomial degrees and penalty terms that yield optimal results.

II. THEORY AND METHODS

A. Theory

Regression analysis assumes that the output data can be represented by a continous function f(x) plus noise. In linear regression this function is approximated linearly with

$$\overline{y} = X\ \theta \tag{2}$$

This is a supervised regression problem where we assume that the true data is generated from a noisy model ex-

pressed as:

$$y = f(x) + \varepsilon \tag{3}$$

where in this project the true signal f(x) is expressed as the Runge function 1 and ε represents the random normal distributed noise.

The input domain for x is [-1,1] with additive Gaussian noise.

1. Ordinary Least Squares (OLS)

The standard approach for OLS is to minimize the Mean Squared Error (MSE) cost function $C(\theta)$:

$$\min_{\theta \in \mathbb{R}^p} \; C(\theta) = \frac{1}{n} \|y - X\theta\|_2^2 \tag{4}$$

Minimizing this function analytically leads to the solution for optimal parameters $\hat{\theta}_{\text{OLS}}$

$$\hat{\theta}_{\text{OLS}} = (X^\top X)^{-1} X^\top y \tag{5}$$

The OLS estimator is unbiased, meaning the expectation value $\mathbb{E}\left[\hat{\theta}_{\text{OLS}}\right]$ equals the true parameter value θ. The existence of a solution to OLS depends on invertibility of the Hessian matrix $H = X^T X$. Problems can arise if the design matrix X is high dimensional or if its columns are linearly dependent (known as super-collinearity), potentially leading to a singular (non-invertible) $X^T X$ matrix [5].

To address the issues described above we can use regularized regression like Ridge and Lasso. They add a penalty term to the standard OLS cost function.

2. Ridge regression (ℓ_2 regularization)

Ridge regression adds an ℓ_2 penalty to the OLS objective, controlled by a hyperparameter $\lambda \geq 0$:

$$\min_{\theta \in \mathbb{R}^p} \; \frac{1}{n} \|y - X\theta\|_2^2 \; + \; \lambda \|\theta\|_2^2. \tag{6}$$

The optimal parameters have the closed form

$$\hat{\theta}_{\text{Ridge}} = (X^\top X + \lambda I)^{-1} X^\top y. \tag{7}$$

Adding λI ensures that $X^\top X + \lambda I$ is invertible for any $\lambda > 0$, alleviating singularity/ill-conditioning issues that can affect OLS. Unlike OLS, the Ridge estimator is biased for $\lambda > 0$, but it typically has reduced variance [5].

3. Lasso regression (L_1 regularization)

The ℓ_1 term is classified as

$$\min_{\theta \in \mathbb{R}^p} \quad \frac{1}{n} \|y - X\theta\|_2^2 \; + \; \lambda \|\theta\|_1 \tag{8}$$

where $\|\theta\|_1 = \sum_{i=1}^{p} |\theta_i|$.

The Lasso cost function normally does not yield an analytical solution. We need to solve it numerically, typically with variants of gradient descent methods. A key difference between Lasso and Ridge, is that where Ridge shrinks some of the coefficiants towards zero, Lasso will reduce some of them entirely to zero. This means that where Ridge keeps all the coefficients, Lasso only keeps a subset of them. This can be an advantage if we want sparser models (and if the true signal is sparse). However, Ridge has an advantage when the true signal is denser and we have multicollinearity. Using Lasso in the wrong way can result in underfitting and oversimplification [5].

4. Convergence and optimization algorithms

s *4.1 Convergence and divergence*

The MSE cost function of both OLS and Ridge are convex functions, where for a convex function any local minimum is also a global minimum, guaranteeing that the Gradient Descent (GD) cost function is also a convex function if the learning rate is sufficiently small [5].

4.2 Gradient descent (GD), fixed learning rate η

Plain GD iteratively updates the parameters θ by taking a step proportional to the negative gradient of the cost function $C(\theta)$:

$$\theta_{k+1} \; = \; \theta_k \; - \; \eta \, \nabla_\theta C(\theta_k) \tag{9}$$

Simply put, the step size is the same for every parameter. In ML models, the full gradient $\nabla_\theta C(\theta)$ must be computed by summing over all datapoints in a dataset. Calculating this full gradient is computationally expensive when the datasets become large [6].

4.3 Stochastic gradient descent (SGD)

With stochastic gradient descent the cost function $C(\theta)$ can be expressed as a sum over the individual losses $c_i(x_i, \theta)$ for N data points:

$$C(\theta) = \frac{1}{N} \sum_{i=1}^{N} c_i(x_i, \theta) \tag{10}$$

SGD introduces randomness and computational efficiency by approximating the full gradient $\nabla_\theta C(\theta)$ by summing over a subset of the data B_k:

$$\theta_{j+1} = \theta_j - \eta_j \sum_{i \in B_k} \nabla_\theta \, c_i(x_i, \theta_j) \tag{11}$$

4.4 Momentum and Adaptive Methods (AdaGrad, RMSProp, ADAM)

These methods modify the learning rate η to speed up convergence [7]:

- Momentum: Keeps a running average of past gradients and moves in that direction.
- AdaGrad: Each parameter has its own learning rate. If the parameter sees big gradients, shrink the learning rate; and if it rarely changes, increase the learning rate.
- RMSProp: Similar to AdaGrad, but uses an exponential moving average of squared gradients, to avoid AdaGrad's learning rate goes to zero problems. Works well on noisy problems.
- ADAM: Combines momentum and RMSProp. Tracks average gradient (momentum) and average squared gradient (RMSProp), then bias-corrects them early on. It is fast and stable on noisy and complex data, and is a solid default.

5. Evaluation metrics

5.1 Mean squared error (MSE)

MSE is the squared difference between the true values (y_i) and the model's predicted values ($\overline{y_i}$). It's squared to punish the big differences more than the small ones, and to avoid positives/negatives canceling each other out. The lower the value, the better (smaller error).

Its mathematical expression is

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

It's best to use MSE to optimize and see the absolute error size. MSE is sensitive to outliers.

5.2 R^2 score function

The R^2 score function represents how much better your model is than just measuring the mean of the training data. It's represented as a quantile, where 1 equals perfect predictions and 0 means no better predictions than the mean, so a high value is better. Definition:

$$R^2 \;=\; 1 \;-\; \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y}_{\text{train}})^2} \tag{13}$$

We use R^2 as a measure of how well our model fits the data.

6. Bias-Variance tradeoff

The MSE can be decomposed into three fundamental components:

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

- Bias squared ($\mathrm{Bias}^2[\tilde{y}]$): Represents the error due to simplifying assumptions in the model (e.g., modeling a complex non-linear function using a low-order polynomial).
- Variance ($\mathrm{Var}[\tilde{y}]$): Measures how much the predicted values $\overline{y}$ changes over different training data sets. Highly flexible models usually have high variance.
- Irreducible error (σ^2): The variance of the noise inherent in the data, which the model cannot reduce.

The Bias-Variance trade-off represents the relationship between a model's complexity (bias) and its sensitivity to the training data (variance). A simple model (low complexity) has high bias and low variance. A complex model (high flexibility) has low bias and high variance, adapting too closely to the noise in the training data. With Ridge and Lasso, we introduce a small bias, but in return, we achieve a greater reduction in variance, usually resulting in lower expected test error (MSE) than with OLS [2].

Bias–variance decomposition. Assume $y = f(x) + \varepsilon$ with $\mathbb{E}[\varepsilon] = 0$ and $\mathrm{Var}(\varepsilon) = \sigma^2$, and let $\tilde{y} = \tilde{y}(x)$ be the prediction produced by a learning algorithm trained on a random dataset (so $\tilde{y}$ is a random variable through the training set). Then

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

Expanding the square and taking expectations gives three terms:

$$\underbrace{\big(f(x) - \mathbb{E}[\tilde{y}]\big)^2}_{\mathrm{Bias}^2[\tilde{y}]} \;+\; \underbrace{\mathbb{E}\big[(\tilde{y} - \mathbb{E}[\tilde{y}])^2\big]}_{\mathrm{Var}[\tilde{y}]} \;+\; \underbrace{\mathbb{E}[\varepsilon^2]}_{\sigma^2},$$

because the cross-terms vanish: $\mathbb{E}[\tilde{y} - \mathbb{E}[\tilde{y}]] = 0$ and ε has zero mean and is independent of the training randomness.

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

Here $\text{Bias}^2[\tilde{y}] = \big(f(x) - \mathbb{E}[\tilde{y}]\big)^2$ and $\text{Var}[\tilde{y}] = \mathbb{E}\big[(\tilde{y} - \mathbb{E}[\tilde{y}])^2\big]$ are pointwise in x; averaging over a test set $\{x_i\}_{i=1}^n$ gives the usual empirical versions [2].

7. *Resampling*

7.1 Bootstrap

We use our training dataset and re-draw samples from it (with replacement), several times. We then fit our model to the new datasets to see how predictions change. For each test point, we compute the average prediction, variance, MSE, and squared bias. This indicates whether our model is stable (low variance) or sensitive (high variance) as we adjust the data [2].

7.2 K-fold Cross-Validation (CV)

K-fold CV is a resampling technique designed to avoid issues where random splitting might lead to an unbalanced influence of certain samples on model building or evaluation. We split the data into K equal parts (or folds). For each setting (degree or regularization parameter) we train on K-1 fold, and evaluate on the left-out fold. We repeat this so every fold is the test set once. Average over the K-errors to get CV-MSE. This approach enables us to train and split the data multiple times, which is particularly helpful when the data is sparse. Using CV-MSE, we can select the optimal complexity and retrain with the chosen setting [1].

8. *Scaling, centering, and data splitting*

Standardizing or rescaling data is crucial for numerical stability and model performance, especially when the design matrix X contains features on drastically different scales, which can occur when using high-degree polynomials (as columns with high polynomials would either explode or vanish). Placing all features on a comparable scale prevents large-scale features from dominating the X^TX matrix. In Ridge or Lasso, where we penalize the size of the coefficients, we need a comparable scale so the strength of the regularization parameter λ has a consistent meaning across features [8].

We split the data into training and testing data. The purpose of the testing data is to simulate unseen data to evaluate a model's performance. It is essential to split the data before performing any standardization, as this keeps the test data "hidden" from the model. We only standardize the training data (this action let's the model "see" the data, which is why we don't do it on the test data). After fitting the transformation parameters to the training data, we apply the fitted transformation to both the test and training matrices. The test set must be transformed using the mean and standard deviation derived from the training set, not from the test set itself. When we use K-fold CV, this principle extends to each fold. When using a fold as the test set, the standardization parameters must be calculated from the other folds (the K-1 folds) [1].

B. Methods - Implementation

We will perform a regression analysis by implementing the stepwise methods discussed in the Theory section for several polynomial degrees. Through all our tests, we will use the Runge function (1), and we will evaluate the methods using MSE and R^2, in addition to convergence loss curves where applicable.

1. *Generating our data set and preprocessing*

We study supervised regression of noisy observations

$$y = f(x) + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2) \tag{15}$$

where $f(x)$ is the Runge function (1) on [-1,1]. In our code, synthetic data is generated using the `make_data(n=300, noise_sd=0.3, seed=42)` function in data.py. It generates n input points uniformly spaced in the [-1, 1] interval, evaluates the Runge function at these points, and adds Gaussian noise, resulting in two arrays as the output: x (inputs) and y (targets). We fix the seed (`seed=42`)throughout the calculations so the results are reproducible by using the function `rng = np.random.default_rng(seed)` .

To give the model the flexibility to capture the shape of the curve, we expand each scalar input (x) into polynomial features using `build_features(x, degree=p, include_bias=False)` function, where the output is our design matrix. This produces columns with $x, x^2, x^3..x^p$ where include_bias=False, means we omit the intercept column. Each column represents the complexity we have chosen for our model. We handle the intercept separately by centering the training targets and adding back the mean at prediction time. We do this because the intercept column contributes nothing to the standardizations, and we don't want Ridge/Lasso to penalize the intercept column. After that, we split the dataset into training and testing sets. The StandardScaler function is fitted on the training matrix. Then, we scale both the training and test matrices using the training mean and standard deviation. We also center the training targets by subtracting their training mean. This is the standardization procedure we will use throughout the project, except when implementing k-fold CV, where we need to adjust our preprocessing procedure to avoid data leakage slightly. In K-fold CV we fit all preprocessing only on the training fold, so we apply the fitted transform to the validation fold. We

achieve this by placing the feature construction function, scaling, and centering within a "pipeline" (a scikit-learn object) so that they are refit within each fold [8] [1].

2. Ordinary Least Square (OLS) for the Runge function

We start with the simplest closed-form model: OLS. The OLS finds a set of coefficients that matches the model's predictions as close as possible to the observed targets. In our implementation (`fit_ols`), we use the data set generated and preprocessing techniques described in Section II B 1, and solve the normal equations directly. The output from this are the parameters θ. We calculate the MSE and R^2 to evaluate the model's fit across different polynomial degrees. We expect the method to perform more poorly as the polynomial degree increases. We also plot how the noise affects MSE and R^2. We expect a noise free model to work much better at higher polynomials than if we implement noise. We plot a noise-free and heavy noise scenario with MSE vs R^2 to see the effect. For the MSE, we expect a U-shaped curve where the intermediate polynomial degrees are the best fit, and for the R^2 we expect the same effect, with an upside-down U-shaped curve, where the low degrees and higher degrees are bad fits. We also inspect $|\theta|_2$ vs p; we expect large norms at high p to indicate ill-conditioning and overfitting.

To improve the performance of our model with high-degree polynomials, we implement the L_2 norm penalty, also known as Ridge regularization.

3. Adding Ridge regression for the Runge function

Data and polynomial features are produced by the same utilities explained in Section II B 2, still using Runge as the target function. In `def fit_ridge(X, y_c, lam, n_factor=True)` we add the L_2 term to the X^TX term from OLS, so this becomes $X^TX + \lambda I$, which reduces variance at the cost of small bias, but improves on MSE. We expect that the coefficients will shrink with the L_2 penalty.

4. Gradient Descent (GD)

We implement a fixed learning rate Gradient Descent for OLS and Ridge, where we replace the closed-form solution with

$$\nabla_\theta \mathcal{L}_{\text{OLS}} = \frac{1}{n} X^\top (X\theta - y_c), \qquad \nabla_\theta \mathcal{L}_{\text{Ridge}} = \frac{1}{n} X^\top (X\theta - y_c) + \alpha\,\theta.$$

Where $\alpha = \lambda/n$ to keep GD consistent with the closed-form solutions. We initialize and standardize the data in the same way as before. We compare our GD solutions to the closed-form solutions by measuring MSE and R^2. Additionally, we visualize convergence using loss vs. iteration curves.

5. Optimizers, Lasso, Stochastic GD

In addition to plain GD, we tested four optimizers — momentum, RMSProp, AdaGrad, and Adam — to update the learning rate. We use the same data initialization and standardization as before; the only difference is the update rule for the learning rate.

For Lasso we first minimize on standardized features and a centered target:

$$\min_\theta \ \frac{1}{2n} \|X\theta - y_c\|_2^2 + \lambda \|\theta\|_1 \tag{16}$$

We compare our code for Lasso to the scikit-learn coordinate descent version, using optimizers in the same manner as with GD for OLS and Ridge.

For each model, we also test SGD, where we approximate the gradient over a randomly sampled mini-batch instead of the full gradient. Per epoch we reshuffle indices and sweep disjoint batches (batch size B). Hyperparameters: learning rate η, batch size B, and epochs E. We use the same standardization/centering as for GD, so updates are well scaled. Momentum/AdaGrad/RMSProp/Adam are applied by replacing the raw step with their individual updates (same as in our full-batch optimizers), but using the mini-batch gradient.

6. Bootstrap

We use the bootstrap to estimate how our model would perform on new data, and to break down the test error into bias^2 and variance. We start with one test/train split, where the test data is fixed. We resample the training test set with replacement **B** times. We train on each set, and predict on the same test set. We re-fit all preprocessing (feature building, scaling, centering) inside each bootstrap training to avoid leakage.

7. K-fold Cross-Validation

We evaluated polynomial regression on the Runge data with noise $\sigma = 0.3$ and degrees $p = 1, \ldots, 15$. For each degree we computed a K-fold (where we used K=10) cross-validation (CV) MSE for:

- **OLS** where we select the degree p with lowest CV MSE.

- **Ridge** (with λ tuned inside each training fold on a small logarithmic grid; the fold's prediction uses the best λ),
- **Lasso** (same tuning strategy as above for λ).

We treat k as validation; the remaining folds are the training set, and we apply preprocessing only to the training folds. We also compared CV against a bootstrap estimate of the test MSE (using the same train/test split as earlier), with a bootstrap size of $B = 200$.

C. Use of AI tools

1. *ChatGPT*

I used ChatGPT to help me create a plan for structuring the code for this project. Before that, I started with part (a), wrote the code, then went to part (b) and realized I could reuse some code from part (a). I ended up with messy code that lacked a clear structure, featuring repeated variables and functions more times than necessary. I wanted to separate the code and make it reusable for all parts of this project, as well as for other projects/exercises later. So I uploaded the project description to ChatGPT and asked for a plan on how to organize the code (not the actual code). ChatGPT provided me with a plan that allows me to create one Python file at a time (data, models, plotting, etc.), so I don't have to think only in terms of parts a, b, c, etc., which was a bit overwhelming. This helps me keep the code cleaner and easier to reuse.

I also use ChatGPT to create all the tables and mathematical equations used here, allowing me to format them in neat LaTeX quickly.

2. *Grammarly*

I have a Grammarly business account through my work that I use for language correction (grammar, spelling, fixing sentences, punctuation, etc).

III. RESULTS AND DISCUSSION

A. Results

a. Setup: Data from the Runge function

$$f(x) = \frac{1}{1 + 25x^2}, \qquad x \in [-1, 1],$$

with additive Gaussian noise $\varepsilon \sim \mathcal{N}(0, \sigma^2)$. For $\sigma = 0.3$ we construct polynomial features up to degree $p \leq 15$, use a single train/test split, apply standardization on the training data only, fit closed-form OLS on the centered target $y_{\text{train}} - \bar{y}_{\text{train}}$, and shift predictions back to the original scale by adding $\bar{y}_{\text{train}}$.

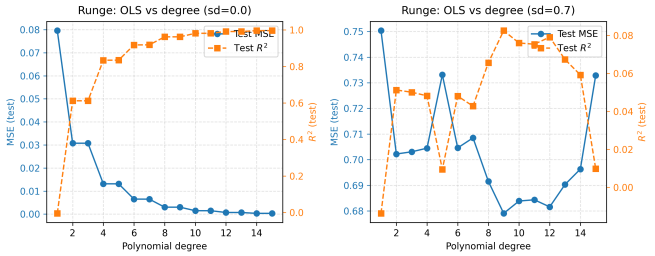

Figure 1. Test MSE vs polynomial degree, for different noise values.

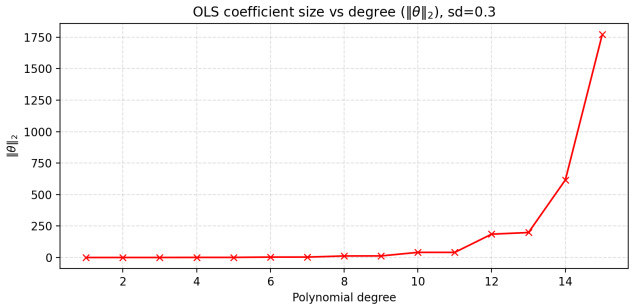

Figure 2. The figure shows the relationship between the ℓ_2 norm of the OLS coefficients and the polynomial degree p. ℓ_2 norm of the coefficients grows quickly with degree.

1. *Simplest CF method: Ordinary Least Square (OLS)*

We tested two different noise scenarios shown in Figure 1: a noiseless dataset and a high noise dataset. For the noiseless scenario test MSE decreases monotonically with degree p and approaches ~ 0. R^2 rises toward ~ 1. A higher degree improves fit. For a high-noise dataset (sd=0.7) MSE is high and flattens out in particular areas, then rises sharply with complexity grade towards 1. R^2 fluctuates as well, but decreases sharply with complexity towards 0.

In figure 2 and beyond, we have settled on a noise level between our two tests in figure 1, sd = 0.3, to replicate real data. In figure 2 we plot the coefficient size vs degree p, and see that it remains modest at low to medium values of p, then grows rapidly after p ≈12.

2. *Introducing the penalty term λ, CF Ridge regression*

In figure 3, as the penalty term λ increases from minimal values, the test MSE is smallest at around $\lambda \approx 10^{-4} - 10^{-5}$ and then increases steadily from there. R^2 has its peak at the same region ($10^{-4} - 10^{-5}$) and steadily decreases from there. In figure 4, the coefficient norm

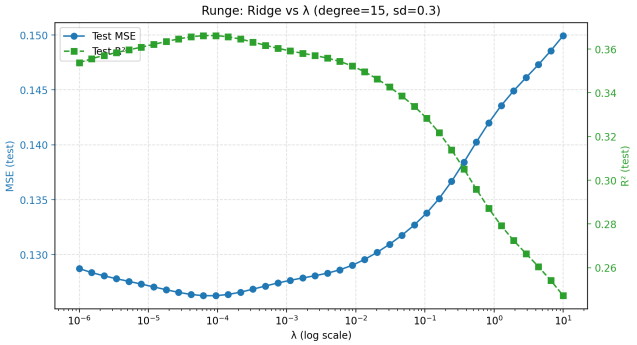

Figure 3. Test MSE and R^2 plotted against different λ values.

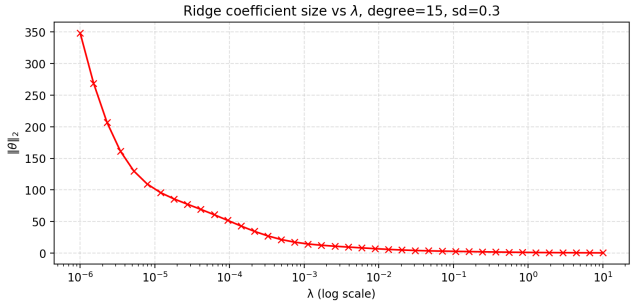

Figure 4. The figure shows the relationship between the l_2 norm of the Ridge coefficients and λ. Ridge coefficients shrink monotonically with λ; most of the shrinkage happens between 10^{-6} and 10^{-3}.

shrinks steadily as λ increases. Shrinkage is most effective at $10^{-6} - 10^{-3}$ then flattens out.

3. *Gradient descent (plain/full batch)*

Table I shows the MSE and R^2 evaluations from GD with a fixed learning rate compared to our closed-form methods. Overall Ridge (CF) had the best performance metrics. For OLS specifically, GD performed better, but for Ridge, CF had better metrics.

Table I. Test performance for OLS and Ridge (CF = closed form, GD = gradient descent).

Model	Solver	MSE	R^2
OLS	CF	0.135149	0.321477
OLS	GD	0.131579	0.339400
Ridge	CF	0.126253	0.366140
Ridge	GD	0.132170	0.336433

Table II shows the printed training loss we calculated from GD and CF. The difference between them should ideally be ≈ 0 for each model; here, both methods have a difference in their training loss. A larger GD loss value indicates that GD didn't fully converge.

Table II. Training loss for OLS and Ridge (CF = closed form, GD = gradient descent).

Model	Loss (CF)	Loss (GD)
Ridge	0.042513	0.045991
OLS	0.040149	0.045871

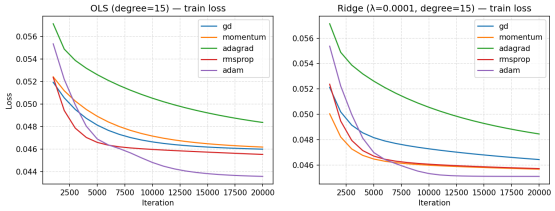

Figure 5. Loss vs iteration curves for plain GD and with optimizers, showing convergence. Adam/RMSProp converges faster/smoother while AdaGrad stalls.

4. *Optimizers, Lasso, SGD*

In Figure 5, GD and momentum cluster pretty tightly for OLS, while RMSProp and Adam drop faster, but RMSProp flattens out earlier, while Adam keeps descending. For Ridge, all optimizers reduce the loss more quickly, but improve the most in terms of momentum from OLS. AdaGrad is performing the worst on both models, even more than the baseline GD.

In Figure 6, we have the same loss curves as for GD. AdaGrad is still the worst-performing optimizer. We see a lot more noise in the mini-batch curves, especially for plain SGD and momentum, but all curves descend faster than the curves in figure 5.

Test metrics for GD are shown in table III. All optimizers reach a comparable test performance. Adam is performing the best for both OLS and Ridge, while AdaGrad has the worst scores. For SGD in Table IV, we observe a worse overall MSE compared to the GD variants, where AdaGrad underperforms for both models.

Table III. Test metrics, GD baseline with optimizer for OLS and Ridge.

Optimizer	OLS		Ridge	
	MSE	R^2	MSE	R^2
gd	0.132044	0.3371	0.131033	0.3421
momentum	0.132831	0.3331	0.131375	0.3404
adagrad	0.137117	0.3116	0.137484	0.3098
rmsprop	0.130348	0.3456	0.131679	0.3389
adam	0.128056	0.3571	0.130480	0.3449

Lasso variants, both scikit-learn and GD, and then with optimizers, have more variance in their results. In table V we see that the coordinate descent (scikit-learn) version exhibits better MSE and R^2. We observe some improvement in our GD model with optimizers, where

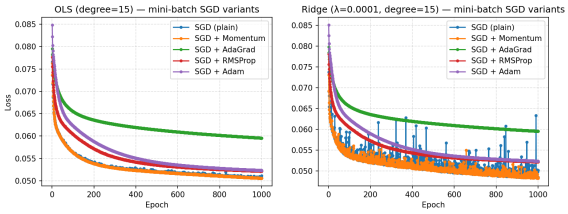

Figure 6. Loss vs iteration curves for plain SGD and with optimizers, showing convergence. Curves are noisier (stochasticity) but converge; Adam/Momentum reduces zig-zagging.

Table IV. Test metrics, SGD baseline, OLS and Ridge with optimizers.

Optimizer	OLS (mini-batch)		Ridge (mini-batch)	
	MSE	R^2	MSE	R^2
sgd	0.142868	0.2827	0.140808	0.2931
momentum	0.143477	0.2797	0.139641	0.2989
adagrad	0.155100	0.2213	0.155068	0.2215
rmsprop	0.146386	0.2651	0.146255	0.2657
adam	0.146501	0.2645	0.146678	0.2636

Adam yields the best test metrics, with an MSE of 0.131237 and an R^2 of 0.3411, compared to a baseline of an MSE of 0.152036 and an R^2 of 0.2367. The coordinate descent model yields the best overall test metrics. GD and SGD are equivalent with the same result.

Table V. Test metrics for Lasso with and without different optimizers.

Optimizer	MSE	R^2
Coordinate Descent	0.129139	0.3517
GD	0.152036	0.2367
sgd	0.152036	0.2367
momentum	0.152041	0.2367
adagrad	0.148984	0.2520
rmsprop	0.132883	0.3329
adam	0.131237	0.3411

5. *Bias-Variance Trade-off*

Figure 7 shows a figure similar to Fig. 2.11 of Hastie, Tibshirani, and Friedman. This plot was created using a single train/test split and shows that the train MSE falls monotonically with the degree, while the test MSE forms a shallow U-shape and reaches a minimum at around 12 degrees.

Figure 8 shows a bootstrap version of the decomposed Bias-Variance, where $Bias^2$ first decreases as the polynomial degree increases, but then grows a little. The variance increases slightly as p also increases. Test MSE grows as p increases. The minimum here is between 8 and 12 degrees.

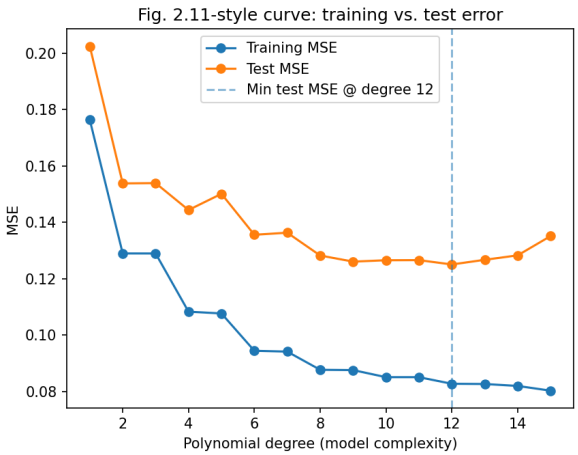

Figure 7. Classic bias–variance: train MSE decreases with p, test MSE forms a shallow U.

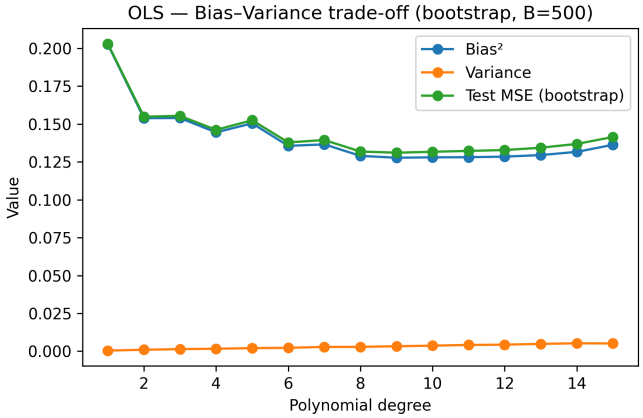

Figure 8. Bootstrap estimates of Bias^2, Var, and Test MSE across degrees (B=200). Bias^2 falls then rises slightly; Var increases with p; MSE minimum around $p \approx 9$–12.

6. *K-fold Cross-Validation*

All models in Figure 9 show a substantial decrease in CV-MSE until model complexity reaches between 9 and 12, where it plateaus. Around p = 13, Ridge achieves a slight improvement over OLS; otherwise, they remain comparable. Lasso is tied with Ridge until the plateau is reached. The shaded region appears to widen up to the plateau and then remains unchanged.

In figure 10, both curves have a minimum around the same degree, around p = 10. The bootstrap curve has slightly higher values,

Our final result is shown in table VI. Here, OLS and Ridge are essentially tied, with Lasso having a slightly higher MSE, but a lower optimal degree. Our bootstrap has a higher MSE

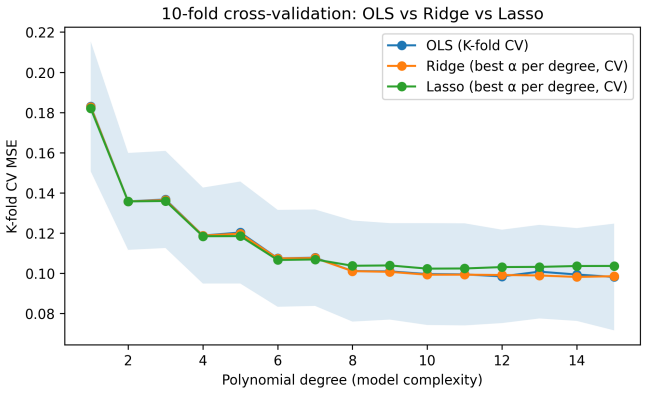

Figure 9. 10-fold Cross-Validation: [OLS vs Ridge vs Lasso] vs polynomial degree p. Lines show mean MSE with CV. The shaded region is the ± 1 standard deviation across folds, only for OLS.

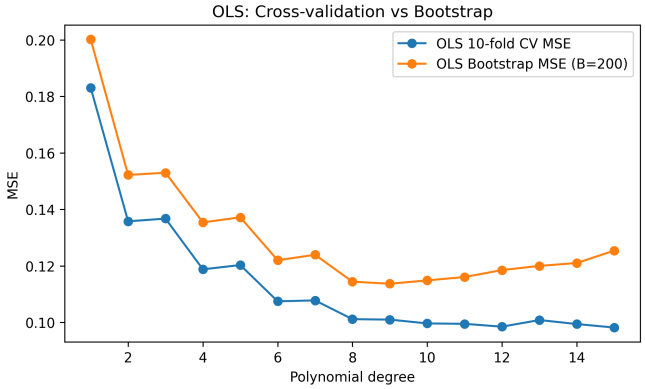

Figure 10. Comparison of CV-MSE and bootstrap MSE vs degree. The minimum occurs at similar degrees; the bootstrap is slightly higher.

Table VI. Best model complexity by method (10-fold CV). Bootstrap row uses the same fixed train/test split as earlier and is not directly comparable to CV.

Method	Best degree p^*	CV MSE
OLS	15	0.098184
Ridge (best α per degree)	14	0.098207
Lasso (best α per degree)	10	0.102341
Bootstrap OLS (fixed split)	15	0.119178

B. Discussion

For the closed-form Ordinary Least Squares (OLS) method, these two curves reproduce the expected bias-variance trade-off. Increasing p reduces bias, but beyond a certain point, the variance dominates, so the test error stops improving and actually gets worse. In figure 1, we see that a noiseless prediction function behaves well at higher polynomial degrees. This is a pure bias regime, with noiseless data higher-degree polynomials approximate the Runge curve better. But for the model with heavy noise we see the more characteristic U-shape we predicted in the method section, meaning that our model gives the best predictions at medium complexity and becomes unstable at higher and lower degrees.

In figure 2 we see the ill-conditioning of high-complexity design matrices using the OLS method. Large coefficients are a classic sign of overfitting, which is why we proceed to test the Ridge method with its penalty term.

Ridge with its penalty term trades variance for bias. At small λ we have low bias and higher variance, and this is where the model performs best on this split with p = 15. As we increase the penalty term, the coefficients are strongly shrunk and variance falls, but bias grows so test MSE and R^2 worsens. The curve for test MSE in 3 is mostly monotone after its minimum at $10^{-4} - 10^{-5}$, further shrinking the coefficients after this will add bias.

Next we replaced the analytical solution that we used for the closed-form OLS and Ridge, with a plain gradient descent with a fixed learning rate. We still used the same data, with the same noise, same polynomial degree p = 15 and the same penalty term we found worked well for the Ridge method earlier. For Ridge in GD form we still use the α-form, as in $\alpha = \frac{\lambda}{n}$.

In table I the results from GD are compared to the CF metrics. For OLS we see that GD attains a slightly lower MSE score than CF despite a higher training loss, as seen in table II, meaning we most likely stopped the GD before reaching full convergence. On Ridge, the CF solution performs best. Since Ridge already shrinks the coefficients with the chosen λ, we get extra under-fitting with GD. We see that if η is too small, GD converges slowly and can stop far from the optimum. If η is too large, GD can oscillate or diverge. We also printed the training loss for CF and GD; we see that they don't match (as they should, indicating that GD has converged completely).

Our results show that the chosen η and iteration number yielded only partial convergence. For OLS, it acted helpfully, but for Ridge, we need a tuned η or iteration number to close the gap between Gd and CF training loss. We may need millions of steps to make GD converge, as GD is slow on ill-conditioned problems, such as high-degree Runge polynomials. Increasing λ also speeds up the convergence.

Adding optimizers to our GD approach changed the behaviour more than it changed the metrics. Momentum and Adam reached low trainig loss in fewer iterations than plain GD, while AdaGrad and RMSProp need to be tuned better. We attempted to adjust η for each optimizer, as one setting doesn't work for all of them; however, selecting the correct one requires further investigation. For metrics, the MSE and R^2 values ended up fairly similar between the different optimizers, differing primarily in how quickly and stably

the methods converged.

Introducing Lasso with optimizers, the coordinate descent (CD) version showed the best results. The CD version quickly identifies a sparse pattern by adjusting one coefficient at a time while keeping the others fixed. If the data indicate that the coefficients are small relative to λ, CD sets them to zero. If it's larger, CD shrinks it by a precise amount. Our GD model updates **all** coefficients at once, and pushes them towards zero (but never all the way). Because of this, the update direction can zigzag and require many steps. Instead of snapping to zero like CD, it can hover around zero, unless we have the exact right η and number of iterations. This means that CD knows the best move and direction for each coefficient, snaps it to zero immediately if necessary, while GD just nudges everything in the direction, often unevenly, which is why CD converges much faster [9]. Adam improves our GD to comparable values to CD by adapting the steps and damping the oscillations from GD described above. We still see that we should try to tune our learning rate more effectively to maximize the benefits of GD methods.

Instead of full-batch GD, we test mini-batch SGD. Mini-batches introduce gradient noise, which speeds up progress but yields a noisier curve. We observe that the full-batch GD yields a smoother curve; however, the number of iterations has been significantly reduced in SGD. Efficiency has been improved in SGD, but we have increased variance. Tuning the number of epochs and η we can probably match the metrics of GD, and we could have used a stop-function to validate our results.

The difference between K-fold CV and bootstrap is the test set - K-fold CV rotates the test set, while bootstrap has a fixed test set. With bootstrap we can separate bias^2 and variance, while K-fold CV is efficient with moderate K values. In our trials, both methods selected similar degrees as the minimum (as seen in figure 10) where MSE was slightly higher for bootstrap. Both has its uses: K-fold CV is good for model selection (best degree p and λ), but we need bootstrap for bias/variance breakdown, we can show where bias^2 falls then rises, how variance grows with complexity and why the minimum is where it is. Bootstrap is more compute-expensive than K-fold CV. K-fold CV gives us the model selection, while bootstrap tells us why this is the best choice.

IV. CONCLUSION

For low-noise cases and low to medium complexity, OLS works best. When introducing Ridge, we see a useful operating range of around 10^{-4} for this exact data, split, and degree, where Ridge matches OLS but keeps coefficients smaller. However, too little shrinkage yields overly large coefficients, and too much shrinkage underfits (resulting in high MSE and low R^2). Overall, Ridge stabilizes the degree 15 polynomial and offers a controllable bias-variance trade-off. Plain GD with a fixed learning rate works, but is computationally inefficient for high-degree polynomials. The learning rate η matters, but a higher η means more progress per step, but limits speed. With optimizers, we can achieve the same metrics more quickly and with fewer computations; however, if we don't tune η properly, we can get divergence. If we want to use Lasso (which is good when data is sparse), we should stick to the CD scikit-learn version. SGD is computationally effective, and by tuning η and the number of epochs, we can achieve comparable test metrics to full-batch GD, which is significantly more computationally expensive. Which regularization method we use matters for flexible models. Lasso/Ridge, tuned by CV, matched or improved upon OLS at higher degrees by controlling the variance. With bootstrap methods, we could predict an optimal degree for our data. Using K-fold Cross-Variance, we confirm the degree range and can select the sensible λ value for Ridge and Lasso. We're left with a recipe to follow for how to optimize our methods for the future:

- Pick the optimal degree by K-fold CV
- If using a more complex model where OLS doesn't give good results anymore: tune λ by CV. Use Ridge if the data is dense, Lasso if the data is sparse.
- Use Adam (gave us mostly the better results in our test above).
- Bootstrap can be used to demonstrate the bias-variance trade-off.

[1] M. Hjorth-Jensen, Machine learning: Lecture notes – week 39, `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html` (2025), accessed 2025-10-12.

[2] M. Hjorth-Jensen, Machine learning: Lecture notes – week 38, `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week38.html` (2025), accessed 2025-10-12.

[3] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning* (Springer, New York, 2009).

[4] Wikipedia contributors, Runge's phenomenon (2025), accessed 2025-09-26.

[5] M. Hjorth-Jensen, Machine learning: Lecture notes – week 36 (2025), accessed 2025-10-12.

[6] M. Hjorth-Jensen, Machine learning: Lecture notes – week 37 (2025), accessed 2025-10-12.

[7] M. Hjorth-Jensen, Optimization and gradient methods, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapteroptimization.html (2023), accessed 2025-10-13.

[8] M. Hjorth-Jensen, Week 35: Introduction to regression and model complexity, https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week35.html (2023), accessed 2025-10-13.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, Journal of Machine Learning Research **12**, 2825 (2011).