

APPM 4600 Project 3: Regularization in Least Squares

Alexey Yermakov

Logan Barnhart

Tyler Jensen

November 20, 2022

Abstract

This project explores data fitting polynomials of various degrees to noisy data. As such, the project solves over-determined systems of equations. The normal equation is used with an extra regularization term, resulting in what is commonly called the “Ridge Estimator”, which can be used to solve these systems in software. This is further generalized by the Tikhonov Estimator, which is explored in this paper as well. Afterwards, Elastic Net is applied to similar problems to those introduced for Ridge Regression. We find that depending upon the application, different methods of regularization improve performance upon Ordinary Least Squares.

1 Introduction

1.1 Assumptions

This project makes heavy use of linear algebra. As such, the reader should be familiar with common matrix operations, such as: matrix multiplication, matrix addition, and what the transpose of a matrix is. Additionally, the reader should be comfortable with the concept of a derivative. Further, the reader should understand how a vector and a matrix are related. Lastly, the reader should be familiar with the 1-Norm and 2-Norm for vectors.

1.2 Background

Data fitting is the process of fitting a function to a set of collected data. In this project, the fitting is done by finding the coefficients of the sum of a set of basis functions. This project considers $\{1, x, x^2, \dots, x^m\}$ as the set of basis functions, where m is chosen differently depending on the degree of the polynomial that is being fit to the data. So, given collected data like $\{x_i, y_i\}$ ($0 \leq i \leq n$), one can try to fit a polynomial of degree m ($p_m(x) = \beta_0 + \beta_1 * x + \dots + \beta_m * x^m$) to the collected data. This project assumes $m < n$, resulting in an over-determined system. To determine how close of a fit a polynomial is to data, one needs to define a metric describing how “close” the approximation is to the data. One simple metric is that at each data point $\{x_i, y_i\}$, the approximation is off by the difference in y -values for each data point: $\sum_i |p_m(x_i) - y_i|$.

This exact idea is used to derive the normal equation [4]. Note that $p_m(x_i) = \beta_0 + \beta_1 * x_i + \dots + \beta_m * x_i^m$. Using this as motivation, the following matrices are introduced:

$$\mathbf{X} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^m \\ 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

So, $\mathbf{X}\boldsymbol{\beta} - \mathbf{y}$ is given by:

$$\mathbf{X}\boldsymbol{\beta} - \mathbf{y} = \begin{bmatrix} p_m(x_0) - y_0 \\ p_m(x_1) - y_1 \\ p_m(x_2) - y_2 \\ \vdots \\ p_m(x_n) - y_n \end{bmatrix} = \begin{bmatrix} \beta_0 + \beta_1 * x_0 + \beta_2 * x_0^2 + \dots + \beta_m * x_0^m - y_0 \\ \beta_0 + \beta_1 * x_1 + \beta_2 * x_1^2 + \dots + \beta_m * x_1^m - y_1 \\ \beta_0 + \beta_1 * x_2 + \beta_2 * x_2^2 + \dots + \beta_m * x_2^m - y_2 \\ \vdots \\ \beta_0 + \beta_1 * x_n + \beta_2 * x_n^2 + \dots + \beta_m * x_n^m - y_n \end{bmatrix}$$

To get the sum of the absolute values of each term in $\mathbf{X}\boldsymbol{\beta} - \mathbf{y}$, one can pass the vector into the 1-Norm. The next step is to find the $\boldsymbol{\beta}$ such that $\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_1$ is minimized, resulting in the formula: $\arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_1$. The solution to this formula then gives the coefficients to $p_m(x)$ that fit the polynomial to the data and minimizes this simple metric. Traditionally, the 2-Norm is used: $\arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2$. This leads to a closed form solution to the equation $\arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2$: $\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$, which is called Ordinary Least Squares.

Throughout the rest of the paper, various regularization terms will be added to the Ordinary Least Squares minimization problem to create an improved data fitting technique, because Ordinary Least Squares is prone to overfitting to data and does not perform well with ill-conditioned systems.

2 Ridge Regression

2.1 Deriving the Ridge Estimator

One way to add regularization is by penalizing the relative size of the coefficients of the approximation by adding a $\gamma \|\boldsymbol{\beta}\|_2^2$ term to the function $\arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2$. This is a regularized extension to Ordinary Least Squares called Ridge Regression.

$$\arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \gamma \|\boldsymbol{\beta}\|_2^2 \quad (1)$$

The solution $\boldsymbol{\beta}$ can be found analytically:

$$\begin{aligned} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \gamma \|\boldsymbol{\beta}\|_2^2 &= (\mathbf{X}\boldsymbol{\beta} - \mathbf{y})^T (\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ &= (\mathbf{X}\boldsymbol{\beta})^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ &= (\boldsymbol{\beta}^T \mathbf{X}^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ &= \boldsymbol{\beta}^T \mathbf{X}^T \mathbf{X} \boldsymbol{\beta} - \boldsymbol{\beta}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X} \boldsymbol{\beta} + \mathbf{y}^T \mathbf{y} + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \end{aligned}$$

So,

$$\begin{aligned}\beta^T \mathbf{X}^T \mathbf{X} \beta - \beta^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X} \beta + \mathbf{y}^T \mathbf{y} + \gamma \beta^T \beta &= \beta^T \mathbf{X}^T \mathbf{X} \beta - 2\mathbf{y}^T \mathbf{X} \beta + \mathbf{y}^T \mathbf{y} + \gamma \beta^T \beta \\ &= (\mathbf{X} \beta)^T (\mathbf{X} \beta) - 2\mathbf{y}^T \mathbf{X} \beta + \mathbf{y}^T \mathbf{y} + \gamma \beta^T \beta\end{aligned}\quad (2)$$

Note that to find the minimum of the Ridge Regression equation given by 1, the derivative of 2 must be equal to zero. Since this problem is globally convex (see 6.1.3), the solution to the problem is where the derivative is equal to zero.

$$\begin{aligned}\frac{d}{d\beta} ((\mathbf{X} \beta)^T (\mathbf{X} \beta) - 2\mathbf{y}^T \mathbf{X} \beta + \mathbf{y}^T \mathbf{y} + \gamma \beta^T \beta) &= 0 \\ 0 &= 2\mathbf{X}^T \mathbf{X} \beta - 2\mathbf{X}^T \mathbf{y} + 2\gamma \mathbf{I} \beta = \mathbf{X}^T \mathbf{X} \beta - \mathbf{X}^T \mathbf{y} + \gamma \mathbf{I} \beta = \mathbf{X}^T \mathbf{X} \beta + \gamma \mathbf{I} \beta - \mathbf{X}^T \mathbf{y} \\ 0 &= (\mathbf{X}^T \mathbf{X} + \gamma \mathbf{I}) \beta - \mathbf{X}^T \mathbf{y} = (\mathbf{X}^T \mathbf{X} + \gamma \mathbf{I}) \beta = \mathbf{X}^T \mathbf{y} \\ \beta &= (\mathbf{X}^T \mathbf{X} + \gamma \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}\end{aligned}\quad (3)$$

Thus, 3 is the solution to the equation for Ridge Regression (equation 1). A more rigorous derivation can be found in the appendix: section 6.1.1.

2.2 Exploring the Ridge Estimator

2.2.1 Fitting A Linear Model

For all numerical results, the code is written in Python 3, random data was generated using Numpy's random number generator. In this section we sample 20 equispaced samples from the line $y = 3x + 2$ on the interval $[-5, 5]$, then Gaussian noise is added from a standard normal distribution. We then randomly sample 10 of these data points and use them to solve for a model using Ridge Regression; these points are called the *training data*. The remaining 10 points are called the *validation data* and are used to calculate the *Residual Sum of Squares* (RSS), the chosen measurement of accuracy. We are fitting to a one-degree polynomial.

$$RSS = \sum_{i=0}^n (y_{predict,i} - y_{valid,i})^2 \quad (4)$$

We begin with $\gamma = 0$ (Ordinary Least Squares) and $\gamma = 0.1$ using a random seed = 50. The results are visualized in Figure 1 and the relevant data is in Table 1.

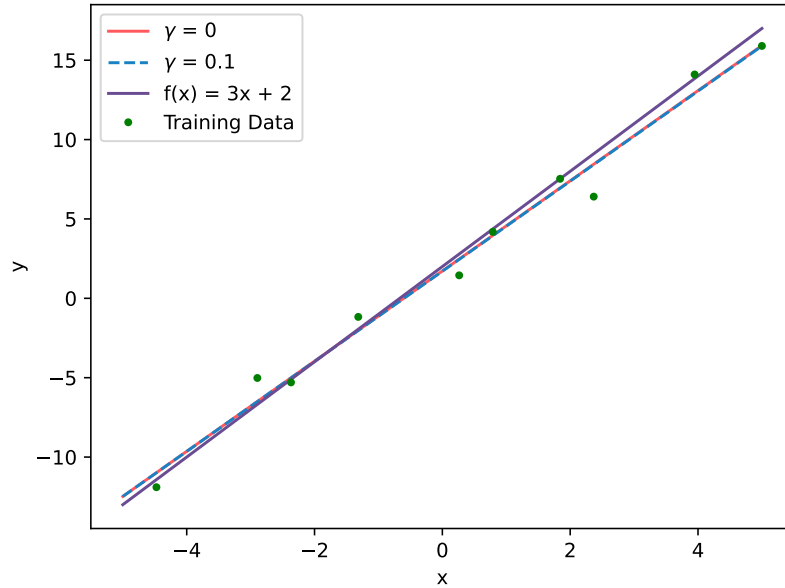


Figure 1: The function $y = 3x + 2$ fitted to a 1-degree polynomial using Ridge Regression for $\gamma = 0$, $\gamma = 0.1$. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

	$\gamma = 0$	$\gamma = 0.1$
RSS	5.023	5.073

Table 1: Residual Sum of Squares for Ridge Regression, using $\gamma = 0$ and $\gamma = 0.1$ for standard noisy evaluations of $y = 3x + 2$ on $x \in [-5, 5]$. 20 equispaced samples were randomly divided into the training and validation data.

From Table 1, we see that Ridge Regression performed worse than Ordinary Least Squares, but we only compared with one γ value, so we try a wide range of γ values. The results are visualized in Figure 2.

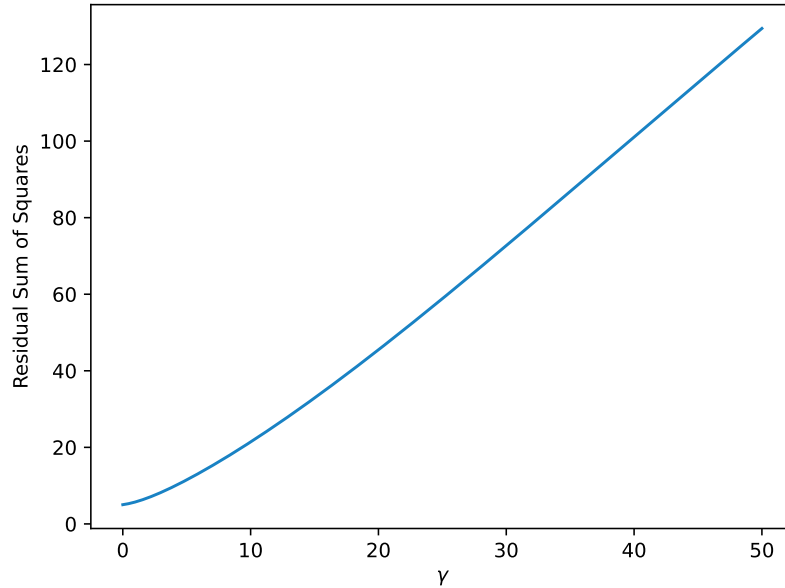


Figure 2: The RSS for $y = 3x + 2$ fitted to a 1-degree polynomial using Ridge Regression for $\gamma \in [0, 50]$. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

It appears that with this data Ordinary Least Squares is always better than Ridge Regression. The issue is that since the data has randomly added Gaussian noise and is randomly partitioned, results are dependant upon the random seed, so we repeat the experiment for 100 different seeds and observe the behavior in aggregate. The results are summarized in Table 2.

	Optimal $\gamma = 0$	Optimal $\gamma > 0$
Number of Seeds	47	53

Table 2: The number of different seeds who had the minimum Residual Sum of Squares for Ridge Regression as either $\gamma = 0$ or $\gamma > 0$ for standard noisy evaluations of $y = 3x + 2$ on $x \in [-5, 5]$. 20 equispaced samples were randomly divided into the training and validation data.

According to Table 2, it appears that Ridge Regression results in a better fit more frequently (or at least as often) as Ordinary Least Squares more often than not resulting in a smaller RSS.

Another way to visualize this result is in Figure 3.

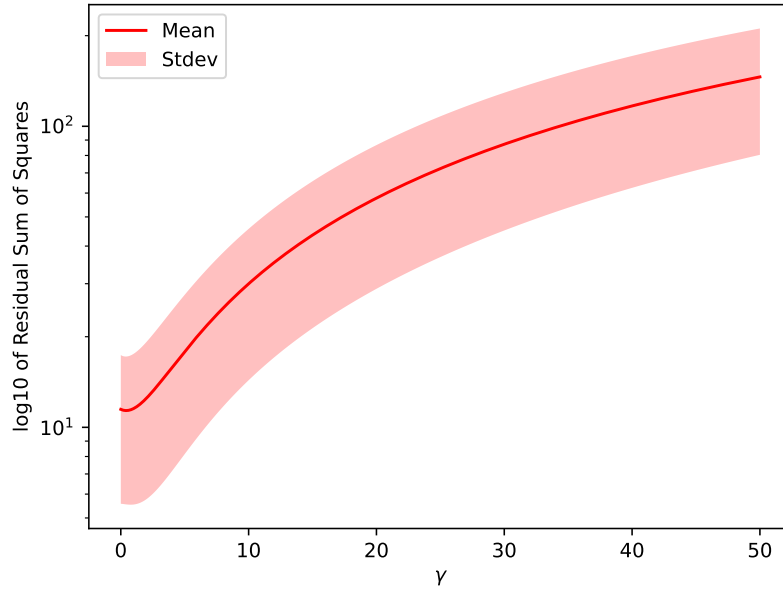


Figure 3: The mean and standard deviation of the RSS for $y = 3x + 2$ fitted to a 1-degree polynomial using Ridge Regression for $\gamma \in [0, 50]$ using 100 different seeds. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

We can see that on average, RSS is minimized at $\gamma \approx 0.4004004$ with mean $RSS \approx 11.3675$. On average, the best model for randomized data with $y = 3x + 2$ is Ridge Regression with $\gamma \approx 0.4004004$.

2.2.2 Fitting A Higher Order Model

Next, we repeat the process, instead sampling data from $y = x^2$ and assuming that the solution has the form $y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$. The Ridge Regression fit is visualized in Figure 4.

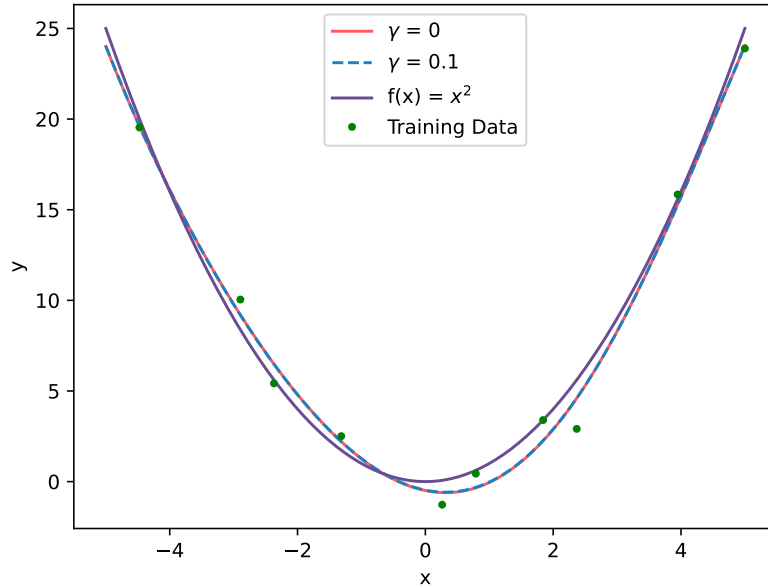


Figure 4: The function $y = x^2$ fitted to a 5-degree polynomial using Ridge Regression for $\gamma = 0$, $\gamma = 0.1$. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

	$\gamma = 0$	$\gamma = 0.1$
RSS	7.621	7.465

Table 3: Residual Sum of Squares for Ridge Regression, using $\gamma = 0$ and $\gamma = 0.1$ for standard noisy evaluations of $y = x^2$ on $x \in [-5, 5]$. 20 equispaced samples were randomly divided into the training and validation data.

From Table 3, we see that Ridge Regression performed better than Ordinary Least Squares for this example, but we're still going to explore these results for many values of γ . The results are visualized in Figure 5.

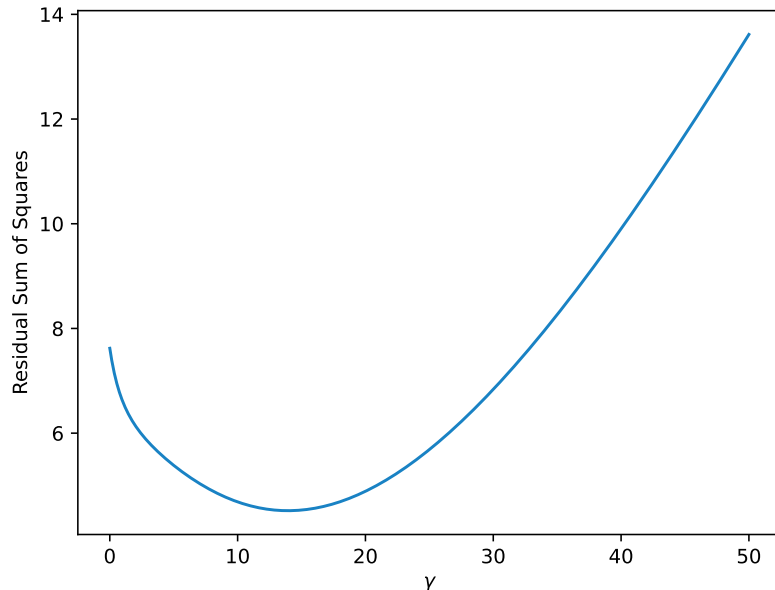


Figure 5: The RSS for $y = x^2$ fitted to a 5-degree polynomial using Ridge Regression for $\gamma \in [0, 50]$. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

We see here that there are many values of γ that outperform Ordinary Least Squares. The optimal estimation occurred around $\gamma = 13$. Similar to the last example though, we need to robustly explore the behavior for many different random seeds to confidently make any conclusions about Ridge Regression. So, we repeat the experiment for 100 different seeds. We summarize our results in Table 4.

	Optimal $\gamma = 0$	Optimal $\gamma > 0$
Number of Seeds	17	83

Table 4: The number of different seeds who had the minimum Residual Sum of Squares for Ridge Regression as either $\gamma = 0$ or $\gamma > 0$ for standard noisy evaluations of $y = x^2$ on $x \in [-5, 5]$. 20 equispaced samples were randomly divided into the training and validation data.

From Table 4, we once again see that regularization results in a better fit than no regularization, but much more frequently than the previous example. We can illustrate this by finding the γ that results in the smallest mean RSS.

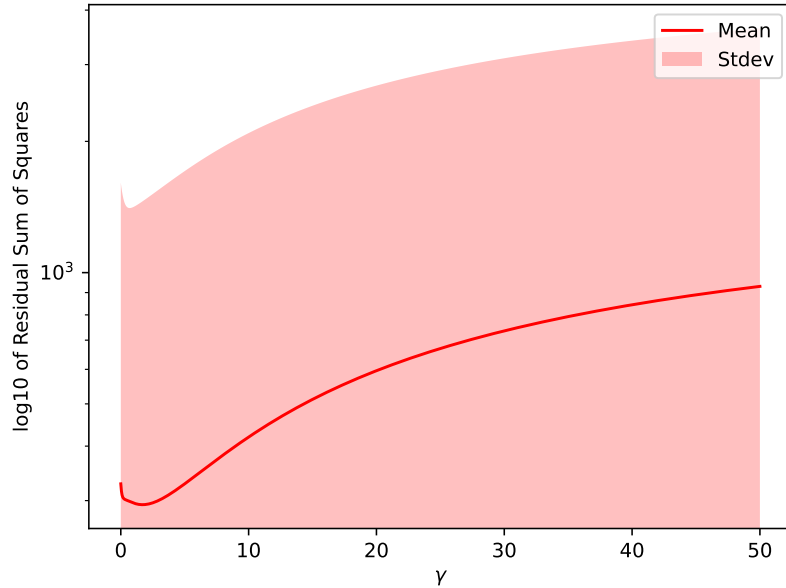


Figure 6: The mean and standard deviation of the RSS for $y = x^2$ fitted to a 5-degree polynomial using Ridge Regression for $\gamma \in [0, 50]$ using 100 different seeds. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

From Figure 6, we can see that our mean RSS is minimized at $\gamma \approx 1.7017$ with mean $RSS \approx 293.6028$. So, on average the best model between Least Squares and Ridge Regression would be Ridge Regression with $\gamma \approx 1.7107$.

One may wonder why Ridge Regression performs better for this specific data compared to the previous example. Because we are trying to fit data from $y = x^2$ to the model $y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$, we don't want all of those coefficients in our result for an accurate fit of this data - ideally, $a = b = c = e = f = 0$ for a perfect fit. When we use Ridge Regression, we penalize the magnitude of these coefficients and make them smaller than Ordinary Least Squares would.

If we compare this to the first example we tested, $y = 3x + 2$ fit to $y = mx + b$, there are no unnecessary polynomial coefficients in the model we chose for this data, thus Ridge Regression is less helpful.

Looking back at the lowest mean RSS for both numerical examples however, it can be seen that Ridge Regression outperforms Ordinary Least Squares.

3 Tikhonov Regression

Before we begin, it's important to add a little bit of machinery about centered differences. Centered differences is a finite difference method used to approximate the derivative of a function, it can be expressed as

$$\left. \frac{df}{dt} \right|_{t_0} \approx \frac{f(t_0 + \Delta t) - f(t_0 - \Delta t)}{2\Delta t}$$

where Δt is the step size.

In Ridge Regression, the magnitude of a solution can be penalized to get more realistic behavior - but magnitude is just the beginning - finite differences can be used to penalize certain traits of the solution in a similar way to Ridge Regression.

The minimization problem used for Ridge Regression can be further generalized by noting that:

$$\arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \gamma \|\boldsymbol{\beta}\|_2^2 = \arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \|\sigma \mathbf{I}\boldsymbol{\beta}\|_2^2$$

where $\sigma = \sqrt{\gamma}$. If $\sigma \mathbf{I}$ is replaced with various other *weight matrices* then the penalization can be applied to other characteristics of $\boldsymbol{\beta}$.

Notably, centered differences can be used to estimate the derivative of a vector with the matrix:

$$\mathbf{D} = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{2} & 0 & \dots & 0 \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \quad (5)$$

Note that \mathbf{D} is $(n-2) \times n$. If this is used as the weight matrix, the minimization problem becomes

$$\arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \lambda^2 \|\mathbf{D}\boldsymbol{\beta}\|_2^2$$

Where λ is just a general weighting constant to control how much the last term is punished. Now the derivative of $\boldsymbol{\beta}$ can be penalized rather than its magnitude.

Simply minimizing the magnitude of $\boldsymbol{\beta}$ can still result in solutions with extreme oscillatory behavior, so although the solution is bounded, it might not be accurately modeling the behavior of the data. When the derivative is penalized, the goal is that a more generalizable solution is found.

3.1 Deriving the Tikhonov Estimator

As was seen when deriving the Ridge Estimator, to find this estimator one needs to solve

$$\frac{d}{d\boldsymbol{\beta}} [||\mathbf{X}\boldsymbol{\beta} - \mathbf{y}||_2^2 + \lambda^2 ||\mathbf{D}\boldsymbol{\beta}||_2^2] = 0$$

expanding that,

$$\frac{d}{d\boldsymbol{\beta}} [(\mathbf{X}\boldsymbol{\beta})^T(\mathbf{X}\boldsymbol{\beta}) - 2\mathbf{y}^T\mathbf{X}\boldsymbol{\beta} + \mathbf{y}^T\mathbf{y} + \lambda^2(\mathbf{D}\boldsymbol{\beta})^T(\mathbf{D}\boldsymbol{\beta})] = 0$$

From the Ridge Estimator derivation it's already known that

$$\begin{aligned}\frac{d}{d\boldsymbol{\beta}}[(\mathbf{X}\boldsymbol{\beta})^T\mathbf{X}\boldsymbol{\beta}] &= 2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} \\ \frac{d}{d\boldsymbol{\beta}}[2\mathbf{y}^T\mathbf{X}\boldsymbol{\beta}] &= 2\mathbf{X}^T\mathbf{y}\end{aligned}$$

So all that is needed is $\frac{d}{d\boldsymbol{\beta}}[(\mathbf{D}\boldsymbol{\beta})^T\mathbf{D}\boldsymbol{\beta}]$.

$$\frac{d}{d\boldsymbol{\beta}} [(\mathbf{D}\boldsymbol{\beta})^T(\mathbf{D}\boldsymbol{\beta})] = 2\mathbf{D}^T\mathbf{D}\boldsymbol{\beta}$$

(See 6.1.2 for a rigorous derivation.)

Now to solve for the $\boldsymbol{\beta}$ that satisfies

$$\frac{d}{d\boldsymbol{\beta}} [||\mathbf{X}\boldsymbol{\beta} - \mathbf{y}||_2^2 + \lambda^2 ||\mathbf{D}\boldsymbol{\beta}||_2^2] = 0$$

or,

$$\begin{aligned}\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{y} + \lambda^2\mathbf{D}^T\mathbf{D}\boldsymbol{\beta} &= 0 \\ \mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + \lambda^2\mathbf{D}^T\mathbf{D}\boldsymbol{\beta} &= \mathbf{X}^T\mathbf{y} \\ (\mathbf{X}^T\mathbf{X} + \lambda^2\mathbf{D}^T\mathbf{D})\boldsymbol{\beta} &= \mathbf{X}^T\mathbf{y}\end{aligned}$$

Thus

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda^2\mathbf{D}^T\mathbf{D})^{-1}\mathbf{X}^T\mathbf{y}$$

Generally speaking, a minimization problem with weight matrix $\boldsymbol{\Gamma}$ of the form

$$\arg \min_{\boldsymbol{\beta}} ||\mathbf{X}\boldsymbol{\beta} - \mathbf{y}||_2^2 + ||\boldsymbol{\Gamma}\boldsymbol{\beta}||_2^2$$

will be solved by

$$\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X} + \boldsymbol{\Gamma}^T \boldsymbol{\Gamma})^{-1} \mathbf{X}^T \mathbf{y}$$

but weight matrices can be customized to such a degree that it's best to verify this property for each case [2].

3.2 Numerically Exploring the Tikhonov Estimator

Now that the solution for $\boldsymbol{\beta}$ is known, we can start with some numerical examples. We begin by sampling 120 equispaced points from the curve $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ and adding Gaussian noise to the function outputs. Half of the points are randomly selected as training data to solve for $\boldsymbol{\beta}$ while the other half are reserved for validation.

We can compare the results of the Tikhonov Estimator (with the \mathbf{D} matrix from 5) with a small $\lambda > 0$, a larger $\lambda > 0$, and with the Ordinary Least Squares Estimator ($\lambda = 0$) in Figure 7.

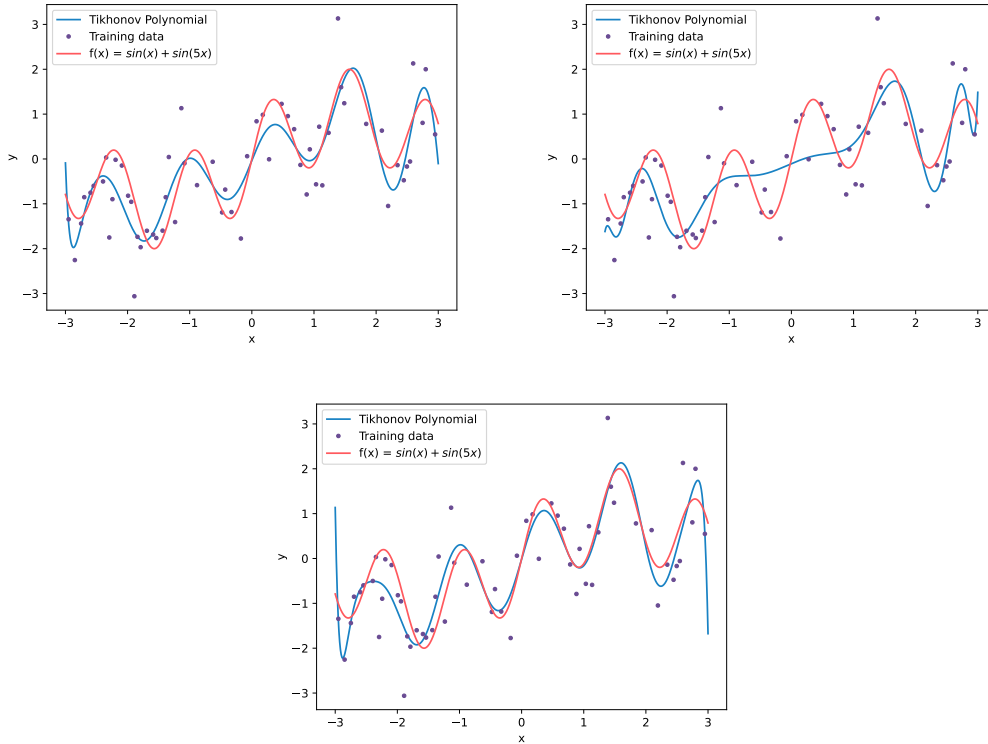


Figure 7: 15th degree Tikhonov fit to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$ (top left), $\lambda = 1$ (top right), and $\lambda = 0$ (bottom) for a single seed. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3, 3]$.

Certainly the Ordinary Least Squares solution has the largest derivative values out of all three plots in Figure 8 while the Tikhonov Estimator with $\lambda = 1$ dramatically shows a penalized derivative for $x \in [-1, 1]$.

We can also compare the RSS values for various λ values in Figure 8.

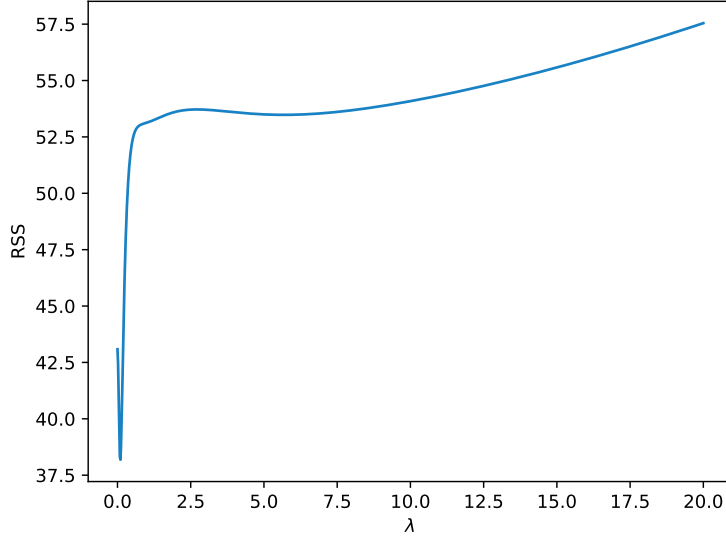


Figure 8: RSS from a 15th degree Tikhonov fit to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda \in [0, 20]$ for a single seed. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3, 3]$.

Figure 8 achieves a minimum error at $\lambda \approx 0.1001$. Conceptually, having some kind of bound on the derivative does make sense; there are several training points that are both larger than the largest function values and smaller than the smallest, so a least squares fit will create a model that exacerbates the oscillatory behavior. By punishing extreme oscillations we achieve a more realistic approximation.

3.2.1 A Discussion on Randomness and Accuracy

An important thing to note is that the accuracy of the estimators is *highly* dependent upon the random points sampled. Due to the randomness, we cannot guarantee that we are fitting to the function ‘fairly’ on the entire interval; the left side of an interval may have a higher concentration of training points and in turn the right side would have a higher concentration of validation points. This leads to a poor fit over the interval containing the validation data which leads to high error.

Although Tikhonov Estimation led to a better fit in the numerical example, the question of if it generally improves an estimate still remains.

We generated 100 different random distributions of training and validation points and found that a majority of the time, Tikhonov did lead to a better estimate.

	Optimal $\lambda = 0$	Optimal $\lambda > 0$
Number of Seeds	24	76

Table 5: The number of different seeds which had the minimum Residual Sum of Squares for Tikhonov as either $\lambda = 0$ or $\lambda > 0$ for standard noisy evaluations of $y = \sin x + \sin 5x$ on $x \in [-3, 3]$. 120 equispaced samples were randomly divided into the training and validation data.

From Table 5, we see that more often than not the Tikhonov Estimator outperformed Ordinary Least Squares in reducing the RSS . So, despite the unpredictable behavior between random samples, regularization was beneficial.

This is further visualized in Figure 9.

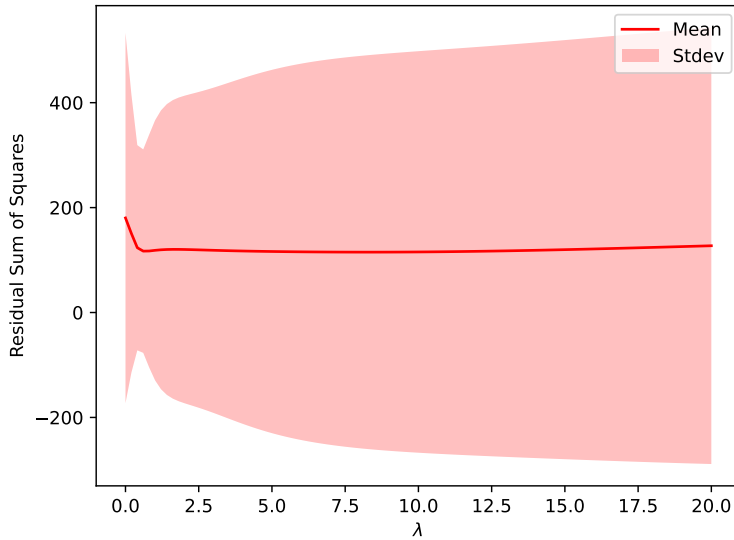


Figure 9: Mean and standard deviation of RSS from a 15th degree Tikhonov fit to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda \in [0, 20]$ for 100 unique seeds. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3, 3]$.

The first thing to note is that the average error is lower for all Tikhonov ($\lambda > 0$) than it is for Ordinary Least Squares ($\lambda = 0$), the second is to see that the standard deviation is very large across all λ values — this shows how dramatically different random samples can influence the results.

Another notable feature of the Tikhonov Estimator for this function was how prone to overfitting the method is. Although a degree 15 polynomial worked moderately well in the first example this is not the typical behavior. If we create a similar graph to Figure 9 but vary the polynomial degree instead of λ we can see a trend in Figure 10.

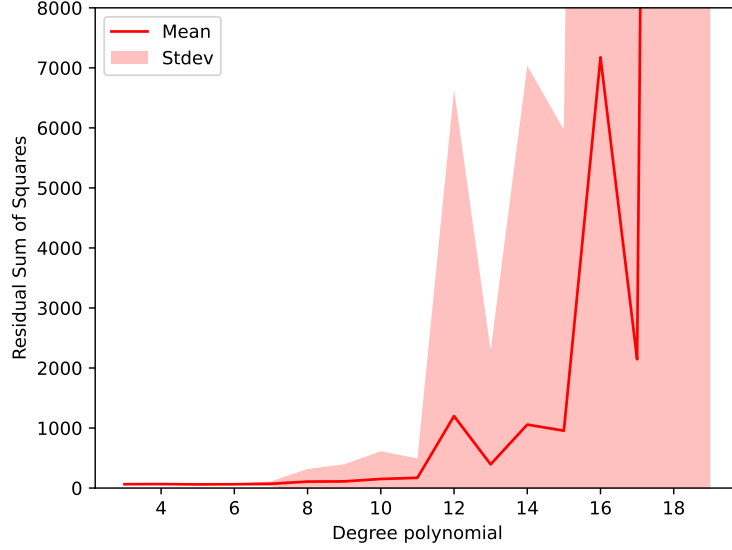


Figure 10: Mean and standard deviation of RSS from different degree 3 to degree 20 polynomial Tikhonov fits to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3, 3]$.

Note that the average error actually increases with polynomial degree. This is a product of high order polynomials overfitting to the training data and is visualized in the Figure 11.

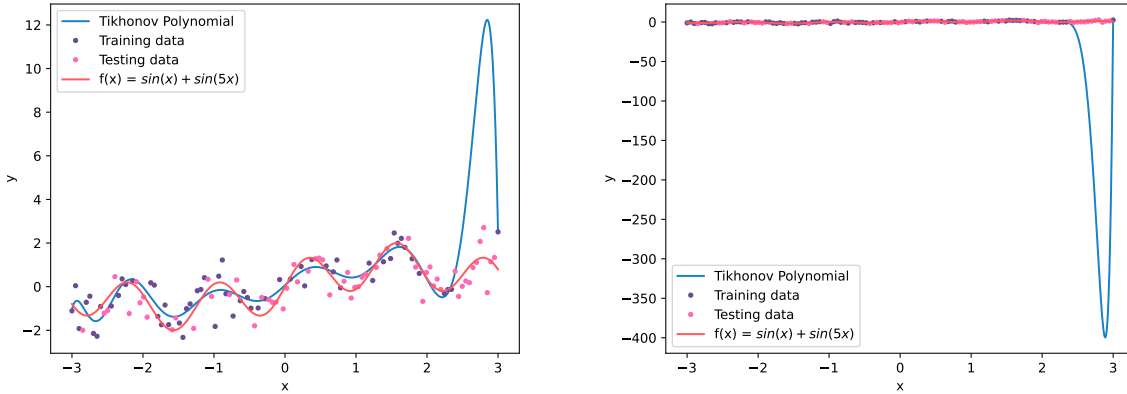


Figure 11: 15 degree (left) and 19 degree (right) Tikhonov fits to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$ for a single seed. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3, 3]$.

Looking closely at the left plot of Figure 11 one can notice that there is a large concentration of validation data on the right end of the interval. The results is a poor fit in that

region and extremely large oscillations as polynomial order is increased - the right plot of Figure 11.

This wasn't really an issue per say, just notable behavior of the estimator. This trend could be avoided by finding a way to evenly distribute the training and validation data, but we avoided this to more accurately simulate randomly collected noisy data.

3.2.2 Other Finite Difference Methods

The weight matrix \mathbf{D} used the centered difference formula to estimate the derivative, and thus $\dim \mathbf{D} = (n - 2) \times n$ since information about the derivatives at the endpoints is lost.

One can instead use the forward or backward differences to create a matrix that loses the derivative information at the left **or** right endpoint.

The formulas for forwards and backwards differences are:

$$\text{Forward Difference: } \left. \frac{df}{dt} \right|_{t_0} \approx \frac{f(t_0 + \Delta t) - f(t_0)}{\Delta t}$$

$$\text{Backward Difference: } \left. \frac{df}{dt} \right|_{t_0} \approx \frac{f(t_0) - f(t_0 - \Delta t)}{\Delta t}$$

Creating weight matrices with these formulas, we obtain the following results.

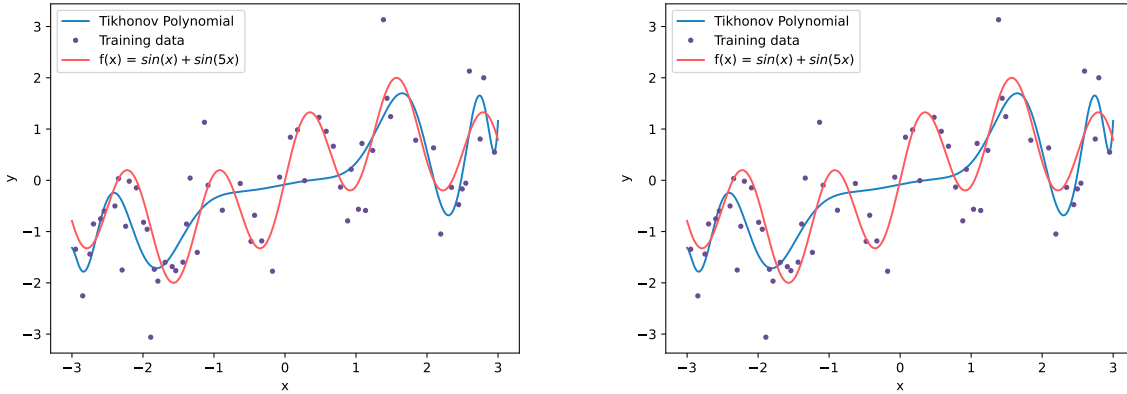


Figure 12: 15 degree Tikhonov fits to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$ for a single seed. The forward difference method was used in the left image and the backward difference method was used in the right image. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3, 3]$.

From Figure 12, the derivative appears to be penalized for both methods and they look *incredibly* similar, which makes sense since their weight matrices will only differ at the first and last rows of each matrix.

We can also extend it from penalizing the derivative of the solution to penalizing the second derivative. Constructing a weight matrix using a centered difference formula for higher order derivatives we get an interesting result, shown in Figure 13.

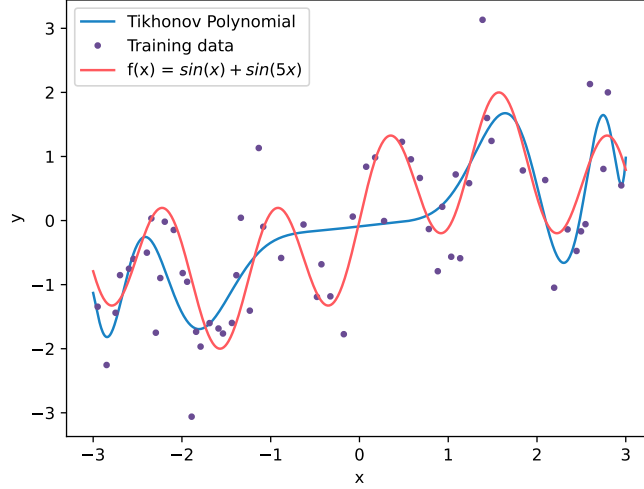


Figure 13: 15 degree Tikhonov fit to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$ for a single seed. The second order centered difference method was used to create the weight matrix. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3, 3]$.

Perhaps counter-intuitively, the result is very similar to the previous figures. But this may make some sense since when the second derivative penalized there will naturally be influence on the first derivative and vice versa.

4 LASSO Regression and Elastic Net

4.1 Introduction

Tikhonov is not the only regularization one can add to Ordinary Least Squares. In fact, there are an infinite amount of different regularizations that can be applied. However, for the independent portion of the project, we will focus on two other regularizations: LASSO and Elastic Net. LASSO Regularization with Ordinary Least Squares is formulated as the following:

$$\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1 \quad (6)$$

Lagrangian form of LASSO.

Note that this is almost identical to Ridge Regression, except the regularization term is using the 1-Norm instead of the 2-Norm. Furthermore, there is a generalization of both Ridge Regression and LASSO called Elastic Net:

$$\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \lambda \left(\frac{(1-\alpha)}{2} \|\boldsymbol{\beta}\|_2^2 + \alpha \|\boldsymbol{\beta}\|_1 \right) \quad (7)$$

Elastic Net Regularization.

This includes both the Ridge Regression and LASSO regularization terms. These regression methods are interesting because they result in different behavior in the optimized weights during the training process from both Ridge Regression and Tikhonov. We will explore both Elastic Net and LASSO (as a special case of Elastic Net) in the independent part of the project in the context of our previous exploration of Ridge Regression.

The benefit of LASSO Regression is that it has been shown to induce sparsity in the solution. This is incredibly valuable when modeling data where the underlying distribution comes from some polynomial but the model has more weights than the power of said polynomial - i.e. the solution has more coefficients than are necessary.

It follows that Elastic Net's strengths lie in its flexibility between penalizing the magnitude of a solution while inducing sparsity if necessary.

4.2 Choosing a Descent Algorithm for Elastic-Net

Unlike the other forms of regression that have been covered so far, there is no known analytical solution for Elastic Net or Lasso Regression. This dramatically changes how one should find the $\boldsymbol{\beta}$ that optimizes our minimization problem.

Note that the solution space is convex - which is proved thoroughly in the appendix (see 6.1.3) - which allows techniques like gradient descent to be used, however there's a better method for these forms of regression: *Coordinate Descent*.

Conceptually, Coordinate Descent can be thought of as fixing all but a single parameter in $\boldsymbol{\beta}$ and optimizing that parameter, then repeating for all the others.

More formally, Coordinate Descent can be implemented using the following algorithm:

Algorithm 1 Coordinate Descent

Require: n , number of times to perform the update for each parameter. \mathbf{X}, \mathbf{y} training data.
 d , the degree fit desired.

Ensure: $\boldsymbol{\beta}$, the optimized weights vector.

Standardize \mathbf{X}

$\boldsymbol{\beta} \leftarrow \mathbf{0}$

for $i \leftarrow 0$ to n **do**

for $j \leftarrow 1$ to d **do**

$\beta_j \leftarrow \frac{S\left(\frac{1}{|\mathbf{X}|} \sum_{k=1}^{|\mathbf{X}|} x_{kj} (y_k - \tilde{y}_k^{(j)}), \lambda\alpha\right)}{1 + \lambda(1-\alpha)}$

end for

end for

$\boldsymbol{\beta}_0 = \frac{1}{|\mathbf{X}|} \sum_{i=0}^{|\mathbf{X}|} \mathbf{y}_i - \mathbf{x}_i^T \boldsymbol{\beta}$

Algorithm 1: Coordinate Descent Algorithm [3]
(See 6.1.4 for a detailed derivation of the formula for $\boldsymbol{\beta}_0$.)

Notationally:

- by standardizing \mathbf{X} it is meant that the columns of \mathbf{X} are altered to have a mean of 0 and standard deviation of 1.
- $\tilde{y}_k^{(j)} = \sum_{l \neq j} x_{kl} \beta_l$
- $S(a, b) = \begin{cases} a - b & \text{if } a > 0 \text{ and } b < |a| \\ a + b & \text{if } a < 0 \text{ and } b < |a| \\ 0 & \text{if } b \geq |a| \end{cases}$
- $|\mathbf{X}|$ is the number of rows in \mathbf{X}
- \mathbf{x}_i is the i^{th} row of \mathbf{X}

Coordinate descent can achieve faster convergences in the context of Elastic Net and LASSO by orders of 10 or even 100 depending on the competing descent algorithm [5]

4.3 Numerically Exploring Elastic Net

This section explores solving the same problem from section 2.2. First, we will fit a degree one polynomial to evaluations from the line $y = 3x + 2$ with Gaussian noise on the interval $x \in [-5, 5]$. Again, we randomly split 20 equidistant points from the domain evenly into a training and validation set.

$\lambda \backslash \alpha$	0	0.5	1
0	5.571		
0.1	4.530	4.779	5.375

Table 6: Residual Sum of Squares for Elastic Net regression for various values of α and λ when fitting noisy function evaluations (Gaussian) of $y = 3x + 2$ to a one degree polynomial.

From Table 6, it's clear that for this particular set of noisy function evaluations and split for training and validation data, regularization helped improve the model fit. However, as was mentioned in a previous section on randomness (3.2.1), these results are stochastic and depend on the inherent randomness of the selected data and noise.

To alleviate some randomness in the results, we will average the results from 100 seeds for various combinations of α and λ . The results are displayed in Figure 14.

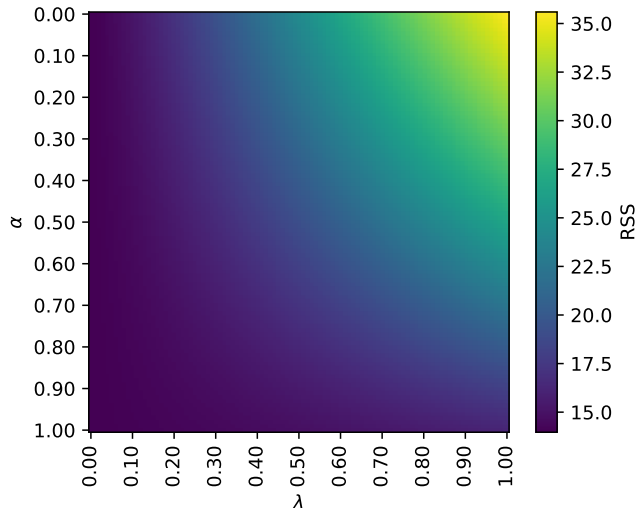


Figure 14: Elastic Net regression using 100 equispaced values of $\alpha \in \{0, 100\}$ and 100 equispaced values of $\lambda \in \{0, 100\}$ when fitting noisy function evaluations (Gaussian) of $y = 3x + 2$ to a one degree polynomial. Each combination of α and λ was evaluated for 100 different seeds and the RSS for each was averaged. The minimum average RSS was 13.968 for $\alpha = 0$ and $\lambda = 0$.

Figure 14 shows that Elastic-Net performs the worst when $\lambda = 1$ and $\alpha = 0$, which is equivalent to Ridge Regression. This contradicts the result of Section 2.2.1 where it was shown that, on average, this model benefits from Ridge Regression. There are several reasons that can explain this discrepancy. First, the Elastic Net solver used here standardizes the \mathbf{X} matrix before performing the minimization process, which does not occur for our Ridge Regression implementation. This changed the underlying problem being solved. Second, the intercept term was included in the regularization term from Equation 1. The intercept term is not regularized in our Elastic Net implementation. This too, changes the underlying problem being solved. So, we cannot expect Ridge Regression and Elastic Net to result in the same values.

We now fit a degree five polynomial to evaluations from the line $y = x^2$ with Gaussian noise on the interval $x \in [-5, 5]$. Again, we randomly split 20 equidistant points from the domain evenly into a training and validation set.

$\lambda \backslash \alpha$	0	0.5	1
0	61.8613		
0.1	62.627	61.713	60.139

Table 7: Residual Sum of Squares for Elastic Net regression for various values of α and λ when fitting noisy function evaluations (Gaussian) of $y = x^2$ to a five degree polynomial.

From Table 7, we see that there is at least one case where regularization appears to be useful in improving the model fit. This is seen by the smallest residual sum of squares being

when $\alpha = 1$ and $\lambda = 0.1$. However, this result is from a single seed and may not hold for other seeds. Performing the same averaging over 100 seeds as for the previous example, we obtain Figure 15.

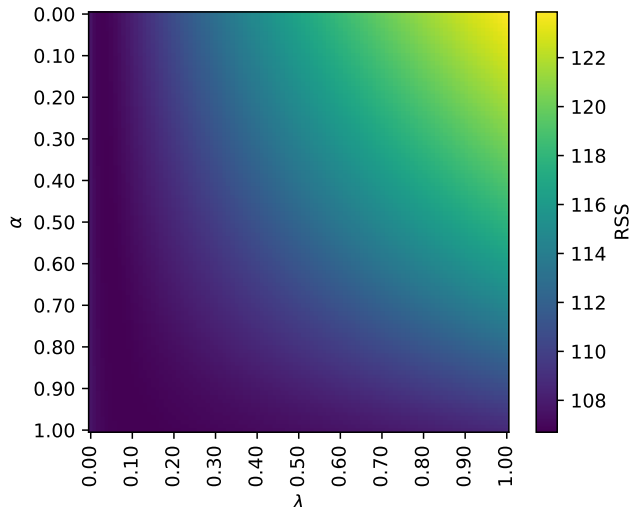


Figure 15: Elastic Net regression using 100 equispaced values of $\alpha \in \{0, 100\}$ and 100 equispaced values of $\lambda \in \{0, 100\}$ when fitting noisy function evaluations (Gaussian) of $y = x^2$ to a five degree polynomial. Each combination of α and λ was evaluated for 100 different seeds and the RSS for each was averaged. The minimum average RSS was 106.698 for $\alpha = 1.0$ and $\lambda = 0.12$.

Figure 15 shows that fitting a five degree polynomial to $y = x^2$ performs the worst when $\alpha = 0$ and $\lambda = 1$, which is pure Ridge Regression. We would expect Ridge Regression to perform poorly when there are many weights that need to be zeroed out, which is the case here. In fact, for the polynomial we're fitting ($y = b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5$), only the b_2 coefficient is non-zero in the true solution. Another natural explanation for this behavior is that we are dealing with noisy function evaluations, so the training data doesn't follow a parabola exactly.

The model performs the best when $\alpha = 1.0$ and $\lambda = 0.12$, which is pure LASSO. This makes sense since LASSO tends to zero out coefficients in the final polynomial.

5 Discussion/Conclusion

Rather than see this report as a search for a tried and true best regression technique, we thought of this more as an exploration of several different methods' strengths and weaknesses. There are applications that would simply require a bound on the magnitude of β but avoid sparsity, in which case Ridge Regression would be an ideal choice. Perhaps control over the magnitude of β is not necessary at all and you need to bound a unique characteristic of the solution - in that case creating an adequate weight matrix and using Tikhonov Regression would be more beneficial.

After thorough numerical exploration however, we can conclude that almost definitely for a given problem there is some Regularized Least Squares technique that will outperform Ordinary Least Squares, particularly when working with noisy data.

It can be shown that LASSO and Ridge Regression enforces assumptions about what type of distribution our data was drawn from - in fact, using Ridge Regression assumes that the data is drawn from a Gaussian Distribution, while LASSO assumes data is drawn from a Laplacian Distribution. Depending on the underlying distribution that the data is truly drawn from, the choice of model may have important effects on accuracy. This is a potential area of future research.

References

- [1] Amir Ali Ahmadi. *Convex and Conic Optimization*. Mar. 2016.
- [2] A.C. Bovik. *Handbook of Image and Video Processing*. Communications, Networking and Multimedia. Elsevier Science, 2010. ISBN: 9780080533612. URL: <https://books.google.com/books?id=OYFYt5C4N94C>.
- [3] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. “Regularization paths for generalized linear models via coordinate descent”. In: *Journal of statistical software* 33.1 (2010), p. 1.
- [4] Gene H Golub, Per Christian Hansen, and Dianne P O’Leary. “Tikhonov regularization and total least squares”. In: *SIAM journal on matrix analysis and applications* 21.1 (1999), pp. 185–194.
- [5] Trevor Hastie. *Fast Regularization Paths via Coordinate Descent*. 2009. URL: <https://hastie.su.domains/TALKS/glmnet.pdf>.

6 Appendix

6.1 Proofs

6.1.1 Deriving Ridge Regression

Below is a more rigorous proof of Section 2.1:

The equation for regularized Least Squares is:

$$\arg \min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \gamma \|\boldsymbol{\beta}\|_2^2 \quad (8)$$

Recalling that $\|\boldsymbol{\beta}\|_2^2 = \boldsymbol{\beta}^T \boldsymbol{\beta}$:

$$\begin{aligned} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \gamma \|\boldsymbol{\beta}\|_2^2 &= (\mathbf{X}\boldsymbol{\beta} - \mathbf{y})^T (\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ &= (\mathbf{X}\boldsymbol{\beta})^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ &= (\boldsymbol{\beta}^T \mathbf{X}^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ &= \boldsymbol{\beta}^T \mathbf{X}^T \mathbf{X} \boldsymbol{\beta} - \boldsymbol{\beta}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X} \boldsymbol{\beta} + \mathbf{y}^T \mathbf{y} + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \end{aligned}$$

Recall:

$$\mathbf{X} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^m \\ 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Where $\dim(\mathbf{X}) = (n+1) \times (m+1)$, $\dim(\boldsymbol{\beta}) = (m+1) \times (1)$, and $\dim(\mathbf{y}) = (n+1) \times (1)$.

Proof that $\boldsymbol{\beta}^T \mathbf{X}^T \mathbf{y} = \mathbf{y}^T \mathbf{X} \boldsymbol{\beta}$. Note first that $(\mathbf{y}^T \mathbf{X} \boldsymbol{\beta})^T = \boldsymbol{\beta}^T \mathbf{X}^T \mathbf{y}$. Further, $\dim(\boldsymbol{\beta}^T \mathbf{X}^T \mathbf{y}) = (1) \times (1) = \dim(\mathbf{y}^T \mathbf{X} \boldsymbol{\beta})$. Also, note that the transpose of a 1×1 matrix is the same matrix: $[c]^T = [c]$. It then follows that $\boldsymbol{\beta}^T \mathbf{X}^T \mathbf{y} = \mathbf{y}^T \mathbf{X} \boldsymbol{\beta}$, completing the proof.

So,

$$\begin{aligned} \boldsymbol{\beta}^T \mathbf{X}^T \mathbf{X} \boldsymbol{\beta} - \boldsymbol{\beta}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X} \boldsymbol{\beta} + \mathbf{y}^T \mathbf{y} + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} &= \boldsymbol{\beta}^T \mathbf{X}^T \mathbf{X} \boldsymbol{\beta} - 2\mathbf{y}^T \mathbf{X} \boldsymbol{\beta} + \mathbf{y}^T \mathbf{y} + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ &= (\mathbf{X}\boldsymbol{\beta})^T (\mathbf{X}\boldsymbol{\beta}) - 2\mathbf{y}^T \mathbf{X} \boldsymbol{\beta} + \mathbf{y}^T \mathbf{y} + \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \quad (9) \end{aligned}$$

The above matrices are then expanded:

$$\begin{aligned}
\mathbf{X}\boldsymbol{\beta} &= \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^m \\ 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \times \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \\
&= \begin{bmatrix} \beta_0 + \beta_1 x_0 + \beta_2 x_0^2 + \dots + \beta_m x_0^m \\ \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \dots + \beta_m x_1^m \\ \beta_0 + \beta_1 x_2 + \beta_2 x_2^2 + \dots + \beta_m x_2^m \\ \vdots \\ \beta_0 + \beta_1 x_n + \beta_2 x_n^2 + \dots + \beta_m x_n^m \end{bmatrix}
\end{aligned} \tag{10}$$

Then, $(\mathbf{X}\boldsymbol{\beta})^T \mathbf{X}\boldsymbol{\beta}$ is just a simple inner product (via the result from 10):

$$\begin{aligned}
(\mathbf{X}\boldsymbol{\beta})^T \mathbf{X}\boldsymbol{\beta} &= \begin{bmatrix} \beta_0 + \beta_1 x_0 + \beta_2 x_0^2 + \dots + \beta_m x_0^m \\ \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \dots + \beta_m x_1^m \\ \beta_0 + \beta_1 x_2 + \beta_2 x_2^2 + \dots + \beta_m x_2^m \\ \vdots \\ \beta_0 + \beta_1 x_n + \beta_2 x_n^2 + \dots + \beta_m x_n^m \end{bmatrix}^T \times \begin{bmatrix} \beta_0 + \beta_1 x_0 + \beta_2 x_0^2 + \dots + \beta_m x_0^m \\ \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \dots + \beta_m x_1^m \\ \beta_0 + \beta_1 x_2 + \beta_2 x_2^2 + \dots + \beta_m x_2^m \\ \vdots \\ \beta_0 + \beta_1 x_n + \beta_2 x_n^2 + \dots + \beta_m x_n^m \end{bmatrix} \\
&= (\beta_0 + \beta_1 x_0 + \dots + \beta_m x_0^m)^2 + (\beta_0 + \beta_1 x_1 + \dots + \beta_m x_1^m)^2 \\
&\quad + \dots + (\beta_0 + \beta_1 x_n + \dots + \beta_m x_n^m)^2
\end{aligned} \tag{11}$$

$$\begin{aligned}
2\mathbf{y}^T \mathbf{X}\boldsymbol{\beta} &= 2 \times \begin{bmatrix} y_0 & y_1 & \dots & y_n \end{bmatrix} \times \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^m \\ 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \times \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \\
&= 2 \times \begin{bmatrix} y_0 + y_1 + y_2 + \dots + y_n \\ y_0 x_0 + y_1 x_1 + y_2 x_2 + \dots + y_n x_n \\ y_0 x_0^2 + y_1 x_1^2 + y_2 x_2^2 + \dots + y_n x_n^2 \\ \vdots \\ y_0 x_0^m + y_1 x_1^m + y_2 x_2^m + \dots + y_n x_n^m \end{bmatrix}^T \times \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \\
&= 2(\beta_0(y_0 + y_1 + \dots + y_n) + \beta_1(y_0 x_0 + y_1 x_1 + \dots + y_n x_n) \\
&\quad + \dots + \beta_m(y_0 x_0^m + y_1 x_1^m + \dots + y_n x_n^m))
\end{aligned} \tag{12}$$

$$\gamma \boldsymbol{\beta}^T \boldsymbol{\beta} = \gamma \beta_0^2 + \gamma \beta_1^2 + \dots + \gamma \beta_m^2 \tag{13}$$

Now the derivatives of 11, 12, and 13 with respect to β are taken to get the derivative of 8 with respect to β . In effect, the result is a vector where the i -th element is the derivative of 8 with respect to β_i .

First, the derivatives of 11 are computed:

$$\begin{aligned}
\frac{d}{d\beta}(\mathbf{X}\beta)^T \mathbf{X}\beta &= \begin{bmatrix} \frac{d}{d\beta_0}(\mathbf{X}\beta)^T \mathbf{X}\beta \\ \frac{d}{d\beta_1}(\mathbf{X}\beta)^T \mathbf{X}\beta \\ \vdots \\ \frac{d}{d\beta_m}(\mathbf{X}\beta)^T \mathbf{X}\beta \end{bmatrix} \\
&= \begin{bmatrix} 2(\beta_0 + \beta_1 x_0 + \dots + \beta_m x_0^m) + \dots + 2(\beta_0 + \beta_1 x_n + \dots + \beta_m x_n^m) \\ 2x_0(\beta_0 + \beta_1 x_0 + \dots + \beta_m x_0^m) + \dots + 2x_n(\beta_0 + \beta_1 x_n + \dots + \beta_m x_n^m) \\ \vdots \\ 2x_0^m(\beta_0 + \beta_1 x_0 + \dots + \beta_m x_0^m) + \dots + 2x_n^m(\beta_0 + \beta_1 x_n + \dots + \beta_m x_n^m) \end{bmatrix} \\
&= 2 \times \begin{bmatrix} \beta_0 \sum_{i=0}^n 1 + \beta_1 \sum_{i=0}^n x_i + \beta_2 \sum_{i=0}^n x_i^2 + \dots + \beta_m \sum_{i=0}^n x_i^m \\ \beta_0 \sum_{i=0}^n x_i + \beta_1 \sum_{i=0}^n x_i^2 + \beta_2 \sum_{i=0}^n x_i^3 + \dots + \beta_m \sum_{i=0}^n x_i^{m+1} \\ \vdots \\ \beta_0 \sum_{i=0}^n x_i^m + \beta_1 \sum_{i=0}^n x_i^{m+1} + \beta_2 \sum_{i=0}^n x_i^{m+2} + \dots + \beta_m \sum_{i=0}^n x_i^{2m} \end{bmatrix} \\
&= 2 \times \begin{bmatrix} \sum_{i=0}^n 1 & \sum_{i=0}^n x_i & \sum_{i=0}^n x_i^2 & \dots & \sum_{i=0}^n x_i^m \\ \sum_{i=0}^n x_i & \sum_{i=0}^n x_i^2 & \sum_{i=0}^n x_i^3 & \dots & \sum_{i=0}^n x_i^{m+1} \\ \sum_{i=0}^n x_i^2 & \sum_{i=0}^n x_i^3 & \sum_{i=0}^n x_i^4 & \dots & \sum_{i=0}^n x_i^{m+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum_{i=0}^n x_i^m & \sum_{i=0}^n x_i^{m+1} & \sum_{i=0}^n x_i^{m+2} & \dots & \sum_{i=0}^n x_i^{2m} \end{bmatrix} \times \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} \\
&= 2 \times \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ x_0 & x_1 & x_2 & \dots & x_n \\ x_0^2 & x_1^2 & x_2^2 & \dots & x_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & x_2^m & \dots & x_n^m \end{bmatrix} \times \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^m \\ 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \times \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} \\
&= 2\mathbf{X}^T \mathbf{X}\beta
\end{aligned} \tag{14}$$

Secondly, the derivatives of 12 are computed:

$$\begin{aligned}
\frac{d}{d\boldsymbol{\beta}} 2\mathbf{y}^T \mathbf{X} \boldsymbol{\beta} &= \begin{bmatrix} \frac{d}{da_0} 2\mathbf{y}^T \mathbf{X} \boldsymbol{\beta} \\ \frac{d}{da_1} 2\mathbf{y}^T \mathbf{X} \boldsymbol{\beta} \\ \vdots \\ \frac{d}{da_m} 2\mathbf{y}^T \mathbf{X} \boldsymbol{\beta} \end{bmatrix} \\
&= 2 \times \begin{bmatrix} y_0 + y_1 + \dots + y_n \\ y_0 x_0 + y_1 x_1 + \dots + y_n x_n \\ y_0 x_0^2 + y_1 x_1^2 + \dots + y_n x_n^2 \\ \vdots \\ y_0 x_0^m + y_1 x_1^m + \dots + y_n x_n^m \end{bmatrix} \\
&= 2 \times \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ x_0 & x_1 & x_2 & \dots & x_n \\ x_0^2 & x_1^2 & x_2^2 & \dots & x_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & x_2^m & \dots & x_n^m \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \\
&= 2\mathbf{X}^T \mathbf{y}
\end{aligned} \tag{15}$$

Lastly, the derivatives of 13 are computed:

$$\begin{aligned}
\frac{d}{d\boldsymbol{\beta}} \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} &= \begin{bmatrix} \frac{d}{d\beta_0} \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ \frac{d}{d\beta_1} \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \\ \vdots \\ \frac{d}{d\beta_m} \gamma \boldsymbol{\beta}^T \boldsymbol{\beta} \end{bmatrix} \\
&= \gamma \times \begin{bmatrix} 2\beta_0 \\ 2\beta_1 \\ 2\beta_2 \\ \vdots \\ 2\beta_m \end{bmatrix} \\
&= 2 \times \gamma \times \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \times \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} \\
&= 2\gamma \mathbf{I} \boldsymbol{\beta}
\end{aligned} \tag{16}$$

Now, recall 9 and note that to find the minimum of the Least Squares equation given by 8 one needs to find where the derivative of 9 is equal to zero:

$$\begin{aligned}
\frac{d}{d\beta}((\mathbf{X}\beta)^T(\mathbf{X}\beta) - 2\mathbf{y}^T\mathbf{X}\beta + \mathbf{y}^T\mathbf{y} + \gamma\beta^T\beta) &= 0 \\
2\mathbf{X}^T\mathbf{X}\beta - 2\mathbf{X}^T\mathbf{y} + 2\gamma\mathbf{I}\beta &= 0 \\
\mathbf{X}^T\mathbf{X}\beta - \mathbf{X}^T\mathbf{y} + \gamma\mathbf{I}\beta &= 0 \\
\mathbf{X}^T\mathbf{X}\beta + \gamma\mathbf{I}\beta - \mathbf{X}^T\mathbf{y} &= 0 \\
(\mathbf{X}^T\mathbf{X} + \gamma\mathbf{I})\beta - \mathbf{X}^T\mathbf{y} &= 0 \\
(\mathbf{X}^T\mathbf{X} + \gamma\mathbf{I})\beta &= \mathbf{X}^T\mathbf{y} \\
\beta &= (\mathbf{X}^T\mathbf{X} + \gamma\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}
\end{aligned} \tag{17}$$

Thus, the equation for Ridge Regression is $\beta = (\mathbf{X}^T\mathbf{X} + \gamma\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$. This completes the proof.

6.1.2 Deriving the Tikhonov Estimator

$$\begin{aligned}
\mathbf{D}\beta &= \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{2} & 0 & \dots & 0 \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} \\
&= \frac{1}{2} \begin{bmatrix} \beta_2 - \beta_0 \\ \beta_3 - \beta_1 \\ \beta_4 - \beta_2 \\ \vdots \\ \beta_n - \beta_{n-2} \end{bmatrix}
\end{aligned}$$

So it follows that

$$(\mathbf{D}\beta)^T(\mathbf{D}\beta) = \begin{bmatrix} \frac{\beta_2 - \beta_0}{2} \\ \frac{\beta_3 - \beta_1}{2} \\ \frac{\beta_4 - \beta_2}{2} \\ \vdots \\ \frac{\beta_n - \beta_{n-2}}{2} \end{bmatrix}^T \begin{bmatrix} \frac{\beta_2 - \beta_0}{2} \\ \frac{\beta_3 - \beta_1}{2} \\ \frac{\beta_4 - \beta_2}{2} \\ \vdots \\ \frac{\beta_n - \beta_{n-2}}{2} \end{bmatrix}$$

$$= \left(\frac{(\beta_2 - \beta_0)^2}{4} + \frac{(\beta_3 - \beta_1)^2}{4} + \dots + \frac{(\beta_n - \beta_{n-2})^2}{4} \right)$$

So it follows that

$$\frac{d}{d\boldsymbol{\beta}} [(\mathbf{D}\boldsymbol{\beta})^T(\mathbf{D}\boldsymbol{\beta})] = \begin{bmatrix} -\frac{\beta_2-\beta_0}{2} \\ -\frac{\beta_3-\beta_2}{2} \\ \frac{\beta_2-\beta_0}{2} - \frac{\beta_4-\beta_2}{2} \\ \vdots \\ \frac{\beta_{n-2}-\beta_{n-4}}{2} - \frac{\beta_n-\beta_{n-2}}{2} \\ \frac{\beta_{n-1}-\beta_{n-3}}{2} \\ \frac{\beta_n-\beta_{n-2}}{2} \end{bmatrix}$$

This initially doesn't seem like anything useful, but note that

$$\mathbf{D}^T \mathbf{D} \boldsymbol{\beta} = \begin{bmatrix} -\frac{1}{2} & 0 & 0 & \dots & 0 \\ 0 & -\frac{1}{2} & 0 & \dots & 0 \\ \frac{1}{2} & 0 & \ddots & \dots & 0 \\ 0 & \frac{1}{2} & 0 & -\frac{1}{2} & 0 \\ \vdots & 0 & \ddots & 0 & -\frac{1}{2} \\ 0 & \dots & 0 & \frac{1}{2} & 0 \\ 0 & \dots & 0 & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \frac{\beta_2-\beta_0}{2} \\ \frac{\beta_3-\beta_1}{2} \\ \vdots \\ \frac{\beta_n-\beta_{n-2}}{2} \end{bmatrix}$$

$$= \begin{bmatrix} -\frac{\beta_2-\beta_0}{4} \\ -\frac{\beta_3-\beta_1}{4} \\ \frac{\beta_2-\beta_0}{4} - \frac{\beta_4-\beta_2}{4} \\ \vdots \\ \frac{\beta_{n-2}-\beta_{n-4}}{4} - \frac{\beta_n-\beta_{n-2}}{4} \\ \frac{\beta_{n-1}-\beta_{n-3}}{4} \\ \frac{\beta_n-\beta_{n-2}}{4} \end{bmatrix}$$

So it follows that

$$\frac{d}{d\boldsymbol{\beta}} [(\mathbf{D}\boldsymbol{\beta})^T(\mathbf{D}\boldsymbol{\beta})] = 2\mathbf{D}^T \mathbf{D} \boldsymbol{\beta}$$

So,

$$\frac{d}{d\boldsymbol{\beta}} [(\mathbf{D}\boldsymbol{\beta})^T(\mathbf{D}\boldsymbol{\beta})] = 2\mathbf{D}^T\mathbf{D}\boldsymbol{\beta}$$

6.1.3 Proving the Elastic Net Solution Space is Convex

It will be shown that

$$\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \lambda \left(\frac{(1-\alpha)}{2} \|\boldsymbol{\beta}\|_2^2 + \alpha \|\boldsymbol{\beta}\|_1 \right)$$

Has a convex solution space with respect to $\boldsymbol{\beta}$.

First, note that an affine function can be defined as any function $f(x)$ where

$$f(\lambda y + (1-\lambda)z) = \lambda f(z) + (1-\lambda)f(z)$$

If we let $f(\boldsymbol{\beta}) = \mathbf{X}\boldsymbol{\beta} - \mathbf{y}$ then

$$\begin{aligned} f(\lambda\boldsymbol{\beta}_1 + (1-\lambda)\boldsymbol{\beta}_2) &= \mathbf{X}(\lambda\boldsymbol{\beta}_1 + (1-\lambda)\boldsymbol{\beta}_2) - \mathbf{y} \\ &= \lambda\mathbf{X}\boldsymbol{\beta}_1 + (1-\lambda)\mathbf{X}\boldsymbol{\beta}_2 - (\lambda\mathbf{y} + (1-\lambda)\mathbf{y}) \\ &= \lambda(\mathbf{X}\boldsymbol{\beta}_1 - \mathbf{y}) + (1-\lambda)(\mathbf{X}\boldsymbol{\beta}_2 - \mathbf{y}) \\ &= \lambda f(\boldsymbol{\beta}_1) + (1-\lambda)f(\boldsymbol{\beta}_2) \end{aligned}$$

Thus $\mathbf{X}\boldsymbol{\beta} - \mathbf{y}$ is affine.

Theorem 1: Let f be affine and g be convex, then $g \circ f$ is also convex.

Proof:

Let x, y be fixed but arbitrary. Let $\lambda \in [0, 1]$ be fixed but arbitrary. Then,

$$\begin{aligned} (g \circ f)(\lambda x + (1-\lambda)y) &= g(f(\lambda x + (1-\lambda)y)) \\ &= g(\lambda f(x) + (1-\lambda)f(y)) \\ &\leq \lambda g(f(x)) + (1-\lambda)g(f(y)) \\ &= \lambda(g \circ f)(x) + (1-\lambda)(g \circ f)(y) \end{aligned} \tag{18}$$

Theorem 2: If two functions f and g are convex, then $f + g$ is convex.

Proof: Let x, y be fixed but arbitrary and let $t \in [0, 1]$ be fixed. Then since f and g are convex:

$$\begin{aligned} f(\lambda x + (1-\lambda)y) + g(\lambda x + (1-\lambda)y) &\leq \lambda f(x) + (1-\lambda)f(y) + \lambda g(x) + (1-\lambda)g(y) \\ &= \lambda(f + g)(x) + (1-\lambda)(f + g)(y) \end{aligned} \tag{19}$$

Thus $f + g$ is convex

Let f and g be convex functions. Then $f + g$ is convex.

Theorem 3: If $f(x)$ is a norm, then $f(x)$ is convex

Proof: Let x, y be fixed but arbitrary and let $t \in [0, 1]$ be fixed. Then since f and g are convex. [1]

$$f(\lambda x + (1 - \lambda)y) \leq f(\lambda x) + f((1 - \lambda)y) = \lambda f(x) + (1 - \lambda)f(y) \quad (20)$$

From Theorems 1 and 3, it follows that $\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2$ is convex.

From Theorem 3 it follows that $\|\boldsymbol{\beta}\|_2^2$ and $\|\boldsymbol{\beta}\|_1$ are both convex.

Finally, from Theorem 2, it must be true that

$$\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \lambda \left(\frac{(1 - \alpha)}{2} \|\boldsymbol{\beta}\|_2^2 + \alpha \|\boldsymbol{\beta}\|_1 \right)$$

is also convex.

6.1.4 Deriving the intercept term for Coordinate Descent

Theorem 4: The intercept term for Coordinate Descent is given by $\beta_0 = \frac{1}{N} \sum_{i=1}^N (y_i - x_i^T \boldsymbol{\beta})$

Proof:

$$\begin{aligned} \frac{d}{d\beta_0} \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \boldsymbol{\beta})^2 + \lambda P_\alpha(\boldsymbol{\beta}) &= \frac{d}{d\beta_0} \frac{-2}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \boldsymbol{\beta}) \\ \frac{1}{N} \sum_{i=1}^N (\beta_0) &= \frac{1}{N} \sum_{i=1}^N (y_i - x_i^T \boldsymbol{\beta}) \\ \beta_0 &= \frac{1}{N} \sum_{i=1}^N (y_i - x_i^T \boldsymbol{\beta}) \end{aligned} \quad (21)$$

6.2 Code

6.2.1 finite_diff.py

```

1 from elastic_net import ElasticNet
2 import matplotlib.pyplot as plt
3 import numpy as np
4 from tabulate import tabulate
5
6 class ElasticNetHelper:
7     def __init__(self, f, degree, x_min, x_max, num_evals_x, num_train_x,
8         noise = 0, verbose = False):
9         '''
10         ' __init__ '
11
12         Initialize the plotting helper for the elastic net solver '
13         ElasticNet '

```

```

12     These parameters are meant to be constant across all generated
    elastic net solvers
13
14     Parameters
15
16     'f': Function that we're fitting to
17     'degree': Degree fit we want
18     'x_min', 'x_max': Domain that we're fitting to
19     'num_evals_x': Number of equispaced points in the domain to use
    for training and validation data
20     'num_train_x': Number of randomly sampled points from 'num_evals_x'
    ' to use for training, rest is for validation
21     'noise': Noise for each generated elastic net solver
22     'verbose': Optionally turn on (True) or off (False) extra prints (
    Defaults to 'False')
23
24     Returns
25
26     Nothing
27     '''
28
29     # Initialize member variables
30     self.f = f
31     self.degree = degree
32     self.x_min = x_min
33     self.x_max = x_max
34     self.num_evals_x = num_evals_x
35     self.num_train_x = num_train_x
36     self.noise = noise
37     self.verbose = verbose
38
39     # Create our data
40     self.x_eval = np.linspace(x_min, x_max, num_evals_x)
41
42     def new_en_solver(self, alpha, _lambda, seed):
43         '''
44         'new_en_solver'
45
46         Creates a new elastic net solve using the given seed
47
48         Parameters
49
50         'alpha', '_lambda': Hyperparameters for 'ElasticNet', see '
    elastic_net.py' for more details
51         'seed': Numpy seed
52
53         Returns
54
55         Nothing, but sets the current elastic net to this en
56         '''
57
58         # Set numpy seed
59         np.random.seed(seed)
60

```

```

61     # Create our splits
62     self.x_train, self.x_val = self.get_split(self.x_eval, self.
num_train_x)
63     self.y_train = (self.f(self.x_train) + np.random.normal(scale=self
.noise, size=self.x_train.shape))
64     self.y_val = (self.f(self.x_val) + np.random.normal(scale=self.
noise, size=self.x_val.shape))
65
66     if self.verbose:
67         print(f"x_train {self.x_train.shape}, x_val {self.x_val.shape
}, num_evals_x {self.num_evals_x}")
68         print(f"y_train {self.y_train.shape}, y_val {self.y_val.shape
}, num_evals_x {self.num_evals_x}")
69
70     # Create the elastic net solve and set it to the current one
71     en = ElasticNet(self.x_train, self.y_train, self.degree, alpha,
_lambda, verbose=self.verbose)
72     self.en = en
73
74     def get_split(self, x_eval, num_train_x):
75         '''
76         'get_split'
77
78         Given the total data get the training and validation split
79
80         Parameters
81
82         x_eval: Total x-values
83         num_train_x: Number of x-values in the training split
84
85         Returns
86
87         x_train: Training split
88         x_val: Validation split
89         '''
90
91     # Create training split
92     x_train = np.random.choice(x_eval, num_train_x, replace=False)
93     x_train = np.sort(x_train)
94
95     # Create validation split
96     train_idx = 0
97     x_val = list()
98     for x in x_eval:
99         if train_idx < num_train_x and x_train[train_idx] == x:
100             train_idx += 1
101         else:
102             x_val += [x]
103     x_val = np.array(x_val)
104
105     return x_train, x_val
106
107     def train(self, j):
108         '''

```



```

109         'train'
110
111         Train the elastic net model j times over each variable
112
113         Parameters
114
115         j: Number of times to iterate over each variable for trainin
116
117         Return Nothing
118         '''
119
120         self.en.iterate_coord_descent(j)
121
122     def get_RSS(self, data):
123         '''
124         'get_RSS'
125
126         Get current elastic net RSS
127
128         Parameters
129
130         data: Either 'val' or 'train' for validation or training data
131         respectively
132
133         Returns RSS
134         '''
135
136         if data == 'val':
137             return self.en.get_RSS(self.x_val, self.y_val)
138         elif data == 'train':
139             return self.en.get_RSS(self.x_train, self.y_train)
140         else:
141             raise Exception("get_RSS: 'data' must be either 'val' or 'train', but is '{data}'")
142
143     def get_elastic_net(self, data):
144         '''
145         'get_elastic_net'
146
147         Get current elastic net formula that's being minimized
148
149         Parameters
150
151         data: Either 'val' or 'train' for validation or training data
152         respectively
153
154         Returns elastic net formula
155         '''
156
157         if data == 'val':
158             return self.en.get_elastic_net(self.x_val, self.y_val)
159         elif data == 'train':
160             return self.en.get_elastic_net(self.x_train, self.y_train)
161         else:

```

```

160         raise Exception("get_elastic_net: 'data' must be either \'val
161         \' or \'train\', but is \'{data}\'")
162
163     def get_weights(self):
164         """
165         'get_weights'
166
167         Returns current elastic net weights
168         """
169         return self.en.get_b()
170
171     def make_plot(self, num_true_x = 100, title = 'title', x_axis = 'x',
172     y_axis = 'y', img_name = 'PLOT.pdf', img_dir = '../Images', train_data
173     = True, val_data = True, f_plot = True, predict_f_plot = True):
174         """
175         'make_plot'
176
177         Makes plot with (optional) training data and (optional) validation
178         data along with (optional) the true function and (optional) the
179         predicted function
180
181         Returns nothing, but makes our plot
182         """
183
184         # Plotting colors
185         color1 = '#FF595E'
186         color2 = '#1982C4'
187         color3 = '#6A4C93'
188         color4 = 'green'
189
190         # Construct save path
191         save_path = f"{img_dir}/{img_name}"
192
193         # Create helpful data
194         true_x = np.linspace(self.x_min, self.x_max, num_true_x)
195         true_y = self.f(true_x)
196
197         # Create initial plots
198         fig, ax = plt.subplots(1,1,figsize=(5,4), dpi=120, facecolor='
199         white', tight_layout={'pad': 1})
200
201         general_marker_style = dict(markersize = 1, markeredgewidth=0)
202         dot_marker_style = dict(markersize = 4, markeredgewidth=0.75),
203         marker='*', markeredgewidth=0.75)
204         data_marker_size = 2
205         scatter_marker_size = 10
206
207         # Create original function plot
208         if f_plot:
209             ax.plot(true_x, true_y, color=color1, label="Original function
210             ", **general_marker_style)
211
212         # Create predicted function plot

```

```

205         if predict_f_plot:
206             pred_y = self.en.get_prediction(true_x)
207             ax.plot(true_x, pred_y, color=color2, label="Elastic Net
function", **general_marker_style)
208
209         # Plot training data
210         if train_data:
211             ax.scatter(self.x_train, self.y_train, s=scatter_marker_size,
color=color3, label=f"Training data")
212
213         # Plot validation data
214         if val_data:
215             ax.scatter(self.x_val, self.y_val, s=scatter_marker_size,
color=color4, label=f"Validation data")
216
217         # Add labels
218         if x_axis is not None:
219             ax.set_xlabel(f"{x_axis}")
220         if y_axis is not None:
221             ax.set_ylabel(f"{y_axis}")
222         if title is not None:
223             ax.set_title(f"{title}")
224         ax.legend()
225
226         # Save image
227         plt.savefig(f'{save_path}')
228         plt.close()
229         print(f'Saved to: {save_path}')
230
231     def print_params(self):
232         '''
233         'print_params'
234
235         Make a nice table of all the parameters of the class
236         '''
237
238         dict_tmp = dict()
239         dict_tmp["Option"] = [
240             "self.degree",
241             "self.x_min",
242             "self.x_max",
243             "self.num_evals_x",
244             "self.num_train_x",
245             "self.noise",
246             "self.verbose"
247         ]
248         dict_tmp["Description"] = [
249             self.degree,
250             self.x_min,
251             self.x_max,
252             self.num_evals_x,
253             self.num_train_x,
254             self.noise,
255             self.verbose

```

```

256     ]
257     print(tabulate(dict_tmp, headers="keys", tablefmt="pretty"))

```

6.2.2 finite_diff.py

```

1 from elastic_net_helper import ElasticNetHelper
2 from prints import *
3
4 f = lambda x: 3*x+2
5
6 for alpha, _lambda in [(0, 0.1), (1, 0.1), (0.5, 0.1), (0,0)]:
7
8     small_banner(f"3x+2, alpha {alpha}, lambda {_lambda}", False, True)
9
10    enp = ElasticNetHelper(f = f , degree = 1, x_min = -5, x_max = 5,\
11                          num_evals_x = 20, num_train_x = 10,\
12                          noise = 1, verbose = False)
13
14    enp.print_params()
15    print()
16
17    enp.new_en_solver(alpha = alpha, _lambda = _lambda, seed = 0)
18
19    print(f"Before training RSS {enp.get_RSS('train')}")
20    print(f"Before training EN {enp.get_elastic_net('train')}")
21    print(f"Before validation RSS {enp.get_RSS('val')}")
22    print(f"Before validation EN {enp.get_elastic_net('val')}")
23    print(f"Before weights")
24    print(enp.get_weights())
25    print()
26
27    enp.train(10)
28    enp.make_plot(num_true_x = 100, title = None, img_name=f"3x+2_a={alpha}
29    _l={_lambda}.pdf")
30
31    print(f"After training RSS {enp.get_RSS('train')}")
32    print(f"After training EN {enp.get_elastic_net('train')}")
33    print(f"After validation RSS {enp.get_RSS('val')}")
34    print(f"After validation EN {enp.get_elastic_net('val')}")
35    print(f"After weights")
36    print(enp.get_weights())
37    print()

```

6.2.3 finite_diff.py

```

1 from elastic_net_helper import ElasticNetHelper
2 from prints import *
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 f = lambda x: 3*x+2
7
8 save_name = "../Images/3x+2_matrix.pdf"
9
10 # Store RSS for each alpha and lambda

```

```

11 num_a = 101
12 num_l = 101
13 num_seeds = 100
14
15 alphas = np.linspace(0,1,num_a)
16 _lambdas = np.linspace(0,1,num_l)
17 RSS_mat = np.zeros((num_a,num_l))
18
19 # Get RSS values, keep track of minimum
20 min_RSS, min_alpha, min_lambda, min_weights = np.inf, -1, -1, None
21
22 small_banner(f"3x+2 matrix plot for {num_seeds} seeds", False, True)
23
24 for alpha_iter in range(len(alphas)):
25     for _lambda_iter in range(len(_lambdas)):
26         enp = ElasticNetHelper(f = f , degree = 1, x_min = -5, x_max = 5,\
27                               num_evals_x = 20, num_train_x = 10,\
28                               noise = 1, verbose = False)
29
30         if alpha_iter == 0 and _lambda_iter == 0:
31             enp.print_params()
32
33         alpha = alphas[alpha_iter]
34         _lambda = _lambdas[_lambda_iter]
35
36         # Count all RSS for this seed
37         RSS_sum = 0
38
39         for seed in np.arange(0,num_seeds,1):
40             enp.new_en_solver(alpha = alpha, _lambda = _lambda, seed =
seed.item())
41             enp.train(10)
42             RSS_sum += enp.get_RSS('val')
43
44         this_RSS = RSS_sum/num_seeds
45         RSS_mat[alpha_iter][_lambda_iter] = this_RSS
46
47         if this_RSS < min_RSS:
48             min_RSS = this_RSS
49             min_alpha = alpha
50             min_lambda = _lambda
51             min_weights = enp.get_weights()
52
53 # Print minima
54 print(f"Minimum RSS is {min_RSS} with alpha {min_alpha} and lambda {
min_lambda}")
55 print(f"Minimum weights were:")
56 print(min_weights)
57 print()
58
59 # Make plot showing the result
60
61 fig, ax = plt.subplots(1,1,figsize=(5,4), dpi=120, facecolor='white',
tight_layout={'pad': 1})

```

```

62
63 # Convert alphas to printable version
64 alpha_label_locs= np.arange(0,num_a,10)
65 _lambda_label_locs= np.arange(0,num_l,10)
66
67 # Make sure last element is there
68 if num_a-1 not in alpha_label_locs:
69     alpha_label_locs = np.append(alpha_label_locs, [num_a-1])
70
71 if num_l-1 not in _lambda_label_locs:
72     _lambda_label_locs = np.append(_lambda_label_locs, [num_l-1])
73
74 alpha_labels= [f'{1.0 if a >= len(alphas) else alphas[a]:0.2f}' for a in
    alpha_label_locs]
75 _lambda_labels= [f'{1.0 if l >= len(_lambdas) else _lambdas[l]:0.2f}' for
    l in _lambda_label_locs]
76
77 a1 = ax.matshow(RSS_mat)
78 plt.colorbar(a1, label="RSS")
79 ax.set_xlabel("$\lambda$")
80 ax.set_ylabel("$\alpha$")
81 ax.set_yticks(alpha_label_locs, alpha_labels, rotation=00)
82 ax.set_xticks(_lambda_label_locs, _lambda_labels, rotation=90)
83 ax.tick_params(labelbottom = True)
84 ax.tick_params(labeltop = False)
85 ax.tick_params(top = False)
86 #plt.show()
87 plt.savefig(save_name)
88 print(f"Saved figure to {save_name}")

```

6.2.4 finite_diff.py

```

1 from elastic_net_helper import ElasticNetHelper
2 from prints import *
3
4 f = lambda x: x**2
5
6 for alpha, _lambda in [(0, 0.1), (1, 0.1), (0.5, 0.1), (0,0)]:
7
8     small_banner(f"x^2, alpha {alpha}, lambda {_lambda}", False, True)
9
10    enp = ElasticNetHelper(f = f , degree = 5, x_min = -5, x_max = 5,\
11        num_evals_x = 20, num_train_x = 10,\
12        noise = 1, verbose = False)
13
14    enp.print_params()
15    print()
16
17    enp.new_en_solver(alpha = alpha, _lambda = _lambda, seed = 0)
18
19    print(f"Before training RSS {enp.get_RSS('train')}")
20    print(f"Before training EN {enp.get_elastic_net('train')}")
21    print(f"Before validation RSS {enp.get_RSS('val')}")
22    print(f"Before validation EN {enp.get_elastic_net('val')}")
23    print(f"Before weights")

```

```

23     print(enp.get_weights())
24     print()
25
26     enp.train(10)
27     enp.make_plot(num_true_x = 100, title = None, img_name=f"x^2_a={alpha}
    _l={_lambda}.pdf")
28
29     print(f"After training RSS {enp.get_RSS('train')}")
30     print(f"After training EN {enp.get_elastic_net('train')}")
31     print(f"After validation RSS {enp.get_RSS('val')}")
32     print(f"After validation EN {enp.get_elastic_net('val')}")
33     print(f"After weights")
34     print(enp.get_weights())
35     print()

```

6.2.5 finite_diff.py

```

1  from elastic_net_helper import ElasticNetHelper
2  from prints import *
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  f = lambda x: x**2
7
8  save_name = "../Images/x^2_matrix.pdf"
9
10 # Store RSS for each alpha and lambda
11 num_a = 101
12 num_l = 101
13 num_seeds = 100
14
15 alphas = np.linspace(0,1,num_a)
16 _lambdas = np.linspace(0,1,num_l)
17 RSS_mat = np.zeros((num_a,num_l))
18
19 # Get RSS values, keep track of minimum
20 min_RSS, min_alpha, min_lambda, min_weights = np.inf, -1, -1, None
21
22 small_banner(f"x^2 matrix plot for {num_seeds} seeds", False, True)
23
24 for alpha_iter in range(len(alphas)):
25     for _lambda_iter in range(len(_lambdas)):
26         enp = ElasticNetHelper(f = f , degree = 5, x_min = -5, x_max = 5,\
27                                num_evals_x = 20, num_train_x = 10,\
28                                noise = 1, verbose = False)
29
30         if alpha_iter == 0 and _lambda_iter == 0:
31             enp.print_params()
32
33         alpha = alphas[alpha_iter]
34         _lambda = _lambdas[_lambda_iter]
35
36         # Count all RSS for this seed
37         RSS_sum = 0

```

```

38
39     for seed in np.arange(0,num_seeds,1):
40         enp.new_en_solver(alpha = alpha, _lambda = _lambda, seed =
seed.item())
41         enp.train(10)
42         RSS_sum += enp.get_RSS('val')
43
44     this_RSS = RSS_sum/num_seeds
45     RSS_mat[alpha_iter][_lambda_iter] = this_RSS
46
47     if this_RSS < min_RSS:
48         min_RSS = this_RSS
49         min_alpha = alpha
50         min_lambda = _lambda
51         min_weights = enp.get_weights()
52
53 # Print minima
54 print(f"Minimum RSS is {min_RSS} with alpha {min_alpha} and lambda {
min_lambda}")
55 print(f"Minimum weights were:")
56 print(min_weights)
57 print()
58
59 # Make plot showing the result
60
61 fig, ax = plt.subplots(1,1,figsize=(5,4), dpi=120, facecolor='white',
tight_layout={'pad': 1})
62
63 # Convert alphas to printable version
64 alpha_label_locs= np.arange(0,num_a,10)
65 _lambda_label_locs= np.arange(0,num_l,10)
66
67 # Make sure last element is there
68 if num_a-1 not in alpha_label_locs:
69     alpha_label_locs = np.append(alpha_label_locs, [num_a-1])
70
71 if num_l-1 not in _lambda_label_locs:
72     _lambda_label_locs = np.append(_lambda_label_locs, [num_l-1])
73
74 alpha_labels= [f'{1.0 if a >= len(alphas) else alphas[a]:0.2f}' for a in
alpha_label_locs]
75 _lambda_labels= [f'{1.0 if l >= len(_lambdas) else _lambdas[l]:0.2f}' for
l in _lambda_label_locs]
76
77 a1 = ax.matshow(RSS_mat)
78 plt.colorbar(a1, label="RSS")
79 ax.set_xlabel("$\\lambda$")
80 ax.set_ylabel("$\\alpha$")
81 ax.set_yticks(alpha_label_locs, alpha_labels, rotation=00)
82 ax.set_xticks(_lambda_label_locs, _lambda_labels, rotation=90)
83 ax.tick_params(labelbottom = True)
84 ax.tick_params(labeltop = False)
85 ax.tick_params(top = False)
86 #plt.show()

```



```

87 plt.savefig(save_name)
88 print(f"Saved figure to {save_name}")

```

6.2.6 finite_diff.py

```

1 import numpy as np
2 from numpy.linalg import norm
3
4 class ElasticNet:
5     def __init__(self, x_data, y_data, degree, alpha, _lambda, b_init = 0,
6         verbose = False):
7         '''
8         Initialize the Elastic Net solver
9
10        Parameters
11
12        x_data:  Numpy array of size (n+1,) for data x-values
13        y_data:  Numpy array of size (n+1,) for data y-values
14        degree:  Degree polynomial we're fitting to
15        alpha:   See "Regularization Paths for Generalized Linear Models
16        via Coordinate Descent" (2010)
17        _lambda: See "Regularization Paths for Generalized Linear Models
18        via Coordinate Descent" (2010)
19        verbose: (True) Enable / (False) disable print statements
20
21        Returns
22
23        Nothing
24        '''
25
26        # Store initial values
27        self.x_data_initial = x_data
28        self.y_data = y_data
29        self.degree = degree
30        self.alpha = alpha
31        self._lambda = _lambda
32        self.verbose = verbose
33
34        # For ease, store value for number of data points
35        self.N = self.x_data_initial.shape[0]
36
37        # Create our X matrix from the paper
38        self.X = self.create_X(self.x_data_initial, self.degree)
39
40        # Standardize x values
41        self.X, self.X_means, self.X_stds = self.standardize_X(self.X)
42
43        # Make initial weights values
44        # TODO: What should these be initialized to? Currently just doing
45        zero
46        self.b = np.ones(degree+1)*b_init

```

```

46         # Get b0 term
47         self.b[0] = self.get_intercept()
48
49     def get_X(self):
50         '''
51         'get_X'
52
53         Returns the X matrix
54         '''
55         return self.X
56
57     def get_means(self):
58         '''
59         'get_means'
60
61         Returns the means for the data matrix
62         '''
63         return self.X_means
64
65     def get_stds(self):
66         '''
67         'get_stds'
68
69         Returns the stds for the data matrix
70         '''
71         return self.X_stds
72
73     def standardize_X(self, X):
74         '''
75         'standardize_X'
76
77         Normalizes X matrix per column
78         Each column has mean zero and sum of squares divided by rows as 1
79
80         Parameters
81
82         X matrix which is standardized
83
84         Returns
85
86         Standardized X matrix, the column means, the column standard
87         deviations
88         '''
89         means = np.mean(X, 0)
90         stds = np.std(X, 0)
91
92         X, means, stds = self.standardize_X_ms(X, means, stds)
93
94         return X, means, stds
95
96     def standardize_X_ms(self, X, means, stds):
97         '''
98         'standardize_X_ms'

```

```

99
100     Normalizes X matrix per column given means and stds
101     Each column has mean zero and sum of squares divided by rows as 1
102
103     Parameters
104
105     X matrix which is standardized
106     Column means which are used in standardization
107     Column standard deviations which are used in standardization
108
109     Returns
110
111     Standardized X matrix, the column means used, the column standard
112     deviations used
113     '''
114
115     # Remove zeros from standard deviations
116     for i in range(len(stds)):
117         if stds[i] == 0. :
118             stds[i] = 1.
119     X = (X - means)/stds
120
121     return X, means, stds
122
123 def unstandardize_X(self, X):
124     '''
125     'unstandardize_X'
126
127     Un-normalizes X matrix per column using the training means and
128     standard deviations
129
130     Parameters
131
132     X which is to be unstandardized
133
134     Returns
135
136     Unstandardized input matrix X
137     '''
138
139     X = X*self.X_std + self.X_mean
140     return X
141
142 def get_b(self):
143     '''
144     'get_b'
145
146     Returns current elastic net weights
147     '''
148     return self.b
149
150 def get_prediction(self, x_eval):
151     '''
152     'get_prediction'

```

```

151
152     Given non-standardized x-values, return our prediction for y-
values, but using weights trained on standardized x-values
153
154     Parameters
155
156     x_eval: x-values we want our predictions at
157
158     Returns
159
160     y-values at those x-values
161     '''
162
163     # Create our X matrix from the paper
164     X = self.create_X(x_eval, self.degree)
165
166     # Standardize x values
167     X, _, _ = self.standardize_X_ms(X, self.X_means, self.X_stds)
168
169     # Get b0 term
170     self.b[0] = self.get_intercept()
171
172     # Get the y values for our X's
173     y = X @ self.b
174
175     return y + self.b[0]
176
177 def get_intercept(self):
178     '''
179     'get_intercept'
180
181     Returns
182
183     Intercept formula obtained from Tyler
184     '''
185
186     b0 = sum((self.y_data - self.X@self.b)[1:])/self.N
187     return b0
188
189 def iterate_coord_descent(self, n):
190     '''
191     'iterate_coord_descent'
192
193     Does n steps of coordinate descent for each weight b[1] to b[-1]
194
195     Parameters
196
197     n: Number of steps to do
198
199     Returns
200
201     Nothing, but updates weights in beta
202     '''
203

```

```

204     for _ in range(n):
205         for j in range(1, self.degree+1):
206             self.step_j(j)
207
208     def step_j(self, j):
209         '''
210         'step_j'
211
212         Does a coordinate descent step for variable j in beta
213         j has to be nonzero since we're not optimizing the intercept
214
215         This is equation 5 in
216         "Regularization Paths for Generalized Linear Models via Coordinate
217         Descent" (2010)
218
219         Parameters
220
221         j: Index into beta that we're optimizing
222
223         Returns
224
225         Nothing, but updates variable j in the weights
226         '''
227
228         if j <= 0:
229             raise Exception(f"step_j: j ({j}) must be greater than 0")
230
231         # Solve for y tilde (j) first
232         y_tilde = np.sum(self.X*self.b,1)-self.X[:,j]*self.b[j]
233
234         # First calculate sigma from equation (5)
235         inner_sum = np.sum(self.X[:,j]*(self.y_data - y_tilde))
236
237         # Then, divide it by N
238         param_1 = inner_sum/self.N
239
240         # Get second parameter for the soft-thresholding operator
241         param_2 = self._lambda*self.alpha
242
243         # Calculate numerator
244         numerator = self.soft_thresholding(param_1, param_2)
245
246         # Calculate denominator
247         denominator = 1+self._lambda*(1-self.alpha)
248
249         # Divide to get result
250         res = numerator/denominator
251
252         if self.verbose:
253             print("y_tilde")
254             print(y_tilde)
255             print("y_data - y_tilde")
256             print(self.y_data - y_tilde)
257             print("inner_sum")

```

```

257         print(inner_sum)
258         print("param_1")
259         print(param_1)
260         print("param_2")
261         print(param_2)
262         print("numerator")
263         print(numerator)
264         print("denominator")
265         print(denominator)
266         print("res")
267         print(res)
268
269     self.b[j] = res
270
271     def soft_thresholding(self, z, y):
272         '''
273         'soft_thresholding'
274
275         This is the soft-thresholding operator, equation 6 in
276         "Regularization Paths for Generalized Linear Models via Coordinate
277         Descent" (2010)
278         '''
279         if y >= abs(z):
280             return 0
281         elif z > 0:
282             return z - y
283         else:
284             return z + y
285
286     def create_X(self, x_data, degree):
287         '''
288         'create_X'
289
290         Creates the X matrix in our paper
291
292         Parameters
293
294         x_data: Numpy array of size (n+1,) for data x-values
295         degree: Degree polynomial we're fitting to
296
297         Returns
298
299         X matrix as described in our paper
300         '''
301         X = np.zeros((x_data.shape[0], degree+1))
302         for col in range(degree+1):
303             X[:,col] = np.power(x_data,col)
304         return X
305
306     def get_elastic_net(self, x_data, y_data):
307         '''
308         'get_elastic_net'
309
310         Gets the elastic net value, formula 1 in

```

```

310     "Regularization Paths for Generalized Linear Models via Coordinate
311     Descent" (2010)
312     that we're trying to minimize
313
314     Parameters
315
316     x_data, y_data: (x,y) points that we're calculating the residual
317     sum of squares for
318
319     Returns
320
321     Elastic net formula that is being minimized (formula 1 in the
322     above paper)
323     '''
324
325     # Calculate RSS term
326     RSS = self.get_RSS(x_data, y_data)
327
328     # Calculate regularization term
329     P = (1-self.alpha)*norm(self.b[1:],2)**2/2 + self.alpha*norm(self.
330     b[1:],1)
331
332     # Return function elastic net is trying to minimize
333     return RSS + P
334
335 def get_RSS(self, x_data, y_data):
336     '''
337     'get_RSS'
338
339     Gets the Residual Sum of Squares
340
341     Parameters
342
343     x_data, y_data: (x,y) points that we're calculating the residual
344     sum of squares for
345
346     Returns
347
348     Elastic net formula that is being minimized (formula 1 in the
349     above paper)
350     '''
351
352     # Create our X matrix and standardize it
353     X = self.create_X(x_data, self.degree)
354     X, _, _ = self.standardize_X(X)
355
356     # Calculate RSS term
357     RSS = sum(np.power(np.sum(X*self.b,1)-y_data,2))
358     RSS = RSS/self.N
359
360     # Return Residual Sum of Squares
361     return RSS

```

6.2.7 estimators.py

```
1 import numpy as np
2 import finite_diff
3
4 class ridge:
5     def __init__(self, gamma = 0, degree = 1):
6         #gamma is regularization constat, degree is degree of polynomial
7         self.gamma = gamma
8         self.degree = degree
9         self.E_ridge = None
10
11     # Construct A matrix
12     def construct_A(self, input_x):
13         A_matrix = np.ones((input_x.shape[0], self.degree + 1))
14         for i in range(A_matrix.shape[1] - 1):
15             A_matrix[:, i + 1] = np.power(input_x, i + 1)
16         return A_matrix
17
18     #Use the closed form solution of the ridge estimator to find solution
19     def fit(self, train_x, train_y):
20         A_matrix = self.construct_A(train_x)
21         b = train_y
22         self.E_ridge = np.linalg.inv(np.transpose(A_matrix)@A_matrix + self.
23             gamma*np.identity(A_matrix.shape[1]))@np.transpose(A_matrix)@b
24
25     #Predict y values for given x values after estimator has been fitted
26     def predict(self, input_x):
27         A_test = self.construct_A(input_x)
28         b_hat = A_test@self.E_ridge
29         return b_hat
30
31     #Caclulate RSS for some validation x and validation y
32     def RSS(self, valid_x, valid_y):
33         b_hat = self.predict(valid_x)
34         rss = np.sum(np.power((b_hat - valid_y),2))
35         return rss
36
37 class tikhonov:
38     ## Weights needs to be a list that will work for implementation of
39     finite_diff.py
40     def __init__(self, _lambda, degree, weights):
41         self._lambda = _lambda
42         self.degree = degree
43         self.xstar = None
44         self.weight_matrix = weights
45
46     def construct_A(self, in_x):
47         A = np.ones((in_x.shape[0], self.degree + 1))
48         for i in range(self.degree):
49             A[:, i + 1] = np.power(in_x, i + 1)
50         return A
```



```

51 def fit(self, train_x, train_y):
52     A = self.construct_A(train_x)
53     b = train_y
54     D = self.weight_matrix
55     self.xstar = np.linalg.inv((np.transpose(A) @ A + self._lambda**2 * np
    .transpose(D) @ D)) @ np.transpose(A) @ b
56
57 def predict(self, test_x):
58     A_test = self.construct_A(test_x)
59     b_hat = A_test @ self.xstar
60     return b_hat
61
62 def RSS(self, test_x, test_y):
63     b_hat = self.predict(test_x)
64     rss = np.sum(np.power((b_hat - test_y), 2))
65     return rss

```

6.2.8 finite_diff.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def generate_D(FD_list):
5     '''
6     Generates the derivative matrix 'D' given a list 'FD_list' that
7     specifies the type of finite difference method for each row
8
9     input:\n
10     'FD_list': A list of tuples of the form (FD, N), where FD is the FD
11     option with stride N (see below) for each row
12
13     output:\n
14     A matrix 'D' with the values filled in from 'FD_list'
15
16     finite difference (FD) options for various strides (N): \n
17     0: empty row
18     1: centered difference          'f'(x) ~ (f(x-N)-f(x+N))/(2*N)
19     2: forward difference          'f'(x) ~ (f(x+N) - f(x))/N
20     3: backward difference         'f'(x) ~ (f(x) - f(x-N))/N
21     4: centered difference 2nd order 'f''(x) ~ (f(x+N)-2*f(x)+f(x-N))/N
22     **2
23     5: forward difference 2nd order 'f''(x) ~ (f(x+2*N)-2*f(x+N)+f(x))/
24     N**2
25     6: backward difference 2nd order 'f''(x) ~ (f(x)-2*f(x-N)+f(x-2*N))/
26     N**2
27     '''
28
29     # Size of our D matrix
30     size = len(FD_list)
31
32     # Initialize the D matrix
33     D = np.zeros((size, size))
34
35     # Populate D matrix

```

```

31     for i in range(size):
32         # Extract values from FD_list
33         FD = FD_list[i][0]
34         N = FD_list[i][1]
35
36         # Get the row we're inserting and a helper value for padding (see
37         # '_get_FD''s 'diff')
38         row, diff = _get_FD(FD, N)
39
40         # Calculate padding
41         padding_before = i - diff
42         padding_after = size - padding_before - len(row)
43
44         # Verify the padding makes sense
45         if padding_before < 0 or padding_after < 0:
46             raise Exception(f"generate_D: Row {i} of the matrix D has
47             negative padding: FD {FD}, N {N}, padding_before {padding_before},
48             padding_after {padding_after}")
49
50         if padding_before + padding_after + len(row) != size:
51             raise Exception(f"generate_D: Row {i} of the matrix D is not
52             the right size: FD {FD}, N {N}, row size {len(row)}")
53
54         # Generate padded row with zeros prepended
55         row = np.pad(row, (padding_before, padding_after), constant_values
56         =0)
57
58         # Replace row D with our row
59         D[i,:] = row
60
61     return D
62
63 def _get_FD(FD, N):
64     """
65     Helper function to get the smallest row-matrix for the finite
66     difference method and calculate how many indices behind the current x
67     we're at (used in 'generate_D') \n
68
69     input:\n
70     'FD': Finite difference method
71     'N': Stride length
72
73     output:\n
74     Tuple (row, diff)
75     'row': The finite difference row
76     'diff': Number of values before the current x that to start of the row
77
78     (ie. for centered difference with an N of 2, return [-1/4, 0, 0, 0,
79     1/4])
80     """
81     if N < 1 or not isinstance(N, int):
82         raise Exception(f"get_FD: N ({N}) must be a positive integer")
83
84     if FD not in [0,1,2,3,4,5,6]:

```

```

77         raise Exception(f"get_FD: FD ({FD}) must be an integer in [0,6]")
78
79     if FD == 0:
80         ret = np.zeros(0)
81         return (ret, 0)
82     if FD == 1:
83         ret = np.zeros(2*N+1)
84         ret[0] = -1/(2*N)
85         ret[-1] = 1/(2*N)
86         return (ret, N)
87     if FD == 2 or FD == 3:
88         # Row is the same, but its position changes on the method
89         ret = np.zeros(N+1)
90         ret[0] = -1/(N)
91         ret[-1] = 1/(N)
92
93         if FD == 2:
94             return (ret, 0)
95         else:
96             return (ret, N)
97     if FD == 4 or FD == 5 or FD == 6:
98         # Row is the same, but its position changes on the method
99         ret = np.zeros(2*N+1)
100         ret[-1] = 1/(N**2)
101         ret[N] = -2/(N**2)
102         ret[0] = 1/(N**2)
103
104         if FD == 4:
105             return (ret, N)
106         elif FD == 5:
107             return (ret, 0)
108         else:
109             return (ret, 2*N)
110
111 def generate_centered_D(N):
112     N -= 2
113     FD_list = [(0, 1)] + [(1,1)]*N + [(0, 1)]
114     D = generate_D(FD_list)
115     D = D[1:-1]
116     return D
117
118 def generate_forward_D(N):
119     N -= 1
120     FD_list = [(2, 1)] * (N) + [(0, 1)]
121     D = generate_D(FD_list)
122     return D
123
124 def generate_backwards_D(N):
125     N -= 1
126     FD_list = [(0,1)] + [(3, 1)] * (N)
127     D = generate_D(FD_list)
128     return D
129
130 def generate_2nd_centered_D(N):

```

```

131     N -= 2
132     FD_list = [(0, 1)] + [(4,1)]*N + [(0, 1)]
133     D = generate_D(FD_list)
134     D = D[1:-1]
135     return D
136
137
138 def test_D_1():
139     '''
140     Test function to make sure D is created properly
141     '''
142     FD_list = [(0, 1), (1, 1), (1, 1), (6, 1), (4, 1), (5, 1), (1, 1), (1,
143     1), (0, 1)]
144     D = generate_D(FD_list)
145     plt.matshow(D)
146     plt.show()
147
148 def test_D_2():
149     '''
150     Test function to make sure D is created properly
151     '''
152     N = 5
153     D = generate_centered_D(N)
154     plt.matshow(D)
155     plt.show()
156
157 #test_D_1()
158 #test_D_2()

```

6.2.9 estimators.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import estimators
4
5 # Folder to save images in
6 save_dir = '../Images/'
7
8 # Seed
9 np.random.seed(10)
10
11 # Option to either show or save the image
12 save_fig = True
13
14 # Create plot 1: Linear
15 f_original = lambda x: x # Non-noisy function
16 sigma = 0.5 # Noise (standard deviation of gaussian noise)
17 f_noisy = lambda x: f_original(x) + np.random.normal(0,sigma,x.shape) #
    Non-noisy function
18
19 x_min = -5
20 x_max = 5
21
22 num_pts_original = 1000 # Points for non-noisy function

```

```

23 num_pts_noisy = 100 # Noisy points
24
25 x_eval_original = np.linspace(x_min,x_max,num_pts_original)
26 x_eval_noisy = np.linspace(x_min,x_max,num_pts_noisy)
27
28 y_eval_original = f_original(x_eval_original)
29 y_eval_noisy = f_noisy(x_eval_noisy)
30
31 plt.plot(x_eval_original,y_eval_original,color='black',label="Original
    Function")
32 plt.scatter(x_eval_noisy,y_eval_noisy,s=5,color='red',label="Noisy
    Function")
33 plt.legend()
34
35 if save_fig:
36     plt.savefig(save_dir + "intro_1.pdf")
37 else:
38     plt.show()
39 plt.close()
40
41 # Create plot 2: Cubic
42 f_original = lambda x: x**3 # Non-noisy function
43 sigma = 10 # Noise (standard deviation of gaussian noise)
44 f_noisy = lambda x: f_original(x) + np.random.normal(0,sigma,x.shape) #
    Non-noisy function
45 f_quad = lambda x: -1*6*(x+0.5)**2
46
47 x_min = -5
48 x_max = 5
49
50 num_pts_original = 1000 # Points for non-noisy function
51 num_pts_noisy = 5 # Noisy points
52
53 x_eval_original = np.linspace(x_min,x_max,num_pts_original)
54 x_eval_noisy = np.random.uniform(x_min,x_max,num_pts_noisy)
55
56 y_eval_original = f_original(x_eval_original)
57 y_eval_noisy = f_noisy(x_eval_noisy)
58
59 R2 = estimators.ridge(gamma=0, degree=2)
60
61 R2.fit(x_eval_noisy,y_eval_noisy)
62 y_eval_d2 = R2.predict(x_eval_original)
63
64 plt.plot(x_eval_original,y_eval_original,color='black',label="Original
    Function")
65 plt.plot(x_eval_original,y_eval_d2,color='blue',linestyle='dotted',label="
    Degree 2 fit")
66 plt.scatter(x_eval_noisy,y_eval_noisy,s=5,color='red',label="Noisy
    Function")
67 plt.ylim(-150,150)
68 plt.legend()
69
70 if save_fig:

```

```

71     plt.savefig(save_dir + "intro_2.pdf")
72 else:
73     plt.show()
74 plt.close()
75
76 # Create plot 3: Cubic
77 f_original = lambda x: x**3 # Non-noisy function
78 sigma = 10 # Noise (standard deviation of gaussian noise)
79 f_noisy = lambda x: f_original(x) + np.random.normal(0,sigma,x.shape) #
    Non-noisy function
80 #f_quad = lambda x: -1*6*(x+0.5)**2
81
82 x_min = -5
83 x_max = 5
84
85 num_pts_original = 100 # Points for non-noisy function
86 num_pts_noisy = 5 # Noisy points
87
88 x_eval_original = np.linspace(x_min,x_max,num_pts_original)
89 x_eval_noisy = np.random.uniform(x_min,x_max,num_pts_noisy)
90
91 deg = 4
92 R2 = estimators.ridge(gamma=0, degree=deg)
93
94 y_eval_original = f_original(x_eval_original)
95 y_eval_noisy = f_noisy(x_eval_noisy)
96
97 R2.fit(x_eval_noisy,y_eval_noisy)
98 y_eval_d2 = R2.predict(x_eval_original)
99
100 plt.plot(x_eval_original,y_eval_original,color='black',label="Original
    Function")
101 plt.scatter(x_eval_noisy,y_eval_noisy,s=5,color='red',label="Noisy
    Function")
102 plt.plot(x_eval_original,y_eval_d2,color='blue',linestyle='dotted',label=f
    "Degree {deg} fit")
103 plt.ylim(-150,150)
104 plt.legend()
105
106 if save_fig:
107     plt.savefig(save_dir + "intro_3.pdf")
108 else:
109     plt.show()
110 plt.close()

```

6.2.10 estimators.py

```

1 '''
2 prints.py
3
4 Contains helper functions for printing things
5 '''
6
7 def small_banner(string, before_space = False, after_space = False):

```

```

8     '''
9     'small_banner'
10
11     Prints something like:
12     #####
13     # <string> #
14     #####
15
16     and adds before/after spacing depending on inputs (False by default)
17     '''
18     if before_space:
19         print()
20
21     new_string = "# " + string + " #"
22     length = len(new_string)
23     print("#"*length)
24     print(new_string)
25     print("#"*length)
26
27     if after_space:
28         print()

```

6.2.11 ridge_2a_driver.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import estimators
4 import sample
5
6 # Save or show plots
7 save_plots = True
8
9 #set colors
10 color1 = '#FF595E'
11 color2 = '#1982C4'
12 color3 = '#6A4C93'
13 colors =[color1, color2, color3]
14
15 #no interval is given of where to sample f, choose [-5, 5]
16 #choose function and number of samples
17 f = lambda x: 3*x + 2
18 a = -5
19 b = 5
20 number_of_samples = 20
21 number_of_train_samples = 10
22
23 xeval = np.linspace(-5,5,100)
24 degree = 1
25 seed = 50
26 #get random sample, and divide into training and validation data
27
28 train_x, train_y, valid_x, valid_y = sample.random_sample_equi(
    number_of_samples, f, a, b, number_of_train_samples, seed = seed)
29

```

```

30 fig_initial, ax_initial = plt.subplots(1,1)
31 ax_initial.plot(xeval, f(xeval), label = 'f(x) = 3x + 2', color = color3)
32 ax_initial.plot(train_x, train_y, '.', color = 'green', label = 'Training
    Data')
33 ax_initial.legend()
34 ax_initial.set_xlabel('x')
35 ax_initial.set_ylabel('y')
36
37 if save_plots:
38     plt.savefig("../Images/2a_only_data.pdf")
39 else:
40     plt.show()
41
42 #make graph for gammas = 0, 0.1, calculate RSS, seed = 50
43 gammas_initial = [0,0.1]
44 linestyle = ['-', '--']
45 rss_initial = []
46 counter = 0
47 for gamma in gammas_initial:
48     ridge = estimators.ridge(gamma, degree)
49     ridge.fit(train_x, train_y)
50     rss_initial.append(ridge.RSS(valid_x, valid_y))
51     ax_initial.plot(xeval, ridge.predict(xeval), linestyle[counter], label
        = '$\gamma$ = ' + str(gamma), color = colors[counter])
52     counter += 1
53 ax_initial.legend()
54 print(rss_initial)
55
56 if save_plots:
57     plt.savefig("../Images/2a_initial_gammas.pdf")
58 else:
59     plt.show()
60 plt.close()
61
62 #make RSS graph for gammas between 0 and 50 for seed = 50
63 gammas_log = np.linspace(0,50,1000)
64 fig_log10, ax_log10 = plt.subplots(1,1)
65 rss_log10 = []
66 for gamma in gammas_log:
67     ridge = estimators.ridge(gamma, degree)
68     ridge.fit(train_x, train_y)
69     rss_log10.append(ridge.RSS(valid_x, valid_y))
70 ax_log10.plot(gammas_log, rss_log10, color = color2)
71 ax_log10.set_xlabel('$\gamma$')
72 ax_log10.set_ylabel('Residual Sum of Squares')
73
74 if save_plots:
75     plt.savefig("../Images/2a_seed50_gammas.pdf")
76 else:
77     plt.show()
78 plt.close()
79
80 #Make RSS graph for gammas between 0 and 50 for seeds 1-100
81 fig, ax = plt.subplots(1,1)

```



```

82 seed_list = range(1,101)
83 gammas = np.linspace(0,50,1000)
84 mean_std_mat = np.zeros((len(seed_list),len(gammas)))
85 gammas_best = []
86 for i in range(len(seed_list)):
87     seed = seed_list[i]
88     train_x, train_y, valid_x, valid_y = sample.random_sample_equi(
        number_of_samples, f, a, b, number_of_train_samples, seed = seed)
89     rss = []
90     for gamma in gammas:
91         ridge = estimators.ridge(gamma, degree)
92         ridge.fit(train_x, train_y)
93         rss.append(ridge.RSS(valid_x, valid_y))
94     gammas_best.append(gammas[np.argmin(rss)])
95     mean_std_mat[i,:] = rss
96     ax.plot(gammas, rss, alpha = 0.2)
97
98 ax.set_xlabel('$\gamma$')
99 ax.set_ylabel('log10 of Residual Sum of Squares')
100 ax.set_yscale('log')
101
102 if save_plots:
103     plt.savefig("../Images/2a_seeds1_100.pdf")
104 else:
105     plt.show()
106 plt.close()
107
108 # Calculate mean and stdev across different seeds
109 means = np.mean(mean_std_mat,axis=0)
110 best_mean = np.min(means)
111 best_gamma = gammas[np.argmin(means)]
112 stdevs = np.std(mean_std_mat,axis=0)
113
114 # Plot our mean and stdev plots
115 plt.plot(gammas, means, color="red", label="Mean")
116 plt.fill_between(gammas, means-stdevs,\
117                 means+stdevs,\
118                 color="red", alpha=0.25, edgecolor=None, label="Stdev")
119 plt.semilogy()
120 plt.legend()
121 plt.xlabel('$\gamma$')
122 plt.ylabel('log10 of Residual Sum of Squares')
123
124 if save_plots:
125     plt.savefig("../Images/2a_seeds1_100mean.pdf")
126 else:
127     plt.show()
128 plt.close()
129
130 plt.plot(gammas, means, color="red", label="Mean")
131 plt.fill_between(gammas, means-stdevs,\
132                 means+stdevs,\
133                 color="red", alpha=0.25, edgecolor=None, label="Stdev")
134 plt.xlim(0,1.5)

```

```

135 plt.ylim(11.25,12)
136 plt.legend()
137
138 if save_plots:
139     plt.savefig("../Images/2a_seeds1_100mean_zoomed.pdf")
140 else:
141     plt.show()
142 plt.close()
143
144 nonzero_gammas_count = np.count_nonzero(gammas_best)
145 print('Best gamma was ' + str(best_gamma))
146 print('Min RSS was ' + str(best_mean))
147 print('We had this many 0 gammas' + str(len(gammas_best) -
    nonzero_gammas_count))
148 print('We had this many nonzero gammas' + str(nonzero_gammas_count))
149 print('Percent nonzero ' + str(nonzero_gammas_count/len(gammas_best)))
150 print('Percent 0 gammas ' + str(1 - (nonzero_gammas_count/len(gammas_best))
    ))
151 print('Mean Best Gamma = ' + str(np.mean(gammas_best)))

```

6.2.12 ridge_2b_driver.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import estimators
4 import sample
5
6 # Save or show plots
7 save_plots = True
8
9 #set colors
10 color1 = '#FF595E'
11 color2 = '#1982C4'
12 color3 = '#6A4C93'
13 colors =[color1, color2, color3]
14
15 #no interval is given of where to sample f, choose [-5, 5]
16 #choose function and number of samples
17 f = lambda x: x**2
18 a = -5
19 b = 5
20 number_of_samples = 20
21 number_of_train_samples = 10
22
23 xeval = np.linspace(-5,5,1000)
24 degree = 5
25 seed = 50
26 #get random sample, and divide into training and validation data
27
28 train_x, train_y, valid_x, valid_y = sample.random_sample_equi(
    number_of_samples, f, a, b, number_of_train_samples, seed = seed)
29
30 #Make Graphs for gammas = 0, 0.1 and calculate RSS, seed = 50
31 gammas_initial = [0,0.1]

```

```

32 linestyle = ['-', '--']
33 fig_initial, ax_initial = plt.subplots(1,1)
34 rss_initial = []
35 counter = 0
36 for gamma in gammas_initial:
37     ridge = estimators.ridge(gamma, degree)
38     ridge.fit(train_x, train_y)
39     rss_initial.append(ridge.RSS(valid_x, valid_y))
40     ax_initial.plot(xeval, ridge.predict(xeval), linestyle[counter], label
41                     = '$\gamma$ = ' + str(gamma), color = colors[counter])
42     counter += 1
43 ax_initial.plot(xeval, f(xeval), label = 'f(x) =  $x^2$ ', color = color3)
44 ax_initial.plot(train_x, train_y, '.', color = 'green', label = 'Training
    Data')
45 ax_initial.legend()
46 ax_initial.set_xlabel('x')
47 ax_initial.set_ylabel('y')
48 print(rss_initial)
49
50 if save_plots:
51     plt.savefig("../Images/2b_initial_gammas.pdf")
52 else:
53     plt.show()
54 plt.close()
55
56 #make RSS graph for gammas between 0 and 50 for seed = 50
57 gammas_log = np.linspace(0,50,1000)
58 fig_log10, ax_log10 = plt.subplots(1,1)
59 rss_log10 = []
60 for gamma in gammas_log:
61     ridge = estimators.ridge(gamma, degree)
62     ridge.fit(train_x, train_y)
63     rss_log10.append(ridge.RSS(valid_x, valid_y))
64 ax_log10.plot(gammas_log, rss_log10, color = color2)
65 ax_log10.set_xlabel('$\gamma$')
66 ax_log10.set_ylabel('Residual Sum of Squares')
67
68 if save_plots:
69     plt.savefig("../Images/2b_seed50_gammas.pdf")
70 else:
71     plt.show()
72 plt.close()
73
74 #Make RSS graph for gammas between 0 and 50 for seeds 1-100
75 fig, ax = plt.subplots(1,1)
76 seed_list = range(1,101) #iterate through all seeds
77 gammas = np.linspace(0,50,1000)
78 mean_std_mat = np.zeros((len(seed_list),len(gammas)))
79 gammas_best = []
80 for i in range(len(seed_list)):
81     seed = seed_list[i]
82     train_x, train_y, valid_x, valid_y = sample.random_sample_equi(
        number_of_samples, f, a, b, number_of_train_samples, seed = seed)
83     rss = []

```

```

83     for gamma in gammas:
84         ridge = estimators.ridge(gamma, degree)
85         ridge.fit(train_x, train_y)
86         rss.append(ridge.RSS(valid_x, valid_y))
87     gammas_best.append(gammas[np.argmin(rss)])
88     mean_std_mat[i,:] = rss
89     ax.plot(gammas, rss, alpha = 0.2)
90
91 ax.set_xlabel('$\gamma$')
92 ax.set_ylabel('log10 of Residual Sum of Squares')
93 ax.set_yscale('log')
94
95 if save_plots:
96     plt.savefig("../Images/2b_seeds1_100.pdf")
97 else:
98     plt.show()
99 plt.close()
100
101 # Calculate mean and stdev across different seeds
102 means = np.mean(mean_std_mat,axis=0)
103 best_mean = np.min(means)
104 best_gamma = gammas[np.argmin(means)]
105 stdevs = np.std(mean_std_mat,axis=0)
106
107 # Plot our mean and stdev plots
108 plt.plot(gammas, means, color="red", label="Mean")
109 plt.fill_between(gammas, means-stdevs,\
110                 means+stdevs,\
111                 color="red", alpha=0.25, edgecolor=None, label="Stdev")
112 plt.semilogy()
113 plt.xlabel('$\gamma$')
114 plt.ylabel('log10 of Residual Sum of Squares')
115 plt.legend()
116
117 if save_plots:
118     plt.savefig("../Images/2b_seeds1_100mean.pdf")
119 else:
120     plt.show()
121 plt.close()
122
123 nonzero_gammas_count = np.count_nonzero(gammas_best)
124 print('Best gamma was ' + str(best_gamma))
125 print('Min RSS was ' + str(best_mean))
126 print('We had this many 0 gammas ' + str(len(gammas_best) -
127     nonzero_gammas_count))
128 print('We had this many nonzero gammas ' + str(nonzero_gammas_count))
129 print('Percent nonzero ' + str(nonzero_gammas_count/len(gammas_best)))
130 print('Percent 0 gammas ' + str(1 - (nonzero_gammas_count/len(gammas_best))))
131 print('Mean Best Gamma = ' + str(np.mean(gammas_best)))

```

6.2.13 tikhonov.py

```

1 import numpy as np

```

```

2
3
4 #this function was consolidated into sample.py, use that file for future
  use/reference
5 def random_sample_equi(number_of_samples, f, a, b, number_of_train_samples
  , mean = 0, std_dev = 1, seed = None):
6
7
8     rng = np.random.default_rng(seed)
9     sample_x = np.linspace(a,b,number_of_samples)
10    sample_x = np.reshape((number_of_samples, 1))
11    gaussian_noise = rng.normal(mean, std_dev, (number_of_samples, 1))
12    sample_y = f(sample_x) + gaussian_noise
13    sample_x_y = np.concatenate((sample_x, sample_y), axis = 1)
14
15    train_data = rng.choice(sample_x_y, size = number_of_train_samples,
  replace = False)
16    valid_data = np.zeros((number_of_samples - number_of_train_samples, 2)
  )
17
18    counter = 0
19    for i in range(sample_x_y.shape[0]):
20        include = True
21        for j in range(train_data.shape[0]):
22            if sample_x_y[i,0] == train_data[j,0]:
23                include = False
24        if include == True:
25            valid_data[counter] = sample_x_y[i]
26            counter += 1
27
28    train_data = train_data[train_data[:,0].argsort()]
29    valid_data = valid_data[valid_data[:,0].argsort()]
30
31    train_x = train_data[:,0]
32    train_y = train_data[:,1]
33    valid_x = valid_data[:,0]
34    valid_y = valid_data[:,1]
35
36
37    return train_x, train_y, valid_x, valid_y

```

6.2.14 sample.py

```

1 import numpy as np
2
3
4 def random_sample(number_of_samples, f, a, b, number_of_train_samples,
  mean = 0, std_dev = 1, seed = None):
5     '''
6     Take in Number of Samples, Function, end points, how many samples
  should be training samples, mean, std_dev, and random seed
7     x values pulled from a uniform [a,b] distribution, adds Gaussian Noise
  to y values
8     Returns training x, training y, valid x, and valid y

```

```

9      '''
10
11     rng = np.random.default_rng(seed) #set seed
12     sample_x = rng.uniform(a,b, (number_of_samples, 1)) #get x values
13     gaussian_noise = rng.normal(mean, std_dev, (number_of_samples, 1))
14     sample_y = f(sample_x) + gaussian_noise #add noise
15     sample_x_y = np.concatenate((sample_x, sample_y), axis = 1) #make a
    single array so x and y are fixed together before random choice
16
17     train_data = rng.choice(sample_x_y, size = number_of_train_samples,
    replace = False) #find train data
18     valid_data = np.zeros((number_of_samples - number_of_train_samples, 2)
    )
19
20     counter = 0 #find validation data
21     for i in range(sample_x_y.shape[0]):
22         include = True
23         for j in range(train_data.shape[0]):
24             if sample_x_y[i,0] == train_data[j,0]:
25                 include = False
26         if include == True:
27             valid_data[counter] = sample_x_y[i]
28             counter += 1
29
30     train_data = train_data[train_data[:,0].argsort()]#sort data so
    plotting is nice
31     valid_data = valid_data[valid_data[:,0].argsort()]
32
33     train_x = train_data[:,0]
34     train_y = train_data[:,1]
35     valid_x = valid_data[:,0]
36     valid_y = valid_data[:,1]
37
38
39     return train_x, train_y, valid_x, valid_y
40
41 def random_sample_equi(number_of_samples, f, a, b, number_of_train_samples
    , mean = 0, std_dev = 1, seed = None):
42     '''
43     Take in Number of Samples, Function, end points, how many samples
    should be training samples, mean, std_dev, and random seed
44     x values are equispaced from [a,b], adds Gaussian Noise to y values
45     Returns training x, training y, valid x, and valid y
46     '''
47
48     rng = np.random.default_rng(seed) # set seed
49     sample_x = np.linspace(a,b,number_of_samples) #get x values
50     sample_x = np.reshape(sample_x, (number_of_samples, 1))
51     gaussian_noise = rng.normal(mean, std_dev, (number_of_samples, 1))
52     sample_y = f(sample_x) + gaussian_noise #add noise
53     sample_x_y = np.concatenate((sample_x, sample_y), axis = 1) #make a
    single array so x and y are fixed together before random choice
54
55     train_data = rng.choice(sample_x_y, size = number_of_train_samples,

```

```

56     replace = False) #find train data
    valid_data = np.zeros((number_of_samples - number_of_train_samples, 2)
57 )
58     counter = 0 #find validation data
59     for i in range(sample_x_y.shape[0]):
60         include = True
61         for j in range(train_data.shape[0]):
62             if sample_x_y[i,0] == train_data[j,0]:
63                 include = False
64         if include == True:
65             valid_data[counter] = sample_x_y[i]
66             counter += 1
67
68     train_data = train_data[train_data[:,0].argsort()] #sort data so
plotting is nice
69     valid_data = valid_data[valid_data[:,0].argsort()]
70
71     train_x = train_data[:,0]
72     train_y = train_data[:,1]
73     valid_x = valid_data[:,0]
74     valid_y = valid_data[:,1]
75
76
77     return train_x, train_y, valid_x, valid_y

```

6.2.15 tikhonov.py

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  import estimators
4  import finite_diff
5  from sample import random_sample_equi
6
7  # Save or show plots
8  save_plots = True
9
10 ## Generating training / testing data
11
12 num_train_samples = 60
13
14 color1 = '#FF595E'
15 color2 = '#1982C4'
16 color3 = '#6A4C93'
17 fname = '$sin(x) + sin(5x)$'
18
19 #function to visualize individual plots depending on input fig. Takes fig,
    an int 1-9, func a vectorized function, and a string of the function
    for plot labels
20 def visualize(fig, func, func_name):
21
22     # visualize tikhonov estimator vs actual function
23     if fig == 1:
24         #set random seed and generate random samples

```

```

25     seed = 50
26     x_train, y_train, x_test, y_test = random_sample_equi(2*
num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev
    = .7)
27
28     #meshes for function and polynomial plotting
29     xeval = np.linspace(-3,3,1000)
30     feval = func(xeval)
31
32     #initialize info for tikhonov
33     degree = 15
34     weights = finite_diff.generate_centered_D(degree + 1)
35
36     for lam in [0, 0.1, 1]:
37         # create and fit tikhonov, predict on test data
38         tikhonov = estimators.tikhonov(lam, degree, weights)
39         tikhonov.fit(x_train, y_train)
40         b_hat = tikhonov.predict(x_test)
41
42         #get polynomial by predictin on entire interval
43         poly = tikhonov.predict(xeval)
44
45         #plot everything
46         plt.plot(xeval, poly, label = 'Tikhonov Polynomial', color = color2)
47         plt.plot(x_train, y_train, '.', label = 'Training data', color =
color3)
48         plt.plot(xeval, feval, label = 'f(x) = ' + func_name, color = color1
    )
49         plt.xlabel('x')
50         plt.ylabel('y')
51         plt.legend()
52
53         if save_plots:
54             plt.savefig(f"../Images/tikhonov_poly_lambda{lam}.pdf")
55         else:
56             plt.show()
57         plt.close()
58
59     #RSS Values vs lambda for a specific seed
60     if fig == 2:
61         #set seed and generate random info, etc. same as above
62         seed = 50
63         x_train, y_train, x_test, y_test = random_sample_equi(2*
num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev
    = .7)
64
65         #generate all lambdas
66         lambdas = np.linspace(0, 20, 1000)
67         degree = 15
68         weights = finite_diff.generate_centered_D(degree + 1)
69
70         #list for storing RSS values
71         RSS_vals = []
72

```



```

73     #create and fit tikhonov for each lambda, collect RSS values.
74     for l in lambdas:
75         tikhonov = estimators.tikhonov(l, degree, weights)
76         tikhonov.fit(x_train, y_train)
77         RSS_vals += [tikhonov.RSS(x_test, y_test)]
78
79     #plot!
80     plt.plot(lambdas, RSS_vals, color = color2)
81     plt.xlabel('$\lambda$')
82     plt.ylabel('RSS')
83
84     if save_plots:
85         plt.savefig("../Images/tikhonovRSS.pdf")
86     else:
87         plt.show()
88     plt.close()
89
90     #find and print minimum lambda
91     print('lambda that achieves minimum: ', lambdas[np.argmin(RSS_vals)])
92
93     #RSS values vs degree for specific seed
94     if fig == 3:
95
96         #initialize same info as prev funcs.
97         seed = 50
98         x_train, y_train, x_test, y_test = random_sample_equi(2*
99         num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev
100         = .7)
101
102         #range of all degrees
103         degrees = range(3, 20)
104         lam = .1
105         RSS_vals = []
106
107         #create and fit tikhonov for each degree, collect and store RSS on val
108         . data.
109         for d in degrees:
110             weights = finite_diff.generate_centered_D(d + 1)
111             tikhonov = estimators.tikhonov(lam, d, weights)
112             tikhonov.fit(x_train, y_train)
113             RSS_vals += [tikhonov.RSS(x_test, y_test)]
114             plt.plot(degrees, RSS_vals, color = color2)
115             plt.xlabel('Degree')
116             plt.ylabel('RSS')
117
118         if save_plots:
119             plt.savefig("../Images/tikhonovRSSvsDEG.pdf")
120         else:
121             plt.show()
122         plt.close()
123
124     #RSS Values for various seeds
125     if fig == 4:
126         # initialize seed range

```

```

124 seeds = range(1,101)
125
126 #list to store best lambda for each seed
127 best_lambdas = []
128
129 #iterate over each seed
130 for seed in seeds:
131     #initialize data, tikhonov info, and lambda range
132     x_train, y_train, x_test, y_test = random_sample_equi(2*
num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev
= .7)
133     lambdas = np.linspace(0, 20, 1000)
134     degree = 15
135     weights = finite_diff.generate_centered_D(degree + 1)
136     RSS_vals = []
137     minRSS = float('inf')
138     minlam = 0
139     # iterate over lambdas and find minimum RSS and equivalent lambda
140     for l in lambdas:
141         tikhonov = estimators.tikhonov(l, degree, weights)
142         tikhonov.fit(x_train, y_train)
143         RSS = tikhonov.RSS(x_test, y_test)
144         RSS_vals += [RSS]
145         if RSS < minRSS:
146             minRSS = RSS
147             minlam = l
148     # add best lambda for each seed
149     best_lambdas += [minlam]
150     # plot the RSS vs Lambda line semi-transparent
151     plt.plot(lambdas, RSS_vals, alpha = .3)
152
153 #plot everything else and print zero and nonzero lambdas
154 plt.semilogy()
155 plt.xlabel('$\lambda$')
156 plt.ylabel('RSS')
157
158 if save_plots:
159     plt.savefig("../Images/lambda_all_seeds.pdf")
160 else:
161     plt.show()
162 plt.close()
163
164 nonzero = np.count_nonzero(best_lambdas)
165 zero = len(best_lambdas) - nonzero
166 print(f'Number of seeds where 0 is the best lambda: {zero} \n Number
of seeds where best lambda is nonzero: {nonzero}')
167
168 #RSS values for various seeds as a single statistical function
169 if fig == 5:
170     seeds = range(1,101)
171     num_lambdas = 100
172     degree = 11
173     weights = finite_diff.generate_centered_D(degree + 1)
174     lambdas = np.linspace(0, 20, num_lambdas)

```

```

175     y_evals = np.zeros((len(seeds), num_lambdas))
176
177     for i in range(len(seeds)):
178         seed = seeds[i]
179         x_train, y_train, x_test, y_test = random_sample_equi(2*
num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev
= .7)
180         RSS_vals = []
181         for j in range(len(lambdas)):
182             l = lambdas[j]
183             tikhonov = estimators.tikhonov(l, degree, weights)
184             tikhonov.fit(x_train, y_train)
185             RSS_val = tikhonov.RSS(x_test, y_test)
186             y_evals[i][j] = RSS_val
187
188     means = np.mean(y_evals, axis=0)
189     stdevs = np.std(y_evals, axis=0)
190
191     plt.plot(lambdas, means, color="red", label="Mean")
192     plt.fill_between(lambdas, means-stdevs, \
193                     means+stdevs, \
194                     color="red", alpha=0.25, edgecolor=None, label="Stdev"
)
195     plt.legend()
196     plt.xlabel("$\lambda$")
197     plt.ylabel("Residual Sum of Squares")
198     plt.savefig("../Images/Tikhonov_5.pdf")
199
200     if save_plots:
201         plt.savefig("../Images/Tikhonov_5.pdf")
202     else:
203         plt.show()
204     plt.close()
205
206 #RSS values for various degrees as a single statistical function
207 if fig == 6:
208     # Make plots of RSS values vs degree (y-axis has mean and standard
deviation) for a specific seed
209
210     seed = 50 # Set a constant seed
211     num_lambdas = 100 # Number of lambdas to evaluate
212     degrees = list(range(1,20)) # Degrees of the polynomials we're fitting
213     lambdas = np.linspace(0, 20, num_lambdas) # Range of lambdas we're
evaluating at
214     y_evals = np.zeros((len(degrees), num_lambdas)) # Matrix which stores
out results
215
216     # Iterate over the different polynomial fits
217     for i in range(len(degrees)):
218         # Set the current degree we're using
219         degree = degrees[i]
220         # Generate our D matrix for centered difference
221         weights = finite_diff.generate_centered_D(degree + 1)
222         # Generate our training and testing data

```

```

223     x_train, y_train, x_test, y_test = random_sample_equi(2*
num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev
    = .7)
224     # List that we're storing our result in for different lambdas
225     RSS_vals = []
226     for j in range(len(lambdas)):
227         l = lambdas[j]
228         # Solve our Tikhonov equation
229         tikhonov = estimators.tikhonov(l, degree, weights)
230         tikhonov.fit(x_train, y_train)
231         RSS_val = tikhonov.RSS(x_test, y_test)
232         # Store our result
233         y_evals[i][j] = RSS_val
234
235     # Calculate the mean and standard deviation for each lambda
236     means = np.mean(y_evals, axis=0)
237     stdevs = np.std(y_evals, axis=0)
238
239     # Make and save our plot
240     plt.plot(lambdas, means, color="red", label="Mean")
241     plt.fill_between(lambdas, means-stdevs, \
242                     means+stdevs, \
243                     color="red", alpha=0.25, edgecolor=None, label="Stdev"
)
244     plt.xlabel("$\lambda$")
245     plt.ylabel("Residual Sum of Squares")
246     plt.legend()
247
248     if save_plots:
249         plt.savefig("../Images/Tikhonov_6.pdf")
250     else:
251         plt.show()
252     plt.close()
253
254     #RSS values vs degree for specific seed and lambda
255     if fig == 7:
256         # Make plots of RSS values vs degree (y-axis has mean and standard
deviation) for a specific lambda
257
258         num_seeds = 100 # Number of seeds we'll iterator over
259         seeds = list(range(0, num_seeds)) # Make our list of seeds
260         degrees = list(range(3, 20)) # Degrees of the polynomials we're
fitting
261         num_degrees = len(degrees)
262
263         # Make our matrix that we'll store our results in
264         RSS_vals = np.zeros((num_seeds, num_degrees))
265
266         # Iterate over our seeds
267         for i in range(num_seeds):
268             seed = seeds[i]
269             # Get our training and testing data
270             x_train, y_train, x_test, y_test = random_sample_equi(2*
num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev

```

```

= .7)
271     # Constant lambda for Tikhonov
272     lam = .1
273     RSS_val = []
274     # Iterate over our degrees
275     for d in degrees:
276         # Generate our centered difference D matrix
277         weights = finite_diff.generate_centered_D(d + 1)
278         # Fit our Tikhonov
279         tikhonov = estimators.tikhonov(lam, d, weights)
280         tikhonov.fit(x_train, y_train)
281         # Store our result
282         RSS_val += [tikhonov.RSS(x_test, y_test)]
283     RSS_vals[i,:] = RSS_val
284
285     # Calculate our mean and standard deviation for each gamma
286     means = np.mean(RSS_vals,axis=0)
287     stdevs = np.std(RSS_vals,axis=0)
288
289     # Make our plot
290     plt.plot(degrees, means, color="red", label="Mean")
291     plt.fill_between(degrees, means-stdevs,\
292                     means+stdevs,\
293                     color="red", alpha=0.25, edgecolor=None, label="Stdev"
294 )
295     plt.ylim(0,8000)
296     plt.xlabel('Degree polynomial')
297     plt.ylabel('Residual Sum of Squares')
298     plt.legend()
299
300     if save_plots:
301         plt.savefig("../Images/Tikhonov_7.pdf")
302     else:
303         plt.show()
304     plt.close()
305
306 if fig == 8:
307     # Make plots training/testing data, our original function, and
308     # Tikhonov fit
309
310     # Iterate over our polynomials
311     for degree in range(15,20):
312         # Constant seed which looks nice for plotting purposes
313         seed = 4596
314         # Get our training and testing data
315         x_train, y_train, x_test, y_test = random_sample_equi(2*
316 num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev
317 = .7)
318         xeval = np.linspace(-3,3,1000)
319         feval = func(xeval)
320         # Generate our D matrix for centered difference
321         weights = finite_diff.generate_centered_D(degree + 1)
322         # Constant lambda for Tikhonov
323         lam = .1

```

```

320     # Fit our Tikhonov regularization
321     tikhonov = estimators.tikhonov(lam, degree, weights)
322     tikhonov.fit(x_train, y_train)
323     # Get our predicted polynomial
324     poly = tikhonov.predict(xeval)
325     # Make our plot
326     plt.plot(xeval, poly, label = 'Tikhonov Polynomial', color = color2)
327     plt.plot(x_train, y_train, '.', label = 'Training data', color =
color3)
328     plt.plot(x_test, y_test, '.', label = 'Testing data', color = '
hotpink')
329     plt.plot(xeval, feval, label = 'f(x) = ' + func_name, color = color1
)
330     plt.xlabel('x')
331     plt.ylabel('y')
332     plt.legend()
333
334     if save_plots:
335         plt.savefig(f"../Images/Tikhonov_8_{degree}.pdf")
336     else:
337         plt.show()
338     plt.close()
339
340 # Fits for different difference formulas
341 if fig == 9:
342     # same as above
343     seed = 50
344     x_train, y_train, x_test, y_test = random_sample_equi(2*
num_train_samples, func, -3, 3, num_train_samples, seed = seed, std_dev
= .7)
345     xeval = np.linspace(-3,3,1000)
346     feval = func(xeval)
347     degree = 15
348     #list of different weight matrices and their names
349     weights = [(finite_diff.generate_forward_D(degree + 1), 'Forward'), (
finite_diff.generate_backwards_D(degree + 1), 'Backwards'), (
finite_diff.generate_2nd_centered_D(degree + 1), '2nd_Deg_Centered')]
350     # generate tikhonov for each weight and create same plot as fig = 1
351     for w in weights:
352         lam = 1
353         tikhonov = estimators.tikhonov(lam, degree, w[0])
354         tikhonov.fit(x_train, y_train)
355         coefs = tikhonov.xstar
356         b_hat = tikhonov.predict(x_test)
357         poly = tikhonov.predict(xeval)
358         plt.plot(xeval, poly, label = 'Tikhonov Polynomial', color = color2)
359         plt.plot(x_train, y_train, '.', label = 'Training data', color =
color3)
360         plt.plot(xeval, feval, label = 'f(x) = ' + func_name, color = color1
)
361         plt.xlabel('x')
362         plt.ylabel('y')
363         plt.legend()
364

```

```
365     if save_plots:
366         plt.savefig(f"../Images/Tikhonov_9_{w[1]}.pdf")
367     else:
368         plt.show()
369     plt.close()
370
371
372 # Uncomment any of the below to produce a specific figure
373 f = lambda x : np.sin(x) + np.sin(5*x)
374 visualize(1, f, fname)
375 #visualize(2, f, fname)
376 #visualize(3, f, fname)
377 #visualize(4, f, fname)
378 #visualize(5, f, fname)
379 #visualize(6, f, fname)
380 #visualize(7, f, fname)
381 #visualize(8, f, fname)
382 #visualize(9, f, fname)
```