Lab 18

Gradient Descent Methods

Lab Objective: *Many optimization methods fall under the umbrella of descent algorithms. The idea is to choose an initial guess, identify a direction from this point along with the objective function decreases, and preform a line search to find a new point where the objective function is smaller than at the initial guess.*

In this lab, we implement two major descent algorithms: the method of steepest descent, which uses the gradient of the objective function as the descent direction, and the conjugate gradient algorithm, which uses a conjugate basis as the descent directions. We then apply these techniques to linear and logistic regression problems.

The Method of Steepest Descent

Recall the basic idea of a line search algorithm to optimize: at the kth iteration, choose a search direction $\mathbf{p}_k$ and a step size α_k, then compute $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$. If $f : \mathbb{R}^n \to \mathbb{R}$ is a scalar-valued function, its gradient $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ points in the direction of steepest increase of f. It follows that the opposite direction is the direction of steepest decrease, or *descent*.

The *method of steepest descent* is a line search method that uses $\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$ as the search direction. A simple version of the method chooses the step size as follows:

1. At the kth iteration, start with a step size of $\alpha = 1$.
2. Check that $f(\mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)) < f(\mathbf{x}_k)$ (i.e. check that the function decreases in the search direction), and if it does, set $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)$.
3. If the function does *not* decrease, set $\alpha = \frac{\alpha}{2}$ and check the search direction again. Repeat this step until the function decreases in the search direction.
4. If α is scaled below a given tolerance, terminate the iteration.

Problem 1. Write a function that accepts a convex objective function $f : \mathbb{R}^n \to \mathbb{R}$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$, an initial guess $x_0 \in \mathbb{R}^n$, a convergence tolerance, and a maximum number of iterations. Use the Method of Steepest Descent to compute the minimizer of f. Continue the algorithm until α is less than the convergence tolerance, or until iterating the maximum number of times. Return the minimizer.

The Conjugate-Gradient Method

Unfortunately, the method of steepest descent can be very inefficient for certain problems: depending on the geometry of the objective function, the sequence of points can "zig-zag" back and forth without making significant progress toward the true minimizer. In contrast, the *Conjugate-Gradient algorithm* ensures that the true global minimizer is reached in at most n steps. See Figure 18.1 for an illustration of this contrast.

There are many methods for solving the linear system of equations $A\mathbf{x} = \mathbf{b}$, each with its own pros and cons. The Conjugate-Gradient algorithm is one such method for solving large systems of equations where other methods, such as Cholesky factorization and simple Gaussian elimination, are unsuitable. However, the algorithm works equally well for optimizing convex quadratic functions, and it can even be extended to more general classes of optimization problems.

The type of linear system that Conjugate-Gradient can solve involves a matrix with special structure. Given a symmetric positive-definite $n \times n$ matrix Q and an n-vector b, we wish to find the n-vector x satisfying

$$Qx = b.$$

A unique solution exists because positive-definiteness implies invertibility. For our purposes here, it is useful to recast this problem as an equivalent optimization problem:

$$\min_{\mathbf{x}} f(\mathbf{x}) := \frac{1}{2}\mathbf{x}^{\mathsf{T}}Q\mathbf{x} - \mathbf{b}^{\mathsf{T}}\mathbf{x} + \mathbf{c}.$$

Note that $\nabla f(\mathbf{x}) = Q\mathbf{x} - \mathbf{b}$, so that minimizing f is the same as solving the equation

$$0 = \nabla f(\mathbf{x}) = Q\mathbf{x} - \mathbf{b},$$

which is the original linear system.

Conjugacy

Unlike the method of steepest descent, the conjugate gradient algorithm chooses a search direction based off of the matrix Q that is guaranteed to be a direction of descent, though not the direction of greatest descent.

Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are said to be *conjugate* with respect to Q if $\mathbf{x}^{\mathsf{T}}Q\mathbf{y} = 0$. A set of vectors $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m\}$ is said to be conjugate if each pair of vectors are conjugate to each other. Note that if $Q = I$, then conjugacy is the same as

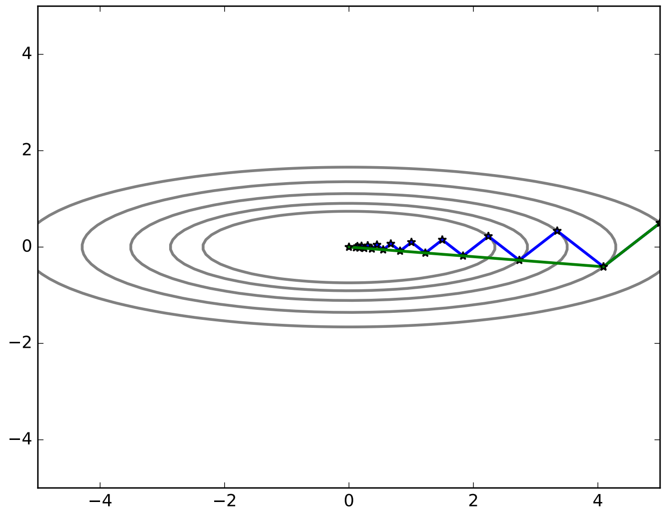

Figure 18.1: Paths traced by Steepest Descent (blue) and Conjugate-Gradient (green). Notice the zig-zagging nature of the Steepest Descent path, as opposed to the direct Conjugate-Gradient path, which finds the minimizer in 2 steps.

orthogonality. Thus, the notion of conjugacy is in some ways a generalization of orthogonality. It turns out that a conjugate set of vectors is linearly independent, and a conjugate basis–which can be constructed in a manner analogous to the Gram-Schmidt orthogonalization process–can be used to diagonalize the matrix Q. These are some of the theoretical reasons behind the effectiveness of the Conjugate-Gradient algorithm.

The Algorithm

If we are given a set of n Q-conjugate vectors, we can simply choose these as our direction vectors and follow the basic descent algorithm. Convergence to the minimizer in at most n steps is guaranteed because each iteration in the algorithm minimizes the objective function over an expanding affine subspace of dimension equal to the iteration number. Thus, at the n-th iteration, we have minimized the function over all of $\mathbb{R}^n$.

Unfortunately, we are not often given a set of conjugate vectors in advance, so how do we produce such a set? As mentioned earlier, a Gram-Schmidt process could be used, and the set of eigenvectors also works, but both of these options are computationally expensive. Built into the algorithm is a way to determine a new conjugate direction based only on the previous direction, which means less memory usage and faster computation. We have stated the details of Conjugate-Gradient in Algorithm 18.1.

Note that the points x_i are the successive approximations to the minimizer, the vectors d_i are the conjugate descent directions, and the vectors r_i, which actually correspond to the steepest descent directions, are used in determining the conjugate directions. The constants α_i and β_i are used, respectively, in the line search, and

Algorithm 18.1 Conjugate-Gradient Algorithm

```
procedure Conjugate-Gradient Algorithm
    Choose initial point $x_0$.
    $r_0 \leftarrow Qx_0 - b, d_0 \leftarrow -r_0, k \leftarrow 0$.
    while $r_k \neq 0$ do
        $\alpha_k \leftarrow \frac{r_k^T r_k}{d_k^T Q d_k}$.
        $x_{k+1} \leftarrow x_k + \alpha_k d_k$.
        $r_{k+1} \leftarrow r_k + \alpha_k Q d_k$.
        $\beta_{k+1} \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$.
        $d_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} d_k$.
        $k \leftarrow k + 1$.
```

in ensuring the Q-conjugacy of the descent directions.

The most numerically expensive computation in the algorithm is matrix-vector multiplication. Notice, however, that each iteration of the algorithm only requires one distinct matrix-vector multiplication, Qd_k. The rest of the operations are simply vector-vector multiplication, addition, and scalar multiplication. This makes for a very fast algorithm. As noted earlier, Conjugate-Gradient is especially preferred when Q is large and sparse. In this case, it may be possible to design a specialized sub-routine that performs matrix-vector multiplication by Q, by taking advantage of its sparseness. Doing so may lead to further speed-ups in the overall algorithm.

We now have an algorithm that can solve certain $n \times n$ linear systems and minimize quadratic functions on $\mathbb{R}^n$ in at most n steps, and sometimes fewer, depending on the spectrum of the matrix Q. Further improvements on convergence may be obtained by preconditioning the matrix, but we do not go into detail here.

Problem 2. Implement the basic Conjugate-Gradient algorithm presented above. Write a function `conjugateGradient()` that accepts a vector b, an initial guess x_0, a symmetric positive-definite matrix Q, and a default tolerance of .0001 as inputs. Continue the algorithm until $\|r_k\|$ is less than the tolerance. Return the solution x^* to the linear system $Qx = b$.

Example

We now work through an example that demonstrates the usage of the Conjugate-Gradient algorithm. We assume that we have already written the specified function in the above problem.

We must first generate a symmetric positive-definite matrix Q. This can be done by generating a random matrix A and setting $Q = A^T A$. So long as A is of full column rank, the matrix Q will be symmetric positive-definite.

```
>>> import numpy as np
>>> from scipy import linalg as la

>>> # initialize the desired dimension of the space
>>> n = 10

>>> # generate Q, b
>>> A = np.random.random((n,n))
>>> Q = A.T.dot(A)
>>> b = np.random.random(n)
```

At this point, check to make sure that Q is nonsingular by examining its determinant (use `scipy.linalg.det()`). Provided that the determinant is nonzero, we proceed by writing a function that performs matrix-vector multiplication by Q (we will not take advantage of sparseness just now), randomly selecting a starting point (Conjugate-Gradient is not sensitive to the location of the starting point), obtaining the answer using our function, and checking it with the answer obtained by `scipy.linalg.solve()`.

```
>>> # generate random starting point
>>> x0 = np.random.random(n)

>>> # find the solution
>>> x = conjugateGradient(b, x0, mult)

>>> # compare to the answer obtained by SciPy
>>> print np.allclose(x, la.solve(Q,b))
```

The output of the print statement should be `True`.

Time the performance of your algorithm and of `scipy.linalg.solve()` on inputs of size 100.

Application: Least Squares and Linear Regression

The Conjugate-Gradient method can be used to solve linear least squares problems, which are ubiquitous in applied science. Recall that a least squares problem can be formulated as an optimization problem:

$$\min_x \|Ax - b\|_2,$$

where A is an $m \times n$ matrix with full column rank, $x \in \mathbb{R}^n$, and $b \in \mathbb{R}^m$. The solution can be calculated analytically, and is given by

$$x^* = (A^T A)^{-1} A^T b,$$

or in other words, the minimizer solves the linear system

$$A^T A x = A^T b.$$

Since A has full column rank, we know that $A^T A$ is an $n \times n$ matrix of rank n, which means it is invertible. We can therefore conclude that $A^T A$ is symmetric positive-definite, so we may use Conjugate-Gradient to solve the linear system and obtain the least squares solution.

Linear least squares is the mathematical underpinning of linear regression, which is a very common technique in many scientific fields. In a typical linear regression problem, we have a set of real-valued data points $\{y_1, \ldots, y_m\}$, where each y_i is paired with a corresponding set of predictor variables $\{x_{i,1}, x_{i,2}, \ldots, x_{i,n}\}$ with $n < m$. The linear regression model posits that

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_n x_{i,n} + \epsilon_i$$

for $i = 1, 2, \ldots, m$. The real numbers $\beta_0, \ldots, \beta_n$ are known as the parameters of the model, and the ϵ_i are independent normally-distributed error terms. Our task is to calculate the parameters that best fit the data. This can be accomplished by posing the problem in terms of linear least squares: Define

$$b = [y_1, \ldots, y_m]^T,$$

$$A = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix},$$

and

$$x = [\beta_0, \beta_1, \ldots, \beta_n]^T.$$

Now use Conjugate-Gradient to solve the system

$$A^T A x = A^T b.$$

The solution $x^* = [\beta_0^*, \beta_1^*, \ldots, \beta_n^*]^T$ gives the parameters that best fit the data. These values can be understood as defining the hyperplane that best fits the data. See Figure 18.2.

Problem 3. Using your Conjugate-Gradient function, solve the linear regression problem specified by the data contained in the file `linregression.txt`. This is a whitespace-delimited text file formatted so that the i-th row consists of $y_i, x_{i,1}, \ldots, x_{i,n}$. Use the function `numpy.loadtxt()` to load in the data. Report your solution.

Non-linear Conjugate-Gradient Algorithms

The algorithm presented above is only valid for certain linear systems and quadratic functions, but the basic strategy may be adapted to minimize more general convex or non-linear functions. There are multiple ways to modify the algorithm, and they all involve getting rid of Q, since there is no such Q for non-quadratic functions. Generally speaking, we need to find new formulas for α_k, r_k, and β_k.

The scalar α_k is simply the result of performing a line-search in the given direction d_k, so we may define

$$\alpha_k = \arg\min_x f(x_k + \alpha d_k).$$

The vector r_k in the original algorithm was really just the gradient of the objective function, and so we may define

$$r_k = \nabla f(x_k).$$

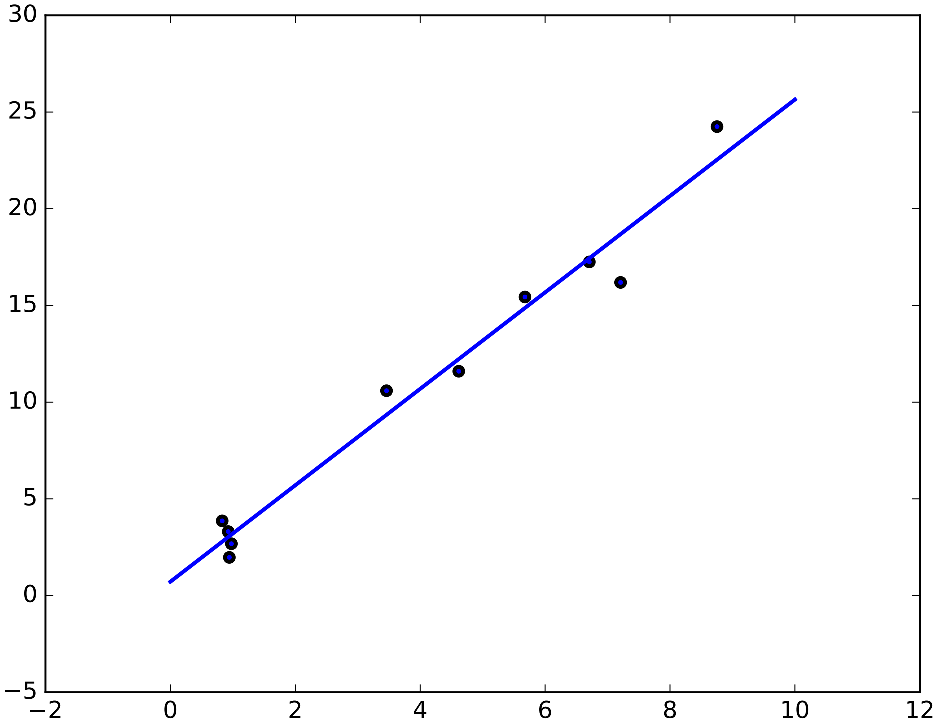

Figure 18.2: Solving the Linear Regression problem results in a best-fit hyperplane.

There are various ways to define the constants β_k in this more general setting, and the right choice will depend on the nature of the objective function. A well-known formula, due to Fletcher and Reeves, is

$$\beta_{k+1} = \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}.$$

Making these adjustments is not difficult, but we will opt instead to use built-in functions in Python. In particular, the SciPy module `scipy.optimize` provides a function `fmin_cg()`, which uses a non-linear Conjugate-Gradient method to minimize general functions. Using this function is easy – we only need to pass to it the objective function and an initial guess.

Application: Logistic Regression

Logistic regression is an important technique in statistical analysis and classification. The core problem in logistic regression involves an optimization that we can tackle using nonlinear Conjugate-Gradient.

As in linear regression, we have a set of data points y_i together with predictor variables $x_{i,1}, x_{i,2}, \ldots, x_{i,n}$ for $i = 1, \ldots, m$. However, the y_i are binary data points – that is, they are either 0 or 1. Furthermore, instead of having a linear relationship between the data points and the response variables, we assume the following probabilistic relationship:

$$\mathbb{P}(y_i = 1 \,|\, x_{i,1}, \dots, x_{i,n}) = p_i,$$

where

$$p_i = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}))}.$$

The parameters of the model are the real numbers $\beta_0, \beta_1, \dots, \beta_n$. Observe that we have $p_i \in (0, 1)$ regardless of the values of the predictor variables and parameters.

The probability of observing the data points y_i under this model, assuming they are independent, is given by the expression

$$\prod_{i=1}^{m} p_i^{y_i}(1 - p_i)^{1-y_i}.$$

We seek to choose the parameters $\beta_0, \dots, \beta_n$ that maximize this probability. To this end, define the *likelihood function* $L : \mathbb{R}^{n+1} \to \mathbb{R}$ by

$$L(\beta_0, \dots, \beta_n) = \prod_{i=1}^{m} p_i^{y_i}(1 - p_i)^{1-y_i}.$$

We can now state our core problem as follows:

$$\max_{(\beta_0, \dots, \beta_n)} L(\beta_0, \dots, \beta_n).$$

Maximizing this function can be problematic for numerical reasons. By taking the logarithm of the likelihood, we have a more suitable objective function whose maximizer agrees with that of the original likelihood function, since the logarithm is strictly monotone increasing. Thus, we define the *log-likelihood function* $l : \mathbb{R}^{n+1} \to \mathbb{R}$ by $l = \log \circ L$.

Finally, we multiply by -1 to turn our problem into minimization. The final statement of the problem is:

$$\min_{(\beta_0, \dots, \beta_n)} -l(\beta_0, \dots, \beta_n).$$

A few lines of calculation reveal that

$$\begin{aligned} l(\beta_0, \dots, \beta_n) = -&\sum_{i=1}^{m} \log(1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}))) + \\ &\sum_{i=1}^{m} y_i(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}). \end{aligned}$$

The values for the parameters that we obtain are known collectively as the *maximum likelihood estimate.*

Let's work through a simple example. We will deal with just one predictor variable, and therefore two parameters. The data is given in Table 18.1. This is obviously just toy data with no meaning, but one can think of the y_i data points as indicating, for example, the presence of absence of a particular disease in subject i, with x_i being the subject's weight, or age, or something of the sort.

In the code below we initialize our data.

Table 18.1: Data for Logistic Regression Example

y	x
0	1
0	2
0	3
0	4
1	5
0	6
1	7
0	8
1	9
1	10

```
>>> y = np.array([0, 0, 0, 0, 1, 0, 1, 0, 1, 1])
>>> x = np.ones((10, 2))
>>> x[:,1] = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Although we have just one predictor variable, we initialized `x` with two columns, the first of which consists entirely of ones, and the second of which contains the values of the predictor variable. This extra column of ones corresponds to the parameter β_0, which, as you will note, is not multiplied by any of the predictor variables in the log-likelihood function.

We next need to write a Python function that returns the value of our objective function for any value of the parameters, (β_0, β_1).

```
>>> def objective(b):
...     #Return -1*l(b[0], b[1]), where l is the log likelihood.
...    return (np.log(1+np.exp(x.dot(b))) - y*(x.dot(b))).sum()
```

Finally, we minimize the objective function using `fmin_cg()`.

```
>>> guess = np.array([1., 1.])
>>> b = fmin_cg(objective, guess)
Optimization terminated successfully.
         Current function value: 4.310122
         Iterations: 13
         Function evaluations: 128
         Gradient evaluations: 32
>>> print b
[-4.35776886  0.66220658]
```

We can visualize our answer by plotting the data together with the function

$$\phi(x) = \frac{1}{1 + \exp(-\beta_0 - \beta_1 x)},$$

using the values β_0, β_1 that we obtained from the minimization.

```
>>> dom = np.linspace(0, 11, 100)
>>> plt.plot(x, y, 'o')
>>> plt.plot(dom, 1./(1+np.exp(-b[0]-b[1]*dom)))
>>> plt.show()
```

Using this procedure, we obtain the plot in Figure 18.3. Note that the graph of ϕ, known as a *sigmoidal curve*, gives the probability of y taking the value 1 at a particular value of x. Observe that as x increases, this probability approaches 1. This is reflected in the data.

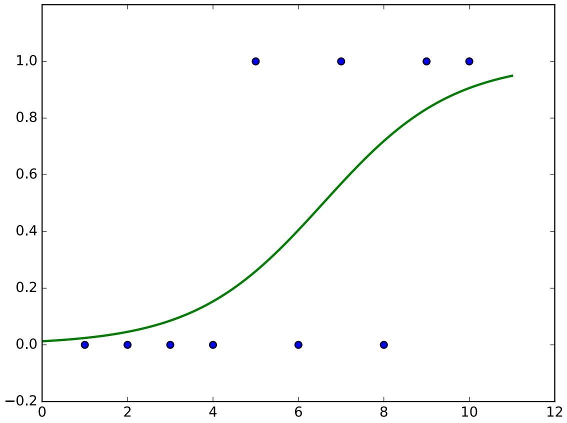

Figure 18.3: Data from the logistic regression example together with the calculated sigmoidal curve.

Problem 4. Following the example given above, find the maximum likelihood estimate of the parameters for the logistic regression data in the file `logregression.txt`. This is a whitespace-delimited text file formatted so that the i-th row consists of $y_i, x_{i,1}, x_{i,2}, x_{i,3}$. Since there are three predictor variables, there are four parameters in the model. Report the calculated values.

You should be able to use much of the code above unchanged. In particular, the function `objective()` does not need any changes. You simply need to set your variables `y` and `x` appropriately, and choose a new initial guess (an array of length four). Note that `x` should be an $m \times 4$ array whose first column consists entirely of ones, whose second column contains the values in the second column of the data file, and so forth.

Logistic regression can become a bit more tricky when some of the predictor variables take on binary or categorical values. In such situations, the data requires a bit of pre-processing before running the minimization.

The values of the parameters that we obtain can be useful in analyzing relationships between the predictor variables and the y_i data points. They can also be used to classify or predict values of new data points.