Gradient descent

Joshua Loftus

2022-10-19

Introduction

First we introduce the basic algorithm and explore some of its important design components: **step sizes** and **convergence checking**.

The final section of this notebook shows a more sophisticated implementation of gradient descent for estimating logistic regression coefficients that uses a few advanced topics, so it's OK to not understand all the code in a first read. Some interesting experiments to try in that section: changing the data generating parameters (`n`, `p`, `sparsity`, etc) or algorithm parameters (`max_steps`, `tol`).

(Note: if there's any place where code is not displayed the source can be seen in the .Rmd file version of this notebook).

Gradient descent basics

We can think of this iterative optimization algorithm as minimizing a function by taking steps in the direction of the steepest decrease (or descent). The path followed by the algorithm is like someone skiiing downhill as fast as possible by always turning in the direction with the steepest downward slope. Here's a simple example using the function $f(x) = x^4$. This starts at the point $x_0 = 1$, and then computes a sequence of inputs $x_1, x_2, \ldots, x_n$ with the goal that $f(x_n)$ converges to a (potentially **local**!) minimum of f. In this case f has a global minimum at $x = 0$ of $f(0) = 0$, and we hope our algorithm will give this correct answer.

```
# Function to minimize
f <- function(x) x^4
# Derivative (because it's univariate)
grad_f <- function(x) 4 * x^3
xn <- 1 # starting point
fn <- f(xn)
step_size <- .1
for (step in 1:5) {
  # Update step
  xn <- xn - step_size * grad_f(xn)
  # Show results
  cat(paste0("At step ", step,
             ", xn = ", round(xn, 5),
             " and f(xn) = ", round(f(xn), 5)), "\n")
}
```

```
## At step 1, xn = 0.6 and f(xn) = 0.1296
## At step 2, xn = 0.5136 and f(xn) = 0.06958
## At step 3, xn = 0.45941 and f(xn) = 0.04454
## At step 4, xn = 0.42062 and f(xn) = 0.0313
## At step 5, xn = 0.39086 and f(xn) = 0.02334
```

Notice that `xn` is getting closer to 0 and `f(xn)` is decreasing. Perhaps if the algorithm takes more steps it will converge to the minimizer of f.

However, I've chosen the values here carefully. Right now it converges very slow. If we increase the number of steps from 5 to 100 it still has not yet reached even `xn = 0.1`, never mind the actual minimizer of 0. If we change the `step_size` or initial starting point `xn` to some other values it may converge faster, or it may diverge to infinity instead of converging to 0.

This shows that **step sizes and the number of steps are important parts of any gradient descent implementation**, and performance will also depend on the function (and its gradient) and starting points.

Step sizes

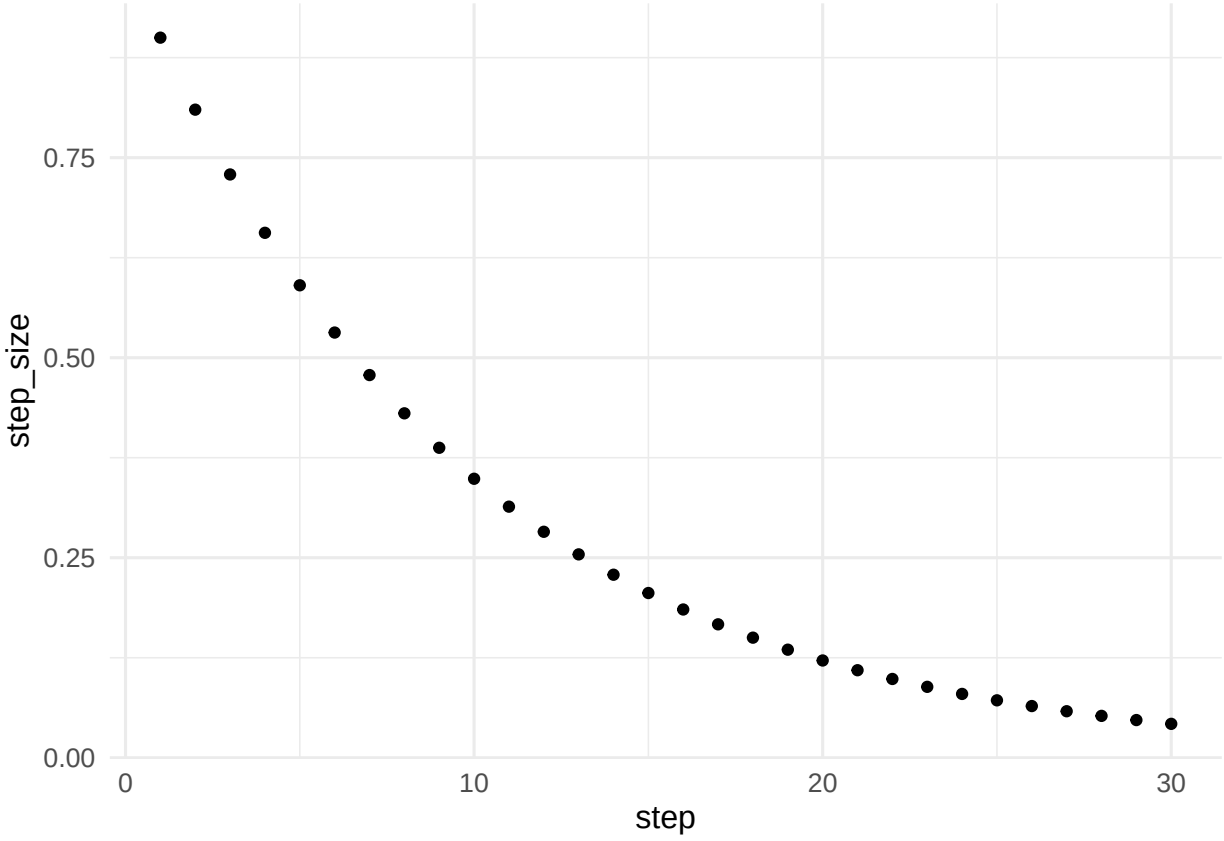
The step size is a scalar multiplier of the gradient. It's denoted γ_n in the Wikipedia article and we'll keep that convention.

We need the step size to decrease to zero so that the algorithm doesn't get stuck jumping back and forth past the minimum value instead of getting closer to it. But how fast should it decrease?

If it starts too small or decreases too fast then gradient descent will require many more tiny steps before it reaches the optimum. On the other hand, if it decreases too slowly then gradient descent may jump around the optimum many times before the step size decreases enough for it to get closer.

Geometric decrease One common approach is to take a number $0 < \gamma < 1$, e.g. $\gamma \approx .9$, and make step n have step size γ^n. The step sizes then would look like this:

```
gamma <- .9
data.frame(step = 1:30) |>
  mutate(step_size = gamma^step) |>
  ggplot(aes(x = step, y = step_size)) + geom_point()
```

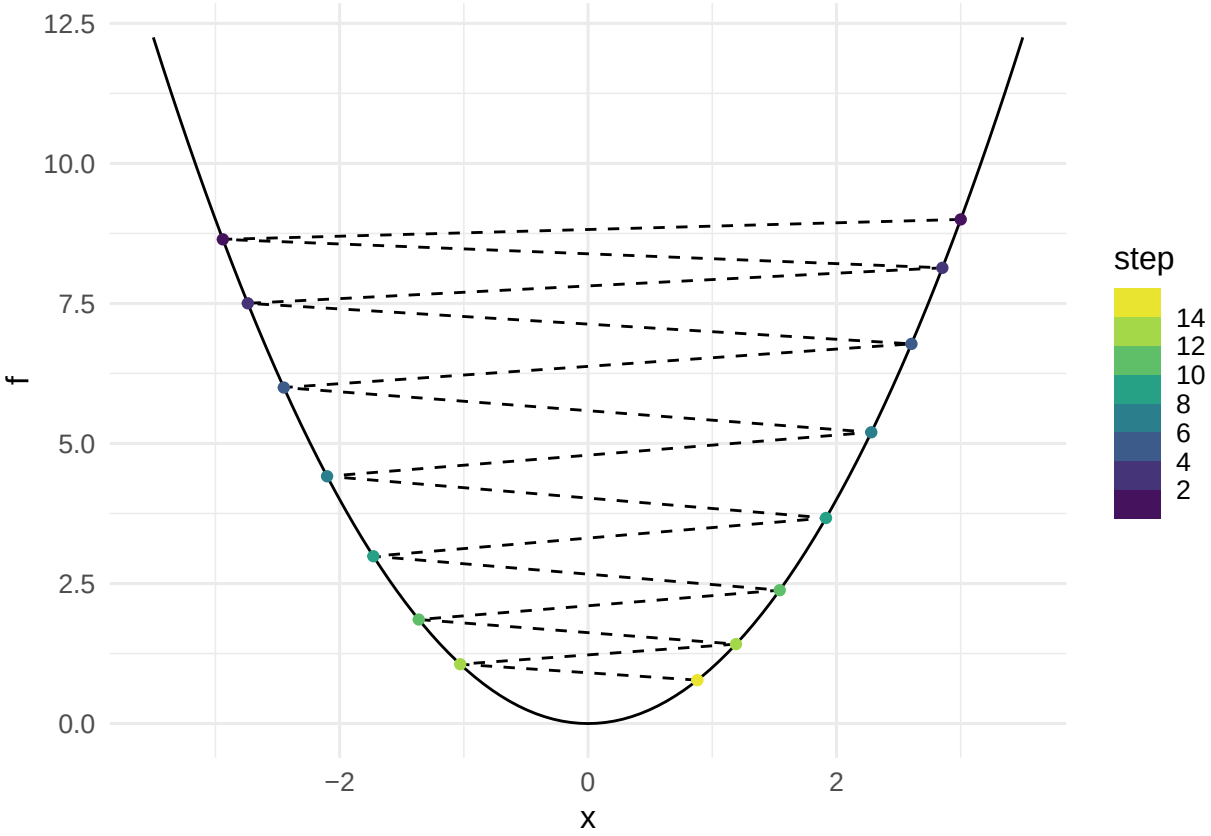


Let's see this in action trying to minimize $f(x) = x^2$.

First we'll try a value of γ which is too large.

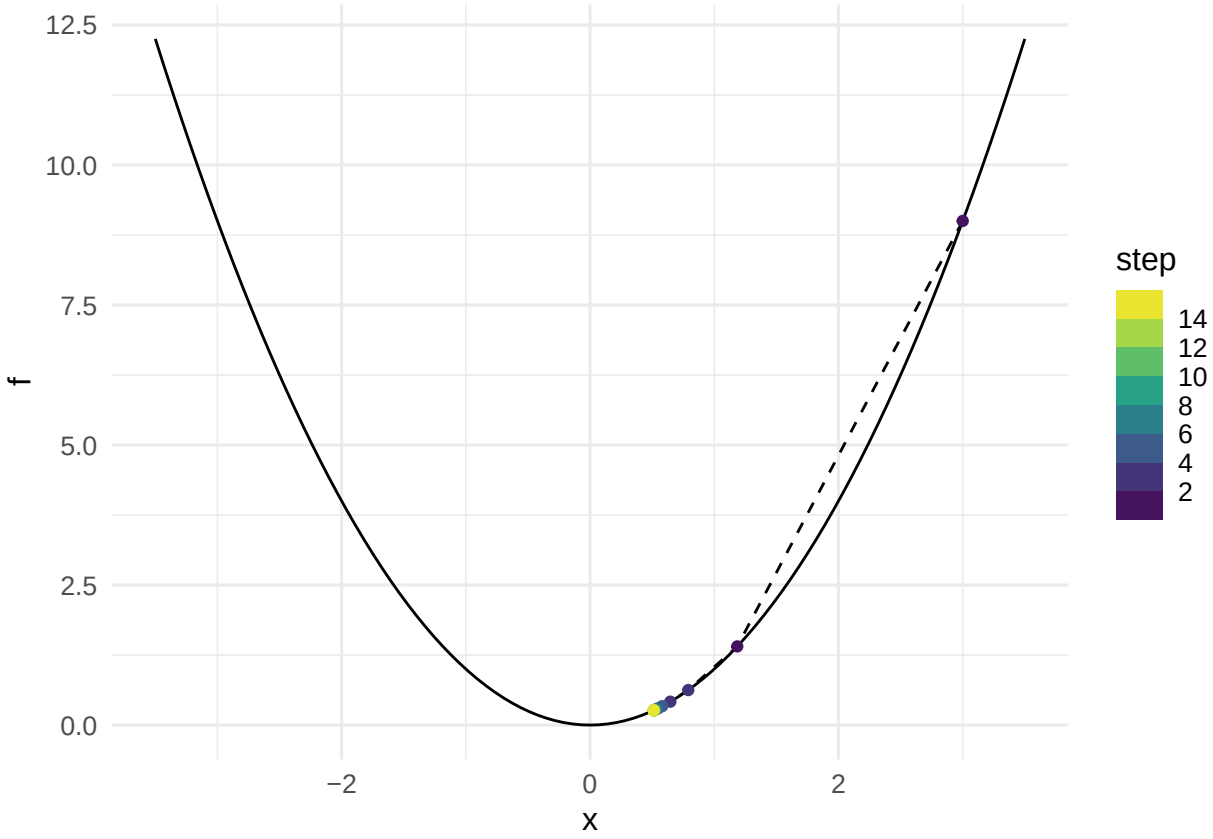
```
f <- function(x) x^2 # Function to optimize
grad_f <- function(x) 2 * x # Gradient of f
max_steps <- 15
x0 <- 3 # starting point
gamma <- .995
# ... (rest of the code not displayed)
```

Now we plot the path of the algorithm. We can see gradient descent keeps jumping past the optimum many times, wasting computation by converging too slowly.



Now we'll try a value of γ which is too small. This time gradient descent converges too fast because the step sizes become numerically close to zero even though we have not yet reached the optimum (minimum of f).

```
gamma <- .55
```



Recall that at a (local) optimal value, the derivative (or gradient) should be zero (or undefined, as in the case of $|x|$). We can use this to check if gradient descent converged (stopped moving) because of the step size and has not yet reached an optimum:

```
grad_f(xnext) # numerically close to zero?
```

```
## [1] 1.025414
```

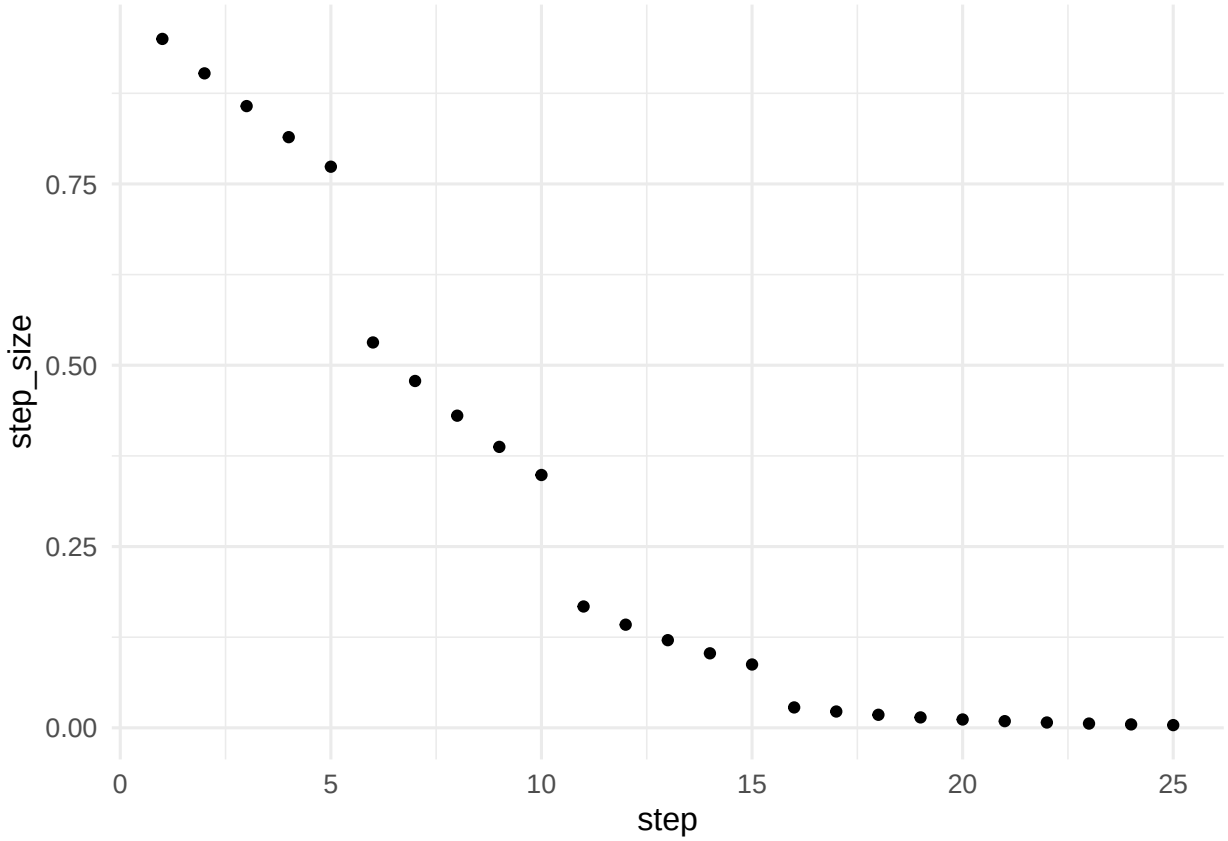
The gradient isn't close to zero, instead the algorithm has stopped because the overall step is small:

```
gamma^step * grad_f(xnext)
```

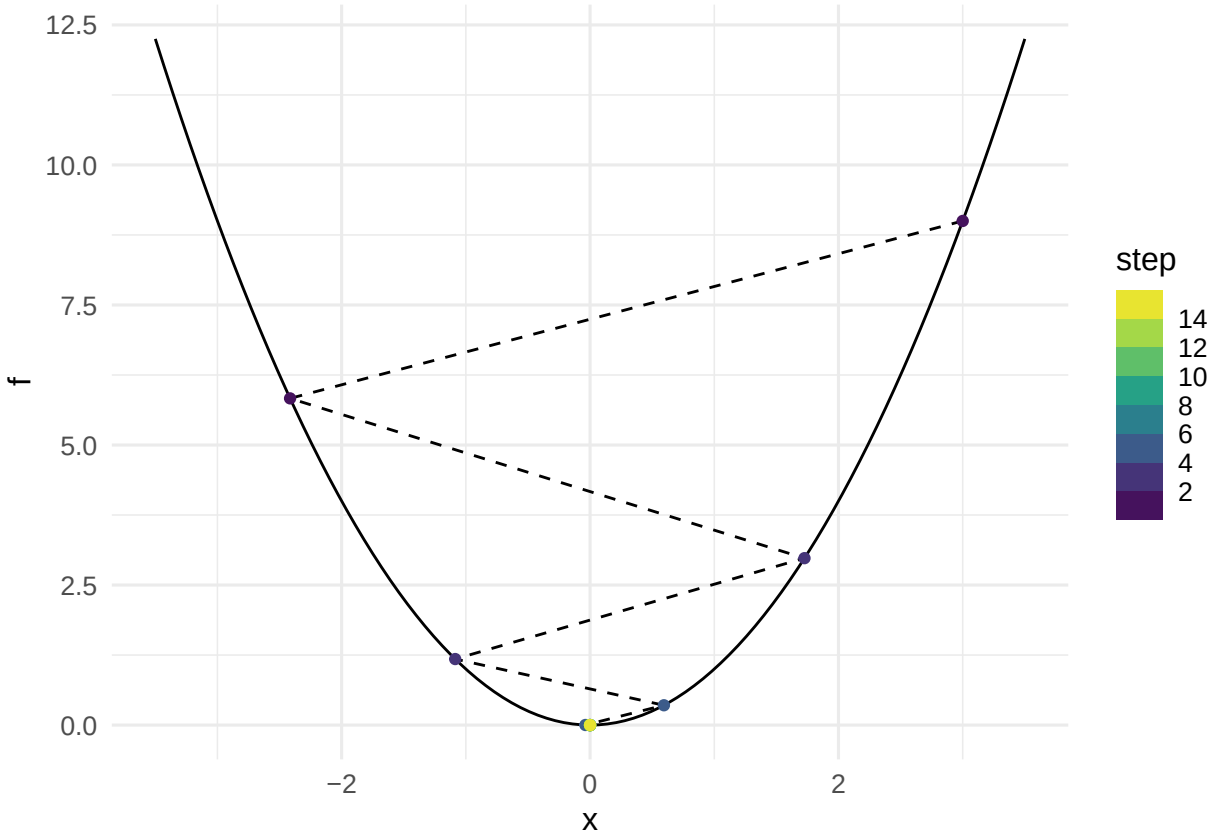
```
## [1] 0.0001307192
```

Summary: Step sizes are decreasing too slow if gradient descent jumps around the optimum and takes a long time to converge. Step sizes are decreasing too fast if gradient descent converges but fails to reach an optimum.

Hand-tuning In practice, people often try different values of γ, check to see if gradient descent is converging too slowly or quickly, then modify γ and try again. Or they might pick a sequence of γ values using multiple phases that looks something like this:



Using these step sizes to minimize $f(x) = x^2$ again, this time the result would converge better:



But strategies like this require "hand tuning," which we usually prefer to avoid. Instead, if they are available we could use "adaptive" methods that work automatically (most of the time).

Final implementation

We'll now create a version with adaptive step sizes and numeric differentiation, and apply it to a logistic regression model.

Log-likelihood function

This is the function we want to minimize to fit a logistic regression model. This is a special case of maximum likelihood estimation, which could also be applied to any other `glm` (or parametric probability model).

```
# See e.g. lecture slides
logL <- function(X, Y, beta) {
  Xbeta <- X %*% beta
  expXbeta <- exp(-Xbeta)
  exp_ratio <- 1/(1+expXbeta)
  sum(Y * log(exp_ratio) + (1-Y) * log(1-exp_ratio))
}
```

Numeric differentiation

Some functions are difficult to differentiate by hand, or may not even have a closed-form expression for their gradient. But we can still use numeric differentiation like this **finite difference** method to approximate the gradient. Instead of taking a limit (as in the definition of derivatives or gradients), we just substitute in a very small value of `h` and compute:

$$\frac{f(x+h) - f(x-h)}{2h}$$

Or, in code: `(f(x+h) - f(x-h))/(2*h)` (note: I spent a long time wondering why I was getting an error message because my code had `2h` instead of `2*h`... this happens sometimes even after many years of experience). The limit of this as `h` goes to zero would be the exact gradient (or derivative) rather than an approximation. But since we use this approximate method we don't need to do any calculus to find an expression for the gradient, we just need to evaluate the function itself.

Reference: Wikipedia

```
# Note: This value of h is chosen to be
# on the order of magnitude of the cube-root
# of .Machine$double.eps, which helps avoid
# some numerical approximation errors
numeric_grad <- function(X, Y, beta, h = 1e-06) {
  # Approximate each coordinate of the gradient
  numerator <- sapply(1:length(beta), function(j) {
    H <- rep(0, length(beta))
    H[j] <- h # step in coordinate j
    logL(X, Y, beta + H) - logL(X, Y, beta - H)
  })
  numerator / (2*h)
}
```

Adaptive step sizes

We'll use the *Barzilai–Borwein method* for choosing a step size. This BB-method requires we keep track of two consecutive values of the function and its gradient in order to compute a step size, and that is why the code for our final implementation is a little more complicated.

Reference: Wikipedia

Generating simulated data

Now we create some data to fit logistic regression models.

```
# Setting parameters
set.seed(1)
# Try changing some parameters
n <- 200
p <- 5
sparsity <- 2
nonzero_beta <- c(1, -1)
# or, e.g. rep(1, sparsity), runif(sparsity, min = -1, max = 1)
true_beta <- c(.5, nonzero_beta, rep(0, p - sparsity))
```

```
# Generating simulated data
X <- cbind(rep(1, n), matrix(rnorm(n*p), nrow = n))
mu <- X %*% true_beta
px <- exp(mu)/(1+exp(mu))
Y <- rbinom(n, 1, prob = px)
train_ld <- data.frame(y = as.factor(Y), x = X)
fit_glm <- glm(Y ~ X-1, family = "binomial")
# to compare with our results later
```

Putting it all together

```
# Initialize and take first step
max_steps <- 50 # try ~3 for comparison
beta_prev2 <- rnorm(ncol(X)) # random start
#beta_prev2 <- c(mean(Y), cor(X[,-1], Y)) # "warm" start
grad_prev2 <- numeric_grad(X, Y, beta_prev2)
beta_prev1 <- beta_prev2 + 0.1 * grad_prev2 / sqrt(sum(grad_prev2^2))
grad_prev1 <- numeric_grad(X, Y, beta_prev1)
previous_loss <- logL(X, Y, beta_prev2)
next_loss <- logL(X, Y, beta_prev1)
# Store coefficient path
beta_mat <- rbind(beta_prev2,
                  beta_prev1,
                  matrix(0, nrow = max_steps - 1, ncol = ncol(X)))
rownames(beta_mat) <- NULL
beta_path <- data.frame(step = 0:max_steps,
                        logL = c(previous_loss, next_loss, rep(0, max_steps - 1)),
                        b = beta_mat)
steps <- 1
```

```
# Repeat until convergence
tol <- 1e-5 # (error) tolerance
while (abs(previous_loss - next_loss) > tol) {
  # Compute BB step size
  grad_diff <- grad_prev1 - grad_prev2
  step_BB <- sum((beta_prev1 - beta_prev2) * grad_diff) / sum(grad_diff^2)
  beta_prev2 <- beta_prev1

  # Update step
  beta_prev1 <- beta_prev1 - step_BB * grad_prev1

  # Shift previous steps
  grad_prev2 <- grad_prev1
  grad_prev1 <- numeric_grad(X, Y, beta_prev1)
  previous_loss <- next_loss
  next_loss <- logL(X, Y, beta_prev1)

  # Print loss every 5 steps, track path, control loop
  if (round(steps/5) == steps/5) print(previous_loss)
  steps <- steps + 1
  beta_path[steps, 2] <- next_loss
  beta_path[steps, 3:ncol(beta_path)] <- beta_prev1
  if (steps > max_steps) break
}
```

```
## [1] -92.35741
## [1] -91.48657
```

How many steps?

```
steps - 1 # not counting starting point
```

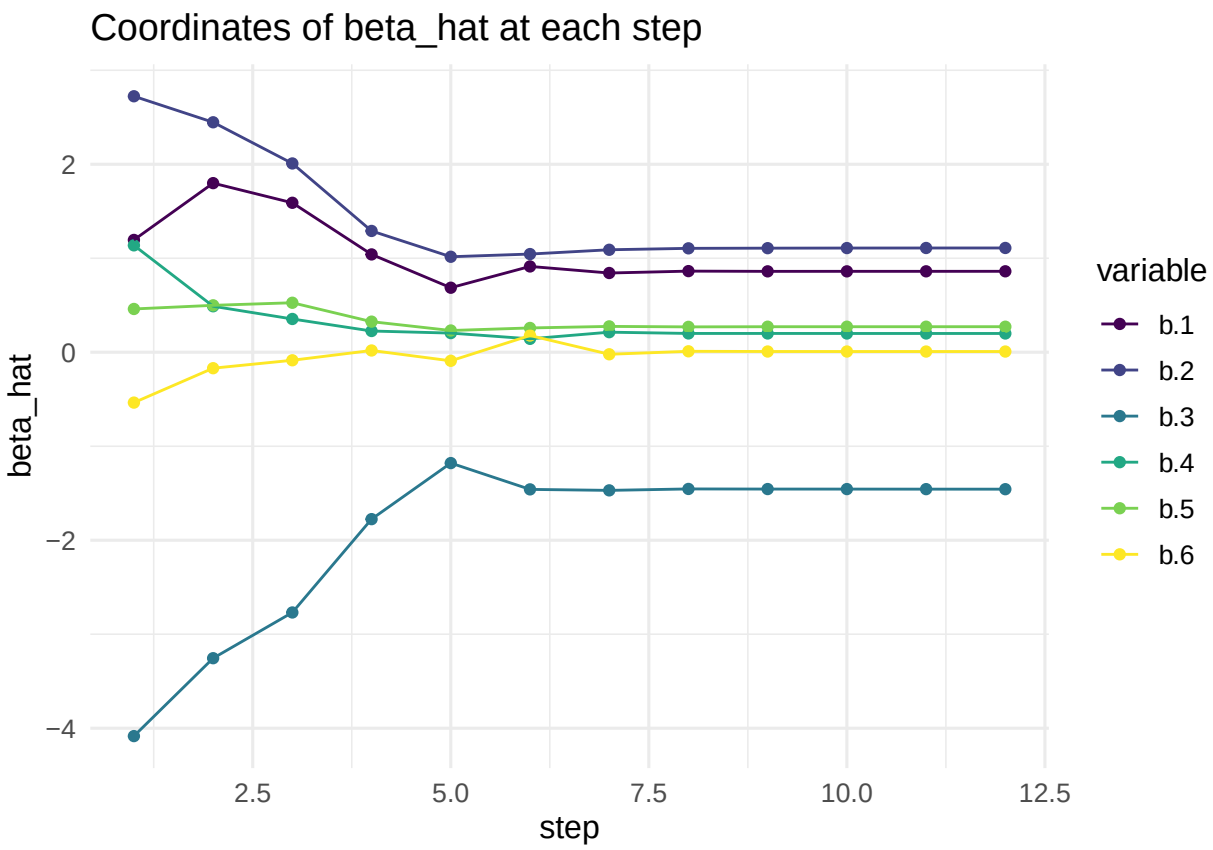
```
## [1] 12
```

```
rbind(
  fit_glm |> coef(),
  beta_prev1,
  true_beta
)
```

Did it work?

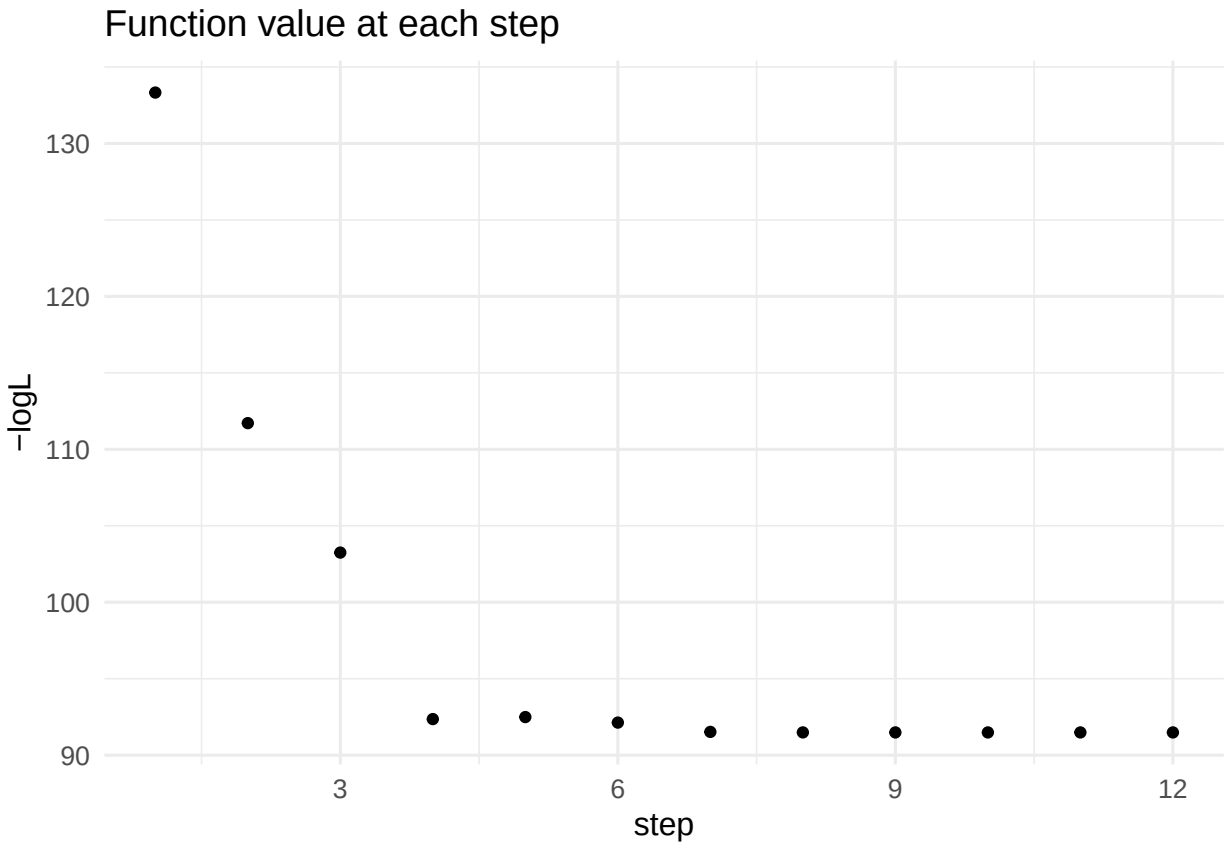
```
##                   X1       X2        X3        X4        X5           X6
##            0.8612564 1.109512 -1.457058 0.1997351 0.2717528 0.007614362
## beta_prev1 0.8611952 1.109387 -1.456916 0.1997292 0.2717247 0.007629400
## true_beta  0.5000000 1.000000 -1.000000 0.0000000 0.0000000 0.000000000
```

Coefficient paths (Leaving out the initial starting point)

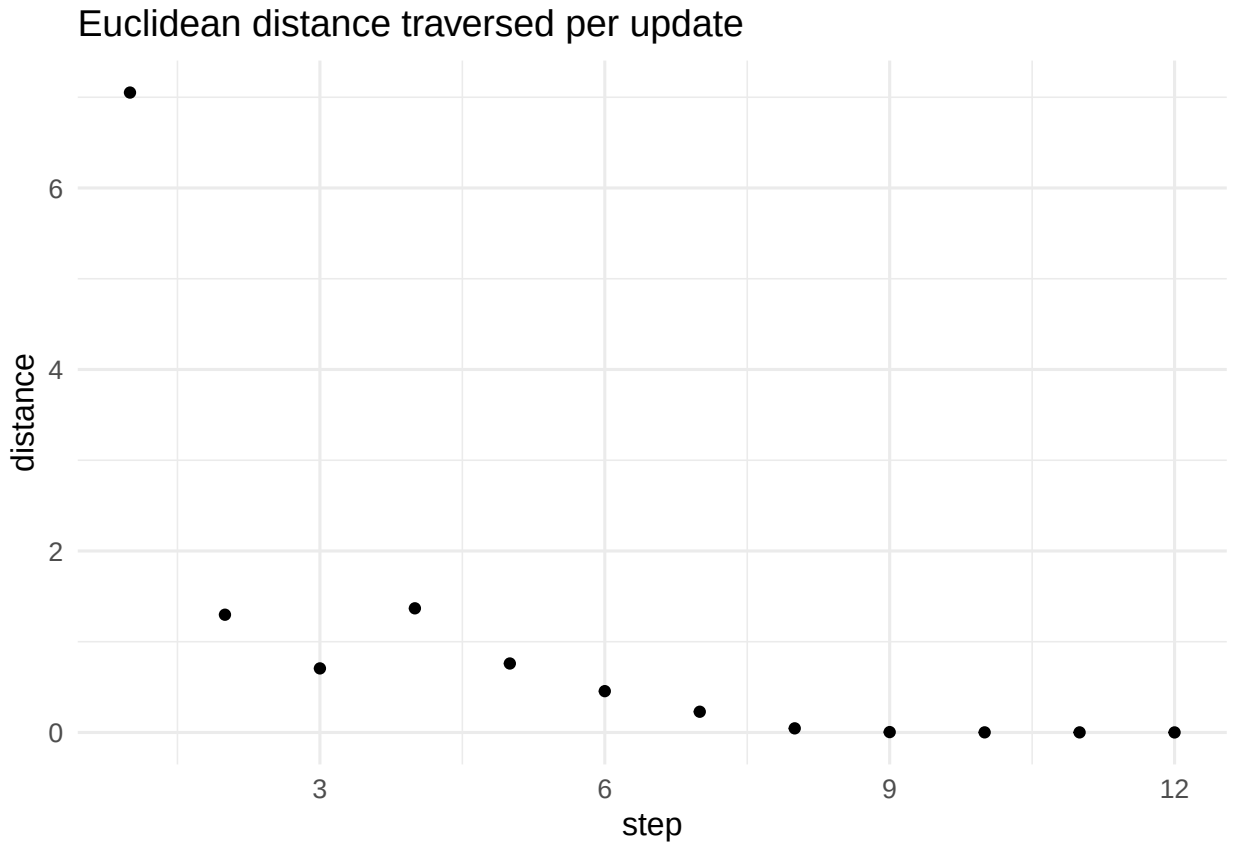


(Note: the coefficient indexes may be a bit confusing because of the intercept term)

Values of the objective function (MLE: maximizing `logL` is the same as minimizing `-logL`)



Update distances How far did gradient descent move (in Euclidean distance of the p-dimensional parameter space) in each step?



How cool is this, exactly?

Note: with numeric differentiation we did not need to do the calculus ourselves to find an expression for the gradient. (People have done a lot of work clearing this path for us, though. . .)

Question: Suppose that instead of logistic regression we wanted to fit some other kind of model. What would we need to change about the code?

Answer: Only the `logL` function! We could optimize many functions (and therefore fit many models with MLE) with almost the same code.

Some takeaways

- Gradient descent can be implemented to converge to a **local** minimum of a function, provided the function is not badly behaved (e.g. discontinuous)
- Many functions that we optimize in statistics and machine learning are convex, in which case any minimizer is a global minimizer.
- In non-convex optimization we have to be more careful about whether gradient descent converges to "bad" optimum values, i.e. ones which are locally optimal but far from globally optimal. This is a common concern in AI/deep networks.
- Convergence usually requires step sizes that decrease to zero. We can use an adaptive step size method or experiment with various approaches by hand.
- We can compute gradients either by having some closed form expression or using numeric approximation.
- We can stop the algorithm after some number of steps and/or check for convergence by seeing if the objective function's value is changing by amounts small enough to be numerically insignificant.