1. Implement Gradient Descent

A general-purpose gradient descent library was created which provides for specification of x_0 (the initial guess), η (the step size), and ϵ (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 f$x_0 > 0$ 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 counter 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).

Numerical gradients were also implemented by calculating finite differences. Note that each gradient descent update accordingly requires two function evaluations (per dimension...). A comparison of results for analytical vs. numerical gradients is provided in the three tables below. For all points tested, analytical and numerical gradients resulted in the same termination value, although analytical gradients were typically able to find this value in approximately 1/5th the function calls.

SAY SOMETHING ABOUT CONVERGENCE CRITERION

Number of function calls

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

Minimum x

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 $p(x)$			
$x_0 = (1.0, 1.0)$	$(0, 0.5)$	$(0, 0.5)$	$(0, 0.5)$

Minimum function value

Function and Initial Guess, x_0	Analytical	Numerical	Scipy
Non-convex $f(x)$			
$x_0 = 1.9$	-1.10	-1.10	-0.07
$x_0 = -1.0$	-0.07	-0.07	-1.10
$x_0 = 0.0$	0.0	$-6.3e-26$	0
Neg. Gaussian $p(x)$			
$x_0 = (1.0, 1.0)$	-17.36	-17.36	-17.36

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 (MAYBE INCLUDE SOME PLOTS HERE???). 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 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 $2\Phi^T(\Phi^T w - y)$. Using this analytical gradient we can apply our gradient descent code from problem 1. Our code employs a fixed step size of η and the termination criterion is as soon as $\epsilon_{n+1} = |SSE(w_n) - SSE(w_{n+1})| \leq \gamma$ where γ is a tolerance parameter. 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.

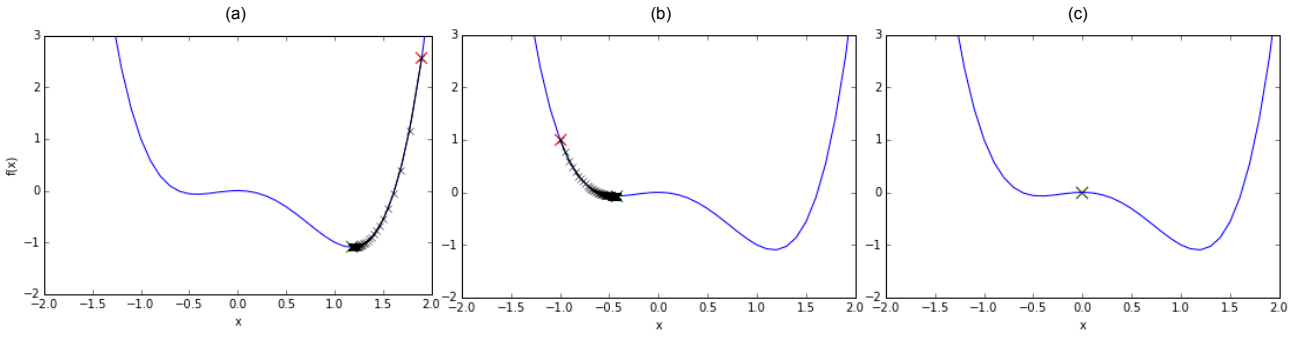

Figure 1. Visualization of gradient descent for three initial guesses, x_0, of the non-convex polynomial $f(x) = x^4 - x^3 - x^2$. Initial guesses are: (a), 1.9; (b) -1.0; (c), 0.0. The red X marks the initial guess, the green X marks the algorithm's final value, and the black X represent sequential 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, $\eta = 0.02$, $\epsilon = 0.0004$.

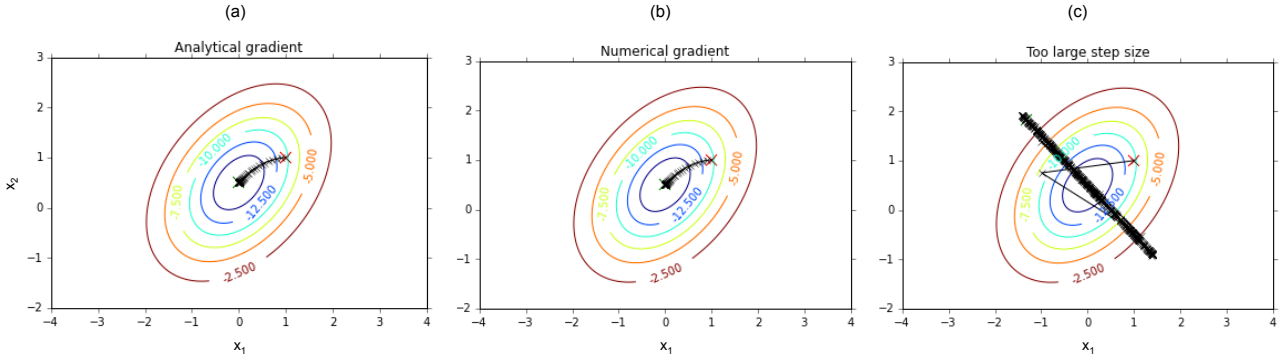

Figure 2. Visualization of gradient descent for the negative bivariate Gaussian $p(x)$, plotted via contours. The red X marks the initial guess x_0, the green X marks the algorithm's final value, and the black X represent sequential values produced by each gradient descent step. Plots (a) and (b) show 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: $x_0 = (1.0, 1.0)$, $\eta = 0.01$, $\epsilon = 0.0004$.

For $M = 1$ gradient descent works quite well.

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

Our gradient descent method converges to essentially the same regression weights as the scipy.optimize.minimize method, 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.e With $M = 3$ however the performance of our solver changes dramatically however. In particular, if w_{OLS} denote the true regression weights then

Solver	Function Calls	$\lvert w - w_{OLS}\rvert_2$	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 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 is becomes very flat quite quicky.

This is a result of our update step not moving us enough when the gradient becomes small. Reducing η to 0.025 just

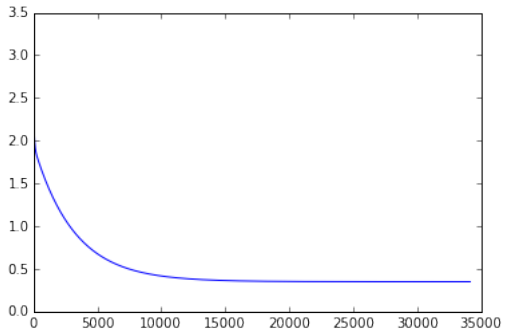

Figure 3. Value of SSE at each iteration of our gradient descent algorithm, $\eta = 0.05$, $w_0 = 0$

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 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 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 the performance of the gradient descent methods depends heavily on the initial guess. I believe that this is because since we are considering large powers of x the function is very flat near the minimum. Thus gradient descent methods have a hard time making progress since locally the function is almost 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.

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 4 using $M = 3, \lambda = 0.01$ doesn't provide good performance.

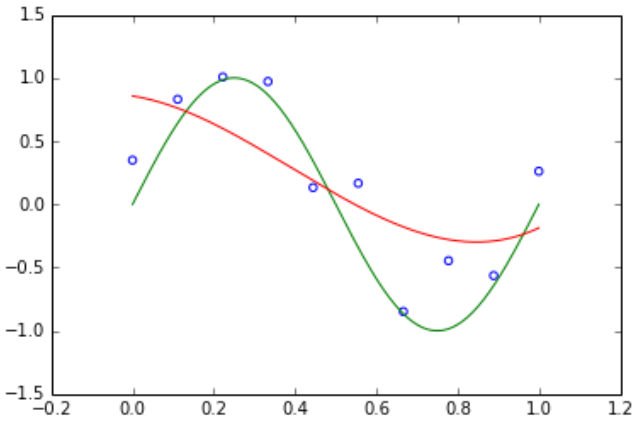

Figure 4. Ridge regression with $M = 3, \lambda = 0.01$. $SSE = 1.54$

In particular the λ weight is too high and keeps the weights close to zero, which results in a relatively flat straight line. Reducing λ to 0.001 as in Figure 5 produces better results, as can be seen by comparing the SSE's.

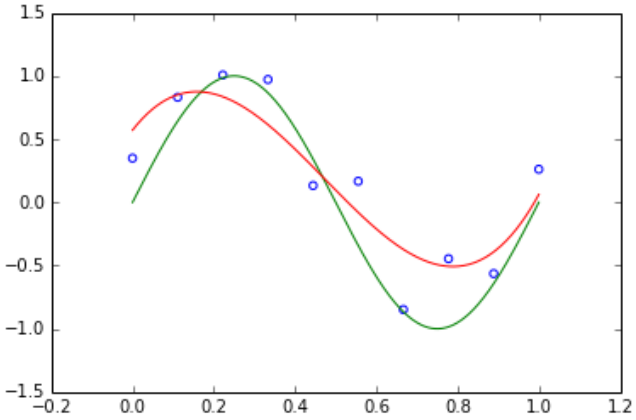

Figure 5. Ridge regression with $M = 3, \lambda = 0.001$. $SSE = 0.588$

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 6.

Thus the regularizer λ helps us to avoid overfitting, contrary to the standard OLS regression as in the previous section.

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

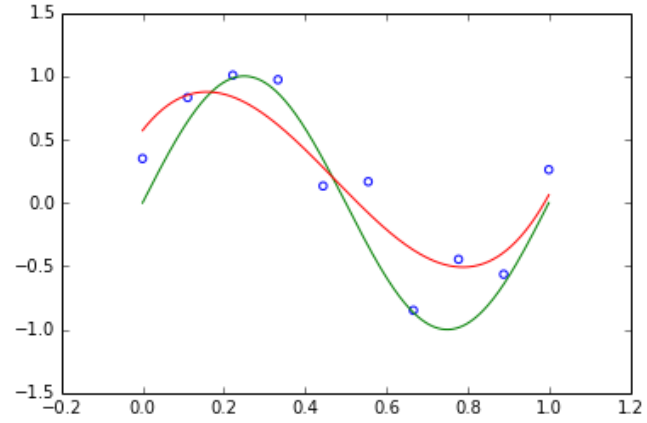

Figure 6. Ridge regression with $M = 9, \lambda = 0.001$. $SSE = 0.418$

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

$SSE_v(M, \lambda)$. 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.

The result of the model selection is $M = 4, \lambda = 0.85$. However this is misleading. If we plot the three datasets and the prediction function resulting from the weights we get Visually the data look linear. However the one outlier in the training set is leading to bad fits for the $M = 1$ case as can be seen in Figure 8

Hence when we perform our model 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 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. The objective function that is being minimized during ridge regression is

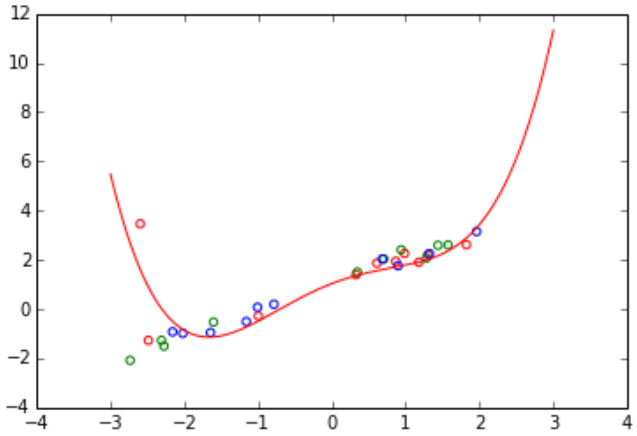

Figure 7. Ridge regression with $M = 3, \lambda = 0.85$. Train datapoints are red, validation are blue, and test are green

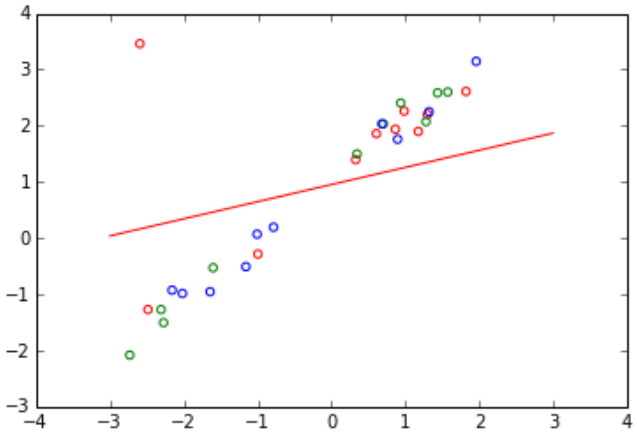

Figure 8. Ridge regression with $M = 1, \lambda = 6.5$. Train datapoints are red, validation are blue, and test are green

$$(\Phi w - y)^T(\Phi w - y) + \lambda ||w||_2^2$$
$$= \frac{1}{N}(\Phi w - y)^T(\Phi w - y) + \frac{\lambda}{N}||w||_2^2$$

Hence, it really make sense to think about $\hat{\lambda} = \frac{\lambda}{N}$ since this removes the dependence on the size of the training data. For the curvefitting dataset we had a $\hat{\lambda}$ value of around 0.1. Thus we test $\lambda \in [0.01, 2]$. The results are shown in Figure 9. Interestingly the plot is very flat. 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 $\hat{\lambda}$ against the MSE of the test set looks quite different however. It decreases fairly continuously as $\hat{\lambda}$ increases, see Figure 10. One possibility for this strange behavior could be that the data is not well modeled by the

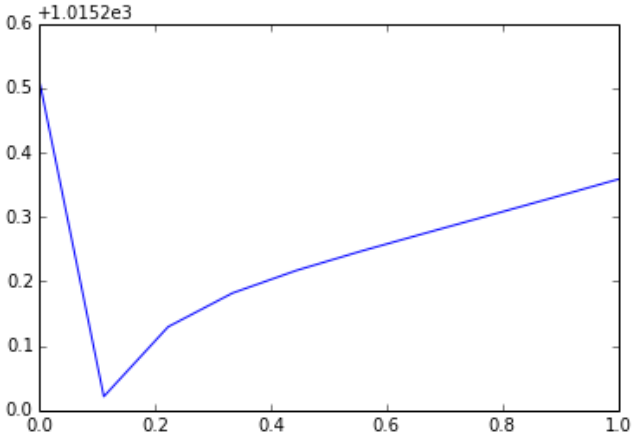

Figure 9. The x-axis is $\hat{\lambda} = \frac{\lambda}{N}$ plotted against the MSE of the validation set on the y-axis.

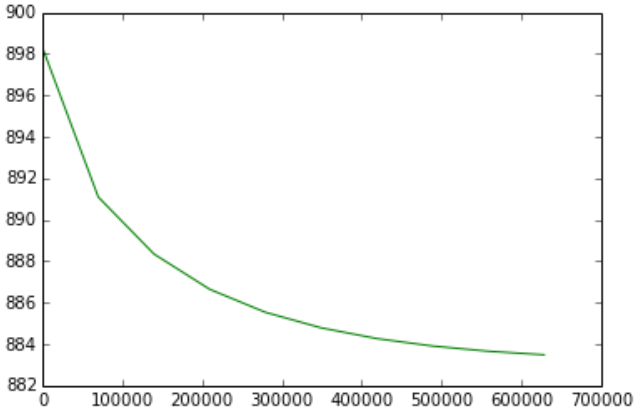

Figure 10. The x-axis is $\hat{\lambda} = \frac{\lambda}{N}$ plotted against the MSE of the test set on the y-axis.

linear model $\hat{y} = w^T x$. Glancing at the data one can see that many of the observations are zero, and then a few have quite a high value, e.g. above 50. If, for example, the true data generating process depends on interactions between features we wouldn't be able to model it well with our linear model. This could be one explanation for the strange performance of the model selection procdedure.

4. Implement Gradient Descent

A general-purpose gradient descent library was created which provides for specification of x_0 (the initial guess), η (the step size), and ϵ (the convergence threshold: if two consecutive steps differ by less than this value, the algorithm terminates).