The Behaviour of Linear Regression With and Without Gradient Descent: a Case Study on the Runge Function

John-Magnus Johnsen
Simen L. Wærstad*
University of Oslo

(Dated: October 6, 2025)

We implement and investigate various solutions to the linear regression problem, namely, ordinary least squares (OLS), ridge, and lasso, as well as the different gradient descent methods, all of which are of central importance data analysis and machine learning. In particular, we will look at their behavior when used to model the Runge function. We find that gradient descent starting at $\mathbf{0}$ has a shrinking effect on the model parameters and that OLS with gradient decent can outperform the analytic solution. Furthermore, we see that the models are not equally stable under stochastic gradient descent.

I. INTRODUCTION

The linear regression problem shows up in a myriad of fields where we want to understand the underlying relationship between data. In particular, we are often interested in using recorded data to create a model so that we can predict the result of future observations.

When doing linear regression we aim to compute the function of a set of predetermined features that optimally predicts the target values. There exists many different types of models, but we will only consider ordinary least squares (OLS), ridge, and lasso.

Ridge and lasso regression are both variations of the more well known OLS regression model. They both have an added hyperparameter that shrinks the model parameters, although in different ways. This is particularly useful when dealing with features that are highly correlated [1, 2].

In the case of both OLS and ridge regression there exists explicit solutions to these problems, allowing us to directly derive the optimal model parameters for the given data. This is, however, not the case for lasso and many other methods[1]. While it is possible to solve lasso with traditional optimization methods [2], there are many other problems were this is infeasible or impossible. This is, for example, the case with neural networks. For these models we have to use numerical methods. One such class of methods is gradient descent methods.

Gradient descent methods treat the parameter space as a multidimensional surface where we, as the name suggests, use the gradient of this surface to choose parameters that locally minimizes (or nearly locally minimizes) the error of the model. These methods work iteratively by computing the gradient at the current position and moving a specified distance in a direction determined by the gradient. This gives us a numerical approach that does not require the model to have analytical solutions.

For large data sets computing the gradient can be computationally expensive. Stochastic gradient descent is a variation of gradient descent were random subset of the training data used to compute an estimate of the gradient. This decreases the computational cost of calculating the gradient at each step and is particularly useful if you have large data sets or a high dimensional parameter space [2, 3].

In practice, picking a good step size, or learning rate, as it is usually known, is a big problem [2, 3]. If the learning rate is too large our guess for the model parameters might diverge. If the learning rate is too small it will take an impractically long time to get a good approximation of the parameters. Various modifications to gradient descent exists that aims to address this. We will consider momentum, AdaGrad, RMSProp and Adam. Each variation makes modifications to how the position is updated, either to help reduce the impact of noise or to adaptively change the effective learning rate.

The Runge function function that is particularly ill behaved when it comes to polynomial fitting. We will use the Runge function as a case study to investigate the behavior of the aforementioned linear regression models and gradient descent methods.

We also discuss the relation between the bias, variance and the mean squared error of a model. This relationships is helpful in evaluating wether the model is over- or under-fitted.

II. METHODS

A. Linear Regression Models

1. Ordinary Least Squares

One of the simplest linear regression models is the ordinary least square (OLS) model. This model is derived by reframing the regression problem as a minimization problem on mean squared error.

Consider the case where we have feature matrix $\boldsymbol{X} \in R^{n\times m}$ and model parameters $\boldsymbol{\theta} \in \mathbb{R}^m$ and we want to approximate the function $y(x)$. Then we can measure

*`https://github.com/Waerstad/4155-Project-1`

the quality of fit using the mean squared error:

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

The function $C(\boldsymbol{\theta})$ is the cost function of our model. We want to find the choice of $\boldsymbol{\theta}$ that minimizes our cost function. This can be done by taking its derivative and setting it equal to zero and solving for $\boldsymbol{\theta}$.

The gradient of C is given by

$$\nabla C(\boldsymbol{\theta}) = \frac{2}{n}\left(\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{X}^T\boldsymbol{y}\right). \tag{2}$$

Setting the gradient equal to zero and solving for $\boldsymbol{\theta}$ then gives the optimal choice of $\boldsymbol{\theta}$. Thus, we have that the optimal parameters for OLS are given by

$$\boldsymbol{\theta}_{\text{OLS}} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{3}$$

The existence of this solution is dependent on the invertibility of $\boldsymbol{X}^T\boldsymbol{X}$. If the columns of $\boldsymbol{X}$ are not linearly dependent, then $\boldsymbol{X}^T\boldsymbol{X}$ is not invertible.

2. Ridge

Ridge regression is a generalization of OLS were the cost function that is minimized is modified. The Ridge regression cost function is given by

$$C(\boldsymbol{\theta}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2, \tag{4}$$

where $\lambda \in \mathbb{R}$ is positive. It is customary to drop the $1/n$ factor from the cost function [2]. The gradient of the cost function is given by

$$\nabla C(\boldsymbol{\theta}) = \frac{2}{n}\left(\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{X}^T\boldsymbol{y}\right) + 2\lambda\boldsymbol{I}_n. \tag{5}$$

Setting the gradient equal to zero and solving for θ then gives the optimal parameters

$$\boldsymbol{\theta}_{\text{Ridge}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}_{\boldsymbol{n}})^{-1}\boldsymbol{X}^T\boldsymbol{y}, \tag{6}$$

where $\boldsymbol{I}_n$ is the $n \times n$ identity matrix.

Minimizing the ridge cost function (equation 4) is equivalent to minimizing the OLS cost function (equation 1) with the added constraint that $\sum_i \theta_i^2 < t$ for some positive t [2].

Ridge has a few interesting properties. Notice that both the cost function and the model parameters reduce to the OLS case (equation 1 and equation 3) when $\lambda = 0$. Hence, OLS is a special case of Ridge. A useful property of Ridge is that, in the case where the inverse in the expression for the OLS parameters (equation 3) does not exits, addition of non-zero $\lambda\boldsymbol{I}_n$, ensures that the inverse in equation 6 does exist.

In addition, the λ term has a shrinking effect where the size of the model parameters are reduced towards zero. This reduction is dependent on the size of the square of the model parameters, thus larger model parameters are penalized more than smaller model parameters. This is particularly useful when the features of the model are highly correlated. High correlation can, in the case of OLS, lead to one coefficient being large and positive but having its effect more or less canceled out by another correlated feature with a large negative coefficient[1]. This can have the consequence that the resulting model generalizes poorly on new data. Penalizing the size of the parameters helps alleviate this problem.

As the penalization depends on the square of the model parameters, then this has the consequence that ridge is not independent of scaling, therefore it is normal to scale the data before fitting the model. For the same reasons we do not include the intercept in the feature matrix $\boldsymbol{X}$, as the intercept would be penalized differently depending on its size. Hence, the intercept is instead estimated by the mean of the target values $\boldsymbol{y}$ [1, 2].

3. Lasso

Lasso is another modification of OLS were we again add a term to the OLS cost function. The lasso cost function is given by

$$C(\boldsymbol{\theta}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1, \tag{7}$$

where $\lambda \in \mathbb{R}$ is positive. The gradient of the cost function is given by

$$\nabla C(\boldsymbol{\theta}) = \frac{2}{n}\left(\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{X}^T\boldsymbol{y}\right) + \lambda\,\text{sgn}(\boldsymbol{\theta}), \tag{8}$$

Where sgn is the entry-wise sign function.

Unlike with OLS and ridge, the optimal choice of $\boldsymbol{\theta}$ function does not have a closed form expression [1], although a solution can be computed using traditional optimization techniques.

Lasso is in many ways similar to ridge, and is also a generalization of OLS. Setting $\lambda = 0$ reduces the cost function to that of OLS (equation 1). Both methods shrink the model parameters, but they do it differently. Similarly to ridge, the lasso optimization problem can be reformulated to optimizing the OLS cost function with an added constraint on the size of the model parameters θ_i. In lasso the constraint is $\sum_i|\theta_i| < t$ for some real positive t [1, 2], unlike ridge, where the constraint is on the sum of squares. In contrast to ridge, if t is sufficiently small, then lasso can make some of the θ_i's exactly zero [1].

B. Gradient Descent

When there exists no exact solution for the model parameters or it is for some other reason impractical or infeasible to use we need another method to arrive at a sufficiently good choice of θ, that is a choice that gives a sufficiently low value of the cost function. One such class of methods is gradient descent, all of which are extensions of simple gradient descent.

1. Simple Gradient Descent

If we view the parameter space of possible $\boldsymbol{\theta}$ as multi-dimensional surface whose height is given by the value of the cost function $C(\boldsymbol{\theta})$. Then we want to find the value $\boldsymbol{\theta}$ that is at the deepest valley on the surface. One way of doing this is to pick a starting point on the surface, that is an initial value of $\boldsymbol{\theta}$, and then travel down the steepest hill some distance. This is then repeated until we end up at the bottom of the valley, which would be a minimum of the cost function.

More formally, we can represent this process as the algorithm shown in algorithm 1. Here, the learning rate ε is equivalent to the step size we take when moving along the gradient. When implementing this algorithm we also need to pick a stopping condition. While theoretically we want the algorithm to stop when the gradient is exactly zero, practically this is not a good stopping condition as it might take many lifetimes to reach zero if it ever does. In addition, numerical errors also pose a problem. Thus, it is common place to either chose a non-zero bound which the gradient must fall bellow, or stop after a set number of iterations, or use a combination of the two.

Algorithm 1 Simple gradient descent [2, 3]

```
Require: Initial model parameter choice $\boldsymbol{\theta}$
Require: Learning rate $\varepsilon$
  while stopping condition not met do
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \nabla C(\boldsymbol{\theta})$
  return $\boldsymbol{\theta}$
```

In the case of a general cost function there might be many local minima which have much higher cost values than the global minimum. Thus, it is a real risk that gradient descent might result in getting stuck in local minima that are not a global minimum, which would be detrimental to the performance of the model. In the case of OLS, ridge, and lasso, they all have convex cost functions (to see this inspect equation 1, 4, and 7 respectively). Consequently, they have a unique minimum which is therefore also the global minimum and therefore we do not need to be concerned about this issue in our case.

In practice, picking the right learning rate is a big issue. If the learning rate is too big we risk taking to large steps and potentially jumping over a minima. We can also risk having $\boldsymbol{\theta}$ diverge. If the learning rate is too small then we will never reach a minima in a reasonable time.

2. Stochastic Gradient Descent

For large data sets it can be very expensive to compute the gradient using all data points. Another option is to randomly select a subset of the data points and compute the gradient only using these points, giving an estimate of the gradient for the full data set. This can be achieved by modifying simple gradient descent (algorithm 1) by choosing a new random subset of the data before computing the gradient for each cycle of the while loop. Each of these subsets are called a mini-batch.

This does introduce noise in the gradient as the gradient depends on the feature matrix $\boldsymbol{X}$ and the target values $\boldsymbol{y}$. Consequently, stochastic gradient descent tends to "zig-zag" while descending down the slope [3]. Therefore, we reduce the computation cost of each iteration at the cost of introducing noise and potentially having to do more iterations in total.

3. Extensions of Gradient Descent

There are many extension of gradient descent. This text will only consider Momentum, AdaGrad, RMSProp, and Adam. All of these methods differ from simple gradient descent (algorithm 1) only in the way they update the values of the model parameters. Here we will only give a short description of each algorithm. For a fuller description of each algorithm see [3] or [2].

Momentum introduces a term that adds a decaying sum of the previous gradients to the update to the model parameters. As the same suggests, this is analogous to ball rolling down hill that builds up momentum so that any small bumps introduced does not have as big of an impact on its direction of movement. Thus, momentum helps alleviate the problems of noise as it also takes into account the gradients at the previous steps.

AdaGrad divides the learning rate ε by the square root of the sum of the squares of the previous gradients. If the sum of the squared gradients is small then the effective learning rate increases, and if the sum is large the effective learning rate decreases. Hence, AdaGrad adaptively changes the value of the learning rate depending on the sum of the previous gradients, addressing some of the difficulty with picking an appropriate learning rate.

RMSProp is a modification of Adagrad where the sum of squares is weighted so that the previous gradient is weighted higher than the sum of the old gradients. It does much of the same thing as AdaGrad but puts more emphasis on the most recent step.

Adam is similar to a combination of momentum and RMSProp, but includes a bias correction. Both the momentum term from momentum and the accumulated sum of the square of the gradients from RMSprop are initialized as zero in each algorithm respectively. Adam tries to correct for this by multiplying each of these two terms by a factor that is greater than one and decreases asymptotically towards one as the number of iterations that have been completed increase.

C. The Data

We used synthetic data generated by sampling the *Runge function* at linearly spaced points on the interval $[-1, 1]$. For the definition of the runge function see Appendix A. To this data normally distributed noise with mean *zero* and standard deviation of 0.1 and 0.5 were added, producing two different data sets. This choice of variance allowed us to evaluate the performance of the models both on data with low noise and a clear trend, and on more chaotic data characterized by high noise. A visualization of the data for the two cases can be seen in Figure 1.

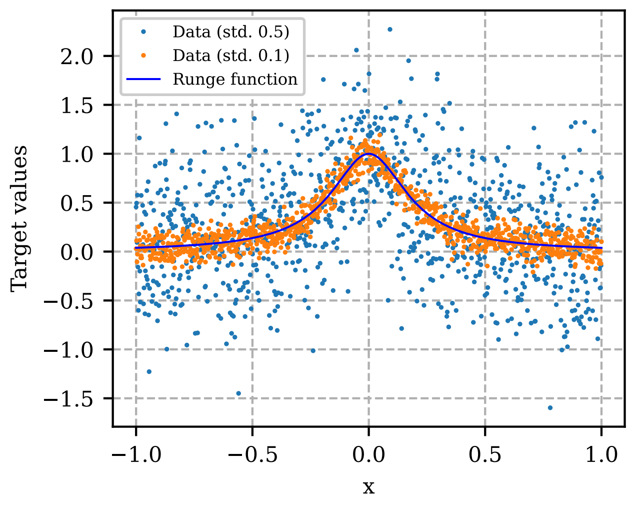

Figure 1: The Runge function and two typical iterations of the two datasets used, generated by adding normally distributed noise to the Runge function. Each data set with $n = 1000$ points and noise sampled from $N(0, 0.1)$ and $N(0, 0.5)$.

1. *Preprocessing of the Data*

Using the uniformly spaced x-values that were used to sample the Runge-function, we computed the polynomial features matrix (i.e. the Vandermonde matrix) for the appropriate degree. All the resulting feature matrices were preprocessed by centering each feature on zero and scaling it to unit variance, known as *standardization.* This is particularly important since, as discussed in section II A 2 and II A 3, ridge and lasso is not invariant to scaling because the penalty applied to the model parameters depend on their size. In practice, this was done using scikit-learn's [4] `StandardScaler` class. In addition, the target values were centered. Before training, the feature matrices and the target values were split into training and test sets using scikit-learn's `train_test_split()` function with a test size of 20 %. To prevent any information from the test set leaking into the model, the test were scaled on the same basis as the training set. In so keeping all aspects of the test data hidden from the model, thus preserving model integrity before prediction. The models were fitted using the training set, and any metrics such as the mean squared error and R^2 were computed by predicting on the test set.

2. *Simulation and Resampling Techniques*

To ensure that our analysis is viable and resting on a statistical sound foundation, one specific simulation approach and two resampling techniques were used. First, *Monte Carlo style* data generation to test the model sensitivity to different sample sizes, and also to look for convergence given certain model configurations and complexities. Then, to examine model limitations and behaviors, we employed the resampling techniques *bootstrapping* and *cross validation.*

The idea of the *Monte Carlo* approach is to have enough data generated so that both the models, and their predictions can be analyzed in the scope of the *law of large numbers.* By training the model and making predictions for a large number of training and test sets, then averaging over all of these runs, we would approach the true ability of the model. As unfavorable and favorable splits of the data, and the consecutive effects on the model and predictions, would diminish.

As real-world data collection usually is expensive and time consuming, having more data generated for verification of the results is simply not possible. With limited number of data points being the norm, resampling techniques are frequently used. For *bootstrapping* sampling with replacement from the original dataset is performed, so that one obtain a new set of data consisting of the same number of data points, but with a slightly different composition. Repeated sampling in this fashion gives an approximation to the distribution behind the data if it exists, and we can take the average over bootstrap samples to have a more general understanding. Bootstrap sampling also allows us to explore the bias-variance trade-off of the model, see appendix B.

Cross validation on the other hand, section the data into so called *folds* before using all but one fold to train the model and the last fold to test it. This is then repeated, using each fold as test fold. By taking the average of all predictions generated from each test fold we can say something about the performance of the model in general.

D. Implementation

1. Linear Regression and Gradient Descent

Each linear regression model was implemented from scratch by first creating a general linear regression class in Python. This class was then used as a framework to create new classes for each of the models OLS, ridge and lasso. Each of the model classes only contains their gradient and closed form solution, except for in the case of lasso, which does not have a closed form solution (see the discussion of lasso in section II A 3).

The analytical solutions for our OLS and ridge models were compared to those of scikit-learn. For 1000 randomly generated polynomials we the computed difference in MSE between the scikit-learn implementation and our implementation. This was done by randomly picking 100 x-values uniformly from $[-1, 1]$. Then, for a uniformly random degree between 1 and 15, coefficients were generated by randomly sampling uniformly from [-50, 50]. The target values were then computed using the generated polynomial and the appropriate feature matrix was computed using the generated x-values. The data was then split into a training and testing pair. The training data was used to compute the statistics used to standardize the training and test features. For each implementation the models were then fit on the training data and the resulting parameters were used to predict on the test features. For ridge, the hyperparameter was set to $\lambda = 0.01$ for all runs. The mean squared error was then computed using the predictions and the testing target values. We thhen calcualted the difference between the MSE for each model type by subtracting the MSE of our model from the MSE of scikit-learn's model. The mean, variance, and the maximum absolute value of the difference were computed. The results are shown in table I.

From table I we see that in both cases the mean is negative, indicating that our models perform worse and gets a higher MSE on average than the scikit-learn models. The variance is also low indicating a tight distribution of the MSE difference for both models. While our implementation does not give exactly the same results as scikit-learn, the maximum difference is small enough that it should be negligible in practice. We note that across the board our ridge implementation performs better than our OLS implementation. This could indicate that the matrix inverse in the model parameter expression for OLS (equation 3) is poorly conditioned and that adding the λ along the diagonal improves its conditioning leading to less numerical error. Hence, the difference could be in part due to the numerical stability of the specific implementation methods.

All the gradient descent methods were implemented as methods in the general linear regression class. In the case of the non-stochastic gradient descent methods the feature matrix $\boldsymbol{X}$ and $\boldsymbol{y}$ is the same for all iterations. For these methods, CPU-cycles were saved by pre-computing the products $\boldsymbol{X}^T\boldsymbol{X}$ and $\boldsymbol{X}^T\boldsymbol{y}$. The stopping condition was chosen to be the 2-norm of the gradient being smaller than a user given threshold, which we will call *precision*, or (logical or) reaching a user specified maximum number of iterations.

Table I: The statistics for the difference in MSE between scikit-learn's implementation and our implementation of OLS and ridge using the analytic expression for the model parameters. The statistics were calculated over 1000 runs using randomly generated polynomials on 100 uniformly distributed random points.

	OLS	Ridge
Mean	-2.230×10^{-11}	-1.065×10^{-12}
Variance	6.576×10^{-21}	5.223×10^{-22}
Max abs.	1.545×10^{-9}	6.362×10^{-10}

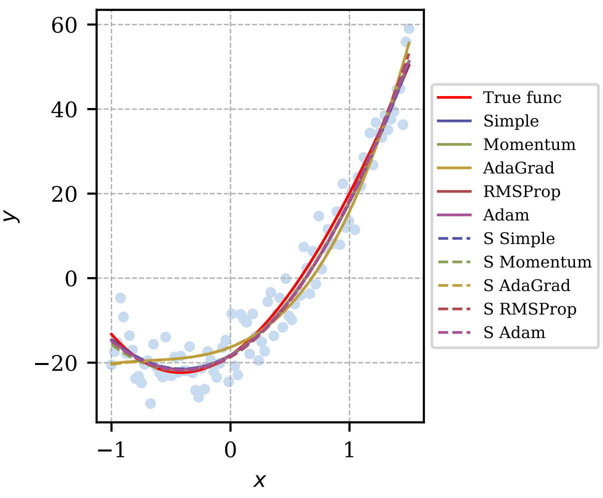

Figure 2: Lasso regression fits with various gradient descent methods for a randomly generated polynomial. The hyperparameter $\lambda = 0.01$ and the learning rate is 0.1. The "S" before method names indicates stochastic gradient descent with mini-batch size = 20. The plotted points is the data on which the models were fit.

To verify that our gradient descent implementations were working as intended we again generated random polynomials and plotted the fit for every combination of model type and gradient descent method. The resulting plot for lasso regression can be seen in figure 2. For plots of the OLS and ridge fits for the same polynomial see appendix C.

A separate mini-batch function was created to create mini-batches of a given size from the supplied data. The data points were picked uniformly with replacement from the supplied data using NumPys random module [5].

2. *Simulation and Resampling*

Performing the *Monte Carlo* simulation we used 500 runs to get stable results, storing values for each run, such as predicted test and train values. Then averaging over all runs.

For *bootstrapping* the data are first split into training and test sets of corresponding input variables and targets, these splits are kept unchanged through out the simulation. Then resampling of train variables and targes are performed for after each prediction is made.

For the *cross validation* algorithm we took inspiration from the code snippet found in **Chapter 5.5. Cross-validation** [2], and used scikit-learn's `RepeatedKFold` class. This class can repeat the run over folds, such that each repeat picks a different partition of the data used for the fold. We ended up only running one repeat, making it equivalent to the `kfold` class. For each set of folds a list of indecies are given so to split the data into a training set and test set. Then two feature matrix is initialized, one for train data, and one for test data These matrices are scaled in the previously described fashion to prevent data leaked, and finally the model is trained and used to predict on both the combined train folds and one test fold. For each test fold, we calculate and store MSE for both train and test prediction. Finally averaging over all repeats.

To have reproducible results, we used the seed functionality in *NumPy*, meaning all stochasticity in a section of code would yield the same results each time. This makes it possible to reproduce every part of the study, even dealing with random numbers as in the data synthesis. Any data or results shown here were generated with a specified seed ensuring that results should be exactly reproducible.

E. Use of AI tools

Microsoft Copilot was used as an aid in writing code. Primarily, it was used to outline the functionalities of different library functions as a supplement to the libraries' documentation. In addition, Microsoft Copilot was used to help debug code by interpreting error messages and give suggestions on possible fixes. An example query can be seen in our GitHub repository [6].

III. RESULTS AND DISCUSSION

A. Analytical solutions of various sample sizes

The initial setup involved having a loop run 500 times and for each iteration generate three samples of 100, 1000, and 10,000 data points with low noise (std. 0.1). Then we performed analytic OLS and Ridge regression using a λ_{Ridge} of 0.0001, and varying the model complexity with polynomials up to degree 15. We predicted on the test split for each polynomial degree and calculated the MSE, before finally averaging over all runs. The associated plot of calculated MSEs can be seen in figure 3. We also calculated the R^2 in the same way, shown in the figure 4.

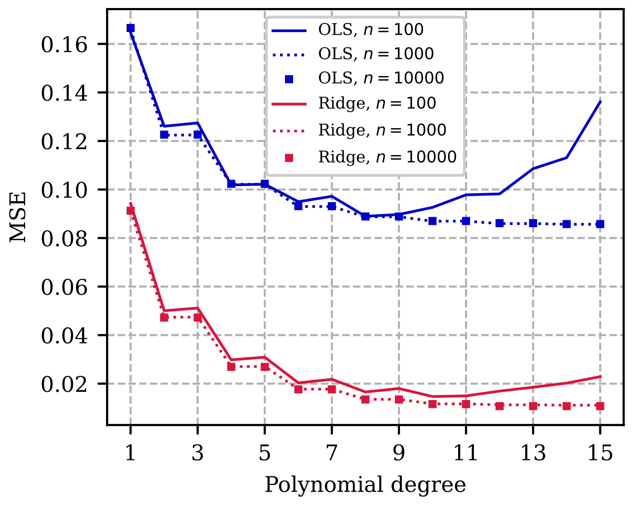

Figure 3: MSE of model prediction on test set for analytic OLS and analytic Ridge with increasing model complexity. Based on three different sample sizes, n. $\lambda_{\text{Ridge}} = 0.0001$.

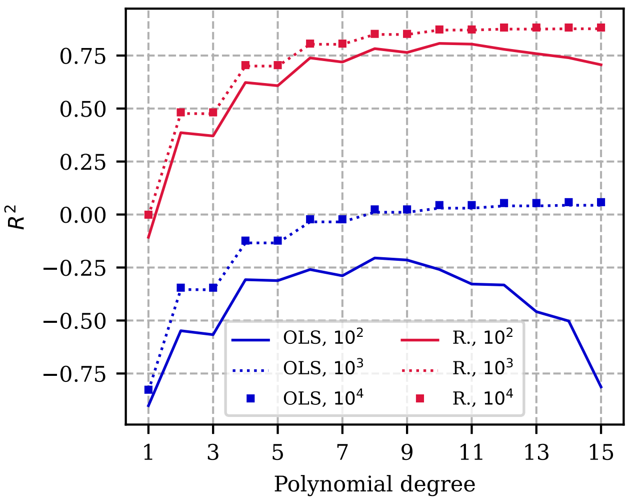

Figure 4: R^2 score of model prediction on test set for analytic OLS and analytic Ridge (R.) with increasing model complexity. Based on three different sample sizes with, $n = \{10^2, 10^3, 10^4\}$ data points. We used $\lambda_{\text{Ridge}} = 0.0001$.

We see that analytic OLS performs much worse than Ridge in this particular case. Looking at the MSE values for both OLS and Ridge there is a clear point where increased model complexity does not improve prediction accuracy considerably. Most notably the R^2 score of OLS reveal that the analytical solution does not explain much

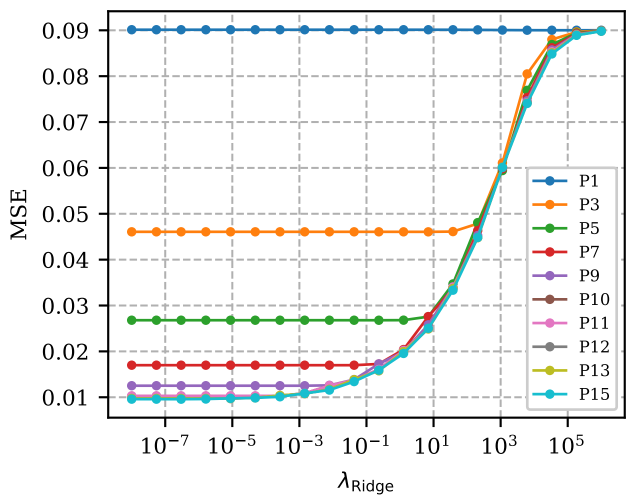

Figure 5: MSE of prediction on test set for a selection of polynomial degrees for analytic Ridge and 20 hyperparameters between 10^{-8} and 10^{6}. Dataset with 1000 points and noise with std. 0.1.

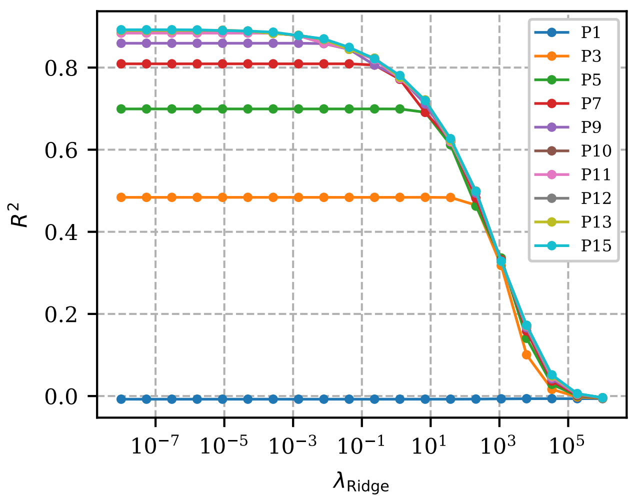

Figure 6: R^2 of prediction on test set for a selection of polynomial degrees for analytic Ridge and 20 hyper parameters between 10^{-8} and 10^{6}. Dataset with 1000 points and noise with std. 0.1.

of the variance in the data. The performance of ridge might be explained by its ability to reduce the the size of the model parameters, leading to a model with less extreme oscillations.

It is also interesting to see that both models show almost no difference in the prediction capabilities when being trained on 1000 or 10,000 data points. This means smaller sample sizes are sufficient to model the data and larger and larger data sets have diminishing returns. This is useful as it can save the time and effort required to generate or gather additional data.

Having seen the analytical solution to Ridge regression outperform OLS we further explored the effects of varying the hyperparameter λ_{Ridge} and its implication on the fit on test data. Similar to the setup above we ran 500 runs, regenerating a sample of 1000 data points for each run. Then for each polynomial degree, computed the prediction on test split with 20 evenly spaced λ_{Ridge} values between 10^{-8} to 10^{6}. A selection of the following MSEs, as averaged over the 500 runs, are plotted against the corresponding lambda value in figure 5. The R^2 scores of the same simulation are to be found in figure 6.

We see that higher model complexity lowers the MSE, and that the optimal value of the hyperparameter lie around 10^{5} in our case. The R^2 score also shows that these particular configurations yield a good model fit to the data, as the score approaches a value of 1.

In the case of very noisy data (std. 0.5) we see a more diverse picture where the polynomial degrees of highest order not necessarily give the best fit, and a small adjustment in hyperparameter might see a higher or lower model complexity perform better. A plot showing the data for this very noise case can be seen in figure 21 in appendix E.

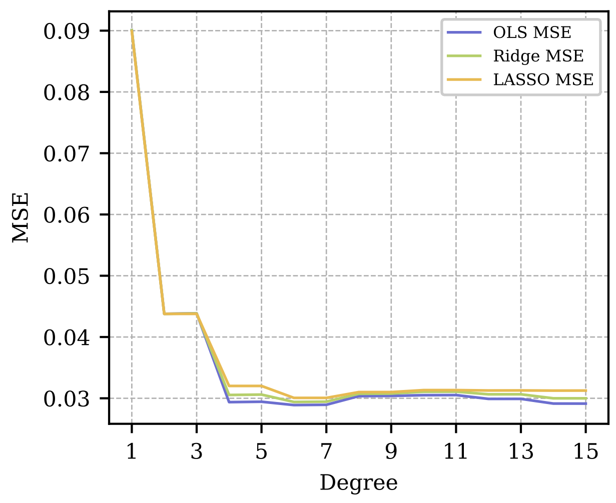

Figure 7: The MSE of OLS, ridge, and lasso ($\lambda = 0.01$), as a function of polynomial degree, computed using $n = 1000$ samples of the Runge function with noise sampled from $N(0, 0.1)$.

B. Performance of Gradient Descent Methods

Repeating the computations done with the analytical solutions, for simple gradient descent we computed and plotted the mean squared error and the R^2 score for OLS, ridge, and lasso (with $\lambda = 0.01$) as a function of model complexity. In our case of using only polynomial features, this is equivalent to polynomial degree. Using $n = 1000$ points with noise sampled from $N(0, 1)$ simple gradient descent was ran with learning rate 0.001 and initial parameter guess $\boldsymbol{\theta} = \mathbf{0}$.The resulting plot of MSE as a function of complexity can be seen in figure 7 and the R^2 score as a function of complexity can be seen in figure 8.

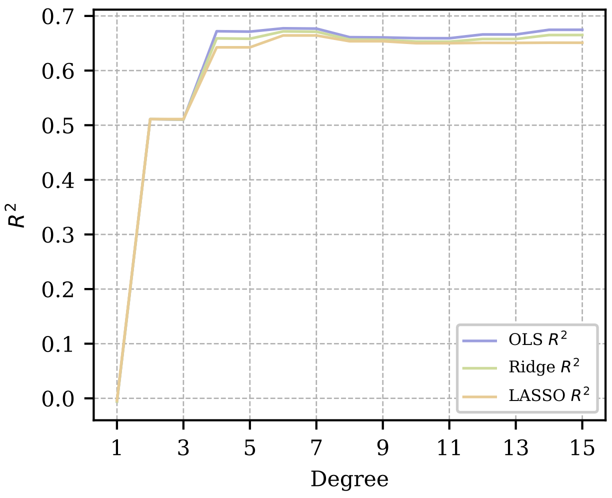

Figure 8: The R^2 score of OLS, ridge, and lasso ($\lambda = 0.01$), as a function of polynomial degree, computed using $n = 1000$ samples of the Runge function with noise sampled from $N(0, 0.1)$.

Figure 7 and figure 8 clearly shows that after degree 4 there seems to be no clear trend in which degree performs better or worse in approximating the Runge function. Both the R^2 score and the MSE seams to stabilize, having almost a linear relation ship with model complexity, except for some pumps, e.g. at degree 6 and 14. After degree 4, once the MSE and the R^2 score begins to stabilize we see that the OLS outperforms Ridge, and Ridge outperform lasso.

Furthermore, the results in figure 7 and 8 are in contrast to the analytic solutions in figure 3 and 4 where OLS performs worse than ridge, getting both lower MSE and higher R^2 scores. We hypothesize that this is due to gradient descent overall performing worse than the analytical version when computing the MSE with respect to the training targets. As the model parameters are approached from the initial guess $\boldsymbol{\theta} = \mathbf{0}$, we expect the parameters to be smaller than the analytical solution. Thus this poorer fit could produce a flatter curve that avoids the oscillations towards the edge of the data set. Therefore, in a way preventing overfitting. The plot of the fitted analytic and simple gradient descent models shown in figure 9 shows that this might indeed be the case.

We also studied MSE as a function of the number of iterations. For each model type the gradient descent methods were ran on a set of 100 samples of Runge Function linearly spaced along the x axis with noise generated by a normal distribution with mean zero and variance 0.1. For each method the precision was set to 0 so that the methods would stop only if it reached the maximum allowed number of iterations, which was set to 10,000. The learning rate was set to 0.001 for all methods and $\boldsymbol{\theta} = \mathbf{0}$ was

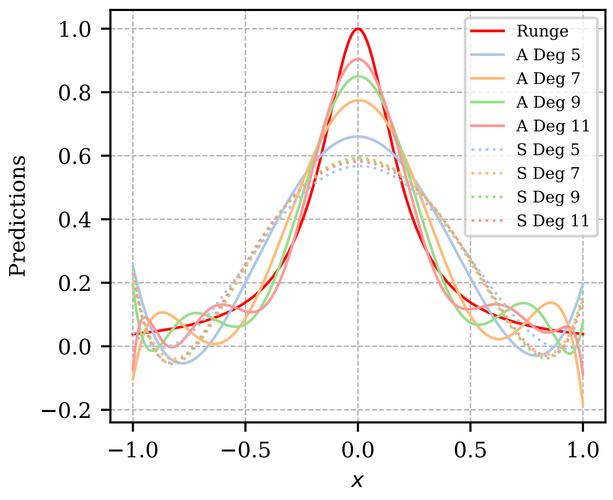

Figure 9: Models fitted to the Runge function with analytic (A) and simple gradient descent (S) OLS. The learning rate was 0.001 and the gradient descent was ran for 10,000 iterations. To fit the models $n = 1000$ points with noise sampled from $N(0, 0.1)$ were used.

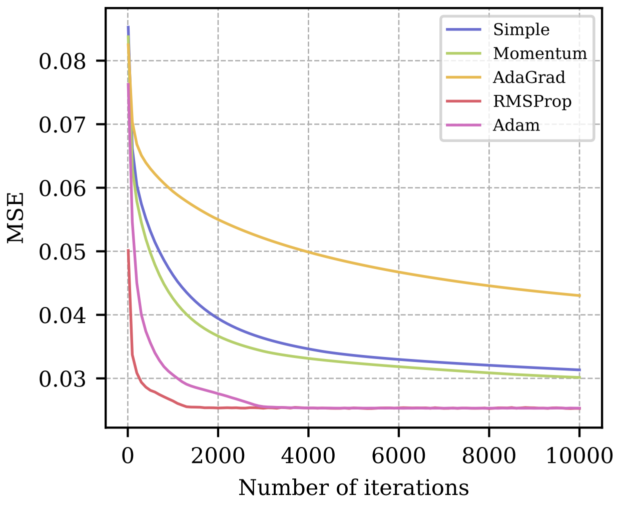

Figure 10: The mean squared error of non-stochastic gradient descent methods applied to lasso regression ($\lambda = 0.01$) as the number of iterations increase The learning rate is 0.001

.

used as the initial parameter guess. The resulting data for the case of lasso with non-stochastic gradient descent is shown in figure 10. For the non-stochastic results for OLS and ridge refer to figure 16 and 17 in appendix D.

For the stochastic variants a mini-batch size of 20 was used at first. All other hyperparameters of the gradient descent methods were set to their implementation defaults, which are specified in the documentation found on our GitHub repository [6]. For ridge and lasso we set the hyperparameter $\lambda = 0.01$.

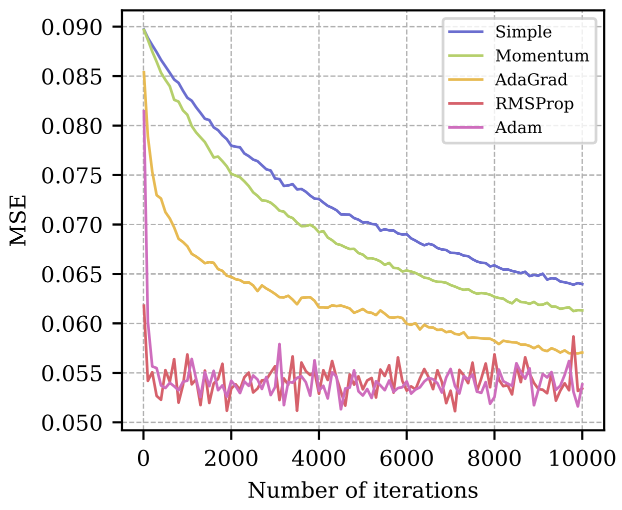

Figure 11: The mean squared error of stochastic gradient descent methods applied to lasso regression ($\lambda = 0.01$) as the number of iterations increase. The learning rate is 0.001 and the mini-batch size is 40.

The choice of mini-batch size 20 lead to a very unstable gradient descent for lasso regression where all methods performed approximately equally bad in the long run. This can be seen in figure 20 in appendix D. This is likely due to the choice of the initial parameter guess $\boldsymbol{\theta} = \mathbf{0}$. Near zero the gradient of lasso's cost function becomes very unstable due to the discontinuity of the $\lambda\,\mathrm{sgn}(\boldsymbol{\theta})$ term in its gradient (equation 8). The term quickly switches from having value $-\lambda$, 0, and λ as it varies around 0. Thus any noise will have a larger impact than on OLS and ridge, which have continuous gradients. The performance of stochastic gradient descent for OLS and ridge with mini-batch size 20 can be seen in figure 18 and 19 in appendix D. Therefore, in the case of lasso we considered a mini-batch size of 40, which reduced the noise, increased performance, and produced results closer to the theoretical expectations. The data for the stochastic lasso with mini-batch size 40 is shown in figure 11.

Comparing the non-stochastic results in figure 10 and the stochastic results in figure 11 we clearly see that using stochastic gradient descent introduces additional noise into the models, indicated by the peaks and valleys in the graphs. We also see that estimating the gradient with a mini-batch also worsens the performance of every model compared to the non-stochastic version, which is in agreement with the theory.

Despite AdaGrad performing significantly worse than simple gradient descent and momentum in the non-stochastic case, it performs significantly better than them in the stochastic case. The underperformance of AdaGrad in the non-stochastic case might be due to the small learning rate chose, namely 0.001, which leads AdaGrad to take too small steps as it adaptively decreases the learning rate as the number of iterations increase. However, as Adagrad adapts its learning rate depending on all previous gradients this seems to reduce the effect of noise leading it to outperform simple gradient descent and momentum in the stochastic case.

In both the stochastic and non-stochastic cases we see that Adam and RMSProp outperforms the other methods. This is expected as they both utilize more sophisticated methods to adaptively change the learning rate based on the previous methods. It does however, seem to introduce more noise as the methods weigh the more recent gradient estimates higher. The additional noise when compared to the other methods could also be partly due to the fact that the methods are closer to the minimum and that this has some interaction effect with the adaptive learning rate.

C. Resampling: bootstrap and cross-validation

Having been reliant on the *Monte Carlo* method we finalize by looking at two different methods of resampling. For simplicity we used analytic OLS even tho we have seen its shortcomings on this dataset. The *bootstrap* method was performed using 1000 data points (generated with std. 0.1) and 1000 bootstrap for each polynomial degree ($P \in [1, 30]$), storing the prediction. At the end of each bootstrap cycle, bias, variance and MSE where calculated for the specific polynomial degree. For a visualization of bia, variance, MSE and model complexity, see figure 12.

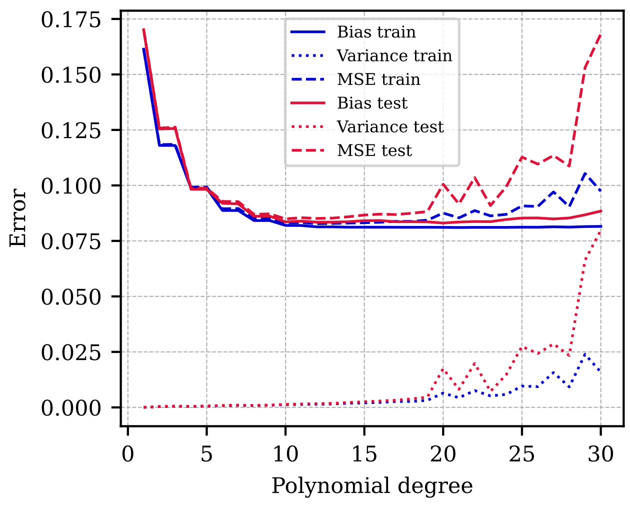

Figure 12: The bias-variance tradeoff visualized for OLS. All metrics calculated both for model prediction on test and training data, for a variety of fitting polynomials. Here for a dataset of 1000 data points with noise (std. 0.1), resampled 1000 times through bootstrapping.

As expected the prediction performed better on the data used to train the model, than for the unseen test data, even tho the difference is small. We see a clear point where increased model complexity does not benefit the bias noteworthy, but instead comes at the cost of an increased variance and MSE. This illustrates the the bias-variance-tradeoff, where increasing model complexity eventually will have the model overfit to the training data.

For the *cross validation* resampling technique we employ all the models we have looked at, OLS, Ridge and Lasso. We use analytic OLS and Ridge, and Lasso with the Adam optimizer for our models, starting out with a 1000 data points of low noise (std. 0.5) using 10 folds. We calculated the MSEs for the test prediction for each of the 10 test folds, averaging over the folds in the end. The resulting MSEs coresponding to model complexity can be viewed in figure 13.

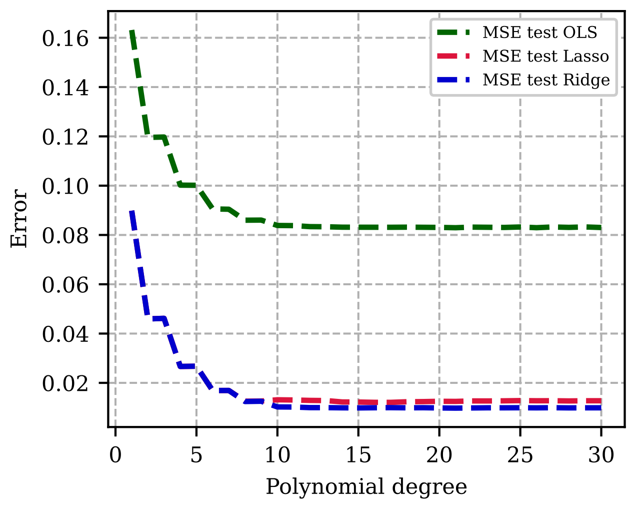

Figure 13: MSEs from cross validation with, $k = 10$. Hyper parameter Ridge and Lasso $\lambda = 0.0001$. $n = 1000$, data points (noise std. 0.1.). Lasso with the Adam optimizer.

We see that prediction MSEs are on par with what we got after a 500 run *Monte Carlo* simulation previously, and that by only using 10 folds. This is quite a significant improvement. Also the MSE stays stable even for high model complexities which might be the result of a good fit, but this should be investigated further. We would normaly expect to see some signs of overfitting in the region of such high complexity.

IV. CONCLUSION

When using analytic solutions we see that OLS gets outperformed by Ridge across the board. However, when using simple gradient descent the result is flipped, with OLS performing slightly better than both ridge and lasso, although the difference is small. In fact, OLS with gradient descent outperforms the analytic solution. Interestingly, it seems that gradient descent with initial guess zero, has a shrinking effect on the model parameters, similar to ridge and lasso.

We also find that regenerating tons of data to train the models not necessarily improve model performance, and that different forms of resampling techniques can effectively make use of a rather small sample size.

In our limited tests of the different gradient descent methods we see that Adam and RMSProp produce the best results in the least number of iterations. We also see that the relative performance of the gradient descent algorithms are the same across stochastic and non-stochastic gradient descent, with the exception of AdaGrad. The bad performance of AdaGrad in the non-stochastic case is likely due to having picked too small of a learning rate. Hence, it would be beneficial to rerun the analysis for various learning rates.

We also see that with initial model parameter guess $\mathbf{0}$, gradient descent with lasso is more unstable than OLS and ridge. In particular, the performance of all gradient descent methods on lasso became worse than any method applied to OLS or ridge at the same mini-batch size of 20.

For future work we suggest looking more deeply into the effect of the learning rate on the performance gradient descent methods, as well as the effect of varying the batch size. It would also be interesting to look more in depth at the effect varying ridge hyperparameter has on the model parameters and compare this to the effect the change in the hyperparameter would have on lasso.

[1] 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`.

[2] M. Hjorth-Jensen, *Applied Data Analysis and Machine Learning* (Department of Physics, 2025), URL `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`.

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

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Journal of Machine Learning Research **12**, 2825 (2011), URL `https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html`.

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al., Nature **585**, 357 (2020), URL `https://doi.org/10.1038/s41586-020-2649-2`.

[6] J.-M. Johnsen and S. L. Wærstad, *4155-project-1*, `https://github.com/Waerstad/4155-Project-1` (2025).

Appendix A: The Runge function

$$f(x) = \frac{1}{1+25x^2} \tag{A1}$$

Appendix B: Derivation of Bias-Variance Trade-Off

We want to show that

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

where

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

and

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

We have that

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

As $\boldsymbol{\varepsilon}$ is independent of $\boldsymbol{f}-\tilde{\boldsymbol{y}}$, then we can pull it out as the factor $\mathbb{E}\left[\varepsilon\right]$ which equals zero. Consequently, the middle term vanishes. Furthermore, the last term is exactly σ^2. Thus, we have

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

. If we now consider only the first term we have

$$\begin{aligned}\mathbb{E}\left[(\boldsymbol{f}-\tilde{\boldsymbol{y}})^2\right] &= \mathbb{E}\left[((\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}]) + (\mathbb{E}[\tilde{\boldsymbol{y}}]-\tilde{\boldsymbol{y}}))^2\right] \\ &= \mathbb{E}\left[(\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}])^2\right] + 2\mathbb{E}\left[(\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}])(\mathbb{E}[\tilde{\boldsymbol{y}}]-\tilde{\boldsymbol{y}})\right] + \mathbb{E}\left[(\mathbb{E}[\tilde{\boldsymbol{y}}]-\tilde{\boldsymbol{y}})^2\right] \\ &= \mathrm{Bias}[\tilde{\boldsymbol{y}}] + 2\mathbb{E}\left[(\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}])(\mathbb{E}[\tilde{\boldsymbol{y}}]-\tilde{\boldsymbol{y}})\right] + \mathrm{Var}[\tilde{\boldsymbol{y}}].\end{aligned}$$

Lastly, we expand the second term. This yields

$$\begin{aligned}&\mathbb{E}\left[\boldsymbol{f}\mathbb{E}[\tilde{\boldsymbol{y}}]\right] - \mathbb{E}\left[\boldsymbol{f}\tilde{\boldsymbol{y}}\right] - \mathbb{E}\left[\mathbb{E}[\tilde{\boldsymbol{y}}]^2\right] + \mathbb{E}\left[\mathbb{E}[\tilde{\boldsymbol{y}}]\tilde{\boldsymbol{y}}\right] \\ &= \boldsymbol{f}\mathbb{E}[\tilde{\boldsymbol{y}}] - f\mathbb{E}[\tilde{\boldsymbol{y}}] - \mathbb{E}[\tilde{\boldsymbol{y}}]^2 + \mathbb{E}[\tilde{\boldsymbol{y}}]^2 \\ &= 0.\end{aligned}$$

It follows that

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

Appendix C: Test Fits for Gradient Descent

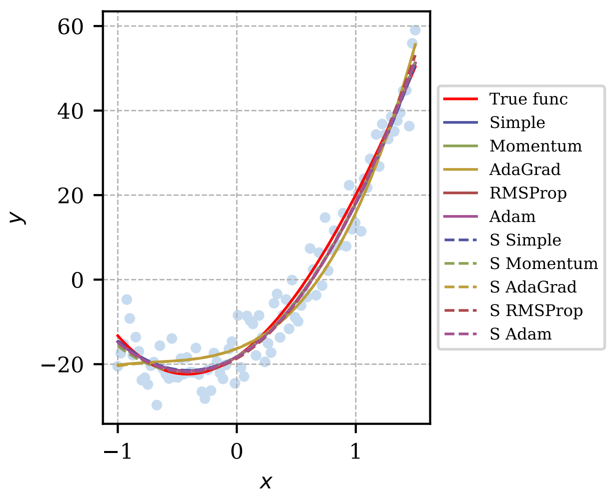

Figure 14: OLS regression fits with various gradient descent methods for a randomly generated polynomial. The learning rate is 0.1. The "S" before method names indicates stochastic gradient descent with mini-batch size = 20

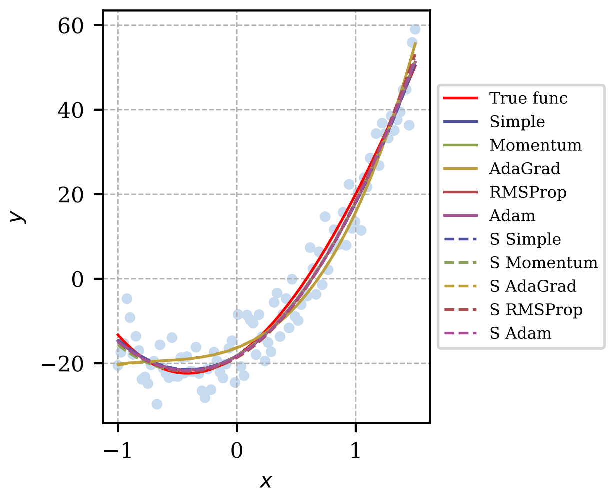

Figure 15: Ridge regression fits with various gradient descent methods for a randomly generated polynomial. The hyperparameter $\lambda = 0.01$ and the learning rate is 0.1. The "S" before method names indicates stochastic gradient descent with mini-batch size = 20

Appendix D: Performance of Gradient Descent Methods

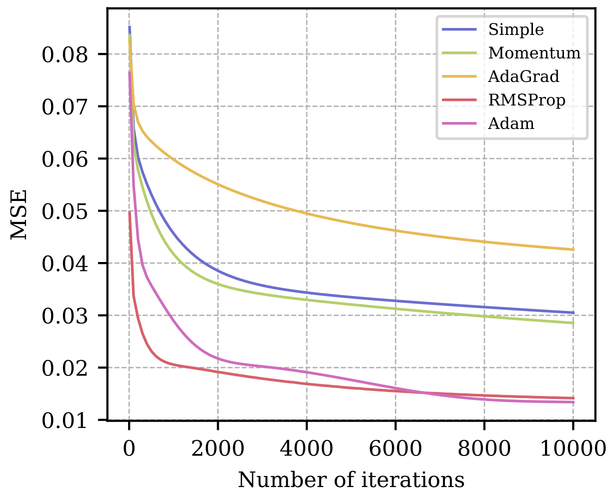

Figure 16: The mean squared error of non-stochastic gradient descent methods applied to OLS regression as the number of iterations increase The learning rate is 0.001

.

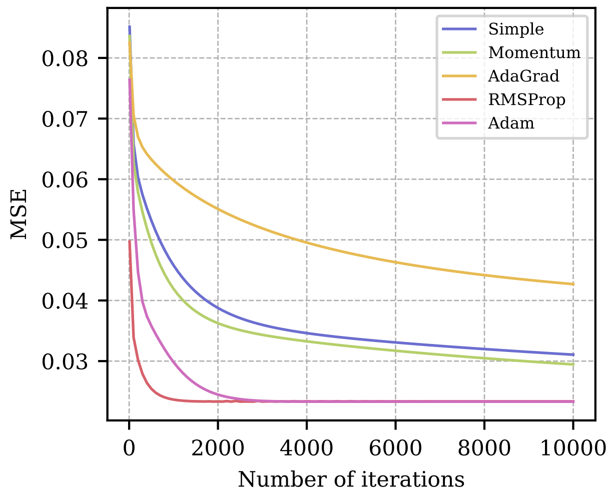

Figure 17: The mean squared error of non-stochastic gradient descent methods applied to ridge regression ($\lambda = 0.01$) as the number of iterations increase The learning rate is 0.001

.

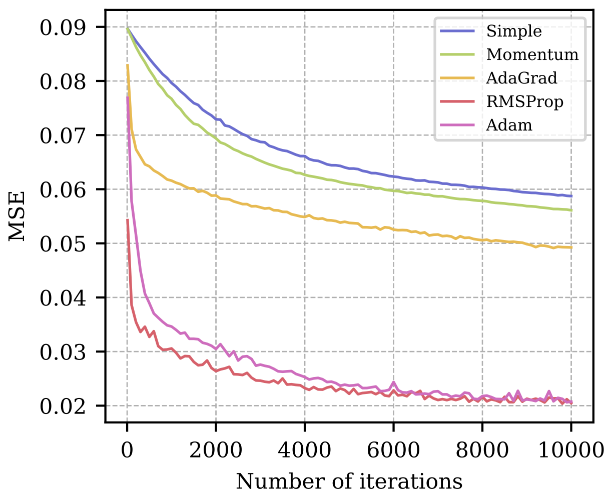

Figure 18: The mean squared error of stochastic gradient descent methods applied to OLS regression as the number of iterations increase. The learning rate is 0.001 and the mini-batch size is 20.

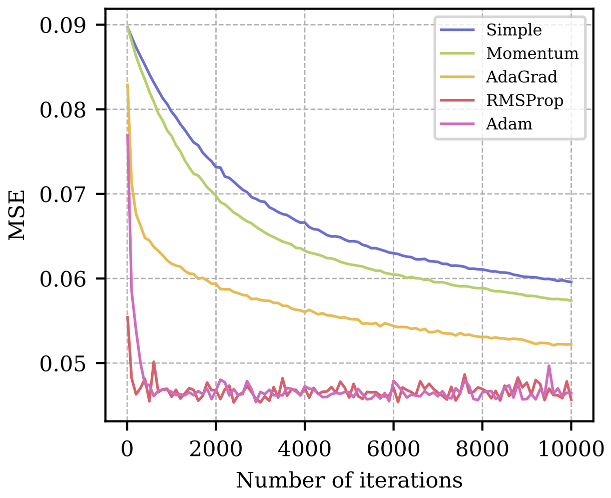

Figure 19: The mean squared error of stochastic gradient descent methods applied to ridge regression ($\lambda = 0.01$) as the number of iterations increase. The learning rate is 0.001 and the mini-batch size is 20.

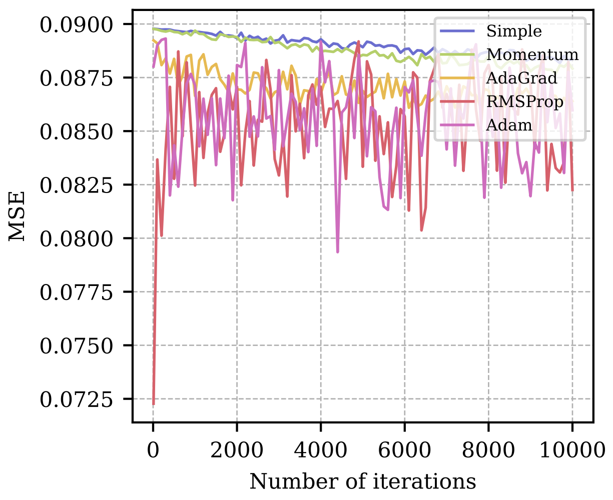

Figure 20: The mean squared error of stochastic gradient descent methods applied to lasso regression ($\lambda = 0.01$) as the number of iterations increase. The learning rate is 0.001 and the mini-batch size is 20.

Appendix E: The ridge hyperparameter and MSE

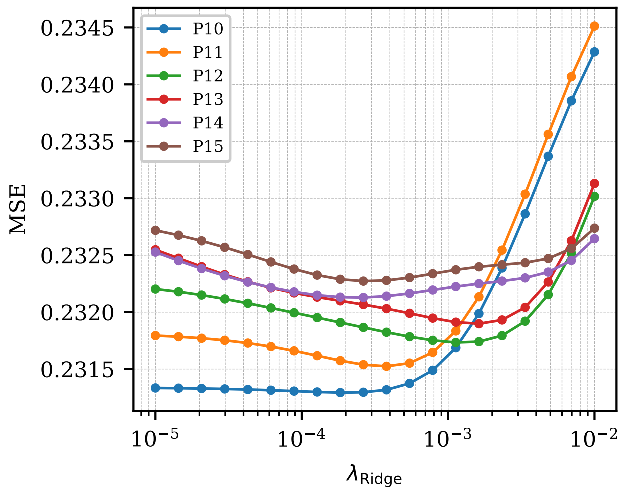

Figure 21: MSE of prediction on test set for a selection of polynomial degrees for analytic Ridge and 20 hyperparameters between 10^{-5} and 10^{-2}. Dataset with 1000 points and noise with std. 0.5. The optimal $\lambda_{\text{Ridge}} = 2.63 \times 10^{-4}$.