Lab: Linear Regression by Maximum Likelihood

answer key

OCEANS 143 - Winter 2024

Learning Goal

In this lab, we'll step through the process of fitting a linear regression model from scratch - by using a simple model as an example, we'll learn the basics of maximum likelihood estimation so that we can apply the approach to more complex models down the line. Everything we'll do in this lab can be done in a single line using the R function `lm`, but by peeking inside the black box, we can learn how to extend the approach when pre-packaged options aren't available.

Simulating and plotting data

In the first part of this lesson, we'll simulate data from a linear model using a known intercept and slope. Then, we'll **develop our own function** to fit a linear regression using **maximum-likelihood estimation**, and see how our estimated intercept and slope compare to the known values.

Recall that in linear regression, the errors are assumed to be normally distributed with variance σ^2 around the best fit regression line:

$$
\begin{aligned}
Y &= \alpha + \beta X + \epsilon \\
\epsilon &\sim \text{Normal}(0, \sigma^2)
\end{aligned}
$$

$$
\begin{aligned}
Y &\sim \text{Normal}(\mu, \sigma^2) \\
\mu &= \alpha + \beta X
\end{aligned}
$$

As you can see, there are **three** parameters in the basic linear regression model: α, the intercept, β, the slope, and σ, the standard deviation of the residuals.

First, let's choose values for the intercept, slope, and residual variance - feel free to replace the values below with ones of your own choice.

```
a <- 1 ## intercept
b <- 2 ## slope
sigma <- 1 ## error standard deviation
```

Next, let's simulate data from a linear regression with these known parameters.

- We'll use the `runif` function, which generates random values from a uniform distribution, to generate random `x` values.
- We'll use the `rnorm` function to generate the errors, which is built into base R and generates random observations from a normal distribution.

- We'll store these values in a `tibble`, which is a special type of data frame, for easy plotting with the `ggplot2` package.

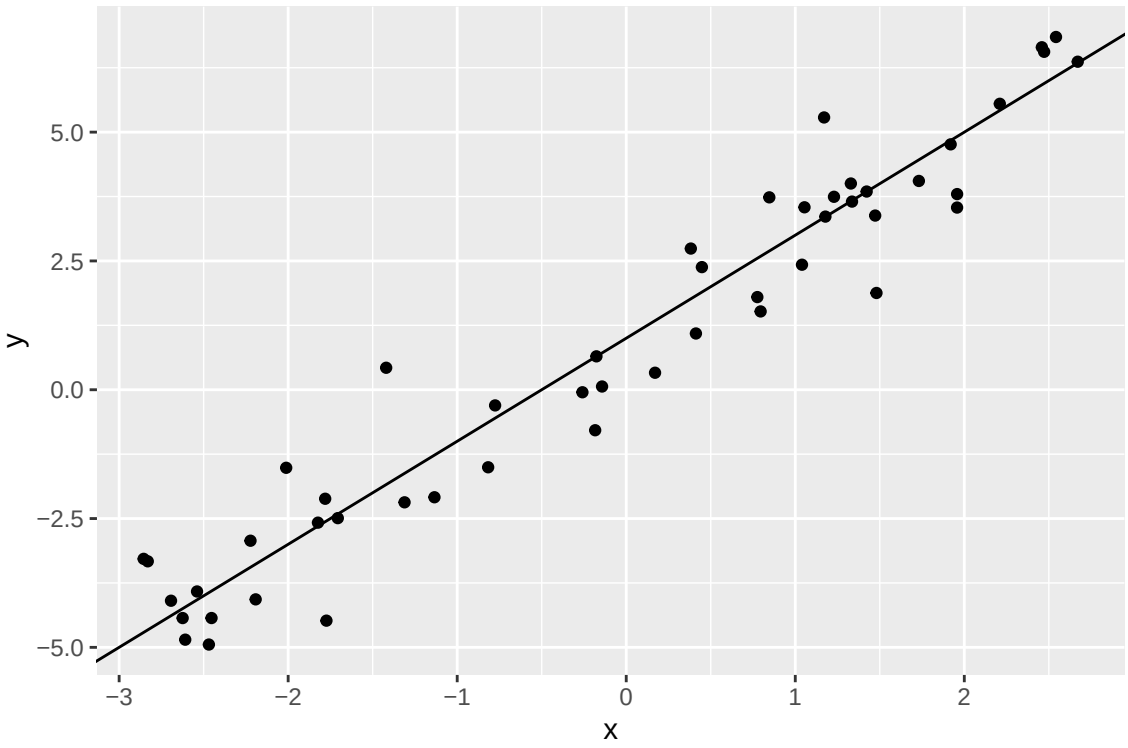
```
n <- 50 ## sample size

## simulate data
sim_data <- data.frame(x = runif(n, min = -3, max = 3))
sim_data$y <- rnorm(n, mean = a + b*sim_data$x, sd = sigma)

## Look at the data structure
head(sim_data)
```

```
##            x         y
## 1  1.4728670  3.380767
## 2  1.2290655  3.745106
## 3  0.8465253  3.734160
## 4 -2.5390409 -3.915400
## 5 -2.4690302 -4.945803
## 6  0.4476555  2.380175
```

Next, let's plot this data. A few things to know about plotting with `ggplot2` in R:

- Always start with a **data frame**, which is "piped" (`%>%`) to the `ggplot` function
- Specify things that depend on the data (the **aesthetics**), such as the `x` and `y` values by passing these values wrapped in `aes()` as the first argument to the `ggplot` function
- Add **geometries**, such as points, bar charts, and error bars, using `geom_` functions (`geom_point`, `geom_bar`, `geom_errorbar`, etc.).

```
sim_data %>% # start with the data
  ggplot(aes(x, y)) + # tell ggplot which aesthetics to use
  geom_point() + # add points
  geom_abline(intercept = a, slope = b) # show the known regression line
```



Finding the best-fit line

To find the best fit line, we want to maximize the **likelihood function**, which is the product of the probability density of each of our data points, assuming a distribution (in this case, the Normal distribution) and parameter values for that distribution θ (in this case, $\theta = \{\alpha, \beta, \sigma\}$):

$$\mathcal{L}(\theta) = f_X(x|\theta) = \prod_{i=1}^{n} f_X(x_i)$$

This is equivalent to maximizing the log-likelihood function, which is computationally easier:

$$l(\theta) = \ln(\mathcal{L}(\theta)) = \sum_{i=1}^{n} \ln[f_X(x_i)]$$

It's also equivalent to **minimizing the negative log-likelihood**, which is what we'll do here.

To do this, we need to express the log-likelihood of the **y values** in R as a function of the parameters. We can switch around some terms in the regression equation above to help with this, so that:

$$\begin{aligned} Y &= \alpha + \beta X + \epsilon \\ \epsilon &\sim \text{Normal}(0, \sigma^2) \end{aligned}$$

becomes:

$$Y \sim \text{Normal}(\underbrace{\alpha + \beta X}_{\text{mean}}, \underbrace{\sigma^2}_{\text{variance}})$$

So, given our parameters α, β, and σ, the likelihood of our y-values is given by summing the log density of each of our y-values under a Normal distribution with mean $\alpha + \beta X$ and standard deviation σ.

Luckily, the Normal density function is already coded into `R` as the `dnorm` function, so we'll write a function for the negative log-likelihood using that. We can express the negative log-likelihood function very simply in R as:

```
-sum(dnorm(y, mean = ..., sd = ..., log = TRUE))
```

Where we'll provide `a + b*x` for the `mean`, and `sigma` for the `sd`. Since the standard deviation, σ, must be positive, we'll estimate it on the log scale. The `optim` function expects all of our parameters to be passed to our objective function (i.e., the negative log likelihood) as a single argument, so we'll pass them as a vector argument called `par` and unpack them in the likelihood function:

```
## negative log-likelihood of y-values given intercept, slope, and error sd
## we pass the parameters as a vector in the order a, b, sigma
lm_nll <- function(par, x, y) {
  a <- par["a"]; b <- par["b"]; sigma <- exp(par["log_sigma"])
  -sum(dnorm(y, mean = a + b*x, sd = sigma, log = TRUE))
}
```

Now, let's pass the `lm_nll` function to the `optim` function to get out a list of parameter values which minimize the negative log likelihood, and are therefore the most probable values given the data we've observed:

```
fit <- optim(
  par = c(a = 0, b = 0, log_sigma = 1),
  fn = lm_nll,
  x = sim_data$x,
  y = sim_data$y,
  hessian = TRUE
)

fit
```

```
## $par
##           a           b   log_sigma
##  0.92243185  1.96277412 -0.09423335
##
## $value
## [1] 66.2316
##
## $counts
## function gradient
##      148       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##                     a           b   log_sigma
```

```
## a         60.36984343  -5.24355260 -0.02396288
## b         -5.24355260 183.93648092 -0.03931643
## log_sigma -0.02396288  -0.03931643 99.98550030
```

We can see that our values for the intercept, slope, and standard deviation of the parameters are pretty close to the known values. We can also compare the intercept and slope to the values we get out from R's built-in linear regression function, `lm`:

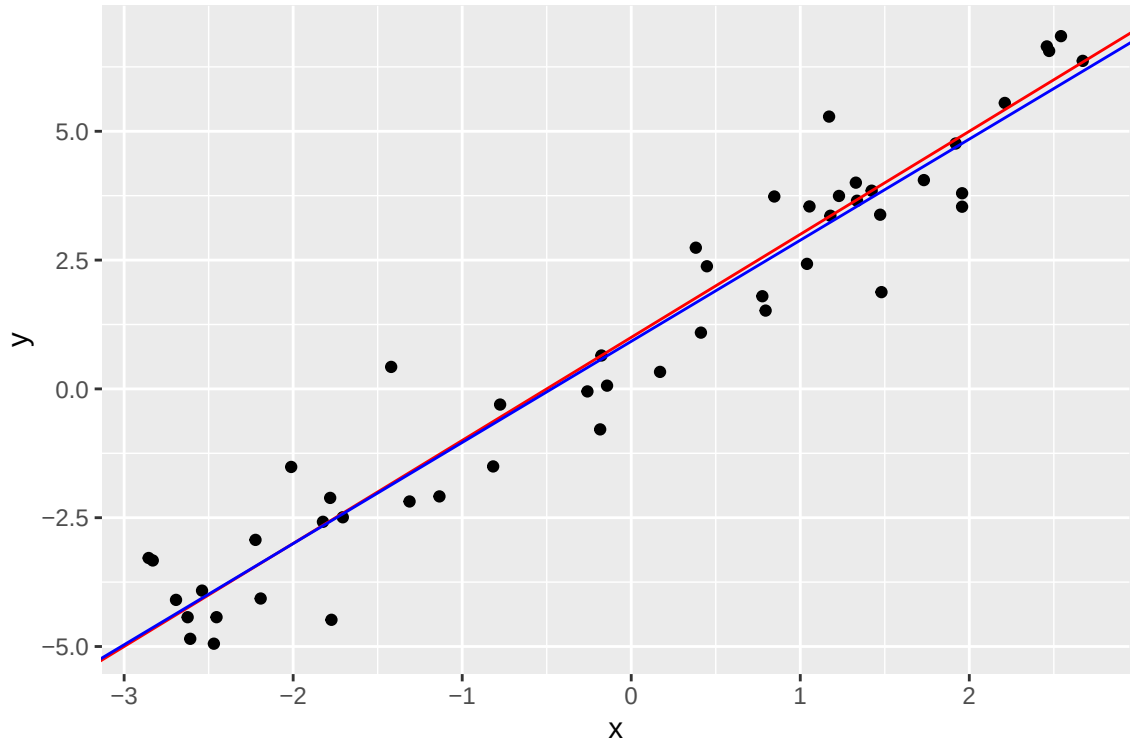
```
coef(lm(y ~ x, data = sim_data))
```

```
## (Intercept)           x
##   0.9222236   1.9626613
```

Assuming everything is working fine, these numbers should match almost exactly.

Let's compare our estimates with the known values of the intercept and slope by plotting both lines on our previous plot:

```
sim_data %>%
  ggplot(aes(x, y)) + # tell ggplot which aesthetics to use
  geom_point() + # add points
  geom_abline(intercept = a, slope = b, color = "red") + # true regression line
  geom_abline(intercept = fit$par[1], slope = fit$par[2], color = "blue") # fit
```



Likelihood surfaces and profiles

One easy way to check that our optimization algorithm is working right is to plot the likelihood (or negative log-likelihood) as a function of the parameters around the maximum likelihood estimates - if our model has

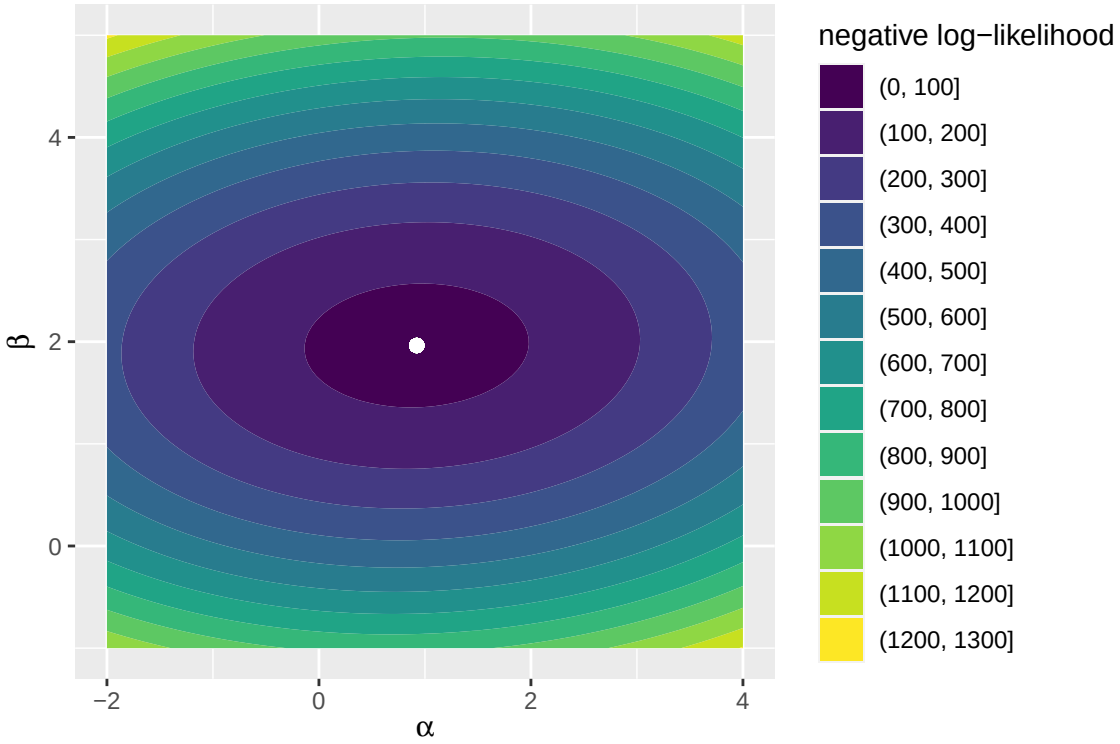
converged, we should see that the maximum likelihood estimates occur in a "valley." For a model with one parameter, we call this a **likelihood curve** - for two parameters, this is a **likelihood surface**.

In the case of our linear regression model, we have three parameters - α, β, and σ. In order to plot a likelihood surface for the intercept and slope, we need to fix (i.e. hold constant) the value of our standard deviation parameter. When we plot a likelihood curve or surface, fixing the other parameter values, we call this a **likelihood slice**. Let's do that here by just fixing σ at it's maximum-likelihood estimate:

```
a_seq <- seq(-2, 4, length.out = 100)
b_seq <- seq(-1, 5, length.out = 100)

### data frame containing all combinations of the intercept and slope values
par_grid <- expand.grid("a" = a_seq, "b" = b_seq)

### loop over data frame, store negative log likelihood
for (i in 1:nrow(par_grid)) {
  par_grid$loglik[i] <- lm_nll(
    c(a = par_grid$a[i], b = par_grid$b[i], fit$par[3]),
    x = sim_data$x,
    y = sim_data$y
  )
}
```

Let's make a contour plot to visualize the likelihood surface for α and β, along with a point where our maximum likelihood estimate is:

```
par_grid %>%
  ggplot(aes(a, b, z = loglik)) +
  geom_contour_filled() +
  labs(x = expression(alpha), y = expression(beta), fill = "negative log-likelihood") +
  geom_point(aes(x = fit$par[1], y = fit$par[2]), size = 2, color = "white")
```



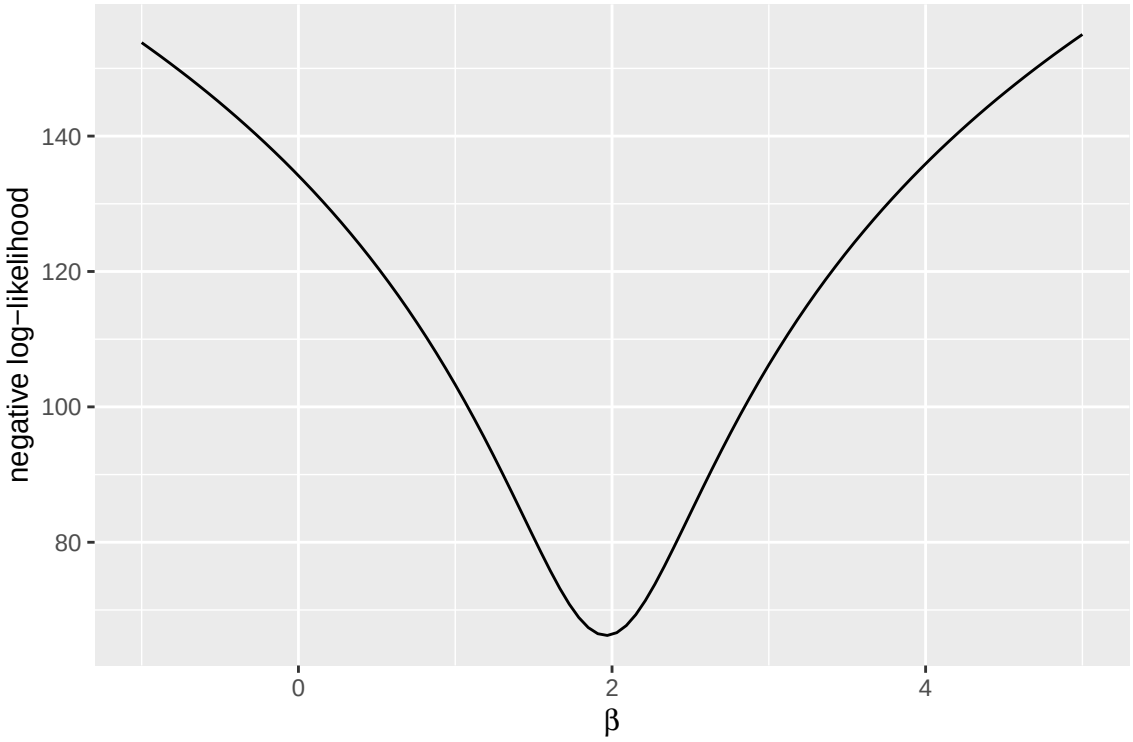
We can see that the maximum-likelihood estimate occurs at the local minimum of our likelihood slice, and that our likelihood surface appears to be a smooth function of our intercept and slope parameters, which is good. For a likelihood surface from a one- or two-parameter model, this should always be the case (and, it happens to also be the case for linear regression, assuming that the standard deviation is that parameter that we're fixing) - but for more complex models, a likelihood slice might give misleading results.

When we have more parameters, we can use **likelihood profiles** for each of our parameters. To calculate a likelihood profile, we set a focal parameter to a range of values, and for each value we optimize the likelihood function with respect to all of the other parameters, storing the resulting negative log-likelihood for each value of the focal parameter.

Let's do this for just the slope, to show how it works - the procedure is the same for any other parameter:

```
## Modify our `lm_log_lik` function so that the slope is a separate argument
## We'll pass this to `optim` to just find the values of the intercept and sd
slope_proflik <- function(pars, b, x, y) {
  a <- pars["a"]; sigma <- exp(pars["log_sigma"]) ## only two pars in this function
  -sum(dnorm(y, mean = a + b*x, sd = sigma, log = TRUE))
}

b_profile <- data.frame(
  b = seq(-1, 5, length.out = 100)  ## values of slope to loop over
)

## loop over values of the slope, store negative log-likelihood
for (i in 1:nrow(b_profile)) {
  b_fit <- optim(
    par = c(a = 0, log_sigma = log(1)), # initial values for intercept and sd
    fn = slope_proflik,  # objective function
    b = b_profile$b[i], # fixed slope value
```

```
    x = sim_data$x,    # x values
    y = sim_data$y    # y values
  )
  b_profile$nll[i] <- b_fit$value
}

## plot slope profile
b_profile %>%
  ggplot(aes(b, nll)) +
  geom_line() +
  labs(x = expression(beta), y = "negative log-likelihood")
```



In a more complex model, we could repeat this procedure for each of the parameters in our model - if we had more than a few, we'd probably want to write a function or loop to do so. The `bblme` and `stats4` packages also provide some useful functions to do this.

Standard Errors

So, we have the maximum-likelihood estimate for our intercept and slope, but what if we want to estimate the uncertainty in these parameters - e.g., their standard error and confidence intervals? We can actually do this using the likelihood profiles that we've just computed, but it can be tedious and inefficient when we have more than a couple parameters. We have a few more options:

- Derive the standard deviation of the parameters analytically (recall that the standard error is the standard deviation of an estimate, such as the intercept and slope) - this is straightforward for the intercept and slope of linear regression, but can be intractable for more complex models, so we won't cover it.

- Use the observed Fisher information, which relates the standard deviation of the parameters to the curvature of the likelihood function around the estimates. Likelihood functions that are more curved, and therefore "sharper" around the estimate, indicate greater certainty in the estimate and therefore a smaller standard deviation. To generate confidence intervals, we assume the parameters are normally distributed, where the mean is our maximum-likelihood estimate and the SD is calculated using Fisher information.
- Bootstrap the estimates by resampling with replacement from the original data and re-obtaining estimates for the intercept and slope. This is more computationally intensive than using the observed Fisher information, but can be used to approximate the distribution of our parameters without assuming they are normally distributed. Additionally, we can use the bootstrap when we're fitting parameters by something other than maximum-likelihood - like, for example, least squares (see the lab exercises).

Here, we'll estimate the standard deviation of the parameters using both the observed Fisher information and bootstrapping.

Fisher Information

When we obtained our maximum-likelihood estimates above, we asked the `optim` function to return the Hessian matrix (`hessian = TRUE`). The Hessian is the matrix of the second-order partial derivatives, and in this case, it's the matrix of the second-order partial derivates of the negative log-likelihood function with respect to our parameters. It measures the curvature of our likelihood function around the maximum-likelihood estimates, and with it we can estimate the uncertainty in the maximum-likelihood estimates themselves.

The variance in our parameter estimates is approximately equal to the inverse of the observed fisher information matrix:

$$\mathcal{J}(\hat{\theta}) = -\frac{\partial^2}{\partial \theta^2} \ln[\mathcal{L}(\hat{\theta})]$$
$$\text{Var}(\hat{\theta}) = 1/\mathcal{J}(\hat{\theta})$$

Because we're minimizing the negative log-likelihood, the Hessian matrix returned by our call to `optim` *is* the observed Fisher Information. To obtain the variance-covariance matrix of our parameters, all we have to do is invert the matrix with `solve`:

```
var_cov <- solve(fit$hessian)
var_cov
```

```
##                       a            b    log_sigma
## a         1.660568e-02 4.733858e-04 4.165922e-06
## b         4.733858e-04 5.450155e-03 2.256571e-06
## log_sigma 4.165922e-06 2.256571e-06 1.000145e-02
```

The variances are on the diagonal, and we can get the standard errors by taking their square root:

```
fisher_se <- sqrt(diag(var_cov))
fisher_se
```

```
##          a          b  log_sigma
## 0.12886303 0.07382516 0.10000726
```

Let's compare these to the standard errors reported by the `lm` function:

```
summary(lm(y ~ x, data = sim_data))
```

```
##
## Call:
## lm(formula = y ~ x, data = sim_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.94950 -0.64468  0.07526  0.50059  2.29191
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.92222    0.13151   7.013 7.03e-09 ***
## x            1.96266    0.07534  26.050  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9288 on 48 degrees of freedom
## Multiple R-squared:  0.9339, Adjusted R-squared:  0.9326
## F-statistic: 678.6 on 1 and 48 DF,  p-value: < 2.2e-16
```

Bootstrapping

Bootstrapping is a simple but powerful technique, which relies on the assumption that the data themselves are representative of the population that we're sampling from, and that we can therefore resample from the data (with replacement) to obtain a new, and unique, representative sample. By estimating the properties of each of our bootstrap samples, we can gauge the variation in the larger population.

Bootstrapping works best when the number of samples we have is relatively large, and when we can resample from the data many times - this later condition is only a matter of compute time.

So, let's try it - we'll take 1000 bootstrap samples from the data, estimate the intercept and slope for each of them, and then take the standard deviation of these estimates.

```
n_obs <- nrow(sim_data) ## number of observations
n_samples <- 1000 ## number of bootstrap samples

## initialize an empty matrix to store parameter estimates for each sample
par_samples <- matrix(nrow = n_samples, ncol = 3)

## recalculate MLE for each sample
for (i in 1:n_samples) {
  sample_rows <- sample(1:n_obs, size = n_obs, replace = TRUE) # sampled rows from data
  new_data <- sim_data[sample_rows,] # subset data to sampled rows

  sample_fit <- optim(
    par = c(a = 0, b = 0, log_sigma = 0), # initial values
    fn = lm_nll, # log likelihood function
    x = new_data$x, # sampled x-values
    y = new_data$y # sampled y-values
  )
```

```
  par_samples[i,] <- sample_fit$par # store sample parameters
}

boot_se <- apply(par_samples, MARGIN = 2, FUN = sd) # calculate column SDs
boot_se
```

```
## [1] 0.12896631 0.07416402 0.10228865
```

As expected, these standard deviations look pretty similar to our estimates based on Fisher information.

Confidence Intervals

Normal approximation

If we assume that the maximum likelihood estimate is normally distributed with mean $\mu = \hat{\theta}$ and standard deviation $\sigma = \text{SE}(\hat{\theta})$, we can calculate $1 - \gamma$ (I use γ to avoid re-using α, which is our slope) confidence intervals as:

$$\text{CI} = (\hat{\theta} - \phi^{-1}(\gamma/2) \times \text{SE}(\hat{\theta}),\ \hat{\theta} + \phi^{-1}(1 - \gamma/2) \times \text{SE}(\hat{\theta}))$$

Where ϕ^{-1} is the inverse cumulative distribution function of the standard Normal distribution. Equivalently:

$$\text{CI} = (\phi^{-1}_{\hat{\theta},\text{SE}(\theta)}(\gamma/2),\ \phi^{-1}_{\hat{\theta},\text{SE}(\theta)}(1 - \gamma/2))$$

Where $\phi^{-1}_{\hat{\theta},\text{SE}(\theta)}$ is the inverse cumulative distribution function for a Normal distribution with mean $\hat{\theta}$ and standard deviation $\text{SE}(\theta)$.

The Normal inverse CDF is returned by the `qnorm` function. We can obtain a confidence interval for our intercept α and slope β:

```
gamma <- 0.05 ## 95% confidence interval

alpha_CI <- qnorm(c(gamma/2, 1 - gamma/2), mean = fit$par[1], sd = fisher_se[1])
beta_CI <- qnorm(c(gamma/2, 1 - gamma/2), mean = fit$par[2], sd = fisher_se[2])

cbind(alpha_CI, beta_CI)
```

```
##      alpha_CI  beta_CI
## [1,] 0.669865 1.818079
## [2,] 1.174999 2.107469
```

We can also use the standard errors we obtained from bootstrapping:

```
alpha_CI <- qnorm(c(gamma/2, 1 - gamma/2), mean = fit$par[1], sd = boot_se[1])
beta_CI <- qnorm(c(gamma/2, 1 - gamma/2), mean = fit$par[2], sd = boot_se[2])

cbind(alpha_CI, beta_CI)
```

```
##       alpha_CI  beta_CI
## [1,] 0.6696625 1.817415
## [2,] 1.1752012 2.108133
```

Bootstrap distribution

We can also obtain confidence intervals from the bootstrap distribution directly, using the `quantile` function. If we were to do this for an actual study, it'd probably be better to take at least 10,000 samples - 1,000 isn't much.

```
alpha_CI <- quantile(par_samples[,1], c(gamma/2, 1 - gamma/2))
beta_CI <- quantile(par_samples[,2], c(gamma/2, 1 - gamma/2))

cbind(alpha_CI, beta_CI)
```

```
##        alpha_CI  beta_CI
## 2.5%  0.6700277 1.810123
## 97.5% 1.1781475 2.104601
```

Making it a function

Just for fun, let's go ahead and toss all the things we've learned into a function that returns estimated parameter values for the intercept and slope. We can then apply this function to any new dataset of our choosing, just like the `lm` function built into base R.

You'll also find that building things into functions makes them easier to read and reuse, and if you want to apply the same model to mulitple datasets or share the code so others can do so on theirs, having your code built into a function is very useful!

Here's the code for the function - there aren't really any new elements here, we've just taken the old elements and put them together.

```
lm_fit <- function(x, y, SE = "fisher", init = c("a" = 0, "b" = 0, "log_sigma" = 0), n_boot = 1000) {

  if (!(SE %in% c("bootstrap", "fisher"))) stop("SE options are bootstrap or fisher")
  if (length(x) != length(y)) stop("x and y lengths differ")

  ## Maximum-likelihood estimate
  MLE <- optim(
    par = init, # initial values
    fn = lm_nll,  # our negative log-likelihood function
    x = x,   # x values
    y = y,   # y values
    hessian = (SE == "fisher") # only return Hessian if Fisher information used
  )

  ## Standard error using either fisher information or bootstrapping
  if (SE == "fisher") {
    var_cov <- solve(MLE$hessian)
    fit_se <- sqrt(diag(var_cov))
  } else {
    n_obs <- length(x) ## number of observations

    ## initialize an empty matrix to store parameter estimates for each sample
    par_samples <- matrix(nrow = n_boot, ncol = 3)

    ## recalculate MLE for each sample
```

```
    for (i in 1:n_boot) {
      samp <- sample(1:n_obs, size = n_obs, replace = TRUE) # sampled observations
      new_x <- x[samp] # subset x to sampled observations
      new_y <- y[samp] # subset y to sampled observations

      sample_fit <- optim(
        par = init, # initial values
        fn = lm_nll, # log likelihood function
        x = new_x, # sampled x-values
        y = new_y # sampled y-values
      )

    par_samples[i,] <- sample_fit$par # store sample parameters
  }
  fit_se <- apply(par_samples, MARGIN = 2, FUN = sd) # calculate column SDs
 }

 ## nicely format output
 data.frame(
   coef = c("intercept", "slope"),
   estimate = MLE$par[1:2],
   SE = fit_se[1:2]
 )

}
```

Lab Exercises

1. Testing our function with new parameters

Now, play around with simulating new data - can you recover the parameters that you simulated the data from? What happens to the estimates and their SE when you decrease the sample size of the simulated data? Do the Fisher information and bootstrap methods agree on small datasets? On large ones?

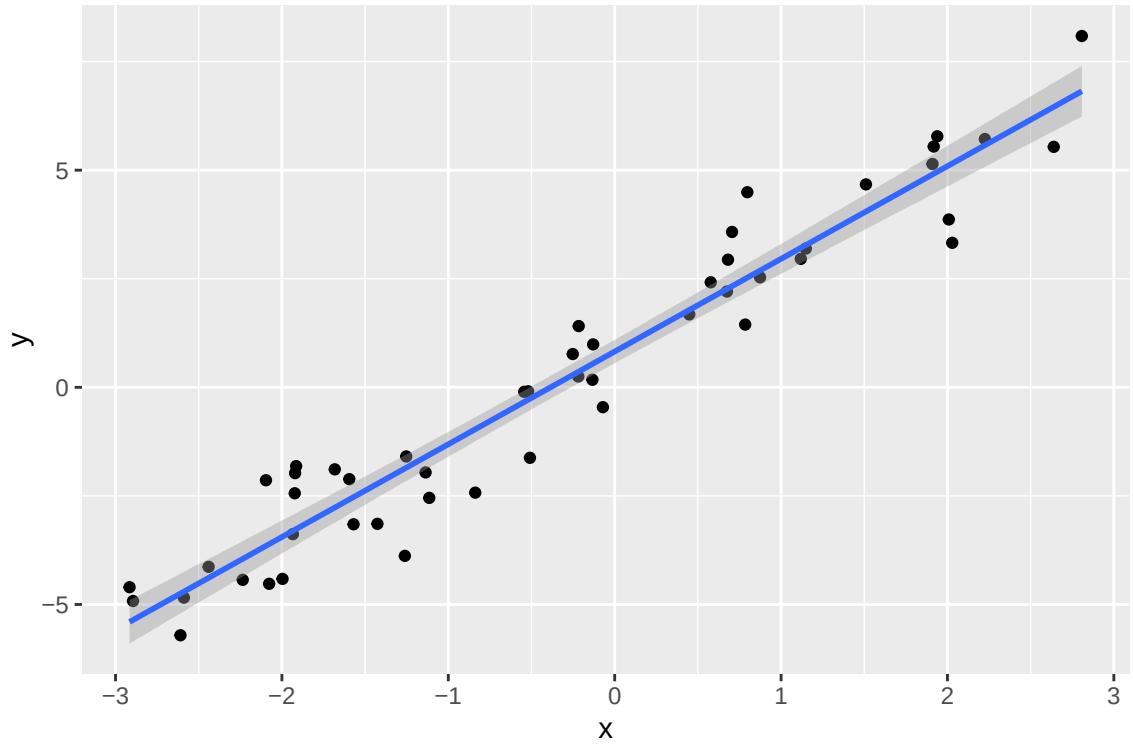
The code to simulate new data is reproduced for you below. Try varying the sample size and the parameters - and make sure to plot the x and y variables against each other first!

> **As we increase the sample size, the estimates do not change in any predictable way, but the variation in estimates across simulations does decrease - estimates are, on average, closer to the true value when we've simulated many observations. The standard error decreases as we increase the sample size - with more data, we are more certain about our maximum likelihood estimates.**

```
a <- 1 ## intercept
b <- 2 ## slope
sigma <- 1 ## error standard deviation
n <- 50 ## sample size

## simulate data
sim_data <- tibble(
  x = runif(n, min = -3, max = 3), ## x values
  y = rnorm(n, mean = a + b*x, sd = sigma) ## regression equation
```

```
)

## plot data
sim_data %>% ggplot(aes(x, y)) + geom_point() + geom_smooth(method = "lm")
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



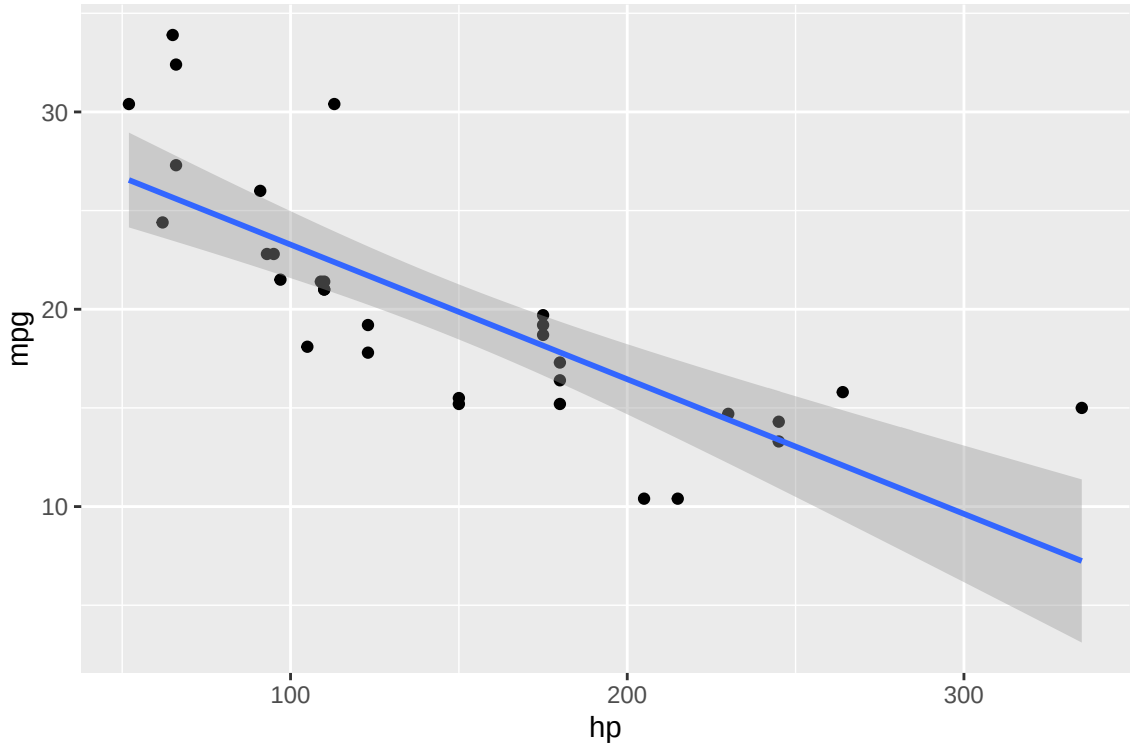

```
## run regression
lm_fit(sim_data$x, sim_data$y, SE = "fisher")
```

```
##        coef  estimate         SE
## a intercept 0.8232013 0.13135599
## b     slope 2.1350759 0.08024952
```

2. Trying it out on new data

Try out our regression function on some of the other datasets built into R - try, for example, building regressions using the variables in the `mtcars` or `iris` datasets. You can view all the datasets you have available by running the `data()` function.

```
mtcars %>% ggplot(aes(hp, mpg)) + geom_point() + geom_smooth(method = "lm")
```




```
with(mtcars, lm_fit(hp, mpg))
```

```
##        coef    estimate          SE
## a intercept 30.09775644 1.581525699
## b     slope -0.06818078 0.009794813
```

3. Adding a predictor

Modify the `lm_nll` function we wrote earlier to fit a multiple regression - i.e., a regression with more than one predictor / x variable. Pass this to `optim` and obtain estimates for the intercept, slope on the first x variable, and slope on the second x variable - make sure you can approximately recover the true parameter values used to simulate the data. **2 pt.**

Note: Rename this new negative log-likelihood function, otherwise it'll mess up the code in the answers after this one.

```
n <- 50 ## sample size

## Parameters - change these and make sure you can recover them from the model
a <- 1 ## intercept
b1 <- 2 ## first variable's slope
b2 <- -1 ## second variable's slope
sigma <- 1 ## error standard deviation

## simulate data
sim_data2 <- data.frame(x1 = runif(n, min = -3, max = 3), x2 = runif(n, min = -3, max = 3))
sim_data2$y <- rnorm(n, mean = a + b1*sim_data2$x1 + b2*sim_data2$x2, sd = sigma)
```

```r
## Define a new negative log-likelihood function
mlm_nll <- function(par, x1, x2, y) {
  a <- par["a"]; b1 <- par["b1"]; b2 <- par["b2"]; sigma <- exp(par["log_sigma"])
  -sum(dnorm(y, mean = a + b1*x1 + b2*x2, sd = sigma, log = TRUE))
}

## Use optim to obtain maximum likelihood estimates
optim(
  par = c(a = 0, b1 = 0, b2 = 0, log_sigma = 1),
  fn = mlm_nll,
  x1 = sim_data2$x1,
  x2 = sim_data2$x2,
  y = sim_data2$y
)
```

```
## $par
##           a          b1          b2   log_sigma
##  1.06519993  1.95218606 -1.09692851 -0.03393036
##
## $value
## [1] 69.25425
##
## $counts
## function gradient
##      213       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Fit this same model again, but make `x1` and *exact* duplicate of `x2`. Re-run the `optim` code a couple times, changing the initial parameter values when you do. Each time, note the estimated slope on `x1` and `x2`, as well as the sum of the slopes on `x1` and `x2`. What happens? Why do you think this happens? **1 pt.**

> **Each time we re-run `optim`, we get different slopes for `x1` and `x2` - that's because if the covariates are exact duplicates of each other, they do not have separable effects. This is an extreme version of something called *colinearity* - when the predictors are highly correlated, it degrades our estimates of their unique effects.**

```r
(par <- optim(
  par = c(a = 0, b1 = 0.1, b2 = 0, log_sigma = 1),
  fn = mlm_nll,
  x1 = sim_data$x1,
  x2 = sim_data$x1,
  y = sim_data$y
)$par)
```

```
## Warning: Unknown or uninitialised column: `x1`.
## Unknown or uninitialised column: `x1`.
```

```
##         a         b1         b2 log_sigma
##       0.0       0.1        0.0        1.0
```

```
par["b1"] + par["b2"]
```

```
##  b1
## 0.1
```

4. Likelihood-ratio tests

While our linear regresion function is great, you've probably noticed a couple things that it's missing which R's `lm` function has - notably, our function doesn't spit out any p-values. The p-values that the `lm` function returns for the intercept and slope are testing the null hypothesis that intercept and slope are 0 - we can test the same null hypothesis, and many other ones that we might be interested in, using the likelihood function.

Although the `lm` function uses the t distribution to calculate p-values, we're going to use **likelihood-ratio tests**. Likelihood ratio tests are great because we can use them to test a flexible variety of null hypothesis - maybe our null expectation for the slope is 2, or maybe we want to test the joint null hypothesis that **both** the intercept and slope are zero. Maybe we're fitting a logistic population growth model, and we want to test whether the intrinsic growth rate r is significantly different from 0.05 or some other hypothesized value - all of these are things we can evaluate with a likelihood ratio test.

We won't go into the theoretical/mathematical justification for the likelihood ratio test, but here's how to do it:

1. Fit the model, allowing all parameters to vary, and extract the negative log-likelihood at the maximum-likelihood estimate. We'll call this the **full model**.
2. Now, fit the model again, but fix (hold constant) one or more of the parameters to a null value. We'll call this the **reduced model**.
3. Compute the χ^2 ("Chi-square") test statistic: $2(-\ln \mathcal{L}_{\text{reduced}} - (-\ln \mathcal{L}_{\text{full}}))$ - i.e. 2 times the difference in negative log-likelihoods between the reduced and full models.
4. Calculate the probability of observing a test statistic as or more extreme than the above value assuming a Chi-square distribution with degrees of freedom equal to the number of fixed parameters (in R, this is the `pchisq` function with the argument `lower.tail = FALSE`).

Do this below and calculate a p-value for the null hypothesis that the slope is 0 (representing no relationship between y and x) using the `anscombe$x1` and `anscombe$y1` variables. Compare the p-value you get here from the one the `lm` function returns. If you use the `slope_proflik` function we defined earlier in obtaining your reduced negative log-likelihood, explain why this works (but this isn't the only way!).

> **Here, I used the `slope_proflik` function we defined earlier for creating a likelihood profile to obtain the negative log-likelihood for the "reduced" model - this works because this function does exactly what we outlined in step 2 of the likelihood ratio test procedure - it fits the model, holding a paramter (the slope) constant. The likelihood ratio test p-value is extremely small, so there's significant evidence (under a threshold significance level of $\alpha = 0.05$) that the slope of the relationship between `x1` and `y1` in the `anscombe` dataset is different than zero. While the p-value that `lm` returns is a little higher, both are quite small.**

```r
### full negative log-likelihood
full <- optim(par = c(a = 0, b = 0, log_sigma = 0), fn = lm_nll, x = anscombe$x1, y = anscombe$y1)
full <- full$value

### reduced negative log-likelihood
reduced <- optim(par = c(a = 0, log_sigma = 0), fn = slope_proflik, b = 0, x = anscombe$x1, y = anscombe
reduced <- reduced$value

### calculate test statistic
chisq <- 2 * (reduced - full)

### likelihood ratio test p-value
p_LRT <- pchisq(chisq, df = 1, lower.tail = FALSE)

### p-value from lm
p_lm <- anova(lm(y1 ~ x1, data = anscombe))$`Pr(>F)`[1]

paste("likelihood ratio p-value:", round(p_LRT, 4))
```

```
## [1] "likelihood ratio p-value: 5e-04"
```

```r
paste("lm p-value:", round(p_lm, 4))
```

```
## [1] "lm p-value: 0.0022"
```

5. Least-squares

In all of the above exercises, we've been using maximum-likelihood estimation, which always requires assuming a generative probability distribution for the data (in the case of linear regression, a Normal distribution). We also tried two methods for estimating the standard error of our parameters - one parametric (the Fisher information) and another non-parametric (bootstrapping).

We can also obtain parameter estimates without assuming a probability distribution, however, and obtain standard error estimates with bootstrapping (Fisher information only works when we have a likelihood function). All we have to do is change the **objective function** that we'd like to minimize from the likelihood function to something else - the **sum of squared errors** is a very popular option:

$$\text{SS} = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

In the case of linear regression, $f(x_i) = \alpha + \beta x_i$. Modify the `lm_nll` function we wrote earlier to return the sum of squares for a given intercept and slope, and find the values of the intercept and slope that minimize it (the "**least squares**" estimate). Compare these to the estimates obtained by maximum likelihood - what do you notice?

> **The least squares estimate is nearly identical to the maximum likelihood estimate. In general, this will not be the case, *except* when we use a normal likelihood function - it turns out that if we assume a normal distribution for the response, maximum likelihood and least squares will provide the same parameter estimates.**

```r
### WRITE THE SUM OF SQUARES FUNCTION ----------------------------------
## sum of squared errors given intercept and slope
## pass the parameters as a vector c(a, b)
sum_sqrs <- function(pars, x, y) {
  a <- pars[1]; b <- pars[2]
  sum((y - (a + b * x))^2)
}

### SIMULATE DATA ------------------------------------------------------

a <- 1 ## intercept
b <- 2 ## slope
sigma <- 1 ## error standard deviation
n <- 50 ## sample size

## simulate data
sim_data <- tibble(
  x = runif(n, min = -3, max = 3), ## x values
  y = rnorm(n, mean = a + b*x, sd = sigma) ## regression equation
)

### OBTAIN PARAMETER ESTIMATES USING SUM OF SQUARES ---------------------
ss_fit <- optim(
  par = c(0, 0), # initial values (a, b)
  fn = sum_sqrs,  # sum of squares function
  x = sim_data$x,   # x values
  y = sim_data$y   # y values
)

ss_fit
```

```
## $par
## [1] 1.039046 2.161070
##
## $value
## [1] 56.01988
##
## $counts
## function gradient
##       73       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```r
### OBTAIN PARAMETER ESTIMATES USING MAXIMUM LIKELIHOOD ------------------
ml_fit <- optim(
  par = c(a = 0, b = 0, log_sigma = 0), # initial values (a, b, sigma)
  fn = lm_nll,  # negative log-likelihood function
  x = sim_data$x,   # x values
  y = sim_data$y   # y values
```

```
)

ml_fit
```

```
## $par
##          a          b  log_sigma
## 1.03894079 2.16094152 0.05659876
##
## $value
## [1] 73.78902
##
## $counts
## function gradient
##      182       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```