Numerical Methods for Basic Optimization Problems

Abstract

In this paper we explore the use of numerical techniques to solve basic optimization problems. We first implement the gradient descent algorithm and then use variations of the algorithm to optimize objective functions with long summations. We apply these techniques to minimize squared error in linear regression. Finally, we use regularization to avoid overfitting.

1 Gradient Descent

Gradient descent is commonly used to find an input, θ, which minimizes a function, $F(\theta)$. The algorithm starts with an initial $\theta^{(0)}$ and iteratively updates it according to the equation: [1]

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t * \nabla_\theta F(\theta^{(t)}) \tag{1}$$

The step size, η_t, is used to scale the learning rate to converge quickly and precisely. η_t may be constant or a function of t which decreases as the iteration number t increases to force convergence (see Section 1.1 for an example with a non-constant η_t).

We tested our implementation of gradient descent on two functions:

1. A two-dimension negative Gaussian where:

$$\mu = [\begin{array}{cc} 10 & 10 \end{array}]^T, \quad \Sigma = 1000I$$

2. A two-dimensional paraboloid (bowl):

$$f(x) = x^T A x - x^T b, \quad A = \begin{bmatrix} 10 & 5 \\ 5 & 10 \end{bmatrix}, \quad b = [\begin{array}{cc} 400 & 400 \end{array}]^T$$

We used a constant step size, and our criterion for convergence depends on whether the difference in the objective function between two iterations falls below a constant convergence threshold. In addition, our implementation allows us to easily use other step size functions and convergence criteria. One convergence criterion to consider in future work is testing whether the norm of the gradient falls below a set threshold.

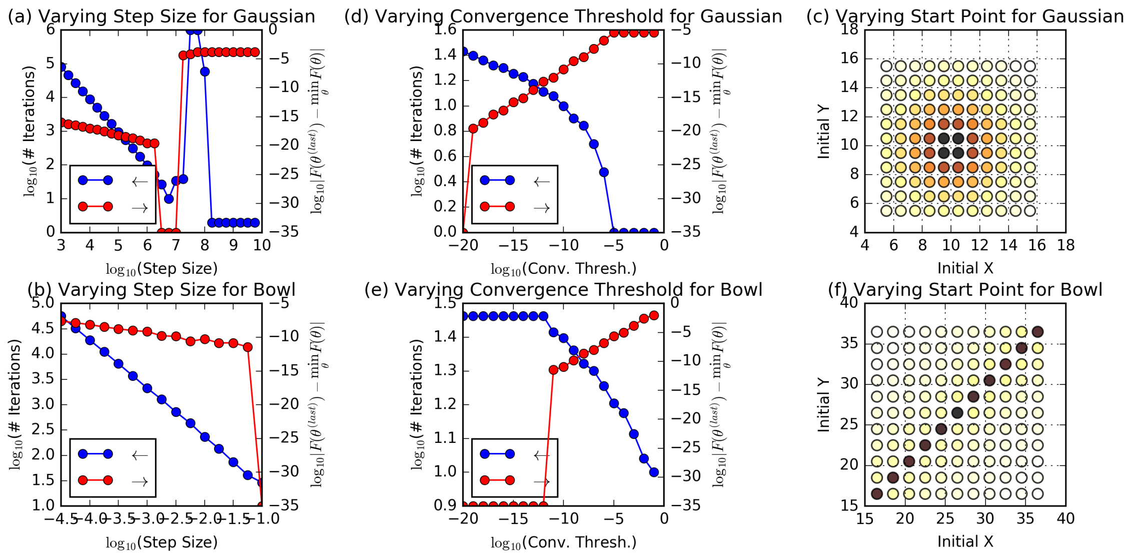

Figure 1: Results from running gradient descent on the Gaussian and bowl functions while varying the step size, convergence threshold, and start point. In plots (c) and (f), the shade represents the number of iterations until convergence where darker shades correspond to fewer iterations.

[1] η_t is the step size and may be a scalar or a d dimensional vector, in which case $*$ represents component-wise vector multiplication.

To begin our analysis, we observed the effect of choice of step size and convergence threshold on the number of iterations until convergence and on the difference between the objective function of the final θ and the analytical minimum of the objective function (that is, $f(\theta^{(\text{last})}) - \min_\theta f(\theta)$). The results are shown in Figure 1.

Figures 1(d) and 1(e) show that, as expected, there is a tradeoff between the number of iterations until convergence and the accuracy of the final answer (i.e. distance to the theoretical minimum). Higher accuracy requires a lower convergence threshold and more iterations.

Figures 1(a) and 1(b) show that, for both the bowl and the Gaussian, for a step size too small, the number of iterations before convergence is proportional to $1/\eta^c$ for some $c \geq 1$. This means taking steps which are too small will quickly lead to infeasible runtimes. Conversely, taking step sizes too large may also lead to problems converging, as the update may overshoot the local minimum and never be able to land on the minimum value. Note that the iterations until convergence and error are also dependent on start point (as shown in Figures 1(c) and 1(f)), so when choosing a step size to use in practice, it is best to use one less than what might be considered ideal for a fixed start point to avoid overshooting at other start points (or use a more adaptive step size).

Since functions used in practice often do not have a closed-form gradient, we implemented a centered finite difference approximation for the gradient:

$$\nabla_\theta F(\theta)_i \approx \frac{F(\theta + \delta \cdot \vec{u}(i)) - F(\theta - \delta \cdot \vec{u}(i))}{2\delta}$$

where $\vec{u}(i)$ is the vector of all 0 except in position i, which has value 1. We verified the implementation by comparing the numeric approximation to the analytical value of the gradient at various points and δ values for the Gaussian and bowl functions. Two plots showing the results are shown in Figure 2. The lack of precision for low values of δ is likely due to loss of floating point precision during subtraction. [2]

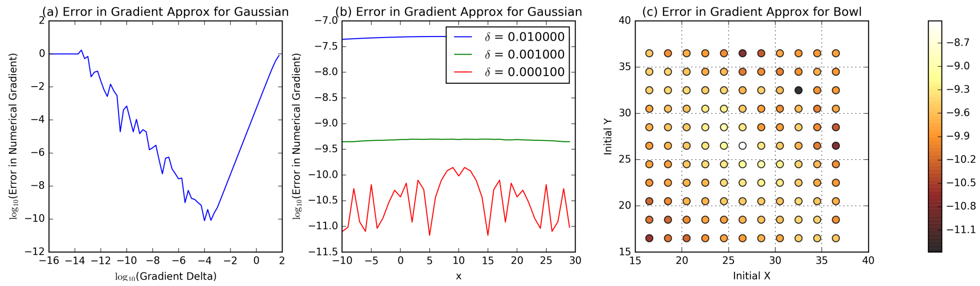

Figure 2: Results showing the performance of the finite difference approximation of the gradient for the Gaussian and bowl functions at various points. For the Gaussian, y is fixed to 5 to allow for a higher resolution of samples in one dimension. $\delta = 0.0001$ was used to generate (c).

1.1 Stochastic Gradient Descent

In problems such as linear regression (see Section 2) the objective function may be a function of many training points (e.g. Equation 3). If gradient descent is used when the objective function is a sum over training points, the process is called batch gradient descent (BGD). Even if the function has an analytical solution to its gradient, evaluating the function over the entire training set for each parameter update may be computationally infeasible.

One solution to this problem is to only evaluate a partial sum for the gradient function during each update. The simplest form of this approach is stochastic gradient descent (SGD), in which only the contribution of a single training point to the objective function's gradient is used per update. The algorithm shuffles the training set, iterates through it, and for each training example, runs an update. When it runs out of training examples, it re-shuffles the training data and repeats. During each iteration, the parameterization is updated using Equation 1, where the gradient is the contribution due to the single training point.

Batch gradient descent updates the parameters once every N point-wise gradient evaluations whereas stochastic gradient descent updates the parameters once every one point-wise gradient evaluation. Stochastic gradient descent converges to the same value as batch gradient descent in a fraction of point-wise evaluations, as shown in in Figure 3. This finding means that in non-vectorized and non-parallelized versions of the algorithms, SGD will converge in a fraction of the time.

It is also important to note that a non-constant η_t is often used in stochastic gradient descent to guarantee convergence. When choosing an η_t for SGD, it should satisfy the Robbins-Monro conditions: $\sum_{t=1}^{\infty} \eta_t = \infty$ and $\sum_{t=1}^{\infty} \eta_t^2 < \infty$. To generate Figure 3, our implementation of SGD uses:

$$\eta_t = \alpha \left(\tau_0 + t\right)^{-\kappa} \tag{2}$$

where α, τ_0, and κ are tunable parameters and $\alpha > 0$, $\tau_0 \geq 1$, and $0.5 < \kappa \leq 1$.

The "jumps" in SGD, as seen in Figure 3(a), are primarily due to the fact that only one training point is used to estimate the gradient, which depends on all training points. A middle-ground between BGD and SGD is called mini-batch gradient descent, which has

[2]The loss of significant bits in the mantissa, a problem common in finite difference approximations.

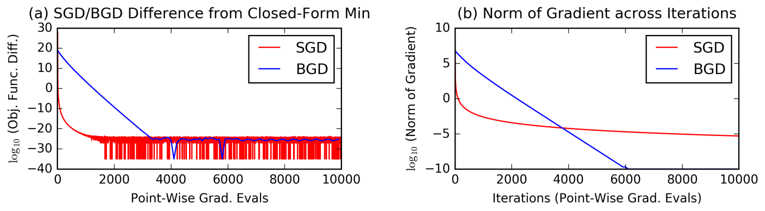

Figure 3: Differing convergence rates of BGD and SGD. Equation 2 was used with $\alpha = 6 \cdot 10^{-5}, \tau_0 = 1, \kappa = 1$ for the η_t in SGD.

fewer point-wise evaluations than BGD but is less "jumpy" than SGD. It is similar to SGD, but it uses more than one training point in its evaluation of the partial sum of the gradient function.

2 Linear Basis Function Regression

2.1 Problem Statement

We now apply gradient descent to linear regression. The objective function minimized in ordinary least squares linear regression is the sum of squared errors in the regression's predictions over the training set, as given in Equation 3.

$$\text{Loss}_{\text{lin}}(w; x, y) = \sum_{i=1}^{N} \left(y^{(i)} - w^T \phi(x^{(i)}) \right)^2 \tag{3}$$

or in matrix form:

$$\text{Loss}_{\text{lin}}(w; X, y) = (\Phi(X)^T w - y)^T (\Phi(X)^T w - y)$$

where X is an $d \times N$ matrix of N d-dimensional training inputs and $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ is a basis function. Φ performs ϕ on each of the column vectors in X, giving $\Phi : \mathbb{R}^{N \times d} \to \mathbb{R}^{N \times d'}$.

2.2 Fitting Data

To show how linear regression works in practice, for this section we use a data set consisting of points generated from

$$y(x) = \cos(\pi x) + 1.5 \cos(2\pi x) \tag{4}$$

(with added noise) and explore methods of fitting it. We also will use two simple bases:

$$\phi_{\text{poly},M}(x) = \left[1, x, x^2, ..., x^M\right]^T$$
$$\phi_{\text{cos},M}(x) = [\cos(\pi x), \cos(2\pi x), ..., \cos(M\pi x)]^T$$

There is a closed-form solution to the maximum likelihood parameterization, $\hat{w}_{\text{ML}}$, derived from setting the gradient of the sum-of-squared errors equal to zero:

$$\hat{w}_{\text{ML}} = \left(\Phi(X)\Phi(X)^T\right)^{-1} \Phi(X)y$$

We use this equation with $\phi_{\text{poly},0}$, $\phi_{\text{poly},1}$, $\phi_{\text{poly},3}$, and $\phi_{\text{poly},10}$ and plot the resulting models' predictions in Figure 4(a).

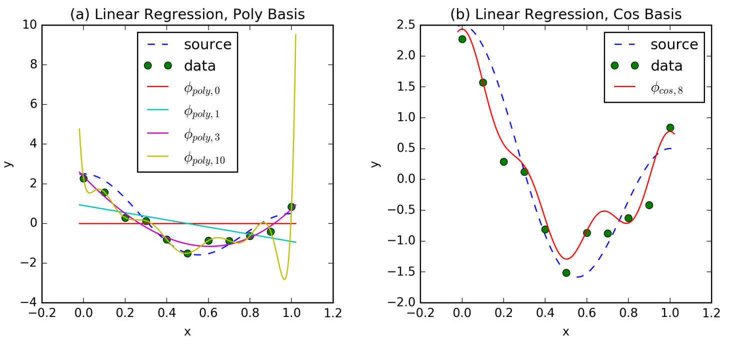

Figure 4: Results of fitting the training data with simple polynomial bases (a) and a cosine basis (b)

In Figure 4(a), although the $\phi_{\text{poly},10}$ basis fits the training points exactly, it overfits the function from which the training points were generated (Equation 4). This is most apparent in the end-behavior of the $\phi_{\text{poly},10}$ fit: the the fit has erratic shifts in the first order derivative between the points near the edges of the training set. In fact, the distribution from which training set was generated is the sum of low-frequency sinusoids and is therefore unlikely to see such large shifts in first derivative behavior.

In contrast, the cosine basis in Figure 4(b) fits the original generator function well. Though it technically incurs greater squared loss than the polynomial fit (0.515 vs 0.000567), it generalizes much better, as the end behavior is less erratic. In addition, since we know the underlying generating function is sinusoidal in nature, the cosine basis is the basis of choice. However, it is interesting to note that the fit shown in Figure 4(b) has form:

$$\hat{f}(x) = 0.769\cos(\pi x) + 1.09\cos(2\pi x) + 0.099\cos(3\pi x) \\ + 0.143\cos(4\pi x) + -0.051\cos(5\pi x) + 0.362\cos(6\pi x) + 0.012\cos(7\pi x) + 0.015\cos(8\pi x)$$

which has high frequency, albeit low amplitude, terms which the generating function, Equation 4, does not have. These high frequency terms are likely due to the noise added to the data when generating the training set.

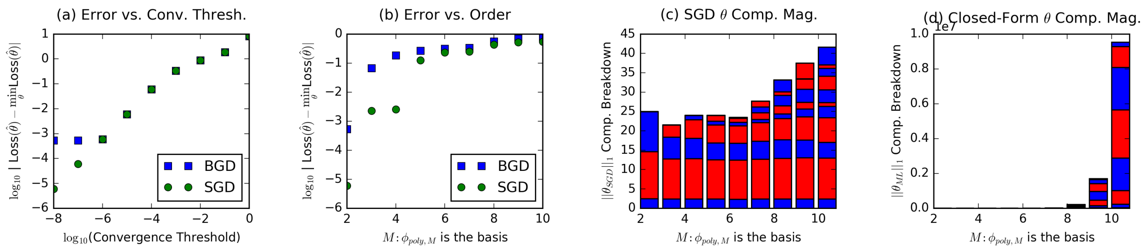

Figure 5: Results of fitting the data with the simple polynomial basis $\phi_{\text{poly},M}$ and SGD and BGD as compared to the closed-form max-likelihood parameterization

We used stochastic gradient descent and batch gradient descent to train the linear regression model iteratively. In our experiments, these numerical methods converged to loss values within five decimal points of the closed-form minimum loss, as shown in Figure 5(a).

The weight vectors found through BGD and SGD had relatively low norms, especially compared to closed-form solutions of higher order polynomial basis. This trend is best shown by comparing Figures 5(c) and (d). The stacked bar height represents the L_1 norm of the weight vector. For higher order bases, the L_1 norm of the closed-form solution is on the order of 10^7 whereas, for our trials, it only reached an L_1 norm of about 45 while. The discrepancy is due to learning rate: the step size and iteration cap of 5000 iterations prevented SGD and BGD from converging to such a large magnitude weight vector. Granted, the loss for the solutions found for SGD and BGD was on the order of 10^0 greater than the minimum closed-form loss, but it also resulted in a smoother fit that may generalize better than the closed-form min-loss weight vector for higher order polynomials. In this manner, the learning rate can help to regularize linear regression for higher order polynomials by limiting the growth of the weight vector.

To allow the high order components of the weight vector to grow to significantly larger values (like those of the min-loss weight vector), we would experiment with lowering the convergence threshold further and setting a component-specific step size: allowing high order weights to grow faster than low order weights to attempt to match a component breakdown similar to Figure 5(d).

3 Ridge Regression

3.1 Overview

Ridge regression extends linear regression using regularization. In ridge regression, the error includes a term that penalizes the magnitude of the weight vector by adding the L_2 norm of the vector into the error, as shown in Equation 5. The parameter λ is the *regularization coefficient*, which determines how much the magnitude of the weight vector should increase the error. This regularization of the weight vector helps to prevent overfitting of the training data.[3] The maximum likelihood estimate of the weight vector $\hat{w}_{\text{ML}}$ in ridge regression is given by Equation 6.

$$\text{Loss}_{\text{ridge}}(w; x, y) = \sum_{n=1}^{N} \left(y^{(i)} - w^T \phi(x^{(i)}) \right)^2 + \lambda w^T w \tag{5}$$

$$\hat{w}_{\text{ML}} = \left(\lambda I + \Phi^T \Phi \right)^{-1} \Phi^T Y \tag{6}$$

In this section, we use ridge regression to minimize the loss function in Equation 5. We continue to use the same polynomial basis $\phi(x) = \phi_{\text{poly},M}(x)$, where M corresponds to the maximum order in the polynomial. We explore how varying the parameters M and λ affects the sum-of-squares error (SSE) after minimizing loss using ridge regression. First, we use the training set as our test set, and then we show the usefulness of ridge regression in preventing overfitting in the case where the training and test sets are different.

3.2 Results

We first discuss the case where the training and tests sets are the same, using the same dataset as in Section 2. Figure 6(a) shows the effect of varying the order M of the polynomial basis affects the SSE on the training set. We see that for small values of λ, total error decreases as the order of the polynomial increases. This result makes sense because for small values of the regularization parameter, we are essentially performing simple linear regression, and as the order of the polynomial increases, we can fit the training data more accurately, so the error on this dataset decreases. As λ increases, the estimator penalizes weight vectors with larger values more strongly, so as the weight vector attempts to use more terms (as M increases), it is more difficult to decrease overall error.

[3] For further explanation, see Bishop pp. 144-145.

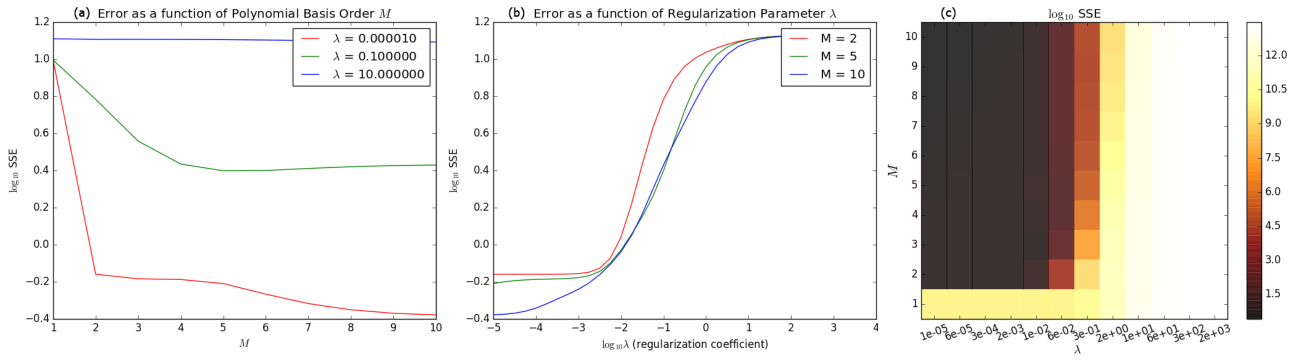

Figure 6: Sum-of-squares error on **training set** using ridge regression where (a) the order of the polynomial basis M is varied, and (b), the regularization coefficient λ is varied. (c) Shows effects of varying M and λ on the SSE of training set following estimation with ridge regression. Error is represented as $\log_{10}$ SSE.

Figure 6(b) shows the effect of varying the regularization parameter λ on the SSE for three different values of M. As λ increases, the SSE increases as it becomes harder to fit the training data when the values of weights in the weight vector are effectively constrained to be smaller.

The dependency between these variables is illustrated in Figure 6(c), which shows the mutual effects of varying M and λ on the overall error. For the case where the the training and test sets are the same, we observe that the optimal parameters are large M and small λ, and that for some values of λ beyond a critical point ($\lambda \approx 2$ in our case), error is large regardless of M.

Next, we explore the effective of regularization in the case where the training and test sets are different. We now use three separate datasets: one for training, one for “validation”, and one for testing. The validation set is used to select the optimal set of parameters M and λ after training using the training set. The test set is then used to assess performance on completely unseen data after the optimal set of parameters is selected on the validation set.

Figure 7(a) shows how the SSE varies as a function of M for two different values of lambda λ, one small ($\lambda = 10^{-5}$) and the other larger ($\lambda = 1$). These results show that for M significantly large (in this case, $M \geq 7$) and λ small, the regression overfits the training set, as error on the training set drops, while error on the validation and test sets increases. However, we see that in the case of larger λ, we avoid the problem of overfitting, but at the same time, do not reduce error on unseen data.

Figure 7(b) shows how SSE varies as a function of the regularization coefficient λ for two different values of M. In the case of M small ($M = 2$), we note how larger values of λ increase error because for small M, it is harder to overfit the training data, and more regularization tends to increase error (as explained above). However, in the case of larger M (e.g. $M = 10$), we see a dip in error on the unseen data for $0.001 \leq \lambda \leq 10$. This result shows that regularization can prevent overfitting the training data when using higher order polynomial bases. Further, when λ is selected optimally, performance on the test set with larger M and larger λ is better than performance on the same test set with smaller M and $\lambda \approx 0$, showing that regularization has the potential to increase overall performance of the regressor on unseen data compared to linear regression without regularization.

We also tried swapping which data are used for training and which are used for testing (leaving the validation set unchanged). In the first case (the same as used to produce the given figures), we found optimal values $M = 2$ and $\lambda = 0.00001$, but for the second case, found $M = 3$ and $\lambda \approx 0.56$ to be optimal. However, as shown in Figure 6(c), the difference between these two cases on SSE is very small. The optimal parameters, particularly the amount of regularization necessary to prevent overfitting, can depend heavily on the given data, so tuning parameters using a validation set is helpful in practice.

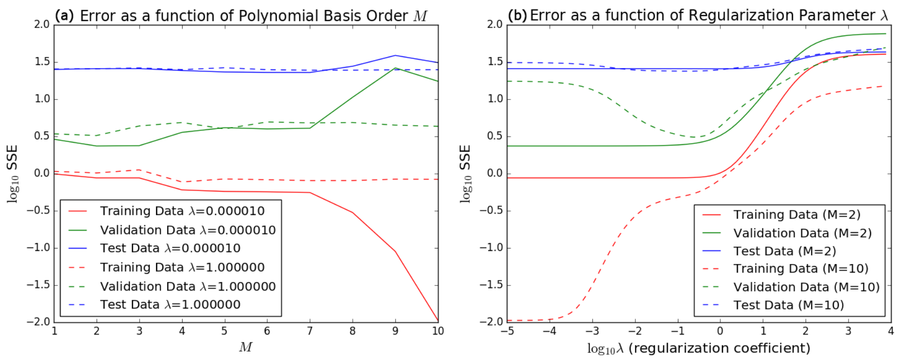

Figure 7: Sum-of-squares error on unseen **test set** using ridge regression where (a) the order of the polynomial basis M is varied, and (b), the regularization coefficient λ is varied.

4 Sparsity and LASSO

4.1 Overview

We now consider another form of regularization known as LASSO, in which our regularization penalty is proportional to the L_1 norm of the weight vector, rather than the L_2 norm (as is the case with ridge regression). LASSO regression minimizes the objective function given in Equation 7. LASSO has the property that if λ is sufficiently large, some of the feature coefficients are driven to zero, which leads to a sparse weight vector (see Bishop p. 145 for further explanation).

$$\text{Loss}_{\text{lasso}}(w; x, y) = \frac{1}{n} \sum_{i=1}^{N} \left(y^{(i)} - w^T \phi(x^{(i)}) \right)^2 + \lambda \sum_{j=1}^{M} |w_j| \tag{7}$$

In this section, we consider a new dataset generated according to $y = w_{\text{true}}^T \cdot \phi(x) + \varepsilon$, where $x, y \in \mathbb{R}$, $w \in \mathbb{R}^{13}$, and ε is a small noise. The feature vector is given by the basis $\phi(x) = (x, \sin(0.4\pi x \times 1), \sin(0.4\pi x \times 2), \ldots, \sin(0.4\pi x \times 12)) \in \mathbb{R}^{13}$. We use LASSO regression with various values of λ to estimate the weights w from the dataset[4], and compare the results to the weights estimated by ridge regression and ordinary least squares.

4.2 Results

Consider Figure 8(a), which shows that the vector w_{true} has only four features with non-zero weight. However, the weights estimated from ordinary least squares and ridge regression (Fig. 8(b) and 8(c)) are all non-zero. Further, in the case of ridge regression, we note that varying λ does not significantly affect the feature weights.

We see that LASSO regression comes the closest to estimating the true feature weights (Fig. 8(d)). We also see that increasing λ encourages the estimator to drive more weights to zero and produce a sparser weight vector. These results validate the expected property of LASSO, particularly that the technique is useful as a means of feature selection.

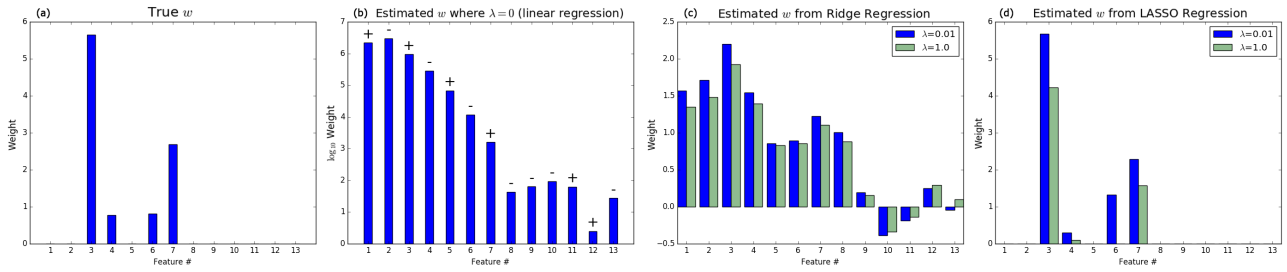

Figure 8: True feature weights (a), and feature weights estimated by ordinary least squares (b), ridge regression (c), and LASSO regression (d). Note that (b) gives the $\log_{10} |w_i|$ value for all features i and denotes whether the original feature value was positive or negative.

Figure 9 compares these various regression techniques on the provided train, validation, and test datasets. Least squares does not generalize well at all by strongly overfitting the training data. Note that the least squares curve exceeds the boundaries of the plot due to the large magnitude of the vector. Ridge regression generalizes to the unseen data decently well, though not as well as LASSO. Moreover, LASSO requires fewer features in order to make predictions, which improves performance over ridge on larger datasets.

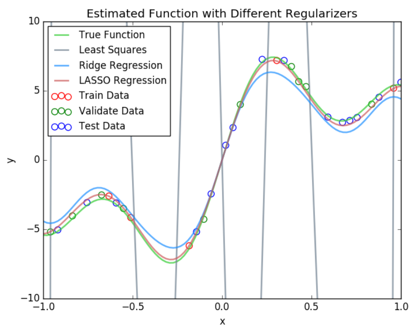

Figure 9: Plots of estimated function with different regularizers. Ridge regression used $\lambda = 1.0$ and LASSO used $\lambda = 0.01$.

References

[1] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[4]We used the Scikit-learn (`http://scikit-learn.org`) implementation of LASSO regression.