Model fitting

Katie Schuler

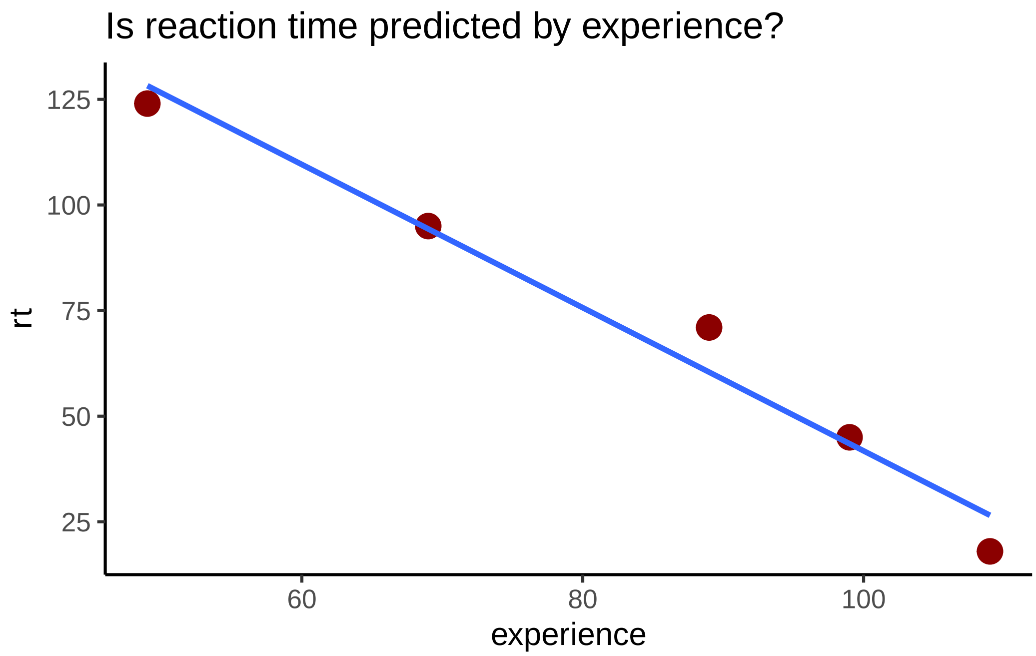
2024-10-22

> ⚠ Under construction
>
> Still working on these. More to come Thursday.

```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.

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

```

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

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

```r
# 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
```

```r
# 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, or *goodness-of-fit*. **Sum of squared error** 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:

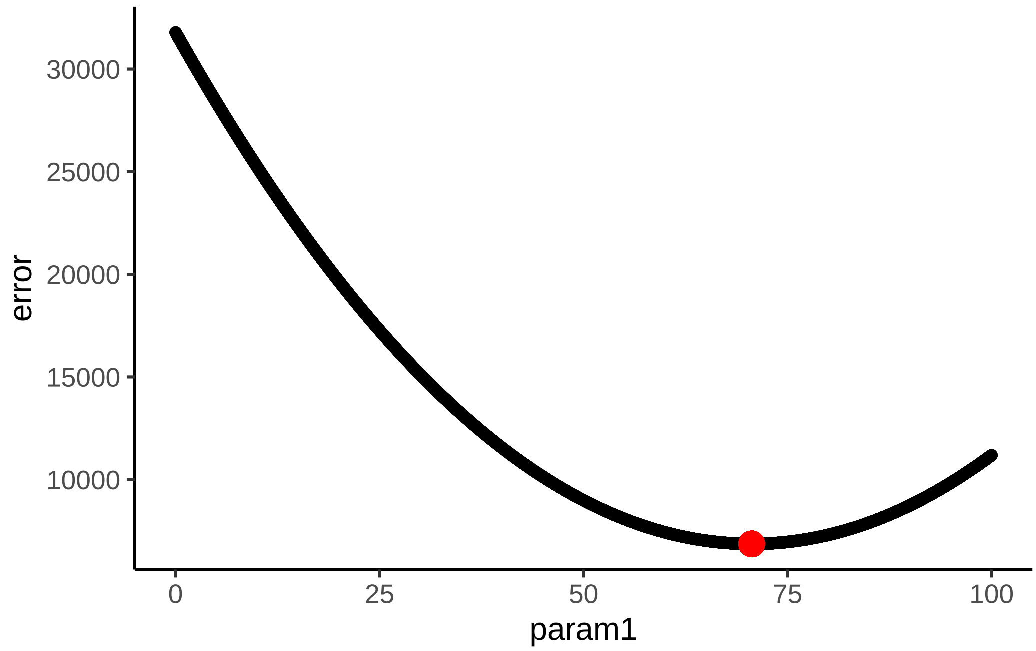
1. **Iterative Optimization** - works for both linear and nonlinear models
2. **Ordinary Least-Squares** - works for linear models only

0.3 Iterative Optimization

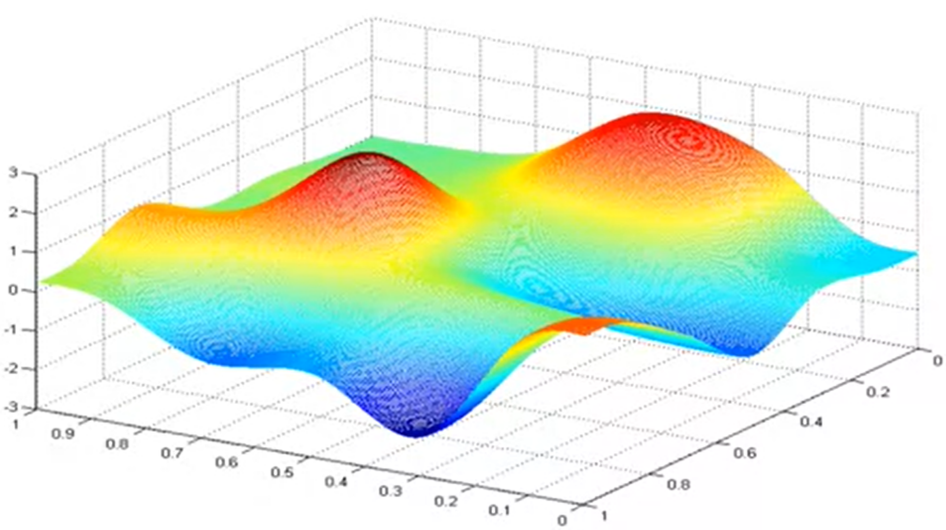
To solve this probem with **Iterative optimization**, we think of finding the best fitting parameters as a *search problem* in which we have a *parameter space* and a *cost function* (or a “loss” function). To find the best fitting parameter estimates, we search through the space to find the point with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

```
Warning in geom_point(size = 4, color = "red", aes(x = 70.6, y = 6869)): All aesthetics have
i Did you mean to use `annotate()`?
```

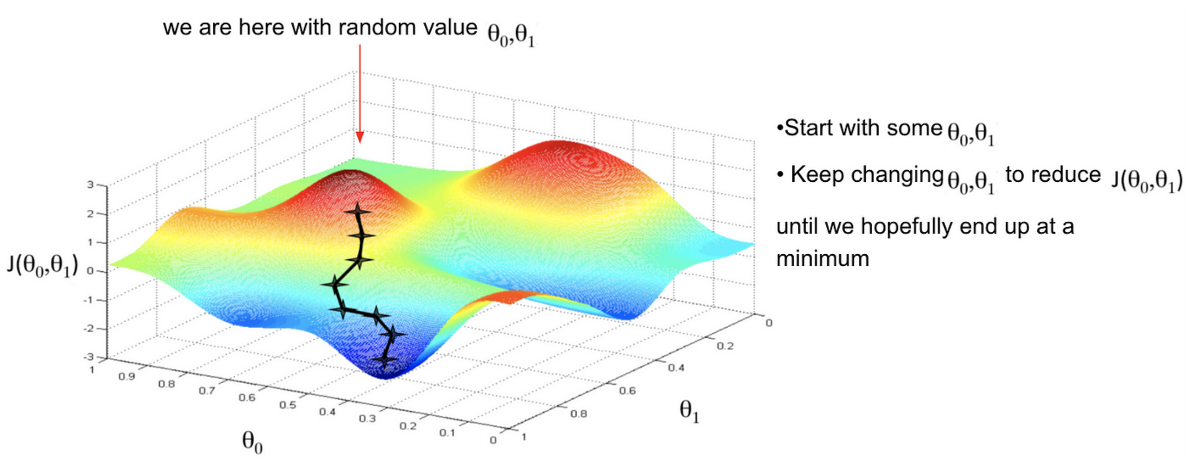


If there are two parameters to estimate (two inputs to the model), the error surface will be a bumpy sheet.



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 `optimg` 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 paramters to test 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
    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:

```r
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