Optimization, Programming Assignment #4

May 7, 2008

Description

In this assignment, you will implement the conjugate gradient method, and BFGS.

Important note for users of python: I (Andy) had a lot of trouble with the sparse matrix routines in scipy. I strongly recommend that you not do problem 1 in python. For this reason, if (and only if) you have been turning in your assignments in python, you have several options on this problem set:

1. Do only problem 2 (it will count for twice as much). If you decide to do this, you should email me, and I will send you python equivalents of the provided matlab code. You will also need to set up a time to meet with me, so that we can go over the results of the first problem together.
2. Do the entire assignment in python. As with option 1, I'll send you python equivalents of the provided matlab code upon request (including code to read the data files of problem 1 into scipy sparse matrices). It may be that my problems with the sparse matrices were entirely my fault, or were a result of the particular versions of scipy which I worked with (I tried those in both the Ubuntu Gutsy and Hardy repositories). There may also be other libraries available for dealing with sparse matrices, which of course you are free to use.
3. Do the entire assignment in matlab. I'm likely to be more forgiving when grading, since this is not only the longest programming assignment so far, but you'll also be doing it in a new language.

You should turn in an archive containing all of your source code, plots, and a document containing a brief description of your code (how to run it, what parameters it takes, etc), and answers to all questions. Email this archive to your TA, at cotter@tti-c.org.

Please remember to turn in a document, and to answer the questions (in particular, talk about what your plots mean)!

Most of this problem set was based on material found in:

- Jorge Nocedal, Stephen J. Wright. "Numerical Optimization". Springer. 1999

1 Quadratic objective

1.1 Conjugate gradient

When the objective function is a quadratic of the form $f(w) = \frac{1}{2} w^T Q w - w^T b$, the conjugate gradient method may be written as follows, given an initial point $w^{(0)}$ and a stopping threshold ϵ (algorithm 5.2 of Nocedal and Wright):

1. $r^{(0)} = Qw^{(0)} - b$
2. $\Delta w^{(0)} = -r^{(0)}$
3. $k = 0$
4. Loop:

5. $\alpha^{(k)} = \frac{\left(r^{(k)}\right)^T r^{(k)}}{\left(\Delta w^{(k)}\right)^T Q \Delta w^{(k)}}$
6. $w^{(k+1)} = w^{(k)} + \alpha^{(k)} \Delta w^{(k)}$
7. $r^{(k+1)} = r^{(k)} + \alpha^{(k)} Q \Delta w^{(k)}$
8. If $\left\| r^{(k+1)} \right\|_2 \leq \epsilon$ then terminate
9. $\beta^{(k+1)} = \frac{\left(r^{(k+1)}\right)^T r^{(k+1)}}{\left(r^{(k)}\right)^T r^{(k)}}$
10. $\Delta w^{(k+1)} = -r^{(k+1)} + \beta^{(k+1)} \Delta w^{(k)}$
11. $k = k + 1$
12. Return $w^{(k)}$

Above, the residuals $r^{(k)}$ are nothing but the gradients $Qw^{(k)} - b$ of our objective function: the above formulation of the algorithm just calculates these quantities insignificantly more efficiently.

We will use the above algorithm to solve a sparse least-squares problem. Given a set of n vectors $x_i \in \mathbb{R}^m$, $y_i \in \mathbb{R}$, we will attempt to find a vector of weights $w \in \mathbb{R}^m$ such that w has small norm, and $x_i^T w \approx y_i$. In particular, we will let $X \in \mathbb{R}^{n \times m}$ be the matrix for which its ith row is x_i, and Y the (column) vector of y_is, and will assume that X is *sparse*. We will attempt to minimize the following objective:

$$f(w) \quad = \quad \|Xw - y\|_2^2 + \lambda \|w\|_2^2$$

Least-squares with a l^2 regularization term ($\lambda \|w\|_2^2$), as we have here, is often known as ridge regression. Writing out the norms as inner products:

$$f(w) \quad = \quad w^T \left(X^T X + \lambda I\right) w - 2w^T X^T y + y^T y$$

Removing the $y^T y$ term, since it is a constant, we may take our objective to be:

$$f(w) \quad = \quad \frac{1}{2} w^T Q w - w^T b$$

Where $Q = 2\left(X^T X + \lambda I\right)$ and $b = 2X^T y$. The minimizer of this objective function is the solution w of the linear system:

$$Qw \quad = \quad b$$

Since $X^T X \succeq 0$ for any X, we will have that $Q = 2\left(X^T X + \lambda I\right) \succ 0$ for $\lambda > 0$, showing that this problem is guaranteed to have a unique solution, for positive λ.

1.1.1 Implementation

Implement the above conjugate gradient method for minimizing the above objective function. Make sure to make use of the sparsity of X! Calculating $\left(X^T X\right) v$ requires m^2 operations even if $X^T X$ has been precalculated (which requires about $\sum_{i=1}^n k_i^2$ operations, where k_i is the number of nonzero elements in the ith row of X), while calculating $X^T (Xv)$ requires $2k$ operations, where k is the number of nonzero entries in X. For the provided dataset, $m^2 = 10^6$ and $k \approx 4 \times 10^5$, so the benefits of sparsity for this problem would be slight, if calculating $X^T X$ weren't so expensive.

Your function should take five parameters: X (the sparse matrix of "input" vectors); Y (the vector of "outputs"); λ (the regularization parameter); $w^{(0)}$ (the initial weight vector); ϵ (the stopping threshold). It should return a list of pairs (w, t), with one entry per iteration, containing weight vectors w and times t. The times should be calculated

by starting a timer at the start of the function (using "tic" in matlab, or storing the result of "time.clock()" in a variable t_0 in python), and then calculating the elapsed time whenever a new set of weights is calculated (using "toc" in matlab, or "time.clock() $- t_0$" in python).

As in the previous assignments, code defensively, and comment:

1. On entry, check that the matrix dimensions are compatible, $\lambda > 0$, and $\epsilon > 0$, raising an error if any of these conditions is not satisfied
2. Check that you do not enter an infinite loop, by raising an error if some ridiculously large number of iterations are performed (recall that, for a quadratic objective, the conjugate gradient method should converge in no more than m iterations, where m is the dimension of the problem)
3. Use sparsity as much as possible! Never calculate $X^T X$ explicitly
4. Comment any portion of your code of which the interpretation is not obvious

1.1.2 Experiments

In this problem, you'll be comparing the amount of time required to solve this problem using the conjugate gradient method to the time required using a matrix factorization. To smooth out the effects of background processes, or imprecision in the timing, you should average all of your timings over multiple (let's say 20) runs.

Code has been provided in "problem1.m" for loading the required datasets as sparse matrices (not that its very complicated). The dataset is contained in two files, "x_sparse_matrix.csv" and "y_vector.csv". All of these files are included in the tarball on the course web page.

First, find p^* by solving $Qw = b$ using the Cholesky factorization ("chol" in matlab). Time how long it takes to optimize the objective by this method, *including* the time spent calculating Q and b. Then, time how long it takes to solve the system, *not including* the time spent calculating Q and b.

Run your implementation of the conjugate gradient method on this problem, with $w^{(0)} = \vec{0}$, $\lambda = 1$ and $\epsilon = 10^{-8}$. Plot the objective function value (not log-error this time) versus runtime. Compare the performance of your conjugate gradient implementation to your language's built-in factorization routines, for this particular (very sparse) problem.

1.2 Preconditioned conjugate gradient (optional)

In addition to being sparse and high-dimensional, the dataset for this problem is somewhat ill-conditioned. Recall that, when we were working on gradient descent, we found that we could solve an ill-conditioned problem more efficiently using steepest descent with a suitable quadratic norm. Further recall that steepest descent under a quadratic norm corresponds exactly to performing a linear change in coordinates.

In the context of solving linear systems, working with a transformed version of a problem is known as *preconditioning* (`http://en.wikipedia.org/wiki/Preconditioning`). For the current problem, minimizing our objective function is equivalent to solving the following for w:

$$Qw \quad = \quad b$$

Where $Q = 2\left(X^T X + \lambda I\right)$ and $b = 2X^T y$. We may instead solve an equivalent transformed version of this problem for w', with $w' = Pw$:

$$QP^{-1}w' \quad = \quad b$$

Where P, the preconditioner, is chosen such that (aside from obviously being nonsingular):

1. P is easy to calculate
2. It is cheap to solve the linear system $Px = c$ for x

As was the case with steepest descent, the "optimal" choice for P is such that $P^T P \approx Q$ (which is the Hessian)–but then calculating P^{-1} would reduce to optimizing our original objective function, which would defeat the purpose (this is analogous to the way in which, if we were using steepest descent with the Hessian norm, it would converge in a single iteration, if the objective were quadratic).

Instead, we will choose P such that it is diagonal, with diagonal elements $P_{i,i} = \sqrt{Q_{i,i}}$ (this is called the Jacobi preconditioner). We may calculate the diagonal elements of $X^T X$ using only k operations (where k is the number of nonzero elements in the matrix X), so the first requirement is met. Furthermore, since P is diagonal, P^{-1} is diagonal, so finding $x = P^{-1}c$ extremely cheap, computationally.

If we plug $w' = Pw$ into our objective function, and throw out the constant $y^T y$ term:

$$\begin{aligned} f(w') &= (w')^T \left(P^{-T} X^T X P^{-1} + \lambda P^{-T} P\right) w' - 2 (w')^T P^{-T} X^T y \\ &= \frac{1}{2} (w')^T Q' w' - (w')^T b' \end{aligned}$$

Where $Q' = 2\left(P^{-\frac{T}{2}} X^T X P^{-\frac{1}{2}} + \lambda P^{-\frac{T}{2}} P^{-\frac{1}{2}}\right)$ and $b' = 2P^{-\frac{T}{2}} X^T y$. We may now use the conjugate gradient method to find a w' which minimizes the above objective, and the calculate $w = P^{-1} w'$ to find the weights in the original coordinate system.

Note that the effect of the preconditioner P is to scale the rows and columns of Q in such a way that its diagonal entries are all 1–in a sense, $P^{-T} P^{-1}$ is a very rough approximation of the inverse Hessian (and $P^T P$ is a very rough approximation of the Hessian).

1.2.1 Implementation

Modify your implementation of the conjugate gradient method to take the inverse preconditioner P^{-1} as a parameter, and modify your code so that it minimizes the preconditioned objective function:

$$f(w') = \frac{1}{2} (w')^T Q' w' - (w')^T b'$$

Where $Q' = 2\left(P^{-\frac{T}{2}} X^T X P^{-\frac{1}{2}} + \lambda P^{-\frac{T}{2}} P^{-\frac{1}{2}}\right)$ and $b' = 2P^{-\frac{T}{2}} X^T y$. Also add a check that the dimensions of P^{-1} are compatible with X and Y. All of the requirements listed in problem 1.1.1 continue to apply (including, most importantly, that you never explicitly calculate $X^T X$). Make sure that your function returns a list of pairs $(w, t) = \left(P^{-1} w', t\right)$, *not* the list of pairs (w', t).

1.2.2 Experiments

Calculate the condition number (the ratio of the largest to smallest eigenvalue) of both Q and $P^{-T} Q P^{-1}$ (use "cond" in matlab). Did the preconditioning work?

Next, run the preconditioned conjugate gradient method on this problem, again , with $w_0 = \vec{0}$, $\lambda = 1$ and $\epsilon = 10^{-8}$. Plot the objective function value versus runtime on top of the plot from problem 1.1.2. Compare the performance of your preconditioned conjugate gradient implementation to both non-preconditioned conjugate gradient, and your language's built-in factorization routines.

As before, you should average your timings over multiple (20) runs.

1.3 BFGS

When the objective function is a quadratic of the form $f(w) = \frac{1}{2} w^T Q w + w^T b$, the BFGS method (with exact line search) may be written as follows, given an initial point $w^{(0)}$, an initial estimate of the inverse Hessian $H^{(0)}$, and a stopping threshold ϵ (algorithm 8.1 of Nocedal and Wright):

1. $k = 0$

2. $\nabla f^{(0)} = Qw^{(0)} - b$
3. Loop
4. $\quad \Delta w^{(k)} = -H^{(k)} \nabla f^{(k)}$
5. $\quad \alpha^{(k)} = -\frac{\left(\Delta w^{(k)}\right)^T \nabla f^{(k)}}{\left(\Delta w^{(k)}\right)^T Q \Delta w^{(k)}}$
6. $\quad w^{(k+1)} = w^{(k)} + \alpha^{(k)} \Delta w^{(k)}$
7. $\quad \nabla f^{(k+1)} = Qw^{(k+1)} - b$
8. $\quad$ If $\left\|\nabla f^{(k+1)}\right\|_2 \leq \epsilon$ then terminate
9. $\quad s^{(k)} = w^{(k+1)} - w^{(k)}$
10. $\quad y^{(k)} = \nabla f^{(k+1)} - \nabla f^{(k)}$
11. $\quad \rho^{(k)} = \frac{1}{\left(y^{(k)}\right)^T s^{(k)}}$
12. $\quad H^{(k+1)} = \left(I - \rho^{(k)} s^{(k)} \left(y^{(k)}\right)^T\right) H^{(k)} \left(I - \rho^{(k)} y^{(k)} \left(s^{(k)}\right)^T\right) + \rho^{(k)} s^{(k)} \left(s^{(k)}\right)^T$
13. $\quad k = k + 1$
14. Return $w^{(k)}$

1.3.1 Implementation

Implement the above BFGS algorithm for minimizing the above objective function. Make sure to make use of the sparsity of X! Also note that, while the update on line 12 is expressed in terms of two matrix-matrix products, it may be implemented more efficiently by multiplying out the entire expression, and parenthesizing in such a way that there are only matrix-vector multiplies.

Your function should take five parameters: X (the matrix of "input" vectors); Y (the vector of "outputs"); λ (the regularization parameter); $w^{(0)}$ (the initial weight vector); $H^{(0)}$ (the initial approximate inverse Hessian); ϵ (the stopping threshold). It should return a list of weight vectors w, one per iteration.

As before, code defensively, and comment:

1. On entry, check that the matrix dimensions are compatible, $\lambda > 0$, and $\epsilon > 0$, raising an error if any of these conditions is not satisfied
2. Check that you do not enter an infinite loop, by raising an error if some ridiculously large number of iterations are performed
3. Use sparsity as much as possible! Never calculate $X^T X$ explicitly
4. Try to avoid matrix-matrix multiplies, where it is possible to use matrix-vector multiplies
5. Comment any portion of your code of which the interpretation is not obvious

1.3.2 Experiments

Run your implementation of the BFGS method on the ridge regression problem, with $w^{(0)} = \vec{0}$, $H^{(0)} = I$, $\lambda = 1$ and $\epsilon = 10^{-8}$. Compare the sequence of iterates $w^{(k)}$ to the sequence which you found using conjugate gradient (they should be identical).

1.3.3 Preconditioning experiments (optional)

Run your implementation of the BFGS method on the ridge regression problem, with $w^{(0)} = \vec{0}$, $H^{(0)} = P^{-T} P^{-1}$, $\lambda = 1$ and $\epsilon = 10^{-8}$, where P is the preconditioner you used in problem 1.2. Compare the sequence of iterates $w^{(k)}$ to the sequence which you found using preconditioned conjugate gradient (they should be identical).

2 Non-quadratic objective

2.1 Line search

Code has been provided in "line_search.m" for performing a line search for which the resulting step length α satisfies the strong Wolfe conditions (algorithm 3.2 of Nocedal and Wright). The function should be called as line_search $(\phi, \phi', c_1, c_2, \beta)$, where $\phi(\alpha) = f(w + \alpha\Delta w)$, $\phi'(\alpha) = \frac{d}{d\alpha} f(w + \alpha\Delta w)$, c_1 and c_2 are parameters to the line search (explained below), and β is a parameter which determines how quickly the αs checked by the line search increase (it may be safely left at its default value of 2).

The strong Wolfe conditions are (equations 3.7ab of Nocedal and Wright):

$$\begin{aligned} f(w + \alpha\Delta w) &\leq f(w) + c_1\alpha \left(\nabla f(w)\right)^T \Delta w \\ \left|\left(\nabla f(w + \alpha\Delta w)\right)^T \Delta w\right| &\leq c_2 \left|\left(\nabla f(w)\right)^T \Delta w\right| \end{aligned}$$

Where $0 < c_1 < c_2 < 1$ in general, though if the line search is being used for conjugate gradient, we should have $0 < c_1 < c_2 < \frac{1}{2}$. The first of these conditions should look familiar: it's the stopping condition which we've been using for backtracking line search. In the second condition, $\left(\nabla f(w + \alpha\Delta w)\right)^T \Delta w$ is nothing but the derivative with respect to α' of $f(w + \alpha'\Delta w)$, evaluated at the point α, while $\left(\nabla f(w)\right)^T \Delta w$ is the same derivative, evaluated at 0. Hence, this condition requires that the derivative of the function along the line $w + \alpha'\Delta w$ be small for our step length α, which may be interpreted as requiring that α be close to a local minimum of the function along the line.

2.1.1 Conjugate gradient implementation

We may perform conjugate gradient on a non-quadratic function as follows, given an initial point $w^{(0)}$, line search parameters c_1, c_2 and β, and a stopping threshold ϵ (algorithm 5.4 of Nocedal and Wright):

1. $\Delta w^{(0)} = -\nabla f\left(w^{(0)}\right)$
2. $k = 0$
3. Loop
4. $\quad \alpha^{(k)} = \text{line\_search}\left(\lambda\alpha.f\left(w^{(k)} + \alpha\Delta w^{(k)}\right), \lambda\alpha.\left(\Delta w^{(k)}\right)^T \nabla f\left(w^{(k)} + \alpha\Delta w^{(k)}\right), c_1, c_2, \beta\right)$
5. $\quad w^{(k+1)} = w^{(k)} + \alpha^{(k)}\Delta w^{(k)}$
6. $\quad$ If $\left\|\nabla f\left(w^{(k+1)}\right)\right\|_2 \leq \epsilon$ then terminate
7. $\quad$ If $k + 1 \equiv 0 \bmod m$ then
8. $\quad\quad \beta^{(k+1)} = 0$
9. $\quad$ Else
10. $\quad\quad \beta^{(k+1)} = \frac{\left(\nabla f\left(w^{(k+1)}\right)\right)^T \nabla f\left(w^{(k+1)}\right)}{\left(\nabla f\left(w^{(k)}\right)\right)^T \nabla f\left(w^{(k)}\right)}$
11. $\quad \Delta w^{(k+1)} = -\nabla f\left(w^{(k+1)}\right) + \beta^{(k+1)}\Delta w^{(k)}$
12. $\quad k = k + 1$

There are in fact several different ways in which the conjugate gradient method may be modified to work on a non-quadratic function, all of which are equivalent for a quadratic objective. The above is called the Fletcher-Reeves method. The method which was covered in class is called the Polak-Ribière method, and differs from Fletcher-Reeves in the calculation of β, where $\beta_{PR}^{(k+1)} = \frac{\left(\nabla f\left(w^{(k+1)}\right)\right)^T \left(\nabla f\left(w^{(k+1)}\right) - \nabla f\left(w^{(k)}\right)\right)}{\left(\nabla f\left(w^{(k)}\right)\right)^T \nabla f\left(w^{(k)}\right)}$.

Pay special attention to lines 7-10. These lines "restart" the conjugate gradient algorithm every m steps (where m is the dimension of the problem), since by setting $\beta = 0$, the search direction p will be set to the negative gradient on line 11, effectively causing the algorithm to start over from the current location. Quoting Nocedal and Wright, page 123:

> "Restarting serves to periodically refresh the algorithm, erasing old information that may not be beneficial. We can even prove a strong theoretical result about restarting: It leads to n-step quadratic convergence, that is, $\left\|w^{(k+m)} - w^*\right\| = O\left(\left\|w^{(k)} - w^*\right\|^2\right)$. With a little thought, we can see that this result is not so surprising. Consider a function f that is strongly convex quadratic in a neighborhood of the solution, but is nonquadratic everywhere else. Assuming that the algorithm is converging to the solution in question, the iterates will eventually enter the quadratic region, and from that point onward, its behavior will simply be that of the linear conjugate gradient method, Algorithm 5.2. In particular, finite termination will occur within n steps of the restart. The restart is important, because the finite-termination property (and other appealing properties of) Algorithm 5.2 holds only when its initial search direction $p^{(0)}$ is equal to the negative gradient."

The Polak-Ribière method has a somewhat more natural "auto-restart" strategy, in which lines 7-10 would be replaced by $\beta^{(k+1)} = \max\left(0, \beta_{PR}^{(k+1)}\right)$. For the Fletcher-Reeves method, using a line search which guarantees the strong Wolfe conditions (as we are doing) results in every Δw being a descent direction. For Polak-Ribière, this is not the case. The restart strategy for this algorithm, therefore, is to restart whenever a non-descent direction is encountered (which is precisely what the max with zero accomplishes). The conventional wisdom seems to be that Polak-Ribière is generally superior to Fletcher-Reeves.

Implement this algorithm. Your function should take five parameters: f (the objective function); ∇f (the gradient); c_1 and c_2 (the line search parameters–β will be left at its default value of 2); $w^{(0)}$ (the initial weight vector); ϵ (the stopping threshold). It should return a list of pairs (w, t), with one entry per iteration, containing weight vectors w and times t.

2.1.2 BFGS implementation

The BFGS algorithm on a non-quadratic objective is essentially the same as that which you already implemented. Given an initial point $w^{(0)}$, initial inverse Hessian approximation $H^{(0)}$, line search parameters c_1, c_2 and β, and a stopping threshold ϵ:

1. $k = 0$
2. Loop
3. $\quad \Delta w^{(k)} = -H^{(k)} \nabla f\left(w^{(k)}\right)$
4. $\quad \alpha^{(k)} = \text{line\_search}\left(\lambda\alpha.f\left(w^{(k)} + \alpha\Delta w^{(k)}\right), \lambda\alpha.\left(\Delta w^{(k)}\right)^T \nabla f\left(w^{(k)} + \alpha\Delta w^{(k)}\right), c_1, c_2, \beta\right)$
5. $\quad w^{(k+1)} = w^{(k)} + \alpha^{(k)}\Delta w^{(k)}$
6. $\quad$ If $\left\|\nabla f\left(w^{(k+1)}\right)\right\|_2 \leq \epsilon$ then terminate
7. $\quad s^{(k)} = w^{(k+1)} - w^{(k)}$
8. $\quad y^{(k)} = \nabla f\left(w^{(k+1)}\right) - \nabla f\left(w^{(k)}\right)$
9. $\quad \rho^{(k)} = \frac{1}{\left(y^{(k)}\right)^T s^{(k)}}$
10. $\quad H^{(k+1)} = \left(I - \rho^{(k)} s^{(k)} \left(y^{(k)}\right)^T\right) H^{(k)} \left(I - \rho^{(k)} y^{(k)} \left(s^{(k)}\right)^T\right) + \rho^{(k)} s^{(k)} \left(s^{(k)}\right)^T$
11. $\quad k = k + 1$
12. Return $w^{(k)}$

Implement this algorithm. Your function should take five parameters: f (the objective function); ∇f (the gradient); c_1 and c_2 (the line search parameters–β will be left at its default value of 2); $w^{(0)}$ (the initial weight vector); $H^{(0)}$ (the initial approximate inverse Hessian); ϵ (the stopping threshold). It should return a list of pairs (w, t), with one entry per iteration, containing weight vectors w and times t.

2.2 Poisson regression

Suppose that we have an iid set of data points $x_i \in \mathbb{R}^m$ and labels $y_i \in \mathbb{N}$ both drawn from some underlying distribution, where:

$$\begin{aligned} P(y \mid x, w) &= \text{Poisson}\left(e^{w^T x}\right) \\ &= \frac{e^{-e^{w^T x}} e^{y w^T x}}{y!} \end{aligned}$$

If we imagine that, say, y_i is the number of times book i was checked out of a library over the course of a year, and x_i is a set of features representing, say, the title, author, genre, release date, publisher (and so on) of the book, then what is being learned by this model–the quantity $e^{w^T x}$–may be interpreted as a linear measure of a book's popularity. Poisson regression is well suited to such problems because of the *law of rare events*. Quoting from Wikipedia (`http://en.wikipedia.org/wiki/Poisson_distribution`):

> "The binomial distribution with parameters n and $\frac{\lambda}{n}$, i.e., the probability distribution of the number of successes in n trials, with probability $\frac{\lambda}{n}$ of success on each trial, approaches the Poisson distribution with expected value λ as n approaches infinity. This limit is sometimes known as the *law of rare events*, although this name may be misleading because the events in a Poisson process need not be rare (the number of telephone calls to a busy switchboard in one hour follows a Poisson distribution, but these events would not be considered rare)"

To put the above in the context of our example: if one wishes to represent the number of times a book will be checked out over the course of a year by a probability distribution, then one could (somewhat unrealistically) discretize time by dividing the entire year up into, say, one-minute chunks, and assume that there is some constant probability that the book will be checked out each minute. The resulting probability distribution over the number of check-outs would then be binomial. Alternatively, we could break the year up into 30-second chunks (thereby doubling the total number of time intervals), in which case we might expect that the probability of the book being checked out over one 30-second chunk would be half that of a one-minute time interval. If we were to repeat this process of doubling our number of chunks and halving our probabilities, then in the limit as the time interval length went to zero (time became continuous), the probability distribution over the number of check-outs would approach a Poisson distribution.

Anyway, ro return to our model. We will minimize the negative conditional log-likelihood of the labels y given the feature vectors x and weights w. Using independence:

$$\begin{aligned} f(w) &= -\log\left(\prod_{i=1}^{n} P(y_i \mid x_i, w)\right) \\ &= \sum_{i=1}^{n}\left(e^{w^T x_i} - y_i w^T x_i + \log(y_i!)\right) \end{aligned}$$

Since $\log(y_i!)$ does not depend on w, we may instead minimize:

$$f(w) = \sum_{i=1}^{n}\left(e^{w^T x_i} - y_i w^T x_i\right)$$

Differentiating the objective function yields that:

$$\begin{aligned} \nabla f(w) &= \sum_{i=1}^{n}\left(e^{w^T x_i} - y_i\right) x_i \\ \nabla^2 f(w) &= \sum_{i=1}^{n} e^{w^T x_i} x_i x_i^T \end{aligned}$$

The expression for the Hessian shows that the objective is convex, since $x_i x_i^T$ is always positive semi-definite, and $e^{w^T x_i}$ must be nonnegative.

2.2.1 Experiments

The file "newton.m" contains an implementation of Newton's method using the provided line search routine.

The dataset may be found in "data.csv", which contains $n = 2500$ (y_i, x_i) pairs, with $y_i \in \mathbb{N}$ and $x_i \in \mathbb{R}^m_+$, where $m = 25$. Using this as the data, perform Poisson regression to find an optimal set of weights w, using Newton's method, conjugate gradient, and BFGS, all using the provided line search. For all three algorithms, you should use the initial point $w^{(0)} = \vec{1}$. For BFGS, let $H^{(0)} = I$ be the initial estimate of the inverse Hessian. The line search parameters should be $c_1 = 10^{-4}$, $c_2 = 0.9$ and $\beta = 2$ (the default) for Newton's method and BFGS, and $c_1 = 10^{-4}$, $c_2 = 0.45$ and $\beta = 2$ for conjugate gradient.

Plot the objective function value versus time, for all three of these algorithms (on the same plot). As before, you should average your timings over multiple (20) runs. What properties of this problem do you think account for the differences in performance?