Better Online Learning (preliminaries)

Su Chen

September 18, 2016

Line search

Part A

- The first Wolfe condition makes sure the step length will lead to some decrease in the target function, in particular the amount of decrease should be proportional to the step length and a tuning parameter c1. However, as long as the given direction is a descending one, we can always pick a step size small enough so that this condition is satisfied for any chosen c1. This is why we need the second Wolfe condition to improve the performance.
- The second condition is against a step size that is too small, meaning it is not big enough to capture enough decrease of the target function in the given descending direction. So we want the rate of decrease of the target function at the end of the step is not as large as the current point. Intuitively, we want a step size to be large enough to jump over the very steep decrease in the target function which makes the algorithm more efficient.
- pseudo-code:

```
alpha = alpha_initial
repeat
if ( !(Wolfe_condition_1) )
    alpha = alpha_smaller
if ( !(Wolfe_condition_2) )
    alpha = alpha_larger
end
```

```
Backtracking = function(f, df, t, p, alpha0, rho, c, ...)
{
  Step = alpha0
  while( f(t+Step*p, ...) > f(t, ...) + c*Step*sum(df*p) )
  {
    Step = Step*rho
  }
  return (Step)
}
```

Part B Gradient descent with backtracking line search

```
Gradient_Desc_backtracking = function(x, y, m, alpha0, rho, c, iter_max)
{
  N = nrow(x)
  P = ncol(x)

  #initialize all the variables
```

```
  beta = matrix(0, P, iter_max)
  nll_ini = Neg_ll(beta[,1], x, y,m)
  nll = rep(nll_ini, iter_max)
  iter = 1

  while (iter < iter_max) #&&  (delta > precision)
  {
    g = Gradient_Cal(beta[,iter], x, y, m)
    desc_direction = -g / sqrt(sum(g^2)) #use the negative gradient as descend direction
    step = Backtracking(f=Neg_ll, df=g, t=beta[,iter], p=desc_direction, alpha0, rho, c, x, y, m)
    beta[,iter+1] = beta[,iter] + step*desc_direction #gradient decsent
    nll[iter+1] = Neg_ll(beta[,iter+1], x, y,  m) #calculate neg log likelihood
    iter = iter + 1 #keep track of iteration
  }
  return (list(iter, beta, nll))
}
```

Quasi_Newton Method

Part A

- The secant condition is for approximating the Hessian matrix using $B_{k+1} = \nabla f_{k+1} - \nabla f_k / x_{k+1} - x_k$, because Hessian is the second derivative, so we can just approximate it by the change in the first derivative (gradient) divided by change in x.
- We want to update the inverse of the approximate Hessian rather than the approximate Hessian itself, so we can save the step to inverse the matrix. But the more important reason is we know matrix inversion is numerically unstable. So the inverse of the approximated Hessian might be a very bad approximation even though we had a good approximated Hessian.
- pseudo-code:

```
beta = beta_initial
iter = 0
H = H_initial
while (iter < iter_max)
{
  desc_direction = -H %*% Gradient(beta)
  step = backtracking(Neg_ll, desc_direction, ...)
  beta_update = beta + step*desc_direction
  H_update = BFGS_formula(beta, beta_update, Gradient(beta), Gradient(beta_update))
  beta = beta_update
  H = H_update
}
```

Part B Quasi Newton's Method with backtracking line search

```
Quasi_Newton_backtracking = function(x, y, m, alpha0, rho, c, iter_max)
{
  N = nrow(x)
  P = ncol(x)
  #initialize all the variables
```

```
  beta = matrix(0, P, iter_max)
  nll_ini = Neg_ll(beta[,1], x, y,m)
  nll = rep(nll_ini, iter_max)
  I = diag(P) #P by P identity matrix
  H = I #initial value for H
  iter = 1
  while (iter < iter_max) #&& (delta > precision)
  {

    g = Gradient_Cal(beta[,iter], x, y, m)
    desc_direction = -H %*% g
    desc_direction = desc_direction / sqrt(sum(desc_direction^2)) #normalize vector length
    step = Backtracking(f=Neg_ll, df=g, t=beta[,iter], p=desc_direction, alpha0, rho, c, x, y, m)
    beta[,iter+1] = beta[,iter] + step*desc_direction
    ### BFGS formula update
    sk = beta[,iter+1] - beta[,iter]
    yk = Gradient_Cal(beta[,iter+1], x, y, m) - g
    rhok = 1 / (sum(yk*sk))
    H_update = (I - rhok * sk %*% t(yk)) %*% H %*% (I - rhok * yk %*% t(sk)) + rhok * sk %*% t(sk)
    H = H_update
    ###
    nll[iter+1] = Neg_ll(beta[,iter+1], x, y, m) #calculate neg log likelihood
    iter = iter + 1 #keep track of iteration
  }
  return (list(iter, beta, nll))
}
```

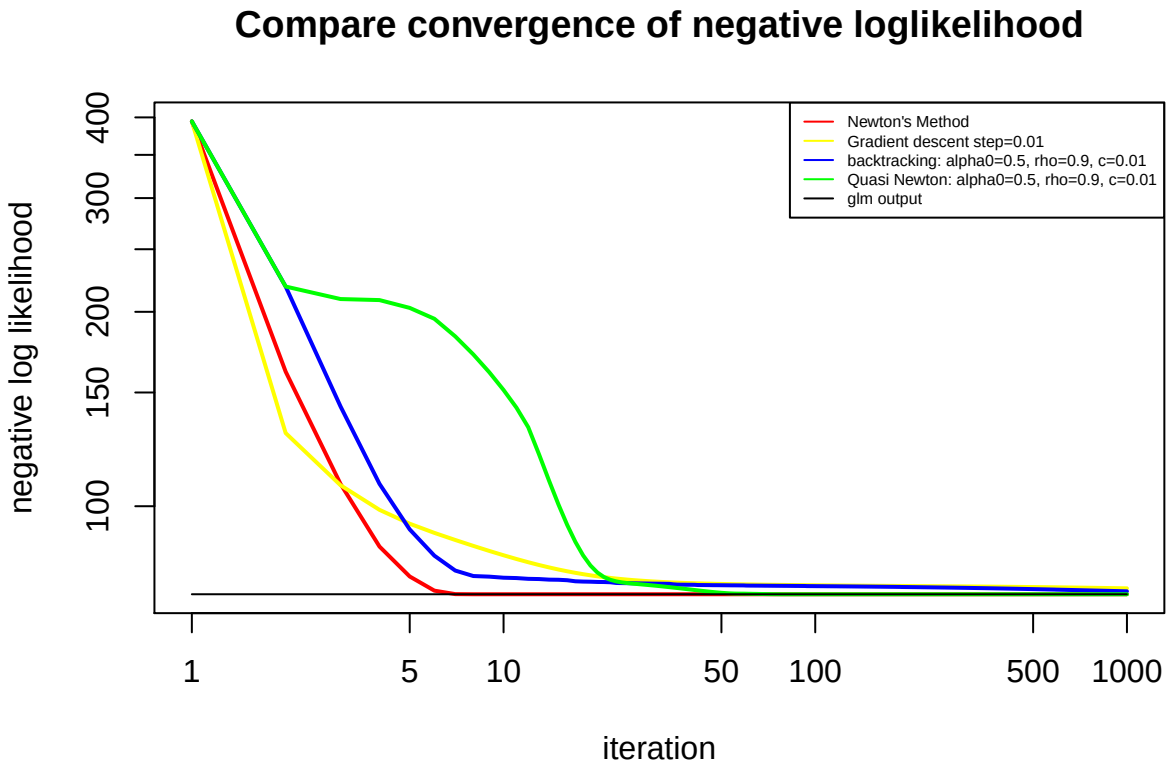
```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

run and compare results using real data

```
Iter_Max = 1000

result_nm = Newton_Method(x=X, y=Y, m=M, iter_max=Iter_Max)

result_gd = Gradient_Desc(x=X, y=Y, m=M, step=0.01, iter_max=Iter_Max)

result_gd_backtracking = Gradient_Desc_backtracking(x=X, y=Y, m=M, alpha0=0.5, rho=0.9, c=0.01, iter_ma

result_qnm_backtracking = Quasi_Newton_backtracking(x=X, y=Y, m=M, alpha0=0.5, rho=0.9, c=0.01, iter_ma
```

plot negtive likelihood to compare convergence for constant step and backtracking

```
plot_x = 1:Iter_Max

plot(x = plot_x, y = result_nm[[3]], type = "l", xlab = "iteration", ylab = "negative log likelihood",
     log = "xy", col = "red", lwd = 2, main = "Compare convergence of negative loglikelihood")
lines(x = plot_x, y = result_gd[[3]], col = "yellow", lwd = 2)
lines(x = plot_x, y = result_gd_backtracking[[3]], col = "blue", lwd = 2)
lines(x = plot_x, y = result_qnm_backtracking[[3]], col = "green", lwd = 2)
lines(x = c(1,Iter_Max), y = rep(nll_glm, 2), col = "black")
legend("topright", cex = .5, c("Newton's Method", "Gradient descent step=0.01", "backtracking: alpha0=0
       col = c("red","yellow","blue","green", "black"), lty = c(1,1,1,1))
```


Compare convergence of negative loglikelihood
negative log likelihood
100
150
200
300
400
1
5
10
50
100
500
1000
iteration
Newton's Method
Gradient descent step=0.01
backtracking: alpha0=0.5, rho=0.9, c=0.01
Quasi Newton: alpha0=0.5, rho=0.9, c=0.01
glm output