6.867: Homework 1

1. Gradient descent

In this section, we explored the simplest and most-commonly discussed method of optimization for differentiable cost functions: gradient descent. We implemented a basic batch gradient descent procedure and applied it to minimize two different objective functions: the negative Gaussian function and the quadratic bowl function. Furthermore, after creating the implementations, we investigated how our choice of starting guess, step size, and convergence criterion/threshold impacted our resulting solution.

1.1. Starting guess

For the quadratic bowl, we saw that starting guess did not have much of an impact on our resulting solution. If the starting guess was farther away from the actual minimum, it simply took the algorithm more iterations to converge. For the negative gaussian, however, the starting guess does make a bit of an impact. The reason is that if we pick a point that is too far away from the center, the graph is basically flat and has close to 0 gradient. This causes our difference to be extremely small and without an extremely large step size our gradient descent simply stops. For both objective functions, we saw that as long as the algorithm converges, the norm of the gradient continues to decrease in magnitude. Additionally, it is worth noting that for alternative objective functions with behavior including plateaus or local minima, starting guess would likely play a larger role. Figure 1 illustrates gradient descent on our two objective functions with starting guesses of various orders of magnitude.

1.2. Step size

Throughout our research, we discovered that step size had a fairly significant impact on the resulting solutions. For example, a step size that was too large could result in the gradient descent algorithm diverging to infinity instead of converging. In this case, the algorithm would overstep the minimum on the first step, thus causing the norm of the new gradient to be larger than the original gradient norm. This would then proceed to compound at every step, ultimately causing the algorithm to diverge to infinity. Conversely, extremely small step sizes resulted in single-approach convergence, in which the gradient was always negative or always positive as we approached the minimum. For step sizes between these two extremes, the guesses oscillated as we approached the minimum, in which the gradient alternated between positive and negative and eventually converged on the minimum (Figure 1, second row). When the algorithm converged, we saw that the norm of the gradient for both objective functions decreased in magnitude.

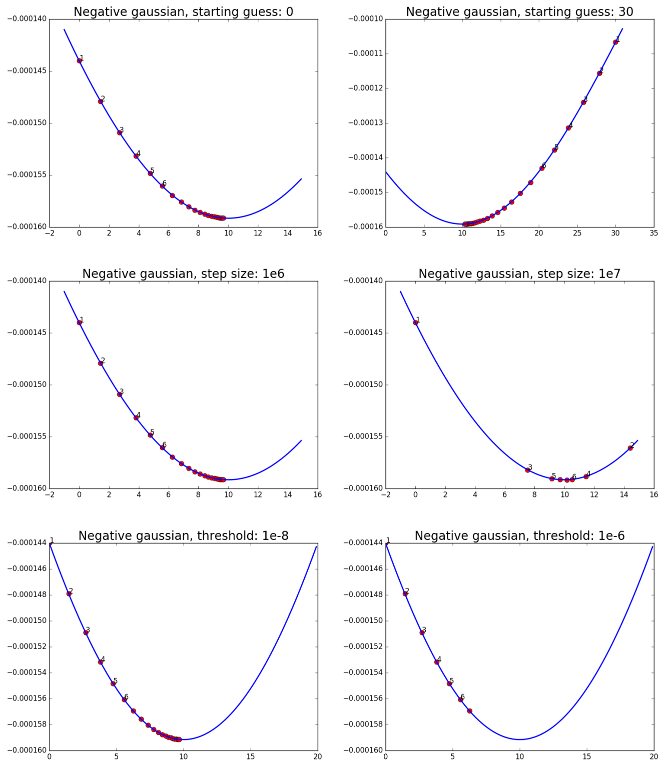

Figure 1. Starting guesses, step sizes, and thresholds.

1.3. Convergence criterion/threshold

In addition to step size, the convergence criterion had a fairly significant impact on the resulting solution. In general, we found that accuracy was inversely proportional to the convergence criterion while speed (and inversely number of iterations) was directly proportional to the convergence criterion. With a large convergence criterion, the accuracy of the resulting solution was reduced while the algorithm ran to completion faster. On the other hand, with a smaller convergence criterion, the accuracy of the resulting solution was increased while the algorithm took many more iterations to complete (third row of Figure 1). We also saw that as long as the algorithm converged, the magnitude of the norm of the gradients consistently decreased. ¿¿¿¿¿¿¿ 4387e0f79fdfab7cdca7d14c724f6088067c6e81

1.4. Numerical gradient descent

In order to verify that our gradient functions were correct, we created a numerical gradient function so that we could validate the gradients at various points. To calculate the numerical gradients, we utilized the central difference approximation. For the quadratic bowl, it appears that no matter what the difference step was, the numerical gradient was always identical to the true gradient. For the negative gaussian, we discovered that by making the difference step smaller, our numerical gradient gained accuracy and became closer to the true gradient. For example, at the point (0,0), the real gradient for the negative Gaussian was

$$[-1.440\,093\,48 \times 10^{-6}, -1.440\,093\,48 \times 10^{-6}].$$

The numerical gradient with a difference step of 100 was

$$[-1.140\,329\,86 \times 10^{-8}, -1.140\,329\,86 \times 10^{-8}].$$

With a difference step of 10, the numerical gradient was

$$[-1.372\,143\,53 \times 10^{-6}, -1.372\,143\,53 \times 10^{-6}].$$

With a difference step of 1, the numerical gradient was

$$[-1.439\,397\,60 \times 10^{-6}, -1.439\,397\,60 \times 10^{-6}].$$

Lastly, with a difference step of 0.1, the numerical gradient was

$$[-1.440\,086\,52 \times 10^{-6}, -1.440\,086\,52 \times 10^{-6}],$$

which was almost identical to the true gradient. From a mathematical perspective this makes intuitive sense because the definition of a derivative is basically the numerical gradient as the difference step approaches 0.

1.5. Batch versus stochastic gradient descent

In the next part of our research, we utilized two forms of gradient descent (batch and stochastic gradient descent) in order to minimize the following function:

$$J(\theta) = \Sigma_i (x^{(i)T}\theta - y^i)^2$$

As before, we had to be very particular in the convergence criterion and step sizes in order to ensure that the gradient descent converged. For batch gradient descent, we chose an initial guess of all 0's with a step size of 10^{-6} and a convergence criterion of 1. This resulted in a θ that converged to [0.50832359, -2.33986133, -6.31150421, 6.80803415, -1.06442692, 6.66532056, 3.39605963, -0.45908695, -12.94348103, 15.73213395], which ultimately gave a minimum $J(\theta)$ of 8343.21291982.

In stochastic gradient descent, instead of evaluating all training data in every parameter update, each update was based on a single training sample. When specifying the learning rate for each step, we used the Robbins-Monro conditions in order to guarantee convergence. Specifically, in our update rule

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta J(\theta_t; x^{(i)}, y^{(i)}),$$

we define

$$\eta_t = (\tau_0 + t)^{-\kappa}$$

where $\kappa = 0.75$ and $\tau_0 = 10^8$. We considered our convergence criterion to be when the change in each sample update was smaller than some threshold value for all samples in the data set. In other words, for a given round of evaluating the stochastic gradient descent, changes for all data samples needed to be within the threshold cutoff in order to terminate the algorithm. Batch and stochastic gradient descent are illustrated in Figure 2, with a starting guess of 0, step size of 10^{-6}, and threshold value of 1.

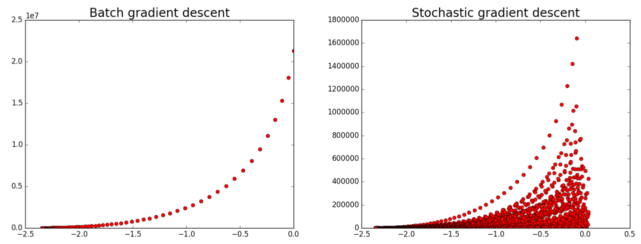

Figure 2. Batch and stochastic gradient descent.

In terms of accuracy, the batch gradient descent was generally more accurate than the stochastic gradient descent. However, in terms of the number of evaluations, the stochastic requires fewer iterations per parameter update. This is because in each iteration of batch gradient descent, n evaluations are necessary, compared with stochastic gradient descent which evaluates the a single point-wise gradient per update iteration. Overall however, we found that batch gradient descent required fewer overall evaluations than stochastic gradient descent.

2. Linear basis function regression

We consider linear basis function regression as a method to benchmark the robustness of the gradient descent solution presented above. By using the closed-form maximum likelihood equation, we can calculate the maximum likelihood weight vector for our list of basis functions to approximate the data in the form of our basis. In this scenario, we are using data generated by $y(x) = \cos(\pi x) + 1.5\cos(2\pi x) + \epsilon(x)$, where $\epsilon(x)$ is some added noise to the dataset. Running linear regression on a simple polynomial basis of order M, where $\phi_0(x) = x^0$, $\phi_1(x) = x^1$, $\phi_2(x) = x^2$, ..., $\phi_M(x) = x^M$, we calculate the maximum likelihood weight vector by the following:

$$w_{ML} = (\Phi^T\Phi)^{-1}\Phi^T y$$

where w_{ML} is the maximum likelihood weight vector and Φ is given by:

$$\Phi = \begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \phi_2(x_0) & \dots & \phi_M(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \phi_2(x_1) & \dots & \phi_M(x_1) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \phi_2(x_n) & \dots & \phi_M(x_n) \end{bmatrix}$$

Our choice of M, the degree of our polynomial basis, largely determines the fit of the regression to the data (Figure 2). More specifically, as small values of M, the polynomial basis cannot adequately capture all the data points. At higher values of M however, overfitting occurs in which the weight vector performs well on the training data, but is not well generalized to new data. The value of M therefore must be carefully considered in order to prevent too high variability in our generated regression polynomial.

After using the closed-form solution to determine the optimum weights for a given M, we then proceeded to reproduce these weights through batch and stochastic gradient descent. In order to use gradient descent we needed an objective function which was simply the sum of squares error function. To create the function we would multiply together the x input and the weight vector to get a guess and then find the difference between this guess and the actual y value. We then squared this difference and summed up all these squared differences throughout the entire dataset. To find the gradient we simply took the derivative of this loss function and verified its accuracy with the numeric gradient. We discovered at multiple M values and weight vectors that the gradients were consistent with one another meaning that our calculated gradient was correct. Due to the nature of the fact that the dataset was created by a cosine function we found that choosing $M = 1$ gave a much larger loss than choosing $M = 2/3$ (basically an order of magnitude of difference). We made the following observations regarding the impact of initial guess, step size, and convergence thresholds on batch gradient descent for M values of 2 and 3.

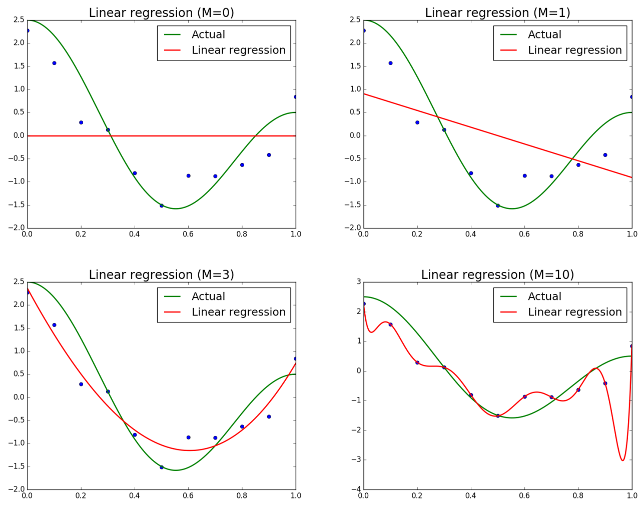

Figure 3. Linear regression for varying values of M.

2.1. Initial guess size

The overall findings we made regarding initial guess size is that as the initial guess size increases exponentially, the number of steps needed to converge only increases linearly. Furthermore, the initial guess size does not have any impact on the actual resulting converged weight (which was [2.393, -11.767, 9.969]. For example, with $M = 2$, and an initial guess of [100, 100, 100] the algorithm converged in 626 steps. Then with an initial guess of [1000, 1000, 1000] the algorithm converged in 1178 steps. Then with an initial guess of [10000, 10000, 10000], the algorithm converged in 1790 steps. The one minor interesting thing we found is that with an initial guess size of [1, 1, 1] it actually took 817 steps which is more than [100, 100, 100] even though it's closer to the actual converged weights. Our hypothesis for this strange phenomenon is that since some components of our initial guess were smaller than the converged value and come were greater, this requires more steps than when all components of the initial guess are larger.

2.2. Step size

We made some fairly interesting findings regarding the impact of step size on gradient descent convergence. Using $M = 2$, the true weights are [2.456, -12.150, 10.338]. We found that a step size equal to or above 0.07 resulted in a divergent solution. For a step size of 0.06 however, we got a converged weight of [2.398, -11.799, 10.001] which was quite close to the actual weights. Surprisingly, at step sizes smaller than 0.06, we actually found the converged weight to be less accurate. For example, at a step size of 0.006, the resulting weight vector was [2.255, -10.929, 9.160]. Although seemingly counterintuitive at first, this could be due to the fact that extremely small step sizes might reduce accuracy because the convergence criterion is met faster.

2.3. Convergence threshold

The overall findings we made regarding convergence threshold is that larger convergence thresholds meant less steps and less accuracy in the converged weights as compared to the true weights calculated using the closed form solution. Using $M = 2$, the true weights are [2.456, -12.150, 10.338]. At a convergence threshold of 0.0001, the converged weight is [2.393, -11.767, 9.969]. At a convergence threshold of 0.001, the converged weight is [2.657, -13.367, 11.504] (clearly farther away from the true weights). At a convergence threshold of 0.1, the converged weight is [3.978, -20.651, 18.206] (extremely far from the true weights).

2.4. Batch versus stochastic gradient descent

When we compared performance of SGD with bath gradient descent, we found that for both $M = 1$ and $M = 2$, SGD was able to converge to a more accurate weight in less single-point evaluation steps. For $M = 2$, the true weights were calculated to be [2.456, -12.150, 10.338]. As mentioned before, it took batch gradient descent (with optimum step size of 0.06) 626 steps (6886 single point evaluations) to converge to [2.398, -11.799, 10.001], while SGD (with $\kappa = 6$ and $t_0 = 100$) was able to converge to [2.434, -12.0154 10.208] with only 2679 single point evaluations. A similar phenomenon was noticed with $M = 1$.

2.5. Cosine basis functions

We can also instead choose our set of basis functions to be the set of cosine functions, where $\phi_1(x) = \cos(\pi x)$, $\phi_2(x) = \cos(2\pi x)$, ..., $\phi_M(x) = \cos(M\pi x)$. This is again calculated for multiple values of M and compared in Figure 3. Interestingly, even when we use the same family of basis functions as used to generate the initial data ($M = 2$), due to the noise $\epsilon(x)$ added to the initial dataset, the maximum likelihood weight vector does not identically match the actual function used:

$$w = \begin{bmatrix} 1 \\ 1.5 \end{bmatrix}$$

$$w_{MLE} = \begin{bmatrix} 0.779 \\ 1.174 \end{bmatrix}$$

where w is the actual weight vector and w_{MLE} is the maximum likelihood estimated weight vector.

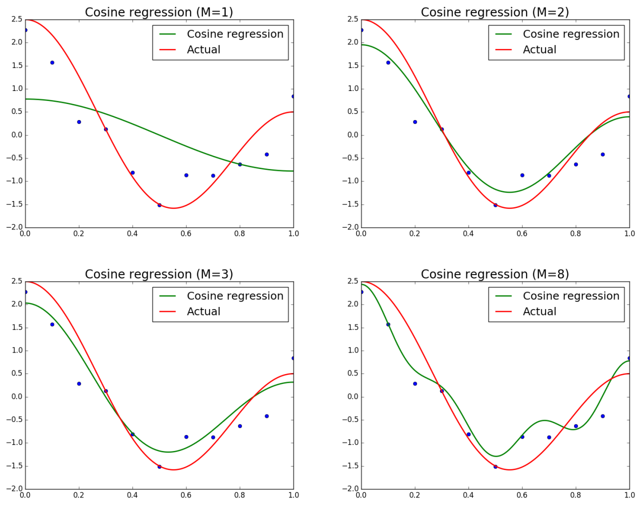

Figure 4. Linear regression for varying values of M.

3. Ridge regression

As illustrated in the previous section, some values of M can result in overfitting to the training set. Thus, in an attempt to reduce overfitting, we implement ridge regression, which adds a regularization parameter λ to the regression function. In this case, we have chosen a regularizer to serve as a weight decay, in order to encourage weight vector values to tend towards zero unless otherwise supported by the data. For any given value of M therefore, with greater weight decay regularization coefficients, the weight vector decays more strongly towards zero. In other words, for higher values of λ, the data must more strongly support greater weight vector values in order to acehive the same magnitude of coefficients in the weight vector.

This is illustrated in Figure 4, where the first row illustrates the fit of ridge regression using a polynomial basis of degree 3 with $\lambda = 0.1$ and 1 and the second row shows the fit of the ridge regression using a polynomial basis of degree 10. As illustrated, the weight vector coefficients in the regression are dampened towards zero with greater values of the regularization coefficient. This is compared with the regression computed with a $\lambda = 0$ to illustrate the decay effect of the regulaization parameter.

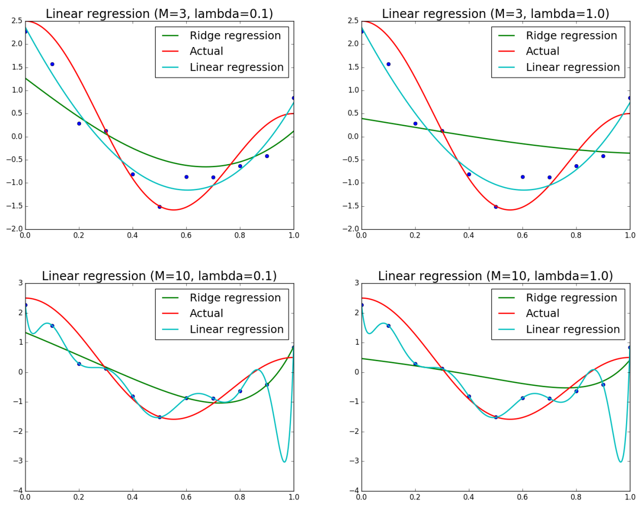

Figure 5. Ridge regression for values of M and λ.

Using additional data sets for validation and testing, we can select for the best M and λ parameters in our regression model using a polynomial basis. More specifically, for many combinations of values of M and λ, the maximum likelihood weight vector w_{MLE} was calculated based on the training data set. Through model selection on the validation data set, the best model and optimal values for M and λ were chosen based on the evaluation metric. This optimal regression model was then run on the testing data set to evaluate the performance of the model on the new data. The maximum likelihood weight vectors were generated for all $M \in [0, 10]$ and $\lambda \in \{0, 0.1, 0.2, ..., 1.4, 1.5\}$, and this was performed multiple times with data set A as the training data and data set B as the testing data, as well as vice versa. The performance of the regression model was evaluated using the sum of squared errors (SSE) and the mean squared errors (MSE); qualitatively, both evaluation metrics result in the same conclusions, although quantitatively, some additional information can be gleaned from each evaluation metric.

When data set A was used as the training data and data set B was used as the testing data, the validation step indicated that $M = 2$ and $\lambda = 0.0$ yielded the optimal regression model with a minimum SSE of 2.35 and a minimum MSE of 0.11. Running this model on the testing data demonstrated that the regression fit the testing data remarkably well, excepting a single outlier in the testing data. Thus, despite a relatively low MSE of 2.58 on the testing data, the SSE was relatively high at 25.75 due to the significant error in the outlier data point. The training, validation, and testing steps are illustrated in Figure 6, in which the best regression line chosen in model selection is plotted against each data set used for training, validation, and testing, respectively.

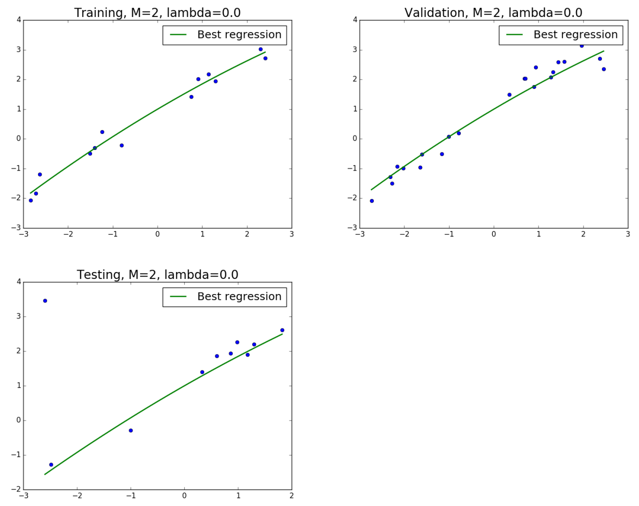

Figure 6. Training on data A, testing on data B.

When data set B was used as the training data and data set A was used as the testing data, the validation step indicated that $M = 3$ and $\lambda = 0.6$ gave the optimal regression model with a minimum SSE of 28.10 and minimum MSE of 1.28. In this case, as the model and maximum likelihood weight vectors were computed based on the data set with the outlier data point, the model was trained to try to account for this outlier data point as well. Thus, during the validation step during model selection, the minimum SSE and MSE were significantly greater than when the data set without the outlier was used for training. This can additionally be seen in the testing step, in which the SSE was 40.09 and the MSE was 3.08. This regression model on the training, validation, and testing data sets is again visualized in Figure 7.

Under both evaluation metrics (SSE and MSE), we can conclude that the polynomial basis yielded better linear regression when trained on data set A and tested on data set B. By examining and comparing both metrics however, we can gain additional information regarding the spread of the data and any potential outliers. This makes intuitive sense when considering the roles of training, validation, and testing and allows us to draw conclusions about the relationships and relative importance of each step with respect to

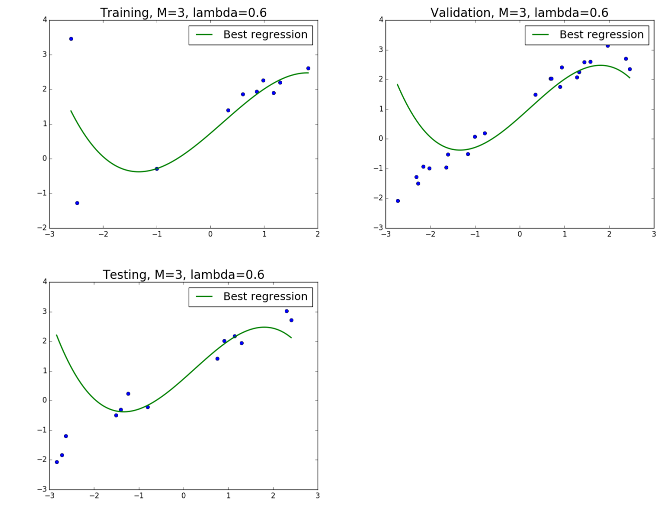

Figure 7. Training on data B, testing on data A.

each other.

4. Sparsity and LASSO

In this section of our research, we begin to tackle problems in which the closed-form solutions do not exist. In this case, we specifically used the LASSO estimator to approximate sparse weights for a dataset generated by the function

$$y = w_{true}^T \phi(x) + \epsilon,$$

(where ϵ is a small amount of noise), working with a sin basis of the form:

$$\phi(x) = (x, \sin(0.4\pi x \times 1), \sin(0.4\pi x \times 2), ..., \sin(0.4\pi x \times 12))$$

By using LASSO with varying values of λ, we can seek to optimize our resulting weight vector for the sine basis with respect to sparseness. Sparseness, the property in which there are few non-zero components of the weight vector, is of particular interest to us because it minimizes the ability for the weight vector to overfit the data. Furthermore, as a sparse feature vector is weighted by some but not all components, it is less suspectible to noise in the data. In this case, LASSO is able to attain sparseness by making the regularization penalty λ use the L_1 norm.

We used the `sklearn` package to implement LASSO, and we found that we were indeed able to produce a sparse approximated weight vector for the data: [0, 0, 5.411, 0.476, 0, 0.994, 2.357, 0, 0, 0, 0, 0, 0]. Comparing this with the true weight vector w_{true}^T, which was given as [0, 0, 5.646, 0.779, 0, 0.811, 2.683, 0, 0, 0, 0, 0, 0], we can see that in both weight vectors, there are only four non-zero components. In contrast, when $\lambda = 0$, we get a weight vector in which all component weights are non-zero.

Figure 7 outlines the results of these various approaches for different values of λ. At $\lambda = 0.1$, the LASSO curve approximates the actual curve surprisingly well despite minor differences. For $\lambda = 0$, in which the regularization penalty is not taken into account, the resulting curve has higher variability and tries to overfit the data too much. Thus, despite not having a closed-form solution in this scenario, we can employ LASSO and principles of sparsity to still generate an acceptable approximation and regression of the data.

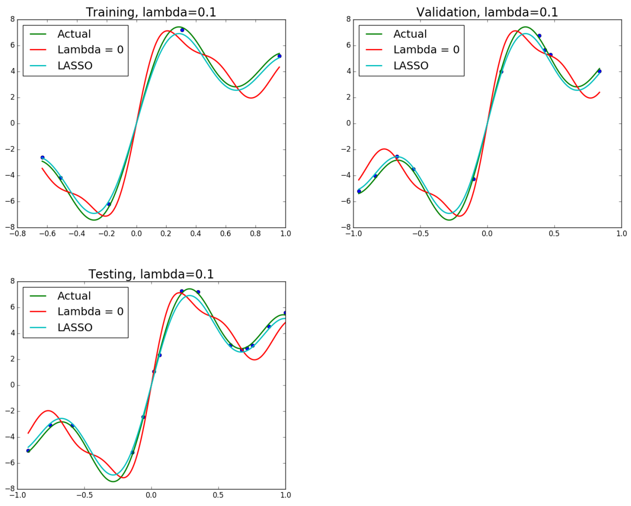

Figure 8. LASSO with varying values of lambda.

Comparing ridge regression with LASSO, we can see how both approaches seek to minimize overfitting in our approximation to the data. Unlike ridge regression which minimizes the component weights of the weight vector however, LASSO allows some weights to take values not close to 0 and instead focuses on optimizing for sparseness. These results, comparing ridge regression versus LASSO, are illustrated in Figure 8.

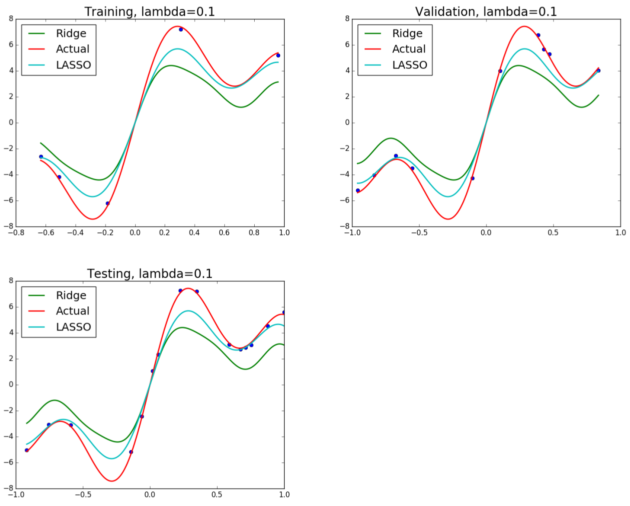

Figure 9. Ridge regression vs. LASSO.