Maximum Likelihood and Numerical optimization (v1)

Thomas Bourany

The University of Chicago

TA session – Mathcamp 2022

September 2022

1 Introduction

The aim of this exercise is to introduce the basics of Maximum Likelihood estimation and Numerical optimization techniques and to use them jointly in a simple example.

2 Maximum Likelihood and Probit and Logit

The idea of the Maximum likelihood estimation is to choose parameters (denoted $\theta \in \Theta \subset \mathbb{R}^K$, with K the number of parameters) to maximize the likelihood function $\mathcal{L}(\theta|x)$ – a function that map the parameters into a value of the probability of obtaining the observed data. Note that the observed values are following the same Data Generating process (DGP) and independent (or i.i.d.): $X \overset{i.i.d.}{\sim} F(x|\theta)$

Depending if the outcome variable is continuous or discrete, for a dataset of n observations, the likelihood writes with either the probability mass function (pmf) or the probability density function (pdf). Often one transforms the function by taking logs (to obtain the log-likelihood)

$$\mathcal{L}^{\text{disc}}(\theta) \equiv \mathcal{L}^{\text{disc}}(\theta|\{x_i\}_{i=1}^n) = \prod_{i=1}^n \mathbb{P}(X = x_i|\theta) \qquad \Rightarrow \qquad \ell(\theta) = \sum_{i=1}^n \log(\mathbb{P}(X = x_i|\theta))$$

$$\mathcal{L}^{\text{cont}}(\theta) \equiv \mathcal{L}^{\text{cont}}(\theta|\{x_i\}_{i=1}^n) = \prod_{i=1}^n f(x_i|\theta) \qquad \Rightarrow \qquad \ell(\theta) = \sum_{i=1}^n \log(f(x_i|\theta))$$

In the context of the Probit and Logit models, the DGP of the dependent variables (now written y) takes the form of a Bernoulli distribution (taking values 0 or 1) and with pmf – with probability of 1 being p – likelihood and log-likelihood functions written as:

$$\mathbb{P}(Y = y) = p^y(1-p)^{1-y} \qquad [pmf]$$

$$\mathcal{L}(\theta|\{y\}) = \prod_{i=1}^n p_i^{y_i}\left(1-p_i\right)^{1-y_j} \qquad [lik.f]$$

$$\ell(\theta|\{y\}) = \sum_{i=1}^n y_i \log\left(p_i\right) + (1-y_i)\log\left(1-p_i\right) \qquad [log{-}lik.f]$$

The difference between probit and logit is the form of the non-linear *link* function.

2.1 Logit

Logit link, the probability of 1, p_i is modeled using the logistic function in addition to a linear function (between the parameters β and independent variables X_i, so $\sigma(X_i'\beta)$ writes:

$$p_i = \sigma(X_i'\beta) = \frac{\exp(X'\beta)}{1+\exp(X'\beta)} = \frac{1}{1+\exp(-X'\beta)}$$

Note that probability is related to the odd-ratio (probability of an event happening compared to the event not happening),

$$\text{logit}(p_i) = \log\left(\frac{p_i}{1-p_i}\right) = X'\beta = \beta_0 + \sum_k \beta_k x_k$$

As a result, we can give the partial derivative of p_i with respect to the parameters β_0 and β_k:

$$\begin{aligned}
\frac{\partial p_i}{\partial \beta_0} &= \frac{\exp(X'\beta)}{(1+\exp(X'\beta))^2} = p_i(1-p_i) \\
\frac{\partial p_i}{\partial \beta_k} &= \frac{x_k \exp(X'\beta)}{(1+\exp(X'\beta))^2} = x_1 p_i(1-p_i)
\end{aligned}$$

And the derivatives of the likelihood functions (which will be equated to zero in the First order conditions of the optimization problem) are derived as:

$$\begin{aligned}
\frac{\partial \ell(\beta \mid \mathbf{y};\mathbf{x})}{\partial \beta_0} &= \sum_{i=1}^{n} y_i \frac{\partial \log(p_i)}{\partial p_i}\frac{\partial p_i}{\partial \beta_0} + (1-y_i)\frac{\partial \log(1-p_i)}{\partial p_i}\frac{\partial p_i}{\partial \beta_0} \\
&= \sum_{i=1}^{n} \frac{y_i}{p_i}\frac{\partial p_i}{\partial \beta_0} - \frac{1-y_i}{1-p_i}\frac{\partial p_i}{\partial \beta_0} \\
&= \sum_{i=1}^{n} y_i - p_i = 0
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \ell(\boldsymbol{\beta} \mid \mathbf{y};\mathbf{x})}{\partial \beta_k} &= \sum_{i=1}^{n} y_i \frac{\partial \log(p_i)}{\partial p_i}\frac{\partial p_i}{\partial \beta_k} + (1-y_i)\frac{\partial \log(1-p_i)}{\partial p_i}\frac{\partial p_i}{\partial \beta_k} \\
&= \sum_{i=1}^{n} \frac{y_i}{p_i}\frac{\partial p_i}{\partial \beta_k} - \frac{1-y_i}{1-p_i}\frac{\partial p_i}{\partial \beta_k} \\
&= \sum_{i=1}^{n} x_i(y_i - p_i) = 0
\end{aligned}$$

2.2 Probit

Probit link, the probability of 1, p_i is modeled using the c.d.f. of the Normal distribution, in addition to a linear function (between the parameters β and independent variables X_i, so $\Phi(X_i'\beta)$ writes:

$$p_i = \Phi(X_i'\beta)$$

with $\Phi(x) = \int_{-\infty}^{x} \phi(y)dy$ and $\phi(y) = \frac{1}{\sqrt{2\pi}} e^{-x^2}$ respectively the c.d.f and p.d.f of the Normal distribution with mean 0 and variance 1.

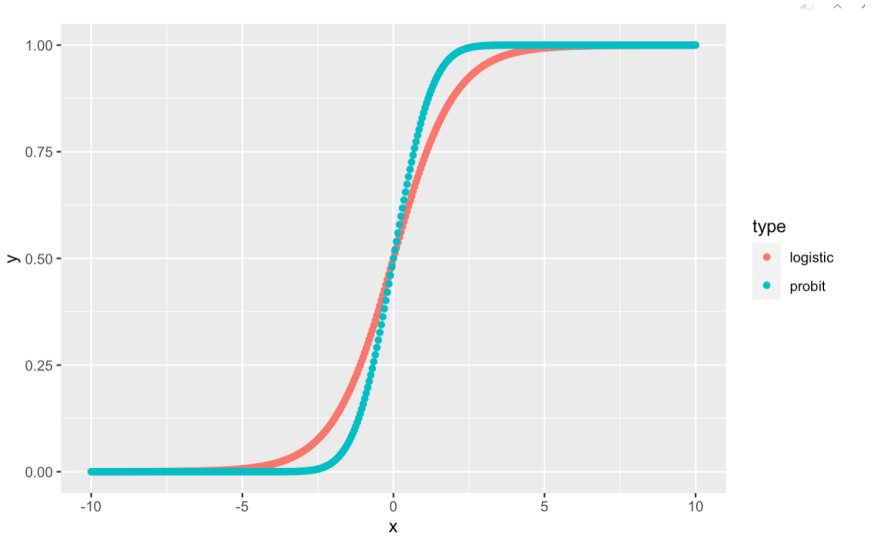
$$\text{probit}\left(p_i\right) = \Phi^{-1}\left(p_i\right) = X_i'\beta = \beta_0 + \sum_k \beta_k x_i$$

Based on the fact that the p.d.f is the derivative of the c.d.f., we have $\frac{\partial\Phi\left(X_i'\beta\right)}{\partial\left(X_i'\beta\right)} = \phi\left(X_i'\beta\right)$. The partial derivatives with respect to the variable β_0 and β_k are respectivalely:

$$\begin{aligned}
\frac{\partial p_i}{\partial \beta_0} &= \frac{\partial \Phi\left(X_i'\beta\right)}{\partial\left(X_i'\beta\right)} \frac{\partial\left(X_i'\beta\right)}{\partial\beta_0} = \varphi\left(X_i'\beta\right) \\
\frac{\partial p_i}{\partial \beta_k} &= \frac{\partial \Phi\left(X_i'\beta\right)}{\partial\left(X_i'\beta\right)} \frac{\partial\left(X_i'\beta\right)}{\partial\beta_k} = \varphi\left(X_i'\beta\right) x_i
\end{aligned}$$

The partial Derivatives of log-likelihood function are

$$\begin{aligned}
\frac{\partial \ell(\beta \mid \mathbf{y}; \mathbf{x})}{\partial \beta_0} &= \sum_{i=1}^{n} y_i \frac{\partial \log\left(p_i\right)}{\partial p_i} \frac{\partial p_i}{\partial \beta_0} + \left(1 - y_i\right) \frac{\partial \log\left(1 - p_i\right)}{\partial p_i} \frac{\partial p_i}{\partial \beta_0} \\
&= \sum_{i=1}^{n} \frac{y_i}{p_i} \frac{\partial p_i}{\partial \beta_0} - \frac{1 - y_i}{1 - p_i} \frac{\partial p_i}{\partial \beta_0} = \sum_{i=1}^{n} \frac{y_i \varphi\left(X_i'\beta\right)}{\Phi\left(X_i'\beta\right)} - \frac{\left(1 - y_i\right)\varphi\left(X_i'\beta\right)}{1 - \Phi\left(X_i'\beta\right)} \\
&= \sum_{i=1}^{n} \frac{\varphi\left(X_i'\beta\right)\left(y_i - \Phi\left(X_i'\beta\right)\right)}{\Phi\left(X_i'\beta\right)\left(1 - \Phi\left(X_i'\beta\right)\right)} \\
\frac{\partial \ell(\beta \mid \mathbf{y}; \mathbf{x})}{\partial \beta_k} &= \sum_{i=1}^{n} y_i \frac{\partial \log\left(p_i\right)}{\partial p_i} \frac{\partial p_i}{\partial \beta_k} + \left(1 - y_i\right) \frac{\partial \log\left(1 - p_i\right)}{\partial p_i} \frac{\partial p_i}{\partial \beta_k} \\
&= \sum_{i=1}^{n} \frac{y_i}{p_i} \frac{\partial p_i}{\partial \beta_k} - \frac{1 - y_i}{1 - p_i} \frac{\partial p_i}{\partial \beta_k} = \sum_{i=1}^{n} \frac{y_i \varphi\left(X_i'\beta\right) x_i}{\Phi\left(X_i'\beta\right)} - \frac{\left(1 - y_i\right)\varphi\left(X_i'\beta\right) x_i}{1 - \Phi\left(X_i'\beta\right)} \\
&= \sum_{i=1}^{n} \frac{\varphi\left(X_i'\beta\right)\left(y_i - \Phi\left(X_i'\beta\right)\right)}{\Phi\left(X_i'\beta\right)\left(1 - \Phi\left(X_i'\beta\right)\right)} x_i
\end{aligned}$$



3 Numerical optimization

In this section, we are concerned with the minimization of a function $\min f(X)$ where $X \subset \mathbb{R}^K$, a set without binding constraints. Constrained optimization works with similar algorithms but includes the constraints in very specific ways (penalization, computation of Lagrange multipliers, etc.). Usually, these routines create sequences of points x_n – called minimizing sequences – such that $\lim_{n\to\infty} f(x_n) = f(x^\star) = \min_x f(x)$. In the following, I'll consider – as it is the convention in optimization/machine learning fields – a minimization. It is easy to switch to a maximization by noticing that $\max f(x) = \min -f(x)$ and below all the gradients $\nabla f(x) = (\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_K})^T$ should become $-\nabla f(x)$ in the case of a maximization.

3.1 Gradient descent

One possible direction to go is to figure out what the gradient $\nabla f(X_n)$ is at the current point and take a step down the gradient towards the minimum. The gradient can be calculated by symbolically differentiating the objective function or by using automatic differentiation like Torch and TensorFlow does. Using a fixed step size α means updating the current point X_n by:

$$x_{n+1} = x_n - \alpha \nabla f(x_n)$$

This leads to taking the path of the steepest descent, which looks vaguely like a ball rolling down a hill toward one of the local minima. Note that the maximization would have the opposite sign $x_{n+1} = x_n + \alpha \nabla f(x_n)$.

The problem with this method is in setting the learning rate. If you set the rate too low, Gradient Descent takes forever to find the solution, taking many tiny steps toward the solution. Setting the learning rate too high, and it will wildly oscillate around the minima without converging. Even worse, the best learning rate changes from function to function, so there isn't a single value that makes for a good default (some data scientists' blogs suggest between 0.01 and 0.1, but honestly, I don't trust that at all).

Some methods compute (ex-ante or iteratively) the optimal learning rate. For example, the optimal rate can be computed by line search, i.e. using the minimization of $\min_{\widetilde{\alpha}} f\big(x_n - \widetilde{\alpha}\nabla f(x_n)\big)$. With a second-order Taylor expansion of $f(x)$, this yields :

$$\alpha^\star(x_n) = \frac{||\nabla f(x_n)||}{\nabla f(x_n)^T H(f)(x_n) \nabla f(x_n)}$$

with $||\nabla f(x_n)|| = \nabla f(x_n)^T \nabla f(x_n)$ the norm of the gradient. Note that it requires more information about the function – the Hessian $H(f)$ – which is costly to compute obviously.

3.2 Conjugate Gradient descent

Another example is conjugate gradient descent, which smooths the sequence of directions taken by the minimizing sequence. A good reference can be found here.

While the theory behind this might be a little involved, the math is pretty simple. The initial search direction S_0 is the same as in gradient descent. Subsequent search directions S_n are computed by:

$$d_n = -\nabla f(x_n) + \beta_n d_{n-1}$$

There are many methods to find the weights β; two common methods are Fletcher-Reeves (FR) and Polak-Ribière (PR)

$$\beta_n^{FR} = \frac{\nabla f(x_n)^T \left(\nabla f(x_n) - \nabla f(x_{n-1})\right)}{||\nabla f(x_{n-1})||}$$
$$\beta_n^{PR} = \frac{||\nabla f(x_n)||}{||\nabla f(x_{n-1})||}$$

The current location x_n is updated with this search direction, using a step size α computed by line search (the optimal rate by minimizing $\min_{\widetilde{\alpha}} f\left(x_n + \widetilde{\alpha} d_n\right)$

$$x_{n+1} = x_n + \alpha_n d_n$$

As a result, the direction is not updated fully by the steepest gradient but some weighted sum of current and past gradients. This is beneficial as in certain cases, as the gradient can change by more than 90 degrees from one iteration to the next, while the sequence of search directions d_n being used is much smoother.

3.3 Newton-Raphson methods

The Newton-Raphson algorithm is used for root finding – i.e. search for the zeros of non-linear system of equation $g(x) = \underline{0}_K$ – or optimization, where the non-linear function $g(x)$ is nothing else than the First Order conditions of the function to minimize $g(x) = \nabla f(x)$. It uses a first-order approximation of the non-linear function:

$$g(x_{n+1}) = 0 \qquad \Rightarrow \qquad g(x_{n+1}) = g(x_n) + J(g)(x_n) \cdot (x_{n+1} - x_n) = 0$$

where $J(g)$ is the jacobian matrix of the function $g : \mathbb{R}^K \to \mathbb{R}^K$,

$$J(g) = \begin{bmatrix} \nabla g_1(x)^T \\ \vdots \\ \nabla g_K(x)^T \end{bmatrix} = \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial x_1} & \cdots & \frac{\partial g_m}{\partial x_n} \end{bmatrix}$$

which implies a new point chosen

$$x_{n+1} = x_n - J(g)(x_n)^{-1} g(x_n)$$

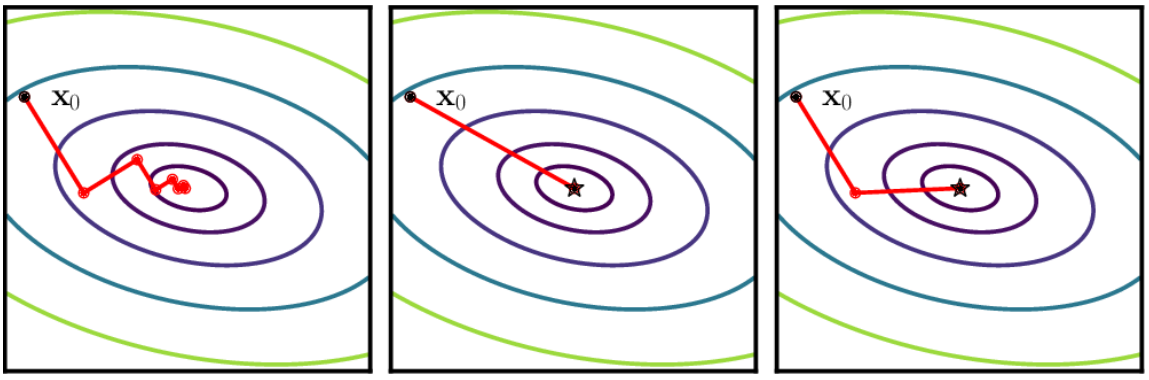
In the case of optimization, the function is the First Order condition, i.e. $g(x_n) = \nabla f(x_n) = 0$ and hence the update of x_n becomes:

$$x_{n+1} = x_n - H(f)(x_n)^{-1} \nabla f(x_n)$$

where $H(f)(x)$ is the Hessian matrix of the function $f : \mathbb{R}^K \to \mathbb{R}$.

Computing and inverting the Hessian matrix is very costly, especially in large dimensions, so, usually, optimization routines solve the linear system

$$H(f)(x_n)(x_{n+1} - x_n) = -\nabla f(x_n)$$



(a) Steepest descent (b) Newton's method (c) Conjugate gradient

Some methods (like BFGS, which is probably one of the best set of methods on the market, or Broyden's family of methods) are quasi-Newton methods, which construct the inverse of the Hessian of $f(x)$ (or Jacobian of $g(x)$) iteratively with the sequence $B_n \approx H_n^{-1}$, where the general idea come from the linear approximation

$$B_{n+1}(x_{n+1} - x_n) = \nabla f(x_{n+1}) - \nabla f(x_n)$$

Hence, the ***BFGS*** algorithm constructs iteratively the sequence B_n and a direction d_n as:

1. the direction d_n is obtained by solving the linear system:

$$B_n d_n = -\nabla f(x_n)$$

2. the new point is updated (again where α can be computed optimally with line search)

$$x_{n+1} = x_n + \alpha d_n$$

3. the inverted Hessian is updated with the formula[1]

$$B_n = B_{n-1} + \frac{y_n y_n^T}{y_n^T(x_n - x_{n-1})} + \frac{B_n(x_{n+1} - x_n)(x_{n+1} - x_n)^T B_n^T}{(x_n - x_{n-1})^T B_n (x_{n+1} - x_n)}$$

with $y_n = \nabla f(x_{n+1}) - \nabla f(x_n)$

This method, as well as all the other above, are implemented (keys in hand so you don't have to code up anything) in optimization packages in many language (at least in Python, Julia, Matlab, and a priori in R too)

3.4 Stochastic gradient descent methods

Last but not least, the Stochastic gradient descent (or SGD), used a lot in machine learning and neural network (deep learning), is concerned with the minimization of a function

$$\min_x \mathbb{E}\big[f(x, Y)\big] \approx \min_x \frac{1}{n} \sum_{i=1}^{N} f(x, y_i)$$

where Y is a random variable and the RHS is the Monte Carlo approximation of the expectation operator, with a sample of size N. This family of algorithm can be traced back to the 50s by the work of Robbins-Monro on stochastic approximation methods. Typically, if one would have to implement the standard gradient descent method, one would update x_n with

$$x_{n+1} = x_n - \alpha \sum_{i=1}^{N} \nabla_x f(x, y_i)$$

But one could notice that in large dimension and large sample size, the computation of the sum of gradients $\nabla_x f(x, y_i)$ can be very costly. As a result, the idea of the Stochastic Gradient Descent is to compute this update using *mini-batch* and compute only a subset of these observations:

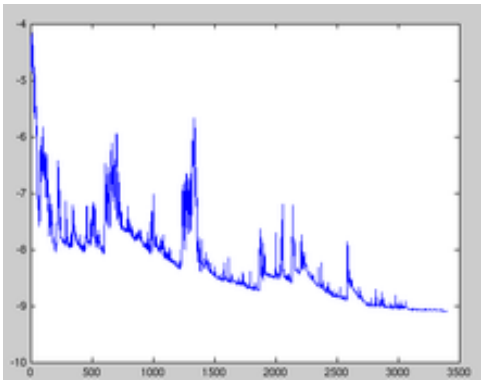
$$x_{n+1} = x_n - \alpha \sum_{i=1}^{m} \nabla_x f(x, y_i)$$

where m is the size of the mini-batch, and $m \ll N$. Sometimes, especially in the case of the perceptron (which is the easiest update of a neural network) the idea is to use simply a batch of one observation $m = 1$.

The advantage is speed and parallelization – one can compute the gradient for each mini-batch in parallel. The inconvenient is that now, no one has any guarantee of convergence of the minimizing sequence x_n toward the local or global minimum. If all the methods introduced above have clear mathematical foundations, proof of convergence and speed and asymptotic properties,

[1] The intuitive reason is that, because the Hessian is symmetric and positive definite, the inverse is also symmetric, and the added terms $B_{n+1} = B_n + \beta y_n y_n^T + \gamma v_n v_n^T$ are such that $y_n = \nabla f(x_{n+1}) - \nabla f(x_n)$ and $v_n = B_n(x_n - x_{n-1})$ and β and γ are chosen optimally, but really no need to remember these subtle details

one can not say the same for stochastic gradient descent methods (one can derive some convergence results only in the case of the perceptron), c.f. the picture above, the rate of convergence fluctuates a lot!



In practice, especially with optimization of hyperparameters (choice of the size of batch m, learning rate α, some properties of the DGP) and cross-validation (training the algorithm on parts of the sample and testing / evaluating it on another part of the sample) and plenty of engineering tricks and rules of thumbs, the methods work pretty well – it's just sad that we don't exactly know why – but hopefully, this field of research is growing very rapidly.

4 Example – Exercise – Labor force participation

In this example, we evaluate which factors are important in determining the participation of women in the labor market. This example was borrowed from Greene's textbook Econometric Analysis[2], using data from uses data from the Mroz (1987) study of the labor supply of married women. The end goal of this dataset is to estimate a wage equation for women using the Heckman Selection model (Heckman (1976) "The Common Structure of Statistical Models of Truncation, Sample Selection, and Limited Dependent Variables and a Simple Estimator for Such Models" Annals of Economic and Social Measurement, 5, 475-492.), but this will be introduced at the end for completeness (but not covered during the TA session).

The variables considered are:

1. Labor force participation (LFP)
2. Woman's age (WA)
3. Square of Woman's age (to be created)
4. Family income, in 1975 dollars (FAMINC)
5. Wife's educational attainment, in years (WE)
6. Number of children (KIDS, to be created as dummy variable 1, if the number is kids is higher than 0, or 0 if not (use KL6 and K618))

Questions

1. Process and clean the data to created the 6 variables needed
2. Use preexisting packages to estimate the logit and probit parameters (glm in R, statsmodels in Python, fitglm for Matlab, glm for Julia)
3. Create a function for the log-likelihood of the logit and probit models, use an optimization routine to minimize this function numerically
4. Create a function for the gradient of the log-likelihood (derivative w.r.t. to parameters c.f. analytical formula above), use that as an input in optimizations routines
5. Use gradient descent methods (manually since you computed the gradient!) to minimize the likelihood (c.f. section 3.1. but no need to use the optimal learning rate)
6. Use Newton-Raphson methods to find the zero of the First order conditions – for that, you would need to compute numerically (or with automatic differentiation) the Hessian of the function, (i.e. the gradient of the gradient)
7. Did the different estimations of these parameters yield similar results, and if not, what could be a potential reason?

[2] https://pages.stern.nyu.edu/~wgreene/Text/econometricanalysis.htm