Aaroh Gokhale, Phillip Wangnick

14 May 2021

Gauss Newton Algorithm

In both Multivariable Calculus and Linear Algebra this year, we've found ways to find the line of best fit for a given data set using least squares. The least squares regression line, while a powerful tool, is not the most optimal tool for modelling data in real life. Many things are not correlated in a linear manner, and might be modelled better with higher order polynomials, exponentials, sinusoidal functions, or some other class of functions.

The Gauss Newton Algorithm is a powerful iterative tool that allows us to get better and better approximations for nonlinear models of data. It modifies Newton's method for optimization in order to achieve this. It first appeared in a paper from Carl Friedrich Gauss called Theoria motus corporum coelestium in sectionibus conicis solem ambientum.

Newton's Method

We will first begin by setting up the prerequiste to the Gauss Newton Algorithm, which is Newton's method. Netwon's method allows us to find the zeroes of a function using its derivatives.

The formula for the next closer approximation for the zero of function is a recursive formula as follows:

For a given function $f : \mathbb{R} \to \mathbb{R}$, let x_n be the sequence of numbers that gets closer to the value of the root of f. Then, $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. This result is obtained through right triangle trignometry by drawing the tangent line of f at x_n; we won't go into the derivation in this paper.

Let's take a look at an example of Newton's method, by trying to approximate the value of $\sqrt{2}$ using the quadratic function $g(x) = x^2 - 2$, the root of which is $\sqrt{2}$.

Let's start with a guess for the zero of $x^2 - 2$. Since for $x = 1$, $g(x) = -1$, and for $x = 2$, $g(x) = 3$, g has to cross the x axis between those two values, so let's pick $x_0 = 2$ for our initial guess. $g'(x) = 2x$

Then:
$x_1 = 2 - \frac{2}{4} = 1.5$
$x_2 = 1.5 - \frac{0.25}{3} = 1.41\overline{6} \approx 1.417$
$x_3 = 1.417 - \frac{0.007889}{2.834} \approx 1.4142$

We were able to approximate $\sqrt{2}$ to 4 decimal places with just 3 iterations of Netwon's method.

Extending Newton's Method for Optimization and for functions whose domain is not $\mathbb{R}^1$

Since optimization is nothing but finding the zeroes of a the derivative of a function, Newton's method can be extended for optimization as well.

The general formula for the next closer approximation for a critical point of a given function f, is $x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$. The function from the original formula is replaced by its derivative, and the derivative is replaced by the second derivative, leading to finding the zero of the derivative, rather than the original function itself.

Similarly, Newton's method for optimization can be extended to functions whose domain is all vectors in $\mathbb{R}^i$. This is done through replacing the derivative of f with the gradient of f, which is a vector filled with the partial derivatives of f, defined as $\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \\ \vdots \\ \frac{\partial f}{\partial x_i} \end{pmatrix}$, and replacing the second derivative with the Hessian, which a matrix filled with the second order partial derivatives of f, defined as $\mathbf{H}_f =$

$\begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_i} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_i} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_i \partial x_1} & \frac{\partial^2 f}{\partial x_i \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_i^2} \end{pmatrix}$. The problem is, matrix division is not defined, and thus division is replaced by multiplication by the inverse of the Hessian.

Putting these together, we obtain $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}^{-1}\nabla f$, where $\mathbf{H}$ is the Hessian matrix of f calculated at $\mathbf{x}_n$, and ∇f is the gradient of f calculated at $\mathbf{x}_n$.

Least Squares, Pseudoinverses, and the Gauss Newton Algorithm

Armed with the tools we need, we can now move on to the problem of minimizing the sum of the squares of residuals for a given data set.

Let's say we want to model a data set $\{x_i, y_i\}$ using the function $f(x, \boldsymbol{\beta})$, where f is a multivariable funciton of x, and the vector $\boldsymbol{\beta}$, which consists of the coefficients of the model. For example, in a quadratic model, $y = ax^2 + bx + c$, $\boldsymbol{\beta}$ will consist of a, b, and c.

Let's define r_i to be the sequence of residuals. Meaning that $r_i = y_i - f(x_i, \boldsymbol{\beta})$. In least squares, our goal is to minimize $\sum r_i^2$.

We can try using the formula we obtained earlier, $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}^{-1}\nabla f$. After plugging in f and $\boldsymbol{\beta}$ in place of f and $\mathbf{x}$, and after much simplification, we obtain the formula $\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n + 2\mathbf{H}^{-1}\mathbf{J}^\top\mathbf{r}$, where $\mathbf{J}$ is the Jacobian matrix of f calculated at $(\mathbf{x}, \boldsymbol{\beta}_n)$, defined as $\mathbf{J} = \begin{pmatrix} \frac{\partial f(x_1)}{\partial \beta_1} & \frac{\partial f(x_1)}{\partial \beta_2} & \cdots & \frac{\partial f(x_1)}{\partial \beta_j} \\ \frac{\partial f(x_2)}{\partial \beta_1} & \frac{\partial f(x_2)}{\partial \beta_2} & \cdots & \frac{\partial f(x_2)}{\partial \beta_j} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(x_i)}{\partial \beta_1} & \frac{\partial f(x_i)}{\partial \beta_2} & \cdots & \frac{\partial f(x_i)}{\partial \beta_j} \end{pmatrix}$, where $\boldsymbol{\beta}_n$ is the coef-

ficient vector of size j, $\mathbf{x}$ is a vector of size i, consisting of all the x data points, and $\mathbf{r}$ is the vector of all the residuals. The derivation of this simplification is outside the scope of this paper.

Now that we have a formula, we have made some progress, but there still remains a problem: The Hessian matrix can be singular. A singular matrix doesn't has a determinant of 0, meaning that it doesn't have an inverse, meaning that we can't always find the next $\boldsymbol{\beta}$.

For this to work for any case, we need to define something called a pseudoinverse: an inverse that only estimates what an actual inverse would do. In order to this, we will drop a term from the Hessian.

The Hessian simplifies to $\mathbf{H} = 2\sum_{i=1}^{m}\left(\frac{\partial r_i}{\partial \beta_j}\frac{\partial r_i}{\partial \beta_k} + r_i\frac{\partial^2 r_i}{\partial \beta_j \partial \beta_k}\right)$.

We can drop the term $r_i\frac{\partial^2 r_i}{\partial \beta_j \partial \beta_k}$.. Then after much simplifation, we obtain the formula $\mathbf{H} \approx 2\mathbf{J}^\top\mathbf{J}$

Finally, plugging in our estimate for the Hessian back in the formula, we get $\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n + 2(2\mathbf{J}^\top\mathbf{J})^{-1}\mathbf{J}^\top\mathbf{r} = \boldsymbol{\beta}_n + (\mathbf{J}^\top\mathbf{J})^{-1}\mathbf{J}^\top\mathbf{r}$

So finally, we obtain the iterative formula, $\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n + (\mathbf{J}^\top\mathbf{J})^{-1}\mathbf{J}^\top\mathbf{r}$

One important thing to note is that when computing the Jacobian Matrix, partial derivatives are taken with respect to β_i, and not with respect to x. This mistake is quite easy to make on your first time using the algorithm. For example, for $f(x, \boldsymbol{\beta}) = \beta_0 + \beta_1 x + \beta_2 x^2$, the partial derivatives in the Jacobian will be $\frac{\partial f}{\partial \beta_0} = 1$, $\frac{\partial f}{\partial \beta_0} = x$, and $\frac{\partial f}{\partial \beta_0} = x^2$, and nowhere are the x terms modified.

Example

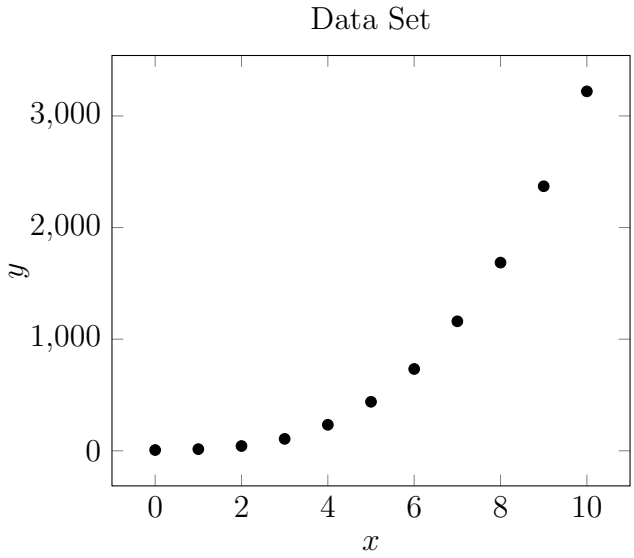
Let's take a look at an example of the Guass Newton Algorithm in action

For the purpose of this example, I wrote a python class for computing these approximations that takes in our initial guess and our desired polynomial model. The class includes a Jacobian matrix calculating function, which is quite simple to implement, since all polynomial models have Jacobian matrices of the form $\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ 1 & x_3 & x_3^2 & \cdots & x_3^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_i & x_i^2 & \cdots & x_i^n \end{pmatrix}$, unless any of the components of $\boldsymbol{\beta}$ are 0, in which case, the column corresponding to that component is also filled with 0s. Another function the class includes is a function that calculates the residual vector, and finally, a third function that executes one iteration of the algorithm, taking in the initial guess as a parameter. All Matrix operations in the algorithm are handled by Python's Numpy library by defining matrices as Numpy arrays.

Let's take a look at the following data set:

x	y
0	7
1	15
2	43
3	107
4	233
5	439
6	733
7	1160
8	1686
9	2370
10	3220

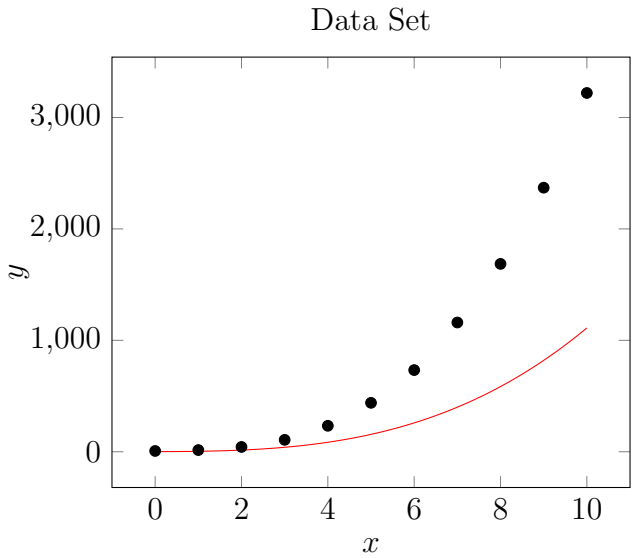
Graph:



Our goal is to model this data set using a cubic model, $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$

Let's start with an intial guess of $\boldsymbol{\beta} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$

Graph:



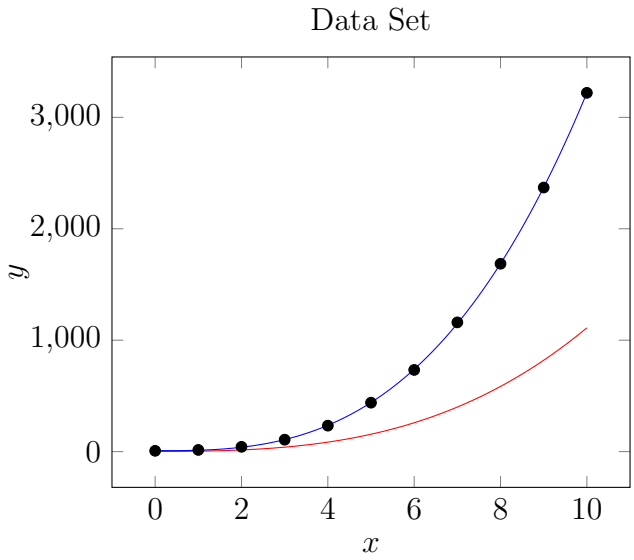
Now let's run $\boldsymbol{\beta}$ through five iterations of the Gauss Newton Algorithm, and see the result:

Iteration 1: $\boldsymbol{\beta} = \begin{pmatrix} 9.74825175 \\ -4.01515152 \\ 3.56934732 \\ 2.89335664 \end{pmatrix}$

Iteration 2: $\boldsymbol{\beta} = \begin{pmatrix} 9.74825175 \\ -4.01515152 \\ 3.56934732 \\ 2.89335664 \end{pmatrix}$

On the second iteration of the algorithm, $\boldsymbol{\beta}$ didn't change at all, and it didn't change after 5 iterations, or 10 iterations, meaning that the changes are so precise that they are outside of Python's float precision.

Graph:



The blue curve corresponds to the graph after the first iteration of the Gauss Newton Algorithm, and the red curve corresponds to the initial guess.

Only one iteration of the Gauss Newton Algorithm was required to achieve the potential of the algorithm for this data set for this model.

Covergence Properties

The Gauss Newton algorithm converges very rapidly in many cases, where only one iteration is needed to achieve its potential, and converges slowly in other cases. In some cases the algorithm may even completely diverge.

Since we use a pseudoinverse in this algorithm, we are approximating a matrix, and hence we must pay attention to condition numbers. If the Hessian matrix is ill-conditioned, meaning that it has a high condition number and is susceptible to large errors from small approximations, the algorithm may diverge.

Let's take a look at an example where the algorithm diverges, and let's analyze the reasons for divergence.

We can look at the Google Trends data for the term "Coronavirus" from 2016. The dataset contains 260 elements, each corresponding to the relative popularity of the term "Coronavirus" in a given time interval. Let's run the data through my Gauss Netwon class in Python, and see what happens.

Every iteration of the algorithm made the last component of the coefficient vector increase by 1, until it became a 2 digit number, after which each iteration increased it by 10, until it reached 3 digits, after which it increased it by 100 every iteration, and so on. Even after 1000 iterations of the algorithm, the coefficient vector did not seem to converge to a finite number. Let's explore a reason why this might be the case.

For a given matrix A, its condition number C decides how much approximating A will affect $A\mathbf{x} = \mathbf{b}$. The larger C is, the greater impact approximation will have on $\mathbf{b}$.

In the Gauss-Newton Algorithm, we approximate the Hessian matrix in order to avoid instances of the Hessian being singular. Therefore, it is important to look at the condition number of the Hessian matrix. The norm of the Hessian is $||\mathbf{H}|| = \sqrt{\lambda_{\max}(\mathbf{H}^\top \mathbf{H})}$, and the norm of the inverse of the Hessian is $||\mathbf{H}^{-1}|| = \sqrt{\lambda_{\max}((\mathbf{H}^{-1})^\top \mathbf{H}^{-1})}$, which is nothing but $\sqrt{\frac{1}{\lambda_{\min}(\mathbf{H}^\top \mathbf{H})}}$, thus giving us the condition number of $C = \sqrt{\lambda_{\max}(\mathbf{H}^\top \mathbf{H})}\sqrt{\frac{1}{\lambda_{\min}(\mathbf{H}^\top \mathbf{H})}} = \sqrt{\frac{\lambda_{\max}(\mathbf{H}^\top \mathbf{H})}{\lambda_{\min}(\mathbf{H}^\top \mathbf{H})}}$. From this, we can conclude that if the largest eigenvalue of $\mathbf{H}^\top \mathbf{H}$ to too much larger than the its smallest eigenvalue, the condition number will be high, and the algorithm may diverge. Since eigenvalues are just roots of the characteristic polynomial, we can analyze the characterisictic polynomail for a given Hessian, and note how dense the

roots are. The denser the roots, the less harm will be caused by approximating the Hessian. The calculation of eigenvalues is an extremely time-consuming process, and not practical for data sets of size 260. However, if one of the eigenvalues is negative, which is the reason why we would approximate the hessian in the first place, we won't be able to calculate the condition number.

Note: The above part of the paper is an original result that has not been peer reviewed, and is therefore subject to error.

Applications

The Gauss Newton Algorithm is extremely powerful due to the fact that it can use any real to real model to make an approximation. Some of the areas that the algorithm is used include:

- Statistics
- Physics
- Economics
- Computer Science

and any area that needs to model a nonlinear relationship bewteen two variables. As we saw in the paper, the algorithm uses concepts learned from Linear Algebra this year, including:

- transposes
- inverses
- multiplication
- norms and condition numbers

Not only that, but the algorithm also uses concepts learned from Multivariable calculus, including:

- partial derivatives
- Jacobians
- gradients

- vector valued and multivariate functions

Some potential weaknesses of the algorithm include:

- divergence when Hessian is ill-conditioned
- divergence when initial guess is too far off from points
- not the best approximation when the inverse of the Hessian exists

Works Cited

https://www.statisticshowto.com/gauss-newton-method/

http://fourier.eng.hmc.edu/e176/lectures/NM/node36.html

https://see.stanford.edu/materials/lsoeldsee263/07-ls-reg.pdf

http://www.math.umd.edu/~petersd/460/html/nonlinls.html

https://www.sciencedirect.com/topics/mathematics/gauss-newton-method

https://www.youtube.com/watch?v=Kln0ZQ7sX8k

https://en.wikipedia.org/wiki/Gauss%E2%80%93Newton_algorithm