


Model fitting

Katie Schuler

2024-10-22

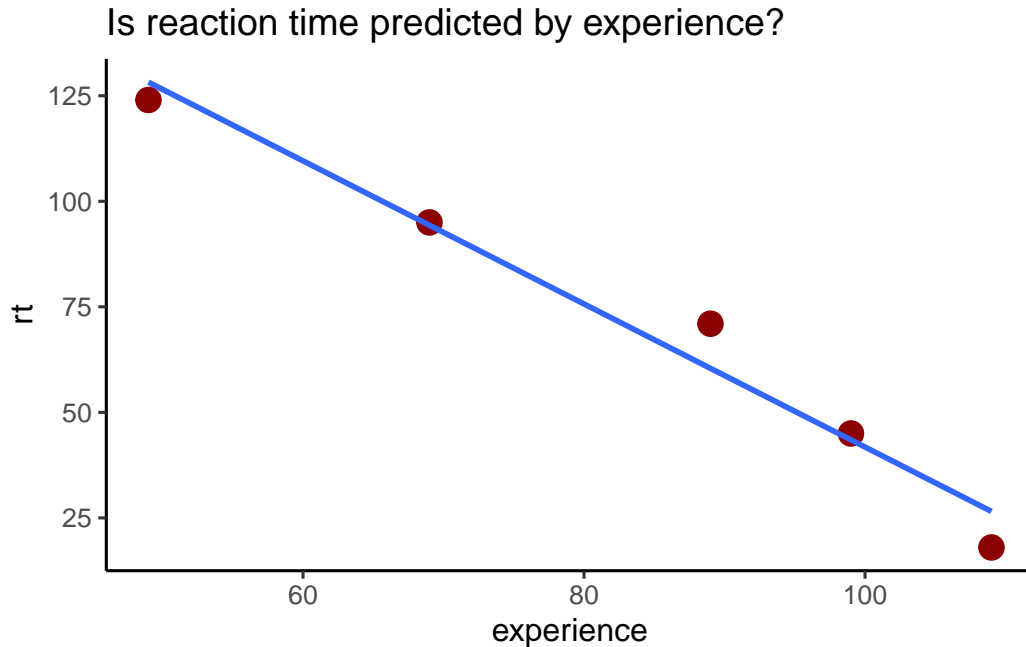
 Under construction

More coming for Thursday!

```
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 The Basics

In order to find the best fitting free parameters, we first need to quantify *goodness-of-fit*. Sum of squared error is one common approach, in which we find take the sum of the squares of the differences between the data and the model fit:

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

- 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 that minimize the sum of squared error. But how do we do that? In this course, we will consider two ways:

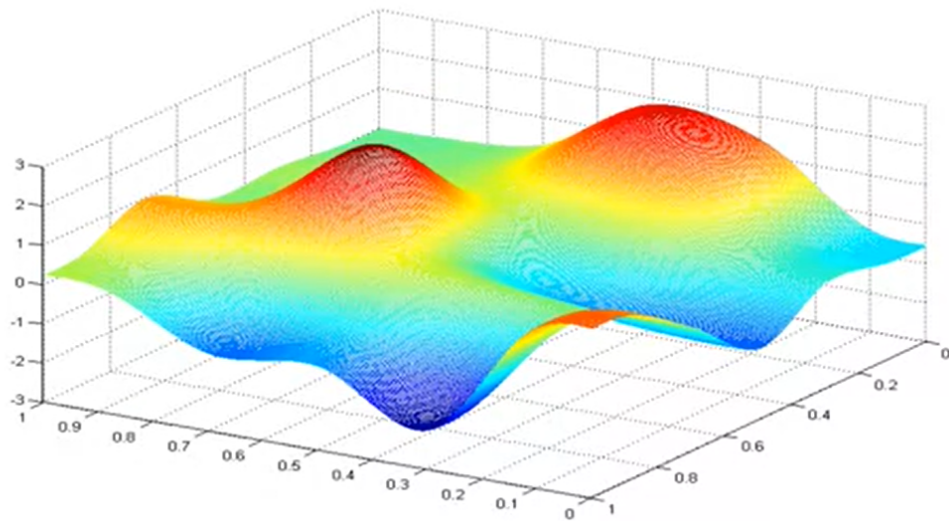
1. **Iterative Optimization** - for linear and nonlinear models
2. **Ordinary Least-Squares** - for linear models

0.3 Iterative Optimization

We can 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 that minimizes the cost function.

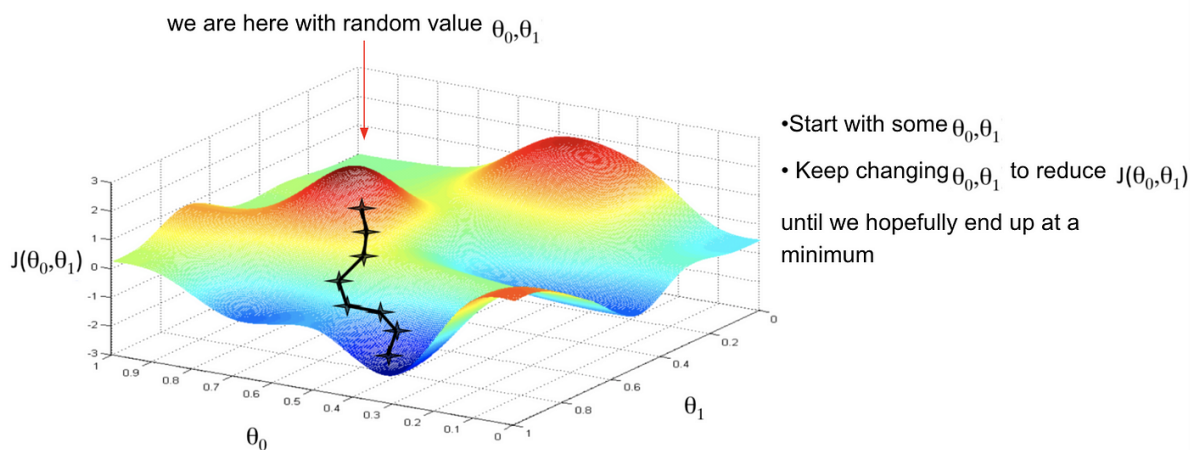
- We already have a cost function: sum of squared error

$$- \sum_{i=1}^n (d_i - m_i)^2$$
- We can visualize iterative optimization as trying to find the minimum point on an **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).



Gradient descent is simple iterative optimization algorithm, widely used in machine learning applications. We can implement gradient descent in R with the `optim` package to find the

best fitting parameter estimates.

1. First we write our cost function — a literal 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).

```
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 parameters to test 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  
  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

0.4 Ordinary Least-Squares