gradient descent

Kebing Liu

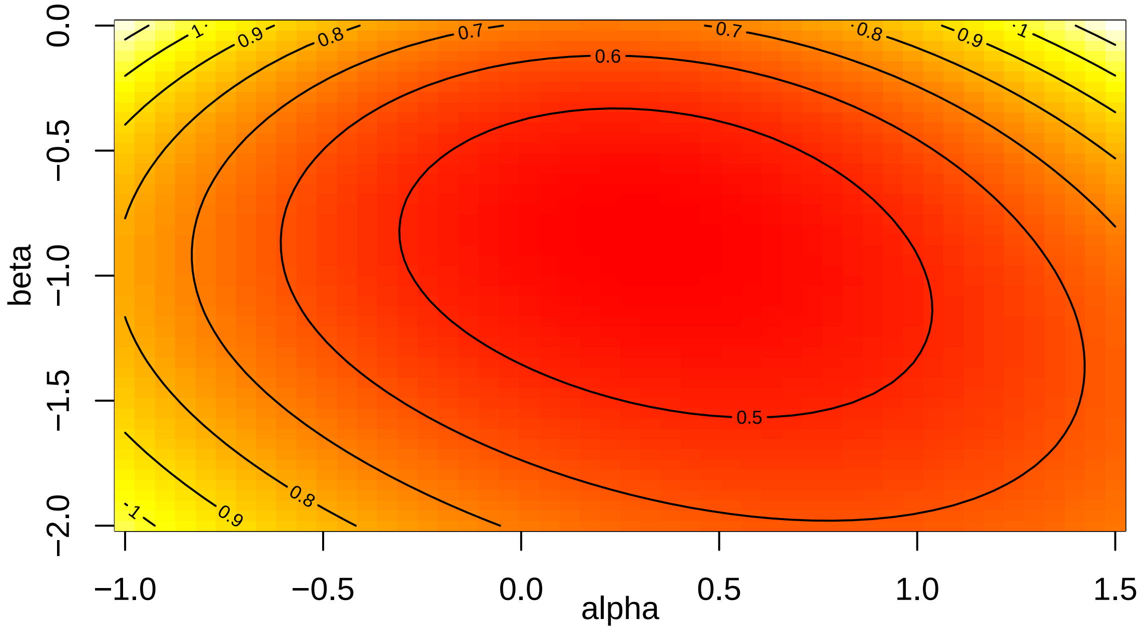
June 9

```
levels <- read.csv(file = "SMM_levels.csv", header = TRUE)

levels$log_tries_taken <- log(levels$tries_taken)
levels$Easy1 <- rep(0, length(levels$difficulty))
levels$Easy1[which(levels$difficulty == "Easy")] <- 1
```

Generate a contour plot

```
objective.fn <- function(alpha, beta) {
  # local variables
  x <- levels$log_tries_taken
  y <- levels$Easy1
  xbar <- mean(x)
  N <- length(x)

  # function on non-vector values
  non.vec <- function(a, b) {
    pi <- pnorm(a + b * (x - xbar))
    - 1/N * sum(y * log(pi / (1 - pi)) + log(1 - pi))
  }

  # apply non.vec
  n <- length(alpha)

  result <- rep(0, n)
  for (i in 1:n) {
    result[i] <- non.vec(alpha[i], beta[i])
  }
  return (result)
}

# generate alpha, beta
a <- seq(-1, 1.5, length.out = 50)
b <- seq(-2, 0, length.out = 50)

# cite from tutorial
# This is the key computation; there are many ways to achieve this.
z <- outer(a, b, objective.fn)   # Outer takes advantage of the fact that
                                 # objective.fn is vectorized simultaneously
image(a, b, z,                   # Matrix as heatmap.
      col = heat.colors(100),    # Palette with 100 levels; visually continuous.
      useRaster = TRUE,          # Less accurate image, but faster.
      ann = FALSE)
```

```r
mtext(text = "alpha", side = 1, line = 1.5) # Suppress default axis labels, and
mtext(text = "beta", side = 2, line = 2)  # draw them closer to the axis.
contour(a, b, z, add = TRUE)   # Add contours
```




```r
# modified from tutotorial
gradientDescent <- function(theta = 0, rhoFn, gradientFn, lineSearchFn,
    testConvergenceFn, maxIterations = 100, tolerance = 1e-06, relative = FALSE,
    lambdaStepsize = 0, lambdaMax = 0.5) {
    # Pre-allocate a matrix to track theta at each iteration.
    SolutionPath = matrix(NA, nrow = maxIterations + 1, ncol = length(theta))
    SolutionPath[1, ] = theta
    converged <- FALSE
    i <- 0

    while (!converged & i <= maxIterations) {
        # use unormalized gradient
        g <- gradientFn(theta)

        lambda <- lineSearchFn(theta, rhoFn, g, lambdaStepsize = lambdaStepsize,
            lambdaMax = lambdaMax)
        thetaNew <- theta - lambda * g
        converged <- testConvergenceFn(thetaNew, theta, tolerance = tolerance,
            relative = relative)
        theta <- thetaNew
```

```r
        # track the sequnce of updates
        i <- i + 1
        SolutionPath[i + 1, ] = theta
    }
    ## Return last value and whether converged or not
    return(list(theta = theta, converged = converged, iteration = i, fnValue = rhoFn(theta),
        SolutionPath = SolutionPath[1:i, ]))
}
```

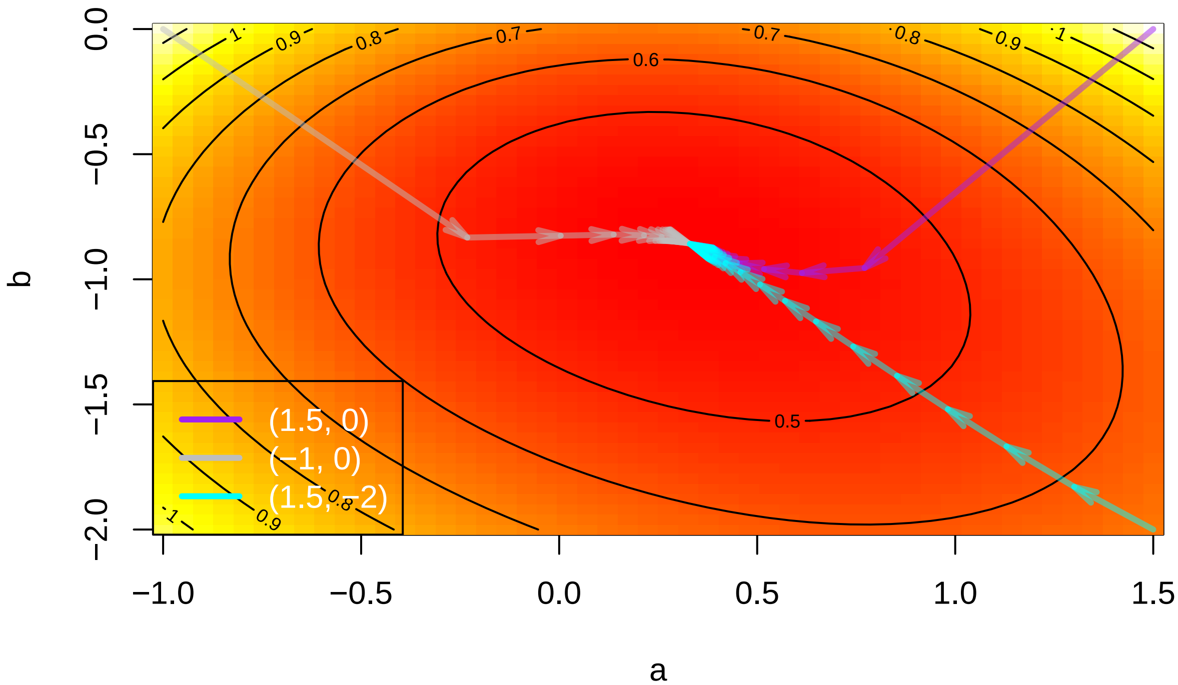
perform unormalized gradient descent using the function

```r
# use code from Q3
createObjProbit <- function(x, y) {
    ## local variable
    xbar <- mean(x)
    N <- length(x)

    ## return this function
    function(theta) {
        alpha <- theta[1]
        beta <- theta[2]
        pi <- pnorm(alpha + beta * (x - xbar))
        -1/N * sum(y * log(pi/(1 - pi)) + log(1 - pi))
    }
}

createGradientProbit <- function(x, y) {
    ## local variables
    xbar <- mean(x)
    N <- length(x)

    # return the function
    function(theta) {
        alpha <- theta[1]
        beta <- theta[2]
        yhat <- alpha + beta * (x - xbar)
        pi <- pnorm(yhat)

        # the gradient of -l(theta) is just negative gradient of
        # l(theta)
        -1/N * c(sum((y - pi)/(pi - pi^2) * dnorm(yhat) * 1), sum((y -
            pi)/(pi - pi^2) * dnorm(yhat) * (x - xbar)))
    }
}

### line searching could be done as a simple grid search
gridLineSearch <- function(theta, rhoFn, g, lambdaStepsize = 0.01, lambdaMax = 1) {
    ## grid of lambda values to search
    lambdas <- seq(from = 0, by = lambdaStepsize, to = lambdaMax)

    ## line search
    rhoVals <- sapply(lambdas, function(lambda) {
        rhoFn(theta - lambda * g)
```

```r
    })
    ## Return the lambda that gave the minimum
    lambdas[which.min(rhoVals)]
}

### Where testCovergence might be (relative or absolute)
testConvergence <- function(thetaNew, thetaOld, tolerance = 1e-10, relative = FALSE) {
    sum(abs(thetaNew - thetaOld)) < if (relative)
        tolerance * sum(abs(thetaOld)) else tolerance
}

# create rho and gradient functions
rho <- createObjProbit(levels$log_tries_take, levels$Easy1)
grad <- createGradientProbit(levels$log_tries_take, levels$Easy1)
```

```r
# initialize starting values
starting.values = list(c(1.5, 0), c(-1, 0), c(1.5, -2))

# apply gradient search
Optim.list = lapply(starting.values, function(theta) {
    gradientDescent(rhoFn = rho, gradientFn = grad, theta = theta, lineSearchFn = gridLineSearch,
        testConvergenceFn = testConvergence, lambdaStepsize = 0.001, lambdaMax = 1,
        tolerance = 0.001)
})
```

```r
# draw the contour plot
image(a, b, z, col = heat.colors(100), useRaster = TRUE)
contour(a, b, z, add = TRUE)

# draw the solution path
colour.list = c("purple", "gray", "cyan")

dummy = lapply(1:3, function(i) {
    solution.path = Optim.list[[i]]$SolutionPath
    n = nrow(solution.path)

    # draw the arrow
    for (j in 1:(n - 1)) {
        arrows(solution.path[j, 1], solution.path[j, 2], solution.path[j +
            1, 1], solution.path[j + 1, 2], length = 0.12, angle = 15,
            lwd = 3, col = adjustcolor(colour.list[i], alpha.f = 0.5))
    }
})
legend("bottomleft", legend = c("(1.5, 0)", "(-1, 0)", "(1.5, -2)"), col = colour.list,
    lwd = 3, text.col = "white")
```




```
mat = cbind(do.call(rbind, starting.values), do.call(rbind, lapply(Optim.list,
    unlist)))
knitr::kable(mat[, 1:7], booktabs = TRUE, col.names = c("$\\alpha_0$",
    "$\\beta_0$", "$\\alpha_*$", "$\\beta_*$", "Converged", "Iterations",
    "Obj. Value"))
```

α_0	β_0	α_*	β_*	Converged	Iterations	Obj. Value
1.5	0	0.3299263	-0.8581372	1	16	0.4158313
-1.0	0	0.3280023	-0.8568127	1	15	0.4158312
1.5	-2	0.3297111	-0.8579883	1	22	0.4158312

```
createStochasticGrad <- function(x, y, nsize = 10) {
    ## local variables
    N <- length(x)
    function(theta) {
        # generate samples
        subset = sample(N, nsize)
        x.new <- x[subset]
        y.new <- y[subset]

        alpha <- theta[1]
        beta <- theta[2]
        xbar <- mean(x.new)
```

```r
        yhat <- alpha + beta * (x.new - xbar)
        pi <- pnorm(yhat)

        # the gradient of -l(theta) is just negative gradient of
        # l(theta)
        -1/N * c(sum((y.new - pi)/(pi - pi^2) * dnorm(yhat) * 1), sum((y.new -
            pi)/(pi - pi^2) * dnorm(yhat) * (x.new - xbar)))
    }
}
```

Perform random sample stochastic gradient descent

```r
# the functions used for gradient descent use fixed step size
fixedLineSearch <- function(theta, rhoFn, g, lambdaStepsize = 0.01, lambdaMax = 1) {
    lambdaStepsize
}

gradientDescent <- function(theta = 0, rhoFn, gradientFn, lineSearchFn,
    testConvergenceFn, maxIterations = 100, tolerance = 1e-06, relative = FALSE,
    lambdaStepsize = 0, lambdaMax = 0.5) {
    # Pre-allocate a matrix to track theta at each iteration.
    SolutionPath = matrix(NA, nrow = maxIterations + 2, ncol = length(theta))
    SolutionPath[1, ] = theta
    converged <- FALSE
    i <- 0

    while (!converged & i <= maxIterations) {
        # use normalized gradient
        g <- gradientFn(theta)
        glength <- sqrt((g[1])^2 + (g[2])^2)  ## gradient direction
        if (glength > 0)
            g <- g/glength

        lambda <- lineSearchFn(theta, rhoFn, g, lambdaStepsize = lambdaStepsize,
            lambdaMax = lambdaMax)
        thetaNew <- theta - lambda * g
        converged <- testConvergenceFn(thetaNew, theta, tolerance = tolerance,
            relative = relative)
        theta <- thetaNew

        # track the seqeunce of updates
        i <- i + 1
        SolutionPath[i + 1, ] = theta
    }
    ## Return last value and whether converged or not
    return(list(theta = theta, converged = converged, iteration = i, fnValue = rhoFn(theta),
        SolutionPath = SolutionPath[1:i, ]))
}
```

```r
grad2 <- createStochasticGrad(levels$log_tries_taken, levels$Easy1, nsize = 25)
starting.values = list(c(1.5, 0), c(-1, 0), c(1.5, -2))

# perform stochastic gradient descent
```

```r
Optim.list2 = lapply(starting.values, function(theta) {
    gradientDescent(rhoFn = rho, gradientFn = grad2, theta = theta, lineSearchFn = fixedLineSearch,
        testConvergenceFn = testConvergence, lambdaStepsize = 0.05, maxIterations = 500)
})
```

```r
par(mfrow = c(1, 3))

# draw the contour
image(a, b, z, col = heat.colors(100))
contour(a, b, z, add = TRUE)

# draw the solution path
colour.list = c("purple", "gray", "cyan")

dummy = lapply(1:3, function(i) {
    solution.path = Optim.list2[[i]]$SolutionPath
    n = nrow(solution.path)

    # draw the arrow
    for (j in 1:(n - 1)) {
        arrows(solution.path[j, 1], solution.path[j, 2], solution.path[j +
            1, 1], solution.path[j + 1, 2], length = 0.12, angle = 15,
            lwd = 3, col = adjustcolor(colour.list[i], alpha.f = 0.5))
    }
    legend("bottomleft", legend = c("(1.5, 0)", "(-1, 0)", "(1.5, -2)"),
        col = colour.list, lwd = 3)
})

plot.iteration <- function(i, ...) {
    max <- max(c(Optim.list2[[1]]$SolutionPath[, i], Optim.list2[[2]]$SolutionPath[,
        i], Optim.list2[[3]]$SolutionPath[, i]))
    min <- min(c(Optim.list2[[1]]$SolutionPath[, i], Optim.list2[[2]]$SolutionPath[,
        i], Optim.list2[[3]]$SolutionPath[, i]))

    # plot the empty coordinate
    plot(1, type = "n", xlab = "iteration", xlim = c(0, 501), ylim = c(min,
        max), ...)

    color.list = c("purple", "gray", "cyan")

    for (j in 1:3) {
        lines(Optim.list2[[j]]$SolutionPath[, i], col = color.list[j])
    }

    # find the minimum fn_value and use the alpha beta
    min.index = 1
    if (Optim.list2[[1]]$fnValue > Optim.list2[[2]]$fnValue) {
        min.index = 2
    }
    if (min.index == 2 && Optim.list2[[2]]$fnValue > Optim.list2[[3]]$fnValue) {
        min.index = 3
    } else if (min.index == 1 && Optim.list2[[1]]$fnValue > Optim.list2[[3]]$fnValue) {
        min.index = 3
    }
```

```
    abline(h = Optim.list2[[min.index]]$theta[i], col = "black")

    legend("bottomleft", legend = c("(1.5, 0)", "(-1, 0)", "(1.5, -2)"),
        col = colour.list, lwd = 3)
}

# plot alpha vs. iteration
plot.iteration(1, ylab = bquote(alpha), main = "alpha vs. iterations")

# plot beta vs. iteration
plot.iteration(2, ylab = bquote(beta), main = "beta vs. iterations")
```