1. Implement Gradient Descent

A general-purpose gradient descent Python library was created which provides for specification of x_0 (the initial guess), η (the step size), and γ (the convergence threshold: if two consecutive steps differ by less than this value, the algorithm terminates). The gradient descent procedure was tested on two functions with well-known optimal values: (i) a non-convex polynomial $f(x)$, and (ii) a negative bivariate Gaussian $p(x)$:

$$f(x) = x^4 - x^3 - x^2$$

$$p(\mathbf{x}) = \frac{-100}{2\pi|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \mathbf{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}) \right)$$

where in $p(\mathbf{x})$, $\mathbf{\Sigma}$ represents the covariance and $\boldsymbol{\mu}$ the mean. Note also that the standard Gaussian has been multiplied by -1 (so that we can pose our optimization as a minimization) and scaled by 100, which is just to help with making plotting more clear.

The effects of initial guesses (x_0) and varying step size (η) are best explained via the plots. For the non-convex polynomial, $f(x)$, the effect of the starting position can be seen from the three plots in Figure 1. Initial guesses $x_0 > 0$ will terminate at the global maximum of the function, whereas initial guesses $x_0 < 0$ get "stuck" at the local minimum in $\mathbb{R}^-$. An initial guess of exactly a local maximum $x_0 = 0$ leads to the algorithm terminating still with $x_{final} = 0$ after only a handful of iterations, since initially $f'(x = 0) = 0$. It is nice to note that local maxima can cause algorithm termination with poor initial guesses, but in practice adding randomization to initial guesses helps mitigate this problem. The effect of the step size can be viewed by observing the contour plots for gradient descent on $p(\mathbf{x})$, the bivariate Gaussian. In plots (a) and (b), the step size is sufficiently small and successful descent occurs to the global minimum. In plot (c), however, the step size is sufficiently large such that the algorithm continually jumps "back and forth" over the minimum, and the algorithm terminates only due to the maximum number of function calls ($n_{max} = 5000$ was used). As expected, as long as the convergence criterion is significantly small, decreasing the convergence criterion further only increases the number of iterations required to converge (plot not shown for brevity).

Numerical gradients were also implemented by calculating finite differences. Note that for numerical gradients, the required amount of function evaluations per gradient descent update step increases by a factor of $2d$, where d is the dimension of the input. A comparison of results for analytical vs. numerical gradients is provided in Tables 1 and 2. For all points tested, analytical and numerical gradients

Function and Initial Guess, x_0	Analytical	Numerical	SciPy
Non-convex $f(x)$			
$x_0 = 1.9$	130	646	24
$x_0 = -1.0$	312	1556	36
$x_0 = 0.0$	2	6	3
Neg. Gaussian $p(x)$			
$x_0 = (1.0, 1.0)$	177	1603	48

Table 1. Number of function calls for our gradient descent implementation using analytical vs. numerical gradients, along with results for SciPy's optimizer.

resulted in the same termination value, although numerical gradients typically 5 times the function calls for the 1-dimensional function, and 10 times the function calls for the 2-dimensional Gaussian. Since these factors are more than $2d$, numerical gradients also required more steps.

The comparison of our gradient descent implementation with `scipy.optimize` is interesting for a couple of reasons. For one, despite best efforts to tune step sizes and convergence criterions, there is no way that our optimizer can match the SciPy optimizer in terms of required function calls. SciPy's default BFGS (Broyden-Fletcher-Goldfarb-Shanno) optimization method, which we used, provides a more sophisticated method of descent than our simple first-order finite differencing. We did not provide any analytical gradients to the optimizer, and so it computed them numerically. Comparing apples to apples, our numerical gradient descent required a factor of approximately 50 times more function calls to converge, and even our analytical gradient descent required 3-10 times more function calls. Despite its fast convergence performance, however, the SciPy optimizer did behave oddly on two of our tested initial conditions for the non-convex polynomial $f(x)$. As shown in Table 2 (no, this was not a typo), the SciPy optimizer returned the non-global minimum when the initial guess was in the basin of the global minimum, and returned the global minimum when it was in the basin of the non-global minimum. This suggests `min(gradDescent, spicy.optimize)` is the safest optimization algorithm for our tested results.

2. Linear Basis Function Regression

We implemented the linear basis function regression in Python and were able to get good agreement of our plots and regression weights with those in Bishop. We omit these plots for brevity. The closed form solution to the linear least squares problem is convenient, however it requires inverting the matrix $\Phi^T\Phi$. An alternative to avoid
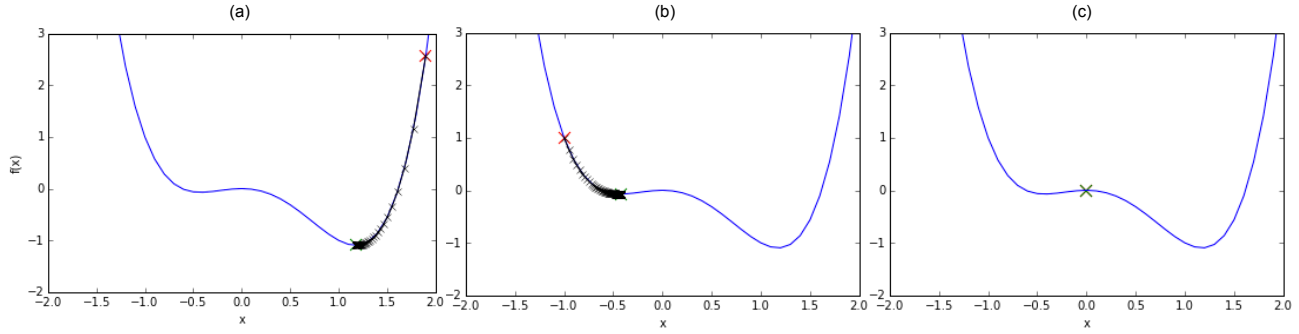
Figure 1. Visualization of gradient descent for three initial guesses, x_0, of the non-convex polynomial $f(x) = x^4 - x^3 - x^2$. Initial guesses are: (a), 1.9; (b) -1.0; (c), 0.0. The red X marks the initial guess, the green X marks the algorithm's final value, and the black X represent sequential values produced by each gradient descent step. Plot (a) converges to the global minimum, while (b) converges to a local, non-global minimum. Plot (c) remains at the initial guess, since $f'(x_0) = 0$. Shown are results using: analytical gradients, $\eta = 0.02$, $\gamma = 0.0004$.
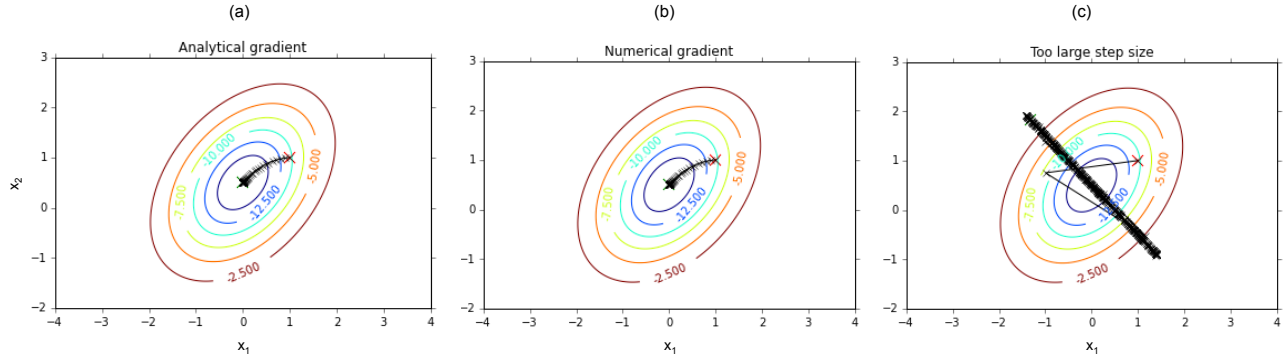

Figure 2. Visualization of gradient descent for the negative bivariate Gaussian $p(x)$, plotted via contours. The red X marks the initial guess x_0, the green X marks the algorithm's final value, and the black X represent sequential values produced by each gradient descent step. Plots (a) and (b) show a comparison of using the analytical vs. numerical gradients: note although the descent path looks similar, the number of function calls is an order of magnitude different. Plot (c) shows the path of gradient descent when the step size is too large ($\eta = 0.2$). Unless otherwise noted, all plots used: $x_0 = (1.0, 1.0)$, $\eta = 0.01$, $\gamma = 0.0004$.

Function and Initial Guess, x_0	Analytical	Numerical	Scipy
Non-convex $f(x)$			
$x_0 = 1.9$	1.17	1.17	-0.43
$x_0 = -1.0$	-0.43	-0.43	1.17
$x_0 = 0.0$	0.0	$2.5e-13$	0
Neg. Gaussian $p(x)$			
$x_0 = (1.0, 1.0)$	$(0, 0.5)$	$(0, 0.5)$	$(0, 0.5)$

Table 2. Minimum value found via our gradient descent implementation using analytical vs. numerical gradients, along with results for SciPy's optimizer.

this matrix inversion is to apply gradient descent to the sum of squared error (SSE) objective function given by $(\Phi w - y)^T (\Phi w - y)$. Differentiating the SSE with respect to w gives a gradient of $2\Phi^T(\Phi^T w - y)$. We checked this analytical gradient against our numerical gradient implementation. Using $h = 1e - 5$ in the central difference computation led to agreement between analytical and numerical gradient to with $1e - 8$ for $M = 3$. Using this analytical gradient we can apply our gradient descent code from Problem 1. Our initial parameter choices are $\eta = 0.05, \gamma = 1 \times 10^{-8}$. The initial guess is set to $w_0 = -w_{OLS}$ where w_{OLS} are the true regression weights. For $M = 1$ gradient descent works quite well.

Solver	Function Calls	Weights
Gradient Descent	113	$(0.820, -1.267)$
SciPy	20	$(0.820, -1.267)$
OLS	-	$(0.820, -1.267)$

Our gradient descent method converges to essentially the same regression weights as `scipy.optimize` and the true OLS regression weights, albeit with an order of magnitude more function calls. This is to be expected as the SSE function is convex in w and hence there is a unique global minimum which can be reached by "following" the gradient downhill. Changing the initial guess to all zeros produces very similar performance in terms of objective value and number of function calls. With $M = 3$ however the performance of our solver changes dramatically however. In particular, if w_{OLS} denote the true regression weights then: *(see table below)*

Solver	Function Calls	$|w - w_{OLS}|_2$	η
Grad. Desc.	$38,513$	0.14	0.05
Grad. Desc.	$72,657$	0.199	0.025
SciPy	114	4×10^{-5}	-

Thus increasing the dimension of the optimization problem from 2 to 4 exposes the weaknesses of our gradient descent method. In particular it uses about 300 times more function calls. The reason is that as we approach the optimum the SSE function becomes very flat and hence the gradient ∇SSE becomes very small. Hence, in our update step $w_{n+1} = w_n - \eta * \nabla SSE(w_n)$ the amount we move, $\eta * \nabla SSE(w_n)$ is becoming arbitrarily small. In particular if we plot function value vs iterations (Figure 3) we see that the plot becomes very flat quite quickly. Thus for most of the iterations of the algorithm we aren't really decreasing the value of the objective very much.
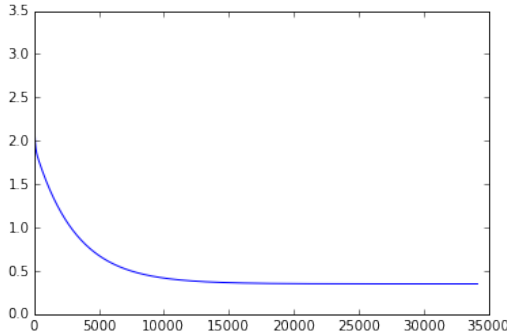

Figure 3. Value of SSE at each iteration of our gradient descent algorithm, $\eta = 0.05$, $w_0 = 0$

This is a result of our update step not moving us enough when the gradient becomes small. Reducing η to 0.025 just exacerbates this problem. In this case we require more iterations and achieve worse fit of the regression weights. If we try increasing $\eta = 0.07$ however the opposite happens. We adjust w_{n+1} by too much and end up jumping to the other side of the "bowl" representing the SSE. Hence we end up bouncing back between different sides of the bowl and the solution ends up exploding. The effect of the inital

guess for $M = 3$ isn't too important. Using $\eta = 0.05$ and choosing initial guess to the origin results in the number of function calls decreasing to about $34,000$ and doesn't change the achieved accuracy of the solution. This makes sense because since we are optimizing a convex function we have no risk of getting stuck in local minima and thus independent of where we start we should be able to follow the gradient down to the minimum.

The results for $M = 9$ are very interesting as well. The value of $SSE(w_{OLS})$ is $1e - 7$. If we set the initial guess for the SciPy minimization to be $w_0 = 1.5 * w_{OLS}$ then it achieves a minimum value of 0.09 however if we set $w_0 = 0$ then the method only achieves $SSE = 0.314$. Our own gradient descent method behaves similarly. Thus in this case the performance of the gradient descent method is quite sensitive to the initial guess. A graphical illustration of the difference in the regression function can be seen in Figures 4 and 5 and the difference in fit is quite striking. Thus the initial guess for gradient descent has a very large impact on the shape of the regression function for the $M = 9$ case. We believe that this is because since we
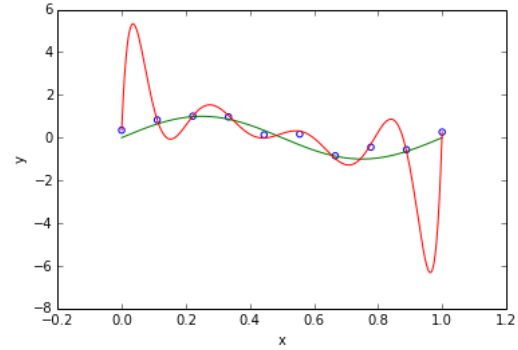

Figure 4. Gradient descent with $M = 9$, $w_0 = 1.5 * w_{OLS}$. Blue are datapoints, green is true regression function, red is estimated regression function

are considering large powers of x the function is very flat near the minimum. Thus gradient descent methods have a hard time making progress towards the minimum since the function is locally very flat.

If the basis functions were of the form $\phi_n(x) = \sin(2\pi n x)$ then we would expect a regression vector of the form $w = (1, 0, 0, ..., 0)$ since the data was actually generated from $\sin(2\pi x)$ with Gaussian noise added. A potential disadvantage is that since the sin function is periodic you are imposing periodicity on your data. In particular x and $x+1$ will map to the same value. In addition the set of functions $\mathcal{B} = \{\sin(2\pi m x)\}_{m \in \mathbb{N}}$ aren't a basis for the space of functions. You need both sin and cos to form a basis. Thus there are functions, and hence datasets that are generated from such functions, that could not be well approximated
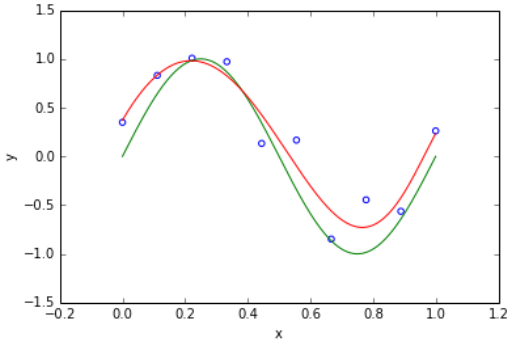
Figure 5. Gradient descent with $M = 9, w_0 = 0 * w_{OLS}$. Blue are datapoints, green is true regression function, red is estimated regression function

by the basis $\mathcal{B}$.

3. Ridge Regression

As we saw in the previous section $M = 3$ provides a fairly good fit to the data. As can be seen in Figure 6 using $M = 3, \lambda = 0.01$ doesn't provide good performance.
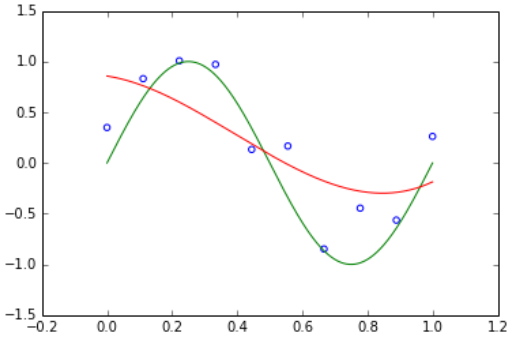

Figure 6. Ridge regression with $M = 3, \lambda = 0.01$. $SSE = 1.54$. Blue are datapoints, green is true regression function, red is estimated regression function

In particular the λ weight is too high and keeps the regression weights close to zero, which results in a relatively flat nearly linear fit. Reducing λ to 0.001 as in Figure 7 produces better results, as can be seen by comparing the SSE's since it doesn't penalize the weights w as much for being far from zero. The interesting thing is that increasing M while keeping λ fixed doesn't adversely affect performance. In particular the prediction curve in red looks fairly similar even as we increase the number of features to $M = 9$ in Figure 8. Thus the regularizer λ helps us to avoid overfitting. In particular there is a very large difference between the fit between Figures 8 and 4.

For the train, validate and test datasets we perform the following model selection procedure. Given (M, λ) we
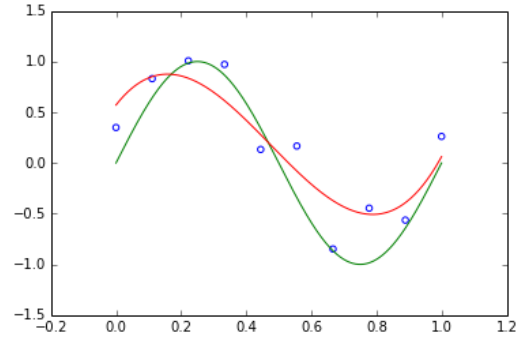

Figure 7. Ridge regression with $M = 3, \lambda = 0.001$. $SSE = 0.588$. Blue are datapoints, green is true regression function, red is estimated regression function
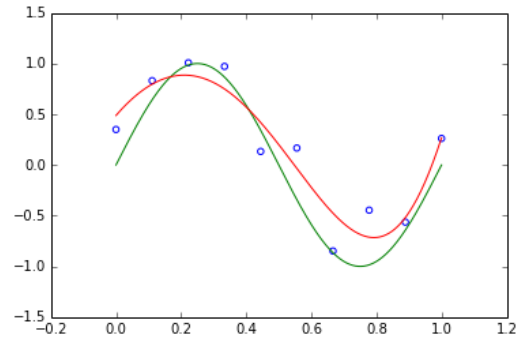

Figure 8. Ridge regression with $M = 9, \lambda = 0.001$. $SSE = 0.418$. Blue are datapoints, green is true regression function, red is estimated regression function

run ridge regression on the training dataset to find the regression weights $w(M, \lambda)$. Then we compute the SSE using weights $w(M, \lambda)$ on the validation dataset, denote this by $SSE_v(M, \lambda)$. We choose (M, λ) to minimize $SSE_v(M, \lambda)$. In essence we are using the training set to choose the weights, and then using the validation set to optimize over the model, given by (M, λ). For each M we search over $\lambda \in [0, 10]$ to minimize $SSE_v(M, \lambda)$. Then we can check how well our model is doing on the test dataset. Table 3 shows the results.

M	λ	SSE Train	SSE Validate	SSE Test
1	6.53	18.36	16.89	16.89
2	1.89	12.4	8.92	24.1
3	0.1	9.84	3.6	23.3
4	0.854	8.7	0.98	27.7
5	8.9	10.3	2.19	36.5

Table 3. SSE for training, validation and test datasets for different M. The specified λ minimizes SSE for validation set given that M.

The result of the model selection is $M = 4, \lambda = 0.85$. However this is misleading result. Consider the plot of the three datasets and the prediction function in Figure 9. Visually the data look linear. However the one outlier in the
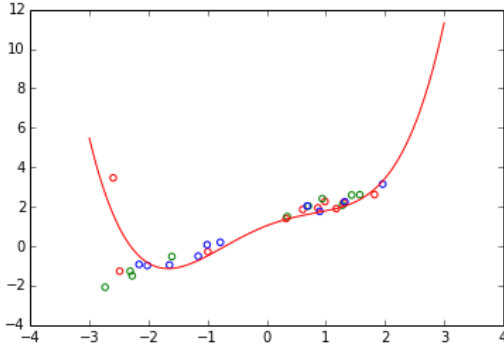

Figure 9. Ridge regression with $M = 4, \lambda = 0.85$. Train datapoints are red, validation are blue, and test are green. Estimated regression function is in red.

training set is leading to bad fits for the $M = 1$ case as can be seen in Figure 10. Hence when we perform our model
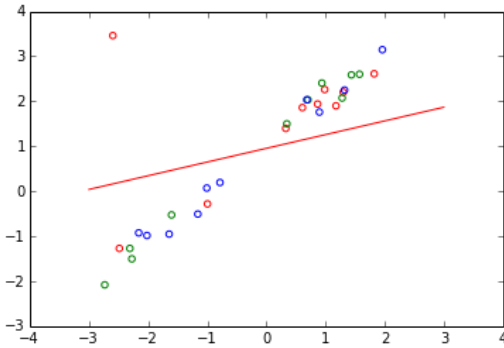

Figure 10. Ridge regression with $M = 1, \lambda = 6.5$. Train datapoints are red, validation are blue, and test are green. Estimated regression function is in red.

selection procedure we end up choose $M = 4$ rather than $M = 1$ because this choice of M allows us to roughly hit the outlier and the bulk of the validation data points. However since the test set extends to larger magnitude x values, we have a terrible fit on the test dataset, even though we get a relatively good fit on the validation data. Thus the lesson is that in the presence of outliers model selection together with ridge regression can lead us to incorrect results.

For the blog feedback there is only one parameter to choose during model selection, namely λ since the feature set Φ has already been specified for us. We ran ridge regression with a variety of λ values and to compare performance. We saw that fairly large λ around 3000 were providing the best performance. Similar to the previous analysis we then

performed a grid search over λ to choose the best model. The results are shown in Figure 11. Note that the Φ matrix is not full rank since several of the features are identically zero for all observations.
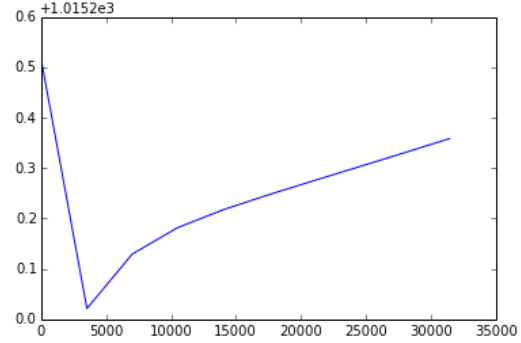

Figure 11. The x-axis is $\hat{\lambda} = \frac{\lambda}{N}$ plotted against the MSE of the validation set on the y-axis.

Interestingly the plot is very flat. In other words the MSE of the validation set isn't very sensitive to the parameter λ. So the choice of regularize λ is not having too much of an effect on the MSE of the validation data. In particular it seems that the fit of the model is much less dependent on λ than in the curvefitting example. The plot of λ against the MSE of the test set looks quite different however. It decreases fairly continuously as λ increases, see Figure 12. Looking at the actual regression weights w shows for $\lambda =$
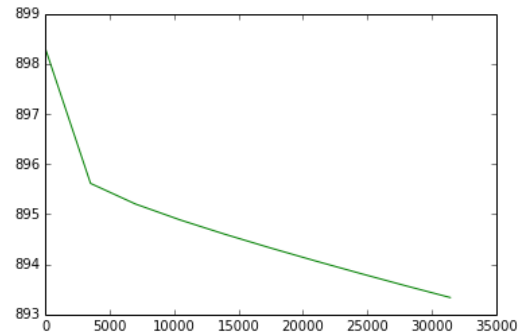

Figure 12. The x-axis is $\hat{\lambda} = \frac{\lambda}{N}$ plotted against the MSE of the test set on the y-axis.

3000 they are less than 1.5 in absolute value.

Now we compare the results to performing feature rescaling using the interval method. Namely for each feature $\phi_i(x)$ we rescale it to

$$\tilde{\phi}(x) = \frac{\phi_i(x) - \min_{x'} \phi_i(x')}{\max_{x'} \phi_i(x') - \min_{x'} \phi_i(x)}$$

where max and min are over the observed values of that feature in the dataset. Some care had to be taken because

some features were identically zero. The rescaling implies that the new features satisfy $\tilde{\phi}(x) \in [0, 1]$. Repeating the above analysis with the rescaled features gives Now we see that the MSE of the validation set is much more responsive to varying λ. However, we again notice that the plots of MSE look very different for the validation and test sets. This is somewhat surprising since neither of these datasets was used to train the regression weights. Interestingly looking at the regression weights after provides an interesting insight. Let w be the regression weights with $\lambda = 350$. Then interestingly we see that most of the values are near zero and with a few notable exceptions see Figure 13. Thus this shows that a small number of features are responsible for predicting the number of comments. Ultimately we achieved a smaller MSE on the test
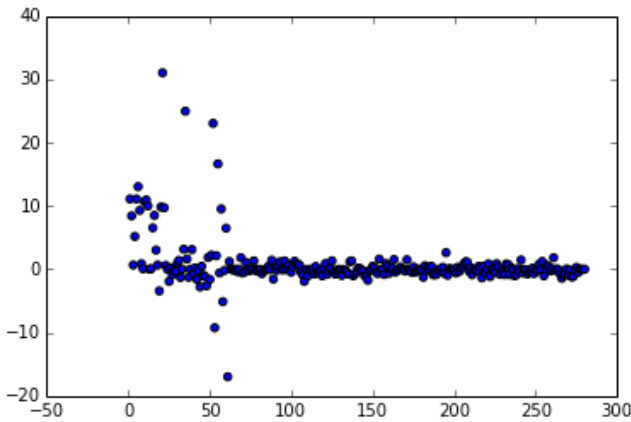

Figure 13. A plot of the regression weights vs their index. Weights calculated using the feature rescaling with $\lambda = 350$.

set when we didn't rescale the features. However, the insight we get from running the ridge regression with the rescaled features is that, as can be seen in Figure 13, a few of the features contain most of the predictive power for the number of comments. The other thing that feature rescaling affects is the role of the regularization term $\lambda\|w\|_2^2$. In particular rescaling the features also to some extent should rescale the weights w. Since the λ term affects all weights equally in the regularization term, it proportionally now drives down the weights on the "important" features from Figure 13 more than it drives down those same weights in the unregularized dataset. Ultimately it is difficult to draw too many more specific conclusions since it is challenging to visualize data with 280 features.

4. Generalizations

Problem 4.1, "Outlier and LAD," was chosen. We implemented minimization of the sum of absolute values of the errors (SAE) via gradient descent, also termed LAD (least absolute deviations) regression. In contrast to OLS,

M	λ	SAE Train	SAE Validate	SAE Test
1	0.85	6.33	2.04	2.85
2	1.11	6.53	2.08	3.16
3	1.86	6.35	4.50	8.58
4	3.57	8.47	3.57	5.24
5	8.09	9.16	3.43	9.77

Table 4. SAE for training, validation and test datasets for different M. The specified λ minimizes SAE for validation set given that M.

there is no matrix manipulation that can lead to a closed form solution for LAD. We instead must use a different approach – here, we descend the SAE objective function via gradient descent. Since the SAE is not differentiable at $|y-h(x)| = 0$, a continuous function for the analytical gradients could not be provided, and so we deferred to letting our optimizer use numerical gradients.

We repeat the model selection experiments that we performed in Problem 3, but for SAE ($|y - h(x)|$) rather than SSE ($(y - h(x))^2$). We search over (M, λ) for the best fit measured by the SAE on the validation dataset, $SAE_v(M, \lambda)$, produced from the regression weights $w(M, \lambda)$ found by LAD regression on the test set. Again, the range $\lambda \in [0, 10]$ was used. Table 4 shows the results.
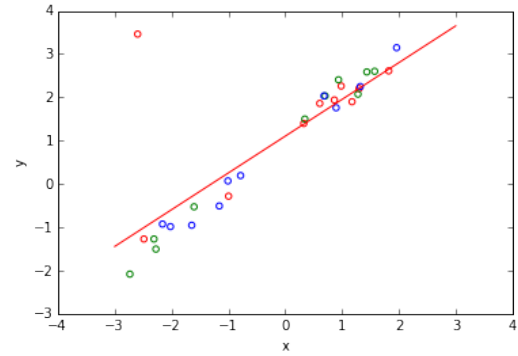

Figure 14. LAD regression with $M = 1, \lambda = 0.85$. Train datapoints are red, validation are blue, and test are green. Estimated regression function is in red.

The choice of $M = 1$ for LAD model selection contrasts with the selection of $M = 4$ for ridge regression model selection, and highlights the ability of LAD to handle outliers. Figure 14 plots the LAD fit for $M = 1$, which clearly ignores the outlier better than the ridge regression fit for $M = 1$ in Figure 10. Model selection for ridge regression resulted in $M = 4$, which as shown in Figure 9 produced a fit that responded strongly to the outlier, whereas $M = 4$ for LAD is still relatively robust to the presence of the outlier (Figure 15). By increasing M further, however, even
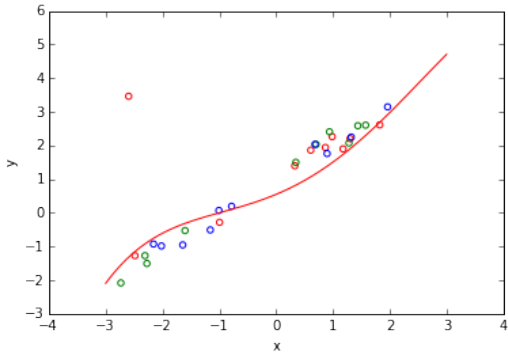
Figure 15. LAD regression with $M = 4, \lambda = 3.57$. Train data-points are red, validation are blue, and test are green. Estimated regression function is in red.

LAD is eventually tempted by the outlier: $M = 5$ for LAD produces an overfit (not shown to save space).

The data visually appear linear, excluding the outlier, and model selection for LAD naturally chooses $M = 1$. It is interesting to note that the regression weights $w_{LAD}(M = 1, \lambda = 0.85)$ outperform the w_{ridge} weights on their own metric: $w_{LAD}(M = 1, \lambda = 0.85)$ produces $SSE_v = 1.38$ and $SSE_{test} = 1.94$. The latter is significantly better than any of the fits produced by ridge regression. The experiments discussed in Problem 3 and Problem 4 suggest that when a dataset contains significant outliers that are not explicitly excluded, better generalizations may be produced with LAD than ridge regression. Explicit outlier identification and exclusion could provide another route to avoiding overfitting.