A Comparison of Linear Regression Techniques as Applied to Runge's Function

Kim Graatrud, Simon Silverstein, and Nicholas Andrés Tran Rodriguez

Universitetet i Oslo

Fitting a model to data is a fundamental practice in all of science. We compare two analytic and seven gradient descent (GD) based methods of doing so, as applied to the linear regression of a polynomial to the one-dimensional Runge's function. Methods are compared on convergence rates, numerical stability, and overall accuracy by MSE. We find that flexible-learning-rate methods of GD converge fastest, and that polynomial degrees from 3 to 7 fit best while avoiding erratic tail behavior. We examined the behavior of the models OLS, Ridge, and Lasso at high model complexities and observed that the approximated MSE for Ridge and Lasso converged. Comparing bootstrapping and k-fold CV with each-other we found that k-fold CV consistently give a lower approximated MSE. We used bootstrap to observe the inverse relationship between MSE, bias, and variance as a function of model complexity and data size.

CONTENTS

I. INTRODUCTION

As a discipline, science is concerned with predicting the world around us. Predictions are made from *models* and compared to some aspects of the world (*data*). Choosing a model is therefore a fundamental part of almost all science; understandably, there are many ways of going about it.

We here focus on methods of linear regression, that is we assume a model of the form

$$\mathbf{X}\boldsymbol{\theta} = \tilde{\boldsymbol{y}}, \tag{1}$$

where $\mathbf{X}$ is a matrix of input features corresponding to some predicted output $\tilde{\boldsymbol{y}}$, and $\boldsymbol{\theta}$ is a set of parameters which our job is to find optimal values of.[1]

We limit our scope to models with polynomial features, so that for some input x_i, the corresponding output y_i is predicted by an N-th degree polynomial evaluated at x_i:

$$\tilde{y}_i = \theta_0 + \theta_1 x_i^1 + \theta_2 x_i^2 + \cdots + \theta_N x_i^N. \tag{2}$$

Our feature matrix is then an $n \times N$ matrix like:

$$\mathbf{X} = \begin{bmatrix} x_0^0 & x_0^1 & \dots & x_0^N \\ x_1^0 & x_1^1 & \dots & x_1^N \\ \vdots & \vdots & \ddots & \vdots \\ x_n^0 & x_n^1 & \dots & x_n^N \end{bmatrix}, \tag{3}$$

given n input data. One measure of how well a set of parameters $\boldsymbol{\theta}$ perform is the Mean Squared Error (MSE). Assuming we are trying to reproduce some true data $\boldsymbol{y} = [y_1, y_2, ..., y_n]$, our model gives us a prediction $\mathbf{X}\boldsymbol{\theta} = \tilde{\boldsymbol{y}} = [\tilde{y}_1, \tilde{y}_2, ..., \tilde{y}_n]$. The MSE is defined as

$$\text{MSE}(\boldsymbol{y}, \tilde{\boldsymbol{y}}) \equiv \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2. \tag{4}$$

It is at times useful to have a measure of performance that is independent of scale and falls between 0 and 1. For this we use the typical coefficient of determination:

$$R^2 \equiv 1 - \sum_{i=0}^{n-1}\frac{(y_i - \tilde{y}_i)^2}{(y_i - \bar{y})^2}, \tag{5}$$

where $\bar{y}$ is arithmetic mean $\bar{y} = (1/n)\sum_{i=0}^{n-1} y_i$.

[1] We use the convention that uppercase bold variables (ex. $\mathbf{X}$) represent matrices, lowercase bold (ex. $\boldsymbol{y}$) represent vectors, and italic (ex. z) represent scalars.

We will compare methods of optimization by modeling the *Runge function*. The Runge function f is given by

$$f(x) = \frac{1}{1+25x^2}. \tag{6}$$

To mimic real data, we add gaussian noise of mean $\mu = 0$ and variance $\sigma^2 = 0.05^2$. Our true data is then given by:

$$y_i = \frac{1}{1+25x_i^2} + N(\mu = 0, \sigma = 0.05). \tag{7}$$

With that, we are ready to begin optimizing $\boldsymbol{\theta}$.

The code used to produce the results for this paper can be found at our *GitHub* repository[2].

II. METHODS

A. Linear regression methods

It is useful to re-frame the problem of optimization as one of minimizing some *cost function*. What cost function we use is determined by what we mean by 'optimal,' or having 'best fit.' One of the simplest method of optimization is Ordinary Least Squares (OLS), which finds optimal parameters by minimizing the MSE. In matrix-vector notation the MSE is given by

$$\text{MSE} = \frac{1}{n}(\boldsymbol{y} - \mathbf{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\theta}), \tag{8}$$

and the optimal parameters $\hat{\boldsymbol{\theta}}$ are

$$\hat{\boldsymbol{\theta}}_{\text{OLS}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{y} \tag{9}$$

(Hastie *et al.*, 2009). Note that this method assumes $\mathbf{X}^T\mathbf{X}$ is invertible.

Consider now the following cost function:

$$C_{\boldsymbol{\theta}} = \frac{1}{n}(\boldsymbol{y} - \mathbf{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\theta}) + \lambda||\boldsymbol{\theta}||_2^2 \tag{10}$$

with a penalty term λ, and with the usual 2-norm $||\boldsymbol{x}||_2 = \sqrt{\sum_i x_i^2}$. Notice that this cost function is similar to (8), only with an added penalty. Linear regression with this cost function is known as *Ridge* regression, and we get an analytical solution using a similar technique as for (9):

$$\hat{\boldsymbol{\theta}}_{\text{Ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_{n\times n})^{-1}\mathbf{X}^T\boldsymbol{y}, \tag{11}$$

where $\mathbf{I}_{n\times n}$ is the $n \times n$ matrix identity. Ridge regression suppresses large parameters, which can be useful when parameters are highly degenerate and has the added benefit that for $\lambda > 0$, $(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_{n\times n})$ is invertible. For $\lambda = 0$, $\hat{\boldsymbol{\theta}}_{\text{Ridge}}$ converges to $\hat{\boldsymbol{\theta}}_{\text{OLS}}$.

[2] https://github.com/KimGraatrud/FYS-STK4155-Projects/tree/main/Project_1

Not all cost functions have an analytical solution. As demonstration, consider the cost function for *Lasso* regression

$$C_{\boldsymbol{\theta}} = \frac{1}{n}(\boldsymbol{y} - \mathbf{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\theta}) + \lambda||\boldsymbol{\theta}||_1, \tag{12}$$

which is similar to (10) but with the ℓ^1-norm ($||\boldsymbol{x}||_1 = \sum_i |x_i|$) instead of the ℓ^2. Taking the derivative as before will introduce a $\lambda\text{sign}(\boldsymbol{\theta})$ term, which cannot be solved for $\boldsymbol{\theta}$. Thus, no analytical solution exists, and we must turn to numerical methods of minimization. We'll focus on gradient-descent based methods for both their generalizability and their prevalence in the field of machine learning.

B. Gradient Descent

We start with a cost function $C_{\boldsymbol{\theta}}$ and define the gradient with respect to $\boldsymbol{\theta}$ in the usual way:

$$\nabla C_{\boldsymbol{\theta}} = \begin{pmatrix} \frac{\partial C_{\boldsymbol{\theta}}}{\partial \theta_0} \\ \frac{\partial C_{\boldsymbol{\theta}}}{\partial \theta_1} \\ \vdots \\ \frac{\partial C_{\boldsymbol{\theta}}}{\partial \theta_N} \end{pmatrix}.$$

The gradient descent technique is laid out in Algorithm 1. By computing the gradient for some initial guess of the parameters, we learn the direction of greatest cost increase in parameter space. We then move our guess in the opposite direction proportional to the gradient by some learning rate η. We repeat until we've found a minimum.

Algorithm 1 Gradient Descent

```
1: Let $\eta$ be the learning rate, typically $\in [10^{-5}, 10^{-1}]$.
2: Initialize $\boldsymbol{\theta}$ with an initial guess.
3: Initialize $\Delta\boldsymbol{\theta} = 0$.
4: while not converged do
5:     Compute the gradient: $\boldsymbol{g} \leftarrow \nabla C_{\boldsymbol{\theta}}$.
6:     Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\eta\boldsymbol{g}$.
7:     Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
8: end while
```

This method works for all cost functions we have introduced. For example, if we choose to use the MSE as the cost function, the gradient is obtained by differentiating (8):

$$\nabla C_{\boldsymbol{\theta}} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \boldsymbol{y}). \tag{13}$$

For consistency, we will refer to gradient descent with (13) as 'OLS gradient descent.' We can do the same with cost function (10):

$$\nabla C_{\boldsymbol{\theta}} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \boldsymbol{y}) + \lambda\boldsymbol{\theta}, \tag{14}$$

which we similarly refer to as 'Ridge gradient descent,' and with cost function (12):

$$\nabla C_{\boldsymbol{\theta}} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \boldsymbol{y}) + \lambda \text{sign}(\boldsymbol{\theta}), \tag{15}$$

'Lasso gradient descent.'

Variations of Algorithm 1 exist to address cases where it performs poorly. For one, it is possible that $C_{\boldsymbol{\theta}}$ has multiple local minima, in which $\nabla C_{\boldsymbol{\theta}} = 0$ effectively 'traps' the algorithm, preventing $\boldsymbol{\theta}$ from changing and finding the global minimum. To combat this, we introduce a 'momentum' term $m\Delta\boldsymbol{\theta}$ with some 'mass' m; see Algorithm 2. This algorithm is inspired by kinematics, in which the momentum of a object at position $\boldsymbol{x}$ and mass m is $m\Delta\boldsymbol{x}$.[3] The two are conceptually similar; even if the gradient $\nabla C_{\boldsymbol{\theta}} = 0$, the parameters will continue updating in the same direction they were before. Higher values of m decide how prone to change the gradient is, effectively giving it inertia.

Algorithm 2 Gradient Descent with Momentum

```
1: Let $\eta$ be the learning rate, typically $\in [10^{-5}, 10^{-1}]$.
2: Let $m$ be the mass, typically 0.2.
3: Initialize $\boldsymbol{\theta}$ with an initial guess.
4: Initialize $\Delta\boldsymbol{\theta} = 0$.
5: while not converged do
6:     Compute the gradient: $\boldsymbol{g} \leftarrow \nabla C_{\boldsymbol{\theta}}$.
7:     Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\eta\boldsymbol{g} + m\Delta\boldsymbol{\theta}$.
8:     Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
9: end while
```

It is also common to want a learning rate that starts large and shrinks during descent, so that large steps are taken early while $\boldsymbol{\theta}$ is far from $\hat{\boldsymbol{\theta}}$. The simplest form of this is having the learning rate decrease exponentially with iteration t: $\eta \to \eta e^{-t}$. Algorithms such as AdaGrad[4] (Algorithm 3, as reproduced on page 304 of Goodfellow *et al.* (2016)) take this one step further by effectively having a separate learning rate for each parameter. Parameters with the greatest change have their learning rates reduced the most.

Algorithm 3 AdaGrad

```
1: Let $\eta$ be the learning rate, typically $\in [10^{-5}, 10^{-1}]$.
2: Let $\delta$ be a small constant for numerical stability, typically $10^{-7}$.
3: Initialize $\boldsymbol{\theta}$ with an initial guess.
4: Initialize $\Delta\boldsymbol{\theta} = 0$.
5: Initialize gradient accumulator $\boldsymbol{r} = 0$.
6: while not converged do
7:     Compute the gradient: $\boldsymbol{g} \leftarrow \nabla C_{\boldsymbol{\theta}}$.
8:     Accumulate the gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$ .
9:     Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.
10:    Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
11: end while
```

Where ADAGrad's accumulator includes the entire descent history, the RMSGrad algorithm (Algorithm 4) reduces the importance of past gradients by a decay parameter ρ. This is done explicitly in line 9 of the algorithm.

Algorithm 4 RMSProp

```
1: Let $\eta$ be the learning rate, typically $\in [10^{-5}, 10^{-1}]$.
2: Let $\rho$ the decay rate, typically 0.99.
3: Let $\delta$ be a small constant for numerical stability, typically $10^{-7}$.
4: Initialize $\boldsymbol{\theta}$ with an initial guess.
5: Initialize $\Delta\boldsymbol{\theta} = 0$.
6: Initialize gradient accumulator $\boldsymbol{r} = 0$.
7: while not converged do
8:     Compute the gradient: $\boldsymbol{g} \leftarrow \nabla C_{\boldsymbol{\theta}}$.
9:     Accumulate the gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$.
10:    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$.
11:    Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
12: end while
```

The most intricate gradient descent variation we explore is ADAM (Algorithm 5). It includes an accumulated first and second moment, and decays each similarly to RMSProp. For a full discussion of this method, see Goodfellow *et al.* (2016).

C. Relationship Between Bias, Variance, and MSE

In this section we derive the connection between the MSE, bias and variance of our model. Let our data y be the sum of a continuous function $f(x)$ and a normal distributed error $\epsilon \sim N(0, \sigma^2)$:

$$y = f(x) + \epsilon. \tag{16}$$

This in turn means we may treat y as a random variable. As in (1), we model with a linear combination of features $\tilde{\boldsymbol{y}} = \mathbf{X}\boldsymbol{\theta}$. The MSE can therefore be expressed as an expectation value:

$$\text{MSE} = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}\left[(y - \tilde{y})^2\right]. \tag{17}$$

[3] To be pedantic, this is only true for small time steps Δt in which the velocity $\boldsymbol{v} = d\boldsymbol{x}/dt \approx \Delta\boldsymbol{x}/\Delta t$, and we scale t so that $\Delta t = 1$.

[4] In algorithms 3, 4, and 5, '$\odot$' symbolizes element-wise multiplication: $\boldsymbol{g} \odot \boldsymbol{g} = [g_1^2, g_2^2, \ldots, g_M^2]$ and the divisions and square roots are applied element-wise.

Algorithm 5 ADAM

```
1:  Let $\eta$ be the learning rate, typically $\in [10^{-5}, 10^{-1}]$.
2:  Let $\rho_1$ and $\rho_2$ be the decay rates, typically $\rho_1 = 0.9, \rho_2 = 0.999$.
3:  Let $\delta$ be a small constant, typically $10^{-8}$.
4:  Initialize $\boldsymbol{\theta}$ with an initial guess.
5:  Initialize $\Delta\boldsymbol{\theta} = 0$.
6:  Initialize first and second moment accumulators $\boldsymbol{r} = 0, \boldsymbol{s} = 0$.
7:  Initialize the timestep $t = 0$.
8:  while not converged do
9:      Compute the gradient: $\boldsymbol{g} \leftarrow \nabla C_{\boldsymbol{\theta}}$.
10:     $t \leftarrow t + 1$.
11:     Accumulate first moment: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$.
12:     Accumulate second moment: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$.
13:     Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$.
14:     Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$.
15:     Compute update: $\Delta\theta \leftarrow -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$.
16:     Update: $\theta \leftarrow \theta + \Delta\theta$.
17: end while
```

Substituting (16) for y:

$$= \mathbb{E}\left[(f + \epsilon - \tilde{y})^2\right]. \tag{18}$$

Now consider that when expanding the square, any first-order ϵ term will have expectation 0, as ϵ is centered around as it is defined as $\epsilon \sim N(0, \sigma^2)$. As expectation values are linear operators we may simplify the expectation as.

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

We also add and subtract by $\mathbb{E}[\tilde{y}]$:

$$= \mathbb{E}\left[\left((f - \mathbb{E}[\tilde{y}] - \tilde{y} - \mathbb{E}[\tilde{y}])\right)^2\right] + \sigma^2. \tag{19}$$

We then expand our expression.

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

Let us focus on the $2ab$ term. Since f and $\tilde{y}$ are independent, we can distribute the expectation:

$$\begin{aligned} \mathbb{E}[2(f-&\mathbb{E}[\tilde{y}])(\tilde{y} - \mathbb{E}[\tilde{y}])] \\ &= 2\mathbb{E}[f - \mathbb{E}[\tilde{y}]]\mathbb{E}[\tilde{y} - \mathbb{E}[\tilde{y}]] \\ &= 2\mathbb{E}[f - \mathbb{E}[\tilde{y}]](\mathbb{E}[\tilde{y}] - \mathbb{E}[\tilde{y}]) \\ &= 2\mathbb{E}[f - \mathbb{E}[\tilde{y}]](0) \\ &= 0. \end{aligned}$$

Using $f \approx y$ we have

$$\begin{aligned} \text{MSE} &= \mathbb{E}[(y - \mathbb{E}[\tilde{y}])^2] + \mathbb{E}[(\tilde{y} - \mathbb{E}[\tilde{y}])^2] + \sigma^2 \qquad (20) \\ &= \text{Bias}[\tilde{y}] + \text{Var}[\tilde{y}] + \sigma^2. \qquad (21) \end{aligned}$$

Since $\mathbb{E}[(y - \mathbb{E}[\tilde{y}])^2]$ is the definition of the bias in $\tilde{y}$ and $\mathbb{E}[(\tilde{y} - \mathbb{E}[\tilde{y}])^2]$ is the variance in $\tilde{y}$. This is the alternate interpretation of our cost function: the sum of our model's bias, its variance, and the variance from the noise.

There is an inherent tension between the bias in and variance of our model. Consider that with increased model complexity, $\tilde{y}$ is better able to reproduce its training data, and the bias decreases. However, increased model complexity makes $\tilde{y}$ begin to reproduce features of the noise, which in turn increases variance when calculate on predictions of test data. This phenomenon is known as the 'bias-variance tradeoff', and is explored further in Figure 10.

D. Stochastic Gradient Descent (SGD)

Calculating the gradient is a computationally expensive task, especially when computing a model for large datasets. It is therefore common to select a random subset of points, referred to here as a *minibatch*, and use their gradient as an estimate for the overall gradient. A minibatch $B = \{(x, y)_1, \ldots, (x, y)_j, \ldots, (x, y)_M\}$ consists of M, also called the *batch size*[5], batch input-output pairs randomly selected from the full dataset with replacement. A feature matrix $\mathbf{X}_{\text{sample}}$ and target $\boldsymbol{y}_{\text{sample}}$ are then constructed from the points in B and are used in place of $\mathbf{X}$ and $\boldsymbol{y}$ in the usual gradient calculation. A new minibatch is selected for every iteration of the descent, since we would like utilize every point within the dataset to contribute to the descent.

E. Resampling Methods

Ideally there would be enough data to split it into one set for training and another for validating that the model accurately reproduces the function $f(x)$. As this is not always the case we therefore introduce resampling methods to approximate metrics from our data. The methods we focus on are *Bootstrapping* and *k-fold cross-validation*. Our goal is to compare these methods in relation to the bias-variance tradeoff introduced in Section II.C. Where we study the effect polynomial degree and number of data points affect the predicted error of models using these methods.

1. Bootstrap

The bootstrap method, as described in (Hastie *et al.*, 2009), is a resampling approach that is used to approx-

[5] It is typical to refer to a number of descent iterations equal to the batch size as one *epoch*; we avoid this language and instead measure by individual iterations.

imate the sampling distribution of a metric. In this exploration we have used bootstrap to estimate the distribution (17) by randomly shuffling and resampling the dataset.

The approximated mean squared error for a model $\tilde{y}$ is then calculated by taking the mean of the distribution, as seen in (22).

$$\widehat{\mathrm{MSE}}_{\mathrm{boot}} = \frac{1}{B}\frac{1}{n}\sum_{b=1}^{B}\sum_{i=1}^{n} L(y_i, \tilde{y}^{*b}(x_i)), \tag{22}$$

where B is the number of bootstraps preformed, n is the number of data-points in the set, L is the loss function for the regression method used, and $\tilde{y}^{*b}(x_i)$ is the prediction made at the b-th bootstrap. The approximation for the bias and variance can be calculated the same way. Due to the randomness in this method, its not guaranteed that all data-points will get to influence the model, therefore increasing variance. This will become apparent when compared to k-fold cross-validation.

2. k-fold cross-validation

k-fold cross-validation is another resampling method described in (Hjorth-Jensen, 2023). Randomly shuffle the dataset and split it into K parts, and let $k = 1, 2, \cdots, K$. Now use the k-th part of the shuffled dataset for validation of the model. Use the remaining $K-1$ to train the model. The cross-validation estimate of the prediction error is then

$$\mathrm{CV}(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{f}^{-k}(x_i)), \tag{23}$$

here N denotes the number of data-points in the dataset, L is the loss function with given parameters, and f^{-k} denotes the function fitted without the k-th fold (Hastie *et al.*, 2009). For models with a tuning parameter α such as Ridge or Lasso regression, the estimate is

$$\mathrm{CV}(\hat{f}, \alpha) = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{f}^{-k}(x_i, \alpha)), \tag{24}$$

where the minimization problem in (24) becomes a matter of finding the tuning parameter $\hat{\alpha}$ that minimizes the $\mathrm{CV}(\hat{f}, \alpha)$. For this exploration we have used $k = 10$.

F. Tools

In the making of this report and in the production of our results we have used the following tools;

1. *Numpy* - scientific programming module for python (Harris *et al.*, 2020).
2. *Matplotlib* - plotting module for python, all figures containing plots have been made with *Matplotlib*.
3. Scikit-Learn - Used some of the functionalities

III. APPLICATION

A. Note on the Data

Starting with 100 evenly spaced input values in the interval $x_i \in [-1, 1]$, we generated output values y_i using (7). We reserved 20% of the data for testing, and trained our models on the remaining 80% of the data. It is common to rescale $\mathbf{X}$ and $\boldsymbol{y}$ so all input features are of the same scale. We neglect to do so, since our x_i and y_i already fall between -1 and 1 and our features are all of the same scale.

B. OLS compared to Ridge Regression

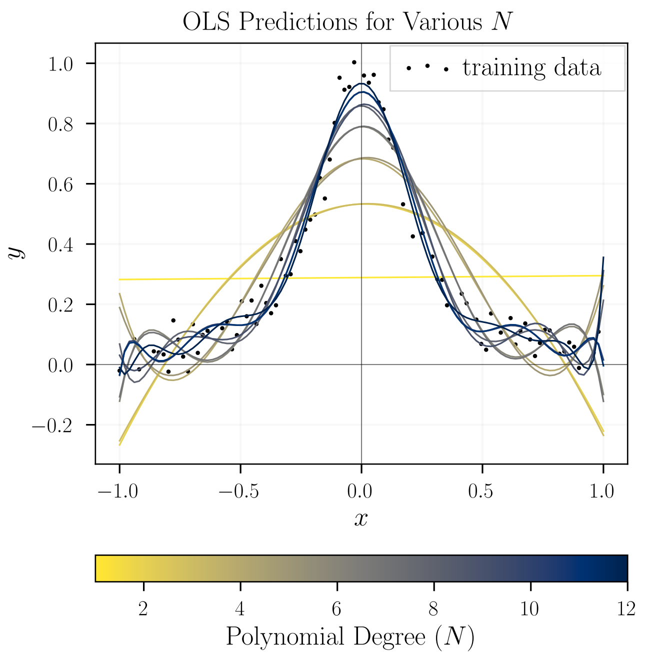

FIG. 1 Demonstration of analytical OLS polynomial fits to $n = 80$ training data points. Fits from degrees 1 to 12 are shown.

We performed linear regression using both the OLS and Ridge methods, the results of which are shown in figures 2 and 3 respectively. As seen in Figure, 2, the OLS method, regressions on small numbers of training data ($n \lesssim 10^2$) typically have large MSE and low R^2 (calculated from predictions of test data), especially for higher order polynomials. This is expected as if we consider the results in Figure 1: starting from $x = 0$ moving towards $x = \pm 1$, the polynomials become more erratic.

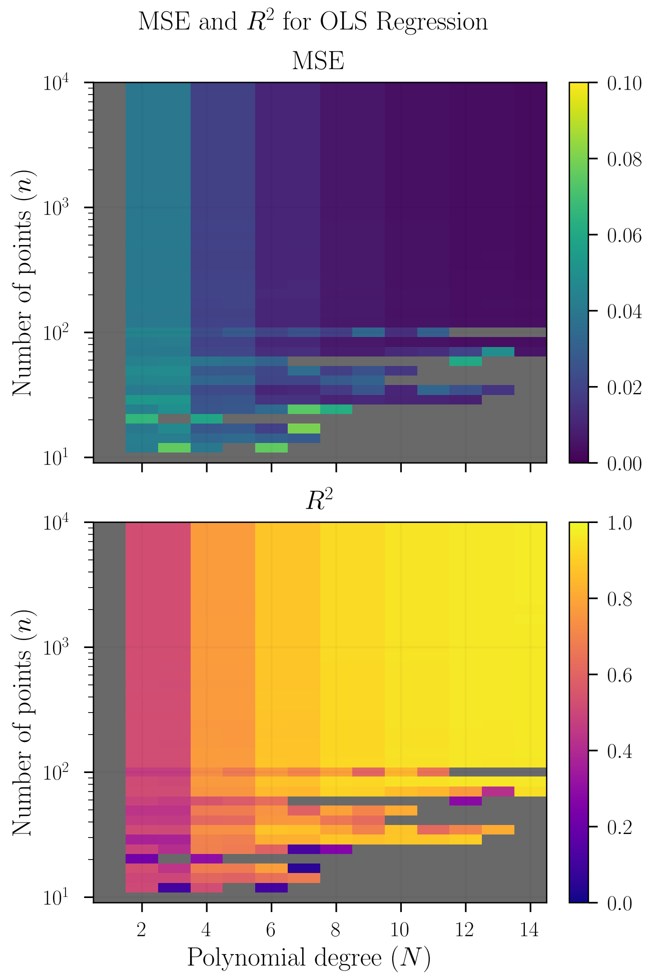

FIG. 2 MSE and R^2 for OLS Regression. Regressions with $R^2 < 0$ have been grayed out. MSE and R^2 are calculated from predictions of the test data.

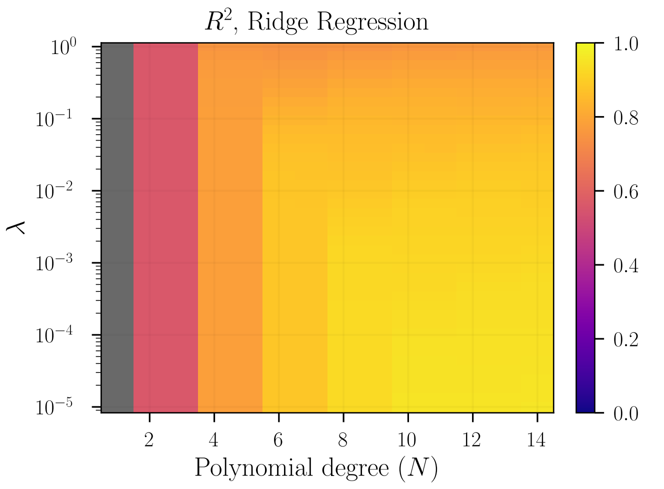

FIG. 3 R^2 for Ridge Regression on $n = 80$ points using various values of λ. Again, regressions with $R^2 < 0$ have been grayed out. MSE follows the same trend.

This is due to the higher-order terms on the models begin to dominate, as the mineralization function does not have a penalty term to stop higher order degrees from dominating. As the models are trained on 80 data points it is also expected that higher order polynomial models diverge from the data. This is as the MSE is dependent on the term $\text{Var} = \mathbb{E}(\tilde{y} - \mathbb{E}(\tilde{y}))$ which effectively states that if there are few points of data the model at higher complexities will then ignore the training-data.

At larger numbers of points ($n \gtrsim 10^2$) we see that the calculated value for MSE and R^2 is entirely dependent on the models polynomial degree. This is expected because shown in figure 11 we see that when we increase the number of points the variance decreases and therefore the MSE decreases and converges towards the bias.

One may recognize in Figure 2 at approximately number of points $n \in [5, 100] \cdot 10^2$ and the models polynomial degree $N \in [10, 14]$ the calculated MSEs and R^2 are approximately the same. This means that from data-points $n = 5 \cdot 10^2$ the models are being trained on enough data to estimate the parameters for the higher order polynomials to be zero. As this trend starts at around $N = 10$, one may think that for more complex models that have polynomial degrees larger than 10, have parameters that are equal to zero.

Seen on Figure 1 notice that the form of the models seem to come in pairs, for example the degree 4 and 5 polynomials are nearly identical. This is an consequence of using the Runge function (6) as our input data. Since the function we are training our models on is symmetric along the x axis and centered at $x = 0$, any polynomial of degree $2N$ will in itself be symmetric and therefore fit best with the Runge function. Therefore for any model that is a polynomial of an odd-degree N will fit best when mimicking the even polynomial of degree $N - 1$.

Turning our attention to Ridge regression in Figure 3, we study the effect of a parameter penalty on linear regression. Small values of $\lambda \in [10^{-5}, 10^{-3}]$ result in nearly identical R^2 to OLS. This is expected as when $\lambda \to 0$ then ridge regression becomes equal to OLS. Values of $\lambda \approx 10^{-1}$ and higher noticeably increase the error by suppressing parameters. This is most pronounced in polynomials of high degree ($N \gtrsim 7$). This is expected as high values of the penalty term $\lambda \in [1, 10] \cdot 10^{-1}$ the parameters are being punished and therefore minimized too much. This in essence removes the degrees of freedom the model has only an effective degree of freedom for each polynomial so that the model can not optimally fit onto the Runge Function. This explains why within the area $N \in [5, 14]$ the calculated R^2 increases with the degrees of freedom for each point, as the effective degrees of freedom is still too few to fit the Runge Function. The high R^2 value is also expected as the penalty factor λ contributes to the bias which in turn contributes increases the value of the calculated R^2. Models with low degree have a low calculated R^2 as the models have too few degrees of freedom for λ to have a meaningful effect on the calculated deviation from the function.

C. Gradient Descent

More interesting than the final MSE is how each method arrived where it did. We observe the MSE for each iteration during training. As a baseline, we use OLS-based gradient descent with a fixed learning rate and no momentum. All methods were tested with $n = 100$, $N = 5$, $\eta = 10^{-1}$, initial guess $\boldsymbol{\theta} = \mathbf{0}$, and made to run for 2×10^4 iterations. Unless otherwise specified, data was not sp

1. OLS with Momentum

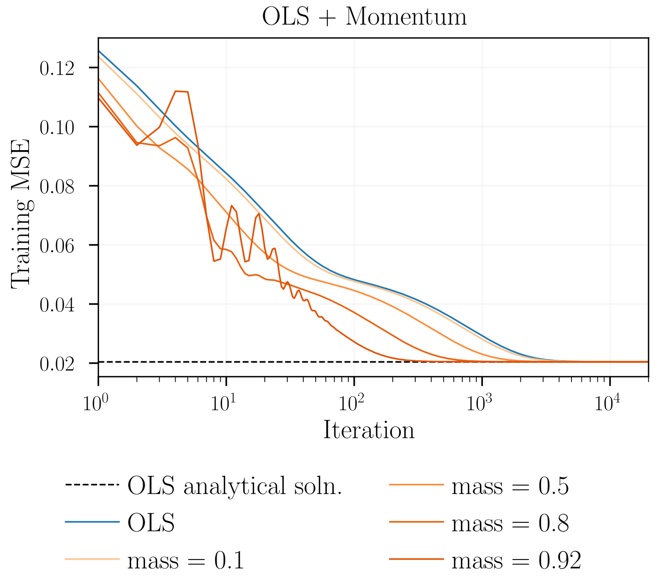

FIG. 4 Behavior of OLS gradient descent while using momentum with various mass values. $n = 100$, $N = 5$, $\eta = 10^{-1}$, initial guess $\boldsymbol{\theta} = \mathbf{0}$.

Figure 4 shows the effect of a momentum term on OLS gradient descent. With a mass of 0.5, the behavior is similar to massless OLS gradient descent, only with consistently faster convergence. This is expected as the momentum "pushes" the gradient towards the local minima. The tradeoff, which is instability, but faster conversance, with momentum is apparent at mass= 0.8 and clear by mass= 0.92: the parameter values begin to oscillate as the high momentum prevents the model from settling into its global minimum. At masses greater than 0.92, the model may fail to converge entirely, so it is not always the case that large values for masses contribute faster convergence. Choice of mass is therefore an important consideration.

2. Lasso

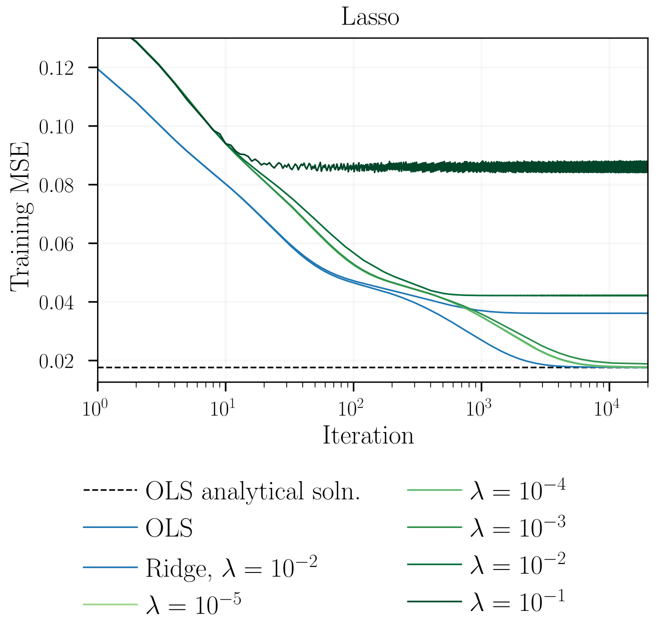

FIG. 5 Behavior of Lasso gradient descent. Ridge-based descent is included for comparison. $n = 100$, $N = 5$, $\eta = 10^{-1}$, initial guess $\boldsymbol{\theta} = \mathbf{0}$.

We perform Lasso gradient descent in Figure 5. We see in the figure Lasso converges overall slower than OLS, though with a similar behavior as their cost functions are closely related. λ determines where the descent ends which is expected as it restricts how large of a value the parameters can have and therefore the degrees of freedom of the model. The penalty factor introduces an inherent bias which contributes to the cost function. This is seen on the aforementioned figure with larger values of λ diminishing $\boldsymbol{\theta}$ and increasing the MSE the lasso gradient descent converges at. For small λ, the method is fairly stable; only at $\lambda = 10^{-1}$ do we begin to see some instability. On the two $\lambda = 10^{-2}$ lines Lasso regression behaves notably different to Ridge regression where Lassos MSE is less than Ridges. This is expected as Lassos cost function is less punished by the λ values than Ridge as Lasso has a $\lambda||\boldsymbol{\theta}||_1$ term while Ridge has a $\lambda||\boldsymbol{\theta}||_2^2$ term. This essentially means that the bias introduced by the λ value is less in Lasso than Ridge which is what we can see with their calculated MSE. Ridge regression is nearly identical to OLS until the $\approx 10^2$-nd iteration where the parameter penalty begins to take over as the maximum value of the parameters are being restricted. As seen the Lasso regression starts with a higher MSE which is expected as the initial value of Lasso gets an extra contribution from its sign function with our initial guess. This does not apply to the Ridge and OLS cost functions and therefore they start with equal values.

3. RMSProp

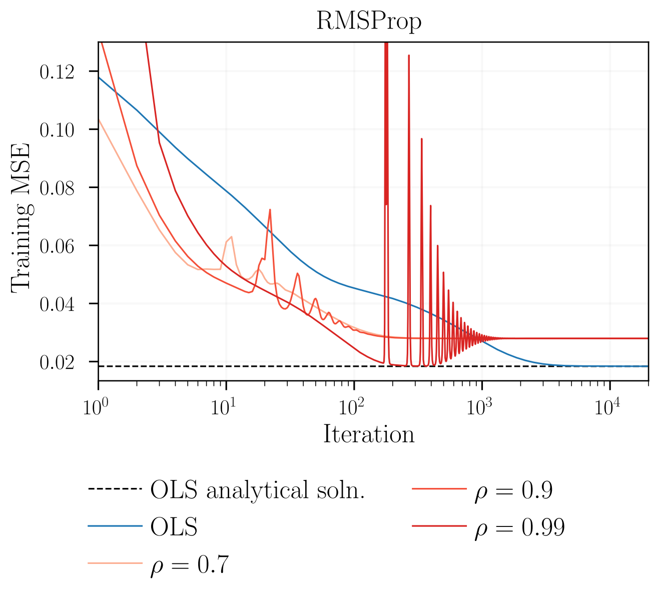

FIG. 6 Behavior of the RMSProp algorithm for three values of decay parameter ρ (all with $\delta = 10^{-7}$ for numerical stability). $n = 100$, $N = 5$, $\eta = 10^{-1}$, initial guess $\boldsymbol{\theta} = \mathbf{0}$.

Looking at Figure 6, we immediately see both the good and bad of the RMSProp algorithm. It converges much faster: reaching an MSE of 0.04 took only ~ 40 iterations versus ~ 300 for OLS. However, the algorithm becomes very unstable when it is close to converging, owing to the division by the gradient when updating $\Delta\theta$. This can be mitigated somewhat by using a larger value of δ. However, we instead recommend using a ρ close to 1 along with a more generous stopping criterion. Notice that with $\rho = 0.99$ RMSProp was very close to the optimal parameters in one tenth the iterations of OLS.

4. ADAM

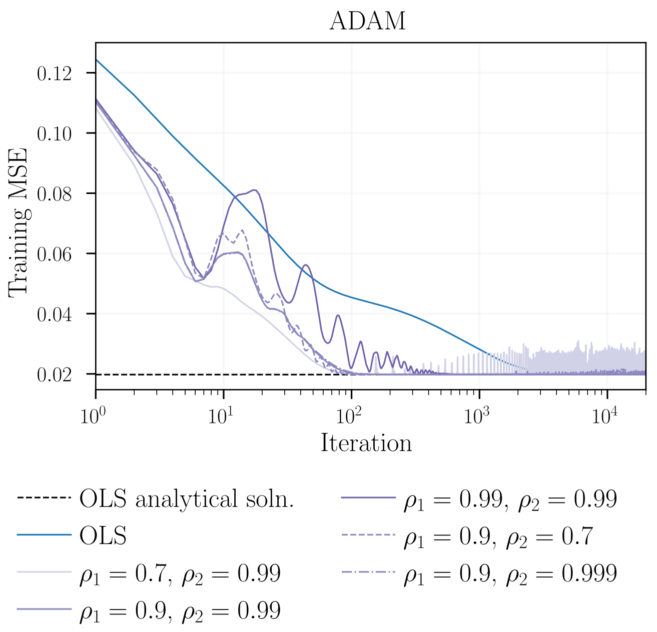

FIG. 7 Behavior of the ADAM algorithm for various combinations of ρ_1 and ρ_2 (all with $\delta = 10^{-8}$). $n = 100$, $N = 5$, $\eta = 10^{-1}$, initial guess $\boldsymbol{\theta} = \mathbf{0}$.

The ADAM algorithm (Figure 7) also converges faster than OLS descent. With ρ_1 farther from 1 (faintest solid line), the initial convergence is quicker. The closer ρ_1 gets to 1, the more oscillations are introduced. The opposite is true for ρ_2; notice how the more eratic dashed line is that with $\rho_2 = 0.7$ (the $\rho_2 = 0.999$ line follows the solid so close that it may be difficult to see). As with RMSProp, we recommend ρ values close to the default along with a stopping criterion.

5. Combined comparison

Figure 8 directly compares each of our gradient descent techniques. We consider OLS gradient descent our base case. Ridge regression follows it closely, diverging only when the parameter penalty grows large enough. Momentum is also similar, converging faster. Lasso performs worse than OLS owing to the contribution from $\lambda||\boldsymbol{\theta}||_1$ in the cost function. ADAGrad does well, and of the three flexible-learning-rate methods is the most stable. RMSProp converges faster but is considerably unstable. Finally, the ADAM algorithm converges fastest overall.

6. Stochastic Gradient Descent

Results from our implementation of stochastic gradient descent are shown in Figure 9. Notice that for the OLS, momentum, ADAGrad, and ADAM methods, increasing the batch size generally decreases the final MSE, with some fluctuations owing to the randomness inherent to SGD. The RMSProp, Lasso, and Ridge methods were all too dominated by their overall bias for batching to have a significant effect on their final accuracy.

D. Resampling Methods

Figure 10 shows the bias-variance tradeoff in action. We observe the estimated error follow the bias for low degrees which equates to the underfitting region of the plot. In the middle we observe a global minima in the estimated error for these models, exactly where bias and variance meet. The degrees from there represent the region where the model begins overfitting to the training data therefore giving growing and the estimated error

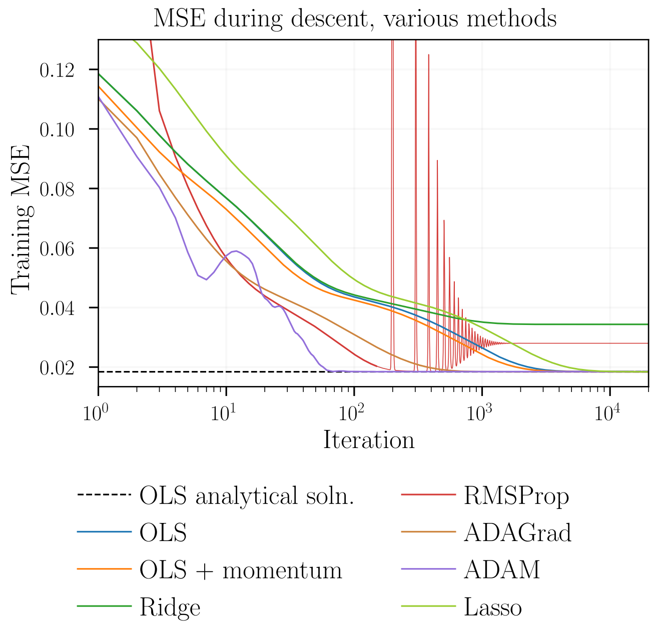

FIG. 8 Comparison of MSE during gradient descent for different methods. Each method fits a 5th order polynomial to the same 100 points with parameters close to default (Momentum: mass= 0.2; Ridge: $\lambda = 10^{-2}$; RMSGrad: $\delta = 10^{-7}, \rho = 0.99$; ADAGrad: $\delta = 10^{-7}$; ADAM: $\delta_1 = 0.9, \delta_2 = 0.9$, Lasso: $\lambda = 10^{-7}$).

FIG. 9 Stochastic results for different batch sizes during the regression of a 5th degree polynomial on 10^4 data points. ΔMSE is the difference between the MSE from the analytical OLS solution and the descent solution after 10^4 iterations. Model parameters are the same as in Figure 8. The final values for momentum and ADAGrad are on the order of 10^{-9} and 10^{-14} respectively.

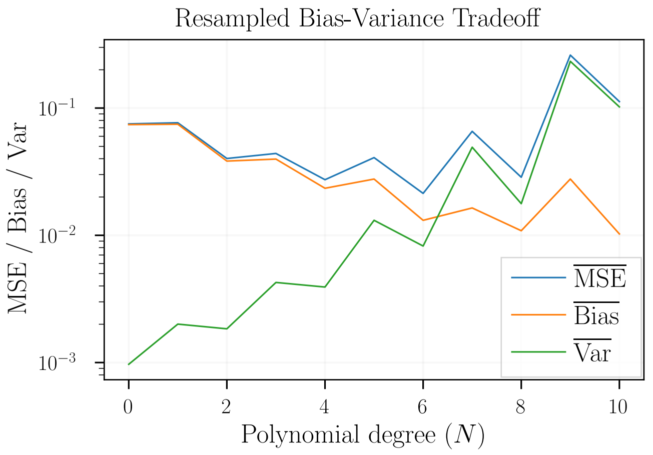

FIG. 10 The estimated error, bias, and variance of the training set as a function of polynomial degree. Computed using $B = 10^3$ bootstraps with replacement on a dataset of $n = 10^2$ points split as 80/20% for training and test data respectively.

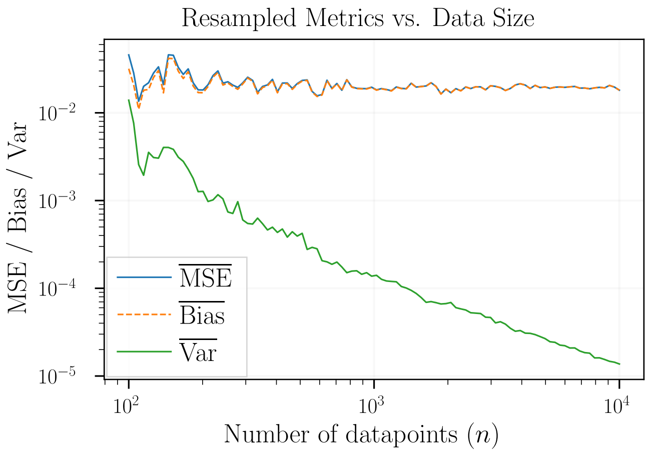

FIG. 11 The estimated error, bias, and variance of the test set as a function of the number of points in the dataset. The polynomial degree is pinned at $d = 5$ and the number of bootstraps is $B = 10^3$ with replacement.

grows accordingly.

In Figure 11 we observe how the estimates of the MSE, bias, and variance behave as we vary the number of datapoints in the set and leave the polynomial degree constant. Observe how the estimate of the MSE begins following the estimate of the bias as the estimate of the variance decreases. This validates (21).

In Figure 12 we compare the bootstrap and the k-fold-CV methods. From the figure we observe that the methods preform similarly, with the estimated error of the training sets closely following each-other up to $N = 23$ where bootstrap begins to grow away from k-fold cross-validation. This is related to the fact that variance grows, as discussed for Figure 10, the bootstrap method ends up sampling the same values further increasing variance. That is where bootstrap and k-fold-CV, bootstrap doesn't guarantee that all data-points influence the model, meaning that bootstrap overestimates the approximated MSE for a high polynomial degree model.

In Figure 13 we observe that OLS begins overfitting to the training data at higher degree polynomials, Ridge and Lasso don't exhibit this behavior. This is a crucial result showing how OLS tends to perfectly replicate the test data at higher model complexities, this can also be seen in Figure 14. This is a direct result of penalizing large coefficients.

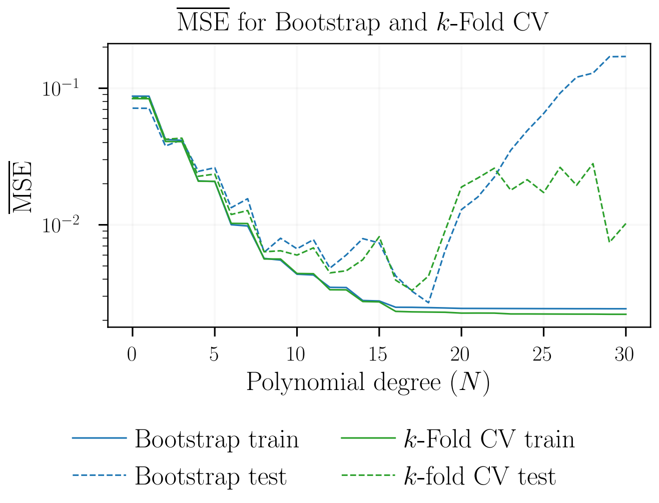

FIG. 12 The estimated error of the training and test datasets vs polynomial degree for the bootstrap and k-fold CV methods. The dataset is $n = 10^2$ points large using a 80/20% split between test and train. Using $B = 10^3$ bootstraps without replacement and $k = 10$ folds.

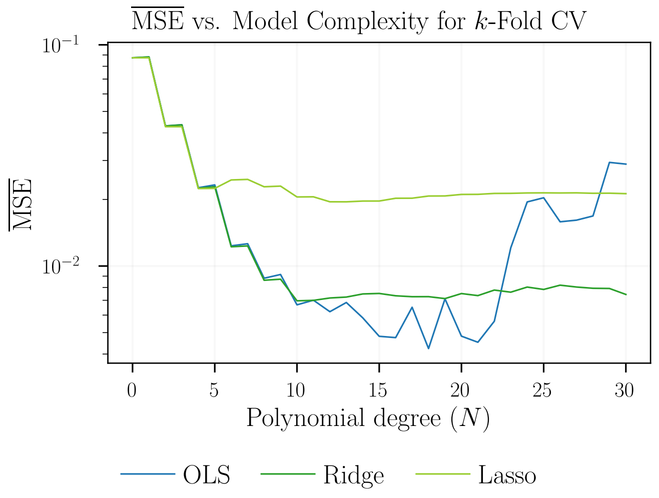

FIG. 13 The estimated error of the test set, using $k = 10$ folds up to a 30-th degree polynomial. The Ridge and Lasso lines represent the best solution among 30 λ values between 10^{-5} to 10^{-5}.

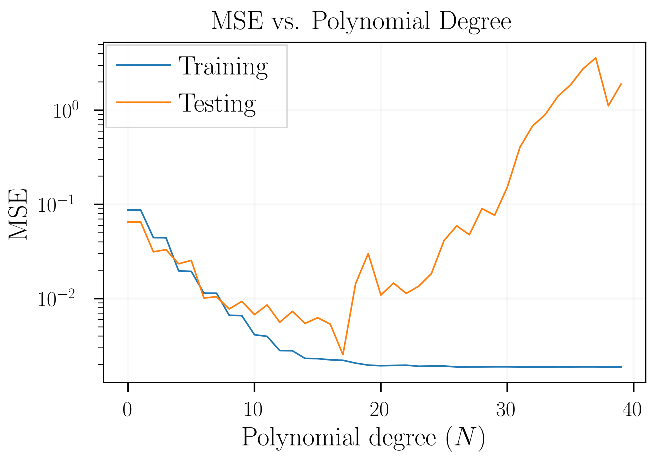

FIG. 14 The test and the training fit for polynomial degrees up to $N = 40$ using OLS.

IV. CONCLUSION

We have explored many different methods of linear regression. Analytic methods such as OLS and Ridge performed better as as model complexity N increased. However, higher-complexity models required more data to be successful. With these, were were able to observe the characteristic bias-variance tradeoff. Gradient descent based methods generally gave better results the longer they iterated and the higher their learning rate. OLS, Ridge, momentum, and Lasso were all numerically stable, with differences in final error resulting only from penalty parameters. As for the flexible-learning-rate methods, ADAM performed the best overall, AdaGrad did fairly well, and RMSProp was consistantly unstable. We went on to explore the Bootstrap and k-fold CV methods of resampling, which successfully approximated the MSE, bias, and variance. These confirmed the relationship between increased data size and decreased variance, and demonstrated that regression methods with a parameter penalty had lower bias at high complexity than those without.

All of our models were tested by fitting polynomials of various degree to a one-dimensional Runge function. We implemented each method in a custom python regression library. 14.

REFERENCES

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

Harris, C. R., 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.* (2020), Nature **585** (7825), 357.

Hastie, T., R. Tibshirani, and J. Friedman (2009), *The Ele-*

ments of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition (Springer Series in Statistics).

Hjorth-Jensen, M. (2023), “Applied data analysis and machine learning – lecture notes,” https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html, accessed: 2025-10-05.