Bernoulli candidate sets that are functions of the unkown parameters masked data

Alex Towell

1/22/2022

Contents

0.0.1 Choices for $p_1, \ldots, p_m$ that are functions of θ and other observables

In many cases, the failure rate increases as the lifetime of the system increases. Thus, a reasonable choice for p_j may be of the form

$$p_j(s) = 1 - \exp(-\beta_j s),$$

which is the cdf of an exponential distribution with rate parameter β_j. As $s \to \infty$, $p_j(s) \to 1$.

$$f(x_1, \ldots, x_m, s|\theta) = \frac{f(s|\theta)}{\sum_{j=1}^m h_j(s|\theta_j)} \sum_{k=1}^m \left\{ h_k(s|\theta_k) \prod_{j=1}^m (1 - \exp(-\beta_j s))^{x_j} \exp(-\beta_j s)^{1-x_j} \right\}. \tag{1}$$

A possibly more interesting choice is, say, $p_j(s|\theta_j) = F_j(s|\theta_j)$, the cdf of X_j, so that a random sample $X_1, \ldots, X_m$ stores more information about θ.

A potentially even more interesting choice is given by

$$p_j(s|\theta_j) = f_{K|S}(j|s, \theta).$$

0.0.1.0.1 Estimating p Under the maximum entropy model, p has a straightforward method of moments estimator given by

$$\hat{p} = \frac{\overline{|C|} - 1}{m - 1},$$

where $\overline{|C|} = \sum_{i=1}^n |C_i|/n$.

Proof.

$$\begin{aligned} E(|C|) &= E\left(\sum_{j=1}^m 1_{\{j=k\}} + 1_{\{j \neq k\}} X_j \right) \\ &= 1 + E\left(\sum_{\substack{j=1 \\ j \neq k}} X_j \right). \end{aligned}$$

Since $X_1, \ldots, X_m$ are independent, this simplifies to

$$\begin{aligned} E(|C|) &= 1 + \sum_{\substack{j=1 \\ j \neq k}}^{m} E(X_j) \\ &= 1 + \sum_{\substack{j=1 \\ j \neq k}}^{m} p \\ &= 1 + (m-1)p. \end{aligned}$$

We may estimate $E(|C|)$ with the sample average $\overline{|C|} = \sum_{i=1}^{n} |C_i|/n$ and we may estimate p with $\hat{p}$,

$$\overline{|C|} = 1 + (m-1)\hat{p}.$$

Solving for $\hat{p}$, we obtain the result

$$\hat{p} = \frac{\overline{|C|} - 1}{m-1}.$$

□

Under different models, where $p_1, \ldots, p_m$ are functions or not all the same constant value, the likelihood approach may be used to estimate them.

0.1 Hypothesis testing

0.1.1 Likelihood ratio test

$$\begin{aligned} H_0 &: \theta \in \Omega_0 \\ H_A &: \theta \notin \Omega_0 \end{aligned}$$

where $\hat{\theta}$ is the unconstrained MLE and $\hat{\theta}_R$ is in the constrained MLE (reduced model).

0.2 Confidence intervals of parameter values

Of course, you may also construct confidence intervals and, if the MLE over θ^F is contained in the confidence interval, we say that the data is compatible with the null model.

0.2.1 Score test

If we have the score function s and Fisher information matrix I, we may construct the hypothesis

$$\begin{aligned} H_0 &: \theta \in \Omega_0 \\ H_A &: \theta \notin \Omega_0 \end{aligned}$$

which can be tested with the Score test

$$S(\hat{\theta}_0) = s'(\hat{\theta}_0) I^{-1}(\hat{\theta}_0) s(\hat{\theta}_0).$$

where $\hat{\theta}_0$ is the MLE over the restricted parameter space Ω_0.

Under the null hypothesis, $S(\hat{\theta}_0)$ is asymptotically distributed χ^2 with k degrees of freedom, where k is the number of constraints imposed by the null hypothesis.

For instance, suppose we wish to test the hypothesis in a exponential series system, the failure rate of component 1 is twice that of component 2. Then, we find the MLE over the reduced model

$$\theta^R = (2\lambda_2, \lambda_2, \lambda_3)$$

and compare that to the MLE over the full model

$$\theta^F = (\lambda_1, \lambda_2, \lambda_3).$$

```
library(masked.data)
```

```
##
## Attaching package: 'masked.data'
```

```
## The following object is masked from 'package:grDevices':
##
##     pdf
```

```
loglik.F <- masked.data::md_kloglike_exp_series_m0(exp_series_data_1)
loglik.R <- function(theta.R)
{
  loglik.F(c(2*theta.R[2],theta.R[2],theta.R[3]))
}
#

# should i allow a parameter function to be specified? use the existing
# log-likelihood, but expose the interface.
```

0.3 Appendix

A general strategy for solving the maximum likelihood equation

$$\hat{\theta} = \text{argmax}_{\theta \in \Omega}\, \ell(\theta)$$

is given by the iterative search strategy that tries to maximize the log-likelihood by, when at at some point $\theta^{(i)}$, moving in a direction d_i,

$$\theta^{(i+1)} = \theta^{(i)} + \alpha_i d_i,$$

where α_i scales the magnitude of the i^{th} step size, such that $\ell(\theta^{(i+1)}) > \ell(\theta^{(i)})$.

Starting at $\theta^{(0)}$, it is hoped that the method produces a sequence of values $\theta^{(0)}, \theta^{(1)}, \ldots, \theta^{(n)}$ such that as $n \to \infty$,

$$\theta^{(n)} \to \text{argmax}_{\theta \in \Omega}\, \ell(\theta).$$

However, note that, as a local search method, there are several issues to consider. The method may:

1. converge to a local maximum,
2. converge to a saddle point,
3. try to step outside of the parameter's support set,
4. exhibit extremely slow convergence, or
5. fail to converge, e.g., maximum is at or near a boundary.

In non-linear optimization, there is no general off-the-shelf solution to avoiding these outcomes. However, a general mitigation strategy, potentially at the cost of convergence rate, is the introduction of randomness into the local search process. For instance, in stochastic gradient ascent, we do not (always) move in the direction of the gradient. We may random perturb the direction in some way from the gradient, in order to try to jump over, say, local maxima. Or, we may use random restarts, i.e., starting over with a randomly chosen starting point θ_0, and choosing the best solution found.

We capture the notion of convergence through a *stopping condition*, which is (normally) some heuristic that evaluates the sufficiency of the current iteration. We model stopping conditions as a Boolean function of the formal parameters of the maximum likelihood solver.

Stopping conditions are normally parameterized by a norm $\|\cdot\|$ and a ϵ such that if $\|\theta_1 - \theta_0\| < \epsilon$, the stopping condition has been met.

We model $\|\cdot\|_1$, $\|\cdot\|_2$, and $\|\cdot\|_\infty$ by the R functions respectively given by:

```
l1_norm <- function(theta0,theta1,...) sum((abs(theta0-theta1)))
l2_norm <- function(theta0,theta1,...) sqrt(sum((theta0-theta1)^2))
inf_norm <- function(theta0,theta1,...) max(abs(theta0-theta1))
```

We model the stopping condition $\|$ `args` $\|$ < `eps`, that takes a `norm`, such as one of the ones defined above, and an `eps`, by the R function given by:

```
eps_stop_cond <- function(eps=1e-3,norm=inf_norm)
  function(theta1,theta2,...) norm(theta1,theta2,...) < eps
```

```
stop_cond_max_iter <- function(stop_cond,max_iter=1000,...)
  function(iter) ifelse (iter > max_iter, TRUE, stop_cond(...))

eps_stop_cond <- function(eps=1e-3,norm=inf_norm)
  function(theta1,theta2,...) norm(theta1,theta2) < eps
```

In R, we implement this algorithm as a function that has the formal parameters given by Table ?.

Parameter	Description
`theta0`	Initial starting point.
`loglike`	A function that models the log-likelihood function. Note that if the formal parameters of the stopping condition are not a function of the log-likelihood, this may be set to `NULL`.
`direction`	A function that models the direction function.
`stop_cond`	Stopping condition, a function that maps its arguments to a `Boolean`. If `stop_cond` is `NULL`, we set it to $\|\theta_1 - \theta_0\|_\infty < 1.0e^{-5}$.
`ls`	Line search function. If `ls` is `NULL`, we set it to output the constant 1. An example of `ls` Golden section search.

```
mle_solver <- function(theta0,
                       direction,
                       stop_cond = stop_cond_max_iter(eps_stop_cond(),...),
                       loglike = NULL,
                       ls = NULL,
                       ...)
{
  if (is.null(stop_cond)
    stop_cond <- eps_stop_cond(1e-5,...)
  if (is.null(ls))
    ls <- function(...) 1

  iter <- 1L
  theta1 <- NULL
  repeat
  {
    d <- direction(theta0,...)
    alpha <- ls(d,theta0,loglike,...)
    theta1 <- theta0 + alpha * d
    if (stop_cond(theta1,theta0,loglike,score,info,alpha,iter,...))
      break
    theta0 <- theta1
```

```
    iter <- iter + 1L
  }

  list(theta.hat=theta1,iter=iter)
}
```

We consider two well-known search strategies, gradient ascent and Newton-Raphson. In gradient ascent, the direction is given by the gradient. In particular, to find the maximum, we find the point at which the gradient of ℓ is zero by solving the first-order approximation $f(\theta) = 0$ where $f(\theta) = \nabla\ell(\theta_0)$

In Newton-Raphson, the direction is given by the If the Newton-Raphson search uses the inverse of the *expected* Fisher information matrix, then this is known as *Fisher scoring*.

0.3.1 Gradient ascent

Gradient ascent is a local iterative method for finding a value that maximizes the log-likelihood by finding a point where the gradient is approximately 0.

The score function s is defined as $s(\theta) = \nabla\ell\big|_\theta$. Starting at some point $\theta^{(i)}$, the method searches in the direction of the gradient of the log-likelihood,

$$\theta^{(i+1)} = \theta^{(i)} + \alpha_i s(\theta^{(i)}),$$

where α_i scales the magnitude of the i^{th} step size such that $\ell(\theta^{(i+1)}) > \ell(\theta^{(i)})$.

In R, we implement this algorithm as a function that has the formal parameters given by Table.

Parameter	Description
`loglike`	A function that models the log-likelihood function. Note that depending upon the other arguments to the formal parameters, this may not be needed.
`score`	A function that models the score function for the likelihood function. If `score` is `NULL`, we numerically approximate it from `loglike`.
`stop_cond`	Stopping condition, a function that maps its arguments to a `Boolean`. If `stop_cond` is `NULL`, we set it to $\|\theta_1 - \theta_0\|_\infty < 1.0e^{-5}$.
`max_iter`	Specifies the maximum number of iterations.

```
gradient_ascent <- function(theta0,
                            loglike = NULL,
                            score = NULL,
                            stop_cond = NULL,
                            max_iter = NULL)
{
  if (is.null(score))
    score <- function(theta) { numDeriv::grad(loglike,theta) }

  function(theta,d)
  {
    d <- score(theta)
    function(alpha) loglike(theta - alpha * d)
  }

  mle_solver(theta0,loglike,score,stop_cond,ls,max_iter)
}
```

0.3.2 Fisher scoring algorithm

Fisher scoring, like gradient ascent, is a local iterative method except it uses both the Hessian and the gradient to choose a direction that tends to converge faster to a solution.

Starting at some point $\theta^{(i)}$, it finds a new point that is in the direction of the gradient of the log-likelihood,

$$\theta^{(i+1)} = \theta^{(i)} + \alpha_i H^{-1}\left(\theta^{(i)}\right) s(\theta^{(i)}).$$

where α_i scales the magnitude of the i^{th} step size such that $\ell(\theta^{(i+1)}) > \ell(\theta^{(i)})$.

In R, we implement this algorithm as a function that has the formal parameters given by Table.

Parameter	Description
`loglike`	A function that models the log-likelihood function. Note that depending upon the other arguments to the formal parameters, this may not be needed.
`info`	A function that models the Fisher information matrix for the likelihood function. If `info` is `NULL`, we numerically approximate it from `loglike` or `score`.
`score`	A function that models the score function for the likelihood function. If `score` is `NULL`, we numerically approximate it from `loglike`.
`stop_cond`	Stopping condition, a function that maps its arguments to a `Boolean`. If `stop_cond` is `NULL`, we set it to $\|\theta_1 - \theta_0\|_\infty < 1.0e^{-5}$.
`ls`	Line search function. If `ls` is `NULL`, we set it to output the constant 1. A good choice for linear search is, say, Golden section search.
`max_iter`	Specifies the maximum number of iterations.

```
fisher_scoring <- function(theta0,
                           loglike = NULL,
                           info = NULL,
                           score = NULL,
                           stop_cond = NULL,
                           max_iter = NULL)
{
  if (is.null(score))
    score <- function(theta) { numDeriv::grad(loglike,theta) }
  if (is.null(info))
    info <- ifelse(is.null(score),
                   function(theta) { -numDeriv::hessian(loglike,theta) },
                   function(theta) { -numDeriv::jacobian(score,theta) }

  d <- function(theta) { matlib::inv(info(theta)) %*% score(theta) }
  mle_solver(theta0,loglike,d,stop_cond,ls,max_iter)
}
```