Regularized Linear Regression Methods Applied to Runge's Function
FYS-STK4155 - Project 1

Amund Solum Alsvik & Jørgen Vestly
University of Oslo

(Dated: October 6, 2025)

Abstract

The process of fitting models to functions is important in order to get a grasp of the shape and trend of the raw data the models are trained on. Approximating a one-dimensional function with high-order polynomial interpolation, makes it easier to predict new unseen data and to smooth out noise in the original data. The challenge with fitting to the Runge's function, is that for higher-order polynomials we get oscillating behavior at the edges of the function. To address this issue, we investigated the effect of regularization in linear regression models applied to this particular problem, using both analytical- and gradient descent methods. Our main finding was that, despite OLS giving the lowest MSE at degree 10, we found that the use of L1 and L2 regularization gave a smoother fit at the same degree. This tells us, that in the critical regions where polynomials are prone to oscillations, regularization makes better predictions on unseen data. Moreover, it suggests that MSE is not sufficient metric to evaluate a fitted model. We also showed with resampling methods that with increasing degrees, OLS tended to become somewhat overfitted compared to Ridge and LASSO.

1. INTRODUCTION

Artificial intelligence has become deeply integrated in the modern world, not only for researchers but also for the population as a whole. The introduction of consumer based large language models such as ChatGPT has brought artificial intelligence into the main stream. The theoretical framework behind artificial intelligence is machine learning. This concept may sound exotic, but the overarching method is based on ideas from elementary multi-variable calculus, namely minimization of cost functions. While many of the basic machine learning methods are conceptually simple, navigating through the large catalog of different approaches and their tradeoffs can be daunting. The aim of this report is to present, apply and compare some of the fundamental methods in machine learning, and also discuss their strength and weaknesses.

The methods used is variants of linear regression models. In this project, we are presented with a vector of data points $y \in \mathbb{R}^n$ which includes random noise $\epsilon \sim N(0, \sigma^2)$. Our data is generated from input points $x \in \mathbb{R}^n$, and we assume that y can be written as

$$y = f(x) + \epsilon,$$

where f is typically an unknown continuous function, such as a polynomial, trigonometric or exponential function. In this project, however, we chose an explicit continuous function

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

called *Runge's Function*, where we evaluate the function on the interval $[-1, 1]$. This choice is not arbitrary. The function is particularly interesting for us, due to something called Runge's phenomenon. In short, interpolating this function with polynomials can result in an oscillating behavior on the edges of the interval $[-1, 1]$, particularly for methods without regularization. For more details on this, read [1]. We will demonstrate this issue and how regularization can improve it.

In section 2 we introduce the methods we use and the theory behind them, while discussing their applications, strengths and weaknesses. In section 3 we show how we implemented these methods, before we finish of with section 4, where we present our results and discuss them. We have also included an appendix A, where we show the derivation of some of the central equations used throughout the project.

2. THEORY AND METHODS

We can equivalently rephrase our dataset $y = f(x) + \epsilon$ as

$$y = X\beta + \epsilon.$$

Where $X \in \mathbb{R}^{n \times p}$ is a *feature matrix*, whose columns contain the chosen basis functions we use to approximate the data, which in our case will be polynomial orders of the input points x. We assume that there exists what we call a *true parameter* $\beta \in \mathbb{R}^p$, such that $X\beta = f(x)$. The goal of the methods studied in this report is to construct an estimate $\hat{\beta}$ of the unknown true parameter from the noisy data y. Thus approximating the true signal $f(x)$ with a linear model

$$\tilde{y} = X\hat{\beta}.$$

The choice of using a polynomial basis in our feature matrix is not arbitrary. Recall the *Weierstrass approximation theorem.*

Theorem ([2])**.** *The polynomials are dense in the space of $C([a,b],\mathbb{R})$ for all $a, b \in \mathbb{R}$, $a < b$. In other words, for each continuous function $f : [a,b] \to \mathbb{R}$, there is a sequence of polynomials (p_n) converging uniformly to f.*

The theorem states that we can approximate any continuous function on a closed real interval arbitrarily well with a polynomial, which justifies our approach of using a polynomial basis to approximate Runge's function.

2.1. Evaluation of Models and Cost Functions

For all our models, our main evaluation tool is the mean squared error, which is given by the function

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

Squaring the differences between the data points and our model ensures that large differences are penalized, while differences less than 1 has a more negligible contribution. This encourages the model to fit data points reasonably well, while strongly discouraging predictions that deviate substantially from the observations. We also have a complimentary way to evaluate our models, called the R^2 number, given by

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

where $\bar{y}$ denotes the mean value of y. We see that if $R^2 \approx 1$, we have an almost perfect fit. However, if R^2 is close to 0, then the difference between the data points and our predictions are almost equal to the difference between the data points and the mean, meaning our model is approximately giving the mean values of y. We can also get a negative R^2 value, which would imply that our model does worse than using the mean values as predictors. Essentially, R^2 measures how well the model explains the general behavior of the dataset.

While we use the MSE to evaluate the performance of our models, each regression model is defined by a *cost function* $C(X,\beta)$. The cost function of a model determines what our model tries to accomplish. We define our optimal parameter as

$$\hat{\beta} = \min_{\beta\in\mathbb{R}^p} C(X,\beta).$$

In the scope of this report all the cost functions will be convex, hence for differentiable cost functions we can find its global minimum by solving the equation

$$\frac{\partial C(X,\beta)}{\partial\beta} = 0$$

for β.

One of the most important aspects in evaluating a regression model, is its ability to generalize. Meaning how well the model performs on unseen data. A common way of assessing this is splitting the dataset into a training set and a test set, where the training set usually consist of around two thirds to four fifths of our data. The model computes an estimator with respect to the training set, and the resulting coefficients are then evaluated on the testing set. We say that the model is generalizable if error is low on both the training and test data.

We now turn to different regression methods, starting with the ordinary least squares method.

2.2. Ordinary Least Squares

In *ordinary least squares* or OLS, we use the mean squared error as our cost function. Hence, our optimal parameter is given by

$$\hat{\beta}_{\mathrm{OLS}} = \min_{\beta\in\mathbb{R}^p} \mathrm{MSE}(\beta) = (X^TX)^{-1}X^Ty. \text{ (A 1)}.$$

Considering that we use MSE as our main way of evaluating models in general, OLS might seem as the most natural approach to regression, as we are minimizing the MSE function. However, it is not necessarily always the most effective one. First, while the matrix $(X^TX)^{-1}$ is always positive semi-definite, it can have eigenvalues that are 0 or very small. In the first case, the matrix would not be invertible, making an analytical expression for the optimal parameter impossible. In the second case, computing the inverse may lead to numerical instability. OLS will typically give the lowest MSE with respect to the training data, but it can suffer in terms of generalizability. We will go into more details on this in the subsequent sections.

2.3. Ridge Regression

As mentioned, X^TX may have eigenvalues that are very small or 0. A natural way to address this is to introduce a regularization parameter $\lambda > 0$, and add a diagonal matrix λI to X^TX. This leads to the *Ridge regression* cost function

$$C_{\mathrm{Ridge}}(X,\beta) = \frac{1}{n}\|y - X\beta\|_2^2 + \lambda\|\beta\|_2^2.$$

This is the MSE we saw from OLS in addition to a penalty term. To minimize this cost function, one has to balance minimizing MSE, while also minimizing the penalty term. The optimal Ridge parameter is given by

$$\hat{\beta}_{\text{Ridge}} = (X^T X + \lambda I)^{-1} X^T y. \text{ (A 2)}$$

Since $\lambda > 0$, we know that $X^T X + \lambda I$ will be positive definite and always invertible, which improves numerical stability. Moreover, the penalty term $\lambda \|\beta|_2^2$ in the Ridge cost function discourages large coefficients, where the size of λ determines how strongly they are penalized. By shrinking the coefficients, Ridge regression produces more stable models, particularly when the data is highly correlated or noisy as seen on page 64 of [3]. We will return to why this shrinkage of coefficients is beneficial in the next section on the bias-variance tradeoff.

2.4. Bias-Variance Tradeoff

Different methods have different bias and variance. The *bias* of an estimator $\hat{\beta}$ with respect to a true parameter β is given by the difference between its expectation and the true parameter. Namely

$$\text{Bias}(\hat{\beta}) = \mathbb{E}[\hat{\beta}] - \beta.$$

The expectation of the OLS parameter is given by

$$\mathbb{E}[\hat{\beta}_{\text{OLS}}] = \beta, \text{ (A 3)}$$

hence it has no bias. While its variance is given by

$$\text{Var}[\hat{\beta}_{\text{OLS}}] = \sigma^2 (X^T X)^{-1}, \text{ (A 4)}$$

where σ^2 is the variance of the noise ϵ. The expectation of the Ridge estimator is given by

$$\mathbb{E}[\hat{\beta}_{\text{Ridge}}] = (X^T X + \lambda I)^{-1} (X^T X)\beta. \text{ (A 5)}$$

Thus we see that for any $\lambda > 0$, Ridge regression will have a nonzero bias. On the other hand, the variance of the Ridge regression is given by

$$\text{Var}[\hat{\beta}_{\text{Ridge}}] = \sigma^2 (X^T X + \lambda I)^{-1} X^T X [(X^T X + \lambda I)^{-1}]^T. \text{(A 6)}$$

We see that the variance of OLS depends on $(X^T X)^{-1}$, which, as we mentioned earlier, can be problematic. There is a significant relationship between bias and variance when it comes to linear regression, called the *bias-variance tradeoff*. In the OLS setting, this tradeoff can be described by the following equation

$$\text{MSE}(y, \tilde{y}) = \text{Bias}[\tilde{y}]^2 + \text{Var}[\tilde{y}] + \sigma^2. \text{(A 7)} \qquad (1)$$

The bias of the model is the expected deviation of the mean of the model from the true data, meaning

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

On the other hand the variance term explains how much the predictions fluctuate across different training sets, and is given by

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

As usual σ^2 denotes the variance of the noise. Equation (1) tells us that a low MSE requires both a low bias and a low variance. While the decomposition is OLS specific, the same principle holds more generally. Methods without a regularization parameter, such as OLS, typically have low bias but can suffer from very high variance, due to large coefficients. Methods with regularization introduces some bias, but usually have lower variance.

Having both a low bias and low variance at the same time is not always feasible. In practice, bias is closely related to model complexity. With very low model complexity, we risk *underfitting*, meaning the model is not able to capture the underlying structure of the dataset. However, if the model is too complex, we risk what we call *overfitting*. Meaning that instead of capturing the structure of the dataset, it will instead adapt too well to the specific noise in the training data. If

$$y = X\beta + \epsilon,$$

we want our model to approximate the true parameter β without picking up too much of the noise ϵ. Models like OLS with very low bias, often compensates for this by producing large coefficients, which again makes the estimator sensitive to small changes in the training data, meaning it increases variance. When choosing a model, the goal is often to find a "sweet spot", where the model is complex enough to capture the structure of the data, but not so complex that we start overfitting to the noise in the dataset.

A graphical interpretation of this phenomenon and sweet spot can be seen in Figure 1. We see that the test sample has the lowest prediction error when model complexity and variance is at an intermediate level. In our results section, we have reproduced similar plots specific to our experiments.

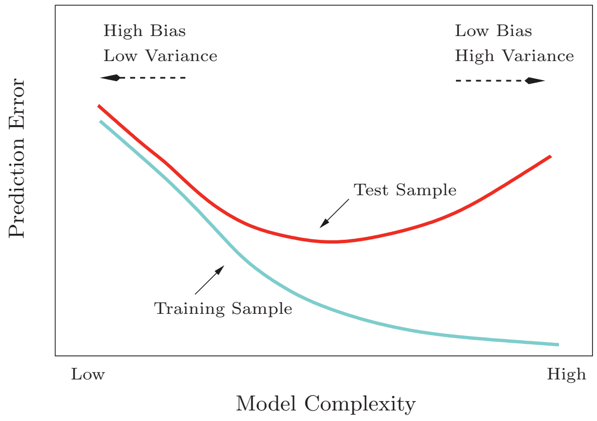


Figure 1: Illustration of the bias–variance tradeoff. Taken from page 38 in [3].

2.5. Gradient Descent

In both OLS and Ridge regression, we have presented closed-form expressions for the optimal parameters. However, in some cases it is not always possible to compute such estimators analytically, forcing us to use numerical methods.

We mentioned earlier that the cost functions we consider in this project are convex, meaning their gradients will be zero at the global minimum. Thus, we get a natural approach to minimizing them numerically. We start with an initial guess of the parameter, often the zero vector. We then iteratively update the parameter by moving in small steps in the direction of the negative gradient of the cost function. We continue this iterative process until we see that our parameter begins to converge, at which point we will have found an approximate minimum of the cost function. The main workhorse behind this method is the following result from multi-variable calculus.

Theorem (Translated from [4])**.** *Assume $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable in a point $a \in \mathbb{R}^n$. The gradient $\nabla f(a)$ points in the direction of the steepest increase of f at a, and the slope in this direction equals $\|\nabla f(a)\|$.*

This approach is the main idea behind *gradient descent.* In gradient descent, the iterative formula for the parameter θ for a cost function $C(X, \theta)$ is given by

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

where the step size $\eta > 0$ is what we call the *learning rate.* In the OLS case, the iterative formula becomes

$$\theta^{(i+1)} = \theta^{(i)} - \eta \frac{2}{n}(X^T(X\theta^{(i)} - y)), \text{(A 8)}$$

while in the Ridge case, the iterative formula is

$$\theta^{(i+1)} = \theta^{(i)} - \eta(\frac{2}{n}X^T(X\theta^{(i)} - y) + 2\lambda\theta^{(i)}). \text{(A 9)}$$

The choice of learning rate can affect how good our approximation of the minimum is. If we choose a large learning rate we risk going past the minimum, which will result in the algorithm oscillating around the minimum instead of converging. A smaller learning rate generally leads to a better approximation, but may require many iterations to approach the minimum. When doing gradient descent, it is common to define a stopping criterion $\delta > 0$, where the algorithm terminates once the norm of the gradient at the current step becomes less than δ, meaning when

$$\|\nabla_\theta C(X, \theta^{(i)})\| < \delta.$$

Typically, we choose δ to be a very small number, indicating that further iterations would only make negligible improvements.

While gradient descent indeed is an effective way to approximate the estimators for both Ridge and OLS, there are situations where the classic gradient descent method can be ineffective. Firstly, the gradient descent method is heavily dependent on the choice of the learning rate η, and, as we explained earlier, the best choice of the learning rate can vary on the situation. Ideally, we would adapt our step size based on the landscape of the cost function, such that η is small when the landscape becomes steep, but large when the landscape is flatter. Secondly, in more complex machine learning models, the cost functions are usually not convex. Instead they might have several local minima and saddle points. In these cases, the gradient descent algorithm can get stuck in one of these points, even though there is a more optimal minimum point somewhere else. Thirdly, if we have a big dataset, the gradient descent algorithm can become computationally expensive. At each iteration we compute the gradient

$$\nabla_\theta C(X, \theta),$$

meaning for n data points, we compute a matrix-vector product with all n rows of X, which can be problematic when n gets very large. These issues leads us to some refinements of the standard gradient descent methods.

A way to address the problem of choosing a learning rate is using *momentum-based gradient descent.* The idea behind this is to let each iterative step have memory of the last, and adjust its step size accordingly. By doing this, we can avoid oscillating around the edges of a minimum point, and instead converge more smoothly. We follow the mathematical formulation of chapter 7.1.2 in [5]. We define the update between two iterations as $\Delta\theta^i = \theta^{(i)} - \theta^{(i-1)}$. We then define a new iterative algorithm

$$\theta^{(i+1)} = \theta^{(i)} - \eta\nabla_\theta C(X, \theta^{(i)}) + \alpha\Delta\theta^{(i)},$$

where $\alpha \in [0, 1]$ is the *momentum parameter*, and $\eta > 0$ is the learning rate. Hence, with the new iterative algorithm, the update is equal to

$$\Delta\theta^{(i)} = \alpha\Delta\theta^{(i-1)} - \eta\nabla_\theta C(X, \theta^{(i-1)}).$$

With this new iterative formula, depending on the choice of momentum parameter, the step size will decrease when the last update was small. On the other hand, if the last update was large, the step size will increase, speeding up in flatter regions. This allows us to reduce oscillation, and more quickly converge to the minimum. There are other approaches to adaptively change the step size. Such as *ADAM*, *AdaGrad* and *RMSProp*. We will not explain these in detail here, but we will showcase these methods in our implementation section.

There is also a method for addressing the problems with getting stuck in local minima and the computational expense of gradient descent. This leads us to the variant of gradient descent called *stochastic gradient descent*. This method will also be shown in our implementation section, but we will explain the general idea here, where we follow the derivation from chapter 7.1.2 of [5], chapter 4 of [6] and the lecture notes written by *Hjort-Jensen* [7]. Assume our dataset $\mathbb{D}$ has n data points. The cost function can be written as a sum over the set of data points

$$C(X, \theta) = \sum_{i=1}^{n} c_i(x_i, \theta).$$

By linearity of the derivative, the gradient can then be written as

$$\nabla_\theta C(X, \theta) = \sum_{i=1}^{n} \nabla_\theta c_i(x_i, \theta).$$

In standard gradient descent, we compute the estimator by using the entire dataset, or alternatively the entire training set. However, in stochastic gradient descent, we split up our dataset in what we call *mini-batches*, meaning we partition the dataset in equally large subsets. We then pick one of these mini-batches at random, for instance mini-batch $D_k \subset \mathbb{D}$. We then approximate our gradient by letting

$$\nabla_\theta C(X, \theta) := \sum_{i \in D_k} \nabla_\theta c_i(x_i, \theta).$$

Immediately, we see that this is much less computationally expensive than taking the gradient for the whole dataset, but how does it help with escaping local minima? The word stochastic in stochastic gradient descent comes from the fact that our approximate of the gradient will contain noise. Since the gradient is an approximation, it may not always points directly to the local minimum, instead due to random variation in our approximation, the direction of the gradient may point in a slightly different direction, allowing our algorithm to escape bad local minimum points.

The choice of mini-batch size is also important. While having large mini-batches gives ut better approximations of the gradient, they will also be more computationally expensive. Moreover, as touched upon earlier, the noise in the approximation is what allows stochastic gradient descent to escape bad local minima. Hence, having a more noisy gradient approximation, meaning choosing smaller mini-batch sizes, gives a better chance to escape bad local minimum points.

2.6. Lasso Regression

We present one more regression method that can occasionally be useful. Namely, *Lasso regression*, where Lasso stands for least absolute shrinkage and selection operator. The cost function for Lasso is given by

$$C_{\text{LASSO}}(X, \beta) = \frac{1}{n}\|y - X\beta\|_2^2 + \lambda\|\beta\|_1. \tag{2}$$

While this cost function is very similar to the cost function of Ridge regression, there is a subtle difference. Our penalty term uses the L_1-norm instead of the L_2-norm. Similar to the Ridge cost function, minimizing the Lasso cost function forces us to balance between minimizing the standard MSE and minimizing the penalty term. Again this will penalize large values, but it also has some added benefits. If we have features in our model that is not very important, Lasso regression can set the corresponding coefficients to zero. This happens because of the specific optimization landscape of the L_1-norm, but the details of this is beyond the scope of this report. See chapter 3.4 in [3] and lecture notes by *Hjort-Jensen* [8] for more details on this.

There is another peculiarity to Lasso regression. Since the L_1-norm is not differentiable, we cannot find a closed expression for the gradient of the Lasso cost function, however there are ways to work around this. We later implement Lasso regression using gradient descent.

2.7. Scaling Data

In many situations, *scaling* our data can be useful. A typical example of scaling the data is transforming the feature matrix such that the columns of the feature matrix has mean equal to 0 and variance equal to 1. Another way is to ensure that all the features lie between 0 and 1. [9]. One can also center the y-data, meaning that we subtract the mean of the entries in y from y. This removes the need for an intercept column in the feature matrix. [10]

One of the obvious reasons for scaling data is if our features come in different units. For example, if we have a model that tries to predict a persons height based on their weight, and height is given in meters while weight is given in grams, the magnitude of the feature for weight will be much larger than for height. In this situation, we risk that the contribution of the the weight would dominate our prediction. This aspect is particularly important to keep in mind when we are using Ridge and Lasso regression, as both these methods penalize large values.

Typically, scaling is not as important for OLS. One can show that OLS is what we call invariant under scaling, meaning scaling will not affect the prediction as shown on page 128 in [6]. However, it can still be useful, as smaller values can lead to better numerical stability. A reason one might not want to scale data when using OLS, is when the variables correspond to specific units, such as mass or velocity. In that case, scaling could have a negative effect on the interpretability of the results.

2.8. Resampling Methods

A dataset can often be difficult to replicate, for instance, if our dataset consists of responses to a survey spanning over many years. However, there are some situations where one might want a similar dataset with difference noise, such as when we want to learn about the statistical properties of our prediction. A dataset can also be small, making it difficult to train and test our models. A common way to tackle this problem is with *resampling methods.* In this project we use two different resampling methods, *bootstrap* resampling and *cross-validation* resampling. We explain the algorithms for the two methods, but will not go into detail on the probabilistic technicalities. For more detail on this, see [10], and for examples on how this is implemented, see our implementation and results sections.

The bootstrap method is fairly straightforward. Assume we have a dataset $\mathbb{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. With replacement, draw n pairs (x_i, y_i) from our dataset, and define a bootstrap dataset $\mathbb{D}^* = \{(x'_1, y'_1), \ldots, (x'_n, y'_n)\}$. Meaning this new dataset likely includes duplicates, while leaving other data points out. Then compute a parameter $\hat{\theta}^*$ with respect to the bootstrap dataset, in the same procedure as we would compute $\hat{\theta}$. Now repeat this process k times to get $\hat{\theta}^{*(1)}, \ldots, \hat{\theta}^{*(k)}$. It turns out that if we now make a histogram of the relative frequency for each entry $\hat{\theta}_j^{*(i)}$, we obtain an estimate of the probability distribution of the coefficient $\hat{\theta}_j$. [10] From here, we can compute the statistical properties of $\hat{\theta}$, such as variance.

Cross-validation is often used when we have limited data to train and test on. The algorithm goes as follows. We shuffle our dataset randomly and split it into k equally sized groups, commonly called *folds.* We then choose one of the folds to be our test set, while using the rest of the folds as the training set. We then train our model with the training data, and evaluate it on the test data. We then repeat the procedure with a different test fold, until all the folds have been used as a test group once. From here, we typically take the mean of the testing error, and use this to evaluate our model.

One of the main benefits of this this is that we get a more reliable estimate of the models generalizability, as our evaluation is much less dependent on how we split our dataset into train and test. This particularly applies for smaller datasets. It is worth noting that the k-fold cross validation algorithm is dependent on choice of k. Choosing a $k = n$ means that we train on the entire dataset except for one point, which is called LOOCV (leave-one-out-cross-validation). This leads to lower bias. Choosing $k = 2$ means that we train on half the data set, and test on the other, leading to higher bias but lower variance. A standard value is $k = 10$. [11]

3. IMPLEMENTATION

In this part, we present an overview of how the different methods and algorithms in the theory section is implemented as code. Furthermore, it will include demonstrations of how results were obtained by displaying sample runs and tuning of parameters.

3.1. Program architecture

The codebase is organized into four main modules. The **functions.py** script provides the core utilities such as polynomial feature construction, closed-form OLS and Ridge estimators, gradient calculations, and evaluation metrics like MSE and R Squared. The **classes.py** script contains object-oriented implementations of optimization routines, including standard gradient descent, momentum, stochastic variants, AdaGrad, RMSProp, and Adam, along with a resampling class for bootstrap and cross-validation. The **plots.py** script serves as the experimentation layer, combining the core functions and optimization classes to run tests, evaluate models, and generate figures such as error vs. polynomial degree, coefficient paths, heatmaps, and convergence comparisons. Finally, **test.py** acts as a simple driver to reproduce the plots and results by calling the routines in **plots.py** with predefined settings. Together, this modular structure separates numerical routines, optimizers, visualization, and drivers, making the code easier to maintain, test, and extend. To check out the full codebase along with documentation, visit the GitHub repository [12].

3.2. Algorithms

Gradient-based Algorithms

The ideas for implementing the various gradient descent methods was taken from section 8.3-8.5 in [13].

a. Ordinary Gradient Descent. For ordinary gradient descent, along with momentum and stochastic, we're only going to show pseudocode. A more in-depth account about these methods is found in the theory part 2 2.5

Algorithm 1 Ordinary Gradient Descent

```
Require: Learning rate $\eta > 0$, number of iterations $T$, objective $C(X, \theta)$ with gradient $\nabla C(X, \theta)$
Ensure: Final parameters $\theta$
  Initialize parameters $\theta \leftarrow 0$
  for $t = 1$ to $T$ do
      $g \leftarrow \nabla C(X, \theta)$
      $\theta \leftarrow \theta - \eta \cdot g$
  end for
  return $\theta$
```

Algorithm 2 Ordinary Gradient Descent with Momentum

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, momentum $\alpha \in [0, 1)$, steps $T$
Initialize $v \leftarrow 0$, $\theta \leftarrow \theta_0$
for $t = 1 \to T$ do
    $g \leftarrow \nabla_\theta C(X, \theta)$
    $v \leftarrow \alpha v - \eta g$            ▷ velocity update
    $\theta \leftarrow \theta + v$
end for
return $\theta$
```

b. Gradient Descent with Momentum.

Algorithm 3 Stochastic Gradient Descent (SGD)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, minibatch size $m$, steps $T$
Initialize $\theta \leftarrow \theta_0$
for $t = 1 \to T$ do
    Sample minibatch $B = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$
    $g \leftarrow \frac{1}{m} \sum_{(x,y) \in B} \nabla_\theta C(X, \theta)$
    $\theta \leftarrow \theta - \eta\, g$
end for
return $\theta$
```

c. Stochastic Gradient Descent.

d. AdaGrad. AdaGrad rescales each parameter by the square root of all past squared gradients plus a small constant, so frequently changing parameters take smaller steps while others can move more.

Algorithm 4 Gradient Descent with AdaGrad

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, stability constant $\delta > 0$, steps $T$
Initialize $r \leftarrow 0$ (same shape as $\theta$), $\theta \leftarrow \theta_0$
for $t = 1 \to T$ do
    $g \leftarrow \nabla_\theta C(X, \theta)$
    $r \leftarrow r + g \odot g$
    $\theta \leftarrow \theta - \frac{\eta}{\sqrt{r} + \delta} \odot g$
end for
return $\theta$
```

Here, $\odot$ denotes elementwise multiplication.

e. RMSProp. RMSProp uses the gradient and divides each component by the square root of past squared gradients plus a small constant, so the step sizes adapt separately for each parameter.

Algorithm 5 RMSProp

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, decay $\rho$, stability $\epsilon$, steps $T$
Initialize $r \leftarrow 0$, $\theta \leftarrow \theta_0$
for $t = 1 \to T$ do
    $g \leftarrow \nabla_\theta C(X, \theta)$
    $r \leftarrow \rho r + (1 - \rho)(g \odot g)$
    $\theta \leftarrow \theta - \eta\, g / (\sqrt{r} + \epsilon)$
end for
return $\theta$
```

f. Adam. This method combines momentum and RMSProp. It maintains both a moving average of gradients and squared gradients, with bias corrections:

Algorithm 6 ADAM

```
Require: Initial parameters $\theta_0$, step size $\eta$, decay rates $\beta_1, \beta_2 \in (0, 1)$, stability $\delta > 0$, steps $T$
Initialize $m \leftarrow 0$, $v \leftarrow 0$, $\theta \leftarrow \theta_0$
for $t = 1 \rightarrow T$ do
    $g \leftarrow \nabla_\theta C(X, \theta)$
    $m \leftarrow \beta_1 m + (1 - \beta_1) g$                ▷ update 1st moment
    $v \leftarrow \beta_2 v + (1 - \beta_2)(g \odot g)$       ▷ update 2nd moment
    $\hat{m} \leftarrow \frac{m}{1 - \beta_1^t}$              ▷ bias correction 1st moment
    $\hat{v} \leftarrow \frac{v}{1 - \beta_2^t}$              ▷ bias correction 2nd moment
    $\theta \leftarrow \theta - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \delta}$
end for
return $\theta$
```

g. LASSO For LASSO gradient descent, we use soft threshold for the updated parameters in each method.

Algorithm 7 Ordinary Gradient Descent w/LASSO

```
Require: Learning rate $\eta > 0$, number of iterations $T$, objective $C(X, \theta)$ with gradient $\nabla C(X, \theta)$
Ensure: Final parameters $\theta$
    Initialize parameters $\theta \leftarrow 0$
    for $t = 1$ to $T$ do
        $g \leftarrow \nabla C(X, \theta)$
        $\alpha \leftarrow \eta \cdot \lambda$
        $\theta \leftarrow \theta - \eta \cdot g$
        $\theta \leftarrow soft(z, \alpha)$
    end for
    return $\theta$
```

In the script *classes.py* in [12], this soft threshold is integrated into every gradient descent method. See *functions.py* for implementation of the soft threshold function. The idea behind soft threshold is taken from [14].

3.3. Experimental Protocol

Machine learning is experimental in nature, and hence the method for deriving results is imperative in terms of evaluation and interpretation. A thorough protocol for conducting machine learning experiments is necessary in order to build up a testing framework and to make sense of the results. For the purpose of this project, we have different tradeoffs for different models and algorithms. If we were to reduce model evaluation to merely minimizing the MSE function, we would disregard other important aspects of a good machine learning model. Other considerations are time-and model complexity, bias-variance tradeoffs, numerical stability, and reproducibility. Later in this section, we'll consider some examples.

This is a general outline of how we performed experimental analysis in this project:

1. In this project, we scale our data for every model, as it seems to improve the results of the methods with regularization. As mentioned in the Theory and Methods section, OLS is invariant under scaling, however scaling did not seem to negatively affect OLS either. Furthermore, since our variables are unit less, and on the off-chance that it could improve numerical stability, we saw no reason to not scale the data for OLS as well. For a model or method of interest, we choose one performance metric (MSE, R^2, convergence etc.). Along with this, we choose one or two hyperparameters (η, λ, iterations, batch-sizes etc.). With this knowledge, we find the set of hyperparameters yielding the best performance of a chosen metric.

2. We use these hyperparameters as fixed inputs to our new method of interest. In the example, we will see how this works with stochastic gradient descent. We compute a new performance metric, and use this for further analysis.

3. Final step: we now obtain the final results, and have enough information to discuss the final performance of the model with a given set of hyperparameters and methods. We might validate our results with benchmarks - we'll use the scikit-learn functionality in the example.

In the following section, where we give accounts for our results, we'll see a consistent usage of the outline above. It is also, in addition to hyperparameter tuning, important to practice benchmark testing. In this report, we are implementing our own code for linear regression models. However, the python library scikit learn has functionality for most of the methods we use. Therefore, comparing results with scikit learn is useful for testing our code. (see below).

```python
def test_benchmark_Ridge():
    ridge = Ridge(alpha=0.01, fit_intercept
        =False)
    ridge.fit(Xtr_s, y_centered)
    y_pred = ridge.predict(Xte_s)

    testRidge = ridge(scaled_train,
        y_centered, lam=0.01)
    predRidge = scaled_test@testRidge

    return np.allclose(y_pred, pred_Ridge)
```

Listing 1: Example of comparing with sklearn

```
print(test_benchmark_Ridge())
---Output---
True
```

Listing 2: Program output

We use the function **allclose** from NumPy, returning True if the arguments are identical. In our case, we compare the prediction from scikit learn using the **Standard**

Scaler, to our own implemented code for Ridge and for scaling. As seen in output, this method returns True. This concludes that both our Ridge function and our code for scaling gives correct results.

3.4. Use of AI tools

We have used ChatGPT as a sparring partner for discussion of the theory, as well as for testing our intuition. In the Theory and Methods section, it was also used to clean up spelling mistakes, clarify arguments and to make general improvements in the text. Other than this ChatGPT have been used for help with the coding, particularly with finding errors in our code and debugging. It was also used for help with formatting the project in overleaf. We have included an overview of the prompts and replies we got from ChatGPT on our Github page [12].

4. RESULTS AND DISCUSSION

In this section we present our various result when applying analytical linear regression methods, as well as numerical linear regression methods to Runge's function. To add some complexity to our experiments, we include some noise in Runge's function. Unless otherwise specified, a noise vector $\epsilon \sim N(0, 0.1^2)$ was added to Runge's function. We have divided our results into three main sections. We begin by comparing results of our analytical models with respect to MSE and R^2. Then we move on to comparing the different gradient descent methods, before we finish off with results from the resampling methods and a discussion of the Bias-Variance tradeoff. In particular how this comes into play when fitting to Runge's function.

Unless otherwise specified, all computations of MSE are done on testing data. We have extensively tested our methods over different parameters, and generated plots for each test. In the report, we only include the plots that are the most visually insightful, however we will refer to other plots that can be found on our Github page [12], for some of the intermediate results..

4.1. Analytical Results

Through our testing, we found that $\lambda = 0.01$ is a good choice of regularization parameter for Ridge regression. Although the MSE generally improves for smaller λ, we found the difference between smaller choices of λ to be negligible for all polynomials degrees, and choosing a near zero λ would defeat the purpose of the methods. We therefore use this value for Ridge regression going forward. See our Github folder [15] for a heatmap of MSE as a function of choice of lambda and polynomial degree.

When comparing the MSE of Ridge and OLS over polynomial degrees and number of data points, the result was rather unsurprising. As can be seen in figure 3, increasing the number of polynomial degrees and data points has a positive effect on MSE. OLS generally performed slightly better than Ridge in regards to MSE, which is to be expected when comparing a method without bias to a method with bias. We also compared the R^2 scores of OLS and Ridge, as can be seen in figure 2. Although the results are similar, OLS did slightly better here as well, which is expected as it performed better on MSE. We also saw the same trends as with MSE, that the R^2 score improves when the number of data points and polynomial degrees increases. We also looked at the MSE and R^2 score for Runge's function without noise for both Ridge and OLS. As expected, the results became better, both lowering the MSE and improving the R^2 score. See our Github folder [15] for figures named with *no_noise*, that includes MSE and R^2 scores for Runge's function without noise.

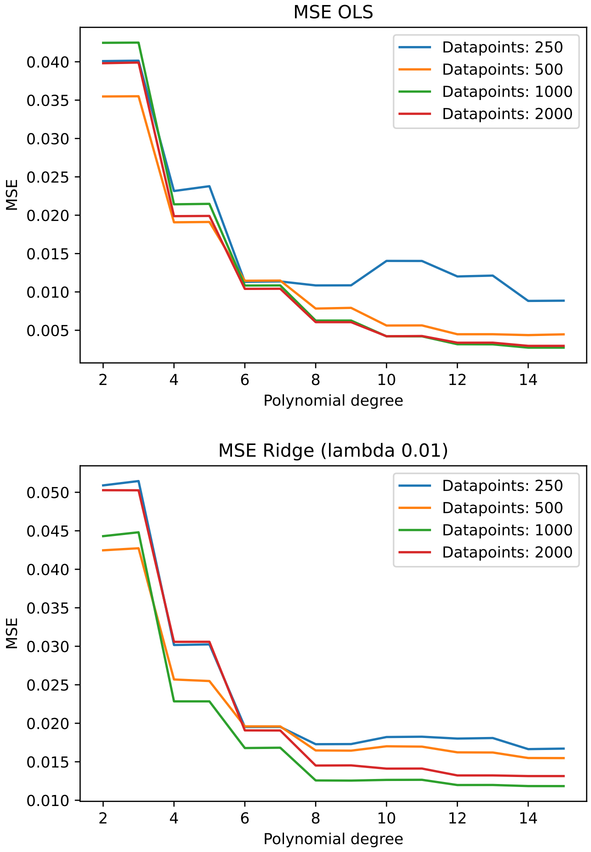


Figure 2: Comparison of OLS and Ridge MSE.

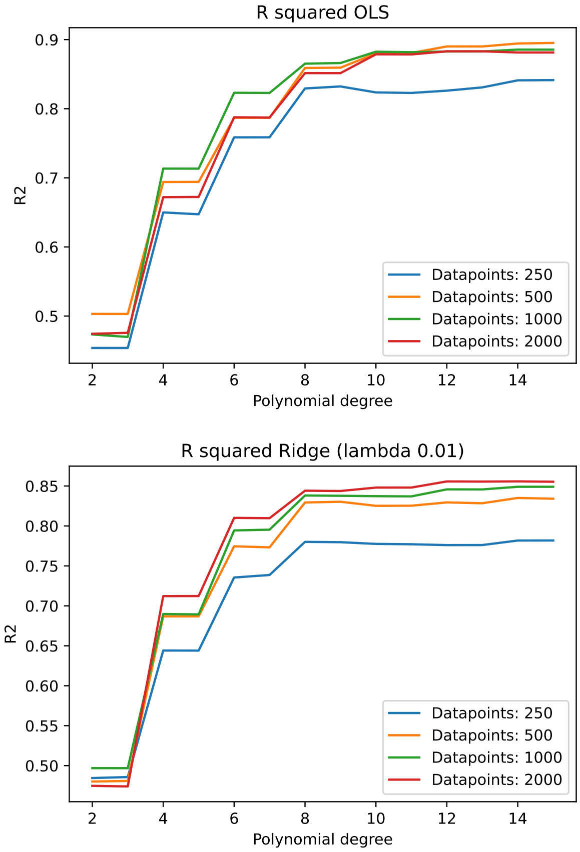


Figure 3: Comparison of OLS and Ridge R^2 score.

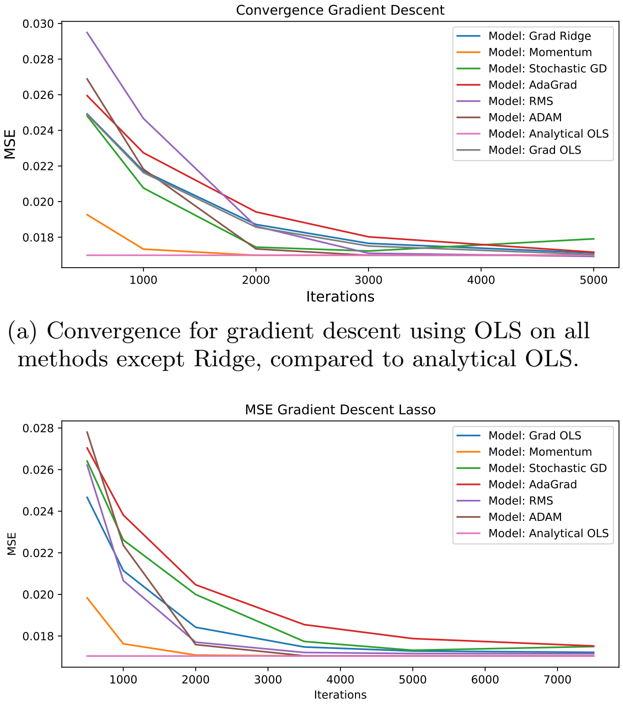
4.2. Gradient Descent Analysis

Through testing the various gradient descent methods, we tried to find the best hyperparameter for each method. From a mixture of informed guesses and experiments, we found that the hyperparameters in figure 4 gave the fastest convergence for each method. For AdaGrad and ADAM, we used the suggested default values from section 8.3-8.5 in [13], and later in this section we will see how we determined the best batch size for stochastic gradient descent. Check various figures in our Github folder [15] with a "GD" ending to see a comparison between choice of hyperparameters for each method. For the different learning rates, we simply tuned the methods until each converged the fastest. We used this as a foundation for comparing each and every gradient descent method.

Figure 5 shows when the different gradient descent methods converge, i.e. reach the MSE of the analytical OLS, as a function of iterations. We immediately see that, for both plots, momentum converges the fastest of the methods. This corresponds to our description of momentum in the theory section. For the adaptive learning rate methods (*AdaGrad, RMSprop, and ADAM)*, we see that ADAM converges quicker due to its in-built momentum. The reason why AdaGrad and RMS converge slower could be due to a lower effective learning rate. Ordinary gradient descent in figure (a) converges at about the same rate for both Ridge and OLS, and this is in accordance with the theory because the learning rate is the same for both. Stochastic gradient descent is typically harder to predict, and converges at different rates for OLS and LASSO. This might be due to random variations, rather than the intrinsic property of the method itself. For stochastic gradient descent we consistently chose batch size=100, since it gives the least unstable convergence across methods (see figure*stoc_batch_size.pdf* in our GitHub folder [15]).

Method	η	Other params
Grad OLS	0.15	-
Grad Ridge	0.15	λ=0.0001
Momentum	0.05	β=0.9
Stochastic	0.2	Batch size=100
AdaGrad	0.05	ρ=0.99
RMSProp	0.001	-
ADAM	0.005	β1=0.9, β2=0.999

Figure 4: Optimizer hyperparameters used in training.



(a) Convergence for gradient descent using OLS on all methods except Ridge, compared to analytical OLS.

(b) Convergence for gradient descent using LASSO.

Figure 5: Convergence for different Gradient Descent methods.

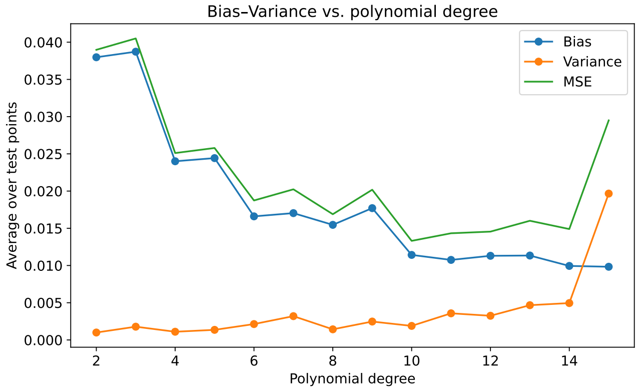


Figure 6: Bias-variance and MSE as function of polynomial degree for bootstrap=1000, data points=100, and noise $\epsilon \sim N(0, 0.1^2)$.

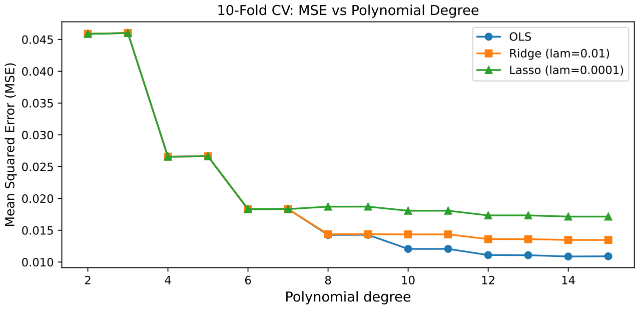


Figure 7: 10-fold comparison for different Linear Regression models, with 100 data points.

4.3. Resampling results and Bias-Variance discussion

In figure 6, there is a balance between bias and variance across model complexity. The MSE is also low using these parameters, which is subject to a balanced and well-performing model. It is *well-performing* in the sense that the MSE is lower compared to that of models having other combinations of bootstraps, data points, and noise. It is *balanced* since the difference between the bias of the model and its variance is relatively small. From the figure, we note that the difference between bias and variance is at its lowest for degree 14. Nevertheless, MSE is slightly lower for degree 10 despite the model being slightly more biased. However, the relative difference between bias and variance is satisfactory low. As a result, degree 10 was concluded to give the best tradeoff for our experiments, and we chose this degree going forward.

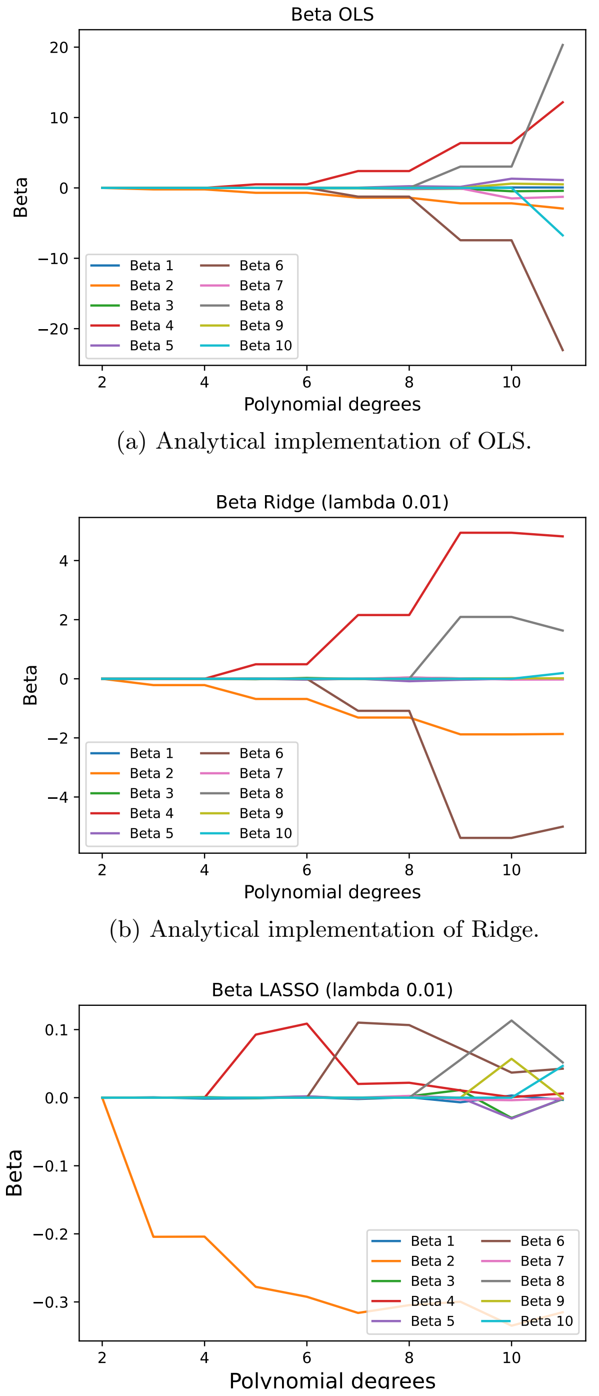
We also experimented with different bootstraps, data points and noise to induce different combinations of bias and variance (see the *g_bvto* plots in our Github folder [15]).

Using 10-fold cross-validation, we get a lower MSE for polynomial degrees higher than 7 for analytical OLS -and Ridge, than for the bootstrap example in 6. Bootstrapping trains on fewer data points than 10-fold (63% versus 90%), see ch. 7.11 in [3]. Therefore, there is less information per fit, often yielding a worse result. Also, for $k = 10$, both bias and variance are low, making the model perform better. Also, as can be seen in figure 9, consistently get a higher MSE when using LOOCV. Since 100 folds for 100 data points introduces the highest possible variance for our dataset, OLS will eventually perform worse than Ridge, and even equal to LASSO (w/high λ) for degree 15. For even lower values of λ, we get that OLS actually performs worse than LASSO (see *100fold_lowl1* in [15]). This is because OLS is prone to high variance, while the models with regularization is less susceptible to this. This suggests that OLS is more prone to overfitting for higher degrees. In our analysis, we found that using $k = 10$ gave the best results for MSE.

We see the benefits of the regularization parameter in Lasso and Ridge immediately when comparing the coefficients OLS, Ridge and Lasso produces respectively, as in figure 8. When polynomial degrees increases, we see that the corresponding coefficients increase quite drastically in OLS, going between -20 and 20, while the increase is more moderate in Ridge, going between -4 and 4. This difference is even more apparent when looking at the coefficients produced by Lasso. We see that the variation between coefficients is small and several coefficients become zero, which corresponds to the theoretical motivation for the method. As explained in the Theory and Methods section, large coefficients is pathological for OLS, potentially hurting its generalizability. A symptom of this is Runge's phenomenon as we touched upon in the introduction, where OLS oscillates on the edges of the interval $[-1, 1]$, while Ridge and Lasso gives more stable results. Our findings culminate with figure 10, where we see that the OLS fit oscillates near the edges of the interval, while this is somewhat mitigated for Ridge and in particular Lasso.

5. CONCLUSION

As mentioned, we divided our results section into three subsections. Firstly, we found that analytical OLS does better on both MSE and R^2 compared to analytical Ridge. Moreover, we found that the results improved for both methods, when fitting to Runge's function without noise. Secondly, when comparing our Gradient descent methods, we found a set of good parameters for each method. We also saw that methods which include momentum converges faster, than those that do not. Lastly, through resampling methods, we found the polynomial degree that seemed to be give the most favorable set of models. We also illustrated the negative effects of OLS having no bias, when comparing the coefficients of OLS, Ridge and Lasso, leading to Runge's



(a) Analytical implementation of OLS.

(b) Analytical implementation of Ridge.

(c) Ordinary gradient descent implementation of Lasso.

Figure 8: Coefficient values as a function of polynomial degree.

phenomenon. This implies that checking MSE and R^2 scores is not sufficient to fully evaluate a model.

While we illustrated several interesting properties of the methods, we could not completely show the benefits of the different gradient descent methods. The optimization landscapes of OLS, Ridge and Lasso are not particularly complex, hence it was difficult to fully harness the strength of each gradient descent method. For future research, it could be interesting to test the gradient descent methods for more complex cost functions, to see the benefits of these methods more clearly. Also, for the methods with adaptive learning rate, we chose the parameters that were generally advised in the literature. We did not extensively test different combinations of parameters, and there could be better parameters than those we found, which more clearly illustrates the strength and weaknesses of these methods.

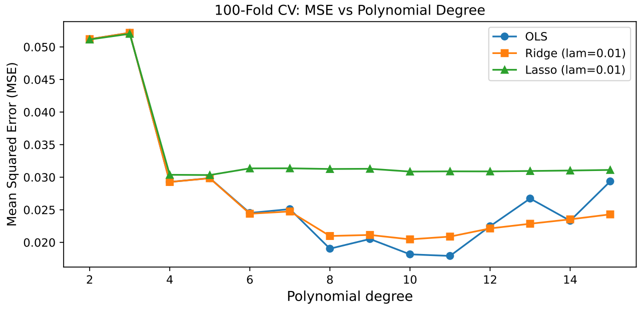


Figure 9: 100-fold comparison for different Linear Regression models, with 100 data points.

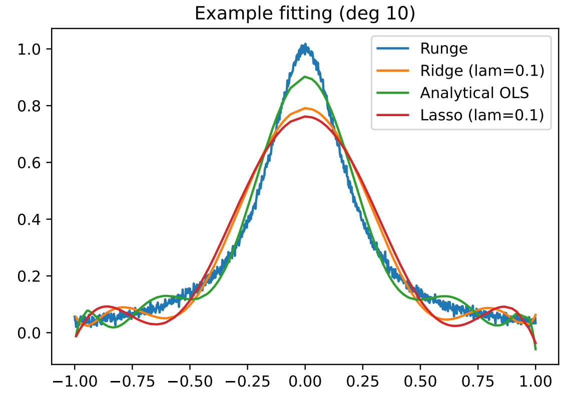


Figure 10: Comparison of analytical Ridge, analytical OLS and Lasso with gradient descent, when fitted to Runge's function.

[1] A. Cosgun and J. Nies, *Runge's phenomenon and its implications in interpolation*, https://math.uni.lu/eml/assets/reports/2022/runge.pdf (2022), accessed:

2025-09-24.

[2] T. L. Lindstrøm, *Spaces: An Introduction to Real Analysis*, vol. 29 of *Pure and Applied Undergraduate Texts* (American Mathematical Society, Providence, Rhode Island, 2017), ISBN 9781470440626.

[3] T. Hastie, R.Tibshirani, and J.Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition. Springer Series in Statistics* (Springer, New York, 2009), URL `https://link.springer.com/book/10.1007%2F978-0-387-84858-7`.

[4] T. L. Lindstrøm and K. Hveberg, *Flervariabel analyse med lineær algebra* (Gyldendal akademisk, Oslo, 2015), 2nd ed., ISBN 9788205472402.

[5] M. P. Deisenroth, A. A. Faisal, and C. S. Ong, *Mathematics for Machine Learning* (Cambridge University Press, Cambridge, 2020), ISBN 9781108455145, URL `https://mml-book.com`.

[6] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (O'Reilly Media, Sebastopol, CA, 2019), 2nd ed., ISBN 978-1492032649.

[7] M. Hjort-Jensen, *Optimization (chapter): Momentum-based gd and related methods*, `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapteroptimization.html#momentum-based-gd` (2025), accessed: 2025-09-24.

[8] Morten Hjort-Jensen, *Machine learning lecture notes, chapter 2*, `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/chapter2.ipynb` (2025), accessed: 2025-09-24.

[9] M. Hjort-Jensen, *Machine learning lecture notes, chapter 1*, `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter1.html` (2025), accessed: 2025-09-24.

[10] M. Hjort-Jensen, *Machine learning lecture notes — chapter 3: More on rescaling data*, `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter3.html#more-on-rescaling-data` (2025), accessed: 2025-09-24.

[11] S. Raschka, Y. H. Liu, and V. Mirjalili, *Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python* (Packt Publishing, Birmingham, UK, 2022), ISBN 9781801819312.

[12] A. Alsvik and J. Vestly, *Project source code*, `https://github.com/JVestly/Project-1/tree/main/` (2025).

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

[14] Morten Hjort-Jensen, *Machine learning programs, lasso.py*, `https://github.com/CompPhysics/MachineLearning/blob/master/doc/Programs/Regression/lasso.py` (2025), accessed: 2025-09-24.

[15] A. Alsvik and J. Vestly, *Project source figures*, `https://github.com/JVestly/Project-1/tree/main/Figures/` (2025).

Appendix A: Derivation of Central Equations

1. Optimal Parameter for OLS

We rewrite

$$\|y - X\beta\|_2^2 = (y - X\beta)^T (y - X\beta).$$

Let $u(\beta) = (y - X\beta)$ for y fixed. By the product rule for inner products we get

$$\begin{aligned}
\frac{\partial\,(y - X\beta)^T\,(y - X\beta)}{\partial\beta} &= \frac{\partial u(\beta)^T u(\beta)}{\partial\beta} \\
&= u(\beta)^T \frac{\partial u(\beta)}{\partial\beta} + u(\beta)^T \frac{u(\beta)}{\partial\beta} \\
&= -(y - X\beta)^T X - (y - X\beta)^T X \\
&= -2(y - X\beta)^T X.
\end{aligned}$$

By setting this expression equal to 0 and solving for β we get that

$$0 = (y - X\beta)^T X = (y^T - \beta^T X^T)X = \beta^T X^T X - y^T X.$$

Hence we get that

$$\beta^T X^T X = y^T X.$$

Which then implies that

$$\beta^T = y^T X (X^T X)^{-1}.$$

We know that $X^T X$ is self adjoint, hence it also has a self adjoint inverse. Thus we get that the optimal parameter is

$$\hat{\beta}_{\mathrm{OLS}} = ((X^T X)^{-1})^T X^T y = (X^T X)^{-1} X^T y.$$

2. Optimal Parameter for Ridge

We rewrite

$$\|y - X\beta\|_2^2 + \lambda\|\beta\|_2^2 = (y - X\beta)^T(y - X\beta) + \lambda\beta^T\beta.$$

For simplicity let

$$f(\beta) = (y - X\beta)^T(y - X\beta) + \lambda\beta^T\beta.$$

Then

$$\frac{\partial f(\beta)}{\partial\beta} = \frac{\partial(y - X\beta)^T(y - X\beta)}{\partial\beta} + \frac{\partial\beta^T\beta}{\partial\beta}.$$

From the derivation of the OLS parameter, we know that

$$\frac{\partial(y - X\beta)^T(y - X\beta)}{\partial\beta} = -2(y - X\beta)^T X.$$

Moreover we compute the partial derivative

$$\frac{\partial\beta^T}{\partial\beta_i} = \lambda\sum_{k=0}^{n-1}\frac{\partial\beta_k^2}{\partial\beta_i} = \lambda\sum_{k=0}^{n-1}2\beta_k\delta_{ki} = 2\lambda\beta_i.$$

Hence we find that

$$\frac{\partial\beta^T\beta}{\partial\beta} = 2\lambda\beta^T.$$

Meaning we have that

$$\frac{\partial f(\beta)}{\partial\beta} = -2(y - X\beta)^T X + 2\lambda\beta^T.$$

We must now solve $\frac{\partial f(\beta)}{\partial\beta} = 0$ for β. We assume that $\lambda > 0$ such that $(X^TX + \lambda I)$ is self adjoint and invertible. We get that

$$-(y - X\beta)^T X + \lambda\beta^T = 0.$$

Hence

$$y^TX - \beta^TX^TX = \lambda\beta^T.$$

Meaning

$$y^TX = \lambda\beta^T + \beta^TX^TX = \beta^T(X^TX + \lambda I).$$

By multiplying $(X^TX + \lambda I)^{-1}$ on both sides we get that

$$\beta^T = y^TX(X^TX + \lambda I)^{-1}.$$

Since $y^TX(X^TX + \lambda I)$ is self adjoint, so is $y^TX(X^TX + \lambda I)^{-1}$. Thus we finally get that

$$\hat{\beta}_{\text{Ridge}} = (X^TX + \lambda I)^{-1}X^Ty.$$

3. Expectation of the OLS Parameter

By letting $y = X\beta + \varepsilon$, we get that

$$\hat{\beta}_{OLS} = (X^TX)^{-1}X^T(X\beta + \varepsilon) = \beta + (X^TX)^{-1}X^T\varepsilon.$$

Since X and β are fixed, and $\mathbb{E}[\varepsilon] = 0$, we get that

$$\mathbb{E}[\hat{\beta}_{OLS}] = \mathbb{E}[\beta] + \mathbb{E}[\varepsilon]X^TX^{-1} = \beta.$$

4. Variance of the OLS Parameter

From the derivation of the expectation of the OLS parameter, we get that

$$\text{Var}(\hat{\beta}_{OLS}) = \text{Var}(\beta + (X^TX)^{-1}X^T\varepsilon) = \text{Var}(\beta) + \text{Var}((X^TX)^{-1}X^T\varepsilon).$$

Since X and β are fixed, we get that

$$\text{Var}(\hat{\beta}_{OLS}) = (X^TX)^{-1}\sigma^2[(X^TX)^{-1}X^T]^T = \sigma^2(X^TX)^{-1}X^TX(X^TX)^{-1} = \sigma^2(X^TX)^{-1}.$$

5. Expectation of the Ridge Parameter

We rewrite

$$\begin{aligned}\hat{\beta}_{Ridge} = (X^TX + \lambda I)^{-1}X^Ty &= (X^TX + \lambda I)^{-1}X^T(X\beta + \varepsilon) \\ &= (X^TX + \lambda I)^{-1}X^TX\beta + (X^TX + \lambda I)^{-1}X^T\varepsilon.\end{aligned}$$

Since X and β are fixed, we get

$$\mathbb{E}[\hat{\beta}_{Ridge}] = (X^TX + \lambda I)^{-1}X^TX\mathbb{E}[\beta] + (X^TX + \lambda I)^{-1}X^T\mathbb{E}[\varepsilon] = (X^TX + \lambda I)^{-1}X^TX\beta.$$

6. Variance of the Ridge Parameter

In a similar fashion as the derivation of the variance of the OLS parameter, we get that

$$\begin{aligned}\text{Var}(\hat{\beta}_{Ridge}) &= \text{Var}((X^TX + \lambda I)^{-1}X^TX\beta + (X^TX + \lambda I)^{-1}X^T\varepsilon) \\ &= \text{Var}((X^TX + \lambda I)^{-1}X^TX\beta) + \text{Var}((X^TX + \lambda I)^{-1}X^T\varepsilon).\end{aligned}$$

Where $\text{Var}((X^TX + \lambda I)^{-1}X^TX\beta) = 0$. Thus we find that

$$\begin{aligned}\text{Var}(\hat{\beta}_{Ridge}) &= \text{Var}((X^TX + \lambda I)^{-1}X^T\varepsilon) \\ &= (X^TX + \lambda I)^{-1}X^T\text{Var}(\varepsilon)[(X^TX + \lambda I)^{-1}X^T]^T \\ &= \sigma^2(X^TX + \lambda I)^{-1}X^TX[(X^TX + \lambda I)^{-1}]^T.\end{aligned}$$

7. Bias-Variance Decomposition

We assume that $y = f + \varepsilon$ where f is the true signal. We approximate $y \approx f$. We get that

$$\begin{aligned} \mathrm{MSE}(y, \tilde{y}) = \mathbb{E}[(y - \tilde{y})^2] &= \mathbb{E}[(f + \varepsilon - \tilde{y})^2] \\ &= \mathbb{E}[(f - \tilde{y} + \varepsilon)^2]. \end{aligned}$$

By expanding we get

$$\begin{aligned} \mathbb{E}[(y - \tilde{y})^2] &= \mathbb{E}[(f - \tilde{y})^2 + 2(f - \tilde{y})\varepsilon + \varepsilon^2] \\ &= \mathbb{E}[(f - \tilde{y})^2] + 2\mathbb{E}[(f - \tilde{y})\varepsilon] + \mathbb{E}[\varepsilon^2]. \end{aligned}$$

Since ε is independent from both f and $\tilde{y}$, we get that

$$2\mathbb{E}[(f - \tilde{y})\varepsilon] = 2\mathbb{E}[(f - \tilde{y})]\mathbb{E}[\varepsilon] = 0.$$

And since $\mathbb{E}[\varepsilon^2] = \sigma^2$ we get

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

We now want to compute the $\mathbb{E}[(f - \tilde{y})^2]$ term. In a similar fashion as the last expression, we expand and get

$$\begin{aligned} \mathbb{E}[(f - \tilde{y})^2] &= \mathbb{E}[[(f - \mathbb{E}[\tilde{y}]) - (\tilde{y} - \mathbb{E}[\tilde{y}])]^2] \\ &= \mathbb{E}[(f - \mathbb{E}[\tilde{y}])^2] - 2\mathbb{E}[(f - \mathbb{E}[\tilde{y}])(\tilde{y} - \mathbb{E}[\tilde{y}])] + \mathbb{E}[(\tilde{y} - \mathbb{E}[\tilde{y}])^2]. \end{aligned}$$

By swapping y for f we get that

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

By using that f is deterministic, and that $\mathbb{E}[\tilde{y}]$ is constant, we have that

$$2\mathbb{E}[(f - \mathbb{E}[\tilde{y}])(\tilde{y} - \mathbb{E}[\tilde{y}])] = 2(f - \mathbb{E}[\tilde{y}])\mathbb{E}[(\tilde{y} - \mathbb{E}[\tilde{y}])] = 2(f - \mathbb{E}[\tilde{y}])\left(\mathbb{E}[(\tilde{y}] - \mathbb{E}[\tilde{y}]\right) = 0.$$

Hence, in conclusion, we get that

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

8. Gradient of the OLS Cost Function

In the derivation of the OLS parameter, we found that

$$\frac{\partial \|y - X\beta\|^2}{\partial \beta} = -2(y - X\beta)^T X.$$

Thus, when swapping to a column vector and swapping the order of the difference, we find that the gradient of the OLS cost function is given by

$$\nabla_\beta C_{\mathrm{OLS}}(X, \beta) = \frac{2}{n}(X^T(X\beta - y)).$$

9. Gradient of the Ridge Cost Function

In the derivation of the Ridge parameter, we found that

$$\frac{\partial \|y - X\beta\|^2 + \lambda \|\beta\|^2}{\partial \beta} = -2(y - X\beta)^T X + 2\lambda \beta^T.$$

Again, we swap the expression from a row vector to a column vector, and swap the order of the difference to find that the gradient of the Ridge cost function is given by

$$\nabla_\beta C_{\text{Ridge}}(X, \beta) = 2(\frac{1}{n} X^T (X\beta - y) + \lambda \beta).$$