Incorporating Symbolic and Automatic Differentiation in a General-Purpose Likelihood Optimization Function

Queenie Zeng (supervisor: Dr. Bolker)

April 28, 2021

Abstract

Maximum Likelihood Estimation plays a crucial role in fitting statistical models. After constructing a model, the goal is to seek for parameter values that best describe the observed data. This is achieved by maximizing the likelihood function, or the conditional probability of the model parameter given the observed data. The optimization is most reliable with gradient-based methods supported by powerful differentiation techniques, such as symbolic differentiation and automatic differentiation. Both methods ensure numerical stability and run time efficiency. The purpose of this project is to build a `R` package that implements a more robust and extensive MLE framework than the default base R `mle` function. The package incorporates symbolic differentiation and automatic differentiation to provide computational advantages when fitting more complex models and estimating non-linear parameters. The package source code can be found on https://github.com/queezzz/qzmle.

1 Introduction

1.1 Likelihood Optimization

Given a sample of data $x_1, ..., x_n$ observed from an unknown population, where each data point is drawn independently and is assumed to follow the same parametric distribution, or independent and identically distributed (i.i.d.). The distribution has the probability density function $f(x_1, ..., x_n|\theta_i)$ with a set of parameters θ_i, then the likelihood function is the probability of obtaining the observed data given a particular value of the distribution parameters, and it is denoted as:

$$\mathrm{L}(\theta|x_i) = \prod_{i=1}^{n} f(x_i|\theta)$$

Therefore, estimating parameter values that will make the observed data most probable is equivalent to maximizing the likelihood function with the optimal parameter values $\widehat{\theta}$ such that:

$$\widehat{\theta} = \mathrm{argmax}_{\theta}\mathrm{L}(\theta)$$

For mathematical conveniences, we often use the likelihood function with logarithmic transformation. Since logarithm is a monotonic function, maximizing the log-likelihood is equivalent to maximizing the likelihood. Also, when derivative-based method is used for the optimization, it involves assuming that the maximum occurs at where the first-order derivatives of the function with respect to each parameters are all zero ($\frac{\partial L}{\partial \theta} = 0$ for all θ_i), and the hessian matrix is negative semi-definite indicating concavity. Since the log-likelihood function is in a form of summation, it makes computing the gradients more convenient because the derivatives of a summation is simply a summation of derivatives. Furthermore, software optimizer comply to the convention of minimizing the objective function rather than maximizing. Thus in this case, we will be minimizing the negative log-likelihood function, which is denoted by the lowercase l:

$$\mathrm{l}(\theta) = -\sum_{i=1}^{n} \log(f(x_i|\theta))$$

1.2 Optimization Methods in `R`

The optimization process can be executed using `R`'s general-purpose optimizer, `optim()`, by inputting the objective function, starting values for function parameters, and selected optimization method. The objective function will be the negative log-likelihood function as discussed, and the initial values should be sensible to the assumed model; but how do we choose which optimization method to use?

The default method provided by `optim()` is a derivative-free method called Nelder-Mead simplex (Nelder, 1965). After the initializing the start point, a "triangular" simplex is constructed and is being reshaped at every iteration in attempt to proceed toward the global minimum. Heuristic search approach like Nelder-Mead gains its popularity for the simplicity of not requiring any information about the objective function, but one of its major drawback is that the convergence of complex functions can be excessively slow and computationally expensive because the function is being evaluated at every step.

Conversely, derivative-based methods is more efficient because it takes the advantage of the assumption that the objective function is smooth and has continuous derivatives, and thus providing clues for the direction of the minimum point. The simplest derivative-based method is the Newton's Method, it assumes the function is quadratic shape around the minimum where the gradient is zero. However, this method requires to derive the inverse of the hessian matrix, and that can be quite computationally expensive and sometimes not possible for functions in peculiar shapes. The safer alternative is to use the quasi-Newton method that exempt from deriving the actual inverted hessian, but instead to approximate the inverted hessian to reduce computing cost (Shanno 1970).

The implementation of this project will use the BFGS method, which is one of the most efficient quasi-Newton methods (Dai 2002), and it is also a built-in method inside `optim`. Noted that the user can pass their gradients to `optim` as well.

1.3 Computation of Gradients

Although the common practices suggest to manually working out the derivatives with pen and paper, it is an extremely time-consuming and error-prone task especially with mathematically complex equations. There also exists problems with no analytic solution available. Therefore, it is more worthwhile to consider computational alternatives that can yield results faster and with more precision.

The calculation of derivatives can be automated through several different techniques, such as numerical differentiation, symbolic differentiation, and automatic differentiation.

Numerical Differentiation

Numerical differentiation is also commonly referred as differentiating using the finite-difference method, such as evaluation using the forward difference formula:

$$f'(x)_{approx} = \frac{f(x+h) - f(x)}{h}$$

The pseudo-code correspondence is very simple to implement as follows:

Algorithm 1: Numerical differentiation using the forward difference formula

Result: Numeric value
Input : The objective function f and a point a
Output: Derivative of objection function $f'(a)$ at point a

```
1 Function NumDiff(f, a, h):
2 |   return(f(a + h) − f(a))/h)
```

However, finite-difference methods are not robust because of the truncation errors, e_t in the approximating process (Stepleman and Winarsky 1979). For instance, the forward differences formula is derived from Taylor expansion of the $f(x+h)$ as follows:

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + ...$$

Then the right hand side of the forward difference formula can derived as the left hand side of the equation below:

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{1}{2}hf''(x) + \frac{1}{6}h^2 f'''(x) + ...$$

Since the forward difference formula is approximating $f'(x)$ with the left hand side of the equation above, the remaining terms on the right hand side are the truncation errors where:

$$e_t = \frac{1}{2}hf''(x) + \frac{1}{6}h^2 f'''(x) + ...$$

As shown, the choice of h is critical to the precision of the approximation. However, the dilemma exists where larger h will cause greater truncation errors, while h being too small will accumulate rounding errors from the computer. Therefore, numerical differentiation innately inherited the lacking of precision and can further result in numerical instability for some functions.

Symbolic differentiation

On the other hand, symbolic differentiation does not require any numerical calculations, but instead producing the algebraic expression of the derivative. The implementation involves storing an extensive list of derivative rules at run time. The objective function will be extracted into sub-expression by the order of elementary operators and prescribed chain-rule, then the algorithm will look up the derivative rules from the pre-existing list to generates new expression modules of the derivatives. The over-simplified pseudo-code is presented below:

Algorithm 2: Symbolic Differentiation

Result: Derivative in expression form
Input : *(Example)* $x^2\cos(x-7)(\sin(x))^{-1}$
Output: $x^2\sin(7-x)\csc(x)+x^2(-\cos(7-x))\cot(x)+2x\cos(7-x)\csc(x)$

```
extract sub-expressions from objective function
for sub-expression in input do
    if operation1 in sub-expression then
        apply derivative rule for operation1
    else if operation2 in sub-expression then
        apply derivative rule for operation2
    ...
end
return Construct final expression with sub-expression derivatives using chain-rule
```

Symbolic differentiation prevails over numerical differentiation for its exact (floating-point) computation and the ease of numerical instability. However, using this technique to solve more complex compositions will involve more computing steps and producing longer expression due to duplicative nature of some derivative rules (ie. the product rules and the quotient rules). As the objective function grow in complexity, there are potential computational disadvantages of slow code and excess use of memory because the it has to process all information that is only available at run time.

Automatic Differentiation

Automatic Differentiation overcomes the shortcomings of the previous two algorithms. A popular implementation in C++ is the **CppAD** based on the idea of "operator loading" (Bell 2005). Starting with the source code for the objective function, the program will generate a computing trajectory, often referred as the "computational graph," where each node correspond to a variable with the a operation that compute its value, and the derivative of the prescribed operation (Griewank and Walther 2003). This preserves the floating-point precision for the intermediate values at each node, and enables the use of more sophisticated programming techniques, such as loops and templates, to reduce the computation cost in magnitude (Hogan 2014).

2 Implementation

R's `optim` uses numerical differentiation to approximate the gradients if is not provided by the users (`optim(..., gn=NULL)`). We want to prevent the pitfalls of finite-difference approximation, so we will specify our gradient function computed using symbolic and automatic differentiation.

There are various options of differentiation engines developed by R's open-source community. For this project, we decide to use the `Deriv` package for symbolic differentiation and the `TMB` package for automatic differentiation.

The `Deriv` package is built entirely using R, so it is very convenient and low-cost for simpler models with smaller data-sets.

The package `TMB`, stands for Template Model Builder, integrates with state-of-the-art C++ packages, such as **CppAD** and **Eigen**, which allows users to perform automatic differentiation through a compiled C++ script inside R (Kristensen et al. 2016). The user will define the objective function in C++ script, compile the script to evaluate the derivatives, then load results back into R for further analysis. However, composing the C++ script may be slightly discouraging for users who are not familiar with the package or the programming syntax of C++. Therefore, the program of this project will automate the process of generating the C++ script once the user inputs the model in R. The flowchart in 1 demonstrates the implementation for both differentiation routines.

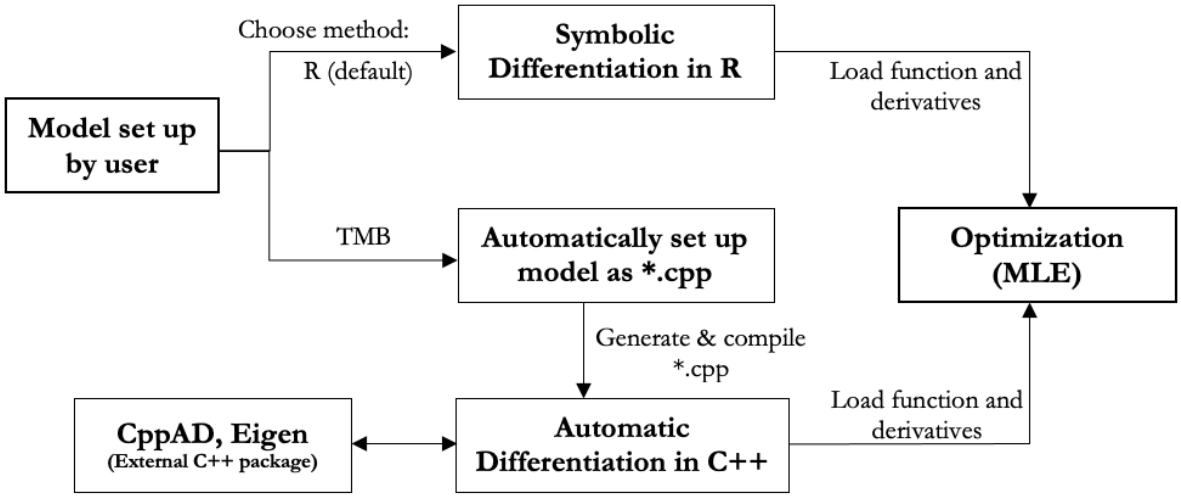

Figure 1: Process flow chart of the implementation: The user will specify the model and choose the differentiation method, then the gradients will be automatically generated and passed into `optim` for optimization

As shown above, generating gradients using symbolic differentiation with `Deriv` will be the default method of the program because it is light-weighted without the extra run-time to compile the script. The use of the `TMB` method is more appropriate for more complex models with larger number of variables or data, where the compilation time is worthwhile to outperform in the evaluation of gradients.

3 Applications

The simplest likelihood function can be expressed as $Y \sim f(\theta_i)$, where the response variable Y follows some distribution f with distributional parameters of θ_i (ie. $Y \sim \text{Binom}(p)$, $Y \sim \text{Gamma}(\alpha, \beta)$, etc.).

This package offers more flexibility in estimating non-linear parameters. For instance, the distributional parameter θ can depend on non-linear parameters w_i:

$$Y \sim f(\theta) \quad \theta = h(w_1, ..., w_i)$$

Those non-linear parameters w_i can use link functions g_i:

$$Y \sim f(\theta) \quad \theta = h(g_1(w_1), ..., g_i(w_i))$$

and may be determined by linear models of other covariates b_i:

$$Y \sim f(\theta) \quad \theta = h(g_1(w_1), ..., g_i(w_i))$$

Example: Tadpole Predation

Consider an example where we wish to model the predation rate of tadpoles of an African frog, *Hyperolius spingularis*, using the experiment data from Vonesh (2005) and can be retrieved in R from the `emdbook` package. Here is a snapshot of the data-set:

```
head(emdbook::ReedfrogPred)
```

```
##   density pred  size surv propsurv
## 1      10   no   big    9      0.9
## 2      10   no   big   10      1.0
## 3      10   no   big    7      0.7
## 4      10   no   big   10      1.0
## 5      10   no small    9      0.9
## 6      10   no small    9      0.9
```

The main variables of "density" indicates the initial tadpole population, "pred" indicates the presence of the predator, "size" indicates whether the tadpole is big or small, and "surv" is the number of surviving tadpoles after the experiment.

Assuming the predator-prey relationship can be modeled by the Holling type II response, where the predator attacks the tadpoles in a constant rate a, and it takes some handling time h for the predator to digest and seek for the next prey. Then the actual number of killed tadpole, k, would follow a binomial distribution with probability p and density N: (Bolker 2008)

$$p = \frac{a}{1 + ahN}$$

$$k \sim \text{Binom}(p,\ N)$$

The number of killed tadpoles can be produced by subtracting the number of survived tadpoles from the initial population. Furthermore, we want to explore whether the attack rate varies with the size of the tadpoles, so the variable "size" with the classes of "big" and "small" are transformed to 1 and 0, respectively stored in the new variable "nsize." Here is an overview of the updated data set:

```
##   density pred  size surv propsurv killed nsize
## 1      10   no   big    9      0.9      1     1
## 2      10   no   big   10      1.0      0     1
## 3      10   no   big    7      0.7      3     1
## 4      10   no   big   10      1.0      0     1
## 5      10   no small    9      0.9      1     0
## 6      10   no small    9      0.9      1     0
```

The attack rate should be strictly positive, so the link function of log is applied to the a. As mentioned, the attack rate is determined by the linear sub-model of prey size, in which b_0 can be interpreted as the attack rate on small tadpoles, and b_1 is the difference between attack rate on big tadpoles and small tadpoles, as shown:

$$\log(a) = b_0 + b_1 x$$

Here we will discussed about how to use the package to solve the model. The package also strikes to be more user-friendly. Instead of writing the likelihood function into a `R` function, the user can write it as a formula with the response variable on the left-hand side. Adapting our example, the formula is defined as follows:

```
form <- killed ~ dbinom(size = density,
                        prob = exp(log_a)/(1 + exp(log_a) * h * density))
```

Subsequently, we load the package and called the package's `mle` function. Note that just like the base R's `mle`, the start values of the parameters are required. The user can define the linear sub-model in the `parameter` argument, and also define the link functions using the `link` argument. The result of output follows the format of `mle` in base R.

```
## devtools::install_github("queezzz/qzmle")
library(qzmle)

qzmle::mle(form,
           start=list(h=4,log_a=c(1,2)),
           parameters=list(log_a~1+nsize),
           links=list(a="log"), data=rfp)
```

```
##
## Call:
## qzmle::mle(form = form, start = list(h = 4, log_a = c(1, 2)),
##     data = rfp, links = list(a = "log"), parameters = list(log_a ~
##         1 + nsize))
##
## Coefficients:
## log_a.(Intercept)       log_a.nsize                 h
##       -1.64056868        0.26665566       -0.03920355
##
## Log-likelihood: -276.15
```

Additionally, the user can decide to use the automatic differention engine using the line `method = "TMB"` to get the equivalent estimates.

Random Effects

The models and examples we have discussed so far have parameters that are referred as *fixed effects*, where the parameter estimate is assumed to be constant across all individual levels. For instance, the previous example assumes that the attack rate for each small tadpoles are the same. However, suppose those tadpoles are sampled from different tanks, instead of estimating the effects of each tanks, we are interested in the variability attributable to the variable "tank." If parameters has heterogeneity among groups and we are interested in estimating the group-level variation, then the parameters is referred as the *random effects* (Pinheiro and Bates 2000).

Consider observed data Y follows some distribution f with distributional parameters of θ with a vector of random effects b and design matrix Z for b, then θ can be denoted as:

$$\theta = \underbrace{h(w_1, ..., w_i)}_{\text{fixed effect}} + \underbrace{Z \cdot b}_{\text{random effect}}$$

and b is assumed to follow the multivariate normal distribution with zero mean (ie. no random effects), and covariance matrix G (Fitzmaurice, Laird, and Ware 2012):

$$b \sim \text{MVN}(0, G)$$

The likelihood function is conditioned on the unobserved random effect, b_i, and it can be obtained by integrating over the distribution of b_i:

$$\text{L}(Y_i|b_i) = \prod_{i=1}^{N} \int f(Y_i|b_i) f(b_i)\, db_i$$

Unlike previous cases, there is no analytic solution for this likelihood function and it can only be approximated by techniques such as Gaussian quadrature or Laplace approximation. Conveniently, the `TMB` package has implemented Laplace approximation for likelihood functions with random effects (Kristensen et al. 2016). Therefore, when modelling random effect using our package, the user will need to specify the argument `method = "TMB"`.

Example: Tadpole Predation at Random blocks

Similar to the previous scenario, we are still interested in modelling the effect of attack rate and handling time on the number of killed tadpoles. Suppose we sample the tadpoles from 20 different tanks (or blocks), and we wonder about how does the attack rate varies for each blocks?

Here is the same formula as last time, except applying logit link function on a to ensure the attack rate is within 0 and 1, and also applying log link function on h to make sure the handling time is strictly positive. Then, the random effect is defined in the `parameter` argument for a on logit scale, as shown below:

```
form <- killed ~ dbinom(size = density,
                        prob = plogis(logit_a)/(1 +plogis(logit_a)*exp(log_h)*density))

qzmle::mle(form,
           start=list(logit_a=c(0), log_h=0),
           links= list(a="logit", h="log"),
           parameters=list(logit_a ~ 1 + (1|block)),
           data=rfpsim, method = "TMB")
```

```
## Note: Using Makevars in /Users/quee/.R/Makevars
##
## Call:
## qzmle::mle(form = form, start = list(logit_a = c(0), log_h = 0),
##     data = rfpsim, links = list(a = "logit", h = "log"), parameters = list(logit_a ~
##         1 + (1 | block)), method = "TMB")
##
## Coefficients:
##  logit_a_param          log_h  logit_a_rand1  logit_a_rand2  logit_a_rand3
##     0.01616502    -1.10313657    -0.02453381     0.02406944    -0.02017801
##  logit_a_rand4  logit_a_rand5  logit_a_rand6  logit_a_rand7  logit_a_rand8
##    -0.01181954    -0.02064827    -0.01465469     0.09710400    -0.03413576
##  logit_a_rand9 logit_a_rand10 logit_a_rand11 logit_a_rand12 logit_a_rand13
##    -0.01594540    -0.02862781    -0.02459762    -0.01437218    -0.01898804
## logit_a_rand14 logit_a_rand15 logit_a_rand16 logit_a_rand17 logit_a_rand18
##    -0.02898147    -0.02958723    -0.02181357    -0.01798696    -0.02283834
## logit_a_rand19 logit_a_rand20  logit_a_logsd
##    -0.01414925    -0.01714184     4.08394799
##
## Log-likelihood: -551.99
```

The output provides the estimates at link function scaled for fixed intercept for attack rate, the handling time, the random intercepts at each block, and the variance of the random effects.

3 Further Steps

This package is still being actively developed, hence there are still some issues to be resolved and new features to be implemented in the distant future. Here are some examples of improvements that are on top of the agenda:

- `predict` method for the fitted objects
- likelihood profiling and confidence interval
- More extensive modelling for random effects

4 Conclusion

The rapid advancement of computational algorithms has offered the power to solve complex real-world problems faster and better. The development of this R package has leveraged the powerful tools of symbolic and automatic differentiation, to improve the maximum likelihood estimation framework with greater flexibility, such as specifying non-linear parameters and random effects. The goal of this project is to provide a robust implementation for scientists and statisticians to easily construct more sophisticated models and solve more difficult problems without the demanding efforts of writing code from scratch.

Reference

Bell, BM. 2005. "CppAD: A Package for c++ Algorithmic Differentiation." 2005. http: //www.coin-or.org/CppAD.

Bolker, Benjamin M. 2008. *Ecological Models and Data in r*. Princeton University Press. https://doi.org/10.2307/j.ctvcm4g37.

Dai, Yu-Hong. 2002. "Convergence Properties of the BFGS Algoritm." *SIAM Journal on Optimization* 13 (3): 693–701. https://doi.org/10.1137/S1052623401383455.

Fitzmaurice, Garrett M., Nan M. Laird, and James H. Ware. 2012. *Applied Longitudinal Analysis*. John Wiley & Sons.

Griewank, Andreas, and Andrea Walther. 2003. "Introduction to Automatic Differentiation." *PAMM* 2 (1): 45–49. https://doi.org/https://doi.org/10.1002/pamm.200310012.

Hogan, Robin J. 2014. "Fast Reverse-Mode Automatic Differentiation Using Expression Templates in c++." *ACM Transactions on Mathematical Software* 40 (4): 26:1–16. https://doi.org/10.1145/2560359.

Kristensen, Kasper, Anders Nielsen, Casper W. Berg, Hans Skaug, and Brad Bell. 2016. "TMB: Automatic Differentiation and Laplace Approximation." *Journal of Statistical Software* 70 (5). https://doi.org/10.18637/jss.v070.i05.

Pinheiro, José, and Douglas Bates. 2000. *Mixed-Effects Models in s and s-PLUS*. Statistics and Computing. New York: Springer-Verlag. https://doi.org/10.1007/b98882.

Shanno, D. F. 1970. "Conditioning of Quasi-Newton Methods for Function Minimization." *Mathematics of Computation* 24 (111): 647–56. https://doi.org/10.2307/2004840.

Stepleman, R. S., and N. D. Winarsky. 1979. "Adaptive Numerical Differentiation." *Mathematics of Computation* 33 (148): 1257–64. https://doi.org/10.1090/S0025-5718-1979-0537969-8.

Vonesh, James R. 2005. "Egg Predation and Predator-Induced Hatching Plasticity in the African Reed Frog, Hyperolius Spinigularis." *Oikos* 110 (2): 241–52. https://doi.org/https://doi.org/10.1111/j.0030-1299.2005.13759.x.