6.857 Homework 1: Gradient Descent and Regressions

1 Introduction

In this paper, we discuss various methods of performing regressions. First, we present the stochastic gradient descent and batch gradient descent algorithms, and discuss the benefits and drawbacks of each. Next, we use gradient descent to perform a linear regression on functions with polynomial and sinusoidal bases. We then improve upon this model with ridge regression, which uses an extra weighting term to prevent overfitting. Finally, we present LASSO estimation, which optimizes this model for the case of sparse weights.

2 Gradient Descent

We start by introducing *gradient descent* iterative optimization algorithm, which calculates the parameters $\mathbf{x}$ that minimize a given objective function $f(\mathbf{x})$. The algorithm repeatedly translates an initial *guess* $\mathbf{x}_0$ in a direction proportional to the negative *gradient* $\nabla f(x)$. By moving in the direction of the negative gradient, the guess tends towards a local minimum.

In every step of the iteration, we update our guess as follows:

$$\mathbf{x}_{n+1} = \mathbf{x} - \lambda \nabla f(\mathbf{x}_n)$$

where λ is the *step size* of the iteration. The algorithm terminates upon the following convergence condition:

$$|f(\mathbf{x}_{n+1}) - f(\mathbf{x}_n)| < \delta$$

where δ is the convergence *threshold*. Upon convergence, the algorithm returns a final guess of $\mathbf{x}_{\text{opt}} = \mathbf{x}_{n+1}$.

In this section, we compare two methods of gradient descent: *batch gradient descent*, which computes a gradient over all samples for each iteration, and *stochastic gradient descent*, which instead uses pointwise gradients.

To analyze the perforamnce of the different algorithms, we use the following functions:

1. *Gaussian* function:

$$f(x) = -\frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp\left[-\frac{1}{2}(x-u)^T \Sigma^{-1}(x-u)\right]$$

2. *Quadratic bowl* function:

$$f(x) = \frac{1}{2}x^T A x - x^T b$$

3. *Least squares error* function:

$$J(\theta) = |X\theta - y|^2$$

2.1 Batch Gradient Descent

In batch gradient descent, we calculate the gradient across the entire sample set for each iteration. In this section, we analyze the affects of step size (λ), threshold (δ), and initial guess $\mathbf{x}_0$ on the performance and accuracy of batch gradient descent.

First, we note that step size has a significant impact on the convergence rate of the algorithm. To test this, we run gradient descent on an arbitrary quadratic bowl function. As we increase the value of λ from 0.0001 to 0.001, the magnitude of the gradient decreases more quickly, and the algorithm converges faster (see Figure 2.1, left). Note that in the figure, all of the gradient descents were run with a fixed size of 2000 iterations.

In addition, the step size and threshold control the accuracy of the gradient descent. As step size increases, we expect a more accurate result because the gradient descent will likely converge well within the threshold, while a smaller step size might lead to convergence at the boundary of the threshold. In addition, as the threshold decreases, we expect a more accurate result because the algorithm needs to run for longer before reaching a stable equilibrium. To verify this, we run gradient descent using a two-dimensional Gaussian with mean $(10, 10)$. As we vary λ and δ, we plot the least-squares error

between the returned value and the actual optimum of $(10, 10)$ (see Figure 2.1, middle). Indeed, the error decreases as λ increases and δ decreases.

Finally, we observe the affect of the initial guess on the rate of convergence. When running gradient descent on a two-dimensional Gaussian with mean $(10, 10)$, one would expect that as the intial guess approaches the optimal value, the convergence rate increases. Running the algorithm with $\lambda = 1000$ and $\delta = 10^{-11}$, we notice that the rate of convergence is much higher as the initial guess approaches the optimal value (see Figure 2.1, right).

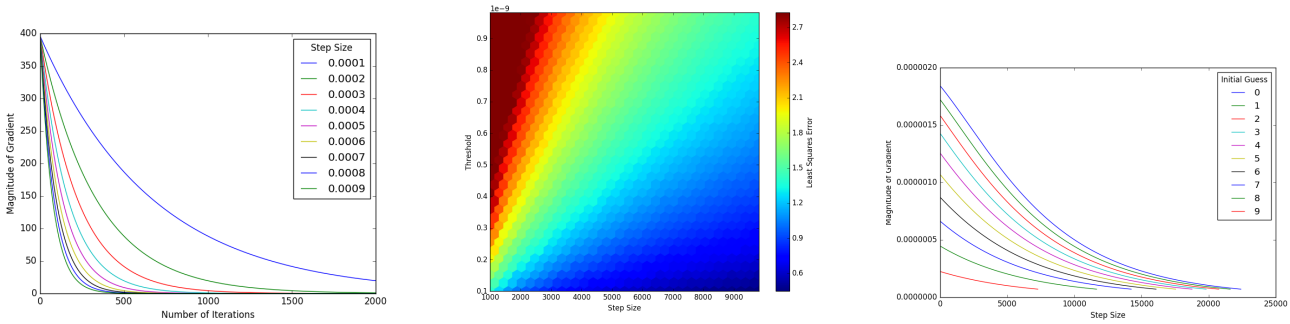

Fig. 2.1: The effects of λ (left), δ (middle), and $\mathbf{x}_0$ (right) on gradient descent.

2.2 Stochastic Gradient Descent

With large datasets, it might be inefficient to compute a gradient over the entire dataset for each iteration. One way to address this is to use stochastic gradient descent, which uses a pointwise gradient instead. In this experiment, we use least-squares regression, which has a pointwise gradient of

$$J(\theta; x^{(i)}, y^{(i)}) = (x^{(i)}\theta_t - y^{(i)})^2$$

On each iteration, we update our value as follows:

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

where $\nabla = (\tau_0 + t)^{-\kappa}$ is the *learning rate*, and τ_0 and κ are constants.

The benefit of stochastic gradient descent is that computations are quicker, which means the algorithm converges in fewer iterations. However, because each iteration utilizes less information, this comes with a decrease in performance. Refer to Figure 2.2 for the difference in convergence rate between the two methods: note that stochastic gradient descent is much noisier, and in the end, does not converge to as accurate of a value as does batch gradient descent.

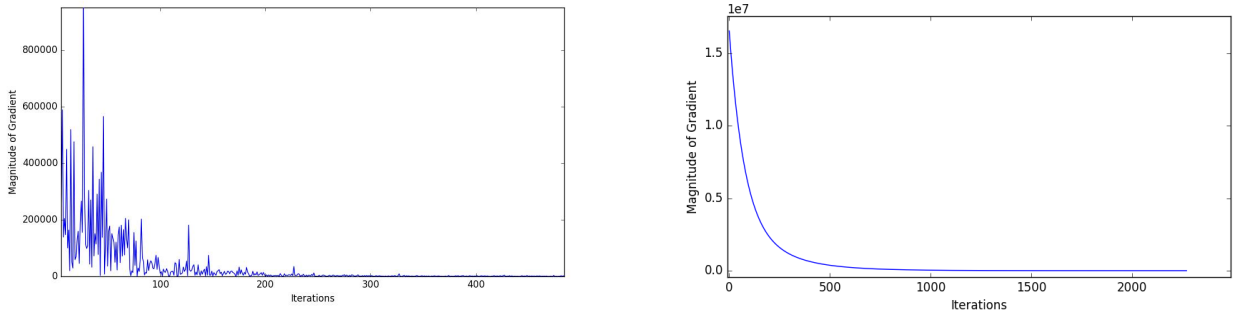

Fig. 2.2: Magnitude of the gradient vs. number of iterations. Stochastic gradient descent (left) is much noisier than batch gradient descent (right).

2.3 Effect of Finite Gradient

When the objective function doesn't have a closed form solution, one can approximate the gradient using a *central finite difference*. In other words, we have

$$\nabla f(x) \approx \nabla_h[f](x) = \frac{f(x + h/2) - f(x - h/2)}{h}$$

Therefore, we only need the objective function to approximate the gradient. Note that as h approaches zero, this approaches the actual value of the gradient; and as h increaes, the value diverges. Figure 2.3illustrates the impact of h on the accuracy of the gradient: even for large h (especially for the quadratic bowl), the approximation is accurate to several orders of magnitude.

h	$\nabla_h[f](x)$
0.01	$-3.1704 \cdot 10^{-7}$
0.1	$-3.1704 \cdot 10^{-7}$
1.0	$-3.1696 \cdot 10^{-7}$
10.0	$-3.0923 \cdot 10^{-7}$
100.0	$-2.6198 \cdot 10^{-8}$
1000.0	≈ 0

h	$\nabla_h[f](x)$
10^9	-280.000000
10^{11}	-279.999991
10^{13}	-280.000944
10^{15}	-280.349077
10^{17}	-299.759591
10^{19}	≈ 0

Fig. 2.3: The change in central finite difference as the value of h increases for Gaussian (left) and quadratic bowl (right). The real values are $-3.1704 \cdot 10^{-7}$ and -280, respectively.

3 Function Basis

Consider a basic prediction problem where given an input, x, we wish to predict a value y (e.g. predict income given GPA, or predict crime rate given temperature). We train on a sample of labeled data (a set of (x, y) pairs) to find a function $f(x)$ that will give the expected y for new inputs x.

One approach to this problem is to model f as the linear combination of *basis functions*. In other words, we manually choose functions $\phi_1(x), \ldots, \phi_n(x)$ and algorithmically pick coefficients $c_1, \ldots, c_n$, which we put together as

$$bmf(x) = c_1\phi_1(x) + \cdots + c_n\phi_n(x).$$

We can rewrite this with

$$\boldsymbol{c} = \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix}$$
$$\boldsymbol{\phi}(x) = \begin{bmatrix} \phi_1(x) & \cdots & \phi_n(x) \end{bmatrix}$$

so that

$$f(x) = \boldsymbol{c} \cdot \boldsymbol{\phi}(x). \tag{1}$$

The choice of the basis functions is somewhat arbitrary and often depend on the problem domain. Common choices are monomials ($\phi_i(x) = x^i$) and sinusoidals ($\phi_i(x) = \sin(i\pi x)$).

As for the algorithm to pick the coefficient vector $\boldsymbol{c}$, we discuss three methods: *linear regression*, *ridge regression*, and *LASSO* (least absolute shrinkage and selection operator). We explore each method over the next three sections.

Before moving on, we develop common notation to set up the algorithmic problem. Suppose we are given the training data as column vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of length m and have chosen n basis functions given by $\boldsymbol{\phi}$. We let $X = \boldsymbol{\phi}(\boldsymbol{x})$ be the m by n matrix where the i-th row is $\boldsymbol{\phi}(x_i)$ and x_i is the i-th element of $\boldsymbol{x}$. Thus, we can roughly frame our algorithm's goal as finding a column vector $\boldsymbol{c}$ of size n such that

$$\boldsymbol{y} \approx X\boldsymbol{c}, \tag{2}$$

where X and $\boldsymbol{y}$ are taken as inputs.

4 Linear Regression

Linear regression is a regression method that simply aims to minimize the square of the magnitude of the difference of the two sides of (2):

$$|\boldsymbol{y} - X\boldsymbol{c}|^2. \tag{3}$$

The expression in (3) is known as the *sum of squares error*, or SSE, as it is the sum of the squares of the differences in predicted versus real y values.

One way to find the optimal $\boldsymbol{c}$ is to expand the expression and set its partial derivative with respect to $\boldsymbol{c}$ equal to zero. Doing so yields

$$\boldsymbol{c} = \left(X^T X\right)^{-1} X^T \boldsymbol{y}. \tag{4}$$

We refer to (4) as the closed form for $\boldsymbol{c}$.

Another way to find $\boldsymbol{c}$ is to perform gradient descent with SSE as the objective function of $\boldsymbol{c}$.

We will illustrate and contrast these methods with an example, and then show the effects of choosing a different function basis. Our example is generated by taking $\boldsymbol{x}$ as numbers from 0 to 1 at intervals of 0.1 and

$$\boldsymbol{y} = \cos(\pi \boldsymbol{x}) + \cos(2\pi \boldsymbol{x}) + \text{noise}.$$

4.1 Polynomials with Closed Form

We use $\boldsymbol{\phi}$ of size $n+1$ whose components are monomials of order 0 through n as our funciton basis. Using this to compute X and evaluate the closed form, we can obtain $\boldsymbol{c}$. We view the results as a plot of the training data points, the generating curve (without noise), and the predictor $f(x) = \boldsymbol{c} \cdot \boldsymbol{\phi}(x)$. This is shown in Figure 4.1 for several values of n.

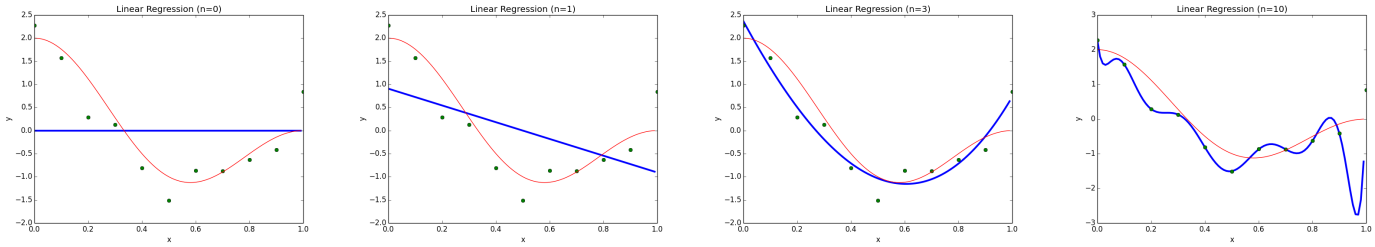

Fig. 4.1: Training data (green), generating curve (red), and predicted curve (blue) for $n = 0, 1, 3, 10$.

We can see that increasing the dimension of the function basis improves the training accuracy of our predictor.

4.2 Polynomials with Gradient Descent

In this section, we compare the performance of batch and stochastic gradient descent on linear regression. See Figure 4.2 for the performance of the two gradient descent algorithms on a polynomial function. It appears that, even for high values of M (the degree of the polynomial), the results of either gradient descent are very inaccurate – in fact, they essentially look quadratic. This is because the values of x^k have extremely small values as k increases for $|x| < 1$. Therefore, the gradient has a very small effect for higher degrees, while smaller degrees (i.e. $k = 0, 1, 2$) have a much larger impact.

To counteract this effect, the learning rate for the stochastic gradient descent needs to be as low as possible for. In this case, we chose $\nu_t = (r + 0.1)^{-0.51}$; with parameters any larger, the result has very little correlation with the actual function. This is likely because a large learning rate would cause smaller-degree terms to have an relatively large weight during the first few iterations, which would overtake the higher-degree terms. Having a smaller, slower-growing learning rate lets the higher-degree terms affect the result for more iterations, and therefore the result more closely fits the input data.

Finally, initial guess and threshold have very little impact on the accuracy of the gradient descent; they simply have an affect on performance.

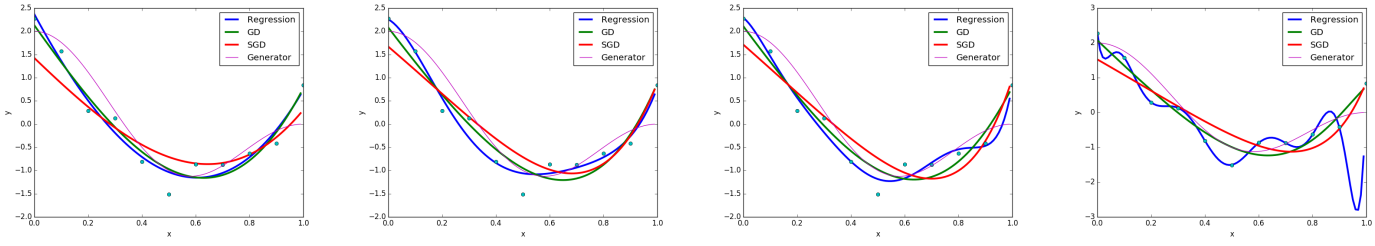

Fig. 4.2: The results for GD and SGD for $M = 3, 5, 7, 10$ (in that order).

4.3 Cosines with Closed Form and Gradient Descent

We now try using the function basis consisting of $\cos(\pi x), \cos(2\pi x), \ldots, \cos(n\pi x)$. In this case the closed form and both versions of gradient descent all give the same coefficient vectors after converging. We plot in Figure 4.3 comparisons for a couple values of n.

A notable observation is that even when the same basis functions were used as in the generating function ($\cos(\pi x)$ and $\cos(2\pi x)$ when $n = 2$), we obtained a slightly different predicted curve. This is due to the added noise.

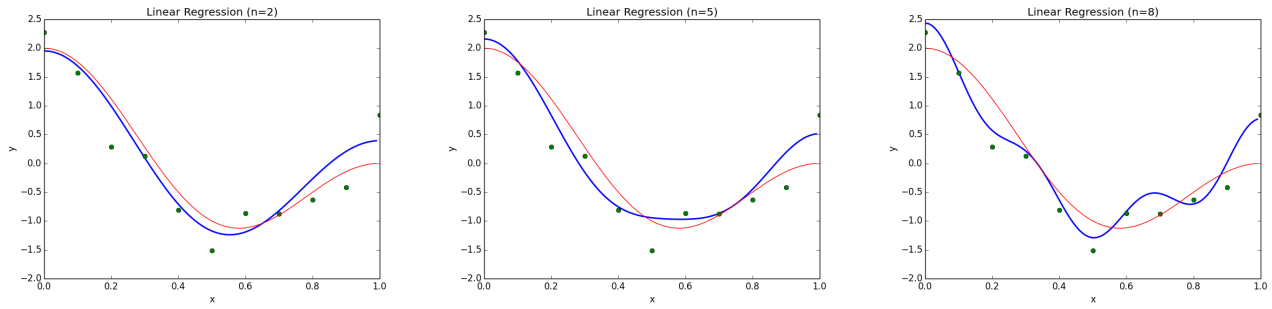

Fig. 4.3: Training data (green), generating curve (red), and predicted curve (blue) for $n = 2, 5, 8$.

Also, when we increase the dimension of the basis ($n = 8$) the predicted curve becomes more complex. Although it is more accurate for the training data, it is more sensitive to the noise and so wouldn't generalize well to new test data. This behavior is called *overfitting*, which results when the coefficients become too large or when too many features (basis functions) are used. We will see below that these issues can be addressed using ridge regression and LASSO, respectively.

5 Ridge Regression

Ridge regression is similar to linear regression in that its choice of $\boldsymbol{c}$ aims to minimize SSE, but in addition also aims to minimize the size of the coefficients themselves. Tangibly, this is achieved by choosing $\boldsymbol{c}$ that gives the minimal value of

$$|\boldsymbol{y} - X\boldsymbol{c}|^2 + \lambda |\boldsymbol{c}|^2 . \tag{5}$$

We can tune the sensitivity to coefficient sizes by adjusting the parameter λ. At the extreme, $\lambda = 0$ will have no consideration of coefficient size and will be the same as linear regression.

Again, there are two ways to arrive at the desired $\boldsymbol{c}$ from here. First, we can expand (5) and set the partial derivative with respect to $\boldsymbol{c}$ equal to zero, which gives

$$\boldsymbol{c} = \left(X^T X + \lambda I\right)^{-1} X^T \boldsymbol{y}.$$

Second, we can perform gradient descent and converge to the optimal value of $\boldsymbol{c}$ numerically.

In general, the advantage of the second term in the objective function is that resulting coefficients are smaller, thus empirically less sensitive to noise and more accurate on unseen data.

5.1 Continuation of Simple Example

In this subsection we apply ridge regression to the same example as the previous section, in which linear regression showed some overfitting. Varying λ and n gives us the plots in Figure 5.1.

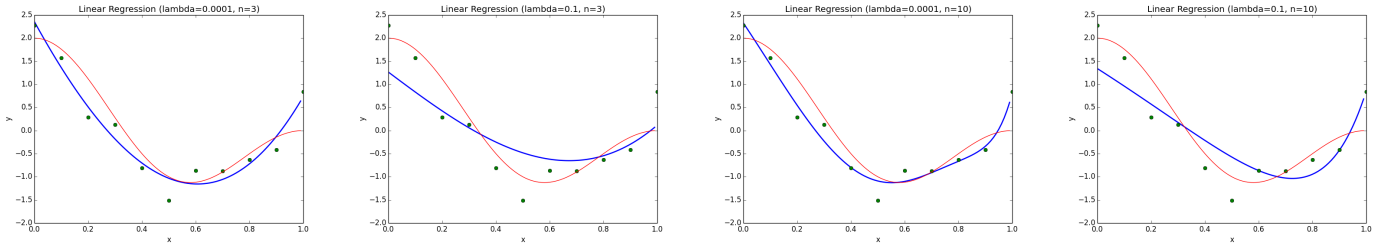

Fig. 5.1: Training data (green), generating curve (red), and predicted curve (blue) for $\lambda = 0.0001, 0.1$ and $n = 3, 10$.

We can see that a good value of λ gives a decent fit, while λ too large results in *underfitting*, where the prediction is not sensitive enough to the training data.

5.2 Training and Validation

As we just illustrated, choosing a good value of λ is crucial for generalizing well. We now present a process for finding a good value for λ (as well as n). Such a process is called *model selection*.

We divide the given labeled data into two groups, one called the training set and one called the validation set. We vary n and λ, and for each combination we train as above on the training set and compute SSE on the validation set. Finally, we select the combination of parameters that resulted in the smallest validation SSE.

To illustrate in an example, we have three sets of data: set A, set B, and a validation set. We perform the above process twice, once training on A and testing on B, and once vice versa. In both cases we try a variety of values of λ and n then evaluate the SSE on the validation set. The results are given in Figure 5.2 The SSE for the first case with $n = 2$ and

	1e-05	0.0001	0.001	0.01	0.1
0	62.00	62.00	62.00	62.01	62.06
1	2.90	2.90	2.90	2.90	2.94
2	2.35	**2.35**	2.35	2.36	2.43
3	2.37	2.37	2.37	2.39	2.55
4	3.58	3.58	3.58	3.59	3.67
5	4.14	4.14	4.14	4.12	3.97
6	3.98	3.98	3.98	3.96	3.89
7	4.07	4.07	4.06	4.03	3.89

	0	0.3	0.6	0.9	1.2
0	72.44	70.95	69.63	68.47	67.44
1	35.19	34.09	33.14	32.34	31.67
2	38.53	38.65	38.76	38.87	38.96
3	30.30	28.46	**28.09**	28.47	29.21
4	130.20	67.54	51.63	46.17	44.35
5	625.43	368.40	287.26	242.74	212.14
6	680.05	533.60	481.26	440.20	406.68
7	492.85	360.39	448.42	544.91	634.85

Fig. 5.2: SSE values on validation set after training with various n (rows) and λ (columns). Left: training on set A. Right: training on set B. Best values bolded.

$\lambda = 0.0001$, tested on set B, is 25.752. In the second case, tested on set A with $n = 3$ and $\lambda = 0.6$, the SSE is 40.087483697. Testing with parameters $n = 4$ and $\lambda = 0.1$ for both cases, which the table would predict to be suboptimal, indeed gives bigger test SSEs of 31.1388659319 and 64.9392925156, respectively.

We can see in this example that the choice of training set matters. Also, we note that the best values for the parameters n and λ depends heavily on the given data.

6 LASSO