Model fitting

Katie Schuler

2024-10-22

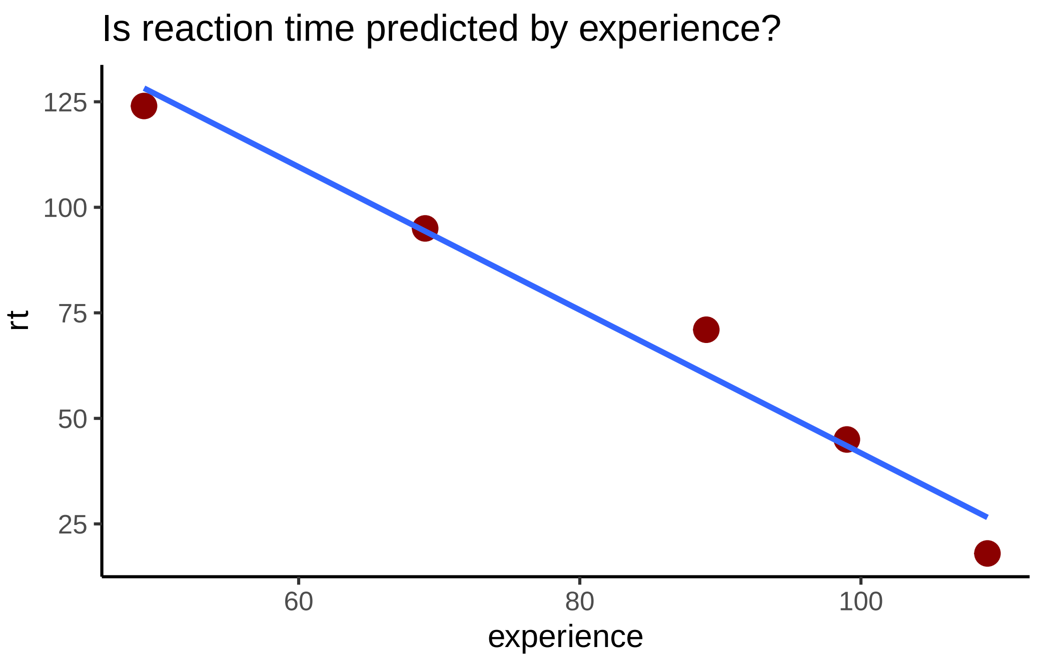
⚠ Under construction

Still working on these

```r
library(tidyverse)
library(modelr)
library(infer)
library(knitr)
library(parsnip)
library(optimg)
library(kableExtra)
theme_set(theme_classic(base_size = 12))

# setup data
data <- tibble(
    experience = c(49, 69, 89, 99, 109),
    rt = c(124, 95, 71, 45, 18)
)
```

Suppose that we have a set of data and we have specified the model we'd like to fit. The next step is to **fit the model** to the data. That is, to find the best estimate of the free parameters (weights) such that the model describes the data as well as possible.



0.1 Fitting Models in R

We will fit linear models using three common methods. During model specification week, we already started fitting models with `lm()` and `infer`. Today we will expand to include the `parsnip` way.

1. **`lm()`**: This is the most basic and widely used function for fitting linear models. It directly estimates model parameters based on the ordinary least-squares method, providing regression outputs such as coefficients, R-squared, etc.
2. **`infer` package**: This package focuses on statistical inference using tidyverse syntax. It emphasizes hypothesis testing, confidence intervals, and bootstrapping, making it ideal for inferential analysis.
3. **`parsnip` package**: Part of the `tidymodels` suite, `parsnip` provides a unified syntax for various modeling approaches (linear, logistic, random forest, etc.). It separates the model specification from the underlying engine, offering flexibility and consistency when working across multiple machine learning algorithms.

Each method has its strengths: `lm()` for simplicity, `infer` for inferential statistics, and `parsnip` for robust model flexibility across different algorithms. To illustrate, we can fit the data in the figure above all 3 ways.

```
# with lm()
lm(rt ~ 1 + experience, data = data)
```

```

Call:
lm(formula = rt ~ 1 + experience, data = data)

Coefficients:
(Intercept)   experience  
    211.271       -1.695  
```

```
# with infer
data %>%
    specify(formula = rt ~ 1 + experience) %>%
    fit()
```

```
# A tibble: 2 x 2
  term       estimate
  <chr>         <dbl>
1 intercept    211.  
2 experience    -1.69
```

```
# with parsnip
linear_reg() %>%
    set_engine("lm") %>%
    fit(rt ~ 1 + experience, data = data)
```

```
parsnip model object


Call:
stats::lm(formula = rt ~ 1 + experience, data = data)

Coefficients:
(Intercept)   experience  
    211.271       -1.695  
```

0.2 Goodness-of-fit

In order to find the best fitting free parameters, we first need to quantify what it means to fit best. **Sum of squared error** (SSE) is one common approach, in which we take the differences between the data and the model fit – also called the "error" or "residuals" – square those differences, and then take their sum.

$SSE = \sum_{i=i}^{n}(d_i - m_i)^2$

- n is the number of data points
- d_i is the i-th data point
- m_i is the model fit for the i-th data point

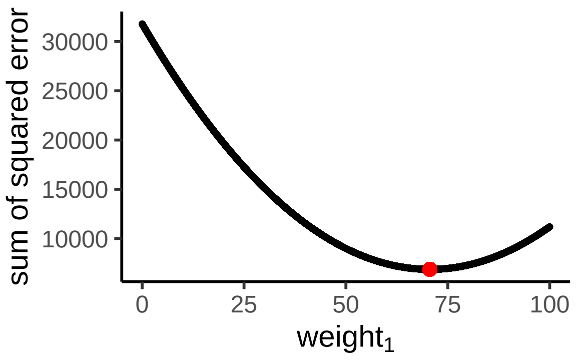
Given this way of quantifying goodness-of-fit, our job is to figure out the set of parameter values with the smallest possible sum of squared error. But how do we do that? There are two common approaches:

1. **Iterative Optimization** - works for both linear and nonlinear models
2. **Ordinary Least-Squares** - works for linear models only

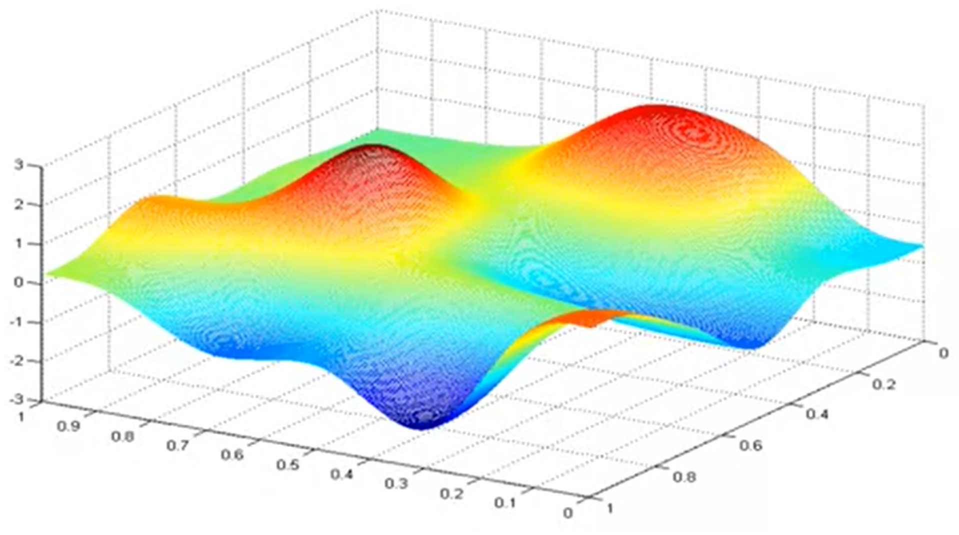
0.3 Iterative Optimization

In **Iterative optimization**, we think of finding the best fitting parameters as a *search problem* in which we have a *parameter space* and a *cost function* (or a "loss" function). To find the best fitting parameter estimates, we search through the space to find the point with the smallest possible cost function.

- We already have a cost function: sum of squared error
    - $\sum_{i=i}^{n}(d_i - m_i)^2$
- We can visualize iterative optimization by plotting our cost function on the y-axis, and our possible paramter weights on the x-axis (and z-axis, and higher dimensions as the number of inputs goes up).
- We call this visualizeation the **error surface**
- If there is one parameter to estimate (one input to the model), the error surface will be a curvy line.

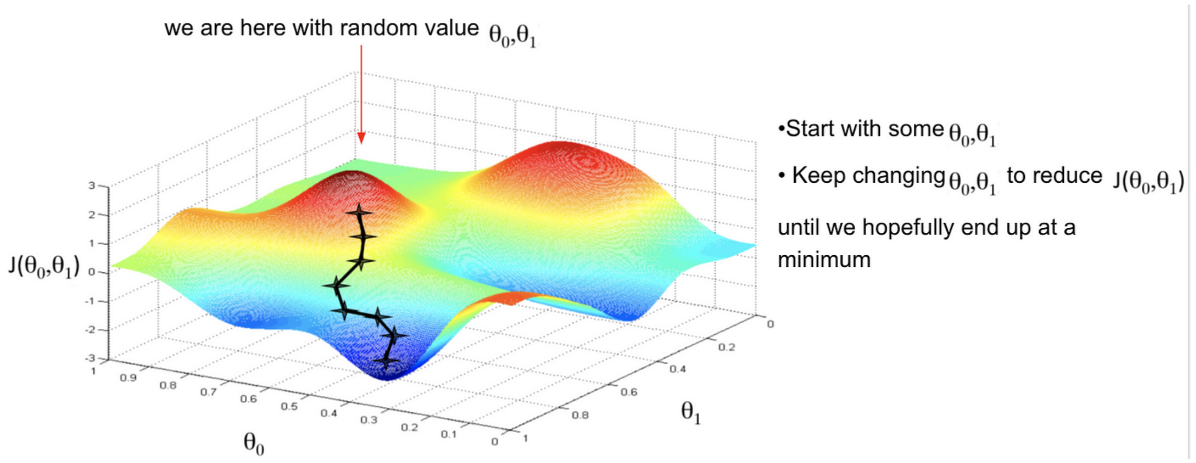


- If there are two parameters to estimate (two inputs to the model), the error surface will be a bumpy sheet.



To search through the parameter space via iterative optimization, we could use any number of iterative optimization algorithms. Many of them follow the same conceptual process (but differ in precise implementation):

1. Start at some point on the error surface (*initial seed*)
2. Look at the error surface in a small region around that point
3. Take a step in some direction that reduces the error
4. Repeat steps 2-4 until improvements are very small (less than some very small predefined number).



If this feels too abstract, we can also understand iterative optimization with a simple metaphor: suppose you were dropped from a plane or helicopter at a random spot in hilly terrain and wanted to find the lowest point. You could solve this with iterative optimization:

1. Start at some point in the hilly terrain (*inital seed*)
2. Look around you to determine the direction in which the ground seems to be sloping downward the most.
3. Take a small step downhill in that direction.
4. Repeat these steps until you reach a spot where all directions around you either ascend or remain flat.

0.3.1 Gradient descent

Gradient descent is one such iterative optimization algorithm. We can implement gradient descent in R with the `optimg` package to find the best fitting parameter estimates for our model.

1. First we write our cost function — our own function in R! — which must take a `data` argument (our data set) and a `par` parameter (a vector of parameter estimates we want to test) to work with `optimg`.

```
SSE <- function(data, par) {
    data %>%
        mutate(prediction = par[1] + par[2] * experience) %>%
        mutate(error = prediction - rt) %>%
        mutate(squared_error = error^2) %>%
        with(sum(squared_error))
}
```

2. Then we pass our data, cost function, and initial seed paramters to the `optimg` function to perform gradient descent.

```
optimg(
    data = data,  # our data
    par = c(0,0), # our starting parameters
    fn = SSE,     # our cost function (which receives data and par)
    method = "STGD" # our iterative optimization algorithm
    )
```

```
$par
[1] 211.26155  -1.69473

$value
[1] 205.138

$counts
[1] 12

$convergence
[1] 0
```

We can compare `optimg`'s estimates to that of `lm()` to see that they are *nearly* identical:

```
lm(rt ~ 1 + experience, data = data)
```

```

Call:
lm(formula = rt ~ 1 + experience, data = data)

Coefficients:
(Intercept)   experience
    211.271       -1.695
```

0.3.2 Nearly identical to `lm()`

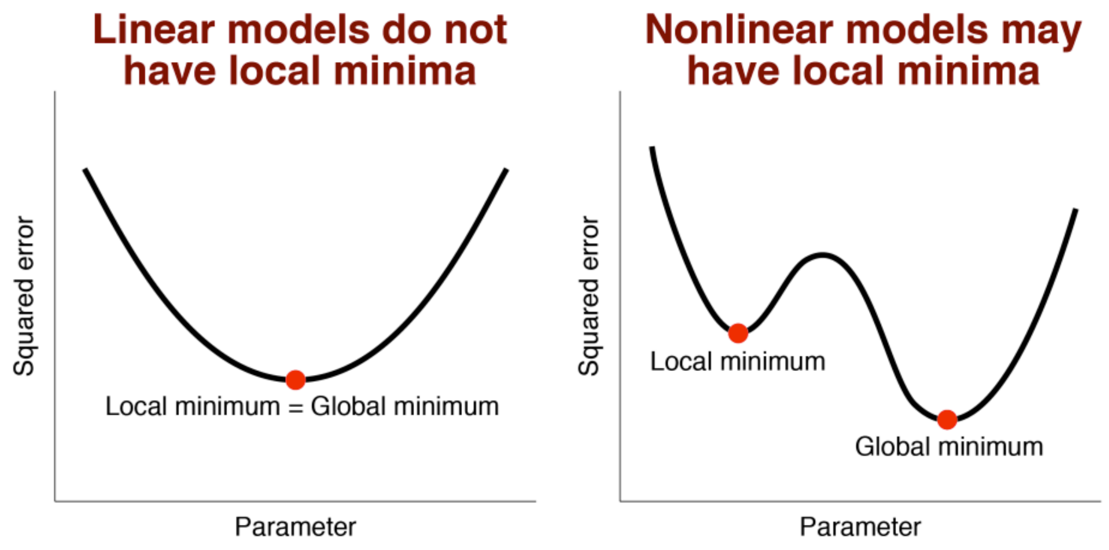
Note that solving for the best fitting free parameters via iterative optimization gets us an *approximate* value of the best fitting free parameters. Based on how the algorithm is implemented, we might decide to stop iterating too early (because we are close enough to the point) or even step over minimum point if our step size is too big.

0.3.3 Local minimum problem

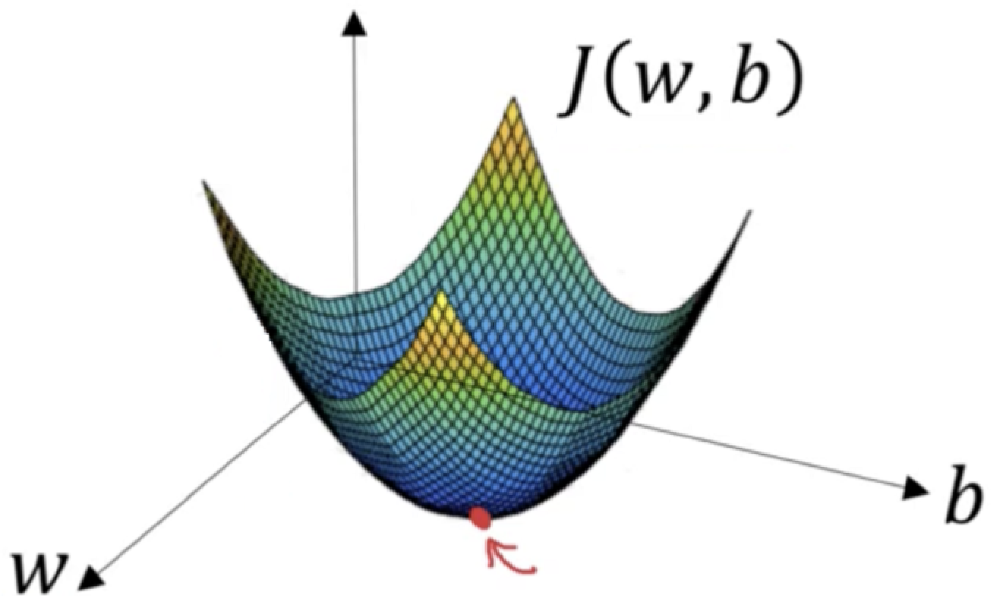
A potential problem with iterative optimization algorithms is the risk of finding a *local minimum.* That is, we find a location on the error surface that is a minimum within some local range (so our algorithm stops looking), but we are not at the absolute minimum (also called the *global minimum*).

- For all linear models, the error surface is shaped like a bowl, so there is *no risk* of a local minimum. As long as an algorithm can adjust parameters to reduce errors, we will eventually be able to get to *approximately* the optimal solution. We can see this clearly in the one or two parameter case (but it generalizes to higher dimensions as well).

Linear v. nonlinear model with one parameter:


Linear models do not have local minima
Squared error
Local minimum = Global minimum
Parameter
Nonlinear models may have local minima
Squared error
Local minimum
Global minimum
Parameter

Linear model with two parameters:



0.4 Ordinary Least-Squares

Another way we can find the best fitting free parameters for linear (or linearizable nonlinear) models is to use the Ordinary Least-Squares (OLS) estimate.

- In OLS, the best-fitting free parameters are found by solving a system of equations (using matrix operations/linear algebra) which leads to a closed-form solution.

- This means that OLS provides *exact values* of the best-fitting parameters in one step (as long as a few necessary conditions are met).
- We can contrast this with iterative optimization algorithms (like gradient descent) which gradually adjust the model parameters over multiple iterations to minimize the error, often requiring many steps to converge on *approximate values* of the best-fitting parameters.

In OLS, the goal is to model the relationship between input variables and the output variable (y) as a linear combination. We express this very generally in our favorite equation, where the output (y) is a weighted sum of inputs (x_i).

- $y = \sum_{i=1}^{n} w_i x_i$

Recall that this general expression has many aliases. That is, the **linear model equation** can be expressed in many ways, but *they are all this same thing*:

1. in **high school algebra**: $y = ax + b$.
2. in **machine learning**: $y = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n$
3. in **statistics**: $y = \beta_0 + \beta_1x_1 + \beta_2x_2 + ... + \beta_nx_n + \varepsilon$
4. in **matrix** notation: $y = Xw + \varepsilon$

The matrix notation is what allows us to appreciate that we can solve for the best fitting free parameters with linear algebra. Let's work this out for our data set predicting `rt ~ 1 + experience`. We can express in matrix notation:

$$\mathbf{y} = \mathbf{X}\mathbf{w} +$$

Where:

- $\mathbf{y}$ is the output vector (`rt`).
- $\mathbf{X}$ is the input matrix (`experience` with an intercept).
- $\mathbf{w}$ is the weight vector (parameter estimates including the intercept).
- ϵ is the vector of errors (residuals).

Because our data set is small, we can expand these to help you picture this visually a little better:

1. **Input Matrix X** (intercept and `experience`):

$$\mathbf{X} = \begin{bmatrix} 1 & 49 \\ 1 & 69 \\ 1 & 89 \\ 1 & 99 \\ 1 & 109 \end{bmatrix}$$

2. **Output Vector**, $\mathbf{y}$ (`rt`):

$$\mathbf{y} = \begin{bmatrix} 124 \\ 95 \\ 71 \\ 45 \\ 18 \end{bmatrix}$$

3. **Weight Vector**, $\mathbf{w}$ (Unknown coefficients including intercept):

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

Putting it all together, the linear model equation becomes, where there is a vector of errors (residuals), .

$$\begin{bmatrix} 124 \\ 95 \\ 71 \\ 45 \\ 18 \end{bmatrix} = \begin{bmatrix} 1 & 49 \\ 1 & 69 \\ 1 & 89 \\ 1 & 99 \\ 1 & 109 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \end{bmatrix}$$

It turns out that we can solve for the weight vector directly via the following equation:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

At this stage, we can take the mathematicians' word for it that this provides an *exact* solution to the best fitting parameter estimates.

Box 1: Summary of OLS solution

This is the end of the math we will uncover about the OLS in this course. However, those of you who have taken linear algebra may appreciate the following abridged summary of how we arrive at the closed form solution by minimizing the sum of squared errors. (If you have not taken linear alegebra, you can safely skip this box. It's not on the exam!).

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

To derive this we (in brief):

1. **Set Up the Linear Model**: Start with the matrix equation $\mathbf{y} = \mathbf{X}\mathbf{w} +$.
2. **Define Residuals**: Use $= \mathbf{y} - \mathbf{X}\mathbf{w}$ to express the errors.
3. **Minimize SSE**: Expand and differentiate the sum of squared errors, setting the derivative to zero.
4. **Derive Normal Equation**: Arrive at the normal equation $\mathbf{X}^\top \mathbf{X}\mathbf{w} = \mathbf{X}^\top \mathbf{y}$.
5. **Compute Weights**: Solve for $\mathbf{w}$ using the closed-form solution.

This process provides the *exact* weights that best fit the linear model to the data.

We can demonstrate this with code:

```
 ols_matrix_way <- function(X, Y){
  solve(t(X) %*% X) %*% t(X) %*% Y
 }
```

We need to construct X and Y (must be matrices):

```
(response_matrix <- data %>% select(rt) %>% as.matrix())
(explanatory_matrix <- data %>% mutate(int = 1) %>% select(int, experience) %>% as.matrix())
```

```
      rt                                      int experience
[1,] 124                                 [1,]   1         49
[2,]  95                                 [2,]   1         69
[3,]  71                                 [3,]   1         89
[4,]  45                                 [4,]   1         99
[5,]  18                                 [5,]   1        109
```

Then we can use our function to generate the OLS solution:

```
ols_matrix_way(explanatory_matrix, response_matrix)
```

```
                   rt
int        211.270690
experience  -1.694828
```

Which is **exactly the same** as that returned by `lm()` (because lm is doing this!)

```
lm(rt ~ experience, data = data)
```

```
Call:
lm(formula = rt ~ experience, data = data)

Coefficients:
(Intercept)   experience
    211.271       -1.695
```

Importantly, if there are more regressors than data points, then there is no OLS solution. The intuition for the underlying math is that if there are more weights than data points, there are infinaty many solutions, all of which acheive zero error. The linear algebra fans among us might appreciate that the xtx component of our equation is ill-defined in this case.

Here'a an example. Suppose we have the following dataset

```
data2 <- tibble(
    y = c(2, 5, 7),
    x = c(1, 2, 3),
    z = c(2, 4, 6),
    a = c(6, 7, 8)
)

(model1 <- lm(y ~ 1 + x, data = data2))
```

```


Call:
lm(formula = y ~ 1 + x, data = data2)

Coefficients:
(Intercept)            x
    -0.3333       2.5000
```

```
(model2 <- lm(y ~ 1 + x + z + a, data = data2))
```

```


Call:
lm(formula = y ~ 1 + x + z + a, data = data2)

Coefficients:
(Intercept)            x            z            a
    -0.3333       2.5000           NA           NA
```

`lm()` is smart and fits the reduced model it *can* fit. If we try to solve this the matrix way via our homegrown function, we get an error.