SDS 385: Homework 3

G. Paulon

September 14, 2016

Problem 1. Line search

(A) Let us recall that, in both the gradient descent and in Newton method, the update of the parameters has the following form:

$$\beta^{(k+1)} = \beta^{(k)} + \alpha^{(k)} s^{(k)},$$

where $\alpha^{(k)}$ is a step size and $s^{(k)}$ is a given direction. In general, the direction has the general expression

$$s^{(k)} = -B_k^{-1} \nabla f_k,$$

that is, the direction is chosen to be opposite to the gradient vector, with a correction introduced by the matrix B_k^{-1}. One can prove that, if B_k is a positive definite matrix, then the direction $s^{(k)}$ is a descent direction. Moreover, in the case of the gradient descent $B_k = \mathcal{I}$, whereas in the Newton method $B_k = H_k$, where H_k is the Hessian matrix of the objective function evaluated with the parameters at iteration k.

Given a choice of the descent direction, the goal is now to optimize the choice of the step size $\alpha^{(k)}$. Ideally, give a direction $s^{(k)}$ one would like to minimize the function

$$\phi(\alpha) = f(\beta^{(k)} + \alpha s^{(k)}).$$

However, solving a minimization problem at each iteration of the algorithm is way too computationally intense. A simpler approach consists in trying several values for the step size, discarding those who do not satisfy appropriate conditions that ensure convergence. In fact, a simple decrease in the objective function, i.e. $f(\beta^{(k)} + \alpha^{(k)} s^{(k)}) < f(\beta^{(k)})$, is not enough for the whole algorithm to converge. We require a sufficient decrease condition, called the *first Wolfe condition*:

$$f(\beta^{(k)} + \alpha s^{(k)}) \leq f(\beta^{(k)}) + c_1 \alpha \nabla f_k^T s^{(k)}, \qquad c_1 \in (0, 1). \tag{1}$$

This expression means that f should decrease at least linearly, with a slope proportional to both α and the directional derivative $\nabla f_k^T s^{(k)}$. In practice, this condition ensures that the step size is small enough to make the algorithm converge. However, as we discussed in the previous chapters, small step sizes often cause slow performances. Since the first Wolfe condition is always satisfied for small values of α, it is mandatory to introduce a *second Wolfe condition* which has the role to penalize small step sizes, i.e.

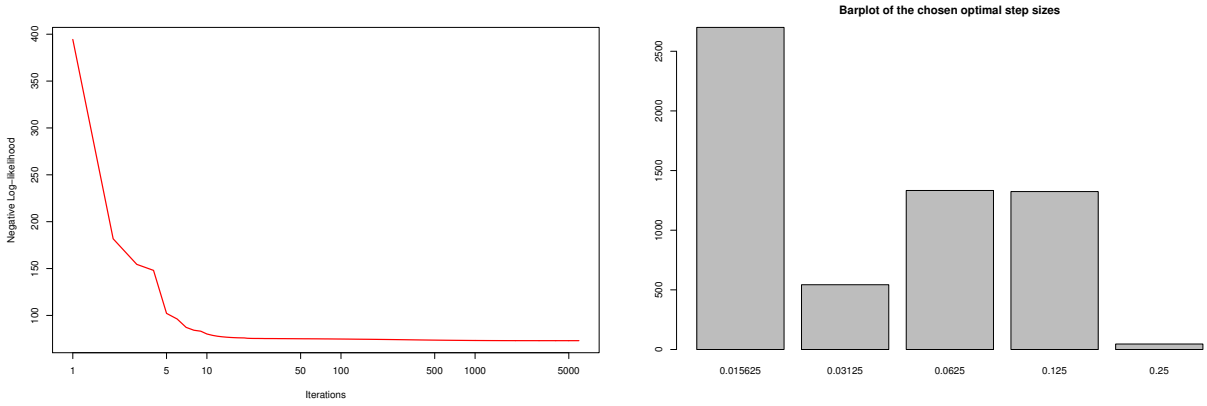
$$\nabla f(\beta^{(k)} + \alpha s^{(k)}) \geq c_2 \nabla f_k^T s^{(k)}, \qquad c_2 \in (c_1, 1). \tag{2}$$

A well known procedure that implements line search is known as *backtracking algorithm*. This algorithm allows us to get rid of (2) by implementing only the condition (1) and by choosing the candidates α in a clever way. In practice, we start from a maximum value of the step size $\alpha = \bar{\alpha}$ and we proceed backwards by shrinking it by a factor ρ until condition (1) is met (α will eventually become small enough that (1) holds). In Algorithm 1 the pseudocode for the algorithm is reported.

Algorithm 1 Backtracking line search algorithm.

function BACKTRACKING(β, s, y, X, m, $c = 0.01$, $\bar{\alpha} = 3$, $\rho = 0.5$)
  Set $\alpha = \bar{\alpha}$;
  do
    Shrink the step size: $\alpha \leftarrow \rho\alpha$;
  while $l(\beta + \alpha s) > l(\beta) + c\alpha\nabla l(\beta)^T s$
  return α;

(B) The gradient descent method coupled with backtracking has been implemented and tested on the same dataset used in the previous chapters. In Figure 1, the values of the negative log-likelihood are displayed and along with the barplot of the optimal values of the step size. The gradient descent converges, in this case, after around 5947 iterations, which represents a great improvement from the algorithm with fixed step size, which converged in around 60000 iterations.



(a) Negative log-likelihood of the data. *(b) Barplot of the optimal size steps α.*

Figure 1: Results of the gradient descent coupled with backtracking.

Problem 2. Quasi Newton

(A) Quasi-Newton methods are an alternative to Newton's method in which the Hessian $\nabla^2 f_k$ is approximated by B_k, a matrix which is updated at every step. In particular, since the Hessian is the "derivative of the first-order derivative", one can use the finite approximation of the derivative, i.e.

$$\nabla^2 f_k(\beta^{(k+1)} - \beta^{(k)}) \approx \nabla f_{k+1} - \nabla f_k.$$

The *secant condition* is just the equation above applied to the matrix B_k that should mimic the behaviour of the Hessian. Therefore

$$B_{k+1} s_k = y_k,$$

where $s_k = \beta^{(k+1)} - \beta^{(k)}$ and $y_k = \nabla f_{k+1} - \nabla f_k$. Moreover, we typically impose also that the approximation of the Hessian matrix should be symmetric. The two conditions coupled together yield to a recursive formula to compute B_k, that is

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}.$$

A practical implementation of BFGS consists in finding an iterative formula to compute directly the inverse of the Hessian, which is

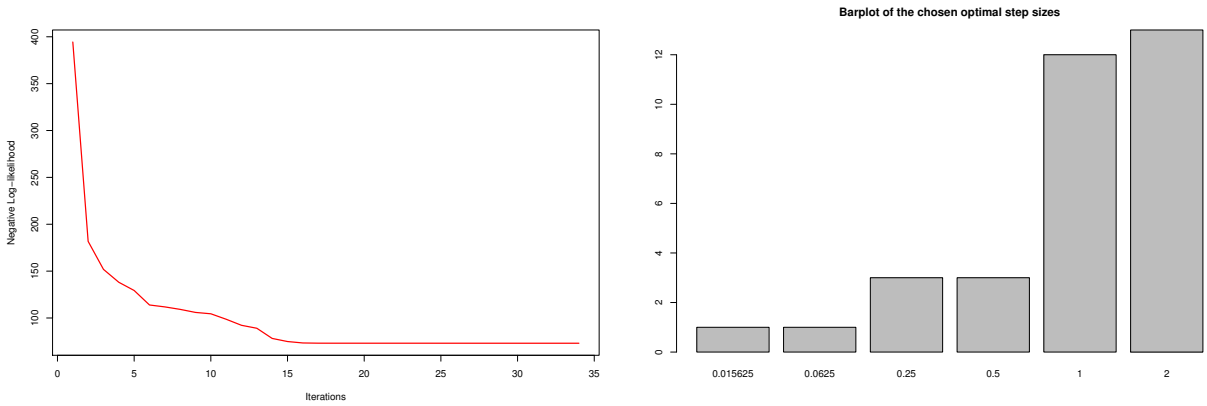
$$B_{k+1}^{-1} = (\mathcal{I} - \rho_k s_k y_k^T) B_k^{-1} (\mathcal{I} - \rho_k y_k s_k^T) + \rho_k s_k s_k^T,$$

where $\rho_k = \frac{1}{y_k^T s_k}$. Then, the descent direction $s^{(k)}$ is simply computed by $s^{(k)} = -B_k^{-1} \nabla f_k$. In Algorithm 2 the pseudocode for the quasi-Newton method coupled with backtracking is reported.

Algorithm 2 BFGS coupled with backtracking line search algorithm.

function BFGS(y, X, m, β_0, T)
  Initialize the data structures that contain $\beta^{(t)}, l(\beta)^{(t)}, \nabla l(\beta)^{(t)}$;
  Initialize the Hessian matrix $B^{-1} = \mathcal{I}$;
  for t in $2 : T$ **do**
    Store the previous gradient: $gradient_{old} \leftarrow gradient_{new}$;
    Find the descent direction: $s^{(t-1)} \leftarrow B^{-1} * gradient_{old}$;
    Compute the step size: $\alpha^{(t-1)} \leftarrow$ BACKTRACKING$(\beta^{(t-1)}, s^{(t-1)}, y, X, m)$;
    Update the parameters: $\beta^{(t)} \leftarrow \beta^{(t-1)} + \alpha^{(t-1)} * s^{(t-1)}$;
    Update the gradient of the log-likelihood: $gradient_{new} \leftarrow$ GRAD.LOGLIK$(\beta^{(t)}, y, X, m)$;
    Compute the quantities: $s_t \leftarrow \alpha^{(t-1)} s^{(t-1)}$; $y_t \leftarrow gradient_{new} - gradient_{old}$; $\rho_t \leftarrow \frac{1}{y_t^T s_t}$;
    Update the inverse of the Hessian matrix: $B^{-1} = (\mathcal{I} - \rho_t s_t y_t^T) B^{-1} (\mathcal{I} - \rho_t y_t s_t^T) + \rho_t s_t s_t^T$;
  return $\beta^{(T)}$;

(B) The BFGS method coupled with backtracking has been implemented and tested on the same dataset used in the previous chapters. In Figure 2, the values of the negative log-likelihood are displayed and along with the barplot of the optimal values of the step size. The gradient descent converges, in this case, after around 34 iterations, which can seem a worse performance than the one obtained with the algorithm with fixed step size, which converged in around 11 iterations. However, in the BFGS version we avoid to compute the Hessian matrix at each iteration, which in high dimensional problems can be cumbersome. The Hessian approximation reduces the per-iteration computational time and it makes the problem more stable (sometimes the Hessian cannot be computed analytically or it could not exist).



(a) Negative log-likelihood of the data. *(b) Barplot of the optimal size steps α.*

Figure 2: Results of the BFGS method coupled with backtracking.

Appendix A

R code

```r
# Compute the probabilities associated with the logit model
comp.wi <- function(X, beta){
  wi <- 1 / (1 + exp(-X %*% beta))
  return(wi)
}

# Compute the negative log-likelihood of the logit model
log.lik <- function(beta, y, X, mi){
  ll <- array(NA, dim = N)
  wi <- comp.wi(X, beta)
  ll <- -sum(y * log(wi + 1E-10) + (mi - y) * log(1 - wi + 1E-10))
  return(ll)
}

# Compute the gradient of the negative log-likelihood of the logit model
grad.loglik <- function(beta, y, X, mi){
  grad.ll <- array(NA, dim = length(beta))
  wi <- comp.wi(X, beta)
  grad.ll <- -apply(X * as.numeric(y - mi * wi), 2, sum)
  return(grad.ll)
}

# Compute the hessian of the negative log-likelihood of the logit model
hessian.loglik <- function(beta, y, X, mi){
  wi <- comp.wi(X, beta)
  W.12 <- sqrt(mi * wi * (1 - wi))
  hes.ll <- crossprod(as.numeric(W.12) * X)
  return(hes.ll)
}
```

Listing A.1: General functions implementing the computation of log-likelihood, its gradient and its Hessian.

```r
# Compute the line search
optimal.step <- function(param.act, direct, ll.act, grad.ll.act, y, X, mi, c=0.01, max.alpha=2,
    rho=0.5){
  opt.alpha <- max.alpha
  while(log.lik(param.act + opt.alpha * direct, y, X, mi) > ll.act + c * opt.alpha * crossprod(
      grad.ll.act, direct)){
    opt.alpha <- opt.alpha * rho
  }
```

```
  return(opt.alpha)
}
```

Listing A.2: Function implementing the backtracking algorithm for the optimal step size.

```
# Function for the gradient descent of the logit model coupled with backtracking
GD.line.search <- function(y, X, mi, beta0, maxiter, tol){
  betas <- array(NA, dim=c(maxiter, length(beta0)))
  betas[1, ] <- beta0 # Initial guess

  ll <- array(NA, dim = maxiter)
  ll[1] <- log.lik(betas[1, ], y, X, mi)

  alpha <- array(NA, dim = maxiter)

  for (iter in 2:maxiter){
    gradient <- grad.loglik(betas[iter-1, ], y, X, mi)
    direct <- -gradient
    alpha[iter-1] <- optimal.step(betas[iter-1, ], direct, ll[iter-1], gradient, y, X, mi)
    betas[iter, ] <- betas[iter-1, ] + alpha[iter-1] * direct
    ll[iter] <- log.lik(betas[iter, ], y, X, mi)

    # Convergence check
    if (abs(ll[iter-1] - ll[iter]) / abs(ll[iter-1] + 1E-3) < tol){
      cat('Algorithm has converged after', iter, 'iterations')
      ll <- ll[1:iter]
      betas <- betas[1:iter, ]
      break;
    }
    else if (iter == maxiter & abs(ll[iter-1] - ll[iter]) / abs(ll[iter-1] + 1E-3) >= tol){
      print('WARNING: algorithm has not converged')
      break;
    }
  }
  return(list("ll" = ll, "beta" = betas[iter, ], "alpha" = alpha[1:(iter-1)]))
}
```

Listing A.3: Function implementing the gradient descent using the backtrack algorithm.

```
# Function for the quasi Newton method of the logit model coupled with backtracking
quasi.newton <- function(y, X, mi, beta0, maxiter, tol){
  p <- length(beta0)

  betas <- array(NA, dim=c(maxiter, p))
  betas[1, ] <- beta0 # Initial guess

  ll <- array(NA, dim = maxiter)
  ll[1] <- log.lik(betas[1, ], y, X, mi)
  gradient_new <- grad.loglik(betas[1, ], y, X, mi)

  alpha <- array(NA, dim = maxiter)

  # We initialize the inverse of the Hessian to the identity matrix
  id.matrix <- diag(rep(1, p))
  Hk <- id.matrix

  for (iter in 2:maxiter){
```

```
    # The previous gradient is now the old one
    gradient <- gradient_new

    # We find the descent direction and we compute the step size
    direct <- - Hk %*% gradient
    alpha[iter-1] <- optimal.step(betas[iter-1, ], direct, ll[iter-1], gradient, y, X, mi)

    # We update the values of the parameters, of the log-likelihood and of the gradient
    sk <- alpha[iter-1] * direct
    betas[iter,] <- betas[iter-1,] + sk
    ll[iter] <- log.lik(betas[iter,], y, X, mi)
    gradient_new <- grad.loglik(betas[iter, ], y, X, mi)

    # We update the inverse of the Hessian matrix
    yk <- gradient_new - gradient
    rhok <- as.numeric(1 / crossprod(yk, sk))
    acc <- tcrossprod(sk, yk)
    Hk <- (id.matrix - rhok * acc) %*% Hk %*% (id.matrix - rhok * t(acc)) + rhok * sk %*% t(sk)

    # Convergence check
    if (abs(ll[iter-1] - ll[iter]) / abs(ll[iter-1] + 1E-3) < tol){
      cat('Algorithm has converged after', iter, 'iterations')
      ll <- ll[1:iter]
      betas <- betas[1:iter, ]
      break;
    }
    else if (iter == maxiter & abs(ll[iter-1] - ll[iter]) / abs(ll[iter-1] + 1E-3) >= tol){
      print('WARNING: algorithm has not converged')
      break;
    }
  }
  return(list("ll" = ll, "beta" = betas[iter, ], "alpha" = alpha[1:(iter-1)]))
}
```

Listing A.4: Function implementing the quasi-Newton method (BFGS) coupled with a backtracking algorithm.