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 $\{1, x, x^2, ..., x^m\}$ 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 ($p_m(x) = \beta_0 + \beta_1 * x + \ldots + \beta_m * x^m$) 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 $p_m(x_i) = \beta_0 + \beta_1 * x_i + \ldots + \beta_m * x_i^m$. Using this as motivation, the following matrices are introduced:

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

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

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

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.

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

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,

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

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.

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

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.

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

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.

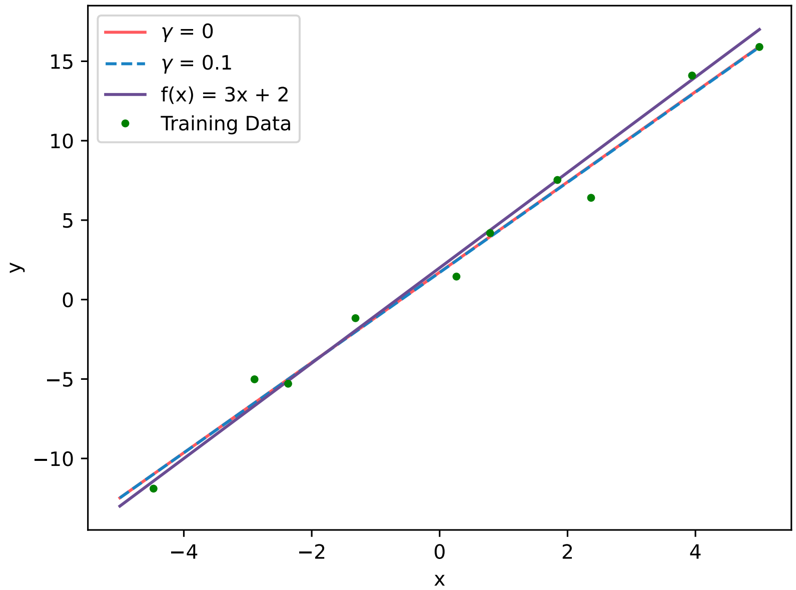

Figure 1: The function $y = 3x + 2$ fitted to a 1-degree polynomial using Ridge Regression for $\gamma = 0$, $\gamma = 0.1$. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

	$\gamma = 0$	$\gamma = 0.1$
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.

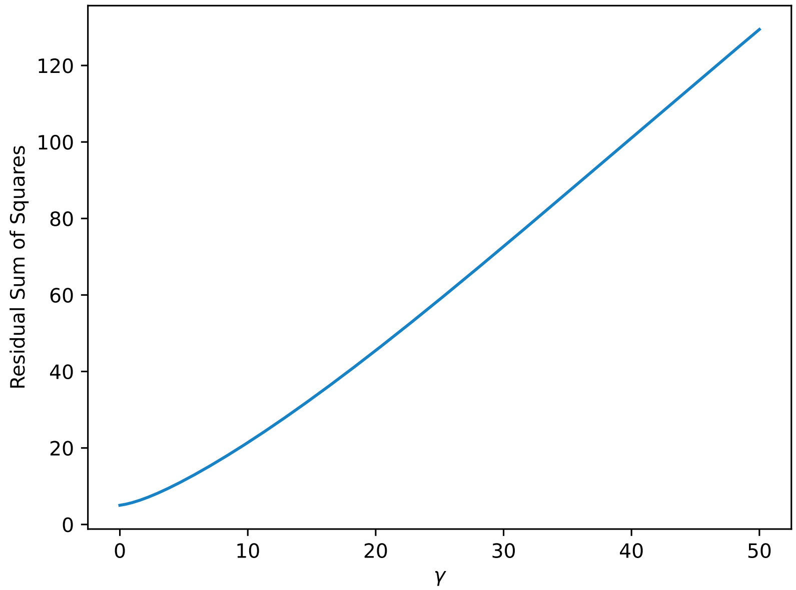

Figure 2: The RSS for $y = 3x + 2$ fitted to a 1-degree polynomial using Ridge Regression for $\gamma \in [0, 50]$. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

It appears that with this data Ordinary Least Squares is always better than Ridge Regresssion. 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.

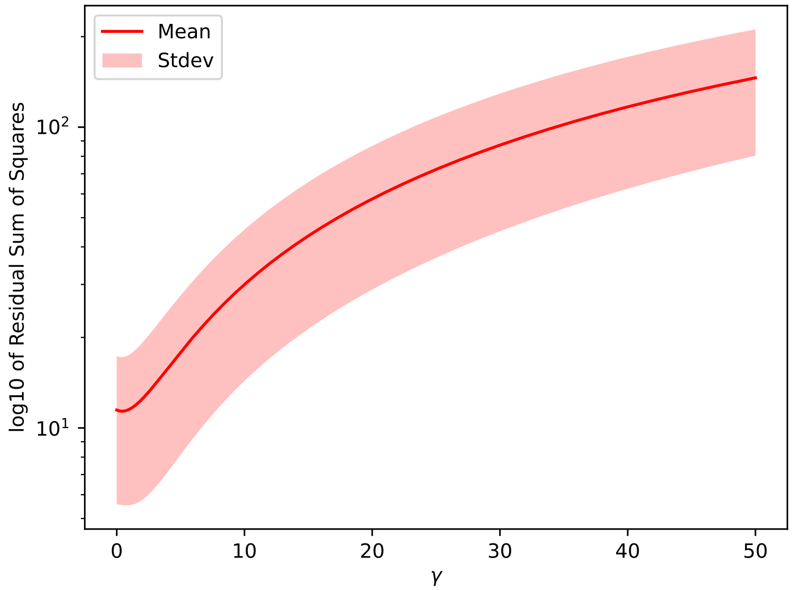

Figure 3: The mean and standard deviation of the RSS for $y = 3x + 2$ fitted to a 1-degree polynomial using Ridge Regression for $\gamma \in [0, 50]$ using 100 different seeds. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian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

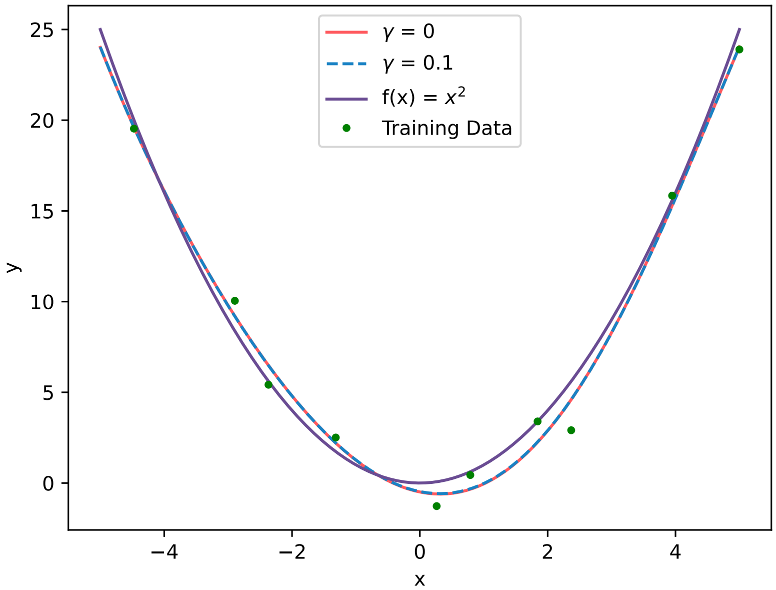

Figure 4: The function $y = x^2$ fitted to a 5-degree polynomial using Ridge Regression for $\gamma = 0$, $\gamma = 0.1$. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

	$\gamma = 0$	$\gamma = 0.1$
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.

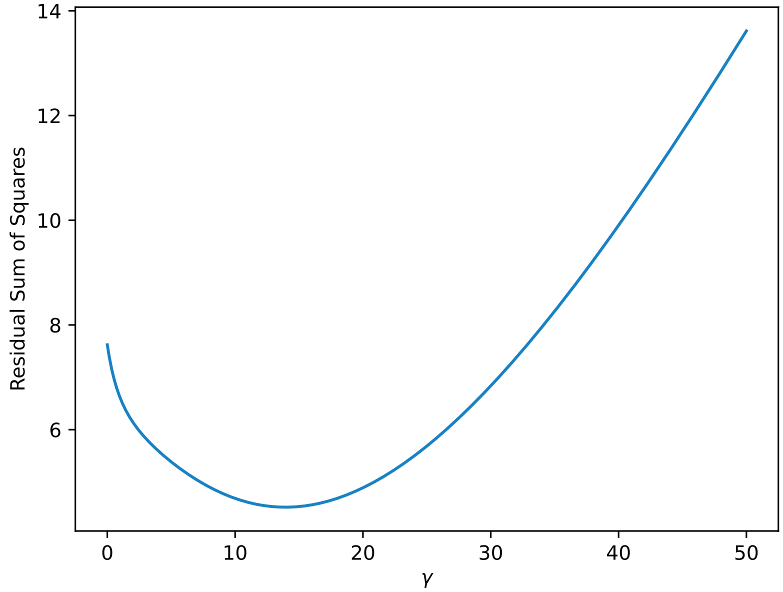

Figure 5: The RSS for $y = x^2$ fitted to a 5-degree polynomial using Ridge Regression for $\gamma \in [0, 50]$. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian noise added to the evaluations.

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.

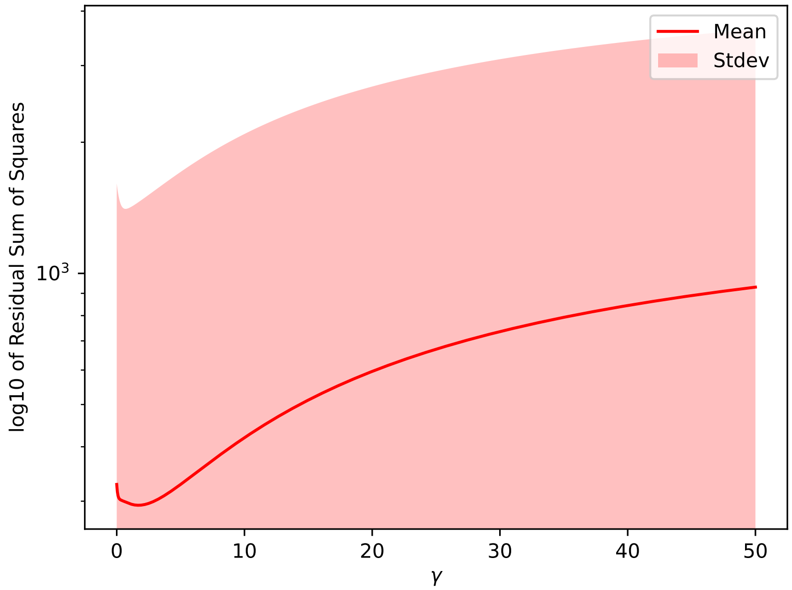

Figure 6: The mean and standard deviation of the RSS for $y = x^2$ fitted to a 5-degree polynomial using Ridge Regression for $\gamma \in [0, 50]$ using 100 different seeds. The training points were sampled from 20 equispaced points in the domain $x \in [-5, 5]$. The training points had random Gaussian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

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

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

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

expanding that,

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

From the Ridge Estimator derivation it's already known that

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

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

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

(See 6.1.2 for a rigorous derivation.)

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

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

or,

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

Thus

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

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

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

will be solved by

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

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.

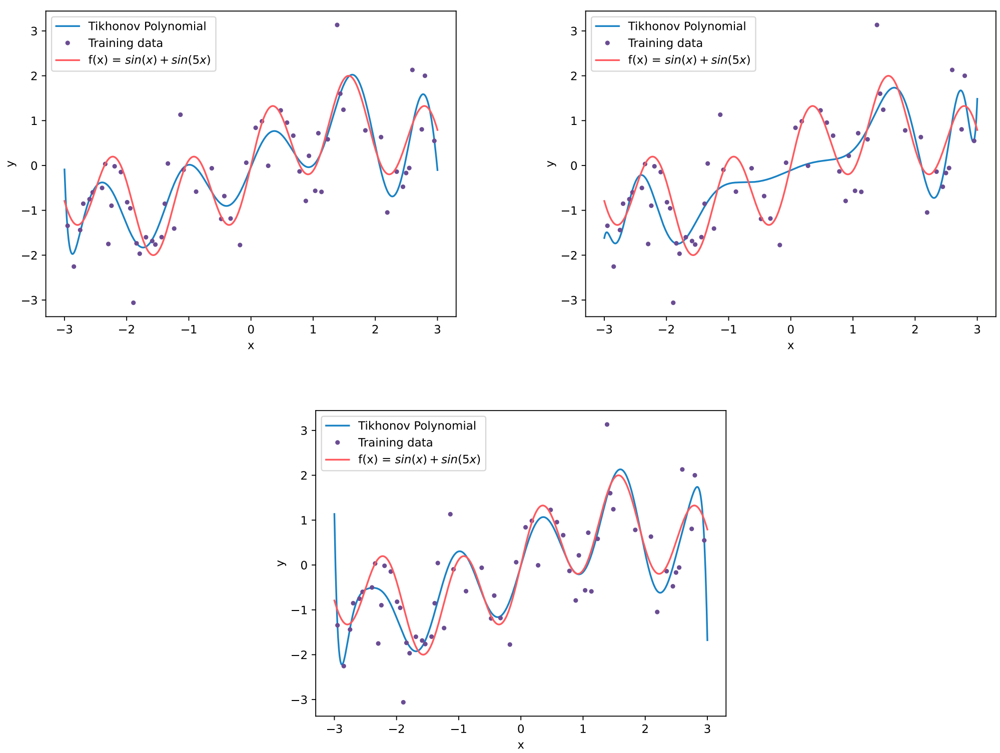

Figure 7: 15th degree Tikhonov fit to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$ (top left), $\lambda = 1$ (top right), and $\lambda = 0$ (bottom) for a single seed. The training data was composed of 60 points whose domain was randomly selected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

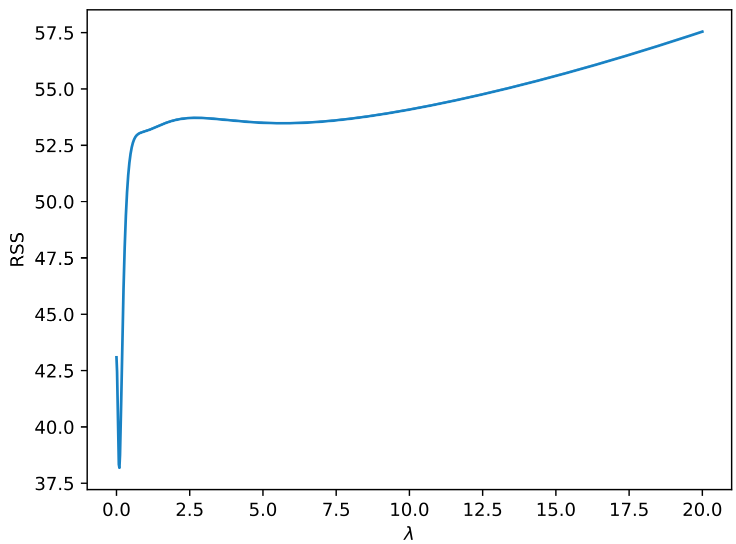

Figure 8: RSS from a 15th degree Tikhonov fit to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda \in [0, 20]$ for a single seed. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3, 3]$.

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.

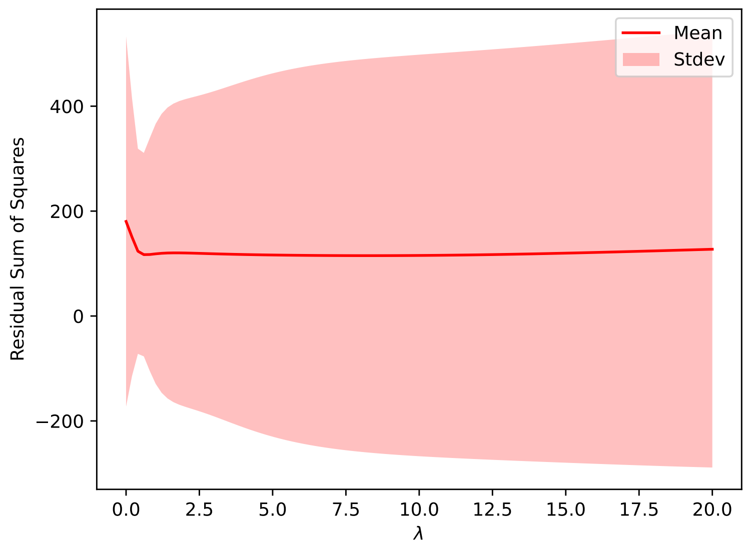

Figure 9: Mean and standard deviation of RSS from a 15th degree Tikhonov fit to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda \in [0, 20]$ for 100 unique seeds. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced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

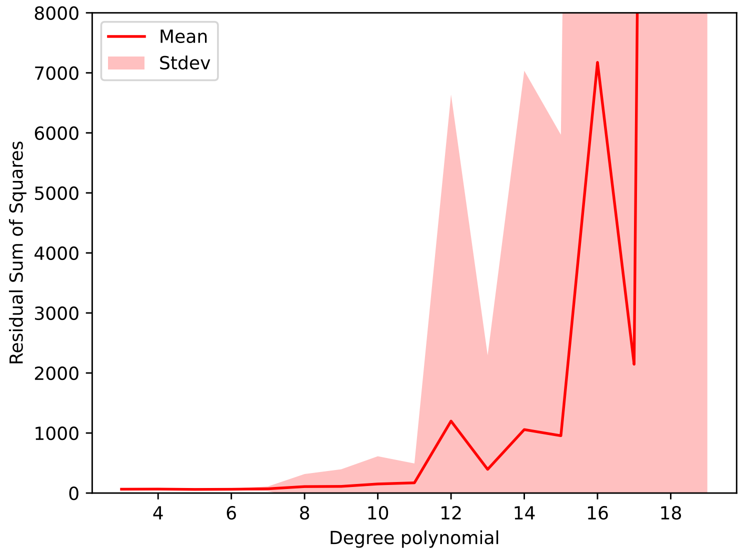

Figure 10: Mean and standard deviation of RSS from different degree 3 to degree 20 polynomial Tikhonov fits to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$. The training data was composed of 60 points whose domain was randomly selected from 120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

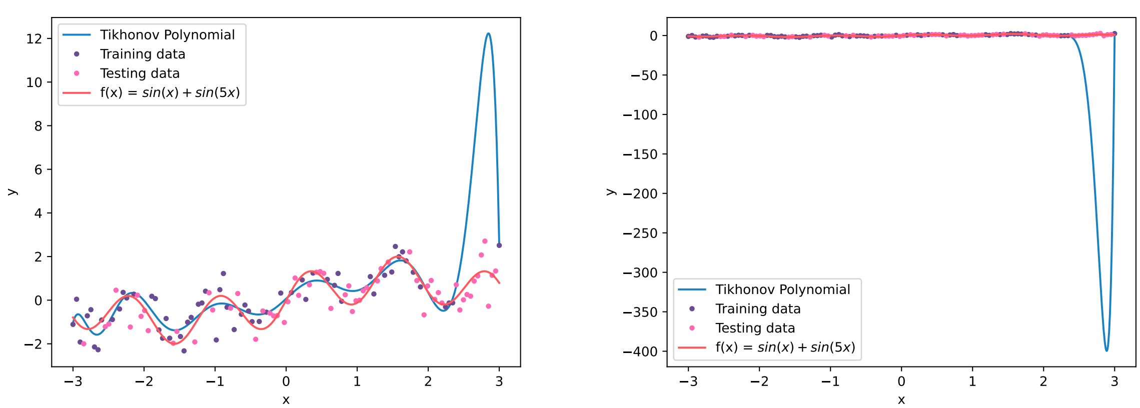

Figure 11: 15 degree (left) and 19 degree (right) Tikhonov fits to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$ for a single seed. The training data was composed of 60 points whose domain was randomly selected from 120 equispaced points on $[-3,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

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.

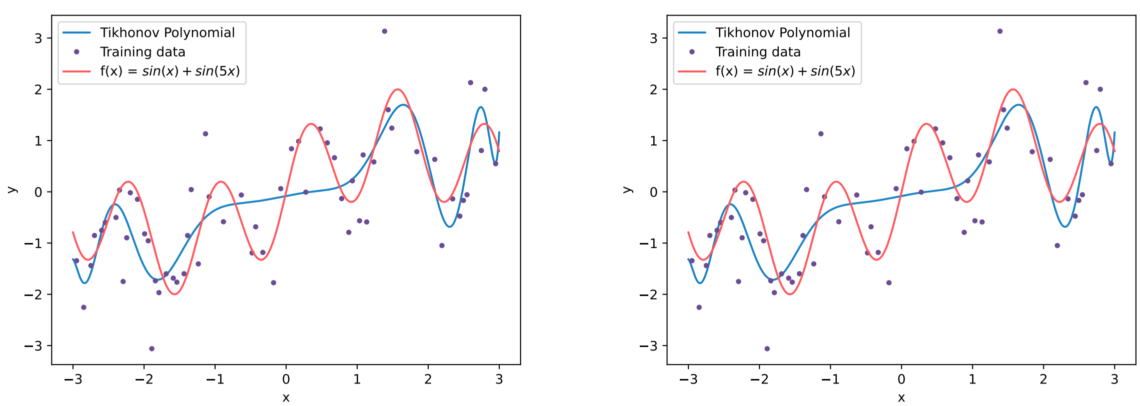

Figure 12: 15 degree Tikhonov fits to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$ for a single seed. The forward difference method was used in the left image and the backward difference method was used in the right image. The training data was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

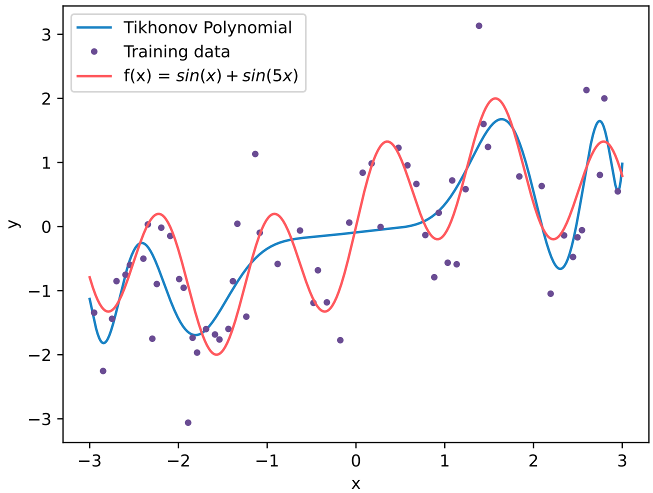

Figure 13: 15 degree Tikhonov fit to noisy function evaluations of $f(x) = \sin(x) + \sin(5x)$ on the interval $[-3, 3]$ for $\lambda = 0.1$ for a single seed. The second order centered difference method was used to create the weight matrix. The training data was composed of 60 points whose domain was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

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

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:

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

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 $\mathbf{X}$

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

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

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

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

  end for

end for

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

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 X are altered to have a mean of 0 and standard deviation of 1.
- $\tilde{y}_k^{(j)} = \sum_{l \neq j} x_{kl} \boldsymbol{\beta}_l$
- $S(a, b) = \begin{cases} a - b & \text{if } a > 0 \text{ and } b < |a| \\ a + b & \text{if } a < 0 \text{ and } b < |a| \\ 0 & \text{if } b \geq |a| \end{cases}$
- $|\mathbf{X}|$ 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.

λ \ α	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.

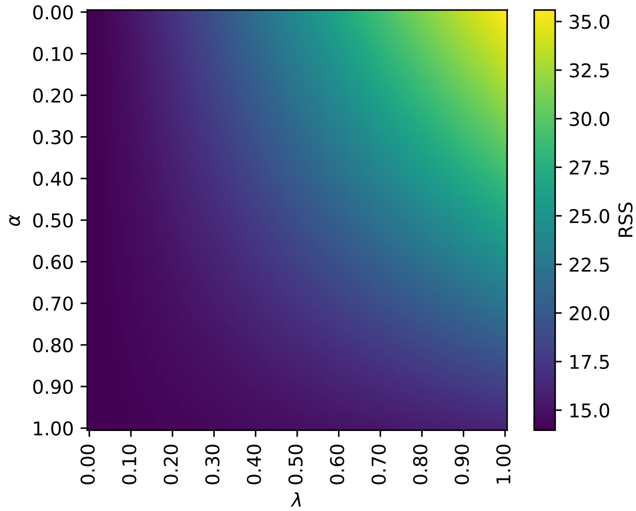

Figure 14: Elastic Net regression using 100 equispaced values of $\alpha \in \{0, 100\}$ and 100 equispaced values of $\lambda \in \{0, 100\}$ when fitting noisy function evaluations (Gaussian) of $y = 3x + 2$ to a one degree polynomial. Each combination of α and λ was evaluated for 100 different seeds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

λ \ α	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.

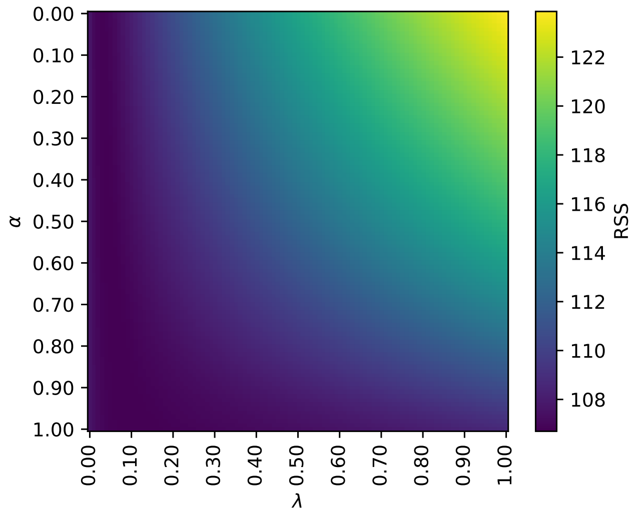

Figure 15: Elastic Net regression using 100 equispaced values of $\alpha \in \{0, 100\}$ and 100 equispaced values of $\lambda \in \{0, 100\}$ when fitting noisy function evaluations (Gaussian) of $y = x^2$ to a five degree polynomial. Each combination of α and λ was evaluated for 100 different seeds 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 $\boldsymbol{\beta}$ but avoid sparsity, in which case Ridge Regression would be an ideal choice. Perhaps control over the magnitude of $\boldsymbol{\beta}$ 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=0YFYt5C4N94C.

[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:

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

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:

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

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,

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

The above matrices are then expanded:

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

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

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

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

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

Now the derivatives of 11, 12, and 13 with respect to $\boldsymbol{\beta}$ are taken to get the derivative of 8 with respect to $\boldsymbol{\beta}$. 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\boldsymbol{\beta}}(\mathbf{X}\boldsymbol{\beta})^T\mathbf{X}\boldsymbol{\beta} &= \begin{bmatrix} \frac{d}{d\beta_0}(\mathbf{X}\boldsymbol{\beta})^T\mathbf{X}\boldsymbol{\beta} \\ \frac{d}{d\beta_1}(\mathbf{X}\boldsymbol{\beta})^T\mathbf{X}\boldsymbol{\beta} \\ \vdots \\ \frac{d}{d\beta_m}(\mathbf{X}\boldsymbol{\beta})^T\mathbf{X}\boldsymbol{\beta} \end{bmatrix} \\
&= \begin{bmatrix} 2(\beta_0+\beta_1x_0+\cdots+\beta_mx_0^m)+\ldots+2(\beta_0+\beta_1x_n+\cdots+\beta_mx_n^m) \\ 2x_0(\beta_0+\beta_1x_0+\cdots+\beta_mx_0^m)+\ldots+2x_n(\beta_0+\beta_1x_n+\cdots+\beta_mx_n^m) \\ \vdots \\ 2x_0^m(\beta_0+\beta_1x_0+\cdots+\beta_mx_0^m)+\ldots+2x_n^m(\beta_0+\beta_1x_n+\cdots+\beta_mx_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+\ldots+\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+\ldots+\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}+\ldots+\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 & \ldots & \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 & \ldots & \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 & \ldots & \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} & \ldots & \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 & \ldots & 1 \\ x_0 & x_1 & x_2 & \ldots & x_n \\ x_0^2 & x_1^2 & x_2^2 & \ldots & x_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & x_2^m & \ldots & x_n^m \end{bmatrix} \times \begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^m \\ 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & 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}\boldsymbol{\beta}
\end{aligned} \tag{14}
$$

Secondly, the derivatives of 12 are computed:

$$\frac{d}{d\boldsymbol{\beta}} 2\mathbf{y}^T\mathbf{X}\boldsymbol{\beta} = \begin{bmatrix} \frac{d}{da_0} 2\mathbf{y}^T\mathbf{X}\boldsymbol{\beta} \\ \frac{d}{da_1} 2\mathbf{y}^T\mathbf{X}\boldsymbol{\beta} \\ \vdots \\ \frac{d}{da_m} 2\mathbf{y}^T\mathbf{X}\boldsymbol{\beta} \end{bmatrix}$$

$$= 2 \times \begin{bmatrix} y_0 + y_1 + \ldots + y_n \\ y_0x_0 + y_1x_1 + \ldots + y_nx_n \\ y_0x_0^2 + y_1x_1^2 + \ldots + y_nx_n^2 \\ \vdots \\ y_0x_0^m + y_1x_1^m + \ldots + y_nx_n^m \end{bmatrix} \tag{15}$$

$$= 2 \times \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ x_0 & x_1 & x_2 & \ldots & x_n \\ x_0^2 & x_1^2 & x_2^2 & \ldots & x_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & x_2^m & \ldots & x_n^m \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

$$= 2\mathbf{X}^T\mathbf{y}$$

Lastly, the derivatives of 13 are computed:

$$\frac{d}{d\boldsymbol{\beta}} \gamma\boldsymbol{\beta}^T\boldsymbol{\beta} = \begin{bmatrix} \frac{d}{d\beta_0} \gamma\boldsymbol{\beta}^T\boldsymbol{\beta} \\ \frac{d}{d\beta_1} \gamma\boldsymbol{\beta}^T\boldsymbol{\beta} \\ \vdots \\ \frac{d}{d\beta_m} \gamma\boldsymbol{\beta}^T\boldsymbol{\beta} \end{bmatrix}$$

$$= \gamma \times \begin{bmatrix} 2\beta_0 \\ 2\beta_1 \\ 2\beta_2 \\ \vdots \\ 2\beta_m \end{bmatrix} \tag{16}$$

$$= 2 \times \gamma \times \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{bmatrix} \times \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix}$$

$$= 2\gamma\mathbf{I}\boldsymbol{\beta}$$

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:

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

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

6.1.2 Deriving the Tikhonov Estimator

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

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

So it follows that

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

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

So it follows that

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

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

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

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

So it follows that

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

So,

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

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]

$$f(\lambda x + (1-\lambda)y) \le f(\lambda x) + f((1-\lambda)(y)) = \lambda f(x) + (1-)f(y) \tag{20}$$

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 $\beta_0 = \frac{1}{N}\sum_{i=1}^{N}(y_i - x_i^T\beta)$

Proof:

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

6.2 Code

6.2.1 finite_diff.py

```
from elastic_net import ElasticNet
import matplotlib.pyplot as plt
import numpy as np
from tabulate import tabulate

class ElasticNetHelper:
    def __init__(self, f, degree, x_min, x_max, num_evals_x, num_train_x,
    noise = 0, verbose = False):
        '''
        `__init__`

        Initialize the plotting helper for the elastic net solver `
    ElasticNet`
```

```
        These parameters are meant to be constant across all generated elastic net solvers

        Parameters

        `f`: Function that we're fitting to
        `degree`: Degree fit we want
        `x_min`, `x_max`: Domain that we're fitting to
        `num_evals_x`: Number of equispaced points in the domain to use for training and validation data
        `num_train_x`: Number of randomly sampled points from `num_evals_x` to use for training, rest is for validation
        `noise`: Noise for each generated elastic net solver
        `verbose`: Optionally turn on (True) or off (False) extra prints (Defaults to 'False')

        Returns

        Nothing
        '''

        # Initialize member variables
        self.f = f
        self.degree = degree
        self.x_min = x_min
        self.x_max = x_max
        self.num_evals_x = num_evals_x
        self.num_train_x = num_train_x
        self.noise = noise
        self.verbose = verbose

        # Create our data
        self.x_eval = np.linspace(x_min, x_max, num_evals_x)

    def new_en_solver(self, alpha, _lambda, seed):
        '''
        `new_en_solver`

        Creates a new elastic net solve using the given seed

        Parameters

        `alpha`, `_lambda`: Hyperparameters for `ElasticNet`, see `elastic_net.py` for more details
        `seed`: Numpy seed

        Returns

        Nothing, but sets the current elastic net to this en
        '''

        # Set numpy seed
        np.random.seed(seed)

```

```
        # Create our splits
        self.x_train, self.x_val = self.get_split(self.x_eval, self.num_train_x)
        self.y_train = (self.f(self.x_train) + np.random.normal(scale=self.noise, size=self.x_train.shape))
        self.y_val = (self.f(self.x_val) + np.random.normal(scale=self.noise, size=self.x_val.shape))

        if self.verbose:
            print(f"x_train {self.x_train.shape}, x_val {self.x_val.shape}, num_evals_x {self.num_evals_x}")
            print(f"y_train {self.y_train.shape}, y_val {self.y_val.shape}, num_evals_x {self.num_evals_x}")

        # Create the elastic net solve and set it to the current one
        en = ElasticNet(self.x_train, self.y_train, self.degree, alpha, _lambda, verbose=self.verbose)
        self.en = en

    def get_split(self, x_eval, num_train_x):
        '''
        `get_split`

        Given the total data get the training and validation split

        Parameters

        x_eval: Total x-values
        num_train_x: Number of x-values in the training split

        Returns

        x_train: Training split
        x_val: Validation split
        '''

        # Create training split
        x_train = np.random.choice(x_eval, num_train_x, replace=False)
        x_train = np.sort(x_train)

        # Create validation split
        train_idx = 0
        x_val = list()
        for x in x_eval:
            if train_idx < num_train_x and x_train[train_idx] == x:
                train_idx += 1
            else:
                x_val += [x]
        x_val = np.array(x_val)

        return x_train, x_val

    def train(self, j):
        '''
```

```
        `train`

        Train the elastic net model j times over each variable

        Parameters

        j: Number of times to iterate over each variable for trainin

        Return Nothing
        '''

        self.en.iterate_coord_descent(j)

    def get_RSS(self, data):
        '''
        `get_RSS`

        Get current elastic net RSS

        Parameters

        data: Either 'val' or 'train' for validation or training data respectively

        Returns RSS
        '''

        if data == 'val':
            return self.en.get_RSS(self.x_val, self.y_val)
        elif data == 'train':
            return self.en.get_RSS(self.x_train, self.y_train)
        else:
            raise Exception("get_RSS: `data` must be either \'val\' or \'train\', but is \'{data}\'")

    def get_elastic_net(self, data):
        '''
        `get_elastic_net`

        Get current elastic net formula that's being minimized

        Parameters

        data: Either 'val' or 'train' for validation or training data respectively

        Returns elastic net formula
        '''

        if data == 'val':
            return self.en.get_elastic_net(self.x_val, self.y_val)
        elif data == 'train':
            return self.en.get_elastic_net(self.x_train, self.y_train)
        else:
```

```python
            raise Exception("get_elastic_net: `data` must be either \'val\' or \'train\', but is \'{data}\'")

    def get_weights(self):
        '''
        `get_weights`

        Returns current elastic net weights
        '''
        return self.en.get_b()

    def make_plot(self, num_true_x = 100, title = 'title', x_axis = 'x', y_axis = 'y', img_name = 'PLOT.pdf', img_dir = '../Images', train_data = True, val_data = True, f_plot = True, predict_f_plot = True):
        '''
        `make_plot`

        Makes plot with (optional) training data and (optional) validation data along with (optional) the true function and (optional) the predicted function

        Returns nothing, but makes our plot
        '''

        # Plotting colors
        color1 = '#FF595E'
        color2 = '#1982C4'
        color3 = '#6A4C93'
        color4 = 'green'

        # Construct save path
        save_path = f"{img_dir}/{img_name}"

        # Create helpful data
        true_x = np.linspace(self.x_min, self.x_max, num_true_x)
        true_y = self.f(true_x)

        # Create initial plots
        fig, ax = plt.subplots(1,1,figsize=(5,4), dpi=120, facecolor='white', tight_layout={'pad': 1})

        general_marker_style = dict(markersize = 1, markeredgecolor='black', marker='o', markeredgewidth=0)
        dot_marker_style = dict(markersize = 4, markeredgecolor='black', marker='*', markeredgewidth=0.75)
        data_marker_size = 2
        scatter_marker_size = 10

        # Create original function plot
        if f_plot:
            ax.plot(true_x, true_y, color=color1, label="Original function", **general_marker_style)

        # Create predicted function plot
```

```
        if predict_f_plot:
            pred_y = self.en.get_prediction(true_x)
            ax.plot(true_x, pred_y, color=color2, label="Elastic Net function", **general_marker_style)

        # Plot training data
        if train_data:
            ax.scatter(self.x_train, self.y_train, s=scatter_marker_size, color=color3, label=f"Training data")

        # Plot validation data
        if val_data:
            ax.scatter(self.x_val, self.y_val, s=scatter_marker_size, color=color4, label=f"Validation data")

        # Add labels
        if x_axis is not None:
            ax.set_xlabel(f"{x_axis}")
        if y_axis is not None:
            ax.set_ylabel(f"{y_axis}")
        if title is not None:
            ax.set_title(f"{title}")
        ax.legend()

        # Save image
        plt.savefig(f'{save_path}')
        plt.close()
        print(f'Saved to: {save_path}')

    def print_params(self):
        '''
        `print_params`

        Make a nice table of all the parameters of the class
        '''

        dict_tmp = dict()
        dict_tmp["Option"] = [
            "self.degree",
            "self.x_min",
            "self.x_max",
            "self.num_evals_x",
            "self.num_train_x",
            "self.noise",
            "self.verbose"
        ]
        dict_tmp["Description"] = [
            self.degree,
            self.x_min,
            self.x_max,
            self.num_evals_x,
            self.num_train_x,
            self.noise,
            self.verbose
```

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

6.2.2 finite_diff.py

```
from elastic_net_helper import ElasticNetHelper
from prints import *

f = lambda x: 3*x+2

for alpha, _lambda in [(0, 0.1), (1, 0.1), (0.5, 0.1), (0,0)]:

    small_banner(f"3x+2, alpha {alpha}, lambda {_lambda}", False, True)

    enp = ElasticNetHelper(f = f , degree = 1, x_min = -5, x_max = 5,\
                               num_evals_x = 20, num_train_x = 10,\
                               noise = 1, verbose = False)
    enp.print_params()
    print()

    enp.new_en_solver(alpha = alpha, _lambda = _lambda, seed = 0)

    print(f"Before training RSS {enp.get_RSS('train')}")
    print(f"Before training EN {enp.get_elastic_net('train')}")
    print(f"Before validation RSS {enp.get_RSS('val')}")
    print(f"Before validation EN {enp.get_elastic_net('val')}")
    print(f"Before weights")
    print(enp.get_weights())
    print()

    enp.train(10)
    enp.make_plot(num_true_x = 100, title = None, img_name=f"3x+2_a={alpha
    }_l={_lambda}.pdf")

    print(f"After training RSS {enp.get_RSS('train')}")
    print(f"After training EN {enp.get_elastic_net('train')}")
    print(f"After validation RSS {enp.get_RSS('val')}")
    print(f"After validation EN {enp.get_elastic_net('val')}")
    print(f"After weights")
    print(enp.get_weights())
    print()
```

6.2.3 finite_diff.py

```
from elastic_net_helper import ElasticNetHelper
from prints import *
import numpy as np
import matplotlib.pyplot as plt

f = lambda x: 3*x+2

save_name = "../Images/3x+2_matrix.pdf"

# Store RSS for each alpha and lamba
```

```
num_a = 101
num_l = 101
num_seeds = 100

alphas = np.linspace(0,1,num_a)
_lambdas = np.linspace(0,1,num_l)
RSS_mat = np.zeros((num_a,num_l))

# Get RSS values, keep track of minimum
min_RSS, min_alpha, min_lambda, min_weights = np.inf, -1, -1, None

small_banner(f"3x+2 matrix plot for {num_seeds} seeds", False, True)

for alpha_iter in range(len(alphas)):
    for _lambda_iter in range(len(_lambdas)):
        enp = ElasticNetHelper(f = f , degree = 1, x_min = -5, x_max = 5,\
                               num_evals_x = 20, num_train_x = 10,\
                               noise = 1, verbose = False)

        if alpha_iter == 0 and _lambda_iter == 0:
            enp.print_params()

        alpha = alphas[alpha_iter]
        _lambda = _lambdas[_lambda_iter]

        # Count all RSS for this seed
        RSS_sum = 0

        for seed in np.arange(0,num_seeds,1):
            enp.new_en_solver(alpha = alpha, _lambda = _lambda, seed =
    seed.item())
            enp.train(10)
            RSS_sum += enp.get_RSS('val')

        this_RSS = RSS_sum/num_seeds
        RSS_mat[alpha_iter][_lambda_iter] = this_RSS

        if this_RSS < min_RSS:
            min_RSS = this_RSS
            min_alpha = alpha
            min_lambda = _lambda
            min_weights = enp.get_weights()

# Print minima
print(f"Minimum RSS is {min_RSS} with alpha {min_alpha} and lambda {
    min_lambda}")
print(f"Minimum weights were:")
print(min_weights)
print()

# Make plot showing the result

fig, ax = plt.subplots(1,1,figsize=(5,4), dpi=120, facecolor='white',
    tight_layout={'pad': 1})
```

```python

# Convert alphas to printable version
alpha_label_locs= np.arange(0,num_a,10)
_lambda_label_locs= np.arange(0,num_l,10)

# Make sure last element is there
if num_a-1 not in alpha_label_locs:
    alpha_label_locs = np.append(alpha_label_locs, [num_a-1])

if num_l-1 not in _lambda_label_locs:
    _lambda_label_locs = np.append(_lambda_label_locs, [num_l-1])

alpha_labels= [f'{1.0 if a >= len(alphas) else alphas[a]:0.2f}' for a in
    alpha_label_locs]
_lambda_labels= [f'{1.0 if l >= len(_lambdas) else _lambdas[l]:0.2f}' for
    l in _lambda_label_locs]

a1 = ax.matshow(RSS_mat)
plt.colorbar(a1, label="RSS")
ax.set_xlabel("$\lambda$")
ax.set_ylabel("$\\alpha$")
ax.set_yticks(alpha_label_locs, alpha_labels, rotation=00)
ax.set_xticks(_lambda_label_locs, _lambda_labels, rotation=90)
ax.tick_params(labelbottom = True)
ax.tick_params(labeltop = False)
ax.tick_params(top = False)
#plt.show()
plt.savefig(save_name)
print(f"Saved figure to {save_name}")
```

6.2.4 finite_diff.py

```python
from elastic_net_helper import ElasticNetHelper
from prints import *

f = lambda x: x**2

for alpha, _lambda in [(0, 0.1), (1, 0.1), (0.5, 0.1), (0,0)]:

    small_banner(f"x^2, alpha {alpha}, lambda {_lambda}", False, True)

    enp = ElasticNetHelper(f = f , degree = 5, x_min = -5, x_max = 5,\
                            num_evals_x = 20, num_train_x = 10,\
                            noise = 1, verbose = False)
    enp.print_params()
    print()

    enp.new_en_solver(alpha = alpha, _lambda = _lambda, seed = 0)

    print(f"Before training RSS {enp.get_RSS('train')}")
    print(f"Before training EN {enp.get_elastic_net('train')}")
    print(f"Before validation RSS {enp.get_RSS('val')}")
    print(f"Before validation EN {enp.get_elastic_net('val')}")
    print(f"Before weights")
```

```python
    print(enp.get_weights())
    print()

    enp.train(10)
    enp.make_plot(num_true_x = 100, title = None, img_name=f"x^2_a={alpha}
   _l={_lambda}.pdf")

    print(f"After training RSS {enp.get_RSS('train')}")
    print(f"After training EN {enp.get_elastic_net('train')}")
    print(f"After validation RSS {enp.get_RSS('val')}")
    print(f"After validation EN {enp.get_elastic_net('val')}")
    print(f"After weights")
    print(enp.get_weights())
    print()
```

6.2.5 finite_diff.py

```python
from elastic_net_helper import ElasticNetHelper
from prints import *
import numpy as np
import matplotlib.pyplot as plt

f = lambda x: x**2

save_name = "../Images/x^2_matrix.pdf"

# Store RSS for each alpha and lamba
num_a = 101
num_l = 101
num_seeds = 100

alphas = np.linspace(0,1,num_a)
_lambdas = np.linspace(0,1,num_l)
RSS_mat = np.zeros((num_a,num_l))

# Get RSS values, keep track of minimum
min_RSS, min_alpha, min_lambda, min_weights = np.inf, -1, -1, None

small_banner(f"x^2 matrix plot for {num_seeds} seeds", False, True)

for alpha_iter in range(len(alphas)):
    for _lambda_iter in range(len(_lambdas)):
        enp = ElasticNetHelper(f = f , degree = 5, x_min = -5, x_max = 5,\
                               num_evals_x = 20, num_train_x = 10,\
                               noise = 1, verbose = False)

        if alpha_iter == 0 and _lambda_iter == 0:
            enp.print_params()

        alpha = alphas[alpha_iter]
        _lambda = _lambdas[_lambda_iter]

        # Count all RSS for this seed
        RSS_sum = 0
```

```python

        for seed in np.arange(0,num_seeds,1):
            enp.new_en_solver(alpha = alpha, _lambda = _lambda, seed = seed.item())
            enp.train(10)
            RSS_sum += enp.get_RSS('val')

        this_RSS = RSS_sum/num_seeds
        RSS_mat[alpha_iter][_lambda_iter] = this_RSS

        if this_RSS < min_RSS:
            min_RSS = this_RSS
            min_alpha = alpha
            min_lambda = _lambda
            min_weights = enp.get_weights()

# Print minima
print(f"Minimum RSS is {min_RSS} with alpha {min_alpha} and lambda {min_lambda}")
print(f"Minimum weights were:")
print(min_weights)
print()

# Make plot showing the result

fig, ax = plt.subplots(1,1,figsize=(5,4), dpi=120, facecolor='white', tight_layout={'pad': 1})

# Convert alphas to printable version
alpha_label_locs= np.arange(0,num_a,10)
_lambda_label_locs= np.arange(0,num_l,10)

# Make sure last element is there
if num_a-1 not in alpha_label_locs:
    alpha_label_locs = np.append(alpha_label_locs, [num_a-1])

if num_l-1 not in _lambda_label_locs:
    _lambda_label_locs = np.append(_lambda_label_locs, [num_l-1])

alpha_labels= [f'{1.0 if a >= len(alphas) else alphas[a]:0.2f}' for a in alpha_label_locs]
_lambda_labels= [f'{1.0 if l >= len(_lambdas) else _lambdas[l]:0.2f}' for l in _lambda_label_locs]

a1 = ax.matshow(RSS_mat)
plt.colorbar(a1, label="RSS")
ax.set_xlabel("$\lambda$")
ax.set_ylabel("$\\alpha$")
ax.set_yticks(alpha_label_locs, alpha_labels, rotation=00)
ax.set_xticks(_lambda_label_locs, _lambda_labels, rotation=90)
ax.tick_params(labelbottom = True)
ax.tick_params(labeltop = False)
ax.tick_params(top = False)
#plt.show()
```

```python
plt.savefig(save_name)
print(f"Saved figure to {save_name}")
```

6.2.6 finite_diff.py

```python
import numpy as np
from numpy.linalg import norm

class ElasticNet:
    def __init__(self, x_data, y_data, degree, alpha, _lambda, b_init = 0, verbose = False):
        '''
        `__init__`

        Initialize the Elastic Net solver

        Parameters

        x_data:  Numpy array of size (n+1,) for data x-values
        y_data:  Numpy array of size (n+1,) for data y-values
        degree:  Degree polynomial we're fitting to
        alpha:   See "Regularization Paths for Generalized Linear Models via Coordinate Descent" (2010)
        _lamdba: See "Regularization Paths for Generalized Linear Models via Coordinate Descent" (2010)
        verbose: (True) Enable / (False) disable print statements

        Returns

        Nothing
        '''

        # Store initial values
        self.x_data_initial = x_data
        self.y_data = y_data
        self.degree = degree
        self.alpha = alpha
        self._lambda = _lambda
        self.verbose = verbose

        # For ease, store value for number of data points
        self.N = self.x_data_initial.shape[0]

        # Create our X matrix from the paper
        self.X = self.create_X(self.x_data_initial, self.degree)

        # Standardize x values
        self.X, self.X_means, self.X_stds = self.standardize_X(self.X)

        # Make initial weights values
        # TODO: What should these be initialized to? Currently just doing zero
        self.b = np.ones(degree+1)*b_init

```

```
        # Get b0 term
        self.b[0] = self.get_intercept()

    def get_X(self):
        '''
        `get_X`

        Returns the X matrix
        '''
        return self.X

    def get_means(self):
        '''
        `get_means`

        Returns the means for the data matrix
        '''
        return self.X_means

    def get_stds(self):
        '''
        `get_stds`

        Returns the stds for the data matrix
        '''
        self.X_stds

    def standardize_X(self, X):
        '''
        `standardize_X`

        Normalizes X matrix per column
        Each column has mean zero and sum of squares divided by rows as 1

        Parameters

        X matrix which is standardized

        Returns

        Standardized X matrix, the column means, the column standard
    deviations
        '''

        means = np.mean(X, 0)
        stds = np.std(X, 0)

        X, means, stds = self.standardize_X_ms(X, means, stds)

        return X, means, stds

    def standardize_X_ms(self, X, means, stds):
        '''
        `standardize_X_ms`
```

```

        Normalizes X matrix per column given means and stds
        Each column has mean zero and sum of squares divided by rows as 1

        Parameters

        X matrix which is standardized
        Column means which are used in standardization
        Column standard deviations which are used in standardization

        Returns

        Standardized X matrix, the column means used, the column standard deviations used
        '''

        # Remove zeros from standard deviations
        for i in range(len(stds)):
            if stds[i] == 0. :
                stds[i] = 1.
        X = (X - means)/stds

        return X, means, stds

    def unstandardize_X(self, X):
        '''
        `unstandardize_X`

        Un-normalizes X matrix per column using the training means and standard deviations

        Parameters

        X which is to be unstandardized

        Returns

        Unstandardized input matrix X
        '''

        X = X*self.X_stds + self.X_means
        return X

    def get_b(self):
        '''
        `get_b`

        Returns current elastic net weights
        '''
        return self.b

    def get_prediction(self, x_eval):
        '''
        `get_prediction`
```

```
        Given non-standardized x-values, return our prediction for y-values, but using weights trained on standardized x-values

        Parameters

        x_eval: x-values we want our predictions at

        Returns

        y-values at those x-values
        '''

        # Create our X matrix from the paper
        X = self.create_X(x_eval, self.degree)

        # Standardize x values
        X, _, _ = self.standardize_X_ms(X, self.X_means, self.X_stds)

        # Get b0 term
        self.b[0] = self.get_intercept()

        # Get the y values for our X's
        y = X @ self.b

        return y + self.b[0]

    def get_intercept(self):
        '''
        `get_intercept`

        Returns

        Intercept formula obtained from Tyler
        '''

        b0 = sum((self.y_data - self.X@self.b)[1:])/self.N
        return b0

    def iterate_coord_descent(self, n):
        '''
        `iterate_coord_descent`

        Does n steps of coordinate descent for each weight b[1] to b[-1]

        Parameters

        n: Number of steps to do

        Returns

        Nothing, but updates weights in beta
        '''

```

```
        for _ in range(n):
            for j in range(1, self.degree+1):
                self.step_j(j)

    def step_j(self, j):
        '''
        `step_j`

        Does a coordinate descent step for variable j in beta
        j has to be nonzero since we're not optimizing the intercept

        This is equation 5 in
        "Regularization Paths for Generalized Linear Models via Coordinate
    Descent" (2010)

        Parameters

        j: Index into beta that we're optimizing

        Returns

        Nothing, but updates variable j in the weights
        '''

        if j <= 0:
            raise Exception(f"step_j: j ({j}) must be greater than 0")

        # Solve for y tilde (j) first
        y_tilde = np.sum(self.X*self.b,1)-self.X[:,j]*self.b[j]

        # First calculate sigma from equation (5)
        inner_sum = np.sum(self.X[:,j]*(self.y_data - y_tilde))

        # Then, divide it by N
        param_1 = inner_sum/self.N

        # Get second parameter for the soft-thresholding operator
        param_2 = self._lambda*self.alpha

        # Calculate numerator
        numerator = self.soft_thresholding(param_1, param_2)

        # Calculate denominator
        denominator = 1+self._lambda*(1-self.alpha)

        # Divide to get result
        res = numerator/denominator

        if self.verbose:
            print("y_tilde")
            print(y_tilde)
            print("y_data - y_tilde")
            print(self.y_data - y_tilde)
            print("inner_sum")
```

```
            print(inner_sum)
            print("param_1")
            print(param_1)
            print("param_2")
            print(param_2)
            print("numerator")
            print(numerator)
            print("denominator")
            print(denominator)
            print("res")
            print(res)

        self.b[j] = res

    def soft_thresholding(self, z, y):
        '''
        `soft_thresholding`

        This is the soft-thresholding operator, equation 6 in
        "Regularization Paths for Generalized Linear Models via Coordinate Descent" (2010)
        '''
        if y >= abs(z):
            return 0
        elif z > 0:
            return z - y
        else:
            return z + y

    def create_X(self, x_data, degree):
        '''
        `create_X`

        Creates the X matrix in our paper

        Parameters

        x_data: Numpy array of size (n+1,) for data x-values
        degree: Degree polynomial we're fitting to

        Returns

        X matrix as described in our paper
        '''
        X = np.zeros((x_data.shape[0], degree+1))
        for col in range(degree+1):
            X[:,col] = np.power(x_data,col)
        return X

    def get_elastic_net(self, x_data, y_data):
        '''
        `get_elastic_net`

        Gets the elastic net value, formula 1 in
```

```
        "Regularization Paths for Generalized Linear Models via Coordinate
 Descent" (2010)
        that we're trying to minimize

        Parameters

        x_data, y_data: (x,y) points that we're calculating the residual
    sum of squares for

        Returns

        Elastic net formula that is being minimized (formula 1 in the
    above paper)
        '''

        # Calculate RSS term
        RSS = self.get_RSS(x_data, y_data)

        # Calculate regularization term
        P = (1-self.alpha)*norm(self.b[1:],2)**2/2 + self.alpha*norm(self.
    b[1:],1)

        # Return function elastic net is trying to minimize
        return RSS + P

    def get_RSS(self, x_data, y_data):
        '''
        `get_RSS`

        Gets the Residual Sum of Squares

        Parameters

        x_data, y_data: (x,y) points that we're calculating the residual
    sum of squares for

        Returns

        Elastic net formula that is being minimized (formula 1 in the
    above paper)
        '''

        # Create our X matrix and standardize it
        X = self.create_X(x_data, self.degree)
        X, _, _ = self.standardize_X(X)

        # Calculate RSS term
        RSS = sum(np.power(np.sum(X*self.b,1)-y_data,2))
        RSS = RSS/self.N

        # Return Residual Sum of Squares
        return RSS
```

6.2.7 estimators.py

```
import numpy as np
import finite_diff

class ridge:
  def __init__(self, gamma = 0, degree = 1):
    #gamma is regularization constat, degree is degree of polynomial
    self.gamma = gamma
    self.degree = degree
    self.E_ridge = None

  # Construct A matrix
  def construct_A(self, input_x):
    A_matrix = np.ones((input_x.shape[0], self.degree + 1))
    for i in range(A_matrix.shape[1] - 1):
      A_matrix[:,i + 1] = np.power(input_x, i + 1)
    return A_matrix

  #Use the closed form solution of the ridge estimator to find solution
  def fit(self, train_x, train_y):
    A_matrix = self.construct_A(train_x)
    b = train_y
    self.E_ridge = np.linalg.inv(np.transpose(A_matrix)@A_matrix + self.
   gamma*np.identity(A_matrix.shape[1]))@np.transpose(A_matrix)@b

  #Predict y values for given x values after estimator has been fitted
  def predict(self, input_x):
    A_test = self.construct_A(input_x)
    b_hat = A_test@self.E_ridge
    return b_hat

  #Caclulate RSS for some validation x and validation y
  def RSS(self, valid_x, valid_y):
    b_hat = self.predict(valid_x)
    rss = np.sum(np.power((b_hat - valid_y),2))
    return rss


class tikhonov:
  ## Weights needs to be a list that will work for implementation of
   finite_diff.py
  def __init__(self, _lambda, degree, weights):
    self._lambda = _lambda
    self.degree = degree
    self.xstar = None
    self.weight_matrix = weights

  def construct_A(self, in_x):
    A = np.ones((in_x.shape[0], self.degree + 1))
    for i in range(self.degree):
      A[:, i + 1] = np.power(in_x, i + 1)
    return A

```

```python
  def fit(self, train_x, train_y):
    A = self.construct_A(train_x)
    b = train_y
    D = self.weight_matrix
    self.xstar = np.linalg.inv((np.transpose(A) @ A + self._lambda**2 * np.transpose(D) @ D)) @ np.transpose(A) @ b

  def predict(self, test_x):
    A_test = self.construct_A(test_x)
    b_hat = A_test @ self.xstar
    return b_hat

  def RSS(self, test_x, test_y):
    b_hat = self.predict(test_x)
    rss = np.sum(np.power((b_hat - test_y), 2))
    return rss
```

6.2.8 finite_diff.py

```python
import numpy as np
import matplotlib.pyplot as plt

def generate_D(FD_list):
    '''
    Generates the derivative matrix `D` given a list `FD_list` that specifies the type of finite difference method for each row

    input:\n
    `FD_list`: A list of tuples of the form (FD, N), where FD is the FD option with stride N (see below) for each row

    ouptut:\n
    A matrix `D` with the values filled in from `FD_list`

    finite difference (FD) options for various strides (N): \n
    0: empty row
    1: centered difference            `f'(x) ~   (f(x-N)-f(x-N))/(2*N)`
    2: forward difference             `f'(x) ~   (f(x+N) - f(x))/N`
    3: backward difference            `f'(x) ~   (f(x) - f(x-N))/N`
    4: centered difference 2nd order  `f''(x) ~  (f(x+N)-2*f(x)+f(x-N))/N**2`
    5: forward difference 2nd order   `f''(x) ~  (f(x+2*N)-2*f(x+N)+f(x))/N**2`
    6: backward difference 2nd order  `f''(x) ~  (f(x)-2*f(x-N)+f(x-2*N))/N**2`
    '''

    # Size of our D matrix
    size = len(FD_list)

    # Initialize the D matrix
    D = np.zeros((size, size))

    # Populate D matrix
```

```python
    for i in range(size):
        # Extract values from FD_list
        FD = FD_list[i][0]
        N = FD_list[i][1]

        # Get the row we're inserting and a helper value for padding (see `_get_FD`'s `diff`)
        row, diff = _get_FD(FD, N)

        # Calculate padding
        padding_before = i - diff
        padding_after = size - padding_before - len(row)

        # Verify the padding makes sense
        if padding_before < 0 or padding_after < 0:
            raise Exception(f"generate_D: Row {i} of the matrix D has negative padding: FD {FD}, N {N}, padding_before {padding_before}, padding_after {padding_after}")

        if padding_before + padding_after + len(row) != size:
            raise Exception(f"generate_D: Row {i} of the matrix D is not the right size: FD {FD}, N {N}, row size {len(row)}")

        # Generate padded row with zeros prepended
        row = np.pad(row, (padding_before, padding_after), constant_values=0)

        # Replace row D with our row
        D[i,:] = row

    return D

def _get_FD(FD, N):
    '''
    Helper function to get the smallest row-matrix for the finite difference method and calculate how many indices behind the current x we're at (used in `generate_D`) \n

    input:\n
    `FD`: Finite difference method
    `N`: Stride length

    output:\n
    Tuple (row, diff)
    `row`: The finite difference row
    `diff`: Number of values before the current x that to start of the row

    (ie. for centered difference with an N of 2, return [-1/4, 0, 0, 0, 1/4])
    '''
    if N < 1 or not isinstance(N, int):
        raise Exception(f"get_FD: N ({N}) must be a positive integer")

    if FD not in [0,1,2,3,4,5,6]:
```

```
        raise Exception(f"get_FD: FD ({FD}) must be an integer in [0,6]")

    if FD == 0:
        ret = np.zeros(0)
        return (ret, 0)
    if FD == 1:
        ret = np.zeros(2*N+1)
        ret[0] = -1/(2*N)
        ret[-1] = 1/(2*N)
        return (ret, N)
    if FD == 2 or FD == 3:
        # Row is the same, but its position changes on the method
        ret = np.zeros(N+1)
        ret[0] = -1/(N)
        ret[-1] = 1/(N)

        if FD == 2:
            return (ret, 0)
        else:
            return (ret, N)
    if FD == 4 or FD == 5 or FD == 6:
        # Row is the same, but its position changes on the method
        ret = np.zeros(2*N+1)
        ret[-1] = 1/(N**2)
        ret[N] = -2/(N**2)
        ret[0] = 1/(N**2)

        if FD == 4:
            return (ret, N)
        elif FD == 5:
            return (ret, 0)
        else:
            return (ret, 2*N)

def generate_centered_D(N):
    N -= 2
    FD_list = [(0, 1)] + [(1,1)]*N + [(0, 1)]
    D = generate_D(FD_list)
    D = D[1:-1]
    return D

def generate_forward_D(N):
    N -= 1
    FD_list = [(2, 1)] * (N) + [(0, 1)]
    D = generate_D(FD_list)
    return D

def generate_backwards_D(N):
    N -= 1
    FD_list = [(0,1)] + [(3, 1)] * (N)
    D = generate_D(FD_list)
    return D

def generate_2nd_centered_D(N):
```

```
    N -= 2
    FD_list = [(0, 1)] + [(4,1)]*N + [(0, 1)]
    D = generate_D(FD_list)
    D = D[1:-1]
    return D


def test_D_1():
    '''
    Test function to make sure D is created properly
    '''
    FD_list = [(0, 1), (1, 1), (1, 1), (6, 1), (4, 1), (5, 1), (1, 1), (1,
     1), (0, 1)]
    D = generate_D(FD_list)
    plt.matshow(D)
    plt.show()

def test_D_2():
    '''
    Test function to make sure D is created properly
    '''
    N = 5
    D = generate_centered_D(N)
    plt.matshow(D)
    plt.show()

#test_D_1()
#test_D_2()
```

6.2.9 estimators.py

```
import numpy as np
import matplotlib.pyplot as plt
import estimators

# Folder to save images in
save_dir = '../Images/'

# Seed
np.random.seed(10)

# Option to either show or save the image
save_fig = True

# Create plot 1: Linear
f_original = lambda x: x # Non-noisy function
sigma = 0.5 # Noise (standard deviation of gaussian noise)
f_noisy = lambda x: f_original(x) + np.random.normal(0,sigma,x.shape) #
    Non-noisy function

x_min = -5
x_max = 5

num_pts_original = 1000 # Points for non-noisy function
```

```
num_pts_noisy = 100 # Noisy points

x_eval_original = np.linspace(x_min,x_max,num_pts_original)
x_eval_noisy = np.linspace(x_min,x_max,num_pts_noisy)

y_eval_original = f_original(x_eval_original)
y_eval_noisy = f_noisy(x_eval_noisy)

plt.plot(x_eval_original,y_eval_original,color='black',label="Original Function")
plt.scatter(x_eval_noisy,y_eval_noisy,s=5,color='red',label="Noisy Function")
plt.legend()

if save_fig:
    plt.savefig(save_dir + "intro_1.pdf")
else:
    plt.show()
plt.close()

# Create plot 2: Cubic
f_original = lambda x: x**3 # Non-noisy function
sigma = 10 # Noise (standard deviation of gaussian noise)
f_noisy = lambda x: f_original(x) + np.random.normal(0,sigma,x.shape) # Non-noisy function
f_quad = lambda x: -1*6*(x+0.5)**2

x_min = -5
x_max = 5

num_pts_original = 1000 # Points for non-noisy function
num_pts_noisy = 5 # Noisy points

x_eval_original = np.linspace(x_min,x_max,num_pts_original)
x_eval_noisy = np.random.uniform(x_min,x_max,num_pts_noisy)

y_eval_original = f_original(x_eval_original)
y_eval_noisy = f_noisy(x_eval_noisy)

R2 = estimators.ridge(gamma=0, degree=2)

R2.fit(x_eval_noisy,y_eval_noisy)
y_eval_d2 = R2.predict(x_eval_original)

plt.plot(x_eval_original,y_eval_original,color='black',label="Original Function")
plt.plot(x_eval_original,y_eval_d2,color='blue',linestyle='dotted',label="Degree 2 fit")
plt.scatter(x_eval_noisy,y_eval_noisy,s=5,color='red',label="Noisy Function")
plt.ylim(-150,150)
plt.legend()

if save_fig:
```

```python
    plt.savefig(save_dir + "intro_2.pdf")
else:
    plt.show()
plt.close()

# Create plot 3: Cubic
f_original = lambda x: x**3 # Non-noisy function
sigma = 10 # Noise (standard deviation of gaussian noise)
f_noisy = lambda x: f_original(x) + np.random.normal(0,sigma,x.shape) #
    Non-noisy function
#f_quad = lambda x: -1*6*(x+0.5)**2

x_min = -5
x_max = 5

num_pts_original = 100 # Points for non-noisy function
num_pts_noisy = 5 # Noisy points

x_eval_original = np.linspace(x_min,x_max,num_pts_original)
x_eval_noisy = np.random.uniform(x_min,x_max,num_pts_noisy)

deg = 4
R2 = estimators.ridge(gamma=0, degree=deg)

y_eval_original = f_original(x_eval_original)
y_eval_noisy = f_noisy(x_eval_noisy)

R2.fit(x_eval_noisy,y_eval_noisy)
y_eval_d2 = R2.predict(x_eval_original)

plt.plot(x_eval_original,y_eval_original,color='black',label="Original
    Function")
plt.scatter(x_eval_noisy,y_eval_noisy,s=5,color='red',label="Noisy
    Function")
plt.plot(x_eval_original,y_eval_d2,color='blue',linestyle='dotted',label=f
    "Degree {deg} fit")
plt.ylim(-150,150)
plt.legend()

if save_fig:
    plt.savefig(save_dir + "intro_3.pdf")
else:
    plt.show()
plt.close()
```

6.2.10 estimators.py

```python
'''
prints.py

Contains helper functions for printing things
'''

def small_banner(string, before_space = False, after_space = False):
```

```
    '''
    `small_banner`

    Prints something like:
    ############
    # <string> #
    ############

    and adds before/after spacing depending on inputs (False by default)
    '''
    if before_space:
        print()

    new_string = "# " + string + " #"
    length = len(new_string)
    print("#"*length)
    print(new_string)
    print("#"*length)

    if after_space:
        print()
```

6.2.11 ridge_2a_driver.py

```
import numpy as np
import matplotlib.pyplot as plt
import estimators
import sample

# Save or show plots
save_plots = True

#set colors
color1 = '#FF595E'
color2 = '#1982C4'
color3 = '#6A4C93'
colors =[color1, color2, color3]

#no interval is given of where to sample f, choose [-5, 5]
#choose function and number of samples
f = lambda x: 3*x + 2
a = -5
b = 5
number_of_samples = 20
number_of_train_samples = 10

xeval = np.linspace(-5,5,100)
degree = 1
seed = 50
#get random sample, and divide into training and validation data

train_x, train_y, valid_x, valid_y = sample.random_sample_equi(
    number_of_samples, f, a, b, number_of_train_samples, seed = seed)

```

```python
fig_initial, ax_initial = plt.subplots(1,1)
ax_initial.plot(xeval, f(xeval), label = 'f(x) = 3x + 2', color = color3)
ax_initial.plot(train_x, train_y,'.', color ='green', label = 'Training
    Data')
ax_initial.legend()
ax_initial.set_xlabel('x')
ax_initial.set_ylabel('y')

if save_plots:
    plt.savefig("../Images/2a_only_data.pdf")
else:
    plt.show()

#make graph for gammas = 0, 0.1, calculate RSS, seed = 50
gammas_initial = [0,0.1]
linestyles = ['-', '--']
rss_initial = []
counter = 0
for gamma in gammas_initial:
    ridge = estimators.ridge(gamma, degree)
    ridge.fit(train_x, train_y)
    rss_initial.append(ridge.RSS(valid_x, valid_y))
    ax_initial.plot(xeval, ridge.predict(xeval),linestyles[counter], label
    = '$\gamma$ = ' + str(gamma), color = colors[counter])
    counter += 1
ax_initial.legend()
print(rss_initial)

if save_plots:
    plt.savefig("../Images/2a_initial_gammas.pdf")
else:
    plt.show()
plt.close()

#make RSS graph for gammas between 0 and 50 for seed = 50
gammas_log = np.linspace(0,50,1000)
fig_log10, ax_log10 = plt.subplots(1,1)
rss_log10 = []
for gamma in gammas_log:
    ridge = estimators.ridge(gamma, degree)
    ridge.fit(train_x, train_y)
    rss_log10.append(ridge.RSS(valid_x, valid_y))
ax_log10.plot(gammas_log, rss_log10, color = color2)
ax_log10.set_xlabel('$\gamma$')
ax_log10.set_ylabel('Residual Sum of Squares')

if save_plots:
    plt.savefig("../Images/2a_seed50_gammas.pdf")
else:
    plt.show()
plt.close()

#Make RSS graph for gammas between 0 and 50 for seeds 1-100
fig, ax = plt.subplots(1,1)
```

```
seed_list = range(1,101)
gammas = np.linspace(0,50,1000)
mean_std_mat = np.zeros((len(seed_list),len(gammas)))
gammas_best = []
for i in range(len(seed_list)):
    seed = seed_list[i]
    train_x, train_y, valid_x, valid_y = sample.random_sample_equi(
   number_of_samples, f, a, b, number_of_train_samples, seed = seed)
    rss = []
    for gamma in gammas:
        ridge = estimators.ridge(gamma, degree)
        ridge.fit(train_x, train_y)
        rss.append(ridge.RSS(valid_x, valid_y))
    gammas_best.append(gammas[np.argmin(rss)])
    mean_std_mat[i,:] = rss
    ax.plot(gammas, rss, alpha = 0.2)

ax.set_xlabel('$\gamma$')
ax.set_ylabel('log10 of Residual Sum of Squares')
ax.set_yscale('log')

if save_plots:
    plt.savefig("../Images/2a_seeds1_100.pdf")
else:
    plt.show()
plt.close()

# Calculate mean and stdev across different seeds
means = np.mean(mean_std_mat,axis=0)
best_mean = np.min(means)
best_gamma = gammas[np.argmin(means)]
stdevs = np.std(mean_std_mat,axis=0)

# Plot our mean and stdev plots
plt.plot(gammas, means, color="red", label="Mean")
plt.fill_between(gammas, means-stdevs,\
                 means+stdevs,\
                 color="red", alpha=0.25, edgecolor=None, label="Stdev")
plt.semilogy()
plt.legend()
plt.xlabel('$\gamma$')
plt.ylabel('log10 of Residual Sum of Squares')

if save_plots:
    plt.savefig("../Images/2a_seeds1_100mean.pdf")
else:
    plt.show()
plt.close()

plt.plot(gammas, means, color="red", label="Mean")
plt.fill_between(gammas, means-stdevs,\
                 means+stdevs,\
                 color="red", alpha=0.25, edgecolor=None, label="Stdev")
plt.xlim(0,1.5)
```

```
plt.ylim(11.25,12)
plt.legend()

if save_plots:
    plt.savefig("../Images/2a_seeds1_100mean_zoomed.pdf")
else:
    plt.show()
plt.close()

nonzero_gammas_count = np.count_nonzero(gammas_best)
print('Best gamma was ' + str(best_gamma))
print('Min RSS was ' + str(best_mean))
print('We had this many 0 gammas' + str(len(gammas_best) - nonzero_gammas_count))
print('We had this many nonzero gammas' + str(nonzero_gammas_count))
print('Percent nonzero ' + str(nonzero_gammas_count/len(gammas_best)))
print('Percent 0 gammas ' +str(1 - (nonzero_gammas_count/len(gammas_best))))
print('Mean Best Gamma = ' + str(np.mean(gammas_best)))
```

6.2.12 ridge_2b_driver.py

```
import numpy as np
import matplotlib.pyplot as plt
import estimators
import sample

# Save or show plots
save_plots = True

#set colors
color1 = '#FF595E'
color2 = '#1982C4'
color3 = '#6A4C93'
colors =[color1, color2, color3]

#no interval is given of where to sample f, choose [-5, 5]
#choose function and number of samples
f = lambda x: x**2
a = -5
b = 5
number_of_samples = 20
number_of_train_samples = 10

xeval = np.linspace(-5,5,1000)
degree = 5
seed = 50
#get random sample, and divide into training and validation data

train_x, train_y, valid_x, valid_y = sample.random_sample_equi(number_of_samples, f, a, b, number_of_train_samples, seed = seed)

#Make Graphs for gammas = 0, 0.1 and calculate RSS, seed = 50
gammas_initial = [0,0.1]
```

```
linestyles = ['-', '--']
fig_initial, ax_initial = plt.subplots(1,1)
rss_initial = []
counter = 0
for gamma in gammas_initial:
    ridge = estimators.ridge(gamma, degree)
    ridge.fit(train_x, train_y)
    rss_initial.append(ridge.RSS(valid_x, valid_y))
    ax_initial.plot(xeval, ridge.predict(xeval),linestyles[counter], label
    = '$\gamma$ = ' + str(gamma), color = colors[counter])
    counter += 1
ax_initial.plot(xeval, f(xeval), label = 'f(x) = $x^2$', color = color3)
ax_initial.plot(train_x, train_y,'.', color ='green', label = 'Training
    Data')
ax_initial.legend()
ax_initial.set_xlabel('x')
ax_initial.set_ylabel('y')
print(rss_initial)

if save_plots:
    plt.savefig("../Images/2b_initial_gammas.pdf")
else:
    plt.show()
plt.close()

#make RSS graph for gammas between 0 and 50 for seed = 50
gammas_log = np.linspace(0,50,1000)
fig_log10, ax_log10 = plt.subplots(1,1)
rss_log10 = []
for gamma in gammas_log:
    ridge = estimators.ridge(gamma, degree)
    ridge.fit(train_x, train_y)
    rss_log10.append(ridge.RSS(valid_x, valid_y))
ax_log10.plot(gammas_log, rss_log10, color = color2)
ax_log10.set_xlabel('$\gamma$')
ax_log10.set_ylabel('Residual Sum of Squares')

if save_plots:
    plt.savefig("../Images/2b_seed50_gammas.pdf")
else:
    plt.show()
plt.close()

#Make RSS graph for gammas between 0 and 50 for seeds 1-100
fig, ax = plt.subplots(1,1)
seed_list = range(1,101) #iterate through all seeds
gammas = np.linspace(0,50,1000)
mean_std_mat = np.zeros((len(seed_list),len(gammas)))
gammas_best = []
for i in range(len(seed_list)):
    seed = seed_list[i]
    train_x, train_y, valid_x, valid_y = sample.random_sample_equi(
    number_of_samples, f, a, b, number_of_train_samples, seed = seed)
    rss = []
```

```
    for gamma in gammas:
        ridge = estimators.ridge(gamma, degree)
        ridge.fit(train_x, train_y)
        rss.append(ridge.RSS(valid_x, valid_y))
    gammas_best.append(gammas[np.argmin(rss)])
    mean_std_mat[i,:] = rss
    ax.plot(gammas, rss, alpha = 0.2)

ax.set_xlabel('$\gamma$')
ax.set_ylabel('log10 of Residual Sum of Squares')
ax.set_yscale('log')

if save_plots:
    plt.savefig("../Images/2b_seeds1_100.pdf")
else:
    plt.show()
plt.close()

# Calculate mean and stdev across different seeds
means = np.mean(mean_std_mat,axis=0)
best_mean = np.min(means)
best_gamma = gammas[np.argmin(means)]
stdevs = np.std(mean_std_mat,axis=0)

# Plot our mean and stdev plots
plt.plot(gammas, means, color="red", label="Mean")
plt.fill_between(gammas, means-stdevs,\
                 means+stdevs,\
                 color="red", alpha=0.25, edgecolor=None, label="Stdev")
plt.semilogy()
plt.xlabel('$\gamma$')
plt.ylabel('log10 of Residual Sum of Squares')
plt.legend()

if save_plots:
    plt.savefig("../Images/2b_seeds1_100mean.pdf")
else:
    plt.show()
plt.close()

nonzero_gammas_count = np.count_nonzero(gammas_best)
print('Best gamma was ' + str(best_gamma))
print('Min RSS was ' + str(best_mean))
print('We had this many 0 gammas ' + str(len(gammas_best) - nonzero_gammas_count))
print('We had this many nonzero gammas ' + str(nonzero_gammas_count))
print('Percent nonzero ' + str(nonzero_gammas_count/len(gammas_best)))
print('Percent 0 gammas ' +str(1 - (nonzero_gammas_count/len(gammas_best))))
print('Mean Best Gamma = ' + str(np.mean(gammas_best)))
```

6.2.13 tikhonov.py

```
import numpy as np
```

```


#this function was consolidated into sample.py, use that file for future use/reference
def random_sample_equi(number_of_samples, f, a, b, number_of_train_samples, mean = 0, std_dev = 1, seed = None):


    rng = np.random.default_rng(seed)
    sample_x = np.linspace(a,b,number_of_samples)
    sample_x = np.reshape((number_of_samples, 1))
    gaussian_noise = rng.normal(mean, std_dev, (number_of_samples, 1))
    sample_y = f(sample_x) + gaussian_noise
    sample_x_y = np.concatenate((sample_x, sample_y), axis = 1)

    train_data = rng.choice(sample_x_y, size = number_of_train_samples, replace = False)
    valid_data = np.zeros((number_of_samples - number_of_train_samples, 2))

    counter = 0
    for i in range(sample_x_y.shape[0]):
        include = True
        for j in range(train_data.shape[0]):
            if sample_x_y[i,0] == train_data[j,0]:
                include = False
        if include == True:
            valid_data[counter] = sample_x_y[i]
            counter += 1

    train_data = train_data[train_data[:,0].argsort()]
    valid_data = valid_data[valid_data[:,0].argsort()]

    train_x = train_data[:,0]
    train_y = train_data[:,1]
    valid_x = valid_data[:,0]
    valid_y = valid_data[:,1]


    return train_x, train_y, valid_x, valid_y
```

6.2.14 sample.py

```
import numpy as np


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

```
    '''

    rng = np.random.default_rng(seed) #set seed
    sample_x = rng.uniform(a,b, (number_of_samples, 1)) #get x values
    gaussian_noise = rng.normal(mean, std_dev, (number_of_samples, 1))
    sample_y = f(sample_x) + gaussian_noise #add noise
    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

    train_data = rng.choice(sample_x_y, size = number_of_train_samples, replace = False) #find train data
    valid_data = np.zeros((number_of_samples - number_of_train_samples, 2))

    counter = 0 #find validation data
    for i in range(sample_x_y.shape[0]):
        include = True
        for j in range(train_data.shape[0]):
            if sample_x_y[i,0] == train_data[j,0]:
                include = False
        if include == True:
            valid_data[counter] = sample_x_y[i]
            counter += 1

    train_data = train_data[train_data[:,0].argsort()]#sort data so plotting is nice
    valid_data = valid_data[valid_data[:,0].argsort()]

    train_x = train_data[:,0]
    train_y = train_data[:,1]
    valid_x = valid_data[:,0]
    valid_y = valid_data[:,1]


    return train_x, train_y, valid_x, valid_y

def random_sample_equi(number_of_samples, f, a, b, number_of_train_samples, mean = 0, std_dev = 1, seed = None):
    '''
    Take in Number of Samples, Function, end points, how many samples should be training samples, mean, std_dev, and random seed
    x values are equispaced from [a,b], adds Gaussian Noise to y values
    Returns training x, training y, valid x, and valid y
    '''

    rng = np.random.default_rng(seed) # set seed
    sample_x = np.linspace(a,b,number_of_samples) #get x values
    sample_x = np.reshape(sample_x, (number_of_samples, 1))
    gaussian_noise = rng.normal(mean, std_dev, (number_of_samples, 1))
    sample_y = f(sample_x) + gaussian_noise #add noise
    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

    train_data = rng.choice(sample_x_y, size = number_of_train_samples,
```

```
    replace = False) #find train data
    valid_data = np.zeros((number_of_samples - number_of_train_samples, 2)
    )

    counter = 0 #find validation data
    for i in range(sample_x_y.shape[0]):
        include = True
        for j in range(train_data.shape[0]):
            if sample_x_y[i,0] == train_data[j,0]:
                include = False
        if include == True:
            valid_data[counter] = sample_x_y[i]
            counter += 1

    train_data = train_data[train_data[:,0].argsort()] #sort data so
    plotting is nice
    valid_data = valid_data[valid_data[:,0].argsort()]

    train_x = train_data[:,0]
    train_y = train_data[:,1]
    valid_x = valid_data[:,0]
    valid_y = valid_data[:,1]


    return train_x, train_y, valid_x, valid_y
```

6.2.15 tikhonov.py

```
import numpy as np
import matplotlib.pyplot as plt
import estimators
import finite_diff
from sample import random_sample_equi

# Save or show plots
save_plots = True

## Generating training / testing data

num_train_samples = 60

color1 = '#FF595E'
color2 = '#1982C4'
color3 = '#6A4C93'
fname = '$sin(x) + sin(5x)$'

#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
def visualize(fig, func, func_name):

  # visualize tikhonov estimator vs actual function
  if fig == 1:
    #set random seed and generate random samples
```

```
    seed = 50
    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)

    #meshes for function and polynomial plotting
    xeval = np.linspace(-3,3,1000)
    feval = func(xeval)

    #initialize info for tikhonov
    degree = 15
    weights = finite_diff.generate_centered_D(degree + 1)

    for lam in [0, 0.1, 1]:
      # create and fit tikhonov, predict on test data
      tikhonov = estimators.tikhonov(lam, degree, weights)
      tikhonov.fit(x_train, y_train)
      b_hat = tikhonov.predict(x_test)

      #get polynomial by predictin on entire interval
      poly = tikhonov.predict(xeval)

      #plot everything
      plt.plot(xeval, poly, label = 'Tikhonov Polynomial', color = color2)
      plt.plot(x_train, y_train, '.', label = 'Training data', color = color3)
      plt.plot(xeval, feval, label = 'f(x) = ' + func_name, color = color1)
      plt.xlabel('x')
      plt.ylabel('y')
      plt.legend()

      if save_plots:
          plt.savefig(f"../Images/tikhonov_poly_lambda{lam}.pdf")
      else:
          plt.show()
      plt.close()

  #RSS Values vs lambda for a specific seed
  if fig == 2:
    #set seed and generate random info, etc. same as above
    seed = 50
    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)

    #generate all lambdas
    lambdas = np.linspace(0, 20, 1000)
    degree = 15
    weights = finite_diff.generate_centered_D(degree + 1)

    #list for storing RSS values
    RSS_vals = []

```

```python
    #create and fit tikhonov for each lambda, collect RSS values.
    for l in lambdas:
      tikhonov = estimators.tikhonov(l, degree, weights)
      tikhonov.fit(x_train, y_train)
      RSS_vals += [tikhonov.RSS(x_test, y_test)]

    #plot!
    plt.plot(lambdas, RSS_vals, color = color2)
    plt.xlabel('$\lambda$')
    plt.ylabel('RSS')

    if save_plots:
        plt.savefig("../Images/tikhonovRSS.pdf")
    else:
        plt.show()
    plt.close()

    #find and print minimum lambda
    print('lambda that achieves minimum: ', lambdas[np.argmin(RSS_vals)])

  #RSS values vs degree for specific seed
  if fig == 3:

    #initialize same info as prev funcs.
    seed = 50
    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)

    #range of all degrees
    degrees = range(3, 20)
    lam = .1
    RSS_vals = []

    #create and fit tikhonov for each degree, collect and store RSS on val. data.
    for d in degrees:
      weights = finite_diff.generate_centered_D(d + 1)
      tikhonov = estimators.tikhonov(lam, d, weights)
      tikhonov.fit(x_train, y_train)
      RSS_vals += [tikhonov.RSS(x_test, y_test)]
    plt.plot(degrees, RSS_vals, color = color2)
    plt.xlabel('Degree')
    plt.ylabel('RSS')

    if save_plots:
        plt.savefig("../Images/tikhonovRSSvsDEG.pdf")
    else:
        plt.show()
    plt.close()

  #RSS Values for various seeds
  if fig == 4:
    # initialize seed range
```

```
    seeds = range(1,101)

    #list to store best lambda for each seed
    best_lambdas = []

    #iterate over each seed
    for seed in seeds:
      #initialize data, tikhonov info, and lambda range
      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)
      lambdas = np.linspace(0, 20, 1000)
      degree = 15
      weights = finite_diff.generate_centered_D(degree + 1)
      RSS_vals = []
      minRSS = float('inf')
      minlam = 0
      # iterate over lambdas and find minimum RSS and equivalent lambda
      for l in lambdas:
        tikhonov = estimators.tikhonov(l, degree, weights)
        tikhonov.fit(x_train, y_train)
        RSS = tikhonov.RSS(x_test, y_test)
        RSS_vals += [RSS]
        if RSS < minRSS:
          minRSS = RSS
          minlam = l
      # add best lambda for each seed
      best_lambdas += [minlam]
      # plot the RSS vs Lambda line semi-transparent
      plt.plot(lambdas, RSS_vals, alpha = .3)

    #plot everything else and print zero and nonzero lambdas
    plt.semilogy()
    plt.xlabel('$\lambda$')
    plt.ylabel('RSS')

    if save_plots:
        plt.savefig("../Images/lambda_all_seeds.pdf")
    else:
        plt.show()
    plt.close()

    nonzero = np.count_nonzero(best_lambdas)
    zero = len(best_lambdas) - nonzero
    print(f'Number of seeds where 0 is the best lambda: {zero} \n Number of seeds where best lambda is nonzero: {nonzero}')

  #RSS values for various seeds as a single statistical function
  if fig == 5:
    seeds = range(1,101)
    num_lambdas = 100
    degree = 11
    weights = finite_diff.generate_centered_D(degree + 1)
    lambdas = np.linspace(0, 20, num_lambdas)
```

```
    y_evals = np.zeros((len(seeds),num_lambdas))

    for i in range(len(seeds)):
      seed = seeds[i]
      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)
      RSS_vals = []
      for j in range(len(lambdas)):
        l = lambdas[j]
        tikhonov = estimators.tikhonov(l, degree, weights)
        tikhonov.fit(x_train, y_train)
        RSS_val = tikhonov.RSS(x_test,y_test)
        y_evals[i][j] = RSS_val

    means = np.mean(y_evals,axis=0)
    stdevs = np.std(y_evals,axis=0)

    plt.plot(lambdas, means, color="red", label="Mean")
    plt.fill_between(lambdas, means-stdevs,\
                     means+stdevs,\
                     color="red", alpha=0.25, edgecolor=None, label="Stdev")
    plt.legend()
    plt.xlabel("$\lambda$")
    plt.ylabel("Residual Sum of Squares")
    plt.savefig("../Images/Tikhonov_5.pdf")

    if save_plots:
        plt.savefig("../Images/Tikhonov_5.pdf")
    else:
        plt.show()
    plt.close()

  #RSS values for various degrees as a single statistical function
  if fig == 6:
    # Make plots of RSS values vs degree (y-axis has mean and standard deviation) for a specific seed

    seed = 50 # Set a constant seed
    num_lambdas = 100 # Number of lambdas to evaluate
    degrees = list(range(1,20)) # Degrees of the polynomials we're fitting
    lambdas = np.linspace(0, 20, num_lambdas) # Range of lambdas we're evaluating at
    y_evals = np.zeros((len(degrees),num_lambdas)) # Matrix which stores out results

    # Iterate over the different polynomial fits
    for i in range(len(degrees)):
      # Set the current degree we're using
      degree = degrees[i]
      # Generate our D matrix for centered difference
      weights = finite_diff.generate_centered_D(degree + 1)
      # Generate our training and testing data
```

```
      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)
      # List that we're storing our result in for different lambdas
      RSS_vals = []
      for j in range(len(lambdas)):
        l = lambdas[j]
        # Solve our Tikhonov equation
        tikhonov = estimators.tikhonov(l, degree, weights)
        tikhonov.fit(x_train, y_train)
        RSS_val = tikhonov.RSS(x_test,y_test)
        # Store our result
        y_evals[i][j] = RSS_val

    # Calculate the mean and standard deviation for each lamdba
    means = np.mean(y_evals,axis=0)
    stdevs = np.std(y_evals,axis=0)

    # Make and save our plot
    plt.plot(lambdas, means, color="red", label="Mean")
    plt.fill_between(lambdas, means-stdevs,\
                    means+stdevs,\
                    color="red", alpha=0.25, edgecolor=None, label="Stdev")
    plt.xlabel("$\lambda$")
    plt.ylabel("Residual Sum of Squares")
    plt.legend()

    if save_plots:
        plt.savefig("../Images/Tikhonov_6.pdf")
    else:
        plt.show()
    plt.close()

  #RSS values vs degree for specific seed and lambda
  if fig == 7:
    # Make plots of RSS values vs degree (y-axis has mean and standard deviation) for a specific lambda

    num_seeds = 100 # Number of seeds we'll iterator over
    seeds = list(range(0,num_seeds)) # Make our list of seeds
    degrees = list(range(3, 20)) # Degrees of the polynomials we're fitting
    num_degrees = len(degrees)

    # Make our matrix that we'll store our results in
    RSS_vals = np.zeros((num_seeds,num_degrees))

    # Iterate over our seeds
    for i in range(num_seeds):
      seed = seeds[i]
      # Get our training and testing data
      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)
      # Constant lambda for Tikhonov
      lam = .1
      RSS_val = []
      # Iterate over our degrees
      for d in degrees:
        # Generate our centered difference D matrix
        weights = finite_diff.generate_centered_D(d + 1)
        # Fit our Tikhonov
        tikhonov = estimators.tikhonov(lam, d, weights)
        tikhonov.fit(x_train, y_train)
        # Store our result
        RSS_val += [tikhonov.RSS(x_test, y_test)]
      RSS_vals[i,:] = RSS_val

    # Calculate our mean and standard deviation for each gamma
    means = np.mean(RSS_vals,axis=0)
    stdevs = np.std(RSS_vals,axis=0)

    # Make our plot
    plt.plot(degrees, means, color="red", label="Mean")
    plt.fill_between(degrees, means-stdevs,\
                     means+stdevs,\
                     color="red", alpha=0.25, edgecolor=None, label="Stdev"
   )
    plt.ylim(0,8000)
    plt.xlabel('Degree polynomial')
    plt.ylabel('Residual Sum of Squares')
    plt.legend()

    if save_plots:
        plt.savefig("../Images/Tikhonov_7.pdf")
    else:
        plt.show()
    plt.close()

  if fig == 8:
    # Make plots training/testing data, our original function, and
   Tikhonov fit

    # Iterate over our polynomials
    for degree in range(15,20):
      # Constant seed which looks nice for plotting purposes
      seed = 4596
      # Get our training and testing data
      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)
      xeval = np.linspace(-3,3,1000)
      feval = func(xeval)
      # Generate our D matrix for centered difference
      weights = finite_diff.generate_centered_D(degree + 1)
      # Constant lambda for Tikhonov
      lam = .1
```

```
    # Fit our Tikhonov regularization
    tikhonov = estimators.tikhonov(lam, degree, weights)
    tikhonov.fit(x_train, y_train)
    # Get our predicted polynomial
    poly = tikhonov.predict(xeval)
    # Make our plot
    plt.plot(xeval, poly, label = 'Tikhonov Polynomial', color = color2)
    plt.plot(x_train, y_train, '.', label = 'Training data', color = color3)
    plt.plot(x_test, y_test, '.', label = 'Testing data', color = 'hotpink')
    plt.plot(xeval, feval, label = 'f(x) = ' + func_name, color = color1)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()

    if save_plots:
        plt.savefig(f"../Images/Tikhonov_8_{degree}.pdf")
    else:
        plt.show()
    plt.close()

# Fits for different difference formulas
if fig == 9:
  # same as above
  seed = 50
  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)
  xeval = np.linspace(-3,3,1000)
  feval = func(xeval)
  degree = 15
  #list of different weight matrices and their names
  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')]
  # generate tikhonov for each weight and create same plot as fig = 1
  for w in weights:
    lam = 1
    tikhonov = estimators.tikhonov(lam, degree, w[0])
    tikhonov.fit(x_train, y_train)
    coefs = tikhonov.xstar
    b_hat = tikhonov.predict(x_test)
    poly = tikhonov.predict(xeval)
    plt.plot(xeval, poly, label = 'Tikhonov Polynomial', color = color2)
    plt.plot(x_train, y_train, '.', label = 'Training data', color = color3)
    plt.plot(xeval, feval, label = 'f(x) = ' + func_name, color = color1)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()

```

```python
        if save_plots:
            plt.savefig(f"../Images/Tikhonov_9_{w[1]}.pdf")
        else:
            plt.show()
        plt.close()


# Uncomment any of the below to produce a specific figure
f = lambda x : np.sin(x) + np.sin(5*x)
visualize(1, f, fname)
#visualize(2, f, fname)
#visualize(3, f, fname)
#visualize(4, f, fname)
#visualize(5, f, fname)
#visualize(6, f, fname)
#visualize(7, f, fname)
#visualize(8, f, fname)
#visualize(9, f, fname)
```