Optimization (cont.)

Lecture 25

Dr. Colin Rundel

Maximum Likelihood Estimation

The idea

Given observed data $x_1, \ldots, x_n$ and a parametric model $f(x|\theta)$, the likelihood is

$$\square(\theta|\mathbf{x}) = \prod_{i=1}^{n} f(x_i|\theta)$$

Typically, log-likelihood is easier to work with,

$$\ell(\theta|\mathbf{x}) = \log \square(\theta|\mathbf{x}) = \sum_{i=1}^{n} \log f(x_i|\theta)$$

The MLE is the parameter value that maximizes the likelihood (equivalently, minimizes the negative log-likelihood),

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} \ \ell(\theta|\mathbf{x}) = \arg\min_{\theta} \ -\ell(\theta|\mathbf{x})$$

Normal MLE

```r
set.seed(1234)
x_norm = rnorm(100, mean = -3.2, sd = 1.25)
c(mean = mean(x_norm), sd = sd(x_norm))
```

```
     mean        sd
-3.395952  1.255507
```

```r
nll_norm = function(theta, x) {
  -sum( dnorm(
    x,
    mean = theta[1],
    sd = theta[2],
    log = TRUE
  ) )
}
```

```r
nll_norm(c(0, 1), x=x_norm)
```

```
[1] 746.5451
```

```r
nll_norm(c(-3, 1), x=x_norm)
```

```
[1] 177.7595
```

```r
nll_norm(c(-3.2, 1.25), x=x_norm)
```

```
[1] 165.374
```

Fitting the Normal MLE - μ only

Let's first try solving for the MLE of μ with a fixed $\sigma = 1.25$:

```
1 nll_norm_mu = \(mu, x) nll_norm(c(mu, 1.25), x=x)
2 optim(c(0), nll_norm_mu, method = "BFGS", x = x_norm)
```

```
$par
[1] -3.395952

$value
[1] 164.1453

$counts
function gradient
       8        3

$convergence
[1] 0

$message
NULL
```

```
1 mean(x_norm)
```

```
[1] -3.395952
```

Fitting the Normal MLE - μ and σ

```r
optim(c(0, 1), nll_norm, method = "BFGS", x = x_norm)
```

```
$par
[1] -56.40254 202.27205

$value
[1] 626.2908

$counts
function gradient
     101      100

$convergence
[1] 1

$message
NULL
```

What has gone wrong here?

The problem

Many distribution's parameters have restricted support - e.g., $\sigma > 0$, shape > 0, rate > 0.

If `optim()` tries a negative value, the log-likelihood returns `NaN`:

```
dnorm(0, mean = 0, sd = -1, log = TRUE)
```

```
[1] NaN
```

```
dgamma(1, shape = -1, rate = 1, log = TRUE)
```

```
[1] NaN
```

These failures can cause the optimizer to fail or give unreliable results, since it relies on evaluating the objective and its gradient.

Failure to take these constraints into account does not always cause an error - it depends on the method, the starting location, the shape of the objective function, and the location of the optimum.

Gamma MLE

For example, the MLE for a Gamma distribution with shape and rate parameters both required to be > 0:

```r
set.seed(5678)
x_gam = rgamma(200, shape = 3, rate = 2)
```

```r
nll_gamma = function(theta, x) {
  -sum( dgamma(
    x,
    shape = theta[1],
    rate = theta[2],
    log = TRUE
  ) )
}
```

```r
optim(
  c(1, 1), nll_gamma, method = "BFGS", x = x
)
```

```
$par
[1] 2.924804 2.013266

$value
[1] 226.3538

$counts
function gradient
      28       10

$convergence
[1] 0

$message
NULL
```

Beta MLE

Similarly, the MLE for a Beta distribution with shape1 and shape2 parameters, both are required to be > 0:

```r
set.seed(9012)
x_beta = rbeta(200, shape1 = 2, shape2 = 5)
```

```r
nll_beta = function(theta, x) {
  -sum( dbeta(
    x,
    shape1 = theta[1],
    shape2 = theta[2],
    log = TRUE
  ) )
}
```

```r
optim(
  c(1, 1), nll_beta, method = "BFGS", x = x_
)
```

```
$par
[1] 2.484288 5.506032

$value
[1] -99.01453

$counts
function gradient
      40       13

$convergence
[1] 0

$message
NULL
```

Solutions?

Other than hoping things will work out, there are a couple of common solutions to this problem of constrained parameters (one bad, and two good):

1. Force the optimizer to stay in the valid region by returning `Inf` or another large number for invalid parameters (bad)
2. Reparameterization - optimize over a transformed, unconstrained space (good)
3. Box constraints - use `L-BFGS-B` with `lower` / `upper` bounds (good)

Naive approach: penalize invalid parameters

This approach is simple but can cause problems - the optimizer may struggle to find a valid region, and the behavior near the boundary can be erratic.

```r
nll_norm_penalty = function(theta, x) {
  if (theta[2] <= 0)
    return(Inf)

  -sum( dnorm(
    x,
    mean = theta[1],
    sd = theta[2],
    log = TRUE
  ) )
}
```

```r
optim(
  c(0,1), nll_norm_penalty, method = "BFGS",
)
```

```
$par
[1] -56.40254 202.27205

$value
[1] 626.2908

$counts
function gradient
     101      100

$convergence
[1] 1

$message
NULL
```

Wait, that did work either - why?

More iterations!

Both now and before our optimization didn't fail, it just ran out of iterations before convergence. We can change this via `control`.

```r
optim(
  c(0,1), nll_norm_penalty,
  method = "BFGS",
  control = list(maxit = 1000),
  x = x_norm
)
```

```
$par
[1] -3.395957  1.249217

$value
[1] 164.1453

$counts
function gradient
     352      329

$convergence
[1] 0

$message
NULL
```

```r
optim(
  c(0,1), nll_norm,
  method = "BFGS",
  control = list(maxit = 1000),
  x = x_norm
)
```

```
Warning in dnorm(x, mean = theta[1], sd = theta[
produced
Warning in dnorm(x, mean = theta[1], sd = theta[
produced
Warning in dnorm(x, mean = theta[1], sd = theta[
produced
Warning in dnorm(x, mean = theta[1], sd = theta[
produced
Warning in dnorm(x, mean = theta[1], sd = theta[
produced
Warning in dnorm(x, mean = theta[1], sd = theta[
produced
Warning in dnorm(x, mean = theta[1], sd = theta[
produced
Warning in dnorm(x, mean = theta[1], sd = theta[
produced

$par
[1] -3.395957  1.249217

$value
[1] 164.1453

$counts
function gradient
     352      329
```

Optimizer behavior

This behavior we've just seen is specific to `BFGS` - other methods will handle things differently:

```
optim(
  c(0,1), nll_norm,
  method = "L-BFGS-B",
  control = list(maxit = 1000),
  x = x_norm
)
```

```
Error in `optim()`:
! L-BFGS-B needs finite values of 'fn'
```

```
optim(
  c(0,1), nll_norm,
  method = "CG",
  control = list(maxit = 1000),
  x = x_norm
)
```

```
$par
[1] -3.395952  1.249214

$value
[1] 164.1453

$counts
function gradient
     190       71

$convergence
[1] 0

$message
NULL
```

```r
optim(
  c(0,1), nll_norm,
  method = "Nelder-Mead",
  x = x_norm
)
```

```
$par
[1] -3.395525  1.249019

$value
[1] 164.1453

$counts
function gradient
      73       NA

$convergence
[1] 0

$message
NULL
```

Why is this a bad idea?

- Essentially we are screwing with the objective function in a way that can make it non-smooth or have weird behavior near the boundary
  - Specifically we have a discontinuity at $\sigma = 0$ where the function jumps from a finite value to `Inf`
  - The is a significant issue for any of the gradient-based methods
- This is also already more-or-less the default behavior - `NaN` values are rejected and backtracking or similar approaches are used to find a valid point
  - We just avoided seeing the `NaN` warnings by returning `Inf`
- We were able to obtain a solution but it was very inefficient - this will not scale well to more complex problems or higher dimensions

Reparameterization

Optimize over $\log(\sigma)$ instead of σ, using $\exp()$ to map back to the positive reals.

```r
nll_norm_reparam = function(theta) {
  -sum(dnorm(
    x_norm, mean = theta[1],
    sd = exp(theta[2]), log = TRUE
  ))
}
```

```r
(res = optim(
  c(0, log(1)), nll_norm_reparam,
  method = "BFGS"
))
```

```
$par
[1] -3.3959601  0.2225083

$value
[1] 164.1453

$counts
function gradient
      27       11

$convergence
[1] 0

$message
NULL
```

```r
c(mu = res$par[1], sigma = exp(res$par[2]))
```

```
       mu     sigma
-3.395960  1.249206
```

Reparameterized Gamma MLE

Both shape and rate must be positive - optimize over their logs.

```r
nll_gamma_reparam = function(theta) {
  -sum(dgamma(
    x_gam,
    shape = exp(theta[1]),
    rate = exp(theta[2]),
    log = TRUE
  ))
}
```

```r
(res_gam = optim(
  c(0, 0), nll_gamma_reparam,
  method = "BFGS"
))
```

```
$par
[1] 1.0732240 0.6997546

$value
[1] 226.3538

$counts
function gradient
      36        9

$convergence
[1] 0

$message
NULL
```

```r
exp(res_gam$par)
```

```
[1] 2.924794 2.013259
```

Box constraints with L–BFGS–B

A final alternative is to specify bounds directly via `lower` and `upper` - however this is only available for `"L–BFGS–B"`.

```
optim(
  c(0, 1), nll_norm,
  method = "L-BFGS-B",
  lower = c(-Inf, 1e-6),
  upper = c(Inf, Inf),
  x = x_norm
)
```

```
$par
[1] -3.395950  1.249205

$value
[1] 164.1453

$counts
function gradient
      17       17

$convergence
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EP
```

```
optim(
  c(1, 1), nll_gamma,
  method = "L-BFGS-B",
  lower = c(1e-6, 1e-6),
  upper = c(Inf, Inf),
  x = x_gam
)
```

```
$par
[1] 2.924803 2.013266

$value
[1] 226.3538

$counts
function gradient
      11       11

$convergence
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EP
```

Regression via `optim()`

Simulated data

```r
set.seed(42)
n = 500
p = 20

beta = rep(0,p+1)
beta[1] = 5       # Intercept
beta[c(5,15)+1] = 10   # Non-zero coefficients
beta[c(10,20)+1] = -10

d = rnorm(n * p) |>
  matrix(ncol=p) |>
  as.data.frame() |>
  as_tibble()
X = model.matrix(~ ., data = d)

y = X %*% beta + rnorm(n)
d$y = y
```

OLS as optimization

The least squares objective is

$$\mathrm{f}(\beta) = (\boldsymbol{y} - \boldsymbol{X}\beta)^T(\boldsymbol{y} - \boldsymbol{X}\beta)$$

which has the gradient

$$\nabla f(\beta) = -2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\beta)$$

```
f = function(beta, X, y) {
  sum((y - X %*% beta)^2)
}
grad = function(beta, X, y) {
  -2 * t(X) %*% (y - X %*% beta)
}
```

Fitting with `optim()`

```r
(res = optim(
  rep(0, length(beta)), f, grad,
  method = "BFGS",
  y = y, X = X
))
```

```
$par
 [1]   5.006625243  -0.014862508  -0.036479479
 [5]   0.007214624   9.987648733  -0.018293866
 [9]  -0.032787559   0.030572363  -9.942283045
[13]   0.115018149  -0.009104052   0.009628565
[17]  -0.037057357  -0.074013580   0.000941734
[21] -10.146223573

$value
[1] 498.7108

$counts
function gradient
     137       28

$convergence
[1] 0

$message
NULL
```

```r
res$par |> round(2)
```

```
 [1]   5.01  -0.01  -0.04   0.05   0.01   9.99
[10]   0.03  -9.94  -0.07   0.12  -0.01   0.01
[19]   0.00  -0.01 -10.15
```

```r
lm(y ~ ., data = d) |>
  coef() |>
  round(2)
```

```
(Intercept)          V1          V2          V3
       5.01       -0.01       -0.04        0.05
         V5          V6          V7          V8
       9.99       -0.02        0.05       -0.03
        V10         V11         V12         V13
      -9.94       -0.07        0.12       -0.01
        V15         V16         V17         V18
       9.93       -0.04       -0.07        0.00
        V20
     -10.15
```

Fitting with `optim()` w/o gradient

```r
(res = optim(
  rep(0, length(beta)), f, method = "BFGS",
  y = y, X = X
))
```

```
$par
 [1]   5.006625243  -0.014862508  -0.036479479
 [5]   0.007214624   9.987648733  -0.018293866
 [9]  -0.032787559   0.030572363  -9.942283045
[13]   0.115018149  -0.009104052   0.009628565
[17]  -0.037057357  -0.074013580   0.000941734
[21] -10.146223573

$value
[1] 498.7108

$counts
function gradient
     136       28

$convergence
[1] 0

$message
NULL
```

```r
res$par |> round(2)
```

```
 [1]   5.01  -0.01  -0.04   0.05   0.01   9.99
[10]   0.03  -9.94  -0.07   0.12  -0.01   0.01
[19]   0.00  -0.01 -10.15
```

```r
lm(y ~ ., data = d) |>
  coef() |>
  round(2)
```

```
(Intercept)          V1          V2          V3
       5.01       -0.01       -0.04        0.05
         V5          V6          V7          V8
       9.99       -0.02        0.05       -0.03
        V10         V11         V12         V13
      -9.94       -0.07        0.12       -0.01
        V15         V16         V17         V18
       9.93       -0.04       -0.07        0.00
        V20
     -10.15
```

Lasso regression

The Lasso adds an L_1 penalty to shrink coefficients toward zero (note: we don't penalize the intercept),

$$\text{RSS}(\beta) + \lambda \sum_{j=1}^{p} |\beta_j|$$

```
lasso_obj = function(beta, lambda) {
  0.5 * sum((y - X %*% beta)^2) / length(y) + lambda * sum(abs(beta[-1]))
}
```

Lasso results

Since the L_1 penalty is not differentiable (gradient does not exist at $\beta_i = 0$), we cannot* use a gradient based method, so instead we can try Nelder-Mead.

```
optim(
  rep(0, length(beta)),
  lasso_obj,
  method = "Nelder-Mead",
  control = list(maxit = 20000),
  lambda = 1
)
```

```
$par
 [1]  3.369242e+00  1.384229e-01  6.083931e-02 -1.916860e-04
 [5] -1.350435e-01  7.295038e+00  4.827552e-01 -1.575115e+00
 [9]  6.245021e-01 -1.073587e-01 -9.872849e+00 -2.794692e-03
[13] -6.978983e-01  6.250777e-01 -1.982127e-01  8.616425e+00
[17]  1.032443e-06  1.506265e-03  2.469692e-02 -8.303844e-07
[21] -9.475197e+00

$value
[1] 48.55695

$counts
function gradient
   10278       NA

$convergence
[1] 0
```

Lasso w/ different lambda values

```
fit_lasso = function(lambda) {
  optim(
    rep(0, length(beta)),
    lasso_obj,
    method = "Nelder-Mead",
    control = list(maxit = 20000),
    lambda = lambda
  )$par
}
```

We'll compare against `glmnet`, which minimizes the same objective (this is the engine used by `parsnip` and `tidymodels` for lasso),

```
fit_glmnet = function(lambda) {
  glmnet::glmnet(X[,-1], y, family = "gaussian", alpha = 1, lambda = lambda, standardize = FA
    coef() |>
    as.matrix() |>
    drop()
}
```

```r
cbind(
  "Truth"          = beta,           "OLS"            = coef(lm(y ~ ., data = d)),
  "optim (λ=1)"    = fit_lasso(1),   "glmnet (λ=1)"   = fit_glmnet(1),
  "optim (λ=5)"    = fit_lasso(5),   "glmnet (λ=5)"   = fit_glmnet(5),
  "optim (λ=7.5)"  = fit_lasso(7.5), "glmnet (λ=7.5)" = fit_glmnet(7.5)
) |> knitr::kable(digits=2)
```

	Truth	OLS	optim (λ=1)	glmnet (λ=1)	optim (λ=5)	glmnet (λ=5)	optim (λ=7.5)	glmnet (λ=7.5)
(Intercept)	5	5.01	3.37	5.01	5.12	5.05	4.21	5.07
V1	0	-0.01	0.14	0.00	0.17	0.00	0.02	0.00
V2	0	-0.04	0.06	0.00	-0.51	0.00	0.00	0.00
V3	0	0.05	0.00	0.00	0.36	0.00	0.00	0.00
V4	0	0.01	-0.14	0.00	0.01	0.00	-0.06	0.00
V5	10	9.99	7.30	8.99	4.77	5.05	2.41	2.59
V6	0	-0.02	0.48	0.00	0.37	0.00	0.01	0.00
V7	0	0.05	-1.58	0.00	0.00	0.00	0.00	0.00
V8	0	-0.03	0.62	0.00	0.04	0.00	0.03	0.00
V9	0	0.03	-0.11	0.00	0.00	0.00	-0.10	0.00
V10	-10	-9.94	-9.87	-8.92	-2.50	-4.77	-0.05	-2.17
V11	0	-0.07	0.00	0.00	0.00	0.00	0.13	0.00
V12	0	0.12	-0.70	0.00	0.10	0.00	-0.01	0.00
V13	0	-0.01	0.63	0.00	0.19	0.00	0.32	0.00
V14	0	0.01	-0.20	0.00	0.09	0.00	0.00	0.00
V15	10	9.93	8.62	8.79	3.71	4.20	0.51	1.33
V16	0	-0.04	0.00	0.00	0.11	0.00	0.00	0.00
V17	0	-0.07	0.00	0.00	0.98	0.00	0.01	0.00
V18	0	0.00	0.02	0.00	0.02	0.00	0.00	0.00
V19	0	-0.01	0.00	0.00	-0.01	0.00	0.02	0.00
V20	-10	-10.15	-9.48	-9.16	-3.45	-5.24	0.00	-2.80

Wait, these don't match?

Both routines minimize the *same* objective, yet the answers differ - notably `optim()` never produces exact zeros, while `glmnet()` does.

Is `lasso_obj` wrong or is `optim()` wrong?

We can check by evaluating `lasso_obj` at each solution (at $\lambda = 5$),

```
1 lasso_obj(fit_lasso(5),  lambda = 5)
```

```
[1] 169.7766
```

```
1 lasso_obj(fit_glmnet(5), lambda = 5)
```

```
[1] 148.7071
```

`glmnet` achieves a *lower* value of our own objective - this does not tell us if `lasso_obj` is correct but it does tell us that `optim()` is failing to find the minimum.

Why does Nelder-Mead struggle here?

Two things are working against us:

- Dimension - Nelder-Mead maintains a simplex of $k + 1$ vertices and is known to degrade badly beyond ~10 parameters. Here we have $k = 21$.
- Non-smoothness - the L_1 penalty has kinks at $\beta_j = 0$, and the lasso minimum typically *lies on* those kinks. Nelder-Mead has no mechanism to snap onto them, so it drifts around each kink leaving small non-zero noise instead of exact zeros.

`glmnet` sidesteps both issues by using coordinate descent with soft-thresholding - it updates one coefficient at a time using a closed-form solution that sets coefficients to *exactly* zero.

A correct objective is not enough - the optimizer has to be able to handle the specific *geometry* of the problem.

Side-quest - what about `optim()` with `BFGS`?

We know this objective is not differentiable, but what if we just ignore that and try to use `BFGS` anyway (letting it approximate the gradient)?

Moral of the story

The real world is complicated, and optimization problems can have all sorts of weird geometry and pathologies that make them difficult to solve.

- Always check the value of the objective at the solution - does it make sense? Is it better than other solutions you know about?
- As long as the issues with the gradient are not too severe, `BFGS` can often still find a reasonably good solution
- Gradient free methods can work but are not efficient and are likely to struggle with high dimensions or non-smoothness
- Problem specific algorithms (e.g., coordinate descent for lasso) are almost always more efficient and reliable than general-purpose optimizers, but require more work to implement

Why glmnet?

Beyond the fact that `glmnet` is highly optimized and widely used, it also solves the entire problem, tuning lambda, not just fitting for a single lambda value. Using `glmnet` we get the entire regularization path, which allows us to see how the coefficients evolve as we adjust the penalty.

```
fit = glmnet::glmnet(X[,-1], y, family = "gaussian", alpha = 1, standardize = FALSE)
```

```
1 fit
```

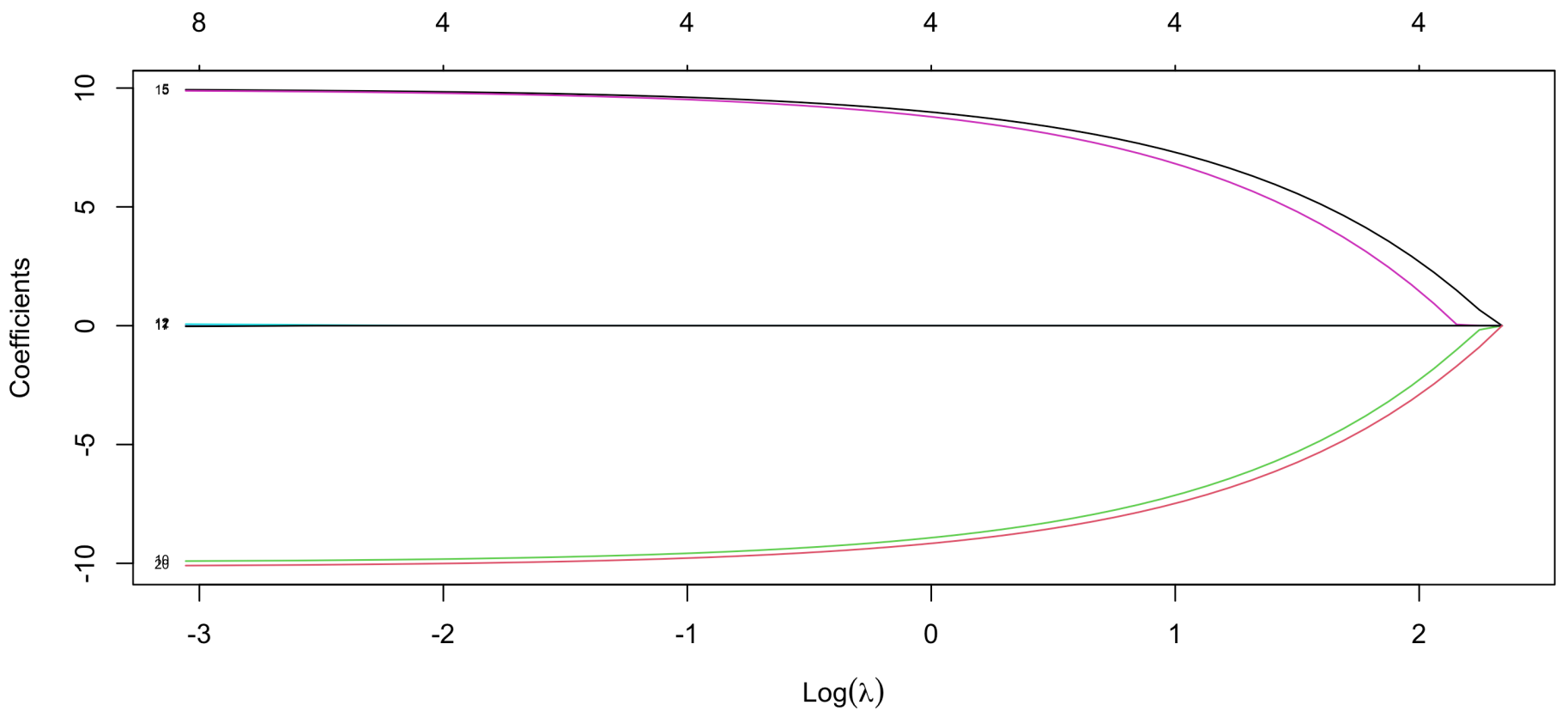
```
Call:  glmnet::glmnet(x = X[, -1], y = y, family = "gau

   Df  %Dev  Lambda
1   0  0.00 10.3800
2   3  8.80  9.4590
3   4 20.44  8.6180
4   4 33.90  7.8530
5   4 45.08  7.1550
6   4 54.36  6.5190
7   4 62.06  5.9400
8   4 68.46  5.4130
9   4 73.77  4.9320
10  4 78.18  4.4940
11  4 81.84  4.0940
12  4 84.88  3.7310
13  4 87.40  3.3990
14  4 89.49  3.0970
15  4 91.23  2.8220
16  4 92.68  2.5710
17  4 93.87  2.3430
18  4 94.87  2.1350
19  4 95.70  1.9450
20  4 96.38  1.7720
21  4 96.95  1.6150
22  4 97.42  1.4710
```

```
1 coef(fit) |> round(2)
```

```
21 x 59 sparse Matrix of class "dgCMatrix"

(Intercept) 5.18  5.13  5.08  5.07  5.07  5.06  5.06  5
V1          .     .     .     .     .     .     .     .
V2          .     .     .     .     .     .     .     .
V3          .     .     .     .     .     .     .     .
V4          .     .     .     .     .     .     .     .
V5          .     0.66  1.49  2.24  2.93  3.55  4.12  4
V6          .     .     .     .     .     .     .     .
V7          .     .     .     .     .     .     .     .
V8          .     .     .     .     .     .     .     .
V9          .     .     .     .     .     .     .     .
V10         .    -0.17 -1.01 -1.80 -2.53 -3.19 -3.79 -4
V11         .     .     .     .     .     .     .     .
V12         .     .     .     .     .     .     .     .
V13         .     .     .     .     .     .     .     .
V14         .     .     .     .     .     .     .     .
V15         .     .     0.05  0.93  1.73  2.46  3.12  3
V16         .     .     .     .     .     .     .     .
V17         .     .     .     .     .     .     .     .
V18         .     .     .     .     .     .     .     .
V19         .     .     .     .     .     .     .     .
V20         .    -0.90 -1.70 -2.45 -3.13 -3.76 -4.32 -4

(Intercept)  5.04  5.04  5.04  5.04  5.03  5.03  5.03
V1           .     .     .     .     .     .     .
V2           .     .     .     .     .     .     .
```

Coefficient trajectories

```
1 plot(fit, label=TRUE, sign.lambda=1)
```



Each line traces a coefficient as λ varies - predictors enter the model (become non-zero) as λ shrinks, with the four true non-zero coefficients (V_5, V_{10}, V_{15}, V_{20}) being

the last to survive as λ grows.

Gaussian Process Regression

The model

A Gaussian process treats the observed responses as a single draw from a multivariate normal whose covariance is determined by the inputs,

$$\underset{n\times 1}{\boldsymbol{y}} \sim \square\left(\underset{n\times 1}{\mathbf{0}}, \underset{n\times n}{\boldsymbol{\Sigma}(\boldsymbol{x};\theta)}\right)$$

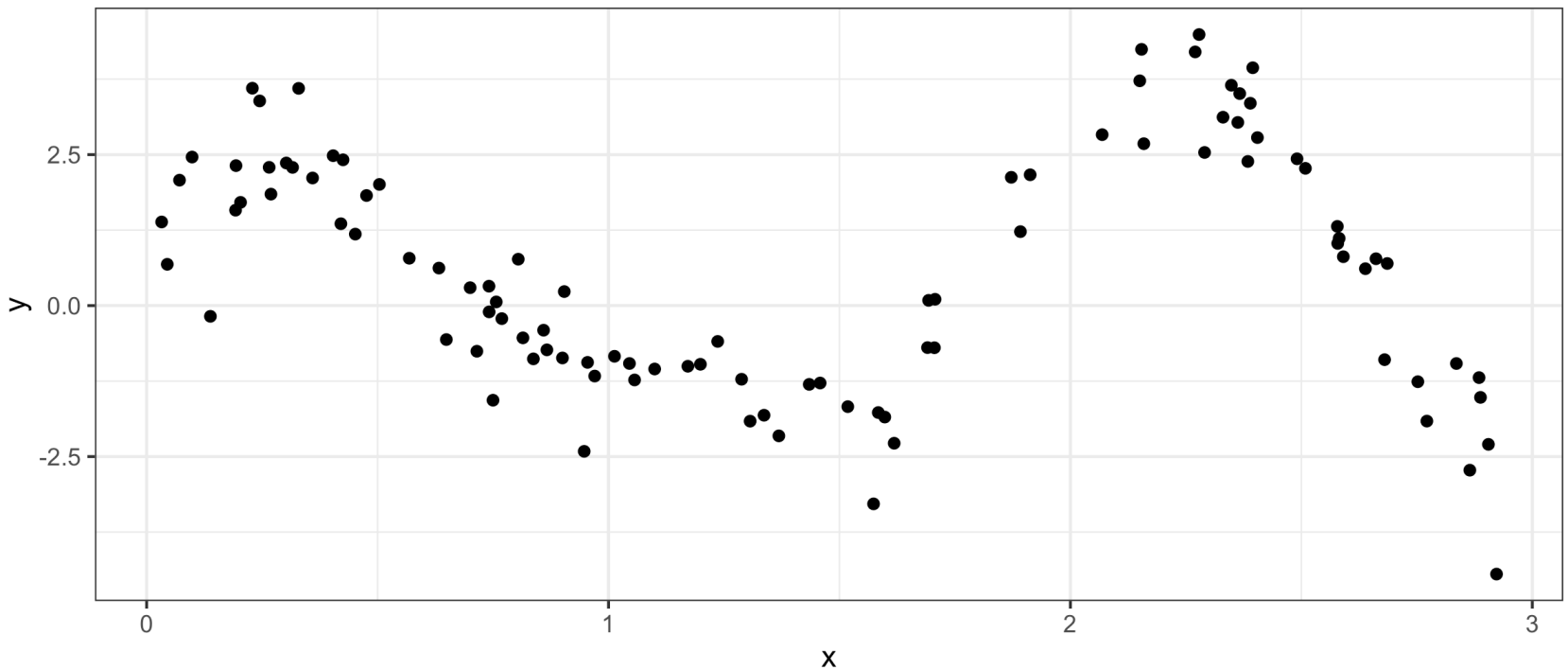
with a squared-exponential covariance plus a "nugget" (observation noise),

$$\Sigma_{ij} = \sigma_1^2 \exp\left(-\frac{(x_i - x_j)^2}{2\,\ell}\right) + \sigma_0^2\, \mathbb{1}_{[i=j]}$$

The three parameters $\theta = (\sigma_0^2, \sigma_1^2, \ell)$ are all strictly positive, so we'll fit on the log scale (just like the Gamma MLEs earlier).

The data

```
d1 = readr::read_csv("data/d1.csv")
(g = ggplot(d1, aes(x = x, y = y)) + geom_point())
```



Covariance and negative log-likelihood

```
calc_dist = function(x1, x2) {
  outer(x1, x2, \(a, b) (a - b)^2)
}

calc_cov = function(D, s2_0, s2_1, l) {
  S = s2_1 * exp(-D / (2 * l))
  S[D==0] = S[D==0] + s2_0
  S
}
```

```
nll_gp = function(log_theta, x, y) {
  theta = exp(log_theta)
  S = calc_cov(calc_dist(x, x), theta[1], theta[2], theta[3])
  -mvtnorm::dmvnorm(y, mean = rep(0, length(y)), sigma = S, log = TRUE)
}
```

Fitting via `optim()`

```
(res = optim(
  log(c(0.1, 1, 0.1)), nll_gp,
  method = "BFGS",
  x = d1$x, y = d1$y
))
```

```
$par
[1] -0.7264603  1.6474921 -1.9732176

$value
[1] 124.3481

$counts
function gradient
      37       12

$convergence
[1] 0

$message
NULL
```

```
(theta = exp(res$par)) |>
  setNames(c("s2_0", "s2_1", "l"))
```

```
     s2_0      s2_1         l
0.4836178 5.1939377 0.1390089
```

Prediction

For a set of new inputs $\boldsymbol{x}_*$ the joint distribution is

$$\begin{pmatrix} \boldsymbol{y} \\ \boldsymbol{y}_* \end{pmatrix} \sim \square\left(\boldsymbol{0}, \begin{pmatrix} \boldsymbol{\Sigma}_x & \boldsymbol{\Sigma}_{x*} \\ \boldsymbol{\Sigma}_{x*}^T & \boldsymbol{\Sigma}_* \end{pmatrix}\right)$$

so the conditional distribution of $\boldsymbol{y}_* \mid \boldsymbol{y}$ is multivariate normal with

$$\boldsymbol{\mu}_* = \boldsymbol{\Sigma}_{x*}^T \boldsymbol{\Sigma}_x^{-1} \boldsymbol{y}, \qquad \boldsymbol{\Sigma}_{*|x} = \boldsymbol{\Sigma}_* - \boldsymbol{\Sigma}_{x*}^T \boldsymbol{\Sigma}_x^{-1} \boldsymbol{\Sigma}_{x*}$$

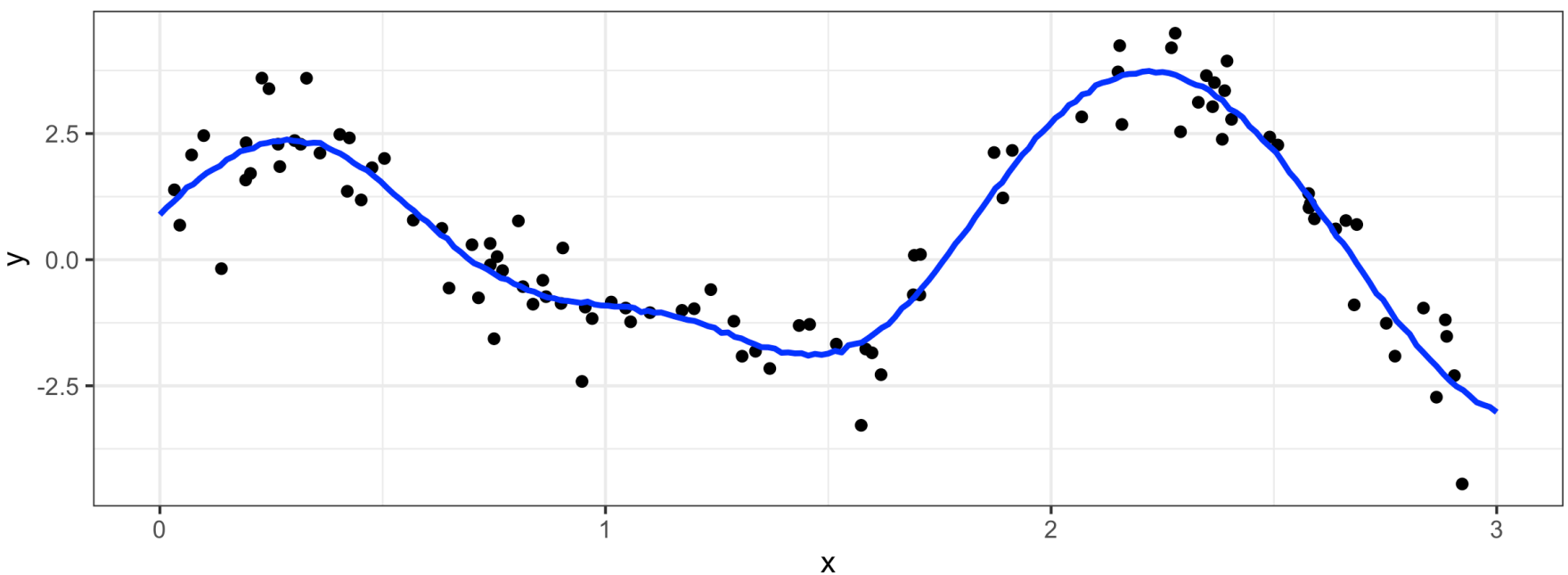
```
x_p = seq(0, 3, length.out = 201)
Sx  = calc_cov(calc_dist(d1$x, d1$x), theta[1], theta[2], theta[3])
Sxy = calc_cov(calc_dist(d1$x, x_p),  theta[1], theta[2], theta[3])
Sy  = calc_cov(calc_dist(x_p,  x_p),  theta[1], theta[2], theta[3])

Sx_inv = solve(Sx)

mu_star    = t(Sxy) %*% Sx_inv %*% d1$y |> drop()
Sigma_star = Sy - t(Sxy) %*% Sx_inv %*% Sxy + 1e-6 * diag(length(x_p))
```

Predictions

Now we draw samples from the predictive MVN and average - with enough draws this approaches the analytic conditional predictive mean $\boldsymbol{\mu}_*$.

```
y_draws = mvtnorm::rmvnorm(1000, mean = mu_star, sigma = Sigma_star)
```

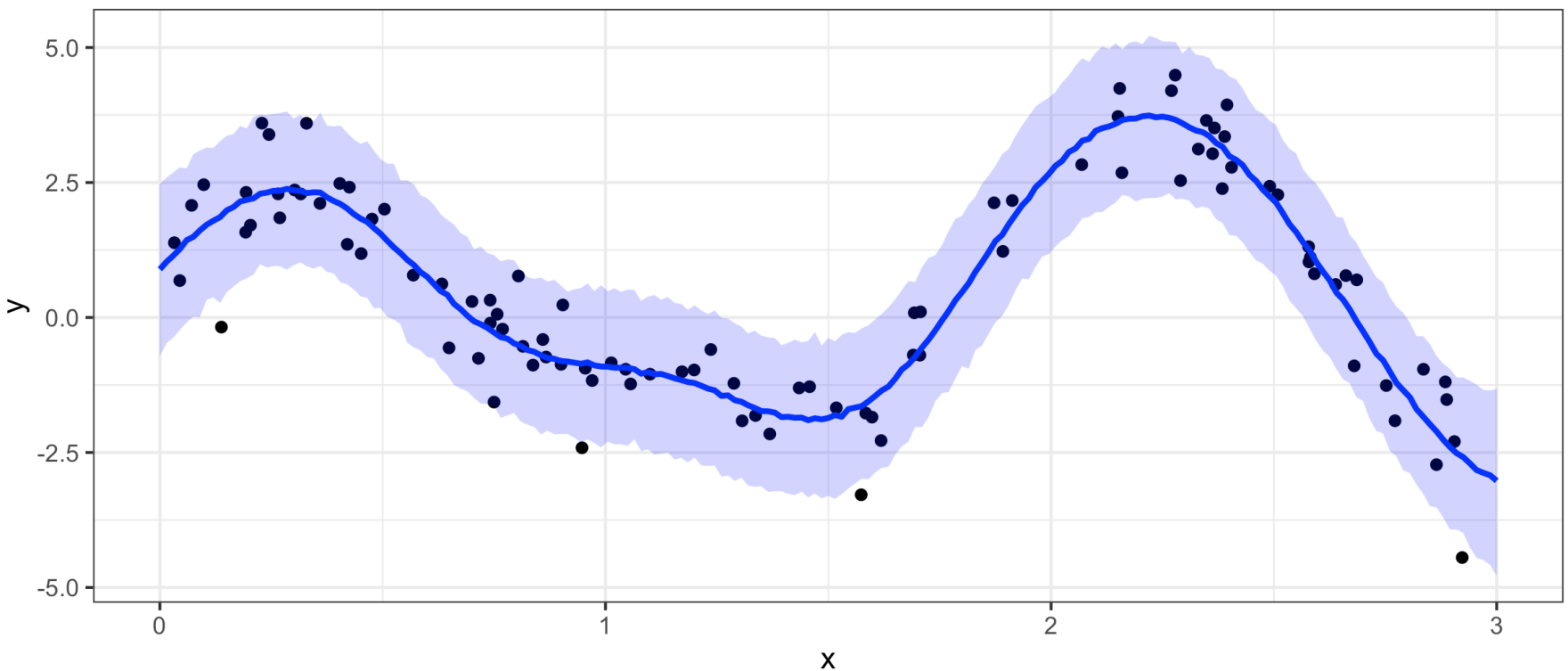
```
(g2 = g + geom_line(
    data = tibble(x = x_p, y = colMeans(y_draws)),
    color = "blue", linewidth = 1
))
```



A 3-parameter model - fit with nothing more than `optim()` and a multivariate normal density - recovers a smooth nonlinear curve directly from the noisy data.

Adding a 95% prediction interval

```
g2 + geom_ribbon(
  data = tibble(
    x = x_p, lower = apply(y_draws, 2, quantile, 0.025), upper = apply(y_draws, 2, quantile,
  ),
  aes(x = x, ymin = lower, ymax = upper),
  fill = "blue", alpha = 0.2, y = NA
)
```



Automatic Differentiation in Base R

The gradient problem

Throughout this lecture we've been supplying hand-coded gradient functions to `optim()`. This works but is:

- Error-prone - easy to make mistakes in the calculus or the code
- Tedious - especially for complex objective functions
- Hard to maintain - changing the objective means updating the gradient too

Base R provides tools for symbolic differentiation that can compute derivatives automatically: `D()`, `deriv()`, and `deriv3()`.

`D()` - symbolic derivatives

`D()` differentiates an R expression with respect to a named variable and returns a new expression.

```
D(expression(x^2), "x")
```

```
2 * x
```

```
D(expression(sin(x) * cos(y)), "x")
```

```
cos(x) * cos(y)
```

```
D(expression(sin(x) * cos(y)), "y")
```

```
-(sin(x) * sin(y))
```

```
D(expression(x^2 + 3*x*y + y^2), "x")
```

```
2 * x + 3 * y
```

```
D(expression(x^2 + 3*x*y + y^2), "y")
```

```
3 * x + 2 * y
```

Using `D()` to build gradient functions

We can evaluate the result of `D()` by providing variable values,

```
1 df_dx = D(expression(x^4 + x^3 - x^2 - x), "x")
2 df_dx
```

```
4 * x^3 + 3 * x^2 - 2 * x - 1
```

```
1 eval(df_dx, list(x = 0))
```

```
[1] -1
```

```
1 eval(df_dx, list(x = 1))
```

```
[1] 4
```

Higher-order and chained derivatives

`D()` can be applied repeatedly for higher-order derivatives,

```
expr = expression(x^4 + x^3 - x^2 - x)
D(expr, "x")
```

```
4 * x^3 + 3 * x^2 - 2 * x - 1
```

```
D(D(expr, "x"), "x")
```

```
4 * (3 * x^2) + 3 * (2 * x) - 2
```

```
D(D(D(expr, "x"), "x"), "x")
```

```
4 * (3 * (2 * x)) + 3 * 2
```

`D()` limitations

`D()` only differentiates with respect to one variable at a time and returns an expression (not a function).

For multivariate functions where we want a callable function that returns both the value *and* gradient, we need `deriv()`.

`deriv()` - gradient functions

`deriv()` takes a formula or expression and a character vector of variable names, and returns a function that computes the value and its `"gradient"` attribute.

```
f_grad = deriv(
  ~ x^2 + 3*x*y + y^2,
  namevec = c("x", "y"),
  function.arg = TRUE
)
f_grad
```

```
function (x, y)
{
    .expr2 <- 3 * x
    .value <- x^2 + .expr2 * y + y^2
    .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
        "y")))
    .grad[, "x"] <- 2 * x + 3 * y
    .grad[, "y"] <- .expr2 + 2 * y
    attr(.value, "gradient") <- .grad
    .value
}
```

```r
f_grad(x = 1, y = 2)
```

```
[1] 11
attr(,"gradient")
     x y
[1,] 8 7
```

```r
f_grad(x = 2, y = 1)
```

```
[1] 11
attr(,"gradient")
     x y
[1,] 7 8
```

`deriv3()` - gradient *and* Hessian

`deriv3()` works exactly like `deriv()` but also computes the Hessian matrix (second derivatives), stored in a `"hessian"` attribute.

```
1 f_all = deriv3(
2   ~ x^2 + 3*x*y + y^2,
3   namevec = c("x", "y"),
4   function.arg = TRUE
5 )
6 f_all(x = 1, y = 2)
```

```
[1] 11
attr(,"gradient")
     x y
[1,] 8 7
attr(,"hessian")
, , x

     x y
[1,] 2 3

, , y

     x y
[1,] 3 2
```

Using `deriv()` with `optim()`

We can use `deriv()` to automatically supply gradients to `optim()`,

```
rosenbrock = deriv(
  ~ (1 - x)^2 + 100 * (y - x^2)^2,
  namevec = c("x", "y"),
  function.arg = TRUE
)
```

```
fn = function(par) {
  res = rosenbrock(x = par[1], y = par[2])
  attributes(res) = NULL
  res
}
```

```
gr = function(par) {
  res = rosenbrock(x = par[1], y = par[2])
  attr(res, "gradient")
}
```

```
optim(c(-1, 1), fn, gr, method = "BFGS")
```

```
$par
[1] 1 1

$value
[1] 5.769286e-17

$counts
function gradient
     117       51

$convergence
[1] 0

$message
NULL
```

What `D()` / `deriv()` support

These functions handle a specific set of R math operations:

Supported	Examples
Arithmetic	`+`, `-`, `*`, `/`, `^`
Exp / log	`exp`, `expm1`, `log`, `log1p`, `log2`, `log10`
Trig	`sin`, `cos`, `tan`, `sinpi`, `cospi`, `tanpi`, `asin`, `acos`, `atan`
Hyperbolic	`sinh`, `cosh`
Other math	`sqrt`, `gamma`, `lgamma`, `digamma`, `trigamma`, `psigamma`, `factorial`, `lfactorial`
Distribution	`dnorm`, `pnorm` (standard normal only)

Not supported - other `d*`/`p*`/`q*` distribution functions, control flow (`if`, `for`), matrix operations, or arbitrary R functions - for those you need numerical differentiation or autodiff packages.

MLE via `deriv()` - Normal

Since `deriv()` cannot differentiate `dnorm()` w.r.t. its `mean`/`sd` parameters, we write the log-density explicitly,

$$\log f(x|\mu,\sigma) = -\tfrac{1}{2}\log(2\pi) - \log\sigma - \frac{(x-\mu)^2}{2\sigma^2}$$

```
nll_norm_d = deriv(
  ~ -( -0.5 * log(2 * pi) - log(sigma) - (x - mu)^2 / (2 * sigma^2) ),
  namevec = c("mu", "sigma"),
  function.arg = c("x", "mu", "sigma")
)
```

```
fn = function(x, theta) nll_norm_d(x, theta[1], theta[2]) |> sum()
gr = function(x, theta) nll_norm_d(x, theta[1], theta[2]) |> attr("gradient") |> colSums()
```

```r
(res = optim(c(0, log(1)),
  fn = \(x, t) fn(x, c(t[1], exp(t[2]))),
  gr = \(x, t) gr(x, c(t[1], exp(t[2]))),
  method = "BFGS",
  x = x_norm
))
```

```
$par
[1] -3.3959525  0.2225137

$value
[1] 164.1453

$counts
function gradient
      43       12

$convergence
[1] 0

$message
NULL
```

```r
c(mu = res$par[1], sigma = exp(res$par[2]))
```

```
       mu     sigma
-3.395953  1.249213
```

MLE via `deriv()` - Gamma

`dgamma()` is not in the derivatives table either, so we write the log-density explicitly,

$$\log f(x|\alpha,\beta) = \alpha \log \beta - \log \Gamma(\alpha) + (\alpha - 1)\log x - \beta x$$

```
nll_gamma_d = deriv(
  ~ -( shape * log(rate) - lgamma(shape) + (shape - 1) * log(x) - rate * x ),
  namevec = c("shape", "rate"),
  function.arg = c("x", "shape", "rate")
)
```

```
fn = function(x, theta) nll_gamma_d(x, theta[1], theta[2]) |> sum()
gr = function(x, theta) nll_gamma_d(x, theta[1], theta[2]) |> attr("gradient") |> colSums()
```

```r
(res = optim(c(0, 0),
  fn = \(x,t) fn(x, exp(t)),
  gr = \(x,t) gr(x, exp(t)),
  method = "BFGS",
  x = x_gam
))
```

```
$par
[1] 1.0732192 0.6997555

$value
[1] 226.3538

$counts
function gradient
      38        9

$convergence
[1] 0

$message
NULL
```

```r
exp(res$par)
```

```
[1] 2.92478 2.01326
```