Analyzing Different Regression and Resampling Methods

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

Regression in machine learning is a fundamental technique for predicting outcomes based on input features. It finds relationships between variables so that predictions on unseen data can be made. A major challenge arises as model complexity increases: low-degree models may underfit, while high-degree polynomial regression can become unstable and overfit the data. In this project, we study the Runge function, a well-known function that highlights the difficulties of high-degree polynomial interpolation. We apply Ordinary Least Squares, Ridge, and Lasso regression, complemented by gradient descent and its variants, including momentum, Adagrad, RMSprop, and ADAM. Resampling techniques such as bootstrap and cross-validation are used to evaluate model generalization and analyze the bias-variance trade-off. The results show that OLS fits become highly unstable for high polynomial degrees, while Ridge and Lasso regularization significantly improve stability and predictive accuracy. Gradient descent methods reproduce the analytical results, though their performance depends strongly on learning-rate strategies. Overall, the study highlights the importance of regularization and resampling for controlling overfitting and improving the reliability of regression models.

I. INTRODUCTION

The aim of this project is to study various regression methods, such as Ordinary Least Squares, Ridge Regression, and Lasso Regression. It focuses on fitting polynomials to a specific one-dimensional function, the Runge function:

$$\frac{1}{1+25x^2}$$

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

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

II. METHODS

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

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

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

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

A detailed description of the methods follows.

A. Ordinary Least Squares

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

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

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

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

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

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

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

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

1. *Implementation*

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

B. Ridge regression

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

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

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

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

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

1. *Implementation*

The inputs of the Ridge function are a feature matrix $\mathbf{X} \in \mathbb{R}^{n\times p}$, a vector of targets $\mathbf{y} \in \mathbb{R}^n$, the regularization parameter λ and the intercept. The output is the vector of the parameters $\boldsymbol{\theta} \in \mathbb{R}^p$, whose fomula is given above. The function uses the same function `np.linalg.pinv` to compute the pseudoinverse of the matrix $\mathbf{X}^\top\mathbf{X}+\boldsymbol{\lambda}\mathbf{I}$, ensuring numerical stability.

C. Lasso regression

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

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

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

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

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

D. Gradient descent

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

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

$$C(\boldsymbol{\theta}) \approx C(\boldsymbol{\theta}_n)+(\boldsymbol{\theta}-\boldsymbol{\theta}_n)^\top\nabla_{\boldsymbol{\theta}}C(\boldsymbol{\theta}_n)+\frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\theta}_n)^\top\mathbf{H}(\boldsymbol{\theta}_n)(\boldsymbol{\theta}-\boldsymbol{\theta}_n),$$

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

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

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

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

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

1. *Implementation*

The iteration start from an initial guess $\boldsymbol{\theta}^{(0)}$, the parameters are updated iteratively according to the rule:

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

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

Algorithm 1 Gradient Descent

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

(AAA: I don't know if we want something like this)

E. Stochastic Gradient Descent (SGD)

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

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

where i_t is a randomly selected data point and η is the learning rate. (AAA: cite lecture)

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

1. *Implementation*

Algorithm 2 Stochastic Gradient Descent

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

F. Momentum

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

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

Moreover, for a function with steep directions and flat directions, a single global η may be inappropriate: steep coordinates require a smaller step size to avoid oscillation and flat coordinates could use a larger step to speed up progress.

In order to mitigate this problem, gradient descent with momentum is introduced: it refers to a method that smoothens the optimization trajectory by adding a term that helps the optimizer remember the past gradients.

Mathematically, let $\boldsymbol{v}_n$ denote the velocity (or accumulated gradient) at iteration n. The update rules for gradient descent with momentum are:

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

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \boldsymbol{v}_{n+1},$$

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

G. Adagrad

Adagrad adapts the learning rate for each parameter based on the historical squared gradients, giving smaller updates to frequently updated parameters and larger updates to infrequent ones. This is particularly useful for sparse data.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla l_{i_t}(\theta_t),$$

where G_t is the sum of squared gradients up to time t and ϵ prevents division by zero.

The accumulation of squared gradients can lead to excessively small learning rates over time, slowing down convergence.

1. *Implementation*

(AAA)

H. RMSprop

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

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

with decay rate ρ 0.9 or 0.99,

the new parameter update is

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

(AAA cite lecture notes) On the other hand, RMSprop requires tuning of the decay rate ρ and the base learning rate η.

1. *Implementation*

(AAA)

I. Adam

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

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

and of the squared gradients

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

and correcting bias

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

the update rule in Adam becomes

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

where ϵ prevents division by zero.

Adam can sometimes converge to slightly worse minima than SGD in certain tasks, and hyperparameter tuning is still required.

1. *Implementation*

(AAA)

J. Gradient descents comparison

Stochastic Gradient Descent (SGD) is appreciated for its simplicity and memory efficiency, particularly when tackling large datasets. However, it sometimes oscillates and may converge slowly.

Momentum builds upon SGD by accelerating convergence in directions of consistent descent, although it can occasionally bypass the lowest point in regions where the cost function curves steeply.

Adagrad's strength lies in its adaptability to sparse data, but it tends to reduce the learning rate over time, which can stall progress.

RMSprop was introduced to counter Adagrad's diminishing learning rate, making it effective for online and non-stationary tasks where data characteristics frequently change.

Adam is more often the preferred method because it merges the advantages of momentum and adaptive learning rates, generally performing robustly in most deep learning scenarios, though it may require careful parameter tuning to reach optimal performance.

(AAA in our case)

K. Bias-variance trade-off and resampling techniques

Start with the mean squared error:

$$C(\boldsymbol{X}, \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}\left[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2\right]$$

where $y = f(x) + \varepsilon$, with $\mathbb{E}[\varepsilon] = 0$ and $\mathrm{Var}(\varepsilon) = \sigma^2$.

Expanding the square:

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

Based on the definition of variance:

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

We have also:

$$\mathbb{E}[(y)^2] = \mathbb{E}[(f+\varepsilon)^2] = \mathbb{E}[(f)^2] + 2\mathbb{E}[f\varepsilon] + \mathbb{E}[\boldsymbol{\varepsilon^2}]$$

where $\mathbb{E}[\boldsymbol{\varepsilon^2}] = \sigma^2$ Expanding the expectation value of f:

$$\mathbb{E}[f(x)] = \int f(x)\, P_y(y)\, dy = f(x) \int P_y(y)\, dy = f(x)$$

That because the probability is normalized to one. This is called unbias expectation value because the expected value of the function is equal to the function itself. Then we have:

$$\mathbb{E}[f\varepsilon] = f \int \varepsilon\, P_\varepsilon(\varepsilon)\, \varepsilon = f\mathbb{E}[\varepsilon] = 0$$

That because $\mathbb{E}[\varepsilon] = 0$ So that means that:

$$\mathbb{E}[(y)^2] = f^2 + \sigma^2$$

Now we compute $\mathbb{E}[y\tilde{y}]$:

$$\mathbb{E}[y\tilde{y}] = \mathbb{E}[(f+\varepsilon)\tilde{y}] = \mathbb{E}[f\tilde{y}] + \mathbb{E}[\varepsilon\tilde{y}]$$

If we now assume that there is a probability distribution behind the model that does not act in the space of f(x), we can pull f outiside as a constant:

$$\mathbb{E}[f\tilde{y}] = f\mathbb{E}[\tilde{y}]$$

Then for the second term we assume that ε and $\tilde{y}$ are independent variables, so we can use the product rule and this term becomes:

$$\mathbb{E}[\varepsilon\tilde{y}] = \mathbb{E}[\varepsilon]\mathbb{E}[\tilde{y}] = 0$$

In the end we obtain:

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

but

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

This implies:

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

1. Bias:
$\mathrm{Bias}[\tilde{y}] = \mathbb{E}[(f-\mathbb{E}[\tilde{y}])^2]$
It's the squared difference between the true function f(x) and the average prediction $\mathbb{E}[\tilde{y}]$ of the model over different training sets.

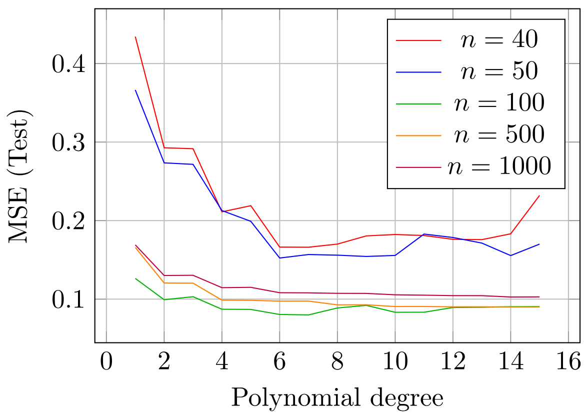
- Measures systematic error introduced by the model. - High bias $\rightarrow$ model is too simple and cannot capture the true relationship (underfitting). - Low bias $\rightarrow$ model is flexible enough to capture the underlying pattern.

2. Variance:
$\mathrm{Var}[\tilde{y}]$

III. RESULTS

Insert figures, tables, and discussions.

Test MSE vs Polynomial degree



IV. DISCUSSION AND CONCLUSION

Summarize findings, bias-variance trade-off, comparisons.