


Model fitting

Katie Schuler

2024-10-22

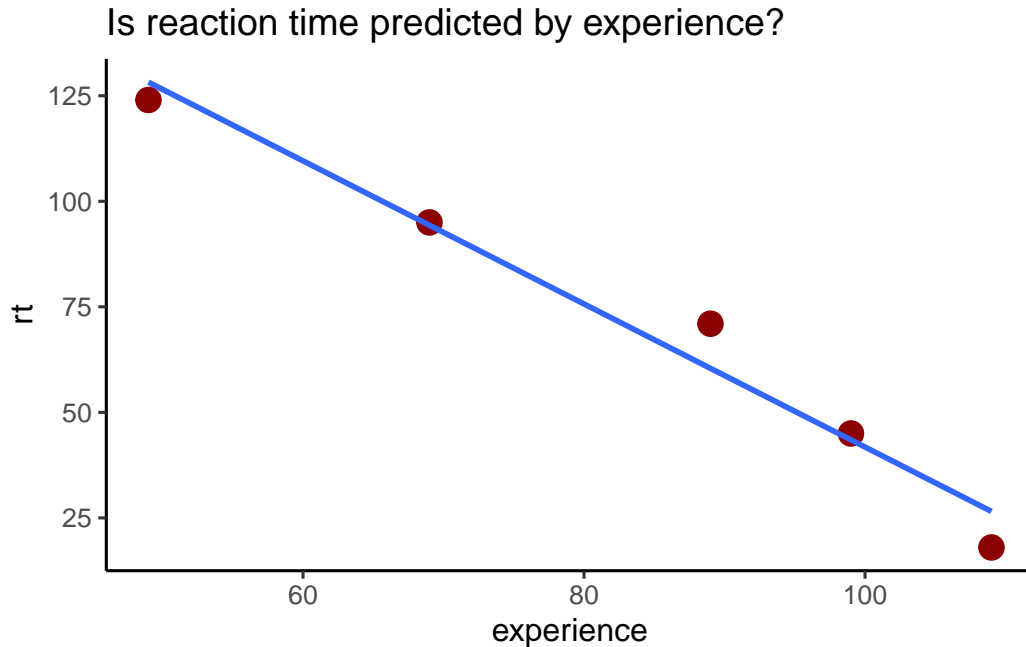
 Under construction

Still working on these. More to come Thursday.

```
library(tidyverse)
library(modelr)
library(infer)
library(knitr)
library(parsnip)
library(opting)
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.

$$SSE = \sum_{i=1}^n (d_i - m_i)^2$$

- 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

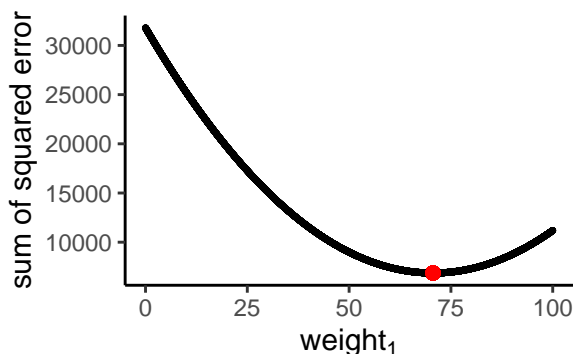
Given this way of quantifying goodness-of-fit, our job is to figure out the set of parameter values with the smallest possible sum of squared error. But how do we do that? There are two common approaches:

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

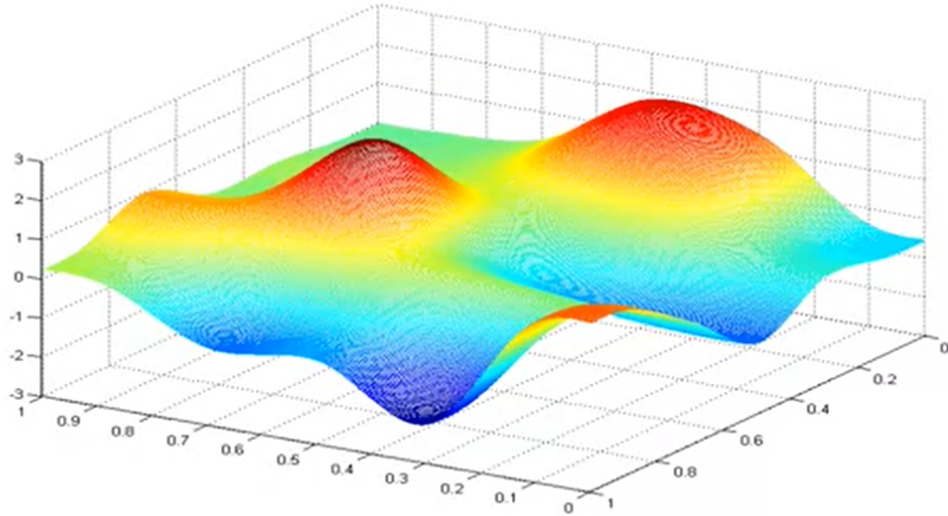
0.3 Iterative Optimization

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
 - $\sum_{i=1}^n (d_i - m_i)^2$
- We can visualize iterative optimization by plotting our cost function on the y-axis, and our possible parameter weights on the x-axis (and z-axis, and higher dimensions as the number of inputs goes up).
- We call this visualization the **error surface**
- If there is one parameter to estimate (one input to the model), the error surface will be a curvy line.

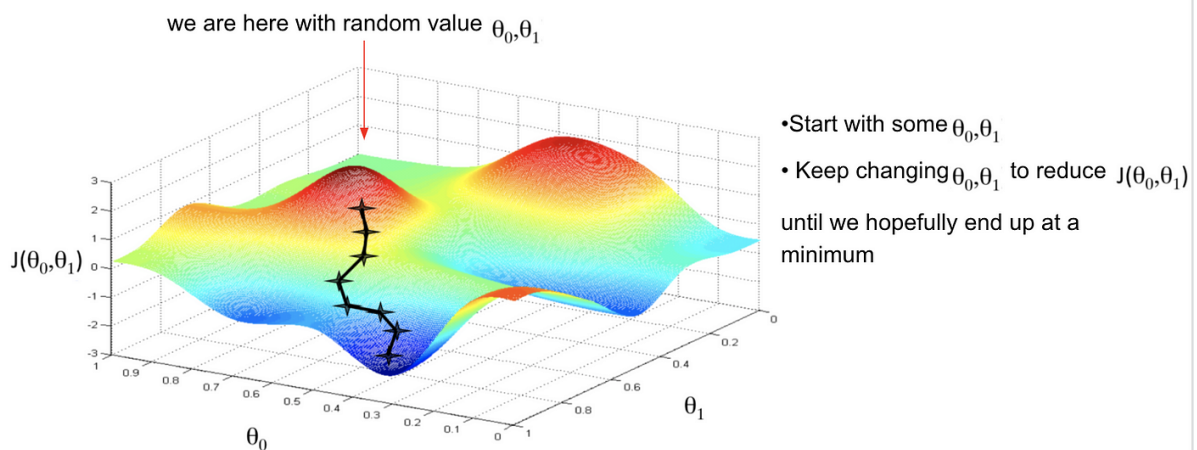


- If there are two parameters to estimate (two inputs to the model), the error surface will be a bumpy sheet.



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 (*initial 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 `optim` 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 `optim`.

```
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 parameters to the `optim` function to perform gradient descent.

```
optim(
  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 `optim`'s estimates to that of `lm()` to see that they are *nearly* identical:

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

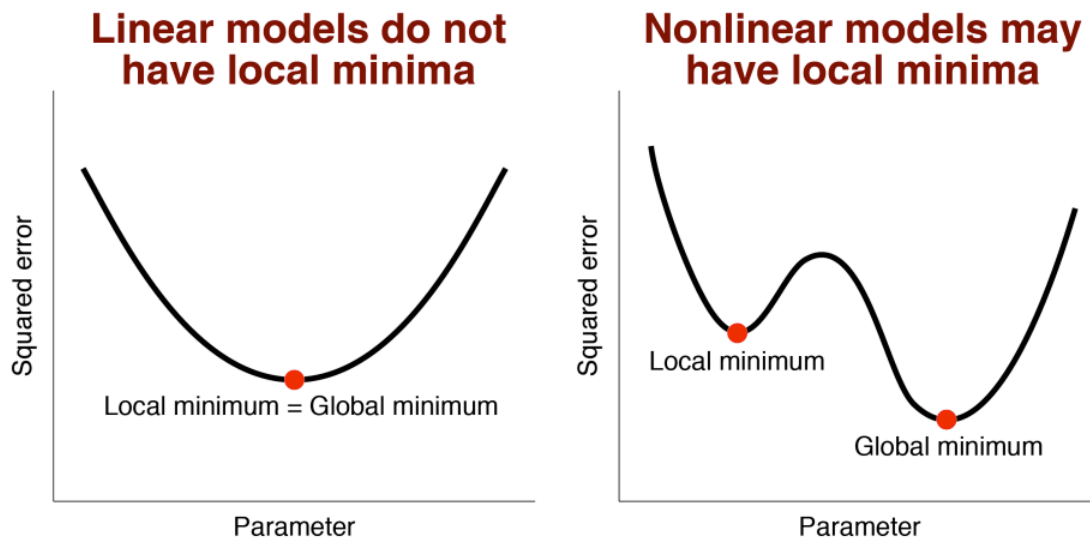
```
Call:
lm(formula = rt ~ 1 + experience, data = data)
```

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

0.3.2 Nearly identical to $\text{lm}()$

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*).



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.