

Linear Regression

Last modified: 2026-10-27

```
# The Python chunks on this page import numpy, pandas, matplotlib, scipy and statsmodels;
# reticulate provisions them.
# The Julia chunks run through JuliaCall.
library(reticulate)
py_require(c("numpy", "pandas", "matplotlib", "scipy", "statsmodels"))
jl <- function(expr) JuliaCall::julia_eval(expr)
f3 <- function(v) sprintf("%.3f", v)
f9 <- function(v) sprintf("%.9f", v)
# A positive number in scientific notation, for display math: 2.4\times10^{2}.
sci <- function(v) {
  e <- floor(log10(v))
  m <- signif(v / 10^e, 2)
  if (m >= 10) {
    m <- m / 10
    e <- e + 1
  }
  sprintf("%.1f\\times10^{%d}", m, e)
}
```

i Source

Consolidated from Brian Hutchinson's Fall 2025 topic notes and Kameron Decker Harris' 2022 weekly notes, and drawing on Logan Sizemore's Spring 2025 lecture on multivariate regression for the feature-expansion section.

1 Linear functions and geometry

1.1 Linear function fundamentals

Definition 1.1 (Affine function). An **affine function** of an input x is a constant plus a multiple of the input. With one input it is

$$h(x) = \beta_0 + \beta_1 x,$$

and with a vector of p inputs it adds one multiple per input,

$$h(\tilde{x}) = \beta_0 + \vec{\beta} \cdot \tilde{x}.$$

In words, the **intercept** β_0 is the output when every input is zero, and each **slope** β_j is how much the output changes when input j goes up by one and the others stay fixed. The graph of an affine function of one input is a line, and of two inputs a plane. Neither has to pass through the origin [1].

Definition 1.2 (Linear function). A **linear function** in the strict sense of linear algebra^a is an **affine function** whose **intercept** β_0 is zero, so that $h(\tilde{x}) = \vec{\beta} \cdot \tilde{x}$.

Example 1.1 (A linear and an affine function with the same slope). Take the slope $\beta_1 = 2$ and compare $h_{\text{lin}}(x) = 2x$, which has intercept 0, with $h_{\text{aff}}(x) = 1 + 2x$, which has intercept 1. Figure 1 plots both. The chunk computes the two functions on a grid of inputs, so the plot and the numbers quoted below come from the same arrays.

```
import numpy as np
import matplotlib.pyplot as plt

# One slope, two intercepts: the only difference between the two functions.
slope = 2.0
intercept_lin, intercept_aff = 0.0, 1.0

# Evaluate both on one grid of inputs; the plot and the text read these arrays.
x_grid = np.linspace(-2, 2, 101)
h_lin = intercept_lin + slope * x_grid
h_aff = intercept_aff + slope * x_grid

fig, ax = plt.subplots(figsize=(4.5, 3.5), layout="constrained")
ax.plot(x_grid, h_lin, label="linear: $2x$");
ax.plot(x_grid, h_aff, label="affine: $1 + 2x$");
# Mark the output at x = 0, which is the intercept by definition.
ax.plot([0], [intercept_aff], "o", color="C1");
ax.axhline(0, color="gray", linewidth=0.5);
ax.axvline(0, color="gray", linewidth=0.5);
ax.set_xlabel("Input, $x$");
ax.set_ylabel("Output, $h(x)$");
fig.legend(loc="outside lower center", ncol=2);
plt.show()
```

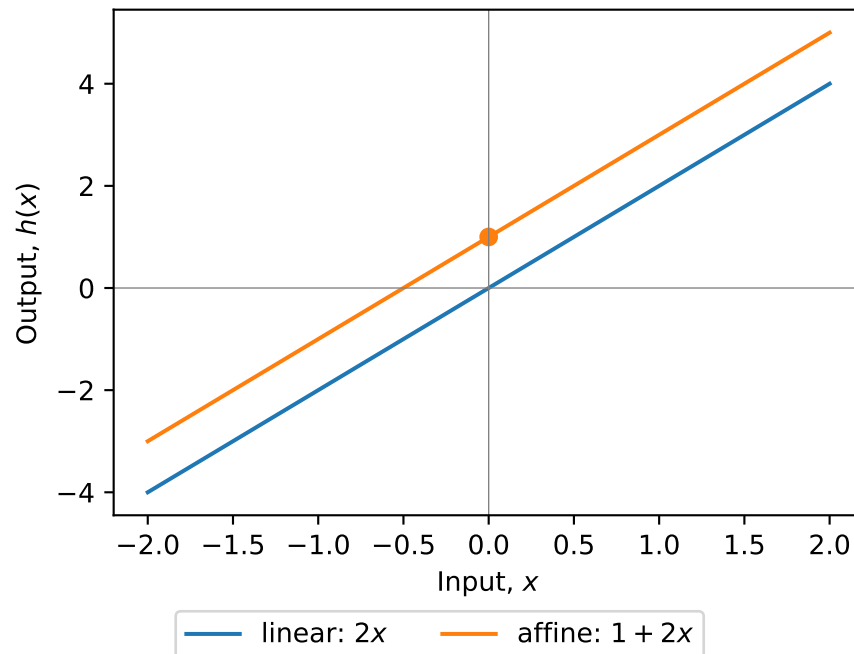


Figure 1: A linear function (intercept 0) and an affine function (intercept 1) with the same slope 2. The affine line is the linear one shifted up by its intercept, so it crosses the vertical axis at 1.

The affine line is the linear line moved up by the intercept. Doubling the input shows the other difference. The next chunk doubles $x = 1$ and compares the output at the doubled input with double the original output.

```
# Compare h(2x) with 2 h(x) at x = 1, for each function.
# A linear function scales exactly, h(2x) = 2 h(x); an affine one does not,
# because doubling the input doubles the slope term but leaves the intercept alone.
x0 = 1.0
gap = {}
for name, intercept in [("linear", intercept_lin), ("affine", intercept_aff)]:
    h_of_double = intercept + slope * (2 * x0)
    double_of_h = 2 * (intercept + slope * x0)
    gap[name] = double_of_h - h_of_double
    print(f"{name}: h(2x) = {h_of_double}, 2 h(x) = {double_of_h}")
```

linear: h(2x) = 4.0, 2 h(x) = 4.0

affine: h(2x) = 5.0, 2 h(x) = 6.0

For the linear function the two numbers agree. For the affine function they differ by 1, which is its intercept.

i Remark (A linear function scales and passes through the origin)

Remark 1.1 (A linear function scales and passes through the origin). A **linear function** satisfies $h(c\tilde{x}) = c h(\tilde{x})$ for every number c , and $h(\vec{0}) = 0$. An **affine function** with $\beta_0 \neq 0$ satisfies neither.

Definition 1.3 (Linear function and linear model). A **linear function** maps an input vector $x \in \mathbb{R}^p$ to a scalar output via a linear combination^a (strictly speaking, affine, as in Definition 1.1) of the features:

$$\begin{aligned} h(\tilde{x}) &= \beta_0 + \sum_{j=1}^p \beta_j x_j \\ &= \beta_0 + \vec{\beta} \cdot \tilde{x} \end{aligned}$$

where $\beta_0 \in \mathbb{R}$ is the intercept (or bias parameter) and $\vec{\beta} = [\beta_1, \beta_2, \dots, \beta_p]^\top \in \mathbb{R}^p$ is the vector of slopes (or feature weights).

- When $p = 1$, the function fits a line:

$$h(x) = \beta_0 + \beta_1 x$$

recovering simple linear regression^b.

- When $p > 1$, the function fits a hyperplane^c, as in multiple linear regression.

In statistics and machine learning, this functional form defines the linear regression model [2].

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-linear-combination>

^b<https://morrison-lab.github.io/sds/correlation-regression.html>

^c<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-hyperplane>

Example 1.2 (A line and a plane for fuel efficiency). The **Auto** data [3] give each car's fuel efficiency (mpg), horsepower and weight. With one feature, the linear function of Definition 1.3 is a line, $h(x) = \beta_0 + \beta_1 x$. With two features it is a plane, $h(\tilde{x}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$. Fit both by least squares, and predict mpg for a car with 100 horsepower and a weight of 2,800 pounds:

1.1.1 Python

```
import numpy as np
import pandas as pd

auto_lm = pd.read_csv("data/auto.csv")
y_lm = auto_lm["mpg"].to_numpy(float)

# One column per feature, with a leading column of ones so that the
# first coefficient is the intercept.
X_line_lm = np.column_stack([np.ones(len(y_lm)), auto_lm["horsepower"]])
X_plane_lm = np.column_stack([X_line_lm, auto_lm["weight"]])
beta_line_lm = np.linalg.lstsq(X_line_lm, y_lm, rcond=None)[0]
beta_plane_lm = np.linalg.lstsq(X_plane_lm, y_lm, rcond=None)[0]

# h(x) = intercept + dot product of the slopes with the features.
h_line_lm = beta_line_lm[0] + beta_line_lm[1:] @ [100]
h_plane_lm = beta_plane_lm[0] + beta_plane_lm[1:] @ [100, 2800]
print(beta_line_lm, beta_plane_lm)

[39.93586102 -0.15784473] [ 4.56402108e+01 -4.73028631e-02 -5.79415736e-03]
print(h_line_lm, h_plane_lm)

24.15138768580511 24.686283910109854
```

1.1.2 R

```
auto_lm <- read.csv("data/auto.csv")
fit_line_lm <- lm(mpg ~ horsepower, data = auto_lm)
fit_plane_lm <- lm(mpg ~ horsepower + weight, data = auto_lm)

# h(x) = intercept + dot product of the slopes with the features.
h_line_lm <- coef(fit_line_lm)[[1]] + sum(coef(fit_line_lm)[-1] * 100)
h_plane_lm <- coef(fit_plane_lm)[[1]] + sum(coef(fit_plane_lm)[-1] * c(100, 2800))
c(line = h_line_lm, plane = h_plane_lm)

      line      plane
24.15139 24.68628
```

The line predicts 24.2 miles per gallon, using horsepower alone. The plane predicts 24.7, because it also lets weight move the prediction: the weight slope is -0.0058 miles per gallon for each extra pound.

1.2 Linear functions, visually

Definition 1.4 (Hyperplane). A hyperplane^a in $\mathbb{R}^p$ is a $(p - 1)$ -dimensional affine subspace^b:

$$\mathcal{H} = \{x \in \mathbb{R}^p : \vec{\beta} \cdot x + \beta_0 = 0\}$$

for a non-zero normal vector^c $\vec{\beta} \in \mathbb{R}^p$ and offset $\beta_0 \in \mathbb{R}$ [4].

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-hyperplane>

^b<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-affine-map>

^c<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-normal-vector>

Example 1.3 (The synthetic data). Let's simulate some data that we can use to demonstrate how linear models work. We'll call this simulated dataset "the synthetic data", and reuse it for the rest of this chapter. Listing 1 generates a sample with a fixed random seed. Each observation has two features,

x_1 and x_2 , drawn independently from the standard normal distribution, and a response y , which is a linear function of the two features plus a small amount of normal noise. The data frame has two further columns, `jitter` and `y_collinear`, which the later examples introduce where they use them. The R chunk defines the data frame `synthetic`, and the Python and Julia chunks of later examples read it from there.

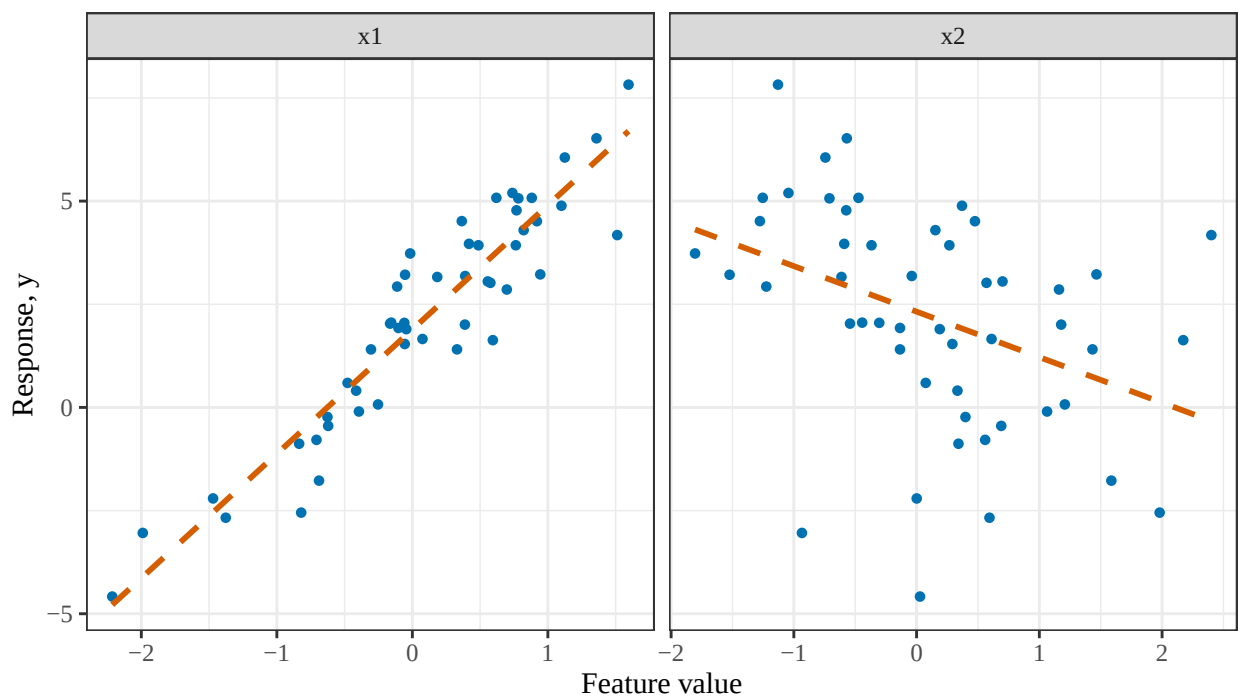
Listing 1 Simulating the synthetic data

```
# Simulate the sample once, in R, so that the R, Python and Julia chunks
# of later examples all analyse identical numbers
# (each language's random number generator would draw different ones).
set.seed(1)
n_syn <- 50
synthetic <- data.frame(
  x1 = rnorm(n_syn),
  x2 = rnorm(n_syn),
  jitter = rnorm(n_syn) # a third, unrelated draw, used to perturb x2 in a later example
)
noise_syn <- rnorm(n_syn, sd = 0.1)
# y depends on both features
synthetic$y <- 2 + 3 * synthetic$x1 - synthetic$x2 + noise_syn
# y_collinear depends on x1 alone, for the examples with collinear features
synthetic$y_collinear <- 2 + synthetic$x1 + noise_syn
```

The sample has 50 observations.

1.2.1 R

```
library(ggplot2)
syf_long <- data.frame(
  feature = rep(c("x1", "x2"), each = nrow(synthetic)),
  value = c(synthetic$x1, synthetic$x2),
  y = rep(synthetic$y, 2)
)
ggplot(syf_long, aes(value, y)) +
  geom_point(colour = "#0072B2", shape = 16) +
  geom_smooth(
    method = "lm", formula = y ~ x, se = FALSE,
    colour = "#D55E00", linetype = "dashed"
  ) +
  facet_wrap(vars(feature), scales = "free_x") +
  labs(x = "Feature value", y = "Response, y")
```

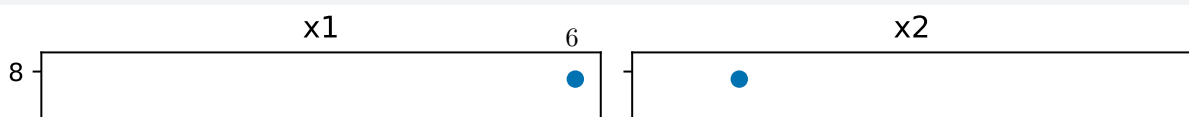


(a)

1.2.2 Python

```
import matplotlib.pyplot as plt
import numpy as np

syf_df = r.synthetic
fig, axes = plt.subplots(1, 2, figsize=(7, 4), sharey=True, layout="constrained")
for ax, feature in zip(axes, ["x1", "x2"]):
    value_syf = syf_df[feature].to_numpy()
    slope_syf, intercept_syf = np.polyfit(value_syf, syf_df["y"], 1)
    grid_syf = np.array([value_syf.min(), value_syf.max()])
    ax.plot(value_syf, syf_df["y"], color="#0072B2", marker="o", linestyle="");
    ax.plot(grid_syf, intercept_syf + slope_syf * grid_syf, color="#D55E00", linestyle="--");
    ax.set_title(feature);
    ax.set_xlabel("Feature value");
axes[0].set_ylabel("Response, y");
plt.show()
```



Example 1.4 (The hyperplane of a fitted linear function). Fit $h(\tilde{x}) = \vec{\beta} \cdot \tilde{x} + \beta_0$ to the two features of the synthetic data by least squares, and find a point on the hyperplane $\vec{\beta} \cdot \tilde{x} + \beta_0 = 0$ with first coordinate 1:

1.2.3 Python

```
import numpy as np

geo = r.synthetic.to_numpy()
x_geo, y_geo = geo[:, :2], geo[:, 3]
design_geo = np.column_stack([np.ones(len(y_geo)), x_geo])
coef_geo = np.linalg.solve(design_geo.T @ design_geo, design_geo.T @ y_geo)
b0_geo, b_geo = coef_geo[0], coef_geo[1:]
point_geo = np.array([1.0, -(b0_geo + b_geo[0]) / b_geo[1]])
print(point_geo, b_geo @ point_geo + b0_geo)

[1.          4.96042842] 0.0
```

1.2.4 R

```
geo <- synthetic
fit_geo <- lm(y ~ x1 + x2, data = geo)
b0_geo <- coef(fit_geo)[["(Intercept)"]]
b_geo <- coef(fit_geo)[c("x1", "x2")]
point_geo <- c(1, -(b0_geo + b_geo[["x1"]]) / b_geo[["x2"]])
point_geo

[1] 1.000000 4.960428

sum(b_geo * point_geo) + b0_geo

[1] 4.440892e-16
```

The normal vector is $\vec{\beta} = (2.967, -1.004)$, both entries non-zero, so the hyperplane in $\mathbb{R}^2$ is a line; the point above lies on it, since $\vec{\beta} \cdot \tilde{x} + \beta_0$ is zero there up to rounding error.

Theorem 1.1 (Linear and affine hyperplanes). *Let $\mathcal{H}$ be the hyperplane of Definition 1.4.*

1. *If $\beta_0 = 0$, then $\mathcal{H}$ passes through the origin and is a linear subspace.*
2. *If $\beta_0 \neq 0$, then $\mathcal{H}$ does not contain the origin. For any $\tilde{x}_0 \in \mathcal{H}$, $\mathcal{H} = \{\tilde{x}_0 + \tilde{v} : \vec{\beta} \cdot \tilde{v} = 0\}$, which is the linear subspace $\{\tilde{v} : \vec{\beta} \cdot \tilde{v} = 0\}$ shifted by $\tilde{x}_0$.*

i Proof

Proof.

1. With $\beta_0 = 0$, $\mathcal{H} = \{\tilde{x} : \vec{\beta} \cdot \tilde{x} = 0\}$, which contains $\vec{0}$ and is closed under addition and scalar multiplication.
2. At the origin, $\vec{\beta} \cdot \vec{0} + \beta_0 = \beta_0 \neq 0$, so $\vec{0} \notin \mathcal{H}$. For $\tilde{x}_0 \in \mathcal{H}$, $\vec{\beta} \cdot \tilde{x}_0 = -\beta_0$, so for any $\tilde{x}$,

$$\begin{aligned} \vec{\beta} \cdot \tilde{x} + \beta_0 = 0 &\iff \vec{\beta} \cdot \tilde{x} = \vec{\beta} \cdot \tilde{x}_0 \\ &\iff \vec{\beta} \cdot \tilde{x} - \tilde{x}_0 = 0. \end{aligned}$$

Hence $\tilde{x} \in \mathcal{H}$ exactly when $\tilde{x} - \tilde{x}_0$ lies in the linear subspace.

□

Example 1.5 (The fitted hyperplane misses the origin). For the hyperplane of Example 1.4, check that the origin is not on it, and that moving from the point $\tilde{x}_0$ in a direction $\tilde{v}$ with $\vec{\beta} \cdot \tilde{v} = 0$ stays on it:

1.2.5 Python

```
direction_geo = np.array([b_geo[1], -b_geo[0]])
print(b0_geo, b_geo @ direction_geo, b_geo @ (point_geo + 2.5 * direction_geo) + b0_geo)
2.011438958958926 0.0 0.0
```

1.2.6 R

```
direction_geo <- c(b_geo[["x2"]], -b_geo[["x1"]])
c(origin = b0_geo,
  direction = sum(b_geo * direction_geo),
  shifted = sum(b_geo * (point_geo + 2.5 * direction_geo)) + b0_geo)

      origin      direction      shifted
2.011439e+00 0.000000e+00 4.440892e-16
```

The value at the origin is the offset $\beta_0 = 2.011$, which is not zero, so this hyperplane is affine and not a linear subspace.

Theorem 1.2 (The graph of a linear function is a hyperplane). *The set of input-output pairs*

$$\{[\tilde{x}; y] \in \mathbb{R}^{p+1} : y = \vec{\beta} \cdot \tilde{x} + \beta_0\}$$

is a hyperplane in $\mathbb{R}^{p+1}$, with normal vector $[\vec{\beta}; -1]$ and offset β_0 .

i Proof

Proof. For $[\tilde{x}; y] \in \mathbb{R}^{p+1}$,

$$\begin{aligned} y = \vec{\beta} \cdot \tilde{x} + \beta_0 &\iff \vec{\beta} \cdot \tilde{x} - y + \beta_0 = 0 \\ &\iff [\vec{\beta}; -1] \cdot [\tilde{x}; y] + \beta_0 = 0. \end{aligned}$$

The vector $[\vec{\beta}; -1]$ is non-zero, since its last entry is -1 . □

Example 1.6 (The fitted values lie on the graph hyperplane). For the fit of Example 1.4, the pairs of an input and its fitted value satisfy the hyperplane equation of Theorem 1.2, and the pairs of an input and its observed response do not:

1.2.7 Python

```
normal_geo = np.append(b_geo, -1.0)
fitted_geo = x_geo @ b_geo + b0_geo
on_fitted = np.column_stack([x_geo, fitted_geo]) @ normal_geo + b0_geo
on_observed = np.column_stack([x_geo, y_geo]) @ normal_geo + b0_geo
print(np.abs(on_fitted).max(), np.abs(on_observed).max())
4.440892098500626e-16 0.2515321410227709
```

1.2.8 R

```
normal_geo <- c(b_geo, -1)
fitted_geo <- fitted(fit_geo)
on_fitted <- cbind(geo$x1, geo$x2, fitted_geo) %*% normal_geo + b0_geo
on_observed <- cbind(geo$x1, geo$x2, geo$y) %*% normal_geo + b0_geo
c(fitted = max(abs(on_fitted)), observed = max(abs(on_observed)))
```

```
      fitted      observed
7.549517e-15 2.515321e-01
```

The fitted pairs deviate from the hyperplane only by rounding error. The largest deviation of an observed pair is 0.25, because the observed pairs differ from the fitted pairs by the residuals.

Definition 1.5 (Half-spaces). The hyperplane $\mathcal{H}$ of Definition 1.4 divides $\mathbb{R}^p$ into two opposing **half-spaces** (illustrated in Figure 3) [5]:

$$\mathcal{H}^+ = \{\tilde{x} \in \mathbb{R}^p : \vec{\beta} \cdot \tilde{x} + \beta_0 \geq 0\}, \quad \mathcal{H}^- = \{\tilde{x} \in \mathbb{R}^p : \vec{\beta} \cdot \tilde{x} + \beta_0 < 0\}$$

Example 1.7 (Training points on each side of the fitted hyperplane). Count the training points of Example 1.4 in each half-space:

1.2.9 Python

```
side_geo = x_geo @ b_geo + b0_geo >= 0
print(side_geo.sum(), (~side_geo).sum())
```

```
39 11
```

1.2.10 R

```
side_geo <- as.matrix(geo[, c("x1", "x2")]) %*% b_geo + b0_geo >= 0
c(positive = sum(side_geo), negative = sum(!side_geo))
```

```
positive negative
      39         11
```

39 of the 50 points lie in $\mathcal{H}^+$, and the rest lie in $\mathcal{H}^-$. In linear binary classification, the hyperplane is the decision boundary, and the two half-spaces are the two predicted classes.

Definition 1.6 (Level set). For a function $h : \mathbb{R}^p \rightarrow \mathbb{R}$, the **level set**^a (or **contour**) at value $c \in \mathbb{R}$ is the pre-image of c [6]:

$$h^{-1}(\{c\}) = \{\tilde{x} \in \mathbb{R}^p : h(\tilde{x}) = c\}$$

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-level-set>

Example 1.8 (A level set of the fitted function). For the fitted function $h(\tilde{x}) = \vec{\beta} \cdot \tilde{x} + \beta_0$ of Example 1.4, the level set at $c = 1$ contains the point with first coordinate 1 that solves $h(\tilde{x}) = c$:

1.2.11 Python

```
level_geo = 1.0
level_point_geo = np.array([1.0, (level_geo - b0_geo - b_geo[0]) / b_geo[1]])
print(level_point_geo, level_point_geo @ b_geo + b0_geo)
```

```
[1.          3.96401683] 1.0
```

1.2.12 R

```
level_geo <- 1
level_point_geo <- c(1, (level_geo - b0_geo - b_geo[["x1"]]) / b_geo[["x2"]])
level_point_geo
```

```
[1] 1.000000 3.964017
```

```
sum(level_point_geo * b_geo) + b0_geo
```

```
[1] 1
```

The value of h at this point equals c , so the point is in $h^{-1}(\{1\})$.

Theorem 1.3 (Level sets of a linear function). *Let $h(x) = \vec{\beta} \cdot x + \beta_0$ with $\vec{\beta} \neq \vec{0}$. Every level set $h^{-1}(\{c\})$ of Definition 1.6 is a hyperplane with normal vector $\vec{\beta}$, parallel to the decision boundary $h^{-1}(\{0\})$. Its dimension is $p - 1$: a point for $p = 1$, a line for $p = 2$, and a two-dimensional plane for $p = 3$.*

i Proof

Proof. For any c ,

$$\begin{aligned} h^{-1}(\{c\}) &= \{x : \vec{\beta} \cdot x + \beta_0 = c\} \\ &= \{x : \vec{\beta} \cdot x + (\beta_0 - c) = 0\}. \end{aligned}$$

This is the hyperplane of Definition 1.4 with normal vector $\vec{\beta}$ and offset $\beta_0 - c$. Different values of c change only the offset, so the level sets share the normal vector and are parallel. A hyperplane in $\mathbb{R}^p$ has dimension $p - 1$ by Definition 1.4. $\square$

Example 1.9 (Parallel level lines and a level point). With two features, the level set of the fitted function of Example 1.4 at value c is the line $x_2 = (c - \beta_0 - \beta_1 x_1) / \beta_2$. Compute the slope and intercept of the level lines at three values of c . With one feature (x_1 alone), the level set is a single point:

1.2.13 Python

```
levels_geo = np.array([-1.0, 0.0, 1.0])
slope_geo = -b_geo[0] / b_geo[1]
intercept_geo = (levels_geo - b0_geo) / b_geo[1]
print(slope_geo, intercept_geo)
```

```
2.956207321500671 [3.00063269 2.0042211 1.00780951]
```

```
design_one = np.column_stack([np.ones(len(y_geo)), x_geo[:, 0]])
coef_one = np.linalg.solve(design_one.T @ design_one, design_one.T @ y_geo)
print((0.0 - coef_one[0]) / coef_one[1])
```

```
-0.6270727820462808
```

1.2.14 R

```
levels_geo <- c(-1, 0, 1)
slope_geo <- -b_geo[["x1"]] / b_geo[["x2"]]
intercept_geo <- (levels_geo - b0_geo) / b_geo[["x2"]]
slope_geo
```

```
[1] 2.956207
```

```
intercept_geo
```

```
[1] 3.000633 2.004221 1.007810
```

```
fit_one <- lm(y ~ x1, data = geo)
(0 - coef(fit_one)[["(Intercept)"]]) / coef(fit_one)[["x1"]]
```

```
[1] -0.6270728
```

All three level lines have the same slope, 2.956, and the intercepts differ, so the lines are parallel. The level set at $c = 0$ of the one-feature fit is the single point $x_1 = -0.627$.

Theorem 1.4 (The normal vector is orthogonal to the level sets and points uphill). Let $h(\underline{x}) = \vec{\beta} \cdot \underline{x} + \beta_0$ with $\vec{\beta} \neq \vec{0}$, and write $\|\vec{\beta}\|$ for the Euclidean norm^a of $\vec{\beta}$.

1. If $\underline{x}, \underline{x}' \in h^{-1}(\{c\})$ lie in the same level set of Definition 1.6, then $\vec{\beta} \perp (\underline{x} - \underline{x}')$.
2. For every $\underline{x} \in \mathbb{R}^p$ and $t \in \mathbb{R}$, $h(\underline{x} + t\vec{\beta}) = h(\underline{x}) + t\|\vec{\beta}\|^2$.
3. Among unit vectors^b $\underline{u}$ ($\|\underline{u}\| = 1$), the change $h(\underline{x} + t\underline{u}) - h(\underline{x})$ with $t > 0$ is largest for $\underline{u} = \vec{\beta}/\|\vec{\beta}\|$.

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-euclidean-norm>

^b<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-unit-vector>

i Proof

Proof. Part 1. Both points satisfy $\vec{\beta} \cdot \underline{x} + \beta_0 = c$ and $\vec{\beta} \cdot \underline{x}' + \beta_0 = c$. Subtracting the second equation from the first gives $\vec{\beta} \cdot \underline{x} - \vec{\beta} \cdot \underline{x}' = 0$. By linearity of the dot product, this is $\vec{\beta} \cdot \underline{x} - \underline{x}' = 0$. That is orthogonality^a.

Part 2. Substituting the definition of h and using linearity of the dot product,

$$\begin{aligned} h(\underline{x} + t\vec{\beta}) &= \vec{\beta} \cdot \underline{x} + t\vec{\beta} \cdot \vec{\beta} + \beta_0 \\ &= \vec{\beta} \cdot \underline{x} + t\vec{\beta} \cdot \vec{\beta} + \beta_0 \\ &= h(\underline{x}) + t\|\vec{\beta}\|^2. \end{aligned}$$

Part 3. For a unit vector $\underline{u}$,

$$\begin{aligned} h(\underline{x} + t\underline{u}) - h(\underline{x}) &= (\vec{\beta} \cdot \underline{x} + t\underline{u} \cdot \vec{\beta} + \beta_0) - (\vec{\beta} \cdot \underline{x} + \beta_0) \\ &= t\vec{\beta} \cdot \underline{u}. \end{aligned}$$

Since $t > 0$, maximizing this change is the same as maximizing $\vec{\beta} \cdot \underline{u}$. By the Cauchy-Schwarz inequality^b, $\vec{\beta} \cdot \underline{u} \leq \|\vec{\beta}\|\|\underline{u}\| = \|\vec{\beta}\|$. Equality holds exactly when $\underline{u} = \alpha\vec{\beta}$ for some $\alpha > 0$. The condition $\|\underline{u}\| = 1$ then forces $\alpha\|\vec{\beta}\| = 1$, so $\underline{u} = \vec{\beta}/\|\vec{\beta}\|$. $\square$

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-orthogonal-vectors>

^b<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#thm-cauchy-schwarz>

i Remark (Reading the normal-vector theorem)

Remark 1.2 (Reading the normal-vector theorem). Part 1 of Theorem 1.4 is why $\vec{\beta}$ is called *normal*: it is orthogonal to every displacement within a level set (Theorem 1.3). In part 2, let $\underline{x}$ lie in the level set $h^{-1}(\{c\})$, so $h(\underline{x}) = c$. A step of $t > 0$ along $\vec{\beta}$ lands in the level set $h^{-1}(\{c + t\|\vec{\beta}\|^2\})$, which has a larger value, so $\vec{\beta}$ points toward increasing values of h . A step with $t < 0$ lands in a level set with a smaller value. Part 3 is the sense in which that direction is the steepest.

Example 1.10 (The coefficient vector is orthogonal to a level line). For the fitted function of Example 1.4, take two points of the level set $h^{-1}(\{1\})$ from Example 1.8, one with first coordinate 1 and one with first coordinate 2. Check that their difference is orthogonal to $\vec{\beta}$, and that a step of $t = 0.5$ along $\vec{\beta}$ raises h by $t\|\vec{\beta}\|^2$ (Theorem 1.4):

1.2.15 Python

```
# two points on the level set at c = 1
pt_a_geo = np.array([1.0, (level_geo - b0_geo - b_geo[0] * 1.0) / b_geo[1]])
pt_b_geo = np.array([2.0, (level_geo - b0_geo - b_geo[0] * 2.0) / b_geo[1]])
# part 1: the dot product with the difference is zero
print(b_geo @ (pt_b_geo - pt_a_geo))

4.440892098500626e-16

# part 2: the change in f after a step of 0.5 along the coefficient vector
step_geo = 0.5
print(((pt_a_geo + step_geo * b_geo) @ b_geo + b0_geo) - level_geo, step_geo * (b_geo @ b_geo))

4.904717949284366 4.904717949284368
```

1.2.16 R

```
# two points on the level set at c = 1
pt_a_geo <- c(1, (level_geo - b0_geo - b_geo[["x1"]] * 1) / b_geo[["x2"]])
pt_b_geo <- c(2, (level_geo - b0_geo - b_geo[["x1"]] * 2) / b_geo[["x2"]])
# part 1: the dot product with the difference is zero
sum(b_geo * (pt_b_geo - pt_a_geo))

[1] 0

# part 2: the change in f after a step of 0.5 along the coefficient vector
step_geo <- 0.5
sum((pt_a_geo + step_geo * b_geo) * b_geo) + b0_geo - level_geo

[1] 4.904718

step_geo * sum(b_geo^2)

[1] 4.904718
```

The dot product in part 1 is zero up to rounding, and the two values in part 2 agree.

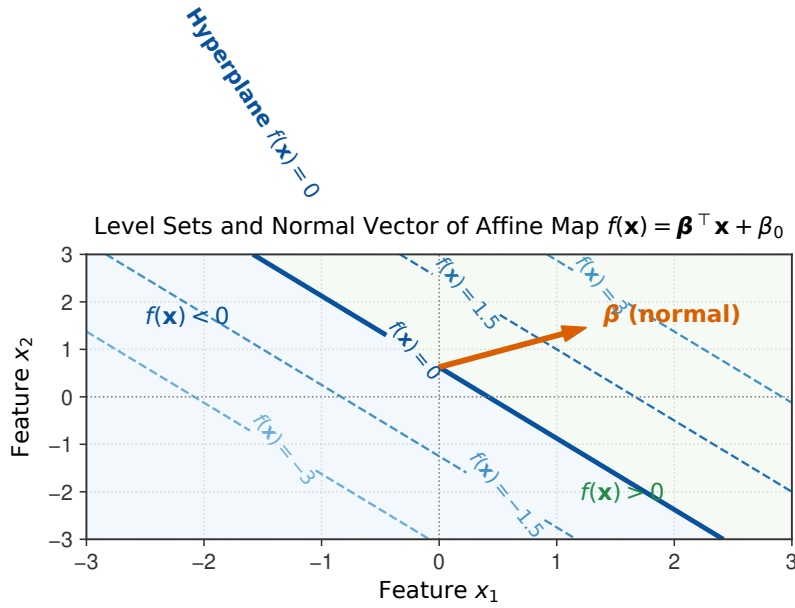


Figure 3: Level sets $h^{-1}(\{c\})$ (Definition 1.6) of an affine function h in $\mathbb{R}^2$, drawn as level lines because $p = 2$. The coefficient vector $\tilde{\beta}$, the normal vector of Definition 1.4, is orthogonal to the level lines and points toward increasing values of $h(x)$ (Theorem 1.4).

Definition 1.7 (Contour plot). For a function $h : \mathbb{R}^2 \rightarrow \mathbb{R}$, a **contour plot** draws several level sets $h^{-1}(\{c_1\}), h^{-1}(\{c_2\}), \dots$ of Definition 1.6 in the plane, each labelled by its value c .

Example 1.11 (A contour plot of the fitted function). Draw the level sets of the fitted function of Example 1.4 at the values $c = -1, 0, 1$:

```
library(ggplot2)
grid_geo <- expand.grid(x1 = seq(-2, 2, length.out = 50), x2 = seq(-2, 2, length.out = 50))
grid_geo$h <- b0_geo + b_geo[["x1"]] * grid_geo$x1 + b_geo[["x2"]] * grid_geo$x2
ggplot(grid_geo, aes(x1, x2, z = h)) +
  geom_contour(aes(colour = factor(after_stat(level))), breaks = levels_geo) +
  labs(colour = "c") +
  coord_equal() +
  theme(legend.position = "bottom")
```

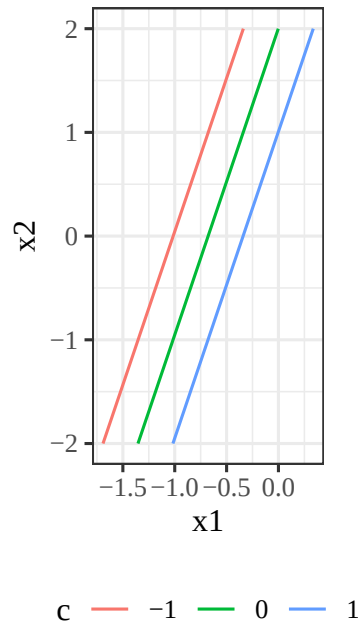


Figure 4: Level lines of the fitted linear function at $c = -1, 0$ and 1 .

The lines are parallel, as Theorem 1.3 says, and the orientation and spacing of the lines show the direction and steepness of the fitted function.

1.3 Linear functions as inner products

Definition 1.8 (Inner product (dot product)). For vectors $\tilde{x}, \tilde{y} \in \mathbb{R}^p$, the standard **dot product** / **inner product**^a is

$$\begin{aligned}\tilde{x} \cdot \tilde{y} &= \tilde{x} \cdot \tilde{y} \\ &= \langle \tilde{x}, \tilde{y} \rangle \\ &= \sum_{j=1}^p x_j y_j\end{aligned}$$

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-dot-product>

Example 1.12 (The inner product of the two synthetic features). Compute the inner product of the two feature columns of [the synthetic data](#), as a sum of products and with the built-in product:

1.3.1 Python

```
x1_geo, x2_geo = x_geo[:, 0], x_geo[:, 1]
print(np.sum(x1_geo * x2_geo), np.dot(x1_geo, x2_geo))
-0.9534465347538759 -0.9534465347538756
```

1.3.2 R

```
c(sum_of_products = sum(geo$x1 * geo$x2), built_in = drop(crossprod(geo$x1, geo$x2)))
sum_of_products      built_in
```

-0.9534465

-0.9534465

Definition 1.9 (Euclidean norm (L_2 norm)). The **Euclidean norm**^a (L_2 norm) of $\tilde{x} \in \mathbb{R}^p$ is induced by the inner product of Definition 1.8:

$$\begin{aligned}\|\tilde{x}\|_2 &= \sqrt{\tilde{x} \cdot \tilde{x}} \\ &= \sqrt{\sum_{j=1}^p x_j^2}\end{aligned}$$

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-euclidean-norm>

Example 1.13 (The norms of the two synthetic features). Compute the Euclidean norm of each feature column of [the synthetic data](#):

1.3.3 Python

```
print(np.sqrt(np.dot(x1_geo, x1_geo)), np.linalg.norm(x2_geo))
5.862940419034619 6.832351161195385
```

1.3.4 R

```
c(x1 = sqrt(sum(geo$x1^2)), x2 = norm(as.matrix(geo$x2), type = "F"))
      x1      x2
5.862940 6.832351
```

Theorem 1.5 (The sign of the inner product). For nonzero $\tilde{x}, \tilde{y} \in \mathbb{R}^p$ with angle^a $\theta \in [0, \pi]$ between them,

$$\tilde{x} \cdot \tilde{y} = \|\tilde{x}\|_2 \|\tilde{y}\|_2 \cos \theta,$$

so:

- $\tilde{x} \cdot \tilde{y} > 0$ when the vectors point broadly in the same direction ($\theta < \pi/2$).
- $\tilde{x} \cdot \tilde{y} = 0$ exactly when the vectors are orthogonal ($\theta = \pi/2$).
- $\tilde{x} \cdot \tilde{y} < 0$ when the vectors point broadly in opposing directions ($\theta > \pi/2$).

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-angle>

i Proof

Proof. By the definition of the angle, $\cos \theta = \tilde{x} \cdot \tilde{y} / (\|\tilde{x}\|_2 \|\tilde{y}\|_2)$, which gives the identity. The product $\|\tilde{x}\|_2 \|\tilde{y}\|_2$ is positive because both vectors are nonzero, so $\tilde{x} \cdot \tilde{y}$ has the sign of $\cos \theta$. On $[0, \pi]$, the cosine is positive for $\theta < \pi/2$, zero at $\pi/2$ and negative for $\theta > \pi/2$. $\square$

Example 1.14 (The angle between the synthetic features). Compute the angle between the two feature columns of [the synthetic data](#), and compare the sign of the cosine with the sign of the inner product:

1.3.5 Python

```
cos_geo = np.dot(x1_geo, x2_geo) / (np.linalg.norm(x1_geo) * np.linalg.norm(x2_geo))
print(cos_geo, np.degrees(np.arccos(cos_geo)), np.dot(x1_geo, x2_geo))
-0.023801848374948625 91.36387425599459 -0.9534465347538756
```

1.3.6 R

```
cos_geo <- sum(geo$x1 * geo$x2) / (sqrt(sum(geo$x1^2)) * sqrt(sum(geo$x2^2)))
c(cosine = cos_geo, degrees = acos(cos_geo) * 180 / pi, inner_product = sum(geo$x1 * geo$x2))
```

cosine	degrees	inner_product
-0.02380185	91.36387426	-0.95344653

The cosine is -0.024, so the angle is 91.4 degrees. The inner product and the cosine have the same sign, as the theorem says.

i Remark (An affine function as a projection)

Remark 1.3 (An affine function as a projection). By Theorem 1.5, evaluating an **affine function** $h(\tilde{x}) = \vec{\beta} \cdot \tilde{x} + \beta_0$ measures the projection of $\tilde{x}$ along the parameter direction $\vec{\beta}$, offset by the intercept β_0 .

Definition 1.10 (The intercept trick (bias trick, 1-padding)). The **intercept trick**, also called the **bias trick** or **1-padding**, writes an **affine function** of $x \in \mathbb{R}^p$ as one **inner product**. It places the constant^a 1 before the entries of $\tilde{x}$ in the design vector $\tilde{D}(\tilde{x})$, and places the intercept^b β_0 before the **slope** coefficients in the parameter vector $\vec{\beta}$:

$$\tilde{D}(\tilde{x}) = \begin{bmatrix} 1 \\ \tilde{x} \end{bmatrix} \in \mathbb{R}^{p^*}, \quad \vec{\beta} = \begin{bmatrix} \beta_0 \\ \vec{\beta}_{\text{slope}} \end{bmatrix} \in \mathbb{R}^{p^*}, \quad p^* = p + 1.$$

In words: the intercept becomes one more entry of $\vec{\beta}$, paired with the entry of $\tilde{D}(\tilde{x})$ that is always 1.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

Example 1.15 (Applying the intercept trick to one input). Take $p = 2$ features, $x = [3, 5]^\top$, intercept^a $\beta_0 = 1$ and **slope** parameters $\vec{\beta}_{\text{slope}} = [2, -1]^\top$. The **intercept trick** gives $\tilde{D}(\tilde{x}) = [1, 3, 5]^\top$ and $\vec{\beta} = [1, 2, -1]^\top$, so $p^* = 3$.

```
bp_x <- c(3, 5)
bp_beta <- c(1, 2, -1)
bp_padded <- c(1, bp_x)
```

The padded **inner product** is 2. The intercept 1 plus the inner product of the slope parameters with $\tilde{x}$, 1, is the same number.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

Exercise 1.1 (The padding is not a convention, it is an identity). Let $\tilde{x} \in \mathbb{R}^p$ and $\vec{\beta}_{\text{slope}} \in \mathbb{R}^p$, and set the design vector $\tilde{D}(\tilde{x}) \in \mathbb{R}^{p^*}$ and full parameter vector $\vec{\beta} \in \mathbb{R}^{p^*}$ with $p^* = p + 1$:

$$\tilde{D}(\tilde{x}) = \begin{bmatrix} 1 \\ \tilde{x} \end{bmatrix} \in \mathbb{R}^{p^*}, \quad \vec{\beta} = \begin{bmatrix} \beta_0 \\ \vec{\beta}_{\text{slope}} \end{bmatrix} \in \mathbb{R}^{p^*}$$

Show that $\vec{\beta} \cdot \tilde{D}(\tilde{x}) = \vec{\beta}_{\text{slope}} \cdot \tilde{x} + \beta_0$.

Then say what goes wrong if the padded 1 is put at the *end* of $\tilde{D}(\tilde{x})$ but the intercept is left at the *start* of $\vec{\beta}$.

Solution

Solution 1.1. Write the inner product as a sum over all p^* coordinates, indexed from 0:

$$\begin{aligned}\vec{\beta} \cdot \underset{\sim}{D}(\underset{\sim}{x}) &= \sum_{i=0}^p \beta_i D(\underset{\sim}{x})_i \\ &= \beta_0 \cdot 1 + \sum_{i=1}^p \beta_i x_i\end{aligned}$$

The zeroth coordinate is β_0 and $D(\underset{\sim}{x})_0 = 1$, so the first term is β_0 . The remaining coordinates are exactly $\vec{\beta}_{\text{slope}}$ and $\underset{\sim}{x}$, so the sum is $\vec{\beta}_{\text{slope}} \cdot \underset{\sim}{x}$. Adding them gives the claim.

The intercept is therefore not a special parameter with its own update rule. It is an ordinary coefficient on an input that happens to be constant, so a single formula covers all $p^* = p + 1$ coordinates with no special case for it.

If the padding is misaligned, the identity fails silently and gives the wrong function rather than an error. With a misaligned design vector $\tilde{D}(\underset{\sim}{x}) = [\underset{\sim}{x}; 1]$ and $\vec{\beta} = [\beta_0; \vec{\beta}_{\text{slope}}]$, the product pairs β_0 with x_1 and β_p with the constant 1:

$$\vec{\beta} \cdot \tilde{D}(\underset{\sim}{x}) = \beta_0 x_1 + \beta_1 x_2 + \cdots + \beta_{p-1} x_p + \beta_p$$

Every coefficient is attached to the wrong feature. The shapes still conform, so no library will complain; the fit will simply be worse than it should be, for no visible reason. Choose the padding position once and use it everywhere.

Theorem 1.6 (The padded 1 absorbs the intercept). *Let $\underset{\sim}{x} \in \mathbb{R}^p$ and $\vec{\beta}_{\text{slope}} \in \mathbb{R}^p$, and set $D(\underset{\sim}{x}) = [1, \underset{\sim}{x}^\top]^\top \in \mathbb{R}^{p^*}$ and $\vec{\beta} = [\beta_0, \vec{\beta}_{\text{slope}}^\top]^\top \in \mathbb{R}^{p^*}$ with $p^* = p + 1$. Then*

$$\vec{\beta} \cdot \underset{\sim}{D}(\underset{\sim}{x}) = \vec{\beta}_{\text{slope}} \cdot \underset{\sim}{x} + \beta_0.$$

i Proof

Proof. This is the solution to Exercise 1.1: see Solution 1.1. □

Example 1.16 (Padding a design vector with a 1). Take $p = 2$, $\underset{\sim}{x} = [3, 5]^\top$, $\beta_0 = 1$ and $\vec{\beta}_{\text{slope}} = [2, -1]^\top$. Then $\underset{\sim}{D}(\underset{\sim}{x}) = [1, 3, 5]^\top$ and $\vec{\beta} = [1, 2, -1]^\top$.

```
bt_x <- c(3, 5)
bt_b0 <- 1
bt_slope <- c(2, -1)
c(padded = sum(c(bt_b0, bt_slope) * c(1, bt_x)), slope_plus_intercept = sum(bt_slope * bt_x) + bt_b0)

      padded slope_plus_intercept
      2              2
```

Both sides equal 2: the slope part contributes 1 and the intercept adds 1.

i Four senses of “bias”

The β_0 that Theorem 1.6 absorbs into $\hat{\beta}$ is the **bias term**. It is the first of four barely-related senses this course gives the word, and the only one of the four that is a number in the model.

The bias term is the constant β_0 , also called the **intercept** or the **offset**. It lets the fitted line not pass through the origin. It measures nothing and it is not an error.

The bias of an estimator is $\mathbb{E}[\hat{\theta}] - \theta$, for an estimator $\hat{\theta}$ of a parameter θ : how far the estimate lands from the truth on average over repeated samples. The hat here marks an estimator, not the 1-padding of Theorem 1.6, which is a separate notation clash to watch for. Least squares is *unbiased* whenever X has full column rank and the errors have mean zero given X , which makes a sentence that reads like a contradiction and is not: the model has a bias term, and least squares estimates that term without bias.

The bias in the bias-variance decomposition is the part of expected prediction error that comes from the learning algorithm’s *average* fitted model, averaged over training samples, differing from the truth. A First Model and Generalization writes it $(\mu(x) - \mu_h(x))^2$, with μ_h the average over samples. It is a property of the algorithm rather than of any one fit, and it is about predictions rather than parameters. The first and third senses are related, and the relation is the opposite of what the shared word suggests: forcing a line through the origin is a smaller family than letting it choose an intercept, so dropping the bias term generally *adds* bias in this sense.

Inductive bias is the set of assumptions a learning algorithm makes about the target function, which restrict or prioritize the models it will return. The learning theory chapter defines it, and compares algorithms by it. It is the source of the third sense rather than a synonym for it: the assumptions determine where the average fitted model lands. It is also the only one of the four that is not a number.

Ask which sense is meant every time you read the word, and name the sense every time you write it. A fifth sense, the fairness one, belongs to the ethics of deploying a model rather than to its arithmetic, and shares only the word with these four.

2 Regression and least squares

2.1 The regression task

Definition 2.1 (Regression task). Regression^a is a supervised learning task where the goal is to learn a mapping $f: \mathcal{X} \rightarrow \mathcal{Y}$ from input feature vectors to quantitative outputs [7]:

$$\mathcal{D} = \{(\tilde{x}_1, y_1), (\tilde{x}_2, y_2), \dots, (\tilde{x}_n, y_n)\} \subset \mathbb{R}^p \times \mathcal{Y}$$

Course Overview introduces regression alongside classification.

^a<https://morrison-lab.github.io/sds/correlation-regression.html>

Definition 2.2 (Univariate regression). A regression task (Definition 2.1) is **univariate** when the target is a scalar, $y \in \mathbb{R}$.

Example 2.1 (The synthetic data have a scalar target). In [the synthetic data](#), each row has one response y_i . Check that the response is a one-dimensional vector with one entry per row:

2.1.1 Python

```
n_resp_geo = len(y_geo)
print(y_geo.ndim, n_resp_geo, geo.shape[0])
```

```
1 50 50
```

2.1.2 R

```
c(is_vector = is.null(dim(geo$y)), length = length(geo$y), rows = nrow(geo))
```

```
is_vector    length    rows
      1         50      50
```

The response has 50 entries, one for each of the 50 rows, so each y_i is a scalar and the task is univariate.

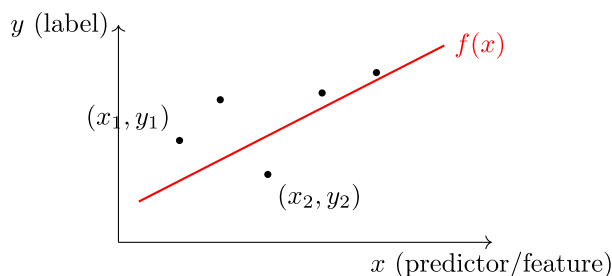


Figure 5: A simple linear regression line $\hat{y} = \beta_1 x + \beta_0$ fitted through a set of observed data points.

Definition 2.3 (Residual). For an observation (x_i, y_i) and model prediction $\hat{y}_i = f(x_i)$, the residual^a is the difference between observed and predicted values [8]:

$$\begin{aligned} e_i &= y_i - \hat{y}_i \\ &= y_i - f(x_i) \end{aligned}$$

- A positive residual ($e_i > 0$) indicates under-prediction ($y_i > \hat{y}_i$).
- A negative residual ($e_i < 0$) indicates over-prediction ($y_i < \hat{y}_i$).

^a<https://morrison-lab.github.io/sds/correlation-regression.html>

Example 2.2 (Residuals of the first five cars). Fit mpg on horsepower in the Auto data [3] by least squares, and compute the residual $e_i = y_i - \hat{y}_i$ of the first five cars:

2.1.3 Python

```
import numpy as np
import pandas as pd

auto_res = pd.read_csv("data/auto.csv")
y_res = auto_res["mpg"].to_numpy(float)
Phi_res = np.column_stack([np.ones(len(y_res)), auto_res["horsepower"]])
beta_res = np.linalg.lstsq(Phi_res, y_res, rcond=None)[0]

# Residual = observed minus predicted, so a positive value is an under-prediction.
fitted_res = Phi_res @ beta_res
resid_res = y_res - fitted_res
print(pd.DataFrame({
    "name": auto_res["name"], "mpg": y_res,
    "fitted": fitted_res.round(2), "residual": resid_res.round(2),
}).head(5))
```

		name	mpg	fitted	residual
0	chevrolet	chevelle malibu	18.0	19.42	-1.42
1	buick	skylark 320	15.0	13.89	1.11

2	plymouth satellite	18.0	16.26	1.74
3	amc rebel sst	16.0	16.26	-0.26
4	ford torino	17.0	17.84	-0.84

2.1.4 R

```
auto_res <- read.csv("data/auto.csv")
fit_res <- lm(mpg ~ horsepower, data = auto_res)

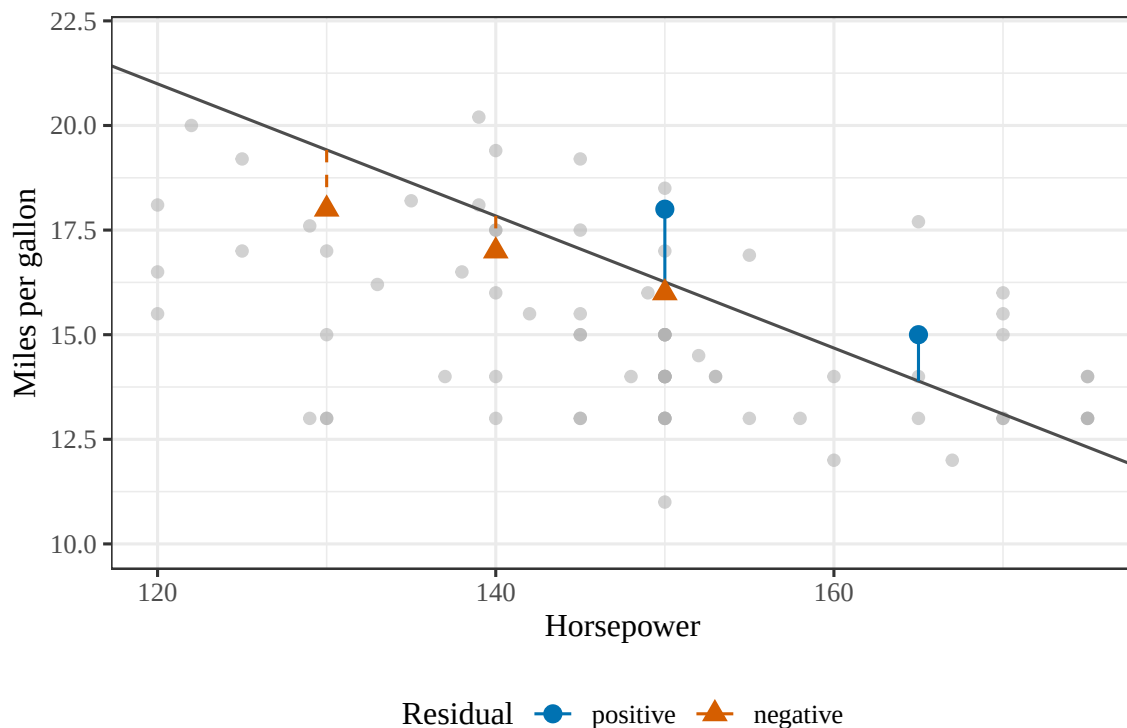
# Residual = observed minus predicted, so a positive value is an under-prediction.
head(data.frame(
  name = auto_res$name, mpg = auto_res$mpg,
  fitted = round(fitted(fit_res), 2), residual = round(residuals(fit_res), 2)
), 5)
```

```
# A tibble: 5 x 4
  name                mpg fitted residual
  <chr>              <dbl> <dbl>    <dbl>
1 chevrolet chevelle malibu    18  19.4   -1.42
2 buick skylark 320           15  13.9    1.11
3 plymouth satellite          18  16.3    1.74
4 amc rebel sst               16  16.3   -0.26
5 ford torino                 17  17.8   -0.84
```

The first car gets -1.4: its observed mpg is below the line, so the line over-predicts it, and its residual is negative. The plymouth satellite gets +1.7: the line under-predicts it, so its residual is positive.

2.1.5 R

```
library(ggplot2)
resf_five <- data.frame(
  horsepower = auto_res$horsepower[1:5],
  mpg = auto_res$mpg[1:5],
  fitted = fitted(fit_res)[1:5],
  sign = factor(
    ifelse(residuals(fit_res)[1:5] >= 0, "positive", "negative"),
    levels = c("positive", "negative")
  )
)
ggplot(auto_res, aes(horsepower, mpg)) +
  geom_point(colour = "grey70", alpha = 0.6) +
  geom_abline(intercept = coef(fit_res)[[1]], slope = coef(fit_res)[[2]], colour = "grey30") +
  geom_segment(
    data = resf_five,
    aes(xend = horsepower, y = fitted, yend = mpg, colour = sign, linetype = sign)
  ) +
  geom_point(data = resf_five, aes(colour = sign, shape = sign), size = 3) +
  scale_colour_manual(values = c(positive = "#0072B2", negative = "#D55E00")) +
  scale_shape_manual(values = c(positive = 16, negative = 17)) +
  scale_linetype_manual(values = c(positive = "solid", negative = "dashed")) +
  coord_cartesian(xlim = c(120, 175), ylim = c(10, 22)) +
  labs(
    x = "Horsepower", y = "Miles per gallon",
    colour = "Residual", shape = "Residual", linetype = "Residual"
  )
)
```



(a)

2.1.6 Python

```
import matplotlib.pyplot as plt

hp_resf = auto_res["horsepower"].to_numpy(float)
fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.plot(hp_resf, y_res, color="grey", marker="o", linestyle="", alpha=0.4, markersize=4);
```

Definition 2.4 (Squared error loss (square loss)). The **squared error loss** (or **square loss**) assigns a non-negative penalty proportional to the square of the residual:

$$L(y, f(\tilde{x})) = (y - f(\tilde{x}))^2$$

Squaring penalizes large errors much more severely than small ones and ensures that positive and negative errors cannot cancel out when summed across the dataset. Summed over the dataset, this forms the residual sum of squares (RSS)^a:

$$\text{RSS} = \sum_{i=1}^n (y_i - f(\tilde{x}_i))^2$$

^a<https://morrison-lab.github.io/sds/correlation-regression.html>

Example 2.3 (Squaring lets one large miss dominate). For the straight-line fit of `mpg` on `horsepower` in [the residual example](#), compute the residual sum of squares and ask what share of it the single worst car contributes:

2.1.7 Python

```
# Squared residuals, one per car; their sum is the RSS.
sq_loss = resid_res ** 2
rss_loss = sq_loss.sum()
worst_loss = sq_loss.argmax()
print(rss_loss, auto_res["name"][worst_loss], resid_res[worst_loss])

9385.915871932417 mazda glc 16.92404664681701

print(sq_loss[worst_loss] / rss_loss)

0.03051629258260845
```

2.1.8 R

```
# Squared residuals, one per car; their sum is the RSS.
sq_loss <- residuals(fit_res)^2
rss_loss <- sum(sq_loss)
worst_loss <- which.max(sq_loss)
c(rss = rss_loss, residual = unname(residuals(fit_res)[worst_loss]),
  share = unname(sq_loss[worst_loss] / rss_loss))
```

```
      rss      residual      share
9.385916e+03 1.692405e+01 3.051629e-02
```

The RSS is 9,386 over 392 cars. The worst car, the mazda glc, is 16.9 miles per gallon off the line, and it alone contributes 3.1% of the total. An average car is off by only 3.8, so squaring lets a few large misses outweigh many small ones.

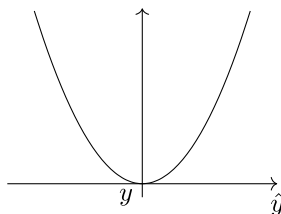


Figure 7: The vertical offsets between observed targets y_i and fitted values $\hat{y}_i$ define the residuals whose squares are summed in the least-squares objective.

i What x , y and β are called elsewhere

Regression is older than machine learning, as the course overview sets out, and the fields that used it first named its pieces first. Every name in the right-hand column is current somewhere.

Here	Also called
x , the inputs	features, predictors, covariates, regressors, independent variables, attributes
y , the outputs	targets, responses, outcomes, labels, dependent variables
β_0 and β , the parameters	coefficients, weights

The symbols move too. This chapter writes the parameters β , following the statistics convention, and keeps the intercept^a as β_0 until the [intercept trick](#) absorbs it. Machine learning texts more often write w for the same vector and b for the same intercept, and optimization texts write θ for any parameter vector at all. A source writing $h(x) = w \cdot x + b$ is writing this chapter's $h(x) = \beta^\top x + \beta_0$. Three of these names carry a distinction rather than just a dialect.

Independent variable is the name to use most carefully. The word *independent* promises that x was set rather than observed, which holds in a designed experiment and rarely holds anywhere else. You did not set x in data you merely collected, so *predictor* or *covariate* is the accurate word.

Label normally implies that y takes one of a fixed set of values. Calling a house price a label will be understood, but it will read as though you expected a classification problem.

Parameter is wider than *coefficient* and *weight*. Those two names belong to a number that multiplies an entry of $\hat{x}$, which under the 1-padding $\hat{x} = [1; x]$ includes β_0 , since it multiplies the constant 1. A parameter is any number the fitting procedure chooses, which in other models includes quantities that multiply nothing, such as a noise variance. A **hyperparameter** is not a narrower kind of parameter at all: it is fixed before the fit runs, and the fit never adjusts it. An outer model-selection procedure may still choose it, by fitting once per candidate and comparing on held-out data.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

i Watch

Andrew Ng sets up the same regression problem in the second CS229 lecture, from 0:45^a, and then sets up the least-squares algorithm, from 14:44^b [9], Lecture 2. The rest of that lecture is gradient descent, the subject of Gradient Descent.

^ahttps://www.youtube.com/watch?v=4b4MUYve_U8&t=45s

^bhttps://www.youtube.com/watch?v=4b4MUYve_U8&t=884s

i Watch

Abu-Mostafa introduces linear regression, from 18:51^a, and derives its algorithm, from 31:41^b [10], Lecture 3.

^a<https://www.youtube.com/watch?v=FIbVs5GbBIQ&t=1131s>

^b<https://www.youtube.com/watch?v=FIbVs5GbBIQ&t=1901s>

i Watch

Starmer explains:

- fitting a line by least squares^a [11]
- linear regression^b as a whole [12]

^a<https://www.youtube.com/watch?v=PaFPbb66DxQ>

^b<https://www.youtube.com/watch?v=7ArmBVF2dCs>

Read

GeeksforGeeks explains linear regression^a [13].

^a<https://www.geeksforgeeks.org/machine-learning/ml-linear-regression/>

Read

The University of Washington's CSE 446 slides cover linear regression^a and linear regression with basis functions^b [14], Lectures 3 and 4.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/03-linear%20regression.pdf>

^b<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/04-linear%20regression%20w%20basis%20funcs.pdf>

2.2 Fitting by least squares

Definition 2.5 (Ordinary least squares objective). Ordinary least squares (OLS)^a estimates the parameters $\vec{\beta}$ by minimizing the empirical risk under squared error loss [15]:

$$\begin{aligned}\hat{\vec{\beta}} &= \arg \min_{\vec{\beta}} \hat{R}(\vec{\beta}) \\ &= \arg \min_{\vec{\beta}} \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2\end{aligned}\tag{1}$$

Here $\hat{y}_i = h(\tilde{x}_i)$ is the fitted value (the prediction) for example i .

In vector notation, with targets $\tilde{y} \in \mathbb{R}^n$ and design matrix $D \in \mathbb{R}^{n \times p^*}$:

$$\begin{aligned}\hat{\vec{\beta}} &= \arg \min_{\vec{\beta}} \frac{1}{n} (\tilde{y} - D\vec{\beta})^\top (\tilde{y} - D\vec{\beta}) \\ &= \arg \min_{\vec{\beta}} \frac{1}{n} \|D\vec{\beta} - \tilde{y}\|^2\end{aligned}$$

^a<https://morrison-lab.github.io/sds/correlation-regression.html>

Example 2.4 (The objective at the least-squares fit and nearby). In the `Auto` data [3], evaluate the objective of Definition 2.5 for `mpg` on `horsepower` at the least-squares coefficients, and at coefficients that shift the intercept or the slope a little. Evaluate it both as the average of squared residuals and in the vector form $\frac{1}{n} \|D\vec{\beta} - \tilde{y}\|^2$:

2.2.1 Python

```

import numpy as np
import pandas as pd

auto_obj = pd.read_csv("data/auto.csv")
y_obj = auto_obj["mpg"].to_numpy(float)
Phi_obj = np.column_stack([np.ones(len(y_obj)), auto_obj["horsepower"]])
beta_obj = np.linalg.lstsq(Phi_obj, y_obj, rcond=None)[0]

def objective_obj(b):
    """Average squared residual, written as  $(1/n) * ||\Phi b - y||^2$ ."""
    return np.sum((Phi_obj @ b - y_obj) ** 2) / len(y_obj)

# The fit, then the fit with the intercept or the slope nudged either way.
nudges_obj = {"fit": (0, 0), "intercept +1": (1, 0), "intercept -1": (-1, 0),
              "slope +0.02": (0, 0.02), "slope -0.02": (0, -0.02)}
values_obj = {k: float(objective_obj(beta_obj + np.array(d))) for k, d in nudges_obj.items()}
print({k: round(v, 2) for k, v in values_obj.items()})

{'fit': 23.94, 'intercept +1': 24.94, 'intercept -1': 24.94, 'slope +0.02': 28.9, 'slope -0.02': 28.9}

# The average-of-squares form gives the same number.
print(np.isclose(values_obj["fit"], np.mean((y_obj - Phi_obj @ beta_obj) ** 2)))

True

```

2.2.2 R

```

auto_obj <- read.csv("data/auto.csv")
y_obj <- auto_obj$mpg
Phi_obj <- cbind(1, auto_obj$horsepower)
beta_obj <- coef(lm(mpg ~ horsepower, data = auto_obj))

#  $(1/n) * ||\Phi b - y||^2$ : the average squared residual.
objective_obj <- function(b) sum((Phi_obj %*% b - y_obj)^2) / length(y_obj)

# The fit, then the fit with the intercept or the slope nudged either way.
nudges_obj <- list(fit = c(0, 0), `intercept +1` = c(1, 0), `intercept -1` = c(-1, 0),
                  `slope +0.02` = c(0, 0.02), `slope -0.02` = c(0, -0.02))
sapply(nudges_obj, function(d) round(objective_obj(beta_obj + d), 2))

      fit intercept +1 intercept -1  slope +0.02  slope -0.02
23.94      24.94      24.94      28.90      28.90

```

The objective is 23.94 at the least-squares coefficients, and every nudge raises it, the smallest rise being to 24.94. The minimizer is the point where no direction lowers the objective.

```

# The objective of @def-least-squares-objective,
# the mean squared error of the line  $b_0 + b_1 \text{ horsepower}$ ,
# for the Auto data of @exm-least-squares-objective,
# on a grid of intercepts and slopes around the least-squares fit.
auto_surf <- read.csv("data/auto.csv")
x_surf <- auto_surf$horsepower
y_surf <- auto_surf$mpg
fit_surf <- lm(y_surf ~ x_surf)
best_b0 <- coef(fit_surf)[[1]]
best_b1 <- coef(fit_surf)[[2]]

b0_grid <- seq(best_b0 - 40, best_b0 + 40, length.out = 90)

```

```

b1_grid <- seq(best_b1 - 0.4, best_b1 + 0.4, length.out = 90)
mse_of <- function(b0, b1) mean((b0 + b1 * x_surf - y_surf)^2)
mse_grid <- outer(b0_grid, b1_grid, Vectorize(mse_of))
best_mse <- mse_of(best_b0, best_b1)

# Contour levels as multiples of the minimum,
# the same on the surface and in the contour plot.
level_mult <- c(1.25, 1.5, 2, 3, 4, 5, 6)
level_surf <- best_mse * level_mult

# The error axis stops here, past the outermost contour.
mse_axis_max <- 150

# The surface and the contour plot below both read the grid as a table.
grid_surf <- expand.grid(b0 = b0_grid, b1 = b1_grid)
grid_surf$mse <- as.vector(mse_grid)

```

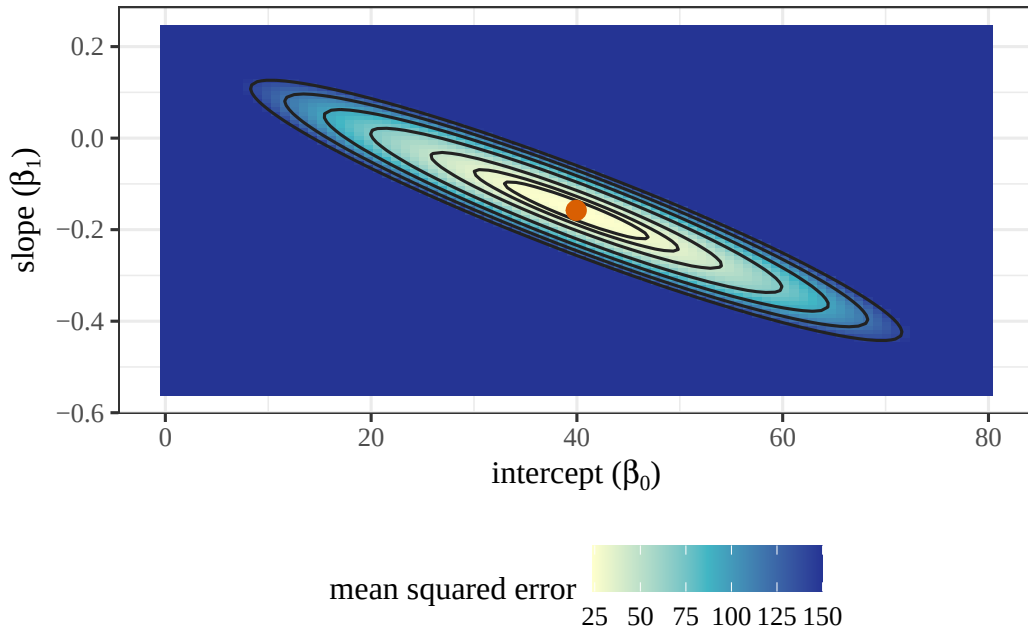


Figure 8: The sample mean squared error (sMSE) of the line $\beta_0 + \beta_1 \cdot \text{horsepower}$ for `mpg` in the `Auto` data, over a grid of the intercept β_0 and the slope β_1 . It is a bowl with one lowest point, at the least-squares fit: $\beta_0 = 39.94$, $\beta_1 = -0.1578$, sMSE 23.94. The black lines are its contours, at 1.25, 1.5, 2, 3, 4, 5, 6 times the minimum. They are long, thin, tilted ellipses, so the error changes least along the long axis, where a higher intercept is offset by a lower slope. On the course website, drag the surface to rotate it, and move the sliders, or click the surface, to choose an intercept and a slope. The blue point marks the choice on the surface and on the gradient and Hessian figures below, and the blue line shows it over the data, beside the orange dotted least-squares line.

Theorem 2.1 (The least squares objective is a convex quadratic). *The objective $\hat{R}(\vec{\beta}) = \frac{1}{n} \|D\vec{\beta} - \tilde{y}\|^2$ of Definition 2.5 is a quadratic function of $\vec{\beta}$. Its Hessian matrix is the same at every $\vec{\beta}$:*

$$\frac{2}{n} D^\top D,$$

and this matrix is positive semidefinite, so $\hat{R}$ is convex.

i Proof

Proof. By Theorem 3.2, dividing by n , $\nabla_{\vec{\beta}} \hat{R}(\vec{\beta}) = \frac{2}{n}(D^\top D \vec{\beta} - D^\top y)$. This is a linear function of $\vec{\beta}$, so $\hat{R}$ is quadratic. Differentiating once more gives the Hessian $\frac{2}{n} D^\top D$. For every $\tilde{v} \in \mathbb{R}^{p^*}$,

$$\begin{aligned} \tilde{v}^\top D^\top D \tilde{v} &= D \tilde{v} \cdot D \tilde{v} \\ &= \|D \tilde{v}\|^2 \\ &\geq 0. \end{aligned}$$

A twice-differentiable function whose Hessian is positive semidefinite everywhere is convex. $\square$

Example 2.5 (Eigenvalues of the Hessian). Take the design matrix with an intercept and the inputs x_1 and x_2 of [the synthetic data](#). Compute the Hessian $\frac{2}{n} D^\top D$ of Theorem 2.1 and its eigenvalues. A symmetric matrix is positive semidefinite exactly when all its eigenvalues are nonnegative:

2.2.3 Python

```
import numpy as np

syn_ls = r.synthetic.to_numpy()
Phi_ls = np.column_stack([np.ones(len(syn_ls)), syn_ls[:, 0], syn_ls[:, 1]])
hessian_ls = 2 / len(syn_ls) * Phi_ls.T @ Phi_ls
eig_ls = np.linalg.eigvalsh(hessian_ls)
print(eig_ls)

[1.29534302  1.74529121  2.20156948]
```

2.2.4 R

```
syn_ls <- synthetic
Phi_ls <- cbind(1, syn_ls$x1, syn_ls$x2)
hessian_ls <- 2 / nrow(Phi_ls) * crossprod(Phi_ls)
eigen(hessian_ls, symmetric = TRUE, only.values = TRUE)$values

[1] 2.201569 1.745291 1.295343
```

The Hessian has 3 eigenvalues, from 1.3 to 2.2. None is negative, so the objective is convex. All are positive, so for these data the objective is strictly convex and has a single minimizer.

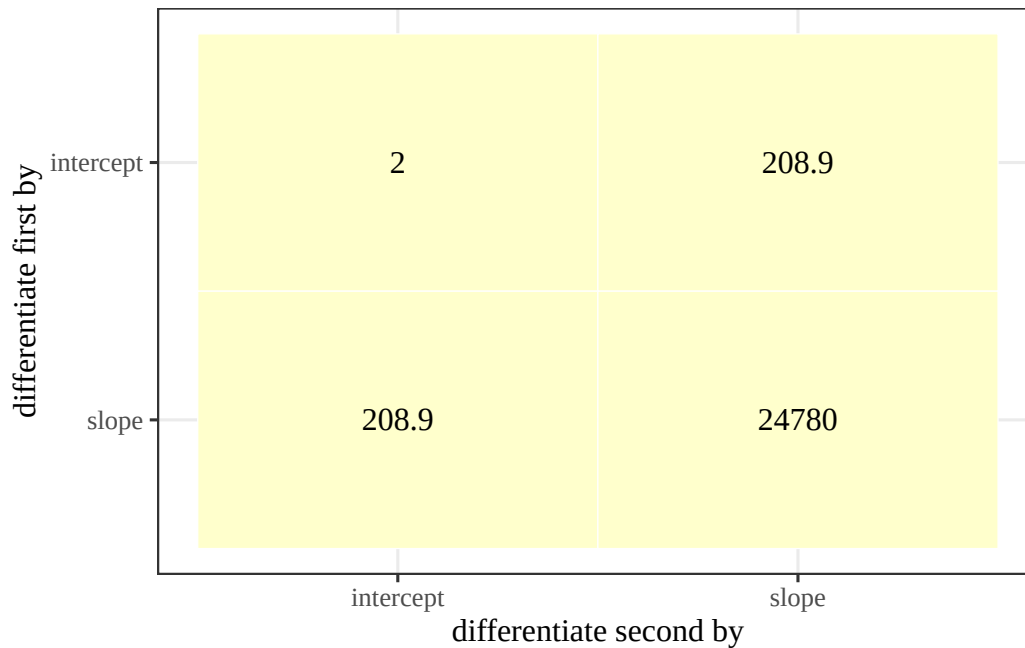


Figure 9: The Hessian of the objective of Theorem 2.1, one surface for each of its four entries, over the same intercepts and slopes as Figure 8. Every surface is flat, at the height of one entry of $\frac{2}{n} D^\top D$ (2, 208.9, 208.9, 24780, reading across the rows), because the objective is quadratic: its curvature is the same at every intercept and slope. The two off-diagonal surfaces are equal, because the matrix is symmetric. On the course website, the blue points follow the choice made in Figure 8.

i Loss, cost, objective, risk

The parts of Equation 1 each have a name, and the literature adds more. **Loss function** names the term inside the average, here $(y_i - \hat{y}_i)^2$, which scores one example; written for any loss, it is $L(y_i, \hat{y}_i)$. **Empirical risk** names the average itself over the whole sample, $\hat{R}(\vec{\beta})$. **Objective function** names whatever is actually handed to the minimizer.

Two further names overlap with those three, and both are worth defining precisely before they cause trouble.

Cost function is used for all three, so it identifies nothing on its own. The course's own notes use it for the *sum* of losses and again for that sum plus a penalty, which are the two readings furthest apart on this list. Say which quantity you mean and leave the word out of the formula.

Error function usually names the summed or averaged quantity rather than the per-example one. Bishop's *Pattern Recognition and Machine Learning* writes $E(\tilde{w})$ for the sum of squared residuals over all N examples, and that is the reading to assume unless a source says otherwise.

In Equation 1 the empirical risk, the objective and both readings of *cost* coincide, because nothing has been added to the average yet. Two of the distinctions start to matter in the chapters that follow.

The hat on $\hat{R}$ is the difference between risk and empirical risk. The **risk** $R(h)$ is the expected loss over the distribution the data came from, which is the quantity you actually care about and the one you can never compute. $\hat{R}(h)$ is its average over the n examples you have. Minimizing the second in the hope of minimizing the first is called **empirical risk minimization**, and the difference $R(h) - \hat{R}(h)$ is the generalization gap. Generalization itself is the ability to predict well on unseen data, which the gap measures rather than defines.

Objective and empirical risk separate once a penalty is added. Later chapters minimize $\hat{R}(h)$ plus a term that grows with the size of β . The objective is then the sum, the empirical risk is still only the first part, and a sentence that says “the loss went down” has stopped saying which of the two it means.

i Fitting, training, learning, estimating

Choosing β from data has four common names. Statistics calls it **fitting** a model or **estimating** its parameters. Machine learning calls it **training** a model or **learning** its parameters. No arithmetic changes between the four.

Inference names opposite ends of the process in the two fields.

Statistics uses it for reasoning from the sample to the population. The course overview gives the goal version, following *An Introduction to Statistical Learning*: inference means understanding f itself, as opposed to predicting y well. The procedures that pursue the goal carry the name too:

- estimating β ,
- putting an interval around it,
- testing a hypothesis about it.

Machine learning systems use it for running the trained model forward on a new x to get a $\hat{y}$. A sentence about “inference time” is about the phase after fitting, and the quantity that phase produces is a prediction.

The clash is sharper than a difference of dialect. The course overview sets inference *against* prediction, and a deployed system uses *inference* for prediction itself. Read the word from the speaker’s field, never from its other sense.

A **model** carries two senses as well. Sometimes it is the family of functions under consideration, here every linear function of x , and sometimes one fitted member of the family, here the single β least squares returned. Ask which is meant whenever a sentence could take either.

2.3 Setting up the design matrix

Definition 2.6 (The design matrix and 1-padding). To absorb the intercept β_0 directly into the parameter vector, we prepend a constant 1 to each input vector $\tilde{x}_i \in \mathbb{R}^p$, forming the design vector $D(\tilde{x}_i) \in \mathbb{R}^{p^*}$ with dimension $p^* = p + 1$:

$$D(\tilde{x}_i) = \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{bmatrix} \in \mathbb{R}^{p^*}, \quad \vec{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix} \in \mathbb{R}^{p^*}$$

so that $f(\tilde{x}_i) = D(\tilde{x}_i) \cdot \vec{\beta} = \beta_0 + \sum_{j=1}^p \beta_j x_{ij}$.

Stacking all n training instances as rows creates the design matrix^a $D(\mathbf{X}) \in \mathbb{R}^{n \times p^*}$ (often written simply as D) and response vector $\tilde{y} \in \mathbb{R}^n$ [16]:

$$\begin{aligned} D(\mathbf{X}) &= \begin{bmatrix} \text{---} & D(\tilde{x}_1)^\top & \text{---} \\ \text{---} & D(\tilde{x}_2)^\top & \text{---} \\ & \vdots & \\ \text{---} & D(\tilde{x}_n)^\top & \text{---} \end{bmatrix} \\ &= \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix} \in \mathbb{R}^{n \times p^*}, \end{aligned} \quad \tilde{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n$$

The matrix-vector product computes all n predictions simultaneously:

$$D\vec{\beta} = \begin{bmatrix} D(\tilde{x}_1) \cdot \vec{\beta} \\ D(\tilde{x}_2) \cdot \vec{\beta} \\ \vdots \\ D(\tilde{x}_n) \cdot \vec{\beta} \end{bmatrix} = \hat{\tilde{y}} \in \mathbb{R}^n$$

and the sum of squared errors becomes the squared Euclidean norm:

$$\sum_{i=1}^n (D(\tilde{x}_i) \cdot \vec{\beta} - y_i)^2 = \|D\vec{\beta} - \tilde{y}\|^2$$

^a<https://morrison-lab.github.io/mds/linear-algebra-projections.html#def-design-matrix>

Example 2.6 (From a data table to the design matrix). Take the first five rows of [the synthetic data](#). Each row of the table is one observation: two features, x_1 and x_2 , and a response y . The design matrix D keeps the features, adds a leading column of ones for the intercept, and drops the response. So it has one row per observation and one column per coefficient. The chunk builds it, fits the coefficients by least squares, and then computes all five predictions with one matrix-vector product.

```
import numpy as np

# Keep the first five rows of the synthetic data (defined in R above).
rows = r.synthetic.iloc[:5]
y_five = rows["y"].to_numpy()

# Design matrix: a column of ones (so the first coefficient is the intercept),
# then one column per feature. Each row is still one observation.
X_five = np.column_stack([np.ones(5), rows["x1"], rows["x2"]])

# Coefficients fitted on all 50 observations, so they are the same ones
# used in later examples; here they only serve to produce predictions.
X_all = np.column_stack([np.ones(len(r.synthetic)), r.synthetic["x1"], r.synthetic["x2"]])
beta_hat = np.linalg.solve(X_all.T @ X_all, X_all.T @ r.synthetic["y"].to_numpy())

# One matrix-vector product gives the prediction for every row at once.
y_hat_five = X_five @ beta_hat

# The same five predictions, one dot product per observation, to show they agree.
y_hat_loop = np.array([np.dot(row, beta_hat) for row in X_five])
print(X_five.shape, beta_hat.shape, y_hat_five.shape)

(5, 3) (3,) (5,)

print(np.allclose(y_hat_five, y_hat_loop))
```

True

Figure 10 draws the three arrays side by side. Each row of D , times the coefficient vector, gives the entry in the same row of $\hat{y}$.

```

import matplotlib.pyplot as plt

def show_table(ax, values, col_labels, title):
    """Draw a small array as a grid of cells, printing each value inside its cell."""
    ax.imshow(values, cmap="Blues", aspect="auto", vmin=-4, vmax=14);
    for i in range(values.shape[0]):
        for j in range(values.shape[1]):
            ax.text(j, i, f"{values[i, j]:.2f}", ha="center", va="center", fontsize=8);
    ax.set_xticks(range(len(col_labels)), col_labels);
    ax.set_yticks([]);
    ax.set_title(title, fontsize=10);

fig, axes = plt.subplots(
    1, 5, figsize=(9, 2.8), layout="constrained",
    gridspec_kw={"width_ratios": [3, 0.35, 1, 0.35, 1]},
)
show_table(axes[0], X_five, ["1", "$x_1$", "$x_2$"], r"design matrix  $D$ ")
show_table(axes[2], beta_hat.reshape(-1, 1), [""], r"coefficients  $\hat{\beta}$ ")
show_table(axes[4], y_hat_five.reshape(-1, 1), [""], r"predictions  $\hat{y}$ ")
# The operators between the tables read "times" and "equals".
for ax, sym in [(axes[1], r"$\times$"), (axes[3], r"$=$")]:
    ax.axis("off");
    ax.text(0.5, 0.5, sym, ha="center", va="center", fontsize=16, transform=ax.transAxes);
plt.show()

```

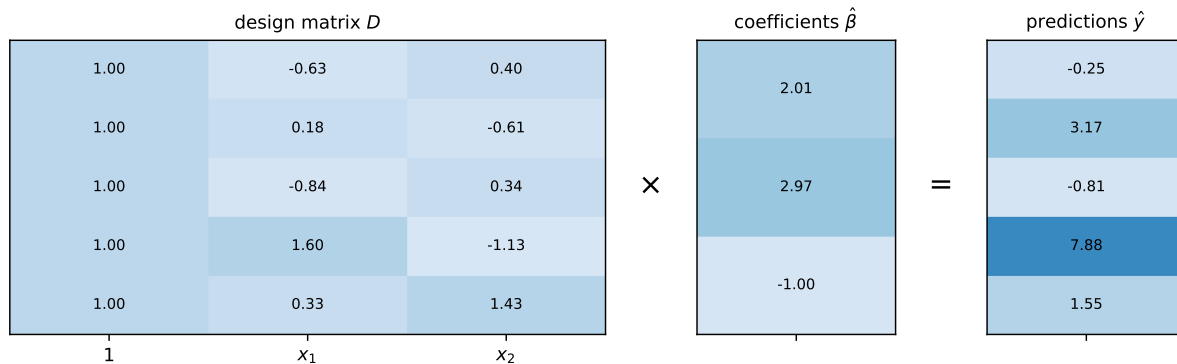


Figure 10: The design matrix of the first five synthetic observations (left, with its column of ones), the fitted coefficient vector (middle), and the predictions that the product gives (right). Each row on the left, combined with the middle column, makes the same row on the right.

Stacking the observations this way lets us write one equation, $D\vec{\beta} = \hat{y}$, instead of n separate ones. It also lets us write the sum of squared errors as $\|D\vec{\beta} - \hat{y}\|^2$ and solve for $\vec{\beta}$ with linear algebra.

Example 2.7 (Extracting the design matrix from a fitted model). Fitting a model with a formula builds the design matrix for you. Both R and Python let you get back the matrix that the fit used, which is the safest way to check that a formula means what you intended: the columns are exactly the features, in order, that the coefficients multiply. Fit $y \sim x_1 + x_2$ to [the synthetic data](#), extract the design matrix from the fitted model, and check that it equals the matrix that we built by hand in Example 2.6.

2.3.1 Python

`statsmodels` stores the design matrix in the fitted model's `model` attribute: `fit.model.exog` is the array, and `fit.model.exog_names` names its columns. The `patsy` package, which `statsmodels` uses to read formulas, builds the same matrix directly from a formula and a data frame with `dmatrix()`.

```
import numpy as np
import statsmodels.formula.api as smf
from patsy import dmatrix

# Fit by least squares; the formula adds the intercept column automatically.
fit_syn_py = smf.ols("y ~ x1 + x2", data=r.synthetic).fit()

# The design matrix that the fit used, and the name of each column.
X_fit = fit_syn_py.model.exog
print(fit_syn_py.model.exog_names)

['Intercept', 'x1', 'x2']

print(X_fit.shape)

(50, 3)

# patsy builds the same matrix without fitting anything, which is useful
# for inspecting a formula before spending time on a model.
X_patsy = np.asarray(dmatrix("x1 + x2", data=r.synthetic))

# The hand-built matrix: a column of ones, then one column per feature.
X_hand = np.column_stack([np.ones(len(r.synthetic)), r.synthetic["x1"], r.synthetic["x2"]])
print(np.allclose(X_fit, X_hand), np.allclose(X_patsy, X_hand))

True True
```

2.3.2 R

`model.matrix()` takes the fitted model, and returns the design matrix with named columns and one row per observation. It also takes a formula and a data frame, so it can build the matrix without fitting anything.

```
# Fit by least squares; the formula adds the intercept column automatically.
fit_syn_r <- lm(y ~ x1 + x2, data = synthetic)

# The design matrix that the fit used; its column names label the coefficients.
X_fit_r <- model.matrix(fit_syn_r)
colnames(X_fit_r)

[1] "(Intercept)" "x1"          "x2"

dim(X_fit_r)

[1] 50 3

# The same matrix from the formula alone, without fitting anything.
X_formula_r <- model.matrix(~ x1 + x2, data = synthetic)

# The hand-built matrix: a column of ones, then one column per feature.
X_hand_r <- cbind(1, synthetic$x1, synthetic$x2)
all.equal(unname(X_fit_r), X_hand_r, check.attributes = FALSE)

[1] TRUE

all.equal(X_fit_r, X_formula_r)

[1] TRUE
```

The matrix has 50 rows, one per observation, and 3 columns: the intercept (`Intercept`) or `Intercept`,

$\mathbf{x1}$ and $\mathbf{x2}$. All three routes give the same numbers, so D is the matrix the formula builds, and the fit hides no step from us. The extracted matrix is also the quickest way to see how a formula encodes a categorical predictor or an interaction, since each of those adds columns that the coefficients then multiply.

Exercise 2.1 (Shapes). Take n training points in p input dimensions, forming design vectors $\tilde{D}(x_i) \in \mathbb{R}^{p^*}$ with dimension $p^* = p + 1$ under the [padding](#) of Exercise 1.1. Give the shape of each of $\tilde{D}(x_i)$, $\tilde{\beta}$, D , y , $D\tilde{\beta}$, $D^\top D$, $D^\top y$, and $(D^\top D)^{-1}D^\top y$.

Solution

Solution 2.1.

object	shape
$\tilde{D}(x_i)$	$p^* \times 1$
$\tilde{\beta}$	$p^* \times 1$
D	$n \times p^*$
y	$n \times 1$
$D\tilde{\beta}$	$n \times 1$
$D^\top D$	$p^* \times p^*$
$D^\top y$	$p^* \times 1$
$(D^\top D)^{-1}D^\top y$	$p^* \times 1$

The last line is worth checking rather than assuming: it has to come out $p^* \times 1$, because it *is* $\tilde{\beta}$. A formula for $\tilde{\beta}$ with any other shape is wrong before it is evaluated.

Exercise 2.2 (Shapes (part 2)). Use the setup of Exercise 2.1. Which of those shapes depend on n , and which do not?

Solution

Solution 2.2. Only D , y , and $D\tilde{\beta}$ carry n . Every object on the solving side — $D^\top D$, $D^\top y$, and $\tilde{\beta}$ itself — is sized by p^* (and thus by p) alone.

That is the structural fact behind the whole method. A million training points in two dimensions still produce a 3×3 system. The data enters only through the sums $D^\top D$ and $D^\top y$, after which the number of rows n no longer appears.

Exercise 2.3 (Shapes (part 3)). Use the setup of Exercise 2.1. Write D out in full for these four points, with $p = 2$ ($p^* = 3$):

x_{i1}	x_{i2}	y_i
1	0	3
2	1	5
0	4	2
3	3	9

Solution

Solution 2.3. Four rows, each a 1 followed by that point's inputs^a:

$$D = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 2 & 1 \\ 1 & 0 & 4 \\ 1 & 3 & 3 \end{bmatrix}_{4 \times 3} \quad y = \begin{bmatrix} 3 \\ 5 \\ 2 \\ 9 \end{bmatrix}_{4 \times 1}$$

Here $n = 4 \geq p^* = 3$ (and $n > p = 2$), so this one can be solvable. That is a necessary condition rather than a sufficient one: two columns that are linearly dependent leave $D^\top D$ singular however many rows there are.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

i Watch

- Hastie and Tibshirani fit a line by least squares in their *Statistical Learning* course, from 5:26^a [17], video 3.1.
- Hastie and Tibshirani extend least squares to several predictors^b [17], video 3.3.
- Starmer builds design matrices^c [18] and explains multiple regression^d [19].

^a<https://www.youtube.com/watch?v=vCHtY6Me5FI&t=326s>

^b<https://www.youtube.com/watch?v=o9hoLdylWko>

^c<https://www.youtube.com/watch?v=CqLGvwi-5Pc>

^d<https://www.youtube.com/watch?v=zITIFTsivN8>

3 Deriving the solution

3.1 The gradient

Definition 3.1 (The gradient). For a multivariable objective function $\hat{R} : \mathbb{R}^{p^*} \rightarrow \mathbb{R}$, the gradient^a is the vector of its first-order partial derivatives:

$$\nabla_{\vec{\beta}} \hat{R}(\vec{\beta}) = \begin{bmatrix} \frac{\partial \hat{R}}{\partial \beta_0} \\ \frac{\partial \hat{R}}{\partial \beta_1} \\ \vdots \\ \frac{\partial \hat{R}}{\partial \beta_p} \end{bmatrix} \in \mathbb{R}^{p^*}$$

Geometrically, the gradient points in the direction of steepest instantaneous ascent on the objective surface, and its magnitude equals the directional derivative along that path [20]. A stationary point is any $\vec{\beta}^*$ where the gradient vanishes: $\nabla_{\vec{\beta}} \hat{R}(\vec{\beta}^*) = \vec{0}$.

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-vector-derivative>

Example 3.1 (The gradient points uphill). In the `Auto` data [3], take the objective of `mpg` on `horsepower`, a function of the two coefficients (β_0, β_1) . Estimate its gradient at $\vec{\beta} = (30, -0.1)$ with finite differences, which nudge one coefficient at a time and divide the change in the objective by the size of the nudge. Then take a small step along the gradient, and a small step against it:

3.1.1 Python

```

import numpy as np
import pandas as pd

auto_grad = pd.read_csv("data/auto.csv")
y_grad = auto_grad["mpg"].to_numpy(float)
Phi_grad = np.column_stack([np.ones(len(y_grad)), auto_grad["horsepower"]])

def risk_grad(b):
    """The average squared residual, as a function of the coefficients."""
    return np.mean((y_grad - Phi_grad @ b) ** 2)

b0_grad = np.array([30.0, -0.1])
eps_grad = 1e-6

# One partial derivative per coefficient: nudge that coefficient alone.
grad_hat = np.array([
    (risk_grad(b0_grad + eps_grad * e) - risk_grad(b0_grad)) / eps_grad
    for e in np.eye(2)
])

# A step of length 0.001 along the unit vector in the gradient's direction, and against it.
unit_grad = grad_hat / np.linalg.norm(grad_hat)
step_up = risk_grad(b0_grad + 0.001 * unit_grad)
step_down = risk_grad(b0_grad - 0.001 * unit_grad)
print(grad_hat, risk_grad(b0_grad), step_up, step_down)

[ -7.78571329 -642.39168999] 44.04270408163266 44.697547696594 43.412645176641256

```

3.1.2 R

```

auto_grad <- read.csv("data/auto.csv")
y_grad <- auto_grad$mpg
Phi_grad <- cbind(1, auto_grad$horsepower)

# The average squared residual, as a function of the coefficients.
risk_grad <- function(b) mean((y_grad - Phi_grad %*% b)^2)

b0_grad <- c(30, -0.1)
eps_grad <- 1e-6

# One partial derivative per coefficient: nudge that coefficient alone.
grad_hat <- sapply(1:2, function(j) {
  e <- diag(2)[, j]
  (risk_grad(b0_grad + eps_grad * e) - risk_grad(b0_grad)) / eps_grad
})

# A step of length 0.001 along the unit vector in the gradient's direction, and against it.
unit_grad <- grad_hat / sqrt(sum(grad_hat^2))
c(gradient = grad_hat, at_b0 = risk_grad(b0_grad),
  step_along = risk_grad(b0_grad + 0.001 * unit_grad),
  step_against = risk_grad(b0_grad - 0.001 * unit_grad))

```

gradient1	gradient2	at_b0	step_along	step_against
-7.785713	-642.391690	44.042704	44.697548	43.412645

The gradient is (-7.8, -642.4), so the objective is falling in both coefficients as they increase. A step of length 0.001 along the gradient raises the objective from 44.0427 to 44.6975; the same step against it

lowers it to 43.4126. The rise per unit of step is 655, close to the gradient's length, 642, as the definition says.

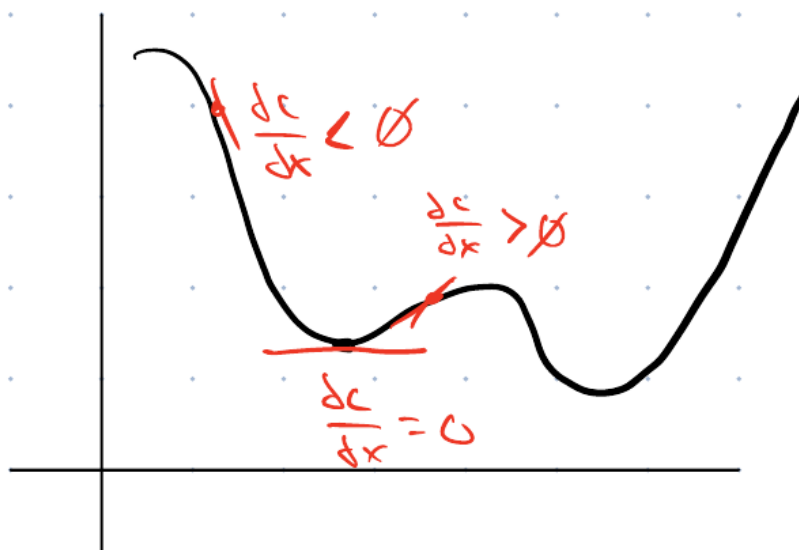


Figure 11: Contour lines of an objective surface showing the gradient vector $\nabla_{\hat{\beta}} \hat{R}$ pointing perpendicular to the tangent of the level sets toward increasing loss.

Exercise 3.1 (Two gradients by hand). Let $f(\hat{\beta}) = 2\beta_0 + 3\beta_1 - \beta_1\beta_2$. Compute $\nabla_{\hat{\beta}} f$.

Solution

Solution 3.1. Differentiate with respect to each coordinate in turn, treating the others as constants:

$$\nabla_{\hat{\beta}} f = \begin{bmatrix} \partial f / \partial \beta_0 \\ \partial f / \partial \beta_1 \\ \partial f / \partial \beta_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 - \beta_2 \\ -\beta_1 \end{bmatrix}$$

Exercise 3.2 (Two gradients by hand (part 2)). Use the setup of Exercise 3.1.

Let $J(z_1, z_2) = 3z_1^2 + 4z_1z_2 + z_2^2$. Compute $\nabla_{\tilde{z}} J$, find every point where it vanishes, and decide whether that point is a minimum.

Solution

Solution 3.2.

$$\nabla_{\tilde{z}} J = \begin{bmatrix} \partial J / \partial z_1 \\ \partial J / \partial z_2 \end{bmatrix} = \begin{bmatrix} 6z_1 + 4z_2 \\ 4z_1 + 2z_2 \end{bmatrix}$$

Setting both components to zero gives $6z_1 + 4z_2 = 0$ and $4z_1 + 2z_2 = 0$. The second gives $z_2 = -2z_1$; substituting into the first gives

$$\begin{aligned} 6z_1 - 8z_1 &= -2z_1 \\ &= 0, \end{aligned}$$

so $z_1 = 0$ and then $z_2 = 0$. The only critical point is $(0, 0)$.

It is not a minimum. A vanishing gradient says only that the point is *critical*. The matrix of second derivatives here is

$$\begin{bmatrix} 6 & 4 \\ 4 & 2 \end{bmatrix}, \quad \begin{aligned} \det &= 6 \cdot 2 - 4 \cdot 4 \\ &= -4 \\ &< 0 \end{aligned}$$

and a negative determinant means^a the surface curves up along one direction and down along another. So $(0, 0)$ is a **saddle point**^b.

Check it directly if the determinant test is unfamiliar: $J(t, 0) = 3t^2 > 0$ while

$$\begin{aligned} J(t, -2t) &= 3t^2 - 8t^2 + 4t^2 \\ &= -t^2 < 0, \end{aligned}$$

so every neighbourhood of the origin contains points where J is larger and points where it is smaller. So solving $\nabla J = \vec{0}$ finds candidates rather than answers. For the error surface of least squares, convexity resolves this.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^b<https://morrison-lab.github.io/mds/optimization.html#def-saddle-point>

Theorem 3.1 (Minimizing a smooth convex function). *Let $g : \mathbb{R}^p \rightarrow \mathbb{R}$ be a smooth convex function and $\mathcal{C} \subseteq \mathbb{R}^p$ a convex set. Then any $\tilde{x}^*$ minimizing g over $\mathcal{C}$ lies either on the boundary of $\mathcal{C}$ or at an unconstrained stationary point where $\nabla g(\tilde{x}^*) = \vec{0}$.*

Conversely, if g is convex, any interior point $\tilde{x}^*$ with $\nabla g(\tilde{x}^*) = \vec{0}$ is a global minimizer of g over $\mathcal{C}$.

i Proof

Proof. For any $\tilde{x} \in \mathcal{C}$, first-order convexity yields $g(\tilde{x}) \geq g(\tilde{x}^*) + \nabla g(\tilde{x}^*) \cdot \tilde{x} - \tilde{x}^*$. When $\nabla g(\tilde{x}^*) = \vec{0}$, the linear term vanishes, giving $g(\tilde{x}) \geq g(\tilde{x}^*)$ for all $\tilde{x} \in \mathcal{C}$. $\square$

Example 3.2 (The gradient vanishes at the minimizer). The objective of `mpg` on `horsepower` in the `Auto` data [3] is a smooth convex function of the two coefficients. By Theorem 3.1, at its minimizer the gradient is zero, and no other point does better. Estimate the gradient at the least-squares coefficients with finite differences, and compare the objective there with its value at the eight points of a grid around it:

3.1.3 Python

```

import numpy as np
import pandas as pd

auto_cm = pd.read_csv("data/auto.csv")
y_cm = auto_cm["mpg"].to_numpy(float)
Phi_cm = np.column_stack([np.ones(len(y_cm)), auto_cm["horsepower"]])
beta_cm = np.linalg.lstsq(Phi_cm, y_cm, rcond=None)[0]

def risk_cm(b):
    return np.mean((y_cm - Phi_cm @ b) ** 2)

# A central difference nudges a coefficient both ways, which is accurate near a minimum.
eps_cm = 1e-5
grad_cm = np.array([
    (risk_cm(beta_cm + eps_cm * e) - risk_cm(beta_cm - eps_cm * e)) / (2 * eps_cm)
    for e in np.eye(2)
])

# The eight neighbours on a grid around the minimizer (the centre is the minimizer itself).
steps_cm = [(a, b) for a in (-1, 0, 1) for b in (-1, 0, 1) if (a, b) != (0, 0)]
gaps_cm = [risk_cm(beta_cm + np.array([0.5 * a, 0.005 * b])) - risk_cm(beta_cm) for a, b in steps_cm]
print(grad_cm, min(gaps_cm))

[0. 0.] 0.037444132653064344

```

3.1.4 R

```

auto_cm <- read.csv("data/auto.csv")
y_cm <- auto_cm$mpg
Phi_cm <- cbind(1, auto_cm$horsepower)
beta_cm <- coef(lm(mpg ~ horsepower, data = auto_cm))

risk_cm <- function(b) mean((y_cm - Phi_cm %*% b)^2)

# A central difference nudges a coefficient both ways, which is accurate near a minimum.
eps_cm <- 1e-5
grad_cm <- sapply(1:2, function(j) {
  e <- diag(2)[, j]
  (risk_cm(beta_cm + eps_cm * e) - risk_cm(beta_cm - eps_cm * e)) / (2 * eps_cm)
})

# The eight neighbours on a grid around the minimizer (the centre is the minimizer itself).
grid_cm <- expand.grid(a = -1:1, b = -1:1)
grid_cm <- grid_cm[!(grid_cm$a == 0 & grid_cm$b == 0), ]
gaps_cm <- mapply(function(a, b) risk_cm(beta_cm + c(0.5 * a, 0.005 * b)) - risk_cm(beta_cm),
  grid_cm$a, grid_cm$b)
c(gradient = grad_cm, smallest_gap = min(gaps_cm))

      gradient1      gradient2  smallest_gap
0.000000e+00 -1.776357e-10  3.744413e-02

```

The estimated gradient is (0.0e+00, -1.8e-10), zero up to the error of the finite difference. Every one of the eight neighbours has a larger objective than the minimizer, the smallest increase being 0.037. The converse in the theorem makes this useful. A convex objective with a zero gradient has no better point anywhere, so we can find the fit by solving for the zero.

Exercise 3.3 (The gradient of the squared error, one point at a time). Take a single training point, so the empirical risk is just

$$J(\vec{\beta}) = (\tilde{D}(\tilde{x}) \cdot \vec{\beta} - y)^2$$

with design vector $\tilde{D}(\tilde{x}) \in \mathbb{R}^{p^*}$ and coefficients $\vec{\beta} \in \mathbb{R}^{p^*}$ ($p^* = p + 1$).

Compute $\frac{\partial J}{\partial \beta_k}$ for a general $k \in \{0, 1, \dots, p\}$.

Solution

Solution 3.3. Name the **residual** $r = \tilde{D}(\tilde{x}) \cdot \vec{\beta} - y$, so $J = r^2$ and the chain rule gives

$$\frac{\partial J}{\partial \beta_k} = 2r \cdot \frac{\partial r}{\partial \beta_k}$$

Now $r = (\sum_{i=0}^p \tilde{D}(\tilde{x})_i \beta_i) - y$, and β_k appears in exactly one term of that sum and nowhere else — not in y , and not in any $\tilde{D}(\tilde{x})_i$, which are data. So

$$\begin{aligned} \frac{\partial r}{\partial \beta_k} &= \frac{\partial}{\partial \beta_k} \sum_{i=0}^p \tilde{D}(\tilde{x})_i \beta_i \\ &= \tilde{D}(\tilde{x})_k \end{aligned}$$

and therefore

$$\frac{\partial J}{\partial \beta_k} = 2(\tilde{D}(\tilde{x}) \cdot \vec{\beta} - y) \tilde{D}(\tilde{x})_k$$

Exercise 3.4 (The gradient of the squared error, one point at a time (part 2)). Use the setup of Exercise 3.3.

Assemble the components into $\nabla_{\vec{\beta}} J$, and write the answer without any sums or indices.

Solution

Solution 3.4. Stacking those p^* components:

$$\nabla_{\vec{\beta}} J = \begin{bmatrix} 2(\tilde{D}(\tilde{x}) \cdot \vec{\beta} - y) \tilde{D}(\tilde{x})_0 \\ 2(\tilde{D}(\tilde{x}) \cdot \vec{\beta} - y) \tilde{D}(\tilde{x})_1 \\ \vdots \\ 2(\tilde{D}(\tilde{x}) \cdot \vec{\beta} - y) \tilde{D}(\tilde{x})_p \end{bmatrix}$$

Every entry carries the same scalar factor $2(\tilde{D}(\tilde{x}) \cdot \vec{\beta} - y)$, so we can pull it out front, leaving the vector $[\tilde{D}(\tilde{x})_0, \tilde{D}(\tilde{x})_1, \dots, \tilde{D}(\tilde{x})_p]^\top$, which is $\tilde{D}(\tilde{x})$:

$$\nabla_{\vec{\beta}} J = 2(\tilde{D}(\tilde{x}) \cdot \vec{\beta} - y) \tilde{D}(\tilde{x})$$

The whole gradient is the residual times the design vector, scaled by two. It points along $\tilde{D}(\tilde{x})$, with a sign set by whether the model over- or under-predicted at this point — which is the rule gradient descent will use throughout these notes.

Exercise 3.5 (The gradient of the squared error, one point at a time (part 3)). Use the setup of Exercise 3.3.

The intercept^a β_0 had its own symbol and its own place in every formula before the **padding** was

introduced. Where did its special case go?

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

Solution

Solution 3.5. It went into $D(\tilde{x})_0 = 1$.

Setting $k = 0$ in Solution 3.3 gives $\partial J / \partial \beta_0 = 2(D(\tilde{x}) \cdot \vec{\beta} - y) \cdot 1$, which is the $k = 0$ instance of the general formula rather than an exception to it. The intercept^a is an ordinary coefficient on a feature that happens to be constant^b, exactly as Exercise 1.1 set up.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Lemma 3.1 (Gradient of the squared error at one point). *For a design vector $D(\tilde{x}) \in \mathbb{R}^{p^*}$, an outcome y and coefficients $\vec{\beta} \in \mathbb{R}^{p^*}$, the gradient of $J(\vec{\beta}) = (D(\tilde{x}) \cdot \vec{\beta} - y)^2$ is*

$$\nabla_{\vec{\beta}} J = 2(D(\tilde{x}) \cdot \vec{\beta} - y) D(\tilde{x}).$$

i Proof

Proof. By Solution 3.4. □

Example 3.3 (The gradient at one point). Take the design vector $D(\tilde{x}) = (1, 2)$, the outcome $y = 1$ and the coefficients $\vec{\beta} = (0, 1)$. Lemma 3.1 gives the gradient $2(D(\tilde{x}) \cdot \vec{\beta} - y) D(\tilde{x})$:

```
phi_gop <- c(1, 2)
b_gop <- c(0, 1)
y_gop <- 1
2 * (sum(phi_gop * b_gop) - y_gop) * phi_gop
```

[1] 2 4

The residual is 1, so the gradient is twice the design vector.

Exercise 3.6 (From one point to all of them). Now take all n training points, so

$$\begin{aligned} J(\vec{\beta}) &= \|D\vec{\beta} - y\|^2 \\ &= \sum_{i=1}^n (D(\tilde{x}_i) \cdot \vec{\beta} - y_i)^2 \end{aligned}$$

Using Exercise 3.3, write $\nabla_{\vec{\beta}} J$ as a sum over the training points.

Solution

Solution 3.6. Differentiation is linear, so the gradient of a sum is the sum of the gradients, and Lemma 3.1 gives each term, with $D(\tilde{x}_i), y_i$ in place of $D(\tilde{x}), y$:

$$\nabla_{\vec{\beta}} J = \sum_{i=1}^n 2(D(\tilde{x}_i) \cdot \vec{\beta} - y_i) D(\tilde{x}_i)$$

The factor $(D(\tilde{x}_i) \cdot \vec{\beta} - y_i)$ is a scalar, so it commutes with $D(\tilde{x}_i)$ and the sum splits:

$$\begin{aligned}\nabla_{\vec{\beta}} J &= 2 \sum_{i=1}^n \tilde{D}(\tilde{x}_i) (\tilde{D}(\tilde{x}_i) \cdot \vec{\beta}) - 2 \sum_{i=1}^n \tilde{D}(\tilde{x}_i) y_i \\ &= 2 \left(\sum_{i=1}^n \tilde{D}(\tilde{x}_i) \tilde{D}(\tilde{x}_i)^\top \right) \vec{\beta} - 2 \sum_{i=1}^n \tilde{D}(\tilde{x}_i) y_i\end{aligned}$$

The step that moved $\vec{\beta}$ outside the sum is associativity: $\tilde{D}(\tilde{x}_i)(\tilde{D}(\tilde{x}_i) \cdot \vec{\beta}) = (\tilde{D}(\tilde{x}_i)\tilde{D}(\tilde{x}_i)^\top)\vec{\beta}$, the left side reading as a vector times a scalar and the right as a $p^* \times p^*$ matrix times a vector. They are the same number in every coordinate, and only the second groups the $\vec{\beta}$ out of the sum.

Lemma 3.2 (Gradient of the squared error over all points). *For design vectors $\tilde{D}(\tilde{x}_1), \dots, \tilde{D}(\tilde{x}_n)$ and outcomes $y_1, \dots, y_n$, the gradient of $J(\vec{\beta}) = \sum_{i=1}^n (\tilde{D}(\tilde{x}_i) \cdot \vec{\beta} - y_i)^2$ is*

$$\nabla_{\vec{\beta}} J = 2 \left(\sum_{i=1}^n \tilde{D}(\tilde{x}_i) \tilde{D}(\tilde{x}_i)^\top \right) \vec{\beta} - 2 \sum_{i=1}^n \tilde{D}(\tilde{x}_i) y_i.$$

i Proof

Proof. By Solution 3.6. □

Example 3.4 (The gradient at two points). Take the design vectors (1, 2) and (1, 3), the outcomes 1 and 2, and the coefficients $\vec{\beta} = (0, 1)$. Lemma 3.2 gives the gradient from the two sums:

```
phi_gap <- rbind(c(1, 2), c(1, 3))
y_gap <- c(1, 2)
b_gap <- c(0, 1)
sum_outer_gap <- tcrossprod(phi_gap[1, ], ) + tcrossprod(phi_gap[2, ], )
sum_y_gap <- phi_gap[1, ] * y_gap[1] + phi_gap[2, ] * y_gap[2]
2 * (sum_outer_gap %*% b_gap - sum_y_gap)
```

```
      [,1]
[1,]      4
[2,]     10
```

Exercise 3.7 (From one point to all of them (part 2)). Use the setup of Exercise 3.6.

Prove the two identities $\sum_{i=1}^n \tilde{D}(\tilde{x}_i) \tilde{D}(\tilde{x}_i)^\top = D^\top D$ and $\sum_{i=1}^n \tilde{D}(\tilde{x}_i) y_i = D^\top y$, by comparing one entry of each side.

Solution

Solution 3.7. The rows of D are the $\tilde{D}(\tilde{x}_i)^\top$, so $D_{ij} = (\tilde{D}(\tilde{x}_i))_j$, and the columns of $D^\top$ are the $\tilde{D}(\tilde{x}_i)$. Compare entry (j, k) :

$$\begin{aligned}
(D^\top D)_{jk} &= \sum_{i=1}^n (D^\top)_{ji} D_{ik} \\
&= \sum_{i=1}^n D_{ij} D_{ik} \\
&= \sum_{i=1}^n (D(\tilde{x}_i))_j (D(\tilde{x}_i))_k \\
&= \left(\sum_{i=1}^n D(\tilde{x}_i) D(\tilde{x}_i)^\top \right)_{jk}
\end{aligned}$$

and entry j of the second:

$$\begin{aligned}
(D^\top \tilde{y})_j &= \sum_{i=1}^n (D^\top)_{ji} y_i \\
&= \sum_{i=1}^n D_{ij} y_i \\
&= \sum_{i=1}^n (D(\tilde{x}_i))_j y_i \\
&= \left(\sum_{i=1}^n D(\tilde{x}_i) y_i \right)_j
\end{aligned}$$

Both hold entry by entry, so both hold as matrices.

Lemma 3.3 (Sums of design vectors as matrix products). *Let D be the design matrix, whose i th row is $D(\tilde{x}_i)^\top$, and let $\tilde{y} = (y_1, \dots, y_n)$. Then*

$$\sum_{i=1}^n D(\tilde{x}_i) D(\tilde{x}_i)^\top = D^\top D \quad \text{and} \quad \sum_{i=1}^n D(\tilde{x}_i) y_i = D^\top \tilde{y}.$$

i Proof

Proof. By Solution 3.7. □

Example 3.5 (The two sums as matrix products). With the design vectors (1, 2) and (1, 3) and the outcomes 1 and 2 of Example 3.4, Lemma 3.3 equates the two sums with matrix products^a. The sums computed one point at a time match the products:

```

phi_gi <- rbind(c(1, 2), c(1, 3))
y_gi <- c(1, 2)
sum_outer_gi <- tcrossprod(phi_gi[1, ]) + tcrossprod(phi_gi[2, ])
sum_outer_gi

```

```

      [,1] [,2]
[1,]     2     5
[2,]     5    13

```

```

all.equal(sum_outer_gi, t(phi_gi) %*% phi_gi)

```

```

[1] TRUE

```

```

all.equal(phi_gi[1, ] * y_gi[1] + phi_gi[2, ] * y_gi[2], drop(t(phi_gi) %*% y_gi))

```

```

[1] TRUE

```

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

Exercise 3.8 (From one point to all of them (part 3)). Use the setup of Exercise 3.6. Use them to reach a form with no sum in it, and factor the result. Check the shape of every factor.

Solution

Solution 3.8. Substituting,

$$\nabla_{\vec{\beta}} J = 2D^\top D\vec{\beta} - 2D^\top \underset{\sim}{y} = 2D^\top (D\vec{\beta} - \underset{\sim}{y})$$

Shapes: $D\vec{\beta} - \underset{\sim}{y}$ is $n \times 1$ — fitted minus observed, so the negative of the residual $y - D\vec{\beta}$; $D^\top$ is $p^* \times n$; the product is $\tilde{p}^* \times 1$, which is the shape a gradient with respect to $\vec{\beta}$ must have.

The factored form says something the summed form hides. The gradient vanishes exactly when $D^\top (D\vec{\beta} - \underset{\sim}{y}) = \vec{0}$ — that is, when the residual vector is orthogonal^a to every column of D . The sign does not matter to that statement: a vector is orthogonal to a column exactly when its negative is.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-orthogonal-matrix>

3.2 The gradient for squared error

Theorem 3.2 (Gradient of the least-squares objective). *Let*

$$\begin{aligned} J(\vec{\beta}) &= \|D\vec{\beta} - \underset{\sim}{y}\|^2 \\ &= (\underset{\sim}{y} - D\vec{\beta})^\top (\underset{\sim}{y} - D\vec{\beta}) \end{aligned}$$

for design matrix $D \in \mathbb{R}^{n \times p^*}$, response $\underset{\sim}{y} \in \mathbb{R}^n$, and coefficients $\vec{\beta} \in \mathbb{R}^{p^*}$. The gradient of J with respect to $\vec{\beta}$ is:

$$\begin{aligned} \nabla_{\vec{\beta}} J(\vec{\beta}) &= 2D^\top (D\vec{\beta} - \underset{\sim}{y}) \\ &= 2D^\top D\vec{\beta} - 2D^\top \underset{\sim}{y} \end{aligned}$$

i Proof

Proof. Take the rows of D to be $D(x_i)^\top$. Lemma 3.2 gives the gradient as two sums, and Lemma 3.3 replaces the two sums with $D^\top D$ and $D^\top \underset{\sim}{y}$, which is the second line. Factoring $2D^\top$ out of the second line gives the first. $\square$

Example 3.6 (Checking the gradient formula against finite differences). Take the design matrix with an intercept and the inputs **x1** and **x2** of the synthetic data, and a coefficient vector $\vec{\beta} = (1, 1, 1)$ that is not the minimizer. Compute the gradient $2D^\top (D\vec{\beta} - \underset{\sim}{y})$ of Theorem 3.2, and compare it with finite differences of $J(\vec{\beta}) = \|D\vec{\beta} - \underset{\sim}{y}\|^2$:

3.2.1 Python

```

import numpy as np

syn_lsg = r.synthetic
Phi_lsg = np.column_stack([np.ones(len(syn_lsg)), syn_lsg["x1"], syn_lsg["x2"]])
y_lsg = syn_lsg["y"].to_numpy()
b_lsg = np.ones(3)

# The formula of the theorem.
grad_formula_lsg = 2 * Phi_lsg.T @ (Phi_lsg @ b_lsg - y_lsg)

# Central finite differences: nudge one coefficient at a time, in both directions.
def J_lsg(b):
    return np.sum((Phi_lsg @ b - y_lsg) ** 2)

eps_lsg = 1e-6
grad_fd_lsg = np.array([
    (J_lsg(b_lsg + eps_lsg * e) - J_lsg(b_lsg - eps_lsg * e)) / (2 * eps_lsg)
    for e in np.eye(3)
])
print(grad_formula_lsg)

[ -97.39305808 -149.19791176  178.94404189]
print(np.max(np.abs(grad_formula_lsg - grad_fd_lsg)))

4.25036432716297e-08

```

3.2.2 R

```

Phi_lsg <- cbind(1, synthetic$x1, synthetic$x2)
y_lsg <- synthetic$y
b_lsg <- rep(1, 3)

# The formula of the theorem.
grad_formula_lsg <- 2 * t(Phi_lsg) %*% (Phi_lsg %*% b_lsg - y_lsg)

# Central finite differences: nudge one coefficient at a time, in both directions.
J_lsg <- function(b) sum((Phi_lsg %*% b - y_lsg)^2)
eps_lsg <- 1e-6
grad_fd_lsg <- sapply(1:3, function(j) {
  e <- diag(3)[, j]
  (J_lsg(b_lsg + eps_lsg * e) - J_lsg(b_lsg - eps_lsg * e)) / (2 * eps_lsg)
})
c(formula = grad_formula_lsg)

  formula1  formula2  formula3
-97.39306 -149.19791  178.94404
max(abs(grad_formula_lsg - grad_fd_lsg))

[1] 2.7233e-08

```

The formula gives (-97.4, -149.2, 178.9). It differs from the finite-difference estimate by at most 2.7e-08, which is rounding error, so the matrix formula and the sum of single-point gradients agree.

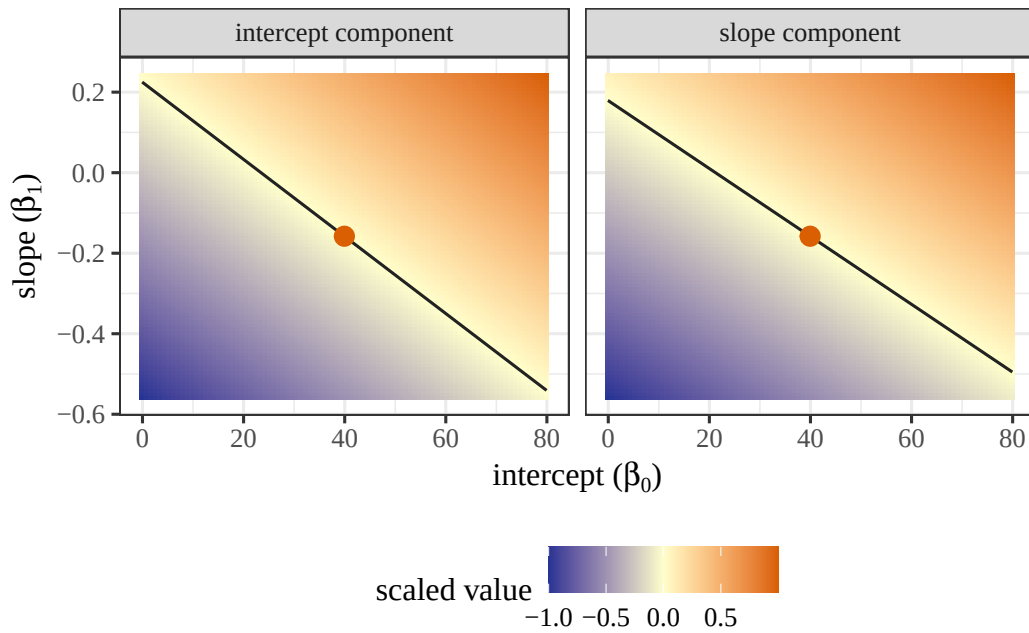


Figure 12: The gradient $\nabla_{\vec{\beta}} \hat{R}$ of the objective of Theorem 3.2, one surface for each of its two components, over the same intercepts and slopes as Figure 8. Each component is a plane, because the objective is quadratic: the gradient is a linear function of $\vec{\beta}$. Both planes cross zero along a line, and the two lines meet at the least-squares fit, where the gradient is zero: $\hat{R}$ is flat there, and nowhere else. On the course website, the blue points follow the choice made in Figure 8; in the handout, each component is scaled by its largest magnitude, and the black line is where it is zero.

Exercise 3.9 (Why two of the four terms are the same). Expanding the objective in matrix notation gives four terms:

$$(\underline{y} - D\vec{\beta})^\top (\underline{y} - D\vec{\beta}) = \underline{y}^\top \underline{y} - \underline{y}^\top D\vec{\beta} - \vec{\beta}^\top D^\top \underline{y} + \vec{\beta}^\top D^\top D\vec{\beta}$$

Give the shape of $\underline{y}^\top D\vec{\beta}$.

Solution

Solution 3.9. Multiply the shapes left to right:

$$\begin{array}{c} \underline{y}^\top \\ 1 \times n \end{array} \begin{array}{c} D \\ n \times p^* \end{array} \begin{array}{c} \vec{\beta} \\ p^* \times 1 \end{array} \longrightarrow 1 \times 1$$

It is a 1×1 matrix — a single number.

Exercise 3.10 (Why two of the four terms are the same (part 2)). Use the setup of Exercise 3.9. Show that $\underline{y}^\top D\vec{\beta} = \vec{\beta}^\top D^\top \underline{y}$, naming the property used at each step.

Solution

Solution 3.10. A 1×1 matrix has one entry, in position (1,1), and transposing sends that entry to position (1,1). So a 1×1 matrix equals its own transpose^a, and

$$\begin{aligned}
\tilde{y}^\top D \vec{\beta} &= (\tilde{y}^\top D \vec{\beta})^\top \\
&= \vec{\beta}^\top D^\top (\tilde{y}^\top)^\top \\
&= \vec{\beta}^\top D^\top \tilde{y}
\end{aligned}$$

Three properties in order: a 1×1 matrix is its own transpose; $(ABC)^\top = C^\top B^\top A^\top$, which reverses the order; and $(y^\top)^\top = y$.

So the two middle terms are the same number, and the expansion collapses to

$$\tilde{y} \cdot \tilde{y} - 2\vec{\beta}^\top D^\top \tilde{y} + \vec{\beta}^\top D^\top D \vec{\beta}$$

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-transpose>

Exercise 3.11 (Why two of the four terms are the same (part 3)). Use the setup of Exercise 3.9. Is $AB = B^\top A^\top$ true for matrices in general?

Solution

Solution 3.11. No. For general matrices AB and $B^\top A^\top$ need not even have the same shape: with A being 2×3 and B being 3×5 , AB is 2×5 while $B^\top A^\top$ is 5×2 . In general, $(AB)^\top = B^\top A^\top$, which is a statement about one product and its transpose^a, not about two products being equal.

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-transpose>

Exercise 3.12 (Why two of the four terms are the same (part 4)). Use the setup of Exercise 3.9. What does the argument in Exercise 3.10 actually rely on?

Solution

Solution 3.12. The argument in Solution 3.10 uses that general fact plus the fact that this particular product is 1×1 , so that its transpose^a is itself. Drop the second and nothing goes through. The step is about a scalar written in matrix notation, so it is available exactly where a product collapses to 1×1 and nowhere else.

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-transpose>

3.3 The gradient by the vector chain rule

Exercise 3.13 (The gradient of the mean squared error by the vector chain rule). For coefficients $\vec{\beta}$, the model predicts $\hat{y}_i = D(x_i) \cdot \vec{\beta}$ for observation i . Its prediction error^a is the prediction minus the observed outcome, $\varepsilon_{\text{pred}_i} = \hat{y}_i - y_i$, the negative of its residual^b. The least squares objective is the average of the squared prediction errors, which is the sample mean squared error (sMSE) of the coefficients $\vec{\beta}$ (Definition 6.1, defined later in this chapter):

$$\text{sMSE}(\vec{\beta}) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^n \varepsilon_{\text{pred}_i}^2.$$

Collect the errors in the $n \times 1$ vector $\varepsilon_{\text{pred}} = D\vec{\beta} - y$, whose entry i is $\varepsilon_{\text{pred}_i}$.

Write $\text{sMSE}(\vec{\beta})$ as a dot product^c of $\varepsilon_{\text{pred}}$ with itself.

^a<https://morrisson-lab.github.io/sds/prediction.html#def-prediction-error>

^b<https://morrisson-lab.github.io/sds/prediction.html#thm-prediction-error-residual>

^c<https://morrisson-lab.github.io/mds/linear-algebra-vectors.html#def-dot-product>

Solution

Solution 3.13. Colors track where each part's result is used later: violet for Solution 3.13, blue for the inner derivative (Solution 3.15), red for the outer derivative (Solution 3.16), and teal for the gradient (Solution 3.17).

Entry i of $\varepsilon_{\text{pred}}$ is $\varepsilon_{\text{pred}_i}$, and a dot product sums the products of matching entries, so

$$\begin{aligned} \text{sMSE}(\vec{\beta}) &= \frac{1}{n} \sum_{i=1}^n \varepsilon_{\text{pred}_i}^2 && \text{(definition of sMSE)} \\ &= \frac{1}{n} \sum_{i=1}^n \varepsilon_{\text{pred}_i} \varepsilon_{\text{pred}_i} && \text{(square as a product)} \\ &= \frac{1}{n} \varepsilon_{\text{pred}} \cdot \varepsilon_{\text{pred}} && \text{(definition of the dot product)} \end{aligned}$$

Exercise 3.14 (The gradient of the mean squared error by the vector chain rule (part 2)). Use the setup of Exercise 3.13.

Use the vector chain rule^a to write the gradient $\frac{\partial}{\partial \vec{\beta}} \text{sMSE}(\vec{\beta})$ as the product of two derivatives: the inner derivative $\frac{\partial}{\partial \vec{\beta}} \varepsilon_{\text{pred}}$ and the outer derivative of the sMSE with respect to $\varepsilon_{\text{pred}}$.

^a<https://morrisson-lab.github.io/mds/vector-calculus.html#thm-chain-vec>

Solution

Solution 3.14. Solution 3.13 writes $\text{sMSE}(\vec{\beta})$ as a scalar function of the vector $\varepsilon_{\text{pred}}$, which is itself a vector-valued function of $\vec{\beta}$. The vector chain rule^a multiplies the two derivatives, inner function first:

$$\begin{aligned} \frac{\partial}{\partial \vec{\beta}} \text{sMSE}(\vec{\beta}) &= \frac{\partial}{\partial \vec{\beta}} \left(\frac{1}{n} \varepsilon_{\text{pred}} \cdot \varepsilon_{\text{pred}} \right) && \text{(@sol-mse-gradient-chain-rule)} \\ &= \underbrace{\left(\frac{\partial}{\partial \vec{\beta}} \varepsilon_{\text{pred}} \right)}_{p \times n} \underbrace{\left(\frac{\partial}{\partial \varepsilon_{\text{pred}}} \left(\frac{1}{n} \varepsilon_{\text{pred}} \cdot \varepsilon_{\text{pred}} \right) \right)}_{n \times 1} && \text{(vector chain rule)} \end{aligned}$$

To go further, we need the two factors. Parts 3 and 4 find them, one at a time, and Solution 3.17 substitutes them back in.

^a<https://morrisson-lab.github.io/mds/vector-calculus.html#thm-chain-vec>

Exercise 3.15 (The gradient of the mean squared error by the vector chain rule (part 3)). Use the setup of Exercise 3.13.

Find the inner derivative $\frac{\partial}{\partial \vec{\beta}} \varepsilon_{\text{pred}}$, and give its shape.

Solution

Solution 3.15. The vector y does not depend on $\vec{\beta}$, so it is constant^a and its derivative is a zero matrix. The derivative of $\vec{\beta}$ with respect to itself is the $p^* \times p^*$ identity matrix $\mathbf{I}$, because entry (i, j) is $\frac{\partial}{\partial \beta_i} \beta_j$, which is 1 when $i = j$ and 0 otherwise. Using the derivative of a matrix-vector product^b, the inner derivative is

$$\begin{aligned}
 \frac{\partial}{\partial \vec{\beta}} \varepsilon_{\text{pred}} &= \frac{\partial}{\partial \vec{\beta}} (D\vec{\beta} - y) && \text{(definition of the error vector)} \\
 &= \frac{\partial}{\partial \vec{\beta}} (D\vec{\beta}) - \frac{\partial}{\partial \vec{\beta}} y && \text{(derivative of a difference)} \\
 &= \left(\frac{\partial}{\partial \vec{\beta}} \vec{\beta} \right) D^\top - \frac{\partial}{\partial \vec{\beta}} y && \text{(derivative of a matrix-vector product)} \\
 &= \left(\frac{\partial}{\partial \vec{\beta}} \vec{\beta} \right) D^\top - \mathbf{0} && (y \text{ is constant}) \\
 &= \mathbf{I} D^\top - \mathbf{0} && \text{(derivative of } \vec{\beta} \text{ is } \mathbf{I}) \\
 &= \mathbf{I} D^\top && \text{(subtract a zero matrix)} \\
 &= D^\top && \text{(multiply by the identity)}
 \end{aligned}$$

Its shape is $p^* \times n$: one row for each coefficient and one column for each observation's error, the denominator layout^c.

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-constant-wrt-vector>

^b<https://morrison-lab.github.io/mds/vector-calculus.html#thm-deriv-matrix-vector>

^c<https://morrison-lab.github.io/mds/vector-calculus.html#def-vector-valued-derivative>

Exercise 3.16 (The gradient of the mean squared error by the vector chain rule (part 4)). Use the setup of Exercise 3.13.

Find the outer derivative.

Solution

Solution 3.16. The scalar $\frac{1}{n}$ does not depend on $\varepsilon_{\text{pred}}$, so it factors out of the derivative. The derivative of a dot product of a vector with itself^a is twice that vector. So the outer derivative is

$$\begin{aligned}
 \frac{\partial}{\partial \varepsilon_{\text{pred}}} \left(\frac{1}{n} \varepsilon_{\text{pred}} \cdot \varepsilon_{\text{pred}} \right) &= \frac{1}{n} \frac{\partial}{\partial \varepsilon_{\text{pred}}} \varepsilon_{\text{pred}} \cdot \varepsilon_{\text{pred}} && \text{(factor out a constant)} \\
 &= \frac{1}{n} (2\varepsilon_{\text{pred}}) && \text{(derivative of a dot product with itself)} \\
 &= \frac{2}{n} \varepsilon_{\text{pred}} && \text{(multiply the scalars)}
 \end{aligned}$$

Its shape is $n \times 1$, the same as $\varepsilon_{\text{pred}}$.

^a<https://morrison-lab.github.io/mds/vector-calculus.html#cor-deriv-normsq>

Exercise 3.17 (The gradient of the mean squared error by the vector chain rule (part 5)). Use the setup of Exercise 3.13.

Substitute Exercise 3.15 and Exercise 3.16 into Exercise 3.14 to find the gradient, without expanding the dot product^a.

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-dot-product>

Solution

Solution 3.17. Return to Solution 3.14, and substitute the two factors:

$$\begin{aligned}
 \frac{\partial}{\partial \vec{\beta}} \text{sMSE}(\vec{\beta}) &= \left(\frac{\partial}{\partial \vec{\beta}} \varepsilon_{\text{pred}} \right) \left(\frac{\partial}{\partial \varepsilon_{\text{pred}}} \left(\frac{1}{n} \varepsilon_{\text{pred}} \cdot \varepsilon_{\text{pred}} \right) \right) && (\text{@sol-mse-gradient-chain-rule-part2}) \\
 &= D^\top \left(\frac{\partial}{\partial \varepsilon_{\text{pred}}} \left(\frac{1}{n} \varepsilon_{\text{pred}} \cdot \varepsilon_{\text{pred}} \right) \right) && (\text{@sol-mse-gradient-chain-rule-part3}) \\
 &= D^\top \left(\frac{2}{n} \varepsilon_{\text{pred}} \right) && (\text{@sol-mse-gradient-chain-rule-part4}) \\
 &= \frac{2}{n} D^\top \varepsilon_{\text{pred}} && (\text{move the scalar to the front})
 \end{aligned}$$

The product of a $p^* \times n$ matrix and an $n \times 1$ vector is $p^* \times 1$, one entry for each coefficient, as a gradient must be. The chain rule never needed the four-term expansion of Exercise 3.9, because it treats the error vector as one object until the last step.

Exercise 3.18 (The gradient of the mean squared error by the vector chain rule (part 6)). Use the setup of Exercise 3.13.

Check that your answer is $\frac{1}{n}$ times the gradient in Theorem 3.2.

Solution

Solution 3.18. Substitute $\varepsilon_{\text{pred}} = D\vec{\beta} - \vec{y}$, and compare with the gradient of J in Theorem 3.2:

$$\begin{aligned}
 \frac{\partial}{\partial \vec{\beta}} \text{sMSE}(\vec{\beta}) &= \frac{2}{n} D^\top \varepsilon_{\text{pred}} && (\text{@sol-mse-gradient-chain-rule-part5}) \\
 &= \frac{2}{n} D^\top (D\vec{\beta} - \vec{y}) && (\text{definition of the error vector}) \\
 &= \frac{1}{n} (2D^\top (D\vec{\beta} - \vec{y})) && (\text{factor out } \frac{1}{n}) \\
 &= \frac{1}{n} \nabla_{\vec{\beta}} J(\vec{\beta}) && (\text{gradient of } J)
 \end{aligned}$$

So the sMSE and J have the same stationary points^a, and setting this gradient to $\vec{0}$ gives the same normal equations.

^a<https://morrison-lab.github.io/mds/optimization.html#def-stationary-point>

Exercise 3.19 (The gradient of the mean squared error by the vector chain rule (part 7)). Use the setup of Exercise 3.13.

Find the Hessian^a of $\text{sMSE}(\vec{\beta})$.

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-hessian>

Solution

Solution 3.19. The Hessian^a is the derivative of the gradient. The gradient from Solution 3.17 is a constant^b matrix times the error vector, so the derivative of a matrix-vector product^c applies again,

with $\mathbf{A} = \frac{2}{n} D^\top$:

$$\begin{aligned}
\mathbf{H} &= \frac{\partial}{\partial \vec{\beta}} \left(\frac{\partial}{\partial \vec{\beta}} \text{sMSE}(\vec{\beta}) \right) && \text{(definition of the Hessian)} \\
&= \frac{\partial}{\partial \vec{\beta}} \left(\frac{2}{n} D^\top \varepsilon_{\text{pred}} \right) && \text{(@sol-mse-gradient-chain-rule-part5)} \\
&= \left(\frac{\partial}{\partial \vec{\beta}} \varepsilon_{\text{pred}} \right) \left(\frac{2}{n} D^\top \right)^\top && \text{(derivative of a matrix-vector product)} \\
&= D^\top \left(\frac{2}{n} D^\top \right)^\top && \text{(@sol-mse-gradient-chain-rule-part3)} \\
&= D^\top \left(\frac{2}{n} D \right) && \text{(transpose of a transpose)} \\
&= \frac{2}{n} D^\top D && \text{(move the scalar to the front)}
\end{aligned}$$

It is $p^* \times p^*$ and does not depend on $\vec{\beta}$.

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-hessian>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^c<https://morrison-lab.github.io/mds/vector-calculus.html#thm-deriv-matrix-vector>

Exercise 3.20 (The gradient of the mean squared error by the vector chain rule (part 8)). Use the setup of Exercise 3.13.

Show that it is positive definite^a when the columns of D are linearly independent^b.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-positive-definite>

^b<https://morrison-lab.github.io/pds/independence.html#def-indpt>

Solution

Solution 3.20. To show it is positive definite^a, take any $p^* \times 1$ vector $\vec{v} \neq \vec{0}$:

$$\begin{aligned}
\vec{v} \cdot (\mathbf{H}\vec{v}) &= \vec{v} \cdot \left(\frac{2}{n} D^\top D \vec{v} \right) && \text{(value of } \mathbf{H} \text{)} \\
&= \frac{2}{n} \vec{v} \cdot (D^\top D \vec{v}) && \text{(move the scalar out of the dot product)} \\
&= \frac{2}{n} (D\vec{v}) \cdot (D\vec{v}) && \text{(move } D^\top \text{ across the dot product)} \\
&= \frac{2}{n} \|D\vec{v}\|^2 && \text{(definition of the squared norm)}
\end{aligned}$$

The third step uses $\vec{v} \cdot (D^\top \vec{u}) = (D\vec{v}) \cdot \vec{u}$, which holds because both sides equal $\sum_{i=1}^n \sum_j u_i D_{ij} v_j$. A squared norm^b is never negative, and it is zero only for the zero vector. When the columns of D are linearly independent^c, $D\vec{v}$ is a combination of them with coefficients $\vec{v} \neq \vec{0}$, so $D\vec{v} \neq \vec{0}$. Then $\vec{v} \cdot (\mathbf{H}\vec{v}) > 0$ for every $\vec{v} \neq \vec{0}$, and $\mathbf{H}$ is positive definite.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-positive-definite>

^b<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

^c<https://morrison-lab.github.io/pds/independence.html#def-indpt>

Exercise 3.21 (The gradient of the mean squared error by the vector chain rule (part 9)). Use the setup of Exercise 3.13.

What does that tell you about the stationary point^a?

^a<https://morrison-lab.github.io/mds/optimization.html#def-stationary-point>

Solution

Solution 3.21. By the second-derivative test for convexity^a, a Hessian^b that is positive definite^c at every $\vec{\beta}$ makes $\text{sMSE}(\vec{\beta})$ strictly convex^d. So the stationary point^e from Solution 3.18 is the unique global minimizer^f. If the columns of D are not linearly independent^g, some $\vec{v} \neq \vec{0}$ has $D\vec{v} = \vec{0}$, so $\mathbf{H}$ is only positive semidefinite and the minimizer is not unique.

^a<https://morrison-lab.github.io/mds/optimization.html#thm-convex-hessian>

^b<https://morrison-lab.github.io/mds/vector-calculus.html#def-hessian>

^c<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-positive-definite>

^d<https://morrison-lab.github.io/mds/algebra-sums.html#def-convex-function>

^e<https://morrison-lab.github.io/mds/optimization.html#def-stationary-point>

^f<https://morrison-lab.github.io/mds/algebra-basics.html#def-global-minimizer>

^g<https://morrison-lab.github.io/pds/independence.html#def-indpt>

3.4 The normal equations

Theorem 3.3 (The normal equations). *The ordinary least squares estimator^a $\hat{\vec{\beta}}$ satisfies the linear system of normal equations^b:*

$$(D^\top D)\vec{\beta} = D^\top y \quad (2)$$

When the matrix $D^\top D \in \mathbb{R}^{p^* \times p^*}$ is invertible^c, there is a unique global minimizer^d, and it has the closed-form solution^e:

$$\hat{\vec{\beta}} = (D^\top D)^{-1} D^\top y \quad (3)$$

^a<https://morrison-lab.github.io/sds/correlation-regression.html>

^b<https://morrison-lab.github.io/sds/correlation-regression.html>

^c<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-matrix-inverse>

^d<https://morrison-lab.github.io/mds/algebra-basics.html#def-global-minimizer>

^e<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-closed-form-solution>

i Proof

Proof. Since $J(\vec{\beta}) = \|D\vec{\beta} - y\|^2$ is a smooth convex function on $\mathbb{R}^{p^*}$, any global minimizer $\hat{\vec{\beta}}$ must satisfy $\nabla_{\vec{\beta}} J(\hat{\vec{\beta}}) = \vec{0}$ by Theorem 3.1. By Theorem 3.2, setting

$$\begin{aligned} \nabla_{\vec{\beta}} J &= 2D^\top D\vec{\beta} - 2D^\top y \\ &= \vec{0} \end{aligned}$$

yields $(D^\top D)\vec{\beta} = D^\top y$. Multiplying both sides on the left by $(D^\top D)^{-1}$ gives $\hat{\vec{\beta}} = (D^\top D)^{-1} D^\top y$. $\square$

Example 3.7 (Solving the normal equations for fuel efficiency). In the `Auto` data [3], regress `mpg` on `horsepower` and `weight`. Form $D^\top D$ and $D^\top y$, solve the normal equations of Theorem 3.3, and compare with the coefficients from a library fit:

3.4.1 Python

```
import numpy as np
import pandas as pd

auto_ne = pd.read_csv("data/auto.csv")
y_ne = auto_ne["mpg"].to_numpy(float)
Phi_ne = np.column_stack([np.ones(len(y_ne)), auto_ne["horsepower"], auto_ne["weight"]])

# Solve (Phi'Phi) beta = Phi'y directly; solve() avoids forming the inverse explicitly.
beta_ne = np.linalg.solve(Phi_ne.T @ Phi_ne, Phi_ne.T @ y_ne)
beta_ne_lib = np.linalg.lstsq(Phi_ne, y_ne, rcond=None)[0]
print(beta_ne)

[ 4.56402108e+01 -4.73028631e-02 -5.79415736e-03]

print(np.allclose(beta_ne, beta_ne_lib))

True
```

3.4.2 R

```
auto_ne <- read.csv("data/auto.csv")
Phi_ne <- cbind(1, auto_ne$horsepower, auto_ne$weight)

# Solve (Phi'Phi) beta = Phi'y directly; solve() with two arguments avoids forming the inverse.
beta_ne <- solve(crossprod(Phi_ne), crossprod(Phi_ne, auto_ne$mpg))
beta_ne_lib <- coef(lm(mpg ~ horsepower + weight, data = auto_ne))
cbind(normal_equations = drop(beta_ne), lm = beta_ne_lib)
```

	normal_equations	lm
(Intercept)	45.640210840	45.640210840
horsepower	-0.047302863	-0.047302863
weight	-0.005794157	-0.005794157

The two routes agree. The intercept is 45.64, each extra horsepower changes the prediction by -0.0473 miles per gallon, and each extra pound by -0.00579. Library routines use a more numerically stable factorization than forming $D^\top D$, but they solve the same problem.

i Remark (Invertibility, sample size, and collinearity)

Remark 3.1 (Invertibility, sample size, and collinearity). For $(D^\top D)^{-1}$ to exist, the design matrix D must have full column rank^a p^* . The matrix $D^\top D$ is the Gram matrix^b of the columns of D : its entry (j, k) is the dot product of columns j and k , and it is symmetric and positive semidefinite. Two distinct conditions can cause invertibility to fail:

1. **Underdetermined system** ($n < p^*$ or $n \leq p$): When there are fewer observations than parameters, the columns of D cannot be linearly independent. If D has full row rank, there exist infinitely many parameter vectors $\vec{\beta}$ that interpolate the training data with zero residual error.
2. **Collinearity among features**: If any feature is an exact linear combination^c of others (or if a feature is constant alongside the 1-padding), the matrix D is rank-deficient and $D^\top D$ is singular regardless of how large n is. Statisticians call this perfect multicollinearity^d.

^a<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

^b<https://morrison-lab.github.io/mds/linear-algebra-rank-nullity.html#def-gram-matrix>

^c<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-linear-combination>

^d<https://morrison-lab.github.io/sds/correlation-regression.html>

Exercise 3.22 (Solve the normal equations by hand). Three training points, one feature each:

$$(x_1, y_1) = (0, 1), \quad (x_2, y_2) = (1, 3), \quad (x_3, y_3) = (2, 3)$$

Fit $h(x) = \beta_0 + \beta_1 x$ by least squares, working in exact fractions throughout. Write down D and $\tilde{y}$, using the design vector of Exercise 1.1.

Solution

Solution 3.22. The design vector for each input is $D(x_i) = (1, x_i)^\top$, so

$$D = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix}, \quad \tilde{y} = \begin{bmatrix} 1 \\ 3 \\ 3 \end{bmatrix}$$

Exercise 3.23 (Solve the normal equations by hand (part 2)). Use the setup of Exercise 3.22. Give the shape of each.

Solution

Solution 3.23. D is 3×2 and $\tilde{y}$ is 3×1 . Here $n = 3$ and $p = 1$, so D has $p^* = p + 1 = 2$ columns.

Exercise 3.24 (Solve the normal equations by hand (part 3)). Use the setup of Exercise 3.22. Compute $D^\top D$ and $D^\top \tilde{y}$.

Solution

Solution 3.24. Both products are sized by p^* alone, as Exercise 2.1 worked out: $D^\top D$ is 2×2 and $D^\top \tilde{y}$ is 2×1 , whatever n is.

$$D^\top D = \begin{bmatrix} 3 & 3 \\ 3 & 5 \end{bmatrix}, \quad D^\top \tilde{y} = \begin{bmatrix} 7 \\ 9 \end{bmatrix}$$

The top-left entry is n , because the first column of D is all ones.

Exercise 3.25 (Solve the normal equations by hand (part 4)). Use the setup of Exercise 3.22. Solve $(D^\top D)\vec{\beta} = D^\top \tilde{y}$ for $\vec{\beta}$.

Solution

Solution 3.25. $D^\top D$ is 2×2 , so invert it by the formula for a 2×2 matrix. Its determinant is $3 \cdot 5 - 3 \cdot 3 = 6$, which is non-zero, so the inverse^a exists:

$$(D^\top D)^{-1} = \frac{1}{6} \begin{bmatrix} 5 & -3 \\ -3 & 3 \end{bmatrix}$$

$$\begin{aligned} \vec{\beta} &= (D^\top D)^{-1} D^\top \tilde{y} \\ &= \frac{1}{6} \begin{bmatrix} 5 & -3 \\ -3 & 3 \end{bmatrix} \begin{bmatrix} 7 \\ 9 \end{bmatrix} \\ &= \frac{1}{6} \begin{bmatrix} 35 - 27 \\ -21 + 27 \end{bmatrix} \\ &= \begin{bmatrix} 4/3 \\ 1 \end{bmatrix} \end{aligned}$$

So the fitted line is $h(x) = \frac{4}{3} + x$.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-matrix-inverse>

Exercise 3.26 (Solve the normal equations by hand (part 5)). Use the setup of Exercise 3.22. Compute the fitted values and the residuals.

Solution

Solution 3.26. The fitted values are $D\vec{\beta}$, and the residuals are $\tilde{y} - D\vec{\beta}$ — observed minus fitted.

$$D\vec{\beta} = \begin{bmatrix} 4/3 \\ 7/3 \\ 10/3 \end{bmatrix}, \quad \tilde{y} - D\vec{\beta} = \begin{bmatrix} -1/3 \\ 2/3 \\ -1/3 \end{bmatrix}$$

The residual sum of squares is

$$\begin{aligned} (-1/3)^2 + (2/3)^2 + (-1/3)^2 &= \frac{1}{9} + \frac{4}{9} + \frac{1}{9} \\ &= \frac{2}{3}. \end{aligned}$$

Exercise 3.27 (Solve the normal equations by hand (part 6)). Use the setup of Exercise 3.22. No residual is zero. Should you expect any to be?

Solution

Solution 3.27. No, you should not expect any residual to be zero. Three points determine a line only when they are collinear, and these are not: the steps in y are +2 then 0 while the steps in x are +1 then +1. Two free parameters cannot satisfy three independent^a equations, so the best a line can do is miss all three by a little. The residuals sum to zero rather than vanishing, which is a different statement and is the first half of Solution 3.28.

^a<https://morrison-lab.github.io/pds/independence.html#def-indpt>

Exercise 3.28 (Solve the normal equations by hand (part 7)). Use the setup of Exercise 3.22. Verify that $D^\top(\tilde{y} - D\vec{\beta}) = \vec{0}$.

Solution

Solution 3.28. Multiply out, column by column:

$$\begin{aligned} D^\top(\tilde{y} - D\vec{\beta}) &= \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} -1/3 \\ 2/3 \\ -1/3 \end{bmatrix} \\ &= \begin{bmatrix} -1/3 + 2/3 - 1/3 \\ 0 + 2/3 - 2/3 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned}$$

Exercise 3.29 (Solve the normal equations by hand (part 8)). Use the setup of Exercise 3.22. Say what that equation says geometrically.

Solution

Solution 3.29. Each entry of $D^\top(y - D\tilde{\beta})$ is the inner product of one column of D with the residual vector. So the equation says the residual is orthogonal^a to every column of D , and therefore to every combination of those columns — the set of all vectors $D\tilde{\beta}$ the model can reach. The fitted values are the point of that space closest to y , and the residual is the remainder, perpendicular to it.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-orthogonal-matrix>

Exercise 3.30 (Solve the normal equations by hand (part 9)). Use the setup of Exercise 3.22. Say why it had to hold, given how $\tilde{\beta}$ was obtained.

Solution

Solution 3.30. It had to hold because it *is* the normal equations. Setting the gradient $-2D^\top y + 2D^\top D\tilde{\beta}$ to zero and dividing by 2 gives $D^\top(y - D\tilde{\beta}) = \vec{0}$ directly. So this is not an extra check on the answer; it is the equation that was solved, rearranged. It does check the arithmetic of the earlier steps.

The first column being all ones is why the residuals sum to zero, and it is also why the fitted line passes through the sample mean^a of the data: $\bar{x} = 1$ and $\bar{y} = 7/3$, and

$$\begin{aligned} h(1) &= \frac{4}{3} + 1 \\ &= \frac{7}{3}. \end{aligned}$$

Drop the intercept^b and both facts go with it.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

Lemma 3.4 (Shortcut formulas for the centered sums). *For data $(x_1, y_1), \dots, (x_n, y_n)$ with sample means $\bar{x}$ and $\bar{y}$, let $S_{xx} = \sum_{i=1}^n (x_i - \bar{x})^2$ and $S_{xy} = \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$. Then*

$$S_{xx} = \sum_{i=1}^n x_i^2 - n\bar{x}^2, \quad S_{xy} = \sum_{i=1}^n x_i y_i - n\bar{x}\bar{y}. \quad (4)$$

i Proof

Proof. Expand the products, then use $\sum_{i=1}^n x_i = n\bar{x}$ and $\sum_{i=1}^n y_i = n\bar{y}$:

$$\begin{aligned} S_{xx} &= \sum_{i=1}^n x_i^2 - 2\bar{x} \sum_{i=1}^n x_i + n\bar{x}^2 \\ &= \sum_{i=1}^n x_i^2 - 2n\bar{x}^2 + n\bar{x}^2 \\ &= \sum_{i=1}^n x_i^2 - n\bar{x}^2, \\ S_{xy} &= \sum_{i=1}^n x_i y_i - \bar{y} \sum_{i=1}^n x_i - \bar{x} \sum_{i=1}^n y_i + n\bar{x}\bar{y} \\ &= \sum_{i=1}^n x_i y_i - n\bar{x}\bar{y} - n\bar{x}\bar{y} + n\bar{x}\bar{y} \\ &= \sum_{i=1}^n x_i y_i - n\bar{x}\bar{y}. \end{aligned}$$

□

Example 3.8 (Both forms of the centered sums, for three points). Take the three points (0, 1), (1, 3), (2, 3) of Exercise 3.22. Compute S_{xx} and S_{xy} from the definitions and from Equation 4:

```
x_cs <- c(0, 1, 2)
y_cs <- c(1, 3, 3)
n_cs <- length(x_cs)

# Definition: sums of centered products.
sxx_def <- sum((x_cs - mean(x_cs))^2)
sxy_def <- sum((x_cs - mean(x_cs)) * (y_cs - mean(y_cs)))

# Shortcut: raw sums minus n times the product of the means; no centering needed.
sxx_short <- sum(x_cs^2) - n_cs * mean(x_cs)^2
sxy_short <- sum(x_cs * y_cs) - n_cs * mean(x_cs) * mean(y_cs)
c(sxx_def, sxx_short, sxy_def, sxy_short)

[1] 2 2 2 2
Both forms give  $S_{xx} = 2$  and  $S_{xy} = 2$ .
```

Theorem 3.4 (The simple linear regression slope is the sample covariance over the sample variance). Let $(x_1, y_1), \dots, (x_n, y_n)$ be data with $n \geq 2$ whose inputs are not all equal. Write $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$ and $\bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$, and let $\widehat{\text{Cov}}(x, y) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$ and $\widehat{\text{Var}}(x) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$ be the sample covariance and the sample variance. The least squares fit $h(x) = \hat{\beta}_0 + \hat{\beta}_1 x$ has

$$\hat{\beta}_1 = \frac{\widehat{\text{Cov}}(x, y)}{\widehat{\text{Var}}(x)}, \quad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}. \quad (5)$$

i Proof

Proof. Let $S_{xx} = \sum_{i=1}^n (x_i - \bar{x})^2$ and $S_{xy} = \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$, as in Lemma 3.4. Then $S_{xx} > 0$ because the x_i are not all equal.

1. The normal equations. The design matrix D has a column of ones and a column of the x_i , so

$$D^\top D = \begin{bmatrix} n & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 \end{bmatrix}, \quad D^\top y = \begin{bmatrix} \sum_{i=1}^n y_i \\ \sum_{i=1}^n x_i y_i \end{bmatrix}.$$

Its determinant is $n \sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i \right)^2$. Using $\sum_{i=1}^n x_i = n\bar{x}$ and then Lemma 3.4,

$$\begin{aligned} n \sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i \right)^2 &= n \left(\sum_{i=1}^n x_i^2 - n\bar{x}^2 \right) \\ &= nS_{xx}, \end{aligned}$$

which is non-zero. So by Theorem 3.3 the minimizer is unique and solves the two normal equations:

$$\begin{aligned} n\beta_0 + \left(\sum_{i=1}^n x_i \right) \beta_1 &= \sum_{i=1}^n y_i, \\ \left(\sum_{i=1}^n x_i \right) \beta_0 + \left(\sum_{i=1}^n x_i^2 \right) \beta_1 &= \sum_{i=1}^n x_i y_i. \end{aligned}$$

2. The intercept. Divide the first equation by n , using $\sum_{i=1}^n x_i = n\bar{x}$ and $\sum_{i=1}^n y_i = n\bar{y}$:

$$\begin{aligned} \beta_0 + \bar{x}\beta_1 &= \bar{y}, \\ \beta_0 &= \bar{y} - \beta_1 \bar{x}. \end{aligned}$$

3. The slope. Substitute $\beta_0 = \bar{y} - \beta_1 \bar{x}$ into the second equation and use $\sum_{i=1}^n x_i = n\bar{x}$:

$$\begin{aligned} n\bar{x}(\bar{y} - \beta_1 \bar{x}) + \left(\sum_{i=1}^n x_i^2 \right) \beta_1 &= \sum_{i=1}^n x_i y_i, \\ \left(\sum_{i=1}^n x_i^2 - n\bar{x}^2 \right) \beta_1 &= \sum_{i=1}^n x_i y_i - n\bar{x}\bar{y}. \end{aligned}$$

By Lemma 3.4, this says $S_{xx}\beta_1 = S_{xy}$. Since $S_{xx} > 0$,

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}} = \frac{S_{xy}/(n-1)}{S_{xx}/(n-1)} = \frac{\widehat{\text{Cov}}(x, y)}{\widehat{\text{Var}}(x)}.$$

The factor $n - 1$ cancels, so dividing by n instead (the population versions) gives the same slope. The intercept is step 2 with $\beta_1 = \hat{\beta}_1$. $\square$

Example 3.9 (The slope of fuel efficiency on weight). In the Auto data [3], regress `mpg` on `weight`. Compute the slope as the sample covariance over the sample variance (Equation 5), and compare it with the coefficient from a library fit:

3.4.3 Python

```
import numpy as np
import pandas as pd

auto_cv = pd.read_csv("data/auto.csv")
x_cv = auto_cv["weight"].to_numpy(float)
y_cv = auto_cv["mpg"].to_numpy(float)

# np.cov already divides by n - 1; np.var needs ddof=1 to match it.
# The ratio is the slope either way, but matching the divisors keeps each piece a sample moment.
slope_cv = np.cov(x_cv, y_cv)[0, 1] / np.var(x_cv, ddof=1)
intercept_cv = y_cv.mean() - slope_cv * x_cv.mean()

# Compare with a library least squares fit of degree 1, which returns (slope, intercept).
fit_cv = np.polyfit(x_cv, y_cv, 1)
print(slope_cv, intercept_cv)

-0.007647342535779581 46.21652454901758
print(np.allclose([slope_cv, intercept_cv], fit_cv))

True
```

3.4.4 R

```
auto_cv <- read.csv("data/auto.csv")

# cov() and var() both divide by n - 1, so their ratio is the slope.
slope_cv <- cov(auto_cv$weight, auto_cv$mpg) / var(auto_cv$weight)
intercept_cv <- mean(auto_cv$mpg) - slope_cv * mean(auto_cv$weight)

# Compare with lm(), which solves the least squares problem by a QR factorization.
fit_cv <- coef(lm(mpg ~ weight, data = auto_cv))
cbind(cov_over_var = c(intercept_cv, slope_cv), lm = fit_cv)

      cov_over_var      lm
```

(Intercept) 46.216524549 46.216524549
 weight -0.007647343 -0.007647343

The two routes agree: the slope is -0.00765 miles per gallon for each extra pound, and the intercept is 46.22. The slope is negative because the covariance of weight and fuel efficiency is -5517.4, while the variance of weight, 721485, is always positive.

The three points of Exercise 3.22 give a check by hand. Their sample means are $\bar{x} = 1$ and $\bar{y} = 7/3$, so $S_{xy} = (-1)(-4/3) + 0 + (1)(2/3) = 2$ and $S_{xx} = 1 + 0 + 1 = 2$. The slope is $S_{xy}/S_{xx} = 1$, the value that exercise finds by solving the normal equations.

For the same slope written with the correlation, see the OLS slope and the correlation^a.

^a<https://morrison-lab.github.io/sds/correlation-regression.html#cor-ols-slope-r>

Corollary 3.1 (The two simple regression slopes multiply to the squared correlation). *Let $(x_1, y_1), \dots, (x_n, y_n)$ be data whose x_i are not all equal and whose y_i are not all equal. Let $\hat{\beta}_{y|x}$ be the slope from regressing y on x , let $\hat{\beta}_{x|y}$ be the slope from regressing x on y , and let $r = \widehat{\text{Cov}}(x, y) / \sqrt{\widehat{\text{Var}}(x) \widehat{\text{Var}}(y)}$ be the sample correlation. Then*

$$\hat{\beta}_{y|x} \hat{\beta}_{x|y} = r^2, \quad \text{so} \quad \sqrt{\hat{\beta}_{y|x} \hat{\beta}_{x|y}} = |r|. \quad (6)$$

Both slopes have the sign of $\widehat{\text{Cov}}(x, y)$, so r has that same sign. The square root of the product is the correlation, not the covariance: the covariance is $r\sqrt{\widehat{\text{Var}}(x) \widehat{\text{Var}}(y)}$, and it has the units of x times the units of y .

i Proof

Proof. By Theorem 3.4 applied to each regression,

$$\hat{\beta}_{y|x} = \frac{\widehat{\text{Cov}}(x, y)}{\widehat{\text{Var}}(x)}, \quad \hat{\beta}_{x|y} = \frac{\widehat{\text{Cov}}(y, x)}{\widehat{\text{Var}}(y)} = \frac{\widehat{\text{Cov}}(x, y)}{\widehat{\text{Var}}(y)},$$

because the covariance is symmetric in its two arguments. Multiply:

$$\begin{aligned} \hat{\beta}_{y|x} \hat{\beta}_{x|y} &= \frac{\widehat{\text{Cov}}(x, y)^2}{\widehat{\text{Var}}(x) \widehat{\text{Var}}(y)} \\ &= \left(\frac{\widehat{\text{Cov}}(x, y)}{\sqrt{\widehat{\text{Var}}(x) \widehat{\text{Var}}(y)}} \right)^2 \\ &= r^2. \end{aligned}$$

Taking square roots gives $|r|$. The variances are positive, so each slope has the sign of $\widehat{\text{Cov}}(x, y)$, which is the sign of r . $\square$

Example 3.10 (The two slopes for fuel efficiency and weight). Regress **weight** on **mpg** as well as **mpg** on **weight**, and compare the product of the slopes with the squared correlation:

```

auto_sc <- read.csv("data/auto.csv")

# Slope of mpg on weight, and the reverse slope of weight on mpg, each as covariance over variance.
slope_y_on_x <- cov(auto_sc$weight, auto_sc$mpg) / var(auto_sc$weight)
slope_x_on_y <- cov(auto_sc$weight, auto_sc$mpg) / var(auto_sc$mpg)

# The square root of their product should equal the absolute correlation;
# the common sign of the slopes should equal the sign of the correlation.
r_sc <- cor(auto_sc$weight, auto_sc$mpg)
c(
  product = slope_y_on_x * slope_x_on_y, r_squared = r_sc^2,
  sqrt_product = sqrt(slope_y_on_x * slope_x_on_y), abs_r = abs(r_sc)
)

```

product	r_squared	sqrt_product	abs_r
0.6926304	0.6926304	0.8322442	0.8322442

The slope of mpg on weight is -0.00765, and the slope of weight on mpg is -90.57. Their product is 0.6926, which is r^2 , and its square root is 0.8322, which is $|r|$. Both slopes are negative, matching the sign of the correlation, -0.8322. The covariance, -5517.4, is a different number: it carries the units of pounds times miles per gallon.

i Watch

Weinberger teaches linear and ridge regression^a in one lecture [21], Lecture 13.

^ahttps://www.youtube.com/watch?v=_21o_ylL0q4

i Watch

Hutchinson teaches linear regression^a (17 min) [22]. The video site requires a login, which enrolled students find in the syllabus.

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#linear_regression

i Read

GeeksforGeeks explains the normal equation^a [23].

^a<https://www.geeksforgeeks.org/machine-learning/ml-normal-equation-in-linear-regression/>

3.5 When $D^\top D$ is not invertible

Definition 3.2 (The Moore–Penrose pseudoinverse and minimum-norm solution). When $D^\top D$ is singular (rank-deficient), the normal equations have infinitely many solutions. Using the eigendecomposition^a $D^\top D = Q\Lambda Q^\top$, where Q is an orthogonal matrix^b of eigenvectors and $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_r, 0, \dots, 0)$ contains eigenvalues with $\lambda_j > 0$, the **Moore–Penrose pseudoinverse** is defined by inverting only the non-zero eigenvalues [24]:

$$\begin{aligned}
 (D^\top D)^+ &= Q\Lambda^+Q^\top \\
 &= Q \begin{bmatrix} 1/\lambda_1 & & & \\ & \ddots & & \\ & & 1/\lambda_r & \\ & & & 0 \end{bmatrix} Q^\top
 \end{aligned}$$

The pseudoinverse solution

$$\begin{aligned}\vec{\beta}^+ &= (D^\top D)^+ D^\top y \\ &= D^+ y\end{aligned}$$

selects the unique least-squares minimizer with the smallest Euclidean norm $\|\vec{\beta}\|_2$.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-eigendecomposition>

^b<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-orthogonal-matrix>

Example 3.11 (Horsepower in two units). In the Auto data [3], add a second copy of horsepower measured in kilowatts, 0.7457 kW per horsepower. The two columns carry the same information, so $D^\top D$ is singular. Fit `mpg` with the pseudoinverse, and compare with the fit on horsepower alone and with another least-squares solution:

3.5.1 Python

```
import numpy as np
import pandas as pd

auto_pi = pd.read_csv("data/auto.csv")
y_pi = auto_pi["mpg"].to_numpy(float)
hp_pi = auto_pi["horsepower"].to_numpy(float)

# Columns: intercept, horsepower, and the same quantity in kilowatts.
Phi_pi = np.column_stack([np.ones(len(y_pi)), hp_pi, 0.7457 * hp_pi])
rank_pi = np.linalg.matrix_rank(Phi_pi)

# The pseudoinverse picks the least-squares solution of smallest norm.
beta_pi = np.linalg.pinv(Phi_pi) @ y_pi

# Any multiple of the null vector of Phi can be added without changing the fit.
null_pi = np.array([0, 0.7457, -1.0])
beta_other_pi = beta_pi + 5 * null_pi
fit_one_pi = np.linalg.lstsq(Phi_pi[:, :2], y_pi, rcond=None)[0]
print(rank_pi, beta_pi)

2 [39.93586102 -0.10143817 -0.07564244]

print(np.allclose(Phi_pi @ beta_pi, Phi_pi @ beta_other_pi), np.allclose(Phi_pi @ beta_pi, Phi_pi[:, :2] @
True True

print(np.linalg.norm(beta_pi), np.linalg.norm(beta_other_pi))
39.936061485610516 40.42017713014687
```

3.5.2 R

```

auto_pi <- read.csv("data/auto.csv")
y_pi <- auto_pi$mpg
hp_pi <- auto_pi$horsepower

# Columns: intercept, horsepower, and the same quantity in kilowatts.
Phi_pi <- cbind(1, hp_pi, 0.7457 * hp_pi)
rank_pi <- qr(Phi_pi)$rank

# The pseudoinverse (MASS::ginv) picks the least-squares solution of smallest norm.
beta_pi <- drop(MASS::ginv(Phi_pi) %*% y_pi)

# Any multiple of the null vector of Phi can be added without changing the fit.
null_pi <- c(0, 0.7457, -1)
beta_other_pi <- beta_pi + 5 * null_pi
fit_one_pi <- coef(lm(mpg ~ horsepower, data = auto_pi))
c(rank = rank_pi, beta = beta_pi)

      rank      beta1      beta2      beta3
2.00000000 39.93586102 -0.10143817 -0.07564244

c(same_fit_as_other = isTRUE(all.equal(drop(Phi_pi %*% beta_pi), drop(Phi_pi %*% beta_other_pi))),
  same_fit_as_one_column = isTRUE(all.equal(drop(Phi_pi %*% beta_pi), drop(Phi_pi[, 1:2] %*% fit_one_pi))),
  same_fit_as_other same_fit_as_one_column
                TRUE                TRUE

c(pinv = sqrt(sum(beta_pi^2)), other = sqrt(sum(beta_other_pi^2)))

      pinv      other
39.93606 40.42018

```

The design matrix has rank 2 with three columns, so the normal equations have infinitely many solutions. The pseudoinverse returns one of them, and it splits the horsepower effect between the two columns. Adding any multiple of the null vector gives another solution with exactly the same predictions, as the second line of output shows, but a larger norm: 40.42 against 39.94. The predictions equal those of the one-feature fit. Only the split between the duplicated columns is arbitrary.

i Read

GeeksforGeeks explains the Moore–Penrose pseudoinverse^a [25].

^a<https://www.geeksforgeeks.org/machine-learning/moore-penrose-pseudoinverse-mathematics/>

Exercise 3.31 (Two ways to lose the inverse). The chapter says $D^\top D$ has no inverse when $n < p^*$ (or $n \leq p$), and also that “any two exactly collinear features make $D^\top D$ singular however large n is”. Those are different failures showing the same symptom.

Take $p = 2$ raw features, so the [design matrix](#) D has $p^* = 3$ columns under the [intercept trick](#).

$$D_A = \begin{bmatrix} 1 & 0 & 2 \\ 1 & 1 & 5 \end{bmatrix}, \quad D_B = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 4 \\ 1 & 3 & 6 \\ 1 & 4 & 8 \end{bmatrix}$$

Compute $\det(D_A^\top D_A)$ and $\det(D_B^\top D_B)$.

Solution

Solution 3.31. Both determinants are zero.

$$D_A^\top D_A = \begin{bmatrix} 2 & 1 & 7 \\ 1 & 1 & 5 \\ 7 & 5 & 29 \end{bmatrix}, \quad D_B^\top D_B = \begin{bmatrix} 4 & 10 & 20 \\ 10 & 30 & 60 \\ 20 & 60 & 120 \end{bmatrix}$$

In $D_B^\top D_B$ the third row is twice the second, so the determinant is zero by inspection. In $D_A^\top D_A$ it takes the cofactor expansion, which gives $2(29 - 25) - 1(29 - 35) + 7(5 - 7) = 8 + 6 - 14 = 0$.

Exercise 3.32 (Two ways to lose the inverse (part 2)). Use the setup of Exercise 3.31.

For each, find a non-zero $\tilde{v}$ with $D\tilde{v} = \vec{0}$, and say what that vector means^a about the fitted model.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Solution

Solution 3.32. A vector in the null space of D is also in the null space of $D^\top D$, so it is enough to find $\tilde{v}$ with $D\tilde{v} = \vec{0}$.

For D_A , the third column is 2 times the first plus 3 times the second:

$$2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 3 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \text{so} \quad \tilde{v}_A = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}$$

For D_B , the third column is 2 times the second, so $\tilde{v}_B = (0, 2, -1)^\top$.

What such a vector means^a: if $\vec{\beta}$ minimizes the risk then so does $\vec{\beta} + t\tilde{v}$, for every real t , because $D(\vec{\beta} + t\tilde{v}) = D\vec{\beta}$ — the two parameter vectors make the **same prediction at every training point**. The fit is not unique. Reporting one $\vec{\beta}$ and reading meaning into its individual entries is then reporting an arbitrary choice from an infinite line of equally good answers.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Exercise 3.33 (Two ways to lose the inverse (part 3)). Use the setup of Exercise 3.31.

One of these two failures can be fixed by collecting more training data and the other cannot. Say which, and why.

Solution

Solution 3.33. D_A 's failure is fixable; D_B 's is not.

D_A has two rows and three columns, and any three vectors in $\mathbb{R}^2$ are linearly dependent. Nothing about the *data* caused this — the shape forces it. Collect a third and fourth point and the columns are free to become independent^a.

D_B has four rows, more than three columns, and is still singular, because its third feature is exactly twice its second. That is a statement about the **feature map** rather than about how many points were collected. Every new point obeys it too, so n can grow without bound and the columns stay dependent. The fix is to drop one of the two features, not to gather more data.

This is the practical reading of the chapter's two conditions: $n < p^*$ ($n \leq p$) is a problem of too little data, and collinearity is a modelling problem.

^a<https://morrison-lab.github.io/pds/independence.html#def-indpt>

Exercise 3.34 (Two ways to lose the inverse (part 4)). Use the setup of Exercise 3.31.

The chapter's remedy is the pseudo-inverse, which returns the solution of smallest $\|\beta\|$. Does that fix the underlying problem in either case?

Solution

Solution 3.34. No, in both cases, and the two failures are worth separating again.

The pseudo-inverse picks one point on that infinite line — the one closest to the origin. That is a well-defined and often sensible choice, but it is a *choice*, not a recovery of information the data does not contain. The predictions it produces are the same as those from any other point on the line, so nothing is lost for prediction on data where the dependence holds.

We lose the individual coefficients. In D_B , features two and three carry identical information, and the minimum-norm solution splits the coefficient between them in a fixed ratio that comes from the norm^a rather than from the data. “This feature has coefficient 0.4” is then a fact about which solution the algorithm^b picked.

And nothing constrains the model off the line the dependence describes. If a future point ever arrives with $x_3 \neq 2x_2$, every point on that line of solutions predicts something different for it, and the training data gives no reason to prefer one.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

^b<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-algorithm>

3.6 When $D^T D$ is nearly singular

Exercise 3.35 (How close to singular?). Example 3.11 made D exactly rank deficient by adding a second copy of horsepower. Exact copies are rare in real data, but nearly dependent columns are common.

The smallest singular value^a of D is 0 exactly when D is rank deficient. How could the smallest singular value tell us that D is *nearly* rank deficient?

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

Solution

Solution 3.35. A small, but not zero, smallest singular value^a says that some combination of the columns of D is close to the zero vector. So the columns are nearly dependent.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

Exercise 3.36 (How close to singular? (part 2)). Use the setup of Exercise 3.35.

Multiply every entry of D by 1000, for instance by changing the units of every column. What happens to the singular values^a, and why does the smallest singular value^b alone not measure how close D is to rank^c deficiency?

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

^b<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

^c<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

Solution

Solution 3.36. Every singular value^a is multiplied by 1000, because $1000 D = \mathbf{U}(1000 \mathbf{D}) \mathbf{V}^T$ for an SVD^b $D = \mathbf{U} \mathbf{D} \mathbf{V}^T$. The smallest singular value grows a thousandfold, yet the columns are exactly as close to dependent as before. The size of the smallest singular value depends on the units of the columns, so it needs a yardstick.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>
^b<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

Exercise 3.37 (How close to singular? (part 3)). Use the setup of Exercise 3.35. What quantity built from the singular values^a is unchanged by that rescaling, and still small or large when D is nearly rank^b deficient?

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>
^b<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

Solution

Solution 3.37. The ratio of the largest singular value^a to the smallest. Both are multiplied by 1000, so the ratio does not change. It is large when the smallest singular value is small compared with the largest, which is when the columns are nearly dependent. It equals ∞ when the columns are exactly dependent.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

Definition 3.3 (Condition number). The **condition number** of an $n \times p$ matrix $\mathbf{A}$ is the ratio of its largest to its smallest singular value^a, counting all $\min(n, p)$ singular values of $\mathbf{A}$, zeros included [26]:

$$\kappa(\mathbf{A}) \stackrel{\text{def}}{=} \frac{\sigma_{\max}}{\sigma_{\min}} = \frac{\sigma_1}{\sigma_{\min(n,p)}}.$$

For a nonzero matrix, it is at least 1. When the columns of a tall matrix $\mathbf{A}$ ($n \geq p$) are dependent, its smallest singular value is 0, and $\kappa(\mathbf{A}) = \infty$.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

- Other norms give other condition numbers. These notes use the one built from the 2-norm, the singular values.
- Squaring the singular values of D gives those of $D^T D$, so $\kappa(D^T D) = \kappa(D)^2$.
- For the symmetric matrix $D^T D$, the singular values are its eigenvalues, so the condition number is the ratio of its largest to its smallest eigenvalue. Gradient Descent shows that this ratio sets how fast gradient descent converges.
- The definition follows Trefethen and Bau [27], Lecture 12.

Example 3.12 (The condition number of an intercept and horsepower). In the `Auto` data [3], take D with a column of ones and a column of horsepower. Compute its singular values and their ratio, then repeat after multiplying every entry by 1000:

3.6.1 Python

```
import numpy as np
import pandas as pd

auto_cn = pd.read_csv("data/auto.csv")
Phi_cn = np.column_stack([np.ones(len(auto_cn)), auto_cn["horsepower"].to_numpy(float)])

sv_cn = np.linalg.svd(Phi_cn, compute_uv=False)
sv_cn_1000 = np.linalg.svd(1000 * Phi_cn, compute_uv=False)
print(sv_cn, sv_cn[0] / sv_cn[-1])

[2204.05745265    6.83706171] 322.3691031299226
```

```
print(sv_cn_1000, sv_cn_1000[0] / sv_cn_1000[-1])
[2204057.45265116    6837.06171358] 322.3691031299226
print(np.linalg.cond(Phi_cn))
322.3691031299226
```

3.6.2 R

```
auto_cn <- read.csv("data/auto.csv")
Phi_cn <- cbind(1, auto_cn$horsepower)

sv_cn <- svd(Phi_cn)$d
sv_cn_1000 <- svd(1000 * Phi_cn)$d
print(sv_cn)
```

```
[1] 2204.057453    6.837062
```

```
print(sv_cn[1] / sv_cn[2])
```

```
[1] 322.3691
```

```
print(sv_cn_1000)
```

```
[1] 2204057.453    6837.062
```

```
print(sv_cn_1000[1] / sv_cn_1000[2])
```

```
[1] 322.3691
```

```
kappa(Phi_cn, exact = TRUE)
```

```
[1] 322.3691
```

The singular values are 2204.1 and 6.84, so $\kappa(D) = 322.4$. After the rescaling the singular values become 2.2×10^6 and 6837, and the condition number is still 322.4. The squared matrix $D^\top D$ has condition number $\kappa(D)^2 = 1.0 \times 10^5$.

Definition 3.4 (Ill-conditioned and well-conditioned matrices). A matrix $\mathbf{A}$ is **ill-conditioned** when its **condition number** is so large that small relative changes in its entries, such as rounding errors, can change the solution of a linear system by a much larger relative amount. The relative change in the solution can be up to $\kappa(\mathbf{A})$ times the relative change in the entries [27], Lecture 12. A solution computed with $\kappa(\mathbf{A}) \approx 10^k$ therefore loses about k of the digits that the floating-point arithmetic carries. The matrix $\mathbf{A}$ is **well-conditioned** when its condition number is close to 1. There is no sharp threshold between the two: how large is too large depends on how many accurate digits the answer needs.

Example 3.13 (Polynomials in horsepower). Fit mpg to a polynomial in **horsepower** of degree $d = 1, 2, 3, 4$ [3]. The columns of D are $1, x, \dots, x^d$. Compute $\kappa(D)$ for raw horsepower x , and again for the standardized horsepower $z = (x - \bar{x})/s_x$:

3.6.3 Python

```
import numpy as np  
import pandas as pd  
  
auto_ic = pd.read_csv("data/auto.csv")  
x_ic = auto_ic["horsepower"].to_numpy(float)  
z_ic = (x_ic - x_ic.mean()) / x_ic.std(ddof=1)  
  
def poly_cond(v, d):  
    return np.linalg.cond(np.column_stack([v ** k for k in range(d + 1)]))  
  
degrees_ic = [1, 2, 3, 4]  
cond_raw_ic = [poly_cond(x_ic, d) for d in degrees_ic]  
cond_std_ic = [poly_cond(z_ic, d) for d in degrees_ic]  
eps_ic = np.finfo(float).eps  
print(cond_raw_ic)  
  
[np.float64(322.3691031299226), np.float64(128782.4331456845), np.float64(55313928.96961322), np.float64(15314271.  
print(cond_std_ic)  
  
[np.float64(1.001277955793012), np.float64(3.5283967220330443), np.float64(10.474705288948204), np.float64(11.51799234  
print(eps_ic)  
  
2.220446049250313e-16
```

```
auto_ic <- read.csv("data/auto.csv")
x_ic <- auto_ic$horsepower
z_ic <- (x_ic - mean(x_ic)) / sd(x_ic)

poly_cond <- function(v, d) kappa(outer(v, 0:d, `^`), exact = TRUE)

degrees_ic <- 1:4
cond_raw_ic <- sapply(degrees_ic, poly_cond, v = x_ic)
cond_std_ic <- sapply(degrees_ic, poly_cond, v = z_ic)
print(cond_raw_ic)

[1] 3.223691e+02 1.287824e+05 5.531393e+07 2.729018e+10

print(cond_std_ic)

[1] 1.001278 3.528397 10.474705 48.472291

.Machine$double.eps

[1] 2.220446e-16
```

4 Linear regression in code

4.1 Fitting models in Python, R, and Julia

i Exact normal equations vs. matrix decomposition

Solving the linear system $(D^\top D)\vec{\beta} = D^\top y$ directly via LU (lower-upper)^a or Cholesky^b decomposition is both faster and more accurate than computing the explicit matrix inverse $(D^\top D)^{-1}$. Modern library solvers (`np.linalg.lstsq`, `lm()`, Julia's `\`) go further by decomposing D directly using QR^c or singular value decomposition (SVD)^d, which avoids squaring the matrix **condition number** and handles rank-deficient systems. All three languages use [the synthetic data](#).

^a<https://morrison-lab.github.io/mds/linear-algebra-projections.html#def-lu>

^b<https://morrison-lab.github.io/mds/linear-algebra-diagonalization.html#thm-cholesky>

^c<https://morrison-lab.github.io/mds/linear-algebra-projections.html#thm-qr>

^d<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

i Watch

Hastie and Tibshirani fit the same models in code in their Python lab^a [17].

^a<https://www.youtube.com/watch?v=mKalBNrxToU>

4.1.1 Python

```
import numpy as np

data = r.synthetic.to_numpy()
x1, x2, y = data[:, 0], data[:, 1], data[:, 3]
n = len(x1)

X = np.column_stack([np.ones(n), x1, x2])

beta_normal = np.linalg.solve(X.T @ X, X.T @ y)
beta = np.linalg.lstsq(X, y, rcond=None)[0]

print(np.allclose(beta_normal, beta))
```

True

```
print(beta.round(2))
```

```
[ 2.01  2.97 -1.  ]
```

4.1.2 R

```
data <- synthetic
n <- nrow(data)

X <- cbind(1, data$x1, data$x2)

beta_normal <- solve(t(X) %*% X, t(X) %*% data$y)
beta <- coef(lm(y ~ x1 + x2, data = data))

all.equal(as.vector(beta_normal), unname(beta))
```

```
[1] TRUE
```

```
round(beta, 2)
```

```
(Intercept)      x1      x2
          2.01    2.97   -1.00
```

4.1.3 Julia

```
using LinearAlgebra

data = synthetic;
n = size(data, 1);
x1, x2, y = data[:, 1], data[:, 2], data[:, 4];

X = [ones(n) x1 x2];

beta_normal = (X' * X) \ (X' * y);
beta = X \ y;

println(isapprox(beta_normal, beta))

true

println(round.(beta, digits = 2))

[2.01, 2.97, -1.0]
```

4.2 What happens when D is rank deficient

i Structural collinearity across libraries

Setting $x_2 = 2x_1$ makes D have rank^a 2 instead of 3, rendering $D^T D$ singular. Each library detects and resolves rank deficiency on the shared dataset according to its own design conventions: NumPy and Julia return the minimum Euclidean norm solution $\hat{\beta}^+$, while R drops the aliased predictor and returns NA.

^a<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

4.2.1 Python

```
import numpy as np

data = r.synthetic.to_numpy()
x1, y = data[:, 0], data[:, 4]          # y_collinear target
n = len(x1)
x2 = 2 * x1                            # exactly collinear with x1
X = np.column_stack([np.ones(n), x1, x2])

print(np.linalg.matrix_rank(X))

2

try:
    np.linalg.solve(X.T @ X, X.T @ y)
except np.linalg.LinAlgError as err:
    print("normal equations:", err)

normal equations: Singular matrix
```

```
beta = np.linalg.lstsq(X, y, rcond=None)[0]
print(beta.round(3))
```

```
[2.011 0.193 0.387]
```

4.2.2 R

```
data <- synthetic
x1 <- data$x1
x2 <- 2 * x1                                # exactly collinear with x1
y <- data$y_collinear
X <- cbind(1, x1, x2)

print(qr(X)$rank)
```

```
[1] 2
```

```
attempt <- try(solve(t(X) %*% X, t(X) %*% y), silent = TRUE)
if (inherits(attempt, "try-error")) {
  cat("normal equations:", conditionMessage(attr(attempt, "condition")), "\n")
}
```

```
normal equations: Lapack routine dgesv: system is exactly singular: U[3,3] = 0
```

```
print(round(coef(lm(y ~ x1 + x2)), 3))
```

```
(Intercept)      x1      x2
      2.011      0.967      NA
```

4.2.3 Julia

```
using LinearAlgebra
```

```
data = synthetic;
n = size(data, 1);
x1, y = data[:, 1], data[:, 5];
# exactly collinear with x1
x2 = 2 .* x1;
X = [ones(n) x1 x2];

println(rank(X))
```

```
2
```

```
try
  (X' * X) \ (X' * y)
catch err
  println("normal equations: ", err)
end
```

```
normal equations: SingularException(3)
```

```
println(round.(X \ y, digits = 3))
```

```
[2.011, 0.193, 0.387]
```

```
# The Python tab's minimum-norm answer, and the same data fitted on (1, x1)
# alone, which is R's representative.
rd_py = beta
```

```

X1 = X[:, :2]
b1 = np.linalg.lstsq(X1, y, rcond=None)[0]
rd_slope = b1[1]
rd_rss_min_norm = float((y - X @ beta) @ (y - X @ beta))
rd_rss_dropped = float((y - X1 @ b1) @ (y - X1 @ b1))

rd_r <- coef(lm(y ~ x1 + x2))
rd_r_combo <- rd_r[["x1"]] + 2 * ifelse(is.na(rd_r[["x2"]]), 0, rd_r[["x2"]])

rd_jl = X \ y;

rd_py <- reticulate::py$rd_py
rd_jl <- jl("rd_jl")
combo <- function(b) b[2] + 2 * b[3]
rss_gap <- abs(reticulate::py$rd_rss_dropped - reticulate::py$rd_rss_min_norm)
rss_gap_text <- if (rss_gap == 0) {
  "are identical"
} else {
  paste0("differ by $", sci(rss_gap), "$, the size of floating-point rounding")
}

```

i Comparing rank-deficient fits on identical data

1. **Refusal of the normal equations:** Under exact collinearity, standard direct solvers refuse to invert $D^\top D$: R detects computational singularity, NumPy raises `LinAlgError: Singular matrix`, and Julia raises `SingularException(3)`. Explicitly forming $D^\top D$ squares [condition numbers](#) and destroys information preserved in D .
2. **All three survive QR^a/SVD^b factorization on the same data:** Because $x_2 = 2x_1$, the data only identifies the combination $\beta_1 + 2\beta_2 = 1$: Python and Julia both yield the minimum Euclidean norm solution (2.011, 0.193, 0.387), where $0.193 + 2(0.387) = 0.967$. R drops the aliased column and returns (2.011, 0.967, NA), where $0.967 + 2(0) = 0.967$.
3. **Different parameter representations yield identical predictions:** Fitting y on $(1, x_1)$ alone yields residual sum of squares 0.461691312, matching `lstsq`'s 0.461691312 (are identical).

^a<https://morrison-lab.github.io/mds/linear-algebra-projections.html#thm-qr>

^b<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

4.3 Numerical stability and near-collinearity

```

import numpy as np

data = r.synthetic.to_numpy()
x1, jitter, y = data[:, 0], data[:, 2], data[:, 4]
n = len(x1)

def report(name, b, X, y):
    resid = y - X @ b
    print("    %-16s -> largest coefficient %8.1e, RSS %.9f"
          % (name, np.abs(b).max(), resid @ resid))

for eps in (1e-4, 1e-12, 0.0):
    x2 = 2 * x1 + eps * jitter
    X = np.column_stack([np.ones(n), x1, x2])
    cx = np.linalg.cond(X)
    print("perturbation %-7g cond(X) = %7.1e  cond(X)^2 = %7.1e  "
          "cond(X'X) = %7.1e" % (eps, cx, cx * cx, np.linalg.cond(X.T @ X)))
    # eps = 0 is exact collinearity

```

```

try:
    report("normal equations", np.linalg.solve(X.T @ X, X.T @ y), X, y)
except np.linalg.LinAlgError as err:
    print("    normal equations -> %s" % err)
report("lstsq", np.linalg.lstsq(X, y, rcond=None)[0], X, y)

```

```

perturbation 0.0001  cond(X) = 4.7e+04   cond(X)^2 = 2.2e+09   cond(X'X) = 2.2e+09
    normal equations -> largest coefficient  5.0e+02, RSS 0.436741178
    lstsq           -> largest coefficient  5.0e+02, RSS 0.436741178
perturbation 1e-12   cond(X) = 4.7e+12   cond(X)^2 = 2.2e+25   cond(X'X) = 6.8e+15
    normal equations -> largest coefficient  2.3e+01, RSS 0.461691312
    lstsq           -> largest coefficient  5.0e+10, RSS 0.436751759
perturbation 0       cond(X) = 3.0e+16   cond(X)^2 = 8.8e+32   cond(X'X) =      inf
    normal equations -> Singular matrix
    lstsq           -> largest coefficient  2.0e+00, RSS 0.461691312

```

```

# Re-runs the loop above for 1e-4 and 1e-12, storing what it printed so
# the prose can quote it.

```

```

def summary(b, X, y):
    resid = y - X @ b
    return dict(largest=float(np.abs(b).max()), rss=float(resid @ resid))

near = {}
for eps in (1e-4, 1e-12):
    x2 = 2 * x1 + eps * jitter
    X = np.column_stack([np.ones(n), x1, x2])
    near["%g" % eps] = dict(
        cond=float(np.linalg.cond(X)),
        cond_xtx=float(np.linalg.cond(X.T @ X)),
        normal=summary(np.linalg.solve(X.T @ X, X.T @ y), X, y),
        lstsq=summary(np.linalg.lstsq(X, y, rcond=None)[0], X, y),
    )

```

```

near <- reticulate::py$near
n4 <- near[["0.0001"]]
n12 <- near[["1e-12"]]
stopifnot(n12$lstsq$rss < n12$normal$rss)
orders <- function(a, b) round(abs(log10(a / b)))
cond_gap <- sprintf(
    "%d orders of magnitude %s",
    orders(n12$cond_xtx, n12$cond^2),
    if (n12$cond_xtx < n12$cond^2) "below" else "above"
)

```

i Condition number squaring and catastrophic cancellation

- **Perturbation $\epsilon = 10^{-4}$:** Both routes agree; $\text{cond}(D^T D) = 2.2 \times 10^9$ equals $\text{cond}(D)^2 = 2.2 \times 10^9$, confirming the condition number squaring.
- **Perturbation $\epsilon = 10^{-12}$:** The normal equations return coefficients of magnitude 23, while `lstsq` inflates coefficients to 5.0×10^{10} to fit noise slightly better (a difference of 9 orders of magnitude).
- **Loss of precision:** At $\epsilon = 10^{-12}$, $\text{cond}(D^T D)$ comes out 10 orders of magnitude below $\text{cond}(D)^2$ because the smallest singular values^a have rounded away to machine epsilon noise.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

i Software versions and platform reproduction

Rendered with NumPy 2.5.4 on Python 3.13.18, R 4.6.1, and Julia 1.13.3. Where a near-singular or rank-deficient matrix falls relative to a solver's threshold depends on the data, floating-point summation order, and the underlying LAPACK build. While exact collinearity mathematically produces zero pivots in exact arithmetic, finite-precision rounding in matrix products can yield condition numbers near machine precision.

5 Predictions, geometry, and extensions

5.1 Predictions and the hat matrix

Definition 5.1 (Prediction at a new point). For an unseen input point $\underline{x}_{\text{new}} \in \mathbb{R}^p$, the design vector is $\underline{D}(\underline{x}_{\text{new}}) = [1, x_{\text{new},1}, \dots, x_{\text{new},p}]^\top \in \mathbb{R}^p$. The linear model makes the point prediction:

$$\begin{aligned}\hat{y}_{\text{new}} &= h(\underline{x}_{\text{new}}) \\ &= \underline{D}(\underline{x}_{\text{new}}) \cdot \hat{\underline{\beta}} \\ &= \underline{D}(\underline{x}_{\text{new}}) \cdot (D^\top D)^{-1} D^\top \underline{y}\end{aligned}$$

Definition 5.2 (The hat matrix). The fitted values on the training data are given by

$$\begin{aligned}\hat{\underline{y}} &= D \hat{\underline{\beta}} \\ &= D(D^\top D)^{-1} D^\top \underline{y}.\end{aligned}$$

The operator mapping observed responses $\underline{y}$ to fitted responses $\hat{\underline{y}}$ is the hat matrix^a $H \in \mathbb{R}^{n \times n}$:

$$H = D(D^\top D)^{-1} D^\top$$

so that $\hat{\underline{y}} = H \underline{y}$ (“ H puts a hat on $\underline{y}$ ”) [28].

^a<https://morrison-lab.github.io/sds/correlation-regression.html>

Example 5.1 (The hat matrix of the synthetic data). Build H for the design matrix of [the synthetic data](#), and check that $H \underline{y}$ equals the fitted values $D \hat{\underline{\beta}}$:

5.1.1 Python

```
import numpy as np

synthetic = r.synthetic.to_numpy()
y_syn = synthetic[:, 3]
X_syn = np.column_stack([np.ones(len(y_syn)), synthetic[:, 0], synthetic[:, 1]])
H_syn = X_syn @ np.linalg.inv(X_syn.T @ X_syn) @ X_syn.T
beta_syn = np.linalg.solve(X_syn.T @ X_syn, X_syn.T @ y_syn)
print(H_syn.shape, np.allclose(H_syn @ y_syn, X_syn @ beta_syn))

(50, 50) True
```

5.1.2 R

```

X_syn <- model.matrix(~ x1 + x2, data = synthetic)
H_syn <- X_syn %*% solve(crossprod(X_syn)) %*% t(X_syn)
fit_syn <- lm(y ~ x1 + x2, data = synthetic)
dim(H_syn)

[1] 50 50

all.equal(as.vector(H_syn %*% synthetic$y), unname(fitted(fit_syn)))

[1] TRUE

```

H is an $n \times n$ matrix even though the model has only 3 coefficients, because it maps all n responses to all n fitted values.

The Hat Matrix $H = X(X^\top X)^{-1}X^\top$ as Orthogonal Projection

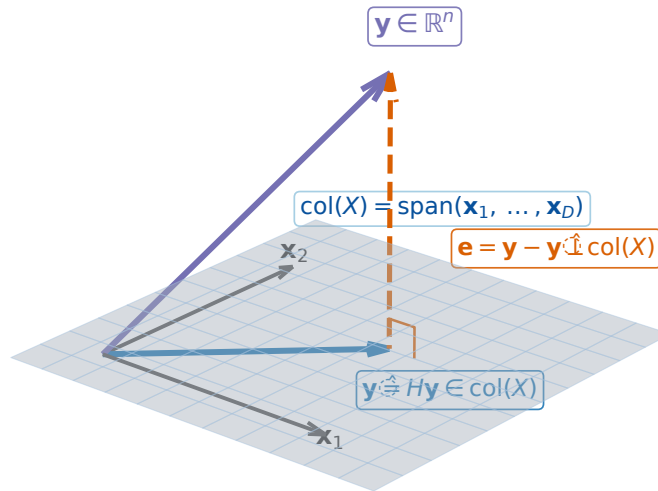


Figure 13: The hat matrix $H = D(D^\top D)^{-1}D^\top$ orthogonally projects¹ the target vector $y \in \mathbb{R}^n$ onto the column space $\text{col}(D)$, minimizing the Euclidean length of the residual vector $y - \hat{y}$.

Theorem 5.1 (The hat matrix is an orthogonal projection). *The hat matrix H of Definition 5.2 is an orthogonal projection matrix^a:*

1. H is symmetric: $H^\top = H$.
2. H is idempotent: $H^2 = H$.
3. The residual vector $y - \hat{y} = (I - H)y$ is orthogonal^b to every column of D : $D^\top(y - \hat{y}) = \vec{0}$.

Geometrically, as illustrated in Figure 13, H projects the observation vector $y \in \mathbb{R}^n$ orthogonally onto the column space of D .

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-projection-matrix>

^b<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-orthogonal-vectors>

i Proof

Proof.

1. Symmetry:

$$\begin{aligned} H^\top &= (D(D^\top D)^{-1} D^\top)^\top \\ &= D(D^\top D)^{-1} D^\top \\ &= H. \end{aligned}$$

2. Idempotence:

$$\begin{aligned} H^2 &= HH \\ &= D(D^\top D)^{-1} D^\top D(D^\top D)^{-1} D^\top \\ &= D(D^\top D)^{-1} D^\top \\ &= H. \end{aligned}$$

3. Orthogonality of the residuals:

$$\begin{aligned} D^\top(\tilde{y} - H\tilde{y}) &= D^\top\tilde{y} - D^\top D(D^\top D)^{-1} D^\top\tilde{y} \\ &= D^\top\tilde{y} - D^\top\tilde{y} \\ &= \vec{0}. \end{aligned}$$

□

Example 5.2 (Checking the projection properties). For the hat matrix of Example 5.1, check each property of Theorem 5.1 (each check prints a true value if the property holds within rounding error):

5.1.3 Python

```
resid_syn = y_syn - H_syn @ y_syn
print(np.allclose(H_syn, H_syn.T),
      np.allclose(H_syn @ H_syn, H_syn),
      np.allclose(X_syn.T @ resid_syn, 0))
```

True True True

5.1.4 R

```
resid_syn <- synthetic$y - H_syn %*% synthetic$y
c(symmetric = isTRUE(all.equal(H_syn, t(H_syn))),
  idempotent = isTRUE(all.equal(H_syn %*% H_syn, H_syn)),
  orthogonal = isTRUE(all.equal(as.vector(t(X_syn) %*% resid_syn), rep(0, ncol(X_syn)))))

symmetric idempotent orthogonal
      TRUE         TRUE         TRUE
```

Definition 5.3 (Leverage). The *leverage* of observation i is the i th diagonal entry of the hat matrix H of Definition 5.2:

$$\begin{aligned} h_i &= H_{ii} \\ &= \tilde{D}(\tilde{x}_i)^\top (D^\top D)^{-1} \tilde{D}(\tilde{x}_i). \end{aligned}$$

It quantifies the influence of observation i on its own fitted response, since $\hat{y}_i = h_i y_i + \sum_{j \neq i} H_{ij} y_j$ [29].

Example 5.3 (Leverages of the synthetic data). Compute the leverages of the synthetic data of Example 5.1, and show the five largest:

5.1.5 Python

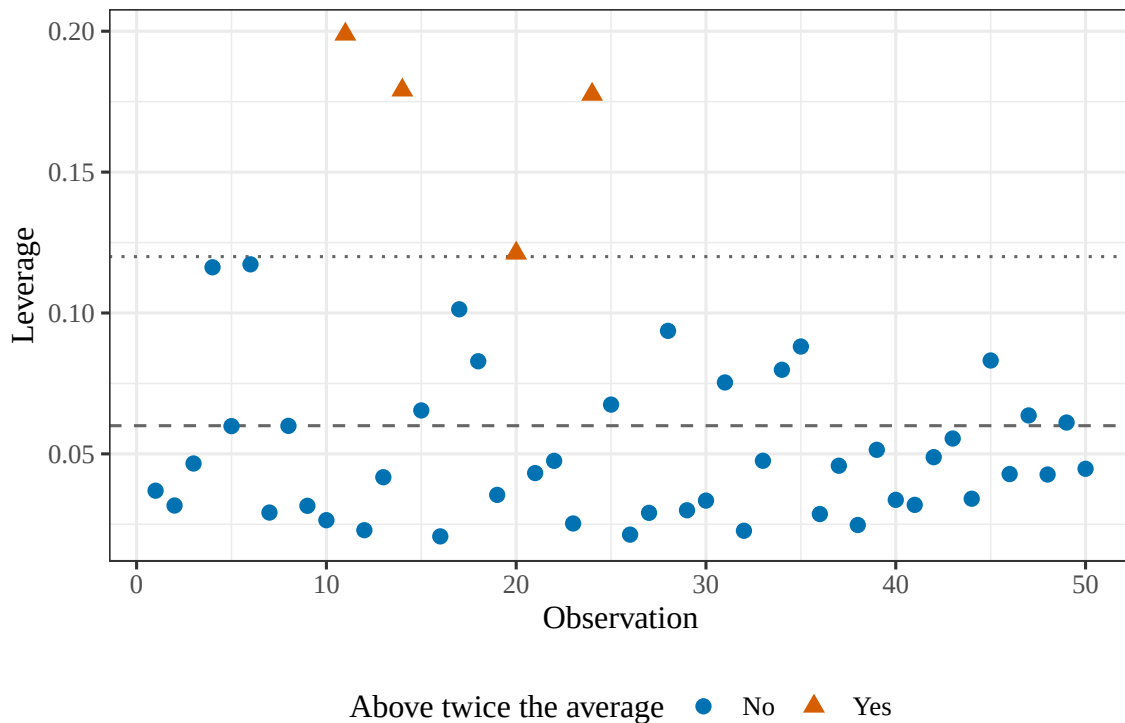
```
h_syn = np.diag(H_syn)
np.round(np.sort(h_syn)[-5:], 4)
array([0.1173, 0.1212, 0.1776, 0.1791, 0.1989])
```

5.1.6 R

```
h_syn <- diag(H_syn)
round(sort(h_syn, decreasing = TRUE)[1:5], 4)
      11      14      24      20      6
0.1989 0.1791 0.1776 0.1212 0.1173
```

5.1.7 R

```
library(ggplot2)
lvf <- data.frame(
  observation = seq_along(h_syn),
  leverage = h_syn,
  high = h_syn > 2 * mean(h_syn)
)
ggplot(lvf, aes(observation, leverage, colour = high, shape = high)) +
  geom_hline(yintercept = mean(h_syn), linetype = "dashed", colour = "grey40") +
  geom_hline(yintercept = 2 * mean(h_syn), linetype = "dotted", colour = "grey40") +
  geom_point(size = 2.5) +
  scale_colour_manual(
    values = c("FALSE" = "#0072B2", "TRUE" = "#D55E00"),
    labels = c("FALSE" = "No", "TRUE" = "Yes")
  ) +
  scale_shape_manual(
    values = c("FALSE" = 16, "TRUE" = 17),
    labels = c("FALSE" = "No", "TRUE" = "Yes")
  ) +
  labs(
    x = "Observation", y = "Leverage",
    colour = "Above twice the average", shape = "Above twice the average"
  )
)
```



(a)

5.1.8 Python

```
import matplotlib.pyplot as plt
import numpy as np

obs_lvf = np.arange(1, len(h_syn) + 1)
high_lvf = h_syn > 2 * h_syn.mean()
fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.axhline(h_syn.mean(), color="grey", linestyle="--");
ax.axhline(2 * h_syn.mean(), color="grey", linestyle=":");
ax.plot(obs_lvf[~high_lvf], h_syn[~high_lvf], color="#0072B2", marker="o", linestyle="");
```

Theorem 5.2 (Bounds on the leverage). *Let p^* be the number of columns of D . The leverages of Definition 5.3 satisfy $0 \leq h_i \leq 1$ for every i , and their mean is*

$$\begin{aligned}\bar{h} &= \frac{\text{tr}(H)}{n} \\ &= \frac{p^*}{n}.\end{aligned}$$

i Proof

Proof. By Theorem 5.1, H is symmetric and idempotent, so

$$\begin{aligned}h_i &= H_{ii} \\ &= (H^2)_{ii} \\ &= \sum_{j=1}^n H_{ij}H_{ji} \\ &= \sum_{j=1}^n H_{ij}^2 \\ &= h_i^2 + \sum_{j \neq i} H_{ij}^2.\end{aligned}$$

The sum of squares is nonnegative, so $h_i \geq 0$. Also $h_i \geq h_i^2$, so $h_i \leq 1$. By the cyclic property of the trace,

$$\begin{aligned}\text{tr}(H) &= \text{tr}(D(D^\top D)^{-1}D^\top) \\ &= \text{tr}((D^\top D)^{-1}D^\top D) \\ &= \text{tr}(I_{p^*}) \\ &= p^*.\end{aligned}$$

Dividing by n gives $\bar{h} = p^*/n$. □

Example 5.4 (Checking the leverage bounds). For the leverages of Example 5.3, the smallest and largest values and the mean are:

5.1.9 Python

```
print(h_syn.min(), h_syn.max(), h_syn.mean(), X_syn.shape[1] / len(h_syn))
0.020715599268734834 0.19892097513195148 0.06 0.06
```

5.1.10 R

```
c(min = min(h_syn), max = max(h_syn), mean = mean(h_syn), p_star_over_n = ncol(X_syn) / length(h_syn))
```

min	max	mean	p_star_over_n
0.0207156	0.1989210	0.0600000	0.0600000

All leverages lie between 0 and 1, and the mean equals p^*/n with $p^* = 3$ and $n = 50$.

Theorem 5.3 (Least squares as maximum likelihood estimation). *Assume the true data-generating process is linear with additive Gaussian noise:*

$$y_i = D_{\sim}(x_i) \cdot \vec{\beta} + \varepsilon_i, \quad \varepsilon_i \stackrel{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$$

Then the ordinary least squares estimator $\hat{\beta} = (D^\top D)^{-1} D^\top y$ is identical to the maximum likelihood estimator (MLE)^a of $\vec{\beta}$, so squared error is the plausible loss under additive Gaussian noise.

^a<https://morrison-lab.github.io/sds/intro-MLEs.html#def-mle>

i Proof

Proof. Because errors ε_i are independent, the conditional likelihood of y given D factors into:

$$p(\tilde{y} \mid D, \vec{\beta}, \sigma^2) = \prod_{i=1}^n \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left\{ -\frac{(y_i - D(\tilde{x}_i) \cdot \vec{\beta})^2}{2\sigma^2} \right\}$$

Taking the log-likelihood yields:

$$\log p(\tilde{y} \mid D, \vec{\beta}, \sigma^2) = -\frac{n}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - D(\tilde{x}_i) \cdot \vec{\beta})^2$$

Maximizing this log-likelihood with respect to $\vec{\beta}$ is mathematically equivalent to minimizing $\sum_{i=1}^n (y_i - D(\tilde{x}_i) \cdot \vec{\beta})^2 = \|D\vec{\beta} - y\|^2$, which is the [least-squares objective](#). $\square$

Example 5.5 (Maximizing the Gaussian likelihood recovers least squares). In the `Auto` data [3], model `mpg` as a linear function of `horsepower` plus Gaussian noise. Let the noise variance be $\sigma^2 = 25$. The maximizing coefficients do not depend on this choice, as the proof of Theorem 5.3 shows. Maximize the log-likelihood numerically from $\vec{\beta} = \vec{0}$, and compare with the least-squares solution:

5.1.11 Python

```
import numpy as np
import pandas as pd
from scipy.optimize import minimize

auto_mle = pd.read_csv("data/auto.csv")
y_mle = auto_mle["mpg"].to_numpy(float)
# Centre horsepower so the optimizer is not slowed by badly scaled coordinates;
# the intercept then means "mpg at average horsepower", and the slope is unchanged.
hp_mean_mle = auto_mle["horsepower"].mean()
Phi_mle = np.column_stack([np.ones(len(y_mle)), auto_mle["horsepower"] - hp_mean_mle])
var_mle = 25.0

def neg_loglik_mle(b):
    """Negative Gaussian log-likelihood of the responses given the coefficients b."""
    resid = y_mle - Phi_mle @ b
    return 0.5 * len(y_mle) * np.log(2 * np.pi * var_mle) + 0.5 * np.sum(resid ** 2) / var_mle

mle_fit = minimize(neg_loglik_mle, x0=np.zeros(2), method="BFGS", options={"gtol": 1e-8})
ols_mle = np.linalg.lstsq(Phi_mle, y_mle, rcond=None)[0]
print(mle_fit.x, ols_mle)

[23.44591852 -0.15784474] [23.44591837 -0.15784473]

print(np.allclose(mle_fit.x, ols_mle, atol=1e-4))

True
```

5.1.12 R

```

auto_mle <- read.csv("data/auto.csv")
y_mle <- auto_mle$mpg
# Centre horsepower so the optimizer is not slowed by badly scaled coordinates;
# the intercept then means "mpg at average horsepower".
Phi_mle <- cbind(1, auto_mle$horsepower - mean(auto_mle$horsepower))
var_mle <- 25

# Negative Gaussian log-likelihood of the responses given the coefficients b.
neg_loglik_mle <- function(b) {
  resid <- y_mle - Phi_mle %*% b
  0.5 * length(y_mle) * log(2 * pi * var_mle) + 0.5 * sum(resid^2) / var_mle
}

mle_fit <- optim(c(0, 0), neg_loglik_mle, method = "BFGS", control = list(reltol = 1e-12))
ols_mle <- coef(lm(y_mle ~ Phi_mle[, 2]))
rbind(mle = mle_fit$par, ols = unname(ols_mle))

```

```

      [,1]      [,2]
mle 23.44592 -0.1578447
ols 23.44592 -0.1578447

```

The numerical maximizer and the least-squares solution agree to within $5.4\text{e-}10$. Changing `var_mle` to any other positive value changes the log-likelihood but not where it peaks, since the variance only scales the sum of squared residuals.

5.2 Multivariate linear regression

Definition 5.4 (Multivariate linear regression). When predicting multiple target variables ($C > 1$), each response vector is $\tilde{y}_i \in \mathbb{R}^C$. The linear model assumes:

$$\tilde{y} = B^\top D(\tilde{x}) + \tilde{\varepsilon}, \quad D(\tilde{x}) \in \mathbb{R}^{p^*}, \quad B \in \mathbb{R}^{p^* \times C}$$

where noise vectors $\tilde{\varepsilon} \sim \mathcal{N}(\vec{0}, \Sigma)$ have zero mean and covariance matrix $\Sigma \in \mathbb{R}^{C \times C}$.

In matrix notation over all n instances:

$$Y = DB + E$$

where $Y \in \mathbb{R}^{n \times C}$ stacks target rows, $D \in \mathbb{R}^{n \times p^*}$ is the design matrix, and $E \in \mathbb{R}^{n \times C}$ stacks noise rows. The objective minimizes the squared Frobenius norm^a of the residual matrix:

$$\hat{B} = \arg \min_B \|Y - DB\|_F^2$$

Write $Y_{*,k}$ and $B_{*,k}$ for the k -th columns of Y and B , respectively.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-frobenius-norm>

Example 5.6 (Two targets at once). In the Auto data [3], let the targets be `mpg` and `weight`, so $C = 2$, and let the features be an intercept and `horsepower`, so $p^* = 2$. Build Y , D , the fitted $\hat{B}$ and the residual matrix E of Definition 5.4, and estimate the noise covariance Σ from the residuals:

5.2.1 Python

```

import numpy as np
import pandas as pd

auto_mr = pd.read_csv("data/auto.csv")
Y_mr = auto_mr[["mpg", "weight"]].to_numpy(float)
Phi_mr = np.column_stack([np.ones(len(Y_mr)), auto_mr["horsepower"]])

# One column of B per target; lstsq fits all columns at once.
B_mr = np.linalg.lstsq(Phi_mr, Y_mr, rcond=None)[0]
E_mr = Y_mr - Phi_mr @ B_mr
# The C x C covariance of the noise: average outer product of the residual rows.
Sigma_mr = E_mr.T @ E_mr / len(Y_mr)
print(Y_mr.shape, Phi_mr.shape, B_mr.shape, E_mr.shape)

(392, 2) (392, 2) (2, 2) (392, 2)

print(np.array2string(Sigma_mr, precision=1, suppress_small=True))

[[ 23.9 -1053.2]
 [-1053.2 181763.8]]

# Bootstrap: resample cars with replacement, refit both targets,
# and recompute the off-diagonal covariance entry each time.
rng_mr = np.random.default_rng(571)
boot_cov_mr = np.empty(2000)
for b in range(2000):
    rows = rng_mr.integers(0, len(Y_mr), len(Y_mr))
    coef_b = np.linalg.lstsq(Phi_mr[rows], Y_mr[rows], rcond=None)[0]
    resid_b = Y_mr[rows] - Phi_mr[rows] @ coef_b
    boot_cov_mr[b] = (resid_b.T @ resid_b / len(rows))[0, 1]
# The 2.5th and 97.5th percentiles of the resampled entries.
cov_ci_mr = np.quantile(boot_cov_mr, [0.025, 0.975])
print(cov_ci_mr)

[-1277.54117641 -822.01445198]

```

5.2.2 R

```

auto_mr <- read.csv("data/auto.csv")
Y_mr <- as.matrix(auto_mr[, c("mpg", "weight")])
Phi_mr <- cbind(1, auto_mr$horsepower)

# One column of B per target; lm with a matrix response fits all columns at once.
B_mr <- coef(lm(Y_mr ~ auto_mr$horsepower))
E_mr <- Y_mr - Phi_mr %*% B_mr
# The C x C covariance of the noise: average outer product of the residual rows.
Sigma_mr <- crossprod(E_mr) / nrow(Y_mr)
rbind(Y = dim(Y_mr), Phi = dim(Phi_mr), B = dim(B_mr), E = dim(E_mr))

[,1] [,2]
Y    392    2
Phi  392    2
B      2    2
E    392    2

round(Sigma_mr, 1)

      mpg    weight
mpg    23.9 -1053.2
weight -1053.2 181763.8

```

```
# Bootstrap: resample cars with replacement, refit both targets,
# and recompute the off-diagonal covariance entry each time.
set.seed(571)
boot_cov_mr <- replicate(2000, {
  rows <- sample(nrow(Y_mr), replace = TRUE)
  coef_b <- coef(lm(Y_mr[rows, ] ~ auto_mr$horsepower[rows]))
  resid_b <- Y_mr[rows, ] - Phi_mr[rows, ] %*% coef_b
  crossprod(resid_b)[1, 2] / length(rows)
})
# The 2.5th and 97.5th percentiles of the resampled entries.
cov_ci_mr <- quantile(boot_cov_mr, c(0.025, 0.975))
cov_ci_mr
```

```
      2.5%      97.5%
-1270.0709 -818.2046
```

Y and E have one row per car and one column per target, D has one column per feature, and $\hat{B}$ has one column per target, so its shape is $p^* \times C$. The diagonal of the covariance matrix holds the residual variance of each target. The off-diagonal entry, -1053.2, is negative: cars whose `mpg` is above the line tend to be lighter than the line predicts, so the errors of the two targets are correlated. Its 95% bootstrap percentile interval, from 2000 resamples of the cars, runs from -1270.1 to -818.2 (in miles per gallon times pounds), so the interval excludes zero. The interval describes which cars were drawn, not the process that produced them beyond that. This is the structure the matrix Σ in the definition allows for.

Theorem 5.4 (The Frobenius objective is a sum over columns). *For every $B \in \mathbb{R}^{p^* \times C}$, the objective of Definition 5.4 equals the sum of the squared-error objectives of the C single-target regressions:*

$$\|Y - DB\|_F^2 = \sum_{k=1}^C \|Y_{*,k} - DB_{*,k}\|^2.$$

i Proof

Proof. The squared Frobenius norm of a matrix is the sum of the squares of its entries. Sum them column by column. Column k of $Y - DB$ is $Y_{*,k} - DB_{*,k}$, and the sum of squares of its entries is $\|Y_{*,k} - DB_{*,k}\|^2$. Adding over $k = 1, \dots, C$ gives the right side. $\square$

Example 5.7 (Two targets from horsepower). In the Auto data [3], predict the two targets `mpg` and `acceleration` from an intercept and `horsepower`. For the least squares $\hat{B}$ and for a second matrix B that shifts every entry of $\hat{B}$ by the same amount, compute the squared Frobenius norm of the residual matrix and the sum of the squared norms of its columns:

5.2.3 Python

```

import numpy as np
import pandas as pd

auto_mv = pd.read_csv("data/auto.csv")
Phi_mv = np.column_stack([np.ones(len(auto_mv)), auto_mv["horsepower"].to_numpy(float)])
Y_mv = auto_mv[["mpg", "acceleration"]].to_numpy(float)
B_hat_mv = np.linalg.lstsq(Phi_mv, Y_mv, rcond=None)[0]
B_other_mv = B_hat_mv + 0.5

def both_sides(B):
    resid = Y_mv - Phi_mv @ B
    return np.linalg.norm(resid, "fro") ** 2, sum(np.sum(resid[:, k] ** 2) for k in range(resid.shape[1]))

sides_hat = both_sides(B_hat_mv)
sides_other = both_sides(B_other_mv)
print(sides_hat, sides_other)

(np.float64(10948.360306521437), np.float64(10948.360306521441)) (np.float64(2480858.3603065214), np.float64(2480858.3603065214))

```

5.2.4 R

```

auto_mv <- read.csv("data/auto.csv")
Phi_mv <- cbind(1, auto_mv$horsepower)
Y_mv <- as.matrix(auto_mv[, c("mpg", "acceleration")])
B_hat_mv <- solve(crossprod(Phi_mv), crossprod(Phi_mv, Y_mv))
B_other_mv <- B_hat_mv + 0.5

both_sides <- function(B) {
  resid <- Y_mv - Phi_mv %*% B
  c(frobenius = norm(resid, "F")^2, columns = sum(apply(resid, 2, function(col) sum(col^2))))
}

rbind(fitted = both_sides(B_hat_mv), other = both_sides(B_other_mv))

      frobenius    columns
fitted  10948.36   10948.36
other  2480858.36 2480858.36

```

For the fitted matrix, both sides equal 10,948.4. For the shifted matrix, they equal 2,480,858 and 2,480,858. The identity holds for every B , not only the minimizer. The fitted matrix has the smaller residual, as the minimizer must.

Theorem 5.5 (Multivariate normal equations). *The least-squares estimator for multivariate linear regression is given by:*

$$\hat{B} = (D^\top D)^{-1} D^\top Y$$

i Proof

Proof. By Theorem 5.4,

$$\|Y - DB\|_F^2 = \sum_{k=1}^C \|Y_{*,k} - DB_{*,k}\|^2,$$

so each column $B_{*,k}$ of B can be optimized independently. By Theorem 3.3, the minimizer for each column is $\hat{B}_{*,k} = (D^\top D)^{-1} D^\top Y_{*,k}$. Concatenating all C columns yields $\hat{B} = (D^\top D)^{-1} D^\top Y$. $\square$

Example 5.8 (One formula, three separate fits). In the Auto data [3], predict mpg, acceleration and weight from an intercept, horsepower and displacement. Compute $\hat{B} = (D^T D)^{-1} D^T Y$ of Theorem 5.5 in one step, and compare each of its columns with a separate least-squares fit of that target:

5.2.5 Python

```
import numpy as np
import pandas as pd

auto_mn = pd.read_csv("data/auto.csv")
Y_mn = auto_mn[["mpg", "acceleration", "weight"]].to_numpy(float)
Phi_mn = np.column_stack([np.ones(len(Y_mn)), auto_mn["horsepower"], auto_mn["displacement"]])

# All three targets at once.
B_mn = np.linalg.solve(Phi_mn.T @ Phi_mn, Phi_mn.T @ Y_mn)

# One univariate fit per target column.
B_cols_mn = np.column_stack([
    np.linalg.lstsq(Phi_mn, Y_mn[:, k], rcond=None)[0] for k in range(Y_mn.shape[1])
])
print(np.array2string(B_mn, precision=4, suppress_small=True))

[[ 37.4695  21.3115 1380.2336]
 [ -0.0583  -0.074   3.1021]
 [ -0.0408   0.0101  6.5494]]

print(np.allclose(B_mn, B_cols_mn))

True
```

5.2.6 R

```
auto_mn <- read.csv("data/auto.csv")
Y_mn <- as.matrix(auto_mn[, c("mpg", "acceleration", "weight")])
Phi_mn <- cbind(1, auto_mn$horsepower, auto_mn$displacement)

# All three targets at once.
B_mn <- solve(crossprod(Phi_mn), crossprod(Phi_mn, Y_mn))

# One univariate fit per target column.
B_cols_mn <- sapply(colnames(Y_mn), function(k) coef(lm(Y_mn[, k] ~ Phi_mn[, 2] + Phi_mn[, 3])))
round(B_mn, 4)

      mpg acceleration    weight
[1,] 37.4695      21.3115 1380.2336
[2,] -0.0583     -0.0740   3.1021
[3,] -0.0408      0.0101   6.5494

isTRUE(all.equal(unname(B_mn), unname(B_cols_mn)))

[1] TRUE
```

$\hat{B}$ has one column per target, and each column matches the fit that ignores the other targets. The multivariate problem decouples this way because the squared Frobenius norm is a sum over the columns of the residual matrix.

i Watch

Hutchinson teaches the multivariate case^a (11 min) [22].

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#linear_regression_pt2

6 Measuring error and non-linear extensions

6.1 Measuring error

i Remark (Three steps of linear regression)

Remark 6.1 (Three steps of linear regression). Fitting a linear model consists of three choices:

1. **Model family:** $f(\tilde{x}) = \tilde{x} \cdot \vec{\beta}$, assuming an affine relationship between inputs and output.
2. **Loss function:** $L(y, \hat{y}) = \varepsilon_{\text{pred}}^2$, penalizing the prediction error^a $\varepsilon_{\text{pred}} = \hat{y} - y$ by its square.
3. **Optimization:** $\hat{\vec{\beta}} = \arg \min_{\vec{\beta}} \sum_{i=1}^n (f(\tilde{x}_i) - y_i)^2$, minimizing empirical risk.

A fitted model raises questions about generalization to new data, comparison with alternative model families, training fit (measured by sMSE and residual standard error), and predictor interpretation.

^a<https://morrison-lab.github.io/sds/prediction.html#def-prediction-error>

Definition 6.1 (Sample mean squared error (sMSE)). The **sample mean squared error (sMSE)**^a of a model f on data $(\tilde{x}_1, y_1), \dots, (\tilde{x}_n, y_n)$ is the average of the squared residuals:

$$\text{sMSE} = \frac{1}{n} \sum_{i=1}^n (y_i - f(\tilde{x}_i))^2$$

^a<https://morrison-lab.github.io/sds/prediction.html#def-prediction-mse>

Example 6.1 (sMSE of a straight-line fit). The Auto data [3] give each car's fuel efficiency (mpg) and horsepower. Fit mpg on horsepower by least squares, and average the squared residuals:

6.1.1 Python

```
import numpy as np
import pandas as pd

auto_metrics = pd.read_csv("data/auto.csv")
hp = auto_metrics["horsepower"].to_numpy(float)
mpg = auto_metrics["mpg"].to_numpy(float)
Phi_metrics = np.column_stack([np.ones(len(hp)), hp])
beta_metrics = np.linalg.lstsq(Phi_metrics, mpg, rcond=None)[0]
residuals_metrics = mpg - Phi_metrics @ beta_metrics
mse_metrics = np.mean(residuals_metrics ** 2)
print(mse_metrics)

23.943662938603104
```

6.1.2 R

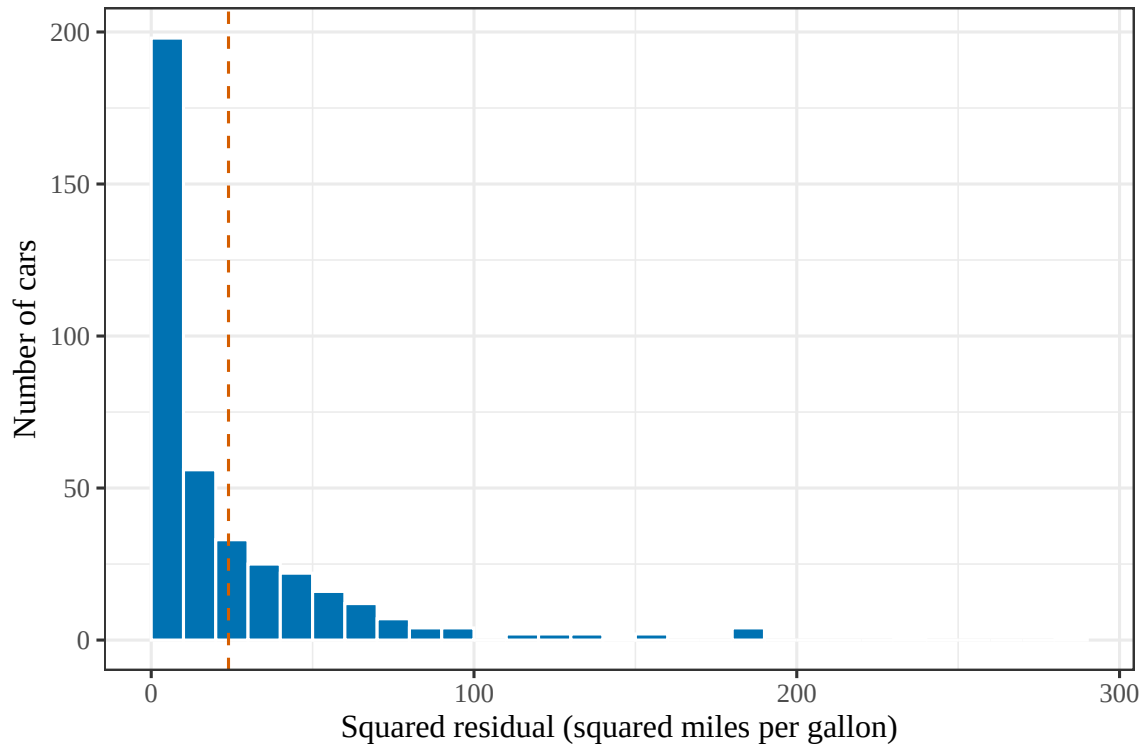
```
auto_metrics <- read.csv("data/auto.csv")
fit_metrics <- lm(mpg ~ horsepower, data = auto_metrics)
mse_metrics <- mean(residuals(fit_metrics)^2)
mse_metrics
```

```
[1] 23.94366
```

The sMSE is 23.94. Its unit is squared miles per gallon, which is hard to picture. Because sMSE divides by n , we can compare it across data sets of different sizes. This sMSE is a training error, since we fitted the line to these same cars. It gets no confidence interval, because the fit adapts to the cars it is scored on: the residuals are not independent draws, and the training error is biased low.

6.1.3 R

```
library(ggplot2)
msef_sq <- data.frame(squared = residuals(fit_metrics)^2)
ggplot(msef_sq, aes(squared)) +
  geom_histogram(binwidth = 10, boundary = 0, fill = "#0072B2", colour = "white") +
  geom_vline(xintercept = mse_metrics, colour = "#D55E00", linetype = "dashed") +
  labs(x = "Squared residual (squared miles per gallon)", y = "Number of cars")
```

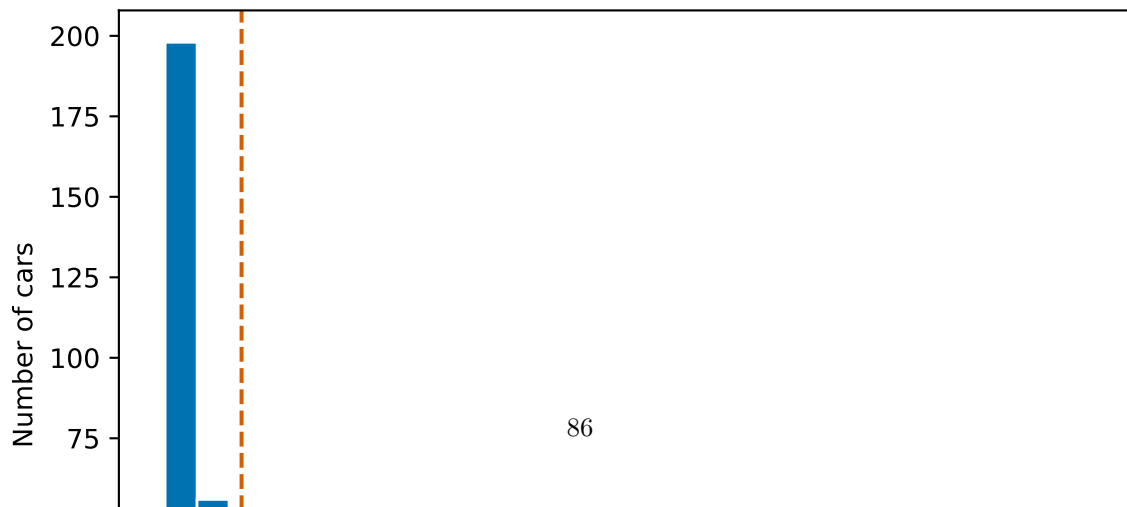


(a)

6.1.4 Python

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.hist(residuals_metrics ** 2, bins=np.arange(0, 300, 10), color="#0072B2", edgecolor="white");
ax.axvline(mse_metrics, color="#D55E00", linestyle="--");
ax.set_xlabel("Squared residual (squared miles per gallon)");
ax.set_ylabel("Number of cars");
plt.show()
```



Definition 6.2 (Sample root mean squared error (sRMSE)). The **sample root mean squared error (sRMSE)**, introduced in Course Overview, of a model f is the square root of the sMSE (Definition 6.1):

$$\begin{aligned}\text{sRMSE} &= \sqrt{\text{sMSE}} \\ &= \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - f(\tilde{x}_i))^2}\end{aligned}$$

Example 6.2 (sRMSE in miles per gallon). Take the square root of the sMSE from Example 6.1:

6.1.5 Python

```
rmse_metrics = np.sqrt(mse_metrics)
print(rmse_metrics)
```

4.893226230065713

6.1.6 R

```
rmse_metrics <- sqrt(mse_metrics)
rmse_metrics
```

[1] 4.893226

The sRMSE is 4.89 miles per gallon, in the same unit as `mpg` itself. A typical prediction from the straight line misses a car's fuel efficiency by roughly 5 miles per gallon. Like the sMSE it comes from, this sRMSE is a training error, so it also gets no confidence interval.

i Remark (Why report sRMSE instead of sMSE)

Remark 6.2 (Why report sRMSE instead of sMSE). Reports of prediction error usually give the sRMSE (Definition 6.2) rather than the sMSE (Definition 6.1), for three reasons.

1. **Units.** Squaring the residuals squares their unit, so the sMSE of a rent prediction is in dollars squared. The square root returns the sRMSE to the unit of the outcome, as Example 6.2 shows for miles per gallon.
2. **Interpretation.** Because the sRMSE is in the outcome's unit, a reader can compare it with a typical outcome value or with the outcome's spread. It is roughly the size of a typical residual, though not exactly, since squaring weights large residuals more heavily.
3. **No loss of ranking.** The square root is increasing, so the sRMSE ranks models in the same order as the sMSE. Taking the root does not change the choice among models.

The sMSE remains the right quantity for the math. We minimize it (or the sum of squared errors) when fitting, because it is a smooth function of the coefficients. It also splits into a sum of squared bias and variance, which the sRMSE does not. The sRMSE is the form we report, in figures and tables.

Definition 6.3 (Residual standard error (RSE, standard error of the regression)). The **residual standard error (RSE)**^a (also called the **standard error of the regression**) divides the sum of squared residuals by the residual degrees of freedom $n - p^* = n - p - 1$ instead of by n :

$$\begin{aligned}\text{RSE} &= \sqrt{\frac{1}{n - p^*} \sum_{i=1}^n (y_i - f(\tilde{x}_i))^2} \\ &= \sqrt{\frac{1}{n - p - 1} \sum_{i=1}^n (y_i - f(\tilde{x}_i))^2}\end{aligned}$$

Here p is the number of predictors, and $p^* = p + 1$ is the number of estimated parameters, including the intercept.

Example 6.3 (RSE of the straight-line fit). The fit in Example 6.1 has $p = 1$ predictor, so $p^* = 2$. Compute the RSE, and a 95% confidence interval for the noise standard deviation σ that it estimates:

6.1.7 Python

```
from scipy import stats

n_metrics = len(mpg)
rss_metrics = np.sum(residuals_metrics ** 2)
rse_metrics = np.sqrt(rss_metrics / (n_metrics - 2))
# With Gaussian errors, RSS / sigma^2 follows a chi-squared distribution
# with n - 2 degrees of freedom; inverting that statement bounds sigma.
rse_ci = np.sqrt(rss_metrics / stats.chi2.ppf([0.975, 0.025], df=n_metrics - 2))
print(rse_metrics, rse_ci)

4.905756919545939 [4.58432256 5.27603679]
```

6.1.8 R

```
rss_metrics <- sum(residuals(fit_metrics)^2)
rse_metrics <- sqrt(rss_metrics / df.residual(fit_metrics))
# With Gaussian errors, RSS / sigma^2 follows a chi-squared distribution
# with n - 2 degrees of freedom; inverting that statement bounds sigma.
chi_sq <- qchisq(c(0.975, 0.025), df = df.residual(fit_metrics))
rse_ci <- sqrt(rss_metrics / chi_sq)
c(rse_metrics, rse_ci)

[1] 4.905757 4.584323 5.276037

performance::model_performance(fit_metrics) |>
  insight::print_md(digits = 3)
```

Table 1: Indices of model performance

AIC	AICc	BIC	R2	R2 (adj.)	RMSE	Sigma
2363.3	2363.4	2375.2	0.606	0.605	4.893	4.906

The RSE is 4.906, slightly larger than the sRMSE of 4.893 from Example 6.2, because it divides by $n - 2 = 390$ instead of $n = 392$. In R, the performance package reports the training RMSE (here the sRMSE) in a column named **RMSE** and the RSE as **Sigma**. The 95% confidence interval for σ runs from 4.58 to 5.28 miles per gallon. It inverts the chi-squared distribution of RSS/σ^2 under Gaussian errors, and it is more sensitive to non-Gaussian errors than the t intervals for the coefficients. Unlike the training sRMSE, the RSE gets an interval, because it estimates the noise level σ of the model and not the error on the cars used to fit.

i Remark (Why RSE can rise when a predictor is added)

Remark 6.3 (Why RSE can rise when a predictor is added). Adding a predictor never increases the training sum of squared residuals. It does shrink the denominator $n - p^*$ of the RSE (Definition 6.3). So the RSE goes up when a new predictor reduces the sum of squares too little to make up for the lost degree of freedom.

i Read

GeeksforGeeks explains regression metrics^a [30], including MSE and RMSE (its names for the sMSE and sRMSE).

^a<https://www.geeksforgeeks.org/machine-learning/regression-metrics/>

6.2 Error decomposition: reducible error and residual variance

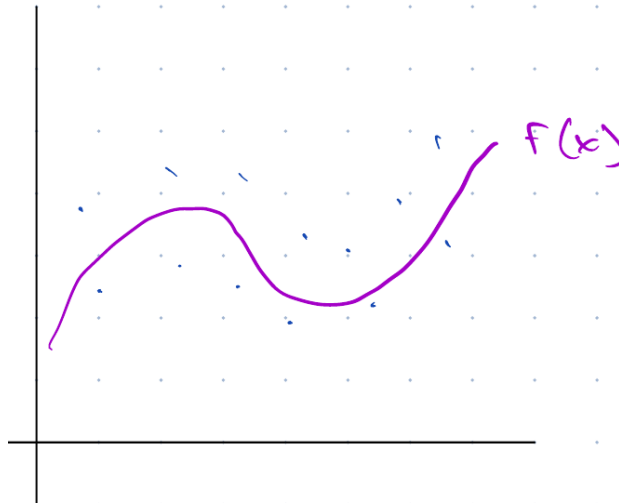


Figure 16: The true data-generating relationship $y = \mu(\tilde{x}) + \varepsilon$ with random noise ε versus the fitted model approximation $f(\tilde{x})$.

Proposition 6.1 (Reducible error and residual variance decomposition). Fix an input^a value x and a fitted model f . Let $\hat{y}$ be the prediction of f at $\tilde{x}$. Let $\varepsilon = Y - \mu(\tilde{x})$ be the noise, the deviation^b of the outcome^c from the true mean function at $\tilde{x}$, so that

$$Y = \mu(\tilde{x}) + \varepsilon$$

The noise has zero mean ($\mathbb{E}[\varepsilon] = 0$), because the true mean function is the population mean^d of the outcome at $\tilde{x}$. Let $\sigma^2 = \text{Var}(\varepsilon)$ be its variance^e at $\tilde{x}$.

The expected squared prediction error^f of $\hat{y}$ decomposes into:

$$\mathbb{E}[(\hat{y} - Y)^2] = \underbrace{(\varepsilon(\hat{y}))^2}_{\text{Reducible error}} + \underbrace{\sigma^2}_{\text{Residual variance}}$$

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/pds/variance-covariance.html#def-deviation>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^d<https://morrison-lab.github.io/pds/expectation.html#def-expectation>

^e<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

^f<https://morrison-lab.github.io/sds/model-validation.html#def-epe>

i Proof

Proof. Substitute $Y = \mu(\tilde{x}) + \varepsilon$. Let $\varepsilon(\hat{y}) = \hat{y} - \mu(\tilde{x})$ be the error of the prediction $\hat{y}$ at $\tilde{x}$. It has a

single value, because the point $\underline{x}$ and the fitted model f are fixed:

$$\begin{aligned}\mathbb{E}[(\hat{y} - Y)^2] &= \mathbb{E}[(\varepsilon(\hat{y}) - \varepsilon)^2] \\ &= \mathbb{E}[(\varepsilon(\hat{y}))^2 - 2\varepsilon\varepsilon(\hat{y}) + (\varepsilon)^2] \\ &= (\varepsilon(\hat{y}))^2 - 2 \underbrace{\mathbb{E}[\varepsilon]}_0 \varepsilon(\hat{y}) + \mathbb{E}[(\varepsilon)^2] \\ &= (\varepsilon(\hat{y}))^2 + \text{Var}(\varepsilon)\end{aligned}$$

The reducible error $(\varepsilon(\hat{y}))^2$ depends on the model family^a and the parameter estimates. The residual variance σ^2 is an inherent property of the data-generating process and the chosen inputs, and provides a theoretical lower bound on the expected prediction error^b of models that use those inputs. $\square$

^a<https://morrison-lab.github.io/sds/model-validation.html#def-model-family>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-epe>

Example 6.4 (Reducible error and residual variance on the synthetic data). For [the synthetic data](#) we know the truth, $\mu(\underline{x}) = 2 + 3x_1 - x_2$, and the noise standard deviation^a, 0.1, so $\sigma^2 = 0.01$. Fit the linear model by least squares, and at the input^b $\underline{x}_0 = (0.5, -1)$ compute the reducible error $(f(\underline{x}_0) - \mu(\underline{x}_0))^2$. Then average the squared prediction error over the 50 noise draws that generated the data, as if each had produced a new response at $\underline{x}_0$:

6.2.1 Python

```
import numpy as np

syn_ed = r.synthetic
Phi_ed = np.column_stack([np.ones(len(syn_ed)), syn_ed["x1"], syn_ed["x2"]])
beta_ed = np.linalg.lstsq(Phi_ed, syn_ed["y"].to_numpy(), rcond=None)[0]

x0_ed = np.array([1.0, 0.5, -1.0])          # intercept, x1, x2
f_star_ed = 2 + 3 * 0.5 - (-1.0)           # the true mean at x0
f_hat_ed = x0_ed @ beta_ed                 # the fitted prediction at x0
reducible_ed = (f_hat_ed - f_star_ed) ** 2

# y at x0 would be mu(x0) plus a noise draw; reuse the 50 draws that generated the data.
noise_ed = np.asarray(r.noise_syn)
mean_sq_err_ed = np.mean((f_hat_ed - (f_star_ed + noise_ed)) ** 2)
print(reducible_ed, np.mean(noise_ed ** 2), mean_sq_err_ed)

2.3498217295359075e-06 0.01002980638696177 0.010055722991748768

from scipy import stats

# 95% confidence interval for the fitted prediction at x0 (the mean-response interval).
df_ed = len(syn_ed) - 3
resid_ed = syn_ed["y"].to_numpy() - Phi_ed @ beta_ed
sigma_ed = np.sqrt(np.sum(resid_ed ** 2) / df_ed)
h0_ed = x0_ed @ np.linalg.inv(Phi_ed.T @ Phi_ed) @ x0_ed
f_hat_ci_ed = f_hat_ed + np.array([-1, 1]) * stats.t.ppf(0.975, df_ed) * sigma_ed * np.sqrt(h0_ed)
# The noise has mean zero by construction, so n * (mean square) / sigma^2 is chi-squared
# with n degrees of freedom; inverting that gives an interval for the noise variance.
n_noise_ed = len(noise_ed)
noise_var_ci_ed = (n_noise_ed * np.mean(noise_ed ** 2)
                   / stats.chi2.ppf([0.975, 0.025], df=n_noise_ed))
print(f_hat_ci_ed, noise_var_ci_ed)
```

```
[4.45345901 4.54347516] [0.00702169 0.01549849]
```

6.2.2 R

```
fit_ed <- lm(y ~ x1 + x2, data = synthetic)
f_star_ed <- 2 + 3 * 0.5 - (-1) # the true mean at x0 = (0.5, -1)
f_hat_ed <- unname(predict(fit_ed, data.frame(x1 = 0.5, x2 = -1)))
reducible_ed <- (f_hat_ed - f_star_ed)^2

# y at x0 would be mu(x0) plus a noise draw; reuse the 50 draws that generated the data.
c(reducible = reducible_ed, noise_variance = mean(noise_syn^2),
  mean_squared_error = mean((f_hat_ed - (f_star_ed + noise_syn))^2))

      reducible      noise_variance mean_squared_error
2.349822e-06      1.002981e-02      1.005572e-02

# 95% confidence interval for the fitted prediction at x0
# (the mean-response interval).
f_hat_ci_ed <- predict(fit_ed, data.frame(x1 = 0.5, x2 = -1),
                      interval = "confidence")
f_hat_ci_ed

      fit      lwr      upr
1 4.498467 4.453459 4.543475

# The noise has mean zero by construction,
# so n * (mean square) / sigma^2 is chi-squared with n degrees of freedom;
# inverting that gives an interval for the noise variance.
noise_var_ci_ed <- length(noise_syn) * mean(noise_syn^2) /
  qchisq(c(0.975, 0.025), df = length(noise_syn))
noise_var_ci_ed

[1] 0.007021688 0.015498491
```

The reducible error is $2.3\text{e-}06$: with 50 observations the fit is already close to the truth at $\tilde{x}_0$. The residual variance is the noise variance^c, 0.0100 in these draws against the true 0.01. Their sum, 0.0100, is close to the average squared error, 0.0101. The 95% confidence interval for the noise variance, from the chi-squared distribution of a Gaussian sample's mean square, runs from 0.0070 to 0.0155, and it contains the true 0.01. The fitted prediction at $\tilde{x}_0$ is 4.498, with a 95% confidence interval from 4.453 to 4.543, which contains the true mean 4.5. The average squared error reuses the noise draws that generated the fitted data, so it gets no interval. No better model can push the error below the noise variance. That is why the second term is called irreducible, given these inputs.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-sd>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^c<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

6.3 Non-linear regression by feature expansion

Definition 6.4 (Non-linear regression by feature expansion). A non-linear relationship can be modelled using linear regression by defining a fixed non-linear feature map $\tilde{D} : \mathbb{R}^p \rightarrow \mathbb{R}^{p^*}$:

$$\begin{aligned} h(\tilde{x}) &= \tilde{\beta} \cdot \tilde{D}(\tilde{x}) \\ &= \tilde{D}(\tilde{x}) \cdot \tilde{\beta} \end{aligned}$$

Stacking transformed inputs into the design matrix $D \in \mathbb{R}^{n \times p^*}$:

$$D = \begin{bmatrix} \text{---} & D(\tilde{x}_1)^\top & \text{---} \\ \text{---} & D(\tilde{x}_2)^\top & \text{---} \\ & \vdots & \\ \text{---} & D(\tilde{x}_n)^\top & \text{---} \end{bmatrix} \in \mathbb{R}^{n \times p^*}$$

Example 6.5 (What the feature map does to a table). In the Auto data [3], one raw feature is horsepower, so $p = 1$. Take the [feature map](#) $D(x) = [1, x, x^2]^\top$ of Definition 6.4, which has $p^* = 3$ entries. Apply it to the first four cars, and compare the straight-line and quadratic fits of mpg on the whole data:

6.3.1 Python

```
import numpy as np
import pandas as pd

auto_fx = pd.read_csv("data/auto.csv")
hp_fx = auto_fx["horsepower"].to_numpy(float)
mpg_fx = auto_fx["mpg"].to_numpy(float)

# Each row of Phi is the feature map applied to one car.
Phi_quad_fx = np.column_stack([np.ones(len(hp_fx)), hp_fx, hp_fx ** 2])
Phi_line_fx = Phi_quad_fx[:, :2]
print(Phi_quad_fx[:4])

[[1.0000e+00 1.3000e+02 1.6900e+04]
 [1.0000e+00 1.6500e+02 2.7225e+04]
 [1.0000e+00 1.5000e+02 2.2500e+04]
 [1.0000e+00 1.5000e+02 2.2500e+04]]

def rmse_fx(Phi):
    beta = np.linalg.lstsq(Phi, mpg_fx, rcond=None)[0]
    return np.sqrt(np.mean((mpg_fx - Phi @ beta) ** 2))

rmse_line_fx, rmse_quad_fx = rmse_fx(Phi_line_fx), rmse_fx(Phi_quad_fx)
print(rmse_line_fx, rmse_quad_fx)

4.893226230065713 4.3571514671419465
```

6.3.2 R

```
auto_fx <- read.csv("data/auto.csv")
hp_fx <- auto_fx$horsepower

# Each row of Phi is the feature map applied to one car.
Phi_quad_fx <- unname(cbind(1, hp_fx, hp_fx^2))
head(Phi_quad_fx, 4)

      [,1] [,2] [,3]
[1,]    1  130 16900
[2,]    1  165 27225
[3,]    1  150 22500
[4,]    1  150 22500

rmse_fx <- function(Phi) sqrt(mean(residuals(lm(auto_fx$mpg ~ Phi - 1))^2))
c(line = rmse_fx(Phi_quad_fx[, 1:2]), quadratic = rmse_fx(Phi_quad_fx))

line quadratic
```

4.893226 4.357151

One raw feature became three columns, so the model is still linear in $\vec{\beta}$ but fits a curve in horsepower. The extra column lowers the training sRMSE from 4.89 to 4.36 miles per gallon (why we report sRMSE), because the relationship between horsepower and mpg bends. These are training errors from fits to the same cars, so they get no confidence intervals: the residuals are not independent draws, and the errors are biased low, more so for the more flexible quadratic model.

Definition 6.5 (Linear in the parameters (linear functional form)). A model $h(x)$ is **linear in the parameters** (it has a **linear functional form**) if it can be written as the **inner product** of the parameter vector $\vec{\beta}$ with a fixed **feature map** $\tilde{D}(x)$ that does not depend on $\vec{\beta}$, as in Definition 6.4:

$$h(x) = \vec{\beta} \cdot \tilde{D}(x).$$

The model can be nonlinear in the inputs x , but each parameter enters only as a multiplier of a feature.

Example 6.6 (A quadratic is linear in the parameters; an exponential decay is not). Let x be a single input.

- The quadratic $h(x) = \beta_0 + \beta_1 x + \beta_2 x^2$ is nonlinear in x . It is **linear in the parameters**, because $\tilde{D}(x) = [1, x, x^2]^\top$ is fixed and $h(x) = \vec{\beta} \cdot \tilde{D}(x)$.
- The exponential decay $h(x) = \beta_0 \exp\{-\beta_1 x\}$ is not linear in the parameters, because β_1 sits inside the exponential, so no fixed $\tilde{D}$ writes it as $\vec{\beta} \cdot \tilde{D}(x)$.

Theorem 6.1 (Normal equations for expanded features). Let $D \in \mathbb{R}^{n \times p^*}$ be the design matrix of Definition 6.4. If $D^\top D$ is invertible, the least squares coefficients of the expanded model are

$$\hat{\vec{\beta}} = (D^\top D)^{-1} D^\top y.$$

i Proof

Proof. The model $h(x) = \vec{\beta} \cdot \tilde{D}(x)$ is linear in $\vec{\beta} \in \mathbb{R}^{p^*}$, and row i of D is $\tilde{D}(x_i)$. The fitted values are therefore $D\hat{\vec{\beta}}$, as in Definition 2.5, and the objective depends on the data only through D and y . Theorem 3.3 uses nothing else about D , so its closed-form solution^a holds with this D . $\square$

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-closed-form-solution>

Example 6.7 (A quadratic in horsepower). In the Auto data [3], let $\tilde{D}(x) = [1, x, x^2]^\top$ for horsepower x . Solve the normal equations of Theorem 6.1 for mpg, and compare with the coefficients from a library fit of the same quadratic:

6.3.3 Python

```
import numpy as np
import pandas as pd

auto_fe = pd.read_csv("data/auto.csv")
hp_fe = auto_fe["horsepower"].to_numpy(float)
mpg_fe = auto_fe["mpg"].to_numpy(float)
Phi_fe = np.column_stack([np.ones(len(hp_fe)), hp_fe, hp_fe ** 2])
beta_fe = np.linalg.solve(Phi_fe.T @ Phi_fe, Phi_fe.T @ mpg_fe)
beta_fe_lib = np.linalg.lstsq(Phi_fe, mpg_fe, rcond=None)[0]
print(beta_fe)
```

```
[ 5.69000997e+01 -4.66189630e-01  1.23053610e-03]
```

```
print(np.allclose(beta_fe, beta_fe_lib))
```

True

6.3.4 R

```
auto_fe <- read.csv("data/auto.csv")
Phi_fe <- cbind(1, auto_fe$horsepower, auto_fe$horsepower^2)
beta_fe <- solve(crossprod(Phi_fe), crossprod(Phi_fe, auto_fe$mpg))
fit_fe <- lm(mpg ~ horsepower + I(horsepower^2), data = auto_fe)
print(drop(beta_fe))
```

```
[1] 56.900099702 -0.466189630  0.001230536
```

```
all.equal(drop(beta_fe), unname(coef(fit_fe)))
```

```
[1] TRUE
```

The intercept, linear coefficient and quadratic coefficient are 56.9, -0.466 and 0.00123. The quadratic coefficient is positive, so the fitted curve is convex: it falls steeply at low horsepower and flattens at high horsepower, the pattern in the scatter plot of Solution 6.5.4. The closed-form solution^a and the library agree.

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-closed-form-solution>

Theorem 6.2 (The number of polynomial features). *The degree- d polynomial expansion of p raw inputs $x_1, \dots, x_p$ has one feature for every monomial*

$$x_1^{a_1} x_2^{a_2} \cdots x_p^{a_p}$$

with integer exponents $a_j \geq 0$ and $a_1 + \cdots + a_p \leq d$. This includes the constant 1 and every cross-term. The number of features is

$$p^* = \binom{p+d}{d}.$$

i Proof

Proof. Add a slack exponent $a_0 = d - (a_1 + \cdots + a_p) \geq 0$. Monomials correspond one-to-one to the tuples $(a_0, a_1, \dots, a_p)$ of nonnegative integers with $a_0 + a_1 + \cdots + a_p = d$.

Count these tuples with stars and bars. Write d stars in a row and place p bars among them. The bars split the stars into $p+1$ groups, and the group sizes are $a_0, a_1, \dots, a_p$. Every tuple arises from exactly one arrangement. An arrangement is a row of $d+p$ symbols in which we choose which d are stars, so there are $\binom{p+d}{d}$ of them. $\square$

Example 6.8 (Counting monomials of the Auto inputs). Expand the three Auto inputs **horsepower**, **weight** and **displacement** to degree three, so p is the number of inputs and d is the degree. List every exponent tuple $(a_1, \dots, a_p)$ with $a_1 + \cdots + a_p \leq d$, count the tuples, and compare with the formula of Theorem 6.2:

6.3.5 Python

```

from itertools import product
from math import comb

p_fc, d_fc = 3, 3
tuples_fc = [a for a in product(range(d_fc + 1), repeat=p_fc) if sum(a) <= d_fc]
print(len(tuples_fc), comb(p_fc + d_fc, d_fc))

20 20

```

6.3.6 R

```

p_fc <- 3
d_fc <- 3
grid_fc <- do.call(expand.grid, rep(list(0:d_fc), p_fc))
c(listed = sum(rowSums(grid_fc) <= d_fc), formula = choose(p_fc + d_fc, d_fc))

  listed formula
      20      20

```

Both counts are 20, the number of columns that the design matrix of this expansion has, including the intercept column. The raw count of $(d + 1)^p = 64$ tuples shrinks to this number once the degree constraint is imposed.

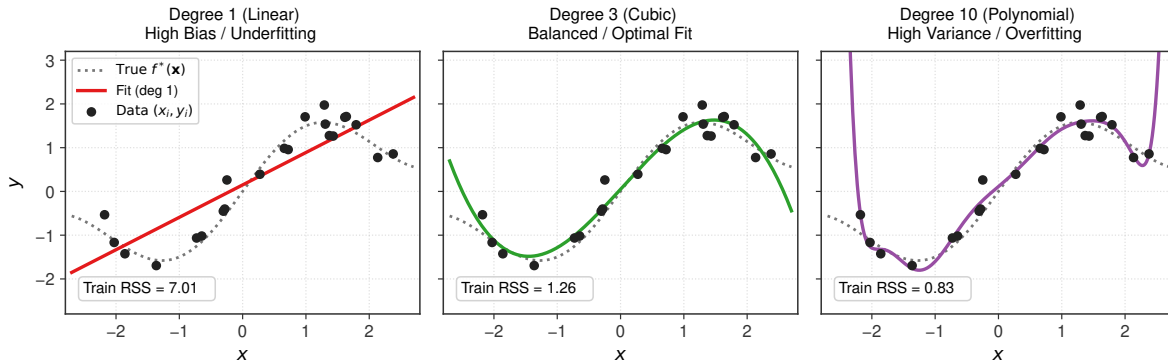


Figure 17: Polynomial feature expansion on non-linear data comparing underfitting (degree 1), balanced fit (degree 3), and overfitting (degree 10).

Definition 6.6 (The Fourier basis expansion). For periodic or smoothly oscillating scalar inputs $x \in \mathbb{R}$ with period T , the degree- k **Fourier basis expansion** maps x into sinusoids of increasing harmonic frequencies:

$$\tilde{D}(x) = \left[1, \cos\left(\frac{2\pi x}{T}\right), \sin\left(\frac{2\pi x}{T}\right), \dots, \cos\left(\frac{2\pi kx}{T}\right), \sin\left(\frac{2\pi kx}{T}\right) \right]^\top \in \mathbb{R}^{2k+1}$$

Unlike high-degree polynomials, which tend to oscillate wildly between training points (Runge's phenomenon) and produce severely **ill-conditioned** matrices, the Fourier basis provides a well-conditioned orthogonal representation for periodic signals.

Example 6.9 (Hours of the day). Take x to be the hour of the day, $0, 1, \dots, 23$, a periodic input with period $T = 24$. Build the degree- $k = 2$ Fourier basis expansion of Definition 6.6, which has $2k + 1 = 5$ columns, and the degree-4 polynomial expansion, which also has 5. Compare the two by computing $D^\top D$ and its **condition number**:

6.3.7 Python

```
import numpy as np

hours_fb = np.arange(24)
T_fb, k_fb = 24, 2

# Columns: 1, then a cosine and a sine for each harmonic 1..k.
cols_fb = [np.ones(24)]
for j in range(1, k_fb + 1):
    cols_fb += [np.cos(2 * np.pi * j * hours_fb / T_fb), np.sin(2 * np.pi * j * hours_fb / T_fb)]
Phi_fourier_fb = np.column_stack(cols_fb)
Phi_poly_fb = np.column_stack([hours_fb ** d for d in range(5)])

gram_fb = Phi_fourier_fb.T @ Phi_fourier_fb
# Adding 0.0 turns the "-0." entries that rounding leaves into plain zeros.
print(gram_fb.round(6) + 0.0)

[[24.  0.  0.  0.  0.]
 [ 0. 12.  0.  0.  0.]
 [ 0.  0. 12.  0.  0.]
 [ 0.  0.  0. 12.  0.]
 [ 0.  0.  0.  0. 12.]]

print(np.linalg.cond(gram_fb), np.linalg.cond(Phi_poly_fb.T @ Phi_poly_fb))
2.0000000000000004 206057799742.09592
```

6.3.8 R

```
hours_fb <- 0:23
T_fb <- 24
k_fb <- 2

# Columns: 1, then a cosine and a sine for each harmonic 1..k.
Phi_fourier_fb <- cbind(1, do.call(cbind, lapply(1:k_fb, function(j) {
  cbind(cos(2 * pi * j * hours_fb / T_fb), sin(2 * pi * j * hours_fb / T_fb))
})))
Phi_poly_fb <- outer(hours_fb, 0:4, `^`)

gram_fb <- crossprod(Phi_fourier_fb)
round(gram_fb, 6)

      [,1] [,2] [,3] [,4] [,5]
[1,]   24    0    0    0    0
[2,]    0   12    0    0    0
[3,]    0    0   12    0    0
[4,]    0    0    0   12    0
[5,]    0    0    0    0   12

c(fourier = kappa(gram_fb, exact = TRUE), polynomial = kappa(crossprod(Phi_poly_fb), exact = TRUE))

      fourier      polynomial
      2 206057799551
```

For equally spaced hours, the Fourier columns are orthogonal: $D^T D$ is diagonal, with 24 for the constant and 12 for each cosine and sine. Its **condition number** is 2.0, the ratio of those two numbers. The polynomial expansion of the same size has a condition number of about $2.1\text{e}+11$, so solving its normal equations would lose about 11 of the roughly 16 digits that double precision carries.

Exercise 6.1 (Count the features a polynomial expansion produces). The chapter states that a degree^a- d polynomial^b expansion on p raw inputs^c, with the constant term and every cross-term included, has $p^* = \binom{p+d}{d}$ features.

Write out every feature for $p = 2$ and $d = 4$, grouped by degree.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 6.1. With $p = 2$ inputs^a and degree^b $d \leq 4$:

Degree	Features
0	1
1	x_1, x_2
2	x_1^2, x_1x_2, x_2^2
3	$x_1^3, x_1^2x_2, x_1x_2^2, x_2^3$
4	$x_1^4, x_1^3x_2, x_1^2x_2^2, x_1x_2^3, x_2^4$

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Exercise 6.2 (Count the features a polynomial expansion produces (part 2)). Use the setup of Exercise 6.1. Count them.

Solution

Solution 6.2. The degrees^a 0 to 4 have 1, 2, 3, 4 and 5 features, which total 15.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Exercise 6.3 (Count the features a polynomial expansion produces (part 3)). Use the setup of Exercise 6.1. Check the count against the formula.

Solution

Solution 6.3.

$$\binom{p+d}{d} = \binom{6}{4} = 15.$$

Exercise 6.4 (Count the features a polynomial expansion produces (part 4)). Use the setup of Exercise 6.1.

Derive the formula for p^* . A feature is a monomial $x_1^{a_1}x_2^{a_2}\cdots x_p^{a_p}$ with every $a_j \geq 0$ and $a_1 + \cdots + a_p \leq d$. Turn that inequality into an equality by introducing one more variable, then count the solutions.

Solution

Solution 6.4. The constraint is $a_1 + \dots + a_p \leq d$ with every $a_j \geq 0$. Introduce a slack variable $a_0 \geq 0$ holding the unused degree^a, so that

$$a_0 + a_1 + \dots + a_p = d$$

Every feature gives exactly one such tuple and every tuple gives exactly one feature, so counting features is now counting non-negative integer solutions of one equation in $p + 1$ unknowns.

That is the stars-and-bars count. Lay out d indistinguishable stars in a row and choose p positions among them for bars; the bars cut the row into $p + 1$ groups, and the group sizes are $a_0, a_1, \dots, a_p$. The row holds $d + p$ symbols altogether and we choose which d of them are stars, so there are

$$\binom{p+d}{d}$$

tuples, hence that many features.

Sanity-check the bijection on a small case. With $p = 3$ and $d = 1$ there is one star and three bars, and the four arrangements

$$\star ||| \quad | \star || \quad || \star | \quad ||| \star$$

are the four monomials $1, x_1, x_2, x_3$ in that order.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Exercise 6.5 (Count the features a polynomial expansion produces (part 5)). Use the setup of Exercise 6.1.

Evaluate the count for $p = 100$ and $d = 3$.

Solution

Solution 6.5. $\binom{103}{3} = \frac{103 \cdot 102 \cdot 101}{6} = 176,851$.

A hundred inputs^a and a cubic is not an extravagant-sounding model, and it asks for more than a hundred and seventy thousand features.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 6.6 (Count the features a polynomial expansion produces (part 6)). Use the setup of Exercise 6.1.

Suppose you have $n = 5000$ training points and use that expansion. What does Exercise 3.31 say about $D^\top D$, and is it the fixable failure or the unfixable one?

Solution

Solution 6.6. D is $5000 \times 176,851$, so $n \ll p^*$ and $D^\top D$ is singular — the first of the two failures in Exercise 3.31, the one forced by shape rather than by the data.

It is the *fixable* one in that exercise's sense: more data would eventually cure it. But “eventually” implies more than 176,851 points, and the null space today has dimension at least $176,851 - 5000 = 171,851$. Reading a coefficient off one solution drawn from a space that size is meaningless, and the minimum-norm solution is exactly such a draw.

So the diagnosis and the remedy diverge here. Calling the failure fixable is true but unhelpful. The real options are a smaller degree^a d , a subset of the p inputs^b, or a penalty that makes the problem well-posed at the n you have.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

i Read

GeeksforGeeks explains polynomial regression^a [31].

^a<https://www.geeksforgeeks.org/machine-learning/polynomial-regression-for-non-linear-data-ml/>

6.4 Choosing how far to expand

i Remark (Choosing expansion degree: bias and variance)

Remark 6.4 (Choosing expansion degree: bias and variance). Choosing the complexity of ϕ (such as polynomial degree d , illustrated in Figure 17) is a hyperparameter decision:

- *Expanding too little (underfitting)*: high bias; μ is not a linear combination^a of the features, leaving large reducible error.
- *Expanding too much (overfitting)*: high variance; $\vec{\beta}$ fits the noise ε , giving low training error but high generalization error.
- *Residual variance*: $\text{Var}(\varepsilon) = \sigma^2$ stays the same whatever the expansion.

Choosing the degree requires held-out validation data or cross-validation; see Model Selection.

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-linear-combination>

i Remark (Numerical stability of high-degree expansions)

Remark 6.5 (Numerical stability of high-degree expansions).

- Powers of a feature ($x, x^2, x^3, \dots$) become strongly collinear over a bounded interval.
- $D^\top D$ becomes severely *ill-conditioned* as the degree grows.
- Factorizing D directly (by QR^a or SVD^b) is much more numerically stable than forming $D^\top D$.

^a<https://morrison-lab.github.io/mds/linear-algebra-projections.html#thm-qr>

^b<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

6.5 Lab: linear regression on the Auto data

The lab exercises in this section and the next are adapted from the lab that accompanies [32, Ch. 3], whose Python code is published as ISLP_labs². The book's lab uses the Boston and Carseats data and the `statsmodels` package; here the same steps use the Auto data. Each solution gives the code in Python with `statsmodels`, as the book's lab does; in Python with the `np.linalg.lstsq` of this chapter; in R with `tidymodels`; and in base R. The `tidymodels` code follows ISLR-tidymodels-labs³, the `tidymodels` version of the book's labs.

The labs below report 95% intervals. These helpers are defined once, here, so no lab depends on another lab's code. Each takes all its inputs as arguments.

```
import numpy as np

def bootstrap_corr_ci(a, b, B=20000, seed=571):
    """95% percentile interval for the correlation of a and b.

    Resamples the cases with replacement B times,
    which mimics drawing new cases from the same population,
```

²https://github.com/intro-stat-learning/ISLP_labs

³<https://github.com/Morrison-Lab/ISLR-tidymodels-labs>

```

    and recomputes the correlation each time.
    """
    rng = np.random.default_rng(seed)
    a, b = np.asarray(a, float), np.asarray(b, float)
    corr = np.empty(B)
    for i in range(B):
        rows = rng.integers(0, len(a), len(a))
        corr[i] = np.corrcoef(a[rows], b[rows])[0, 1]
    return np.quantile(corr, [0.025, 0.975])

def bootstrap_vif_ci(features, B=20000, seed=571):
    """95% percentile interval for each feature's VIF.

    The VIF of feature j is the jth diagonal entry
    of the inverse of the features' correlation matrix,
    which gives the same value as 1 / (1 - R^2) without a regression per feature.
    """
    rng = np.random.default_rng(seed)
    F = np.asarray(features, float)
    vifs = np.empty((B, F.shape[1]))
    for b in range(B):
        rows = rng.integers(0, len(F), len(F))
        vifs[b] = np.diag(np.linalg.inv(np.corrcoef(F[rows], rowvar=False)))
    return np.quantile(vifs, [0.025, 0.975], axis=0)

# Formats an interval's endpoints for the prose
ci_text <- function(bounds, digits) {
  sprintf("%.*f to %.*f", digits, bounds[[1]], digits, bounds[[2]])
}

# 95% percentile interval for R^2 from resampled cars.
# Resampling rows with replacement mimics
# drawing a new set of cars from the same population.
# The refits use lm.fit directly, because 2,000 workflow fits would be slow.
boot_r2_ci <- function(formula, data, n_boot = 2000) {
  set.seed(571)
  design <- model.matrix(formula, data)
  y <- model.response(model.frame(formula, data))
  r2 <- vapply(seq_len(n_boot), \ (b) {
    rows <- sample(nrow(design), replace = TRUE)
    sse <- sum(lm.fit(design[rows, ], y[rows])$residuals^2)
    1 - sse / sum((y[rows] - mean(y[rows]))^2)
  }, numeric(1))
  unname(quantile(r2, c(0.025, 0.975)))
}

# 95% percentile interval for the correlation of a and b
boot_cor_ci <- function(a, b, n_boot = 20000) {
  set.seed(571)
  corr <- vapply(seq_len(n_boot), \ (i) {
    rows <- sample(length(a), replace = TRUE)
    cor(a[rows], b[rows])
  }, numeric(1))
  unname(quantile(corr, c(0.025, 0.975)))
}

```

```

# 95% percentile interval for each feature's VIF.
# The VIF of feature j is the jth diagonal entry
# of the inverse of the features' correlation matrix,
# which gives the same value as 1 / (1 - R^2) without a regression per feature.
boot_vif_ci <- function(features, n_boot = 2000) {
  set.seed(571)
  vifs <- vapply(seq_len(n_boot), \ (b) {
    resample <- features[sample(nrow(features), replace = TRUE), ]
    diag(solve(cor(resample)))
  }, numeric(ncol(features)))
  apply(vifs, 1, quantile, probs = c(0.025, 0.975))
}

# The 95% interval for a linear combination w'beta
# of a fitted model's coefficients:
# the estimate plus or minus a t quantile times its standard error,
# from the coefficients' covariance matrix
lincomb_ci <- function(model, w) {
  estimate <- sum(w * coef(model))
  half <- qt(0.975, df.residual(model)) * sqrt(drop(t(w) %*% vcov(model) %*% w))
  c(estimate = estimate, lower = estimate - half, upper = estimate + half)
}

```

Exercise 6.7 (Lab: fit a line to the Auto data). This is the first of six short lab exercises on the Auto data. Work in pairs, at a computer. Use Python or R; each solution shows both. Download the Auto data from [data/auto.csv](https://morrison-lab.github.io/lds/chapters/data/auto.csv)^a, the file used in this chapter. Save it as `data/auto.csv` next to your code, because the solutions read it from that path. It has 392 cars, and comes from the book [32]; the Python package ISLP^b and the R package ISLR2^c also provide it. Predict a car's fuel economy `mpg` from its `horsepower`. Load the data and look at its first rows.

^a<https://morrison-lab.github.io/lds/chapters/data/auto.csv>

^b<https://islp.readthedocs.io/>

^c<https://cran.r-project.org/package=ISLR2>

Solution

Solution 6.7.

6.5.1 Python (statsmodels)

```

import numpy as np
import pandas as pd
import statsmodels.api as sm

```

```

# ISLP's load_data("Auto") reads this same file.
Auto = pd.read_csv("data/auto.csv")
print(Auto.head())

```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
2	18.0	8	318.0	...	70	1	plymouth satellite
3	16.0	8	304.0	...	70	1	amc rebel sst
4	17.0	8	302.0	...	70	1	ford torino

[5 rows x 9 columns]

6.5.2 Python (NumPy)

```
import numpy as np
import pandas as pd
from scipy import stats

# ISLP's load_data("Auto") reads this same file.
Auto = pd.read_csv("data/auto.csv")
print(Auto.head())
```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
2	18.0	8	318.0	...	70	1	plymouth satellite
3	16.0	8	304.0	...	70	1	amc rebel sst
4	17.0	8	302.0	...	70	1	ford torino

[5 rows x 9 columns]

```
y = Auto["mpg"].to_numpy()
horsepower = Auto["horsepower"].to_numpy(float)
weight = Auto["weight"].to_numpy(float)
n = len(y)
```

6.5.3 R (tidymodels)

```
library(tidymodels)

auto <- read_csv("data/auto.csv")
lm_spec <- linear_reg() |> set_engine("lm")
```

6.5.4 R (base)

```
auto <- read_csv("data/auto.csv")
head(auto)

# A tibble: 6 x 9
  mpg cylinders displacement horsepower weight acceleration year origin name
  <dbl>   <int>      <dbl>      <int>  <int>      <dbl> <int> <int> <chr>
1    18     8        307        130  3504         12    70     1 chev~
2    15     8        350        165  3693        11.5   70     1 buic~
3    18     8        318        150  3436         11    70     1 plym~
4    16     8        304        150  3433         12    70     1 amc ~
5    17     8        302        140  3449        10.5   70     1 ford~
6    15     8        429        198  4341         10    70     1 ford~
```

1. The file has 392 rows, one per car.

Exercise 6.8 (Lab: fit a line to the Auto data (part 2)). Use the setup of Exercise 6.7.

Build the design matrix D with a column of ones and a column of horsepower. Compute $\hat{\beta}$ with `np.linalg.lstsq` in Python, or `lm()` in R.

Solution

Solution 6.8.

6.5.5 Python (statsmodels)

```
mpg = Auto["mpg"]
X_auto_simple = pd.DataFrame({"intercept": np.ones(Auto.shape[0]),
                              "horsepower": Auto["horsepower"]})
auto_simple = sm.OLS(mpg, X_auto_simple).fit()

# Each coefficient with its 95% confidence interval
print(pd.concat([auto_simple.params, auto_simple.conf_int()], axis=1)
      .set_axis(["coef", "lower", "upper"], axis=1))
```

	coef	lower	upper
intercept	39.935861	38.525212	41.346510
horsepower	-0.157845	-0.170517	-0.145172

6.5.6 Python (NumPy)

```

def fit(columns):
    Phi = np.column_stack([np.ones(n)] + columns)
    beta_hat = np.linalg.lstsq(Phi, y, rcond=None)[0]
    return beta_hat, Phi

def mean_squared_error(beta_hat, Phi):
    return np.mean((Phi @ beta_hat - y) ** 2)

def rmse(beta_hat, Phi):
    return np.sqrt(mean_squared_error(beta_hat, Phi))

def r_squared(beta_hat, Phi):
    sst = np.sum((y - y.mean()) ** 2)
    return 1 - mean_squared_error(beta_hat, Phi) * n / sst

def ols_summary(beta_hat, Phi, y_obs=y):
    """Residual degrees of freedom, the estimated noise variance, and  $(\Phi^T \Phi)^{-1}$ ."""
    df = len(y_obs) - Phi.shape[1]
    sigma2 = np.sum((y_obs - Phi @ beta_hat) ** 2) / df
    # The pseudoinverse gives  $(\Phi^T \Phi)^{-1}$  without forming  $\Phi^T \Phi$ ,
    # which is badly conditioned for high-degree polynomials
    pinv = np.linalg.pinv(Phi)
    return df, sigma2, pinv @ pinv.T

def coef_ci(beta_hat, Phi):
    """95% interval for each coefficient: the estimate plus or minus a t quantile times its standard error"""
    df, sigma2, PtP_inv = ols_summary(beta_hat, Phi)
    half = stats.t.ppf(0.975, df) * np.sqrt(sigma2 * np.diag(PtP_inv))
    return np.column_stack([beta_hat - half, beta_hat + half])

def mean_ci(beta_hat, Phi, Phi_new, prediction=False, y_obs=y):
    """95% interval at the rows of Phi_new: for the mean response, or with prediction=True for one new response"""
    df, sigma2, PtP_inv = ols_summary(beta_hat, Phi, y_obs)
    # The variance of the fitted value, plus the noise of one new response for a prediction interval
    variance = sigma2 * (np.sum((Phi_new @ PtP_inv) * Phi_new, axis=1) + prediction)
    half = stats.t.ppf(0.975, df) * np.sqrt(variance)
    fitted_new = Phi_new @ beta_hat
    return fitted_new - half, fitted_new + half

def bootstrap_r2_ci(Phi, B=2000, seed=571):
    """95% percentile interval for  $R^2$  from B resamples of the cars, refitted each time."""
    rng = np.random.default_rng(seed)
    r2 = np.empty(B)
    for b in range(B):
        rows = rng.integers(0, n, n)
        beta_b = np.linalg.lstsq(Phi[rows], y[rows], rcond=None)[0]
        sse = np.sum((y[rows] - Phi[rows] @ beta_b) ** 2)
        r2[b] = 1 - sse / np.sum((y[rows] - y[rows].mean()) ** 2)
    return np.quantile(r2, [0.025, 0.975])

beta_simple, Phi_simple = fit([horsepower])
print(np.column_stack([beta_simple, coef_ci(beta_simple, Phi_simple)]))

[[39.93586102 38.52521179 41.34651025]
 [-0.15784473 -0.17051701 -0.14517246]]

```

6.5.7 R (tidymodels)

```
# Each coefficient with its 95% confidence interval
tm_simple <- lm_spec |> fit(mpg ~ horsepower, data = auto)
tidy(tm_simple, conf.int = TRUE)

# A tibble: 2 x 7
  term          estimate std.error statistic    p.value conf.low conf.high
  <chr>          <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 (Intercept)   39.9      0.717     55.7 1.22e-187  38.5     41.3
2 horsepower   -0.158    0.00645   -24.5 7.03e- 81  -0.171   -0.145
```

6.5.8 R (base)

```
fit_simple <- lm(mpg ~ horsepower, data = auto)
# The table includes each coefficient's 95% confidence interval
parameters::parameters(fit_simple) |>
  parameters::print_md(digits = 4)
```

Parameter	Coefficient	SE	95% CI	t(390)	p
(Intercept)	39.9359	0.7175	(38.5252, 41.3465)	55.6598	< .001
horsepower	-0.1578	0.0064	(-0.1705, -0.1452)	-24.4891	< .001

```
coef_ci_simple <- py$coef_ci(py$beta_simple, py$Phi_simple)
rmse_simple <- py$rmse(py$beta_simple, py$Phi_simple)
```

2.

The fit uses $n = 392$ cars. The line is $\widehat{\text{mpg}} = 39.9 - 0.158 \times \text{horsepower}$. Each interval in the coefficient tables is a 95% confidence interval for a coefficient, the estimate^a plus or minus a t quantile times its standard error (Definition 7.3). The intercept^b's interval is 38.5 to 41.3, and the slope^c's is -0.171 to -0.145.

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

^c<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Exercise 6.9 (Lab: fit a line to the Auto data (part 3)). Use the setup of Exercise 6.7.

Draw a scatter plot of `mpg` against `horsepower` with the fitted line. Does a straight line describe the pattern well?

Solution

Solution 6.9.

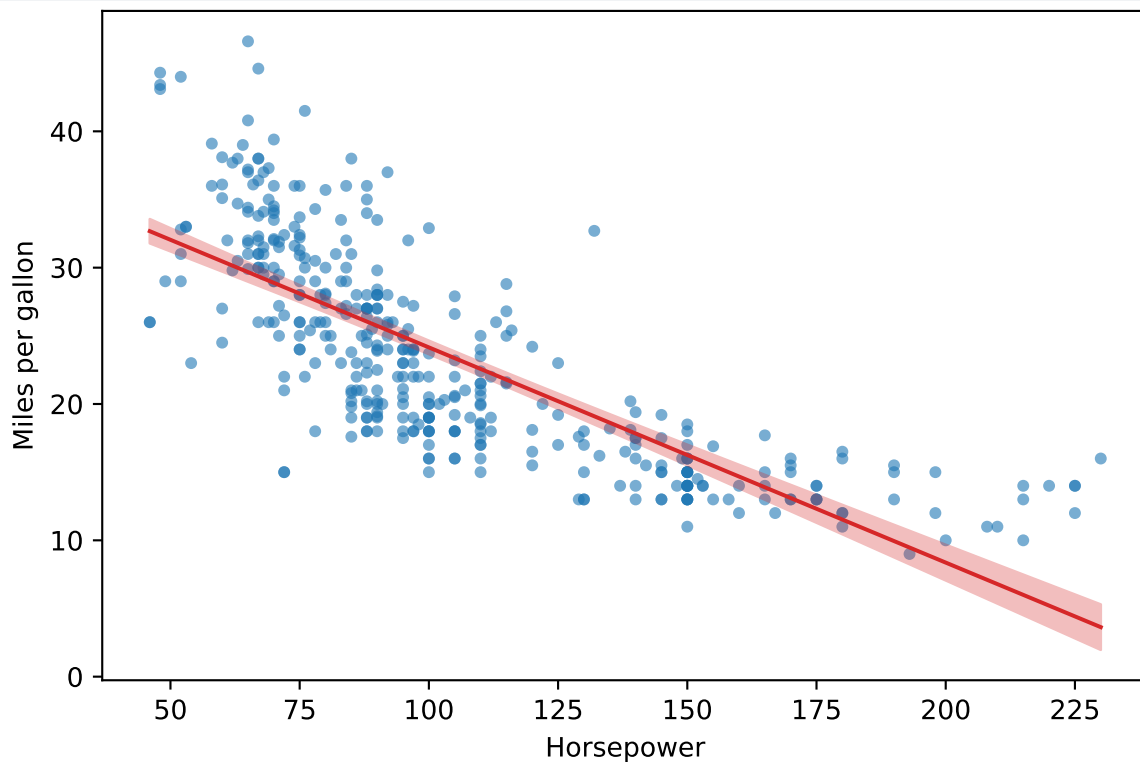
6.5.9 Python (statsmodels)

```

import matplotlib.pyplot as plt
import numpy as np

hp_grid = np.linspace(Auto["horsepower"].min(), Auto["horsepower"].max(), 100)
# The band is the 95% confidence interval for the mean mpg at each horsepower
band = auto_simple.get_prediction(
    pd.DataFrame({"intercept": 1.0, "horsepower": hp_grid})).conf_int(alpha=0.05)
fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.scatter(Auto["horsepower"], mpg, s=12, alpha=0.6);
ax.fill_between(hp_grid, band[:, 0], band[:, 1], color="C3", alpha=0.3);
ax.plot(hp_grid,
        auto_simple.params["intercept"] + auto_simple.params["horsepower"] * hp_grid,
        color="C3");
ax.set_xlabel("Horsepower");
ax.set_ylabel("Miles per gallon");
plt.show()

```



6.5.10 Python (NumPy)

```
import matplotlib.pyplot as plt

grid = np.linspace(horsepower.min(), horsepower.max(), 100)
Phi_grid = np.column_stack([np.ones(100), grid])
band_lower, band_upper = mean_ci(beta_simple, Phi_simple, Phi_grid)
fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.scatter(horsepower, y, s=12, alpha=0.6);
ax.fill_between(grid, band_lower, band_upper, color="C3", alpha=0.3);
ax.plot(grid, beta_simple[0] + beta_simple[1] * grid, color="C3");
ax.set_xlabel("Horsepower");
ax.set_ylabel("Miles per gallon");
plt.show()
```

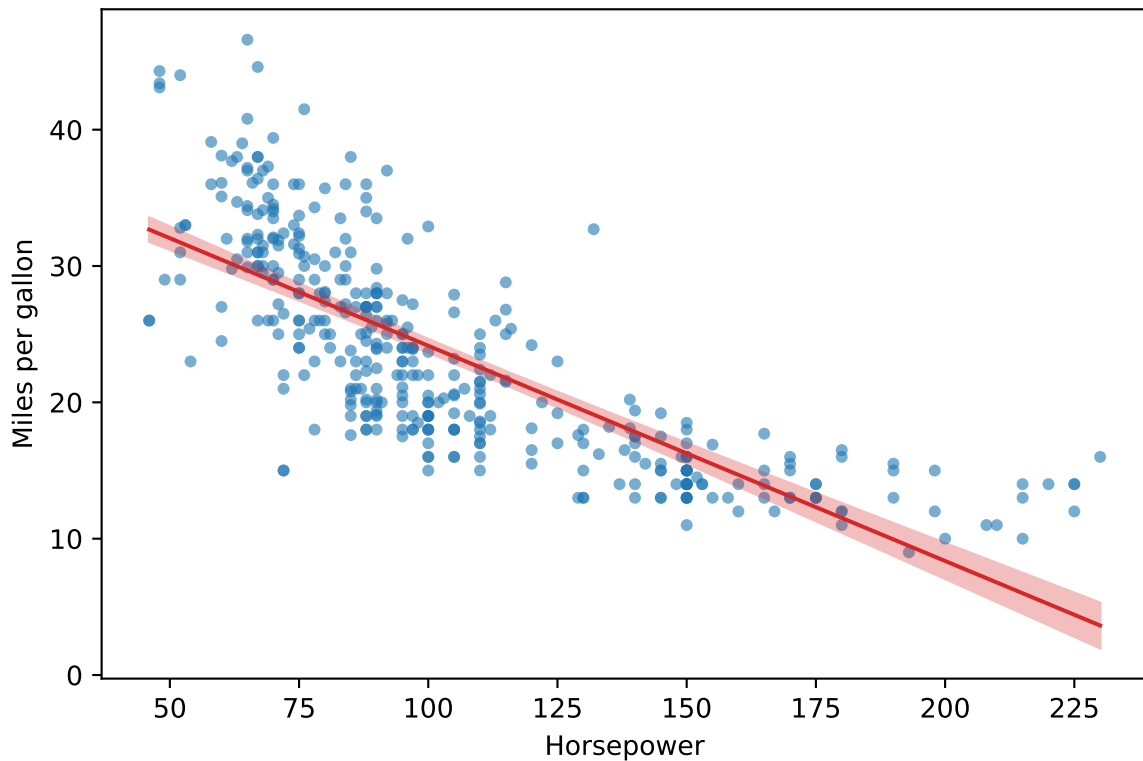


Figure 18: Fuel economy against horsepower for the Auto data, with the least squares line and its 95% confidence band for the mean.

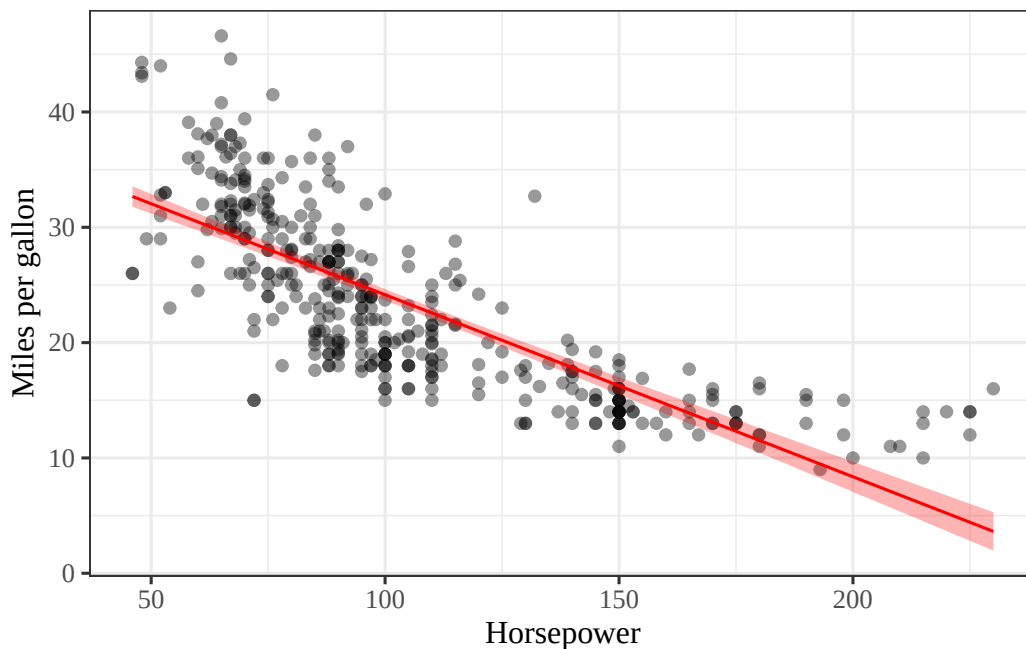
6.5.11 R (tidymodels)

```
library(tidymodels)

# The band is the 95% confidence interval for the mean mpg at each horsepower
hp_grid <- tibble(horsepower = seq(min(auto$horsepower), max(auto$horsepower),
                                   length.out = 100))

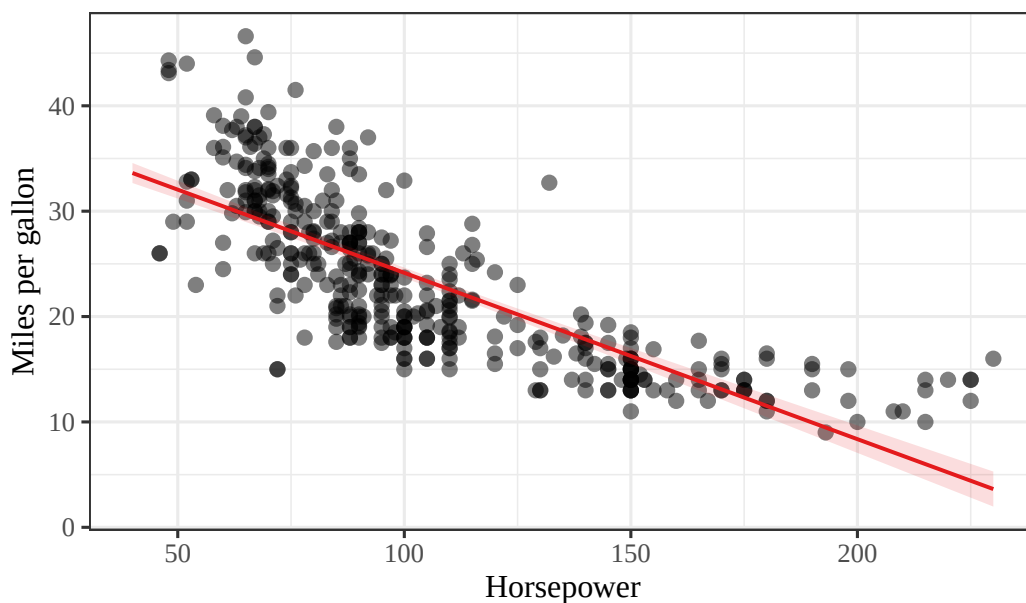
simple_band <- bind_cols(
  hp_grid,
  predict(tm_simple, new_data = hp_grid),
  predict(tm_simple, new_data = hp_grid, type = "conf_int")
)

ggplot() +
  geom_point(data = auto, aes(horsepower, mpg), alpha = 0.4) +
  geom_ribbon(data = simple_band,
            aes(horsepower, ymin = .pred_lower, ymax = .pred_upper),
            fill = "red", alpha = 0.3) +
  geom_line(data = simple_band, aes(horsepower, .pred), colour = "red") +
  labs(x = "Horsepower", y = "Miles per gallon")
```



6.5.12 R (base)

```
sjPlot::plot_model(fit_simple, type = "pred", terms = "horsepower",
                   show.data = TRUE, title = "") +
  ggplot2::labs(x = "Horsepower", y = "Miles per gallon")
```



3. The scatter plot shows a curve, not a line: the points bend, and the line misses them in the same direction at both ends. That is the pattern a later exercise fixes with a quadratic term. The shaded band in the figure is the 95% confidence interval^a for the mean^b mpg at every horsepower.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Exercise 6.10 (Lab: fit a line to the Auto data (part 4)). Use the setup of Exercise 6.7. Compute the sMSE and $R^2 = 1 - \text{SSE}/\text{SST}$, where SSE is the sum of squared residuals and SST is the sum of squared deviations^a of mpg from its sample mean^b.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-deviation>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Solution

Solution 6.10.

6.5.13 Python (statsmodels)

```
# Training root mean squared error, in miles per gallon
print(np.sqrt(auto_simple.ssr / auto_simple.nobs))
4.8932262300657134
print(auto_simple.rsquared)
0.6059482578894348
```

```
# R^2 has no formula for its interval, so refit on resampled cars.
# Resampling rows with replacement mimics drawing a new set of 392 cars from the same population.
def bootstrap_rsquared_ci(X, B=2000, seed=571):
    """95% percentile interval for R^2 from B resamples of the cars."""
    rng = np.random.default_rng(seed)
    X_arr, mpg_arr = np.asarray(X, float), mpg.to_numpy()
    rsquared = np.empty(B)
    for b in range(B):
        rows = rng.integers(0, len(mpg_arr), len(mpg_arr))
        rsquared[b] = sm.OLS(mpg_arr[rows], X_arr[rows]).fit().rsquared
    return np.quantile(rsquared, [0.025, 0.975])

print(bootstrap_rsquared_ci(X_auto_simple))

[0.55644218 0.65267338]
```

6.5.14 Python (NumPy)

```
print(rmse(beta_simple, Phi_simple))
4.893226230065713

print(r_squared(beta_simple, Phi_simple))
0.6059482578894351

r2_ci_simple = bootstrap_r2_ci(Phi_simple)
print(r2_ci_simple)

[0.55644218 0.65267338]
```

6.5.15 R (tidymodels)

```
# Training root mean squared error and R^2 of a fitted model
auto_metrics <- function(model, data = auto) {
  augment(model, new_data = data) |>
  summarise(rmse = sqrt(mean((mpg - .pred)^2)),
            r_squared = rsq_trad_vec(mpg, .pred))
}

auto_metrics(tm_simple)

# A tibble: 1 x 2
  rmse r_squared
<dbl> <dbl>
1 4.89 0.606

# R^2 has no formula for its interval,
# so boot_r2_ci (in the interval helpers) refits on resampled cars.
boot_r2_ci(mpg ~ horsepower, auto)

[1] 0.5598592 0.6525384
```

6.5.16 R (base)

```
rmse <- function(model) sqrt(mean(residuals(model)^2))
r_squared <- function(model) summary(model)$r.squared
c(rmse = rmse(fit_simple), r_squared = r_squared(fit_simple))

rmse r_squared
4.8932262 0.6059483
```

```
# R^2 has no formula for its interval,
# so boot_r2_ci (in the interval helpers) refits on resampled cars.
boot_r2_ci(mpg ~ horsepower, auto)
```

```
[1] 0.5598592 0.6525384
```

4. The exercise asks for the sample mean squared error (sMSE); we report its square root, the sample root mean squared error (sRMSE), which is in miles per gallon (why we report sRMSE). The training sRMSE is 4.9 miles per gallon. It has no interval, because the same cars chose the coefficients and measure the error, so it understates the error on new cars, and an interval around it would describe the wrong quantity. The coefficient of determination is $R^2 = 0.61$, with a 95% bootstrap interval of 0.56 to 0.65: the 2.5th and 97.5th percentiles of R^2 over 2,000 resamples of the cars drawn with replacement, each resample refitted. The Python and R tabs draw different random resamples, so their bootstrap endpoints agree only up to simulation error.

Exercise 6.11 (Lab: fit a line to the Auto data (part 5)). Use the setup of Exercise 6.7. Predict mpg for a car with 98 horsepower. Say in words what the slope^a means^b.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Solution

Solution 6.11.

6.5.17 Python (statsmodels)

```
# The prediction for a 98-horsepower car, with the confidence interval for the mean
# (mean_ci_*) and the prediction interval for one car (obs_ci_*)
new_car = pd.DataFrame({"intercept": [1.0], "horsepower": [98]})
print(auto_simple.get_prediction(new_car).summary_frame(alpha=0.05).to_string())
```

	mean	mean_se	mean_ci_lower	mean_ci_upper	obs_ci_lower	obs_ci_upper
0	24.467077	0.251262	23.973079	24.961075	14.809396	34.124758

6.5.18 Python (NumPy)

```
# The prediction for a 98-horsepower car, with the interval for the mean and for one car
Phi_98 = np.array([[1.0, 98.0]])
print(beta_simple @ [1, 98])
```

24.467077152512417

```
mean_ci_98 = np.array(mean_ci(beta_simple, Phi_simple, Phi_98)).ravel()
pred_ci_98 = np.array(mean_ci(beta_simple, Phi_simple, Phi_98, prediction=True)).ravel()
print(mean_ci_98, pred_ci_98)
```

[23.97307896 24.96107534] [14.80939607 34.12475823]

6.5.19 R (tidymodels)

```
# The prediction for a 98-horsepower car,
# with the interval for the mean and for one car
new_car <- tibble(horsepower = 98)
predict(tm_simple, new_data = new_car)
```

A tibble: 1 x 1

.pred
<dbl>

```
1 24.5
predict(tm_simple, new_data = new_car, type = "conf_int")
```

```
# A tibble: 1 x 2
  .pred_lower .pred_upper
      <dbl>      <dbl>
1      24.0      25.0
```

```
predict(tm_simple, new_data = new_car, type = "pred_int")
```

```
# A tibble: 1 x 2
  .pred_lower .pred_upper
      <dbl>      <dbl>
1      14.8      34.1
```

6.5.20 R (base)

```
# The prediction for a 98-horsepower car,
# with the interval for the mean and for one car
new_car <- tibble::tibble(horsepower = 98)
predict(fit_simple, new_car, interval = "confidence")
```

```
      fit      lwr      upr
1 24.46708 23.97308 24.96108
```

```
predict(fit_simple, new_car, interval = "prediction")
```

```
      fit      lwr      upr
1 24.46708 14.8094 34.12476
```

5. A 98-horsepower car is predicted to get 24.5 miles per gallon. The 95% confidence interval^a for the average of such cars is 24.0 to 25.0 (Definition 7.7), and the 95% prediction interval for one such car is 14.8 to 34.1. The prediction interval is wider because it includes the noise of a single car around the line. The slope^b says that, across these cars, each extra horsepower goes with 0.16 fewer miles per gallon on average (95% confidence interval 0.15 to 0.17 fewer). It describes a pattern in the data, and does not say what would happen to one car if its engine were changed.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Exercise 6.12 (Lab: add a second feature). Add **weight** to the model of Exercise 6.7. Fit **mpg** on **horsepower** and **weight**. Report the coefficients, the sMSE and R^2 .

Solution

Solution 6.12.

6.5.21 Python (statsmodels)

```
import statsmodels.api as sm

X_auto_multiple = sm.add_constant(Auto[["horsepower", "weight"]])
auto_multiple = sm.OLS(mpg, X_auto_multiple).fit()

# Each coefficient with its 95% confidence interval
print(pd.concat([auto_multiple.params, auto_multiple.conf_int()], axis=1)
      .set_axis(["coef", "lower", "upper"], axis=1))
```

```
      coef      lower      upper
const  45.640211  44.080724  47.199698
```

```

horsepower -0.047303 -0.069097 -0.025509
weight      -0.005794 -0.006782 -0.004807
print(np.sqrt(auto_multiple.ssr / auto_multiple.nobs)) # training sRMSE
4.2239131670230385
print(auto_multiple.rsquared)
0.7063752737298348
print(bootstrap_rsquared_ci(X_auto_multiple))
[0.66805951 0.74469229]
# The correlation of two features has no simple interval either,
# so bootstrap_corr_ci (in the interval helpers)
# resamples the cars and recomputes it each time

print(Auto["horsepower"].corr(Auto["weight"]))
0.8645377375741444
print(bootstrap_corr_ci(Auto["horsepower"], Auto["weight"]))
[0.83368218 0.89020865]

```

6.5.22 Python (NumPy)

```

beta_multiple, Phi_multiple = fit([horsepower, weight])
print(np.column_stack([beta_multiple, coef_ci(beta_multiple, Phi_multiple)]))

[[ 4.56402108e+01  4.40807235e+01  4.71996981e+01]
 [-4.73028631e-02 -6.90970417e-02 -2.55086845e-02]
 [-5.79415736e-03 -6.78177296e-03 -4.80654177e-03]]

print(rmse(beta_multiple, Phi_multiple))
4.2239131670230385
print(r_squared(beta_multiple, Phi_multiple))
0.7063752737298348
r2_ci_multiple = bootstrap_r2_ci(Phi_multiple)
print(r2_ci_multiple)
[0.66805951 0.74469229]
# The correlation of two features has no simple interval either,
# so bootstrap_corr_ci (in the interval helpers)
# resamples the cars and recomputes it each time

print(np.corrcoef(horsepower, weight)[0, 1])
0.864537737574144
corr_ci_hp_weight = bootstrap_corr_ci(horsepower, weight)
print(corr_ci_hp_weight)
[0.83368218 0.89020865]

```

6.5.23 R (tidymodels)

```

library(tidymodels)

# Each coefficient with its 95% confidence interval
tm_multiple <- lm_spec |> fit(mpg ~ horsepower + weight, data = auto)
tidy(tm_multiple, conf.int = TRUE)

# A tibble: 3 x 7
  term      estimate std.error statistic  p.value conf.low conf.high
<chr>      <dbl>     <dbl>     <dbl>   <dbl>   <dbl>   <dbl>
1 (Intercept) 45.6      0.793      57.5 2.32e-192 44.1     47.2
2 horsepower  -0.0473    0.0111     -4.27 2.49e- 5 -0.0691 -0.0255
3 weight      -0.00579 0.000502    -11.5 1.12e- 26 -0.00678 -0.00481

auto_metrics(tm_multiple)

# A tibble: 1 x 2
  rmse r_squared
<dbl> <dbl>
1 4.22 0.706

boot_r2_ci(mpg ~ horsepower + weight, auto)

[1] 0.6692914 0.7438732

# The correlation of two features has no simple interval either,
# so boot_cor_ci (in the interval helpers)
# resamples the cars and recomputes it each time
auto |> summarise(correlation = cor(horsepower, weight))

# A tibble: 1 x 1
  correlation
      <dbl>
1 0.865

boot_cor_ci(auto$horsepower, auto$weight)

[1] 0.8338286 0.8898624

```

6.5.24 R (base)

```

# Each coefficient with its 95% confidence interval
fit_multiple <- lm(mpg ~ horsepower + weight, data = auto)
parameters::parameters(fit_multiple) |>
  parameters::print_md(digits = 4)

```

Parameter	Coefficient	SE	95% CI	t(389)	p
(Intercept)	45.6402	0.7932	(44.0807, 47.1997)	57.5397	< .001
horsepower	-0.0473	0.0111	(-0.0691, -0.0255)	-4.2673	< .001
weight	-0.0058	0.0005	(-0.0068, -0.0048)	-11.5346	< .001

```
c(rmse = rmse(fit_multiple), r_squared = r_squared(fit_multiple))
```

```
      rmse r_squared  
4.2239132 0.7063753
```

```
boot_r2_ci(mpg ~ horsepower + weight, auto)
```

```
[1] 0.6692914 0.7438732
```

```
# The correlation of two features has no simple interval either,  
# so boot_cor_ci (in the interval helpers)  
# resamples the cars and recomputes it each time
```

```
cor(auto$horsepower, auto$weight)
```

```
[1] 0.8645377
```

```
boot_cor_ci(auto$horsepower, auto$weight)
```

```
[1] 0.8338286 0.8898624
```

```
coef_ci_multiple <- py$coef_ci(py$beta_multiple, py$Phi_multiple)
```

The sample root mean squared error (sRMSE) falls from 4.9 to 4.2 miles per gallon, and R^2 rises from 0.61 (95% bootstrap interval 0.56 to 0.65) to 0.71 (0.67 to 0.74). The intervals are percentiles of R^2 over 2,000 resamples of the cars, each refitted. As in Solution 6.5.4, the training sRMSE has no interval, because the same cars chose the coefficients and measure the error. The intercept is 45.64, the `horsepower` coefficient is -0.0473, and the `weight` coefficient is -0.00579. The 95% confidence intervals of the coefficients are in the tables above.

Exercise 6.13 (Lab: add a second feature (part 2)). Use the setup of Exercise 6.12. Did the `horsepower` coefficient change?

Solution

Solution 6.13. The `horsepower` coefficient moved from -0.158 (95% confidence interval^a -0.171 to -0.145) to -0.047 (-0.069 to -0.026), a much smaller effect.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

Exercise 6.14 (Lab: add a second feature (part 3)). Use the setup of Exercise 6.12. Compute the correlation^a between `horsepower` and `weight`.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-correlation>

Solution

Solution 6.14. The correlation^a of the two features is 0.86, with a 95% bootstrap interval of 0.83 to 0.89.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-correlation>

Exercise 6.15 (Lab: add a second feature (part 4)). Use the setup of Exercise 6.12. Use it to explain why.

Solution

Solution 6.15. Heavy cars tend to have powerful engines, so in the one-feature fit `horsepower` also stands in for `weight`. In the two-feature fit, its coefficient is the change in `mpg` per `horsepower` with `weight` held fixed.

Exercise 6.16 (Lab: add a second feature (part 5)). Use the setup of Exercise 6.12. Can the training error of the larger model be higher than that of the smaller one? Use the normal equations to explain.

Solution

Solution 6.16. No. The smaller model is the larger one with the `weight` coefficient set to 0, and least squares finds the lowest error over all coefficients. So adding a column can never raise the training error. So training error cannot tell us whether a bigger model is better, a question for the next lectures on generalization.

Exercise 6.17 (Lab: a curve, and a held-out check). Fit `mpg` on `horsepower` and `horsepower2`. Report the training sMSE.

Solution

Solution 6.17.

6.5.25 Python (statsmodels)

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from scipy import stats

# Columns 1, hp, hp^2, ..., hp^degree
def poly_design(hp, degree):
    return pd.DataFrame({f"hp^{k}": hp.astype(float) ** k for k in range(degree + 1)})

auto_quadratic = sm.OLS(mpg, poly_design(Auto["horsepower"], 2)).fit()
print(np.sqrt(auto_quadratic.ssr / auto_quadratic.nobs)) # training sRMSE
4.3571514671419465
```

6.5.26 Python (NumPy)

```
beta_quadratic, Phi_quadratic = fit([horsepower, horsepower ** 2])
print(rmse(beta_quadratic, Phi_quadratic))
4.3571514671419465
```

6.5.27 R (tidymodels)

```
library(tidymodels)

quadratic_recipe <- recipe(mpg ~ horsepower, data = auto) |>
  step_mutate(horsepower2 = horsepower^2)
tm_quadratic <- workflow() |>
  add_recipe(quadratic_recipe) |>
  add_model(lm_spec) |>
  fit(auto)
auto_metrics(tm_quadratic)

# A tibble: 1 x 2
  rmse r_squared
```

```
<dbl>    <dbl>
1  4.36    0.688
```

6.5.28 R (base)

```
fit_quadratic <- lm(mpg ~ horsepower + I(horsepower^2), data = auto)
rmse(fit_quadratic)
```

```
[1] 4.357151
```

1. The training sample root mean squared error (sRMSE) of the quadratic is 4.4 miles per gallon, against 4.9 for the straight line. The curve fits better. As in Solution 6.5.4, a training sRMSE has no interval, because the same cars chose the coefficients and measure the error.

Exercise 6.18 (Lab: a curve, and a held-out check (part 2)). Use the setup of Exercise 6.17.

Use the even-numbered rows (0, 2, 4, ... counting from 0) as a training set and the odd-numbered rows as a test set, 196 cars each. For polynomial^a degrees^b 1, 2 and 5 in `horsepower`, fit on the training set and report the sMSE on both sets.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Solution

Solution 6.18.

6.5.29 Python (statsmodels)

```
# The sRMSE of held-out cars, with a 95% t interval for the mean of their squared errors,
# whose endpoints are then square-rooted
def rmse_with_ci(squared_errors):
    mean_squared = squared_errors.mean()
    half = (stats.t.ppf(0.975, len(squared_errors) - 1)
            * squared_errors.std(ddof=1) / np.sqrt(len(squared_errors)))
    return np.sqrt([mean_squared, mean_squared - half, mean_squared + half])

auto_train = np.arange(Auto.shape[0]) % 2 == 0
errors_by_degree = []
for degree in [1, 2, 5]:
    X_poly = poly_design(Auto["horsepower"], degree)
    poly_results = sm.OLS(mpg[auto_train], X_poly[auto_train]).fit()
    test_squared = ((mpg[~auto_train] - poly_results.predict(X_poly[~auto_train])) ** 2).to_numpy()
    errors_by_degree.append([degree, np.sqrt(poly_results.ssr / poly_results.nobs),
                             *rmse_with_ci(test_squared)])
print(pd.DataFrame(errors_by_degree,
                   columns=["degree", "train RMSE", "test RMSE", "test lower", "test upper"])
      .to_string(index=False))
```

degree	train RMSE	test RMSE	test lower	test upper
1	5.003756	4.796201	4.244935	5.290333
2	4.532780	4.175060	3.598475	4.681156
5	4.498467	4.135151	3.559981	4.639557

6.5.30 Python (NumPy)

```

# The sRMSE of held-out cars, with a 95% t interval for the mean of their squared errors,
# whose endpoints are then square-rooted
def rmse_with_ci(squared_errors):
    mean_squared = squared_errors.mean()
    half = (stats.t.ppf(0.975, len(squared_errors) - 1)
            * squared_errors.std(ddof=1) / np.sqrt(len(squared_errors)))
    return np.sqrt([mean_squared, mean_squared - half, mean_squared + half])

train = np.arange(n) % 2 == 0
test = ~train

errors_by_degree_np = []
for degree in [1, 2, 5]:
    Phi = np.column_stack([horsepower ** k for k in range(degree + 1)])
    beta_hat = np.linalg.lstsq(Phi[train], y[train], rcond=None)[0]
    train_rmse = np.sqrt(np.mean((Phi[train] @ beta_hat - y[train]) ** 2))
    test_squared = (Phi[test] @ beta_hat - y[test]) ** 2
    errors_by_degree_np.append([degree, train_rmse, *rmse_with_ci(test_squared)])
    print(degree, np.round(errors_by_degree_np[-1][1:], 3))

1 [5.004 4.796 4.245 5.29 ]
2 [4.533 4.175 3.598 4.681]
5 [4.498 4.135 3.56  4.64 ]

```

6.5.31 R (tidymodels)

```

# Even-numbered rows counting from 0 are R's odd-numbered rows
auto_train <- auto |> filter(row_number() %% 2 == 1)
auto_test <- auto |> filter(row_number() %% 2 == 0)

poly_fit <- function(degree) {
  poly_recipe <- recipe(mpg ~ horsepower, data = auto_train) |>
    step_poly(horsepower, degree = degree, options = list(raw = TRUE))
  workflow() |>
    add_recipe(poly_recipe) |>
    add_model(lm_spec) |>
    fit(auto_train)
}

# The sRMSE of held-out cars,
# with a 95% t interval for the mean of their squared errors,
# whose endpoints are then square-rooted
test_rmse_ci <- function(model, data) {
  squared_errors <- (augment(model, new_data = data)$pred - data$mpg)^2
  n_test <- length(squared_errors)
  half <- qt(0.975, n_test - 1) * sd(squared_errors) / sqrt(n_test)
  tibble(test_rmse = sqrt(mean(squared_errors)),
         test_lower = sqrt(mean(squared_errors) - half),
         test_upper = sqrt(mean(squared_errors) + half))
}
map(c(1, 2, 5), \(degree) {
  model <- poly_fit(degree)
  bind_cols(tibble(degree, train_rmse = auto_metrics(model, auto_train)$rmse),
            test_rmse_ci(model, auto_test))
}) |>
  list_rbind()

# A tibble: 3 x 5
  degree train_rmse test_rmse test_lower test_upper
  <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
1     1         5.00         4.80         4.24         5.29
2     2         4.53         4.18         3.60         4.68
3     5         4.50         4.14         3.56         4.64

```

6.5.32 R (base)

```

# The sRMSE of held-out cars,
# with a 95% t interval for the mean of their squared errors,
# whose endpoints are then square-rooted
train <- seq_len(nrow(auto)) %% 2 == 1
rmse_by_degree <- function(degree) {
  model <- lm(mpg ~ poly(horsepower, degree, raw = TRUE),
             data = dplyr::filter(auto, train))
  squared_errors <- function(rows) {
    (auto$mpg[rows] - predict(model, dplyr::filter(auto, rows)))^2
  }
  held_out <- squared_errors(!train)
  n_held_out <- length(held_out)
  half <- qt(0.975, n_held_out - 1) * sd(held_out) / sqrt(n_held_out)
  c(degree = degree,
     train_rmse = sqrt(mean(squared_errors(train))),
     test_rmse = sqrt(mean(held_out)),
     test_lower = sqrt(mean(held_out) - half),
     test_upper = sqrt(mean(held_out) + half))
}
t(sapply(c(1, 2, 5), rmse_by_degree))

  degree train_rmse test_rmse test_lower test_upper
[1,]      1   5.003756  4.796201   4.244935   5.290333
[2,]      2   4.532780  4.175060   3.598475   4.681156
[3,]      5   4.498467  4.135151   3.559981   4.639557

quadratic_table <- as.data.frame(do.call(rbind, py$errors_by_degree_np))
names(quadratic_table) <- c("degree", "train", "test", "lower", "upper")
quadratic_table[] <- lapply(quadratic_table, unlist)
# Whether the three test-sRMSE intervals share a common value,
# and whether every test error is below its training error
intervals_overlap <- max(quadratic_table$lower) <= min(quadratic_table$upper)
test_below_train <- all(quadratic_table$test < quadratic_table$train)

```

2. The test sRMSE is 4.80 miles per gallon for degree^a 1 (95% interval 4.24 to 5.29), 4.18 for degree 2 (3.60 to 4.68) and 4.14 for degree 5 (3.56 to 4.64). Each interval is a 95% t interval for the mean^b of the 196 squared errors on the test cars, with its endpoints square-rooted, so it reflects which test cars were drawn, with the fitted curve held fixed.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Exercise 6.19 (Lab: a curve, and a held-out check (part 3)). Use the setup of Exercise 6.17. Plot the three fitted curves over a scatter plot of the test cars.

Solution

Solution 6.19.

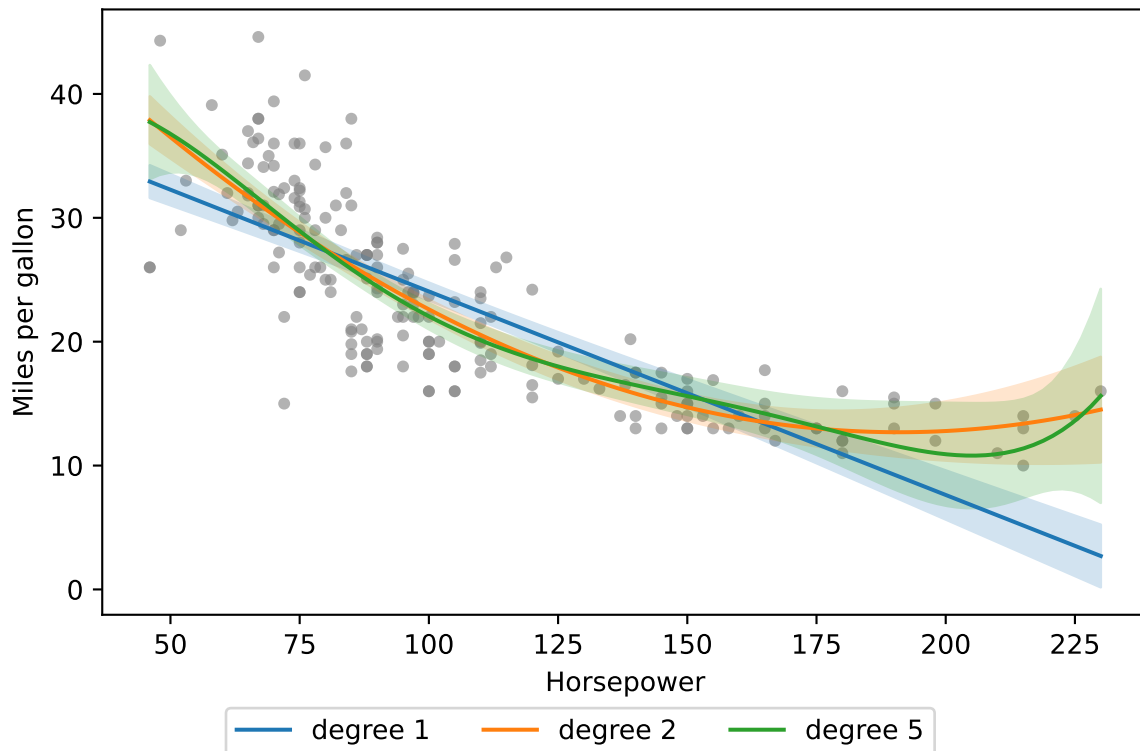
6.5.33 Python (statsmodels)

```

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm

hp_grid = pd.Series(np.linspace(Auto["horsepower"].min(), Auto["horsepower"].max(), 100))
fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.scatter(Auto["horsepower"][~auto_train], mpg[~auto_train], s=12, alpha=0.6, color="gray");
for degree in [1, 2, 5]:
    X_poly = poly_design(Auto["horsepower"], degree)
    poly_results = sm.OLS(mpg[auto_train], X_poly[auto_train]).fit()
    grid_design = poly_design(hp_grid, degree)
    # The band is the 95% confidence interval for the mean mpg at each horsepower
    band = poly_results.get_prediction(grid_design).conf_int(alpha=0.05)
    (line,) = ax.plot(hp_grid, poly_results.predict(grid_design), label=f"degree {degree}")
    ax.fill_between(hp_grid, band[:, 0], band[:, 1], color=line.get_color(), alpha=0.2);
ax.set_xlabel("Horsepower");
ax.set_ylabel("Miles per gallon");
fig.legend(loc="outside lower center", ncol=3);
plt.show()

```



6.5.34 Python (NumPy)

```

grid = np.linspace(horsepower.min(), horsepower.max(), 100)
fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.scatter(horsepower[test], y[test], s=12, alpha=0.6, color="gray");
for degree in [1, 2, 5]:
    Phi = np.column_stack([horsepower ** k for k in range(degree + 1)])
    beta_hat = np.linalg.lstsq(Phi[train], y[train], rcond=None)[0]
    Phi_grid = np.column_stack([grid ** k for k in range(degree + 1)])
    # The band is the 95% confidence interval for the mean mpg at each horsepower,
    # from the fit to the training cars
    band_lower, band_upper = mean_ci(beta_hat, Phi[train], Phi_grid, y_obs=y[train])
    (line,) = ax.plot(grid, Phi_grid @ beta_hat, label=f"degree {degree}")
    ax.fill_between(grid, band_lower, band_upper, color=line.get_color(), alpha=0.2);
ax.set_xlabel("Horsepower");
ax.set_ylabel("Miles per gallon");
fig.legend(loc="outside lower center", ncol=3);
plt.show()

```

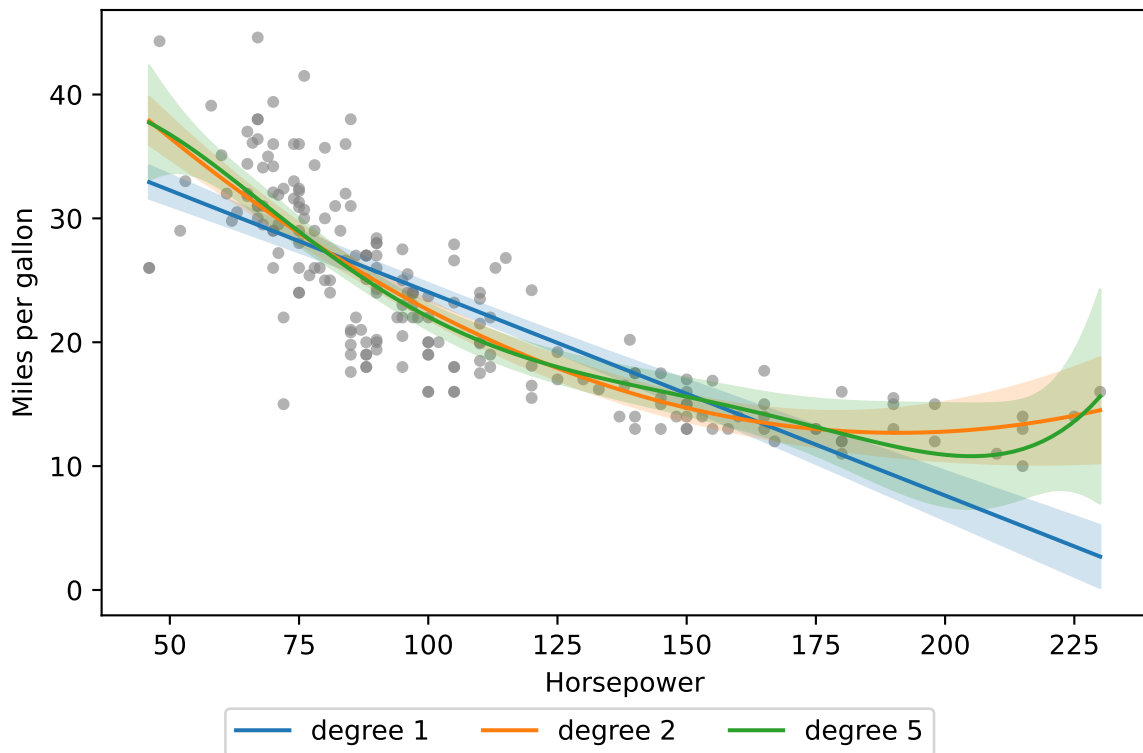


Figure 19: Test cars with polynomial fits of degree 1, 2 and 5 to horsepower, fitted on the training cars, each with its 95% confidence band for the mean.

6.5.35 R (tidymodels)

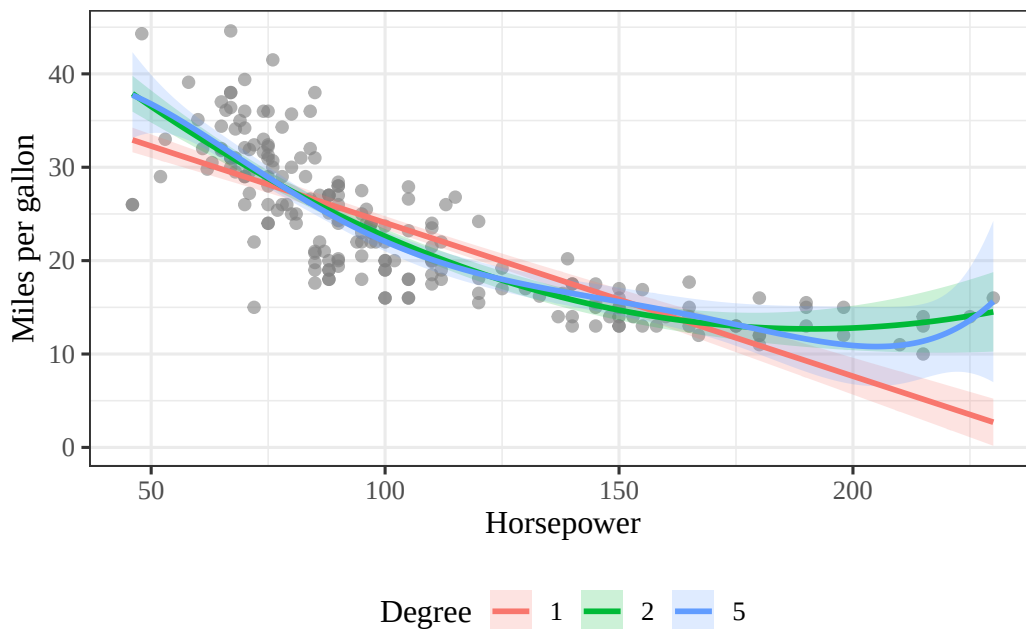
```

library(tidymodels)

# The bands are the 95% confidence intervals
# for the mean mpg at each horsepower
hp_grid <- tibble(horsepower = seq(min(auto$horsepower),
                                   max(auto$horsepower)))

curves <- map(c(1, 2, 5), \(degree) {
  model <- poly_fit(degree)
  bind_cols(hp_grid,
            predict(model, new_data = hp_grid),
            predict(model, new_data = hp_grid, type = "conf_int"),
            degree = factor(degree))
}) |>
  list_rbind()
ggplot() +
  geom_point(data = auto_test, aes(horsepower, mpg),
            colour = "grey50", alpha = 0.6) +
  geom_ribbon(data = curves,
            aes(horsepower, ymin = .pred_lower, ymax = .pred_upper,
                fill = degree),
            alpha = 0.2) +
  geom_line(data = curves, aes(horsepower, .pred, colour = degree),
            linewidth = 1) +
  labs(x = "Horsepower", y = "Miles per gallon",
       colour = "Degree", fill = "Degree")

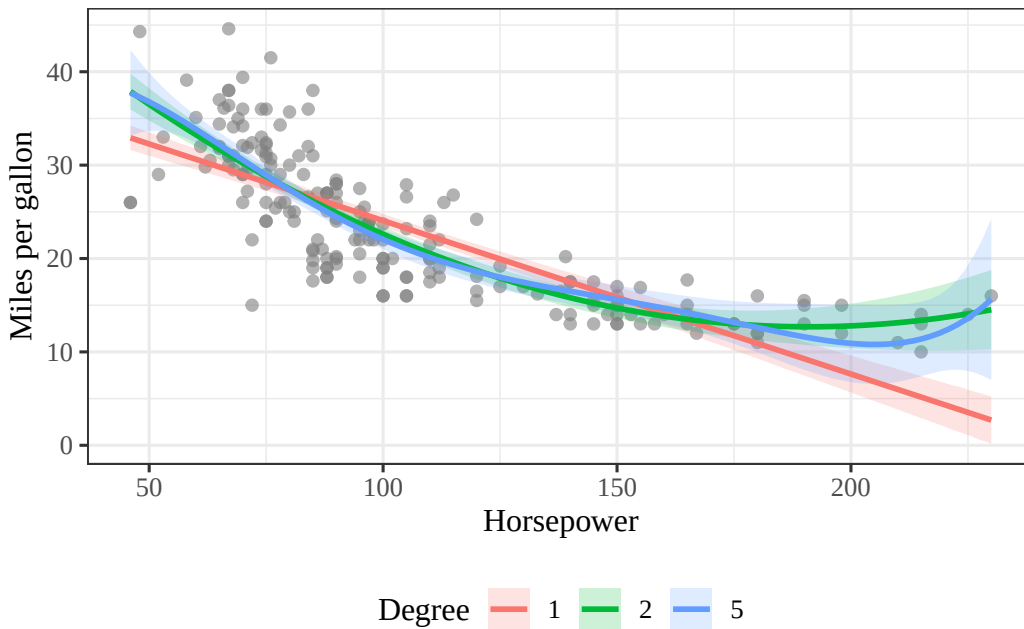
```



6.5.36 R (base)

```
library(ggplot2)

test_cars <- dplyr::filter(auto, !train)
grid <- tibble::tibble(
  horsepower = seq(min(auto$horsepower), max(auto$horsepower), length.out = 100)
)
train_cars <- dplyr::filter(auto, train)
# The bands are the 95% confidence intervals
# for the mean mpg at each horsepower
poly_curves <- c(1, 2, 5) |>
  lapply(function(degree) {
    model <- lm(mpg ~ poly(horsepower, degree, raw = TRUE), data = train_cars)
    fit_band <- predict(model, grid, interval = "confidence")
    tibble::tibble(grid, mpg = fit_band[, "fit"],
                   lower = fit_band[, "lwr"], upper = fit_band[, "upr"],
                   degree = factor(degree))
  }) |>
  dplyr::bind_rows()
ggplot(test_cars, aes(horsepower, mpg)) +
  geom_point(colour = "grey50", alpha = 0.6) +
  geom_ribbon(data = poly_curves,
            aes(ymin = lower, ymax = upper, fill = degree),
            alpha = 0.2) +
  geom_line(data = poly_curves, aes(colour = degree), linewidth = 1) +
  labs(x = "Horsepower", y = "Miles per gallon",
       colour = "Degree", fill = "Degree")
```



3. The shaded bands in the figures are 95% confidence intervals^a for the mean^b mpg at each horsepower, from the fit to the training cars.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Exercise 6.20 (Lab: a curve, and a held-out check (part 4)). Use the setup of Exercise 6.17. Which degree^a would you choose?

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Solution

Solution 6.20. Going from degree^a 1 to degree 2 lowers both errors by a lot. Going from degree 2 to degree 5 lowers both only slightly. Degree 2 gets nearly all of the gain with 3 coefficients, so it is the natural choice here.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

Exercise 6.21 (Lab: a curve, and a held-out check (part 5)). Use the setup of Exercise 6.17. What does the comparison of the two error columns show that the training error alone does not?

Solution

Solution 6.21. The training error always falls as the degree^a rises, as in Solution 6.5.24, so a small fall is weak evidence for the extra terms. The test error is the honest column, because we did not use those cars to choose the coefficients. The intervals all overlap, but the same test cars score every degree, so the three errors move together, and an overlap would not show that the degrees^b tie. Here every test error is smaller than its training error. The usual order is the reverse, and a split like this one shows that one split is a noisy estimate^c: a different split can give a different order. Measuring that gap, and choosing a model by test error, is the subject of Generalization.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

^c<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Exercise 6.22 (Lab: use a category as a feature). The `origin` column codes where each car was made: 1 for American, 2 for European and 3 for Japanese. The numbers are labels, and not amounts, so a column of 1, 2 and 3 would be wrong as a feature. Build two indicator columns, one that is 1 for European cars and one that is 1 for Japanese cars. Fit `mpg` on these two columns alone.

Solution

Solution 6.22.

6.5.37 Python (statsmodels)

```
import pandas as pd
import statsmodels.api as sm

place = Auto["origin"].map({1: "American", 2: "European", 3: "Japanese"})
# drop_first drops "American", the first level, which becomes the baseline
origin_columns = pd.get_dummies(place, drop_first=True, dtype=float)

auto_origin = sm.OLS(mpg, sm.add_constant(origin_columns)).fit()
# Each coefficient with its 95% confidence interval
print(pd.concat([auto_origin.params, auto_origin.conf_int()], axis=1)
      .set_axis(["coef", "lower", "upper"], axis=1))
```

coef	lower	upper
------	-------	-------

```
const      20.033469  19.230049  20.836890
European   7.569472   5.845776   9.293167
Japanese  10.417164   8.790110  12.044217
```

6.5.38 Python (NumPy)

```
origin = Auto["origin"].to_numpy()
european = (origin == 2).astype(float)
japanese = (origin == 3).astype(float)

beta_origin, Phi_origin = fit([european, japanese])
print(np.column_stack([beta_origin, coef_ci(beta_origin, Phi_origin)]))

[[20.03346939 19.23004902 20.83688976]
 [ 7.56947179  5.84577627  9.2931673 ]
 [10.41716352  8.79011016 12.04421689]]
```

6.5.39 R (tidymodels)

```
library(tidymodels)

auto_origin <- auto |>
  mutate(origin = factor(origin, levels = 1:3,
                          labels = c("American", "European", "Japanese")))

origin_recipe <- recipe(mpg ~ origin, data = auto_origin) |>
  step_dummy(origin)
tm_origin <- workflow() |>
  add_recipe(origin_recipe) |>
  add_model(lm_spec) |>
  fit(auto_origin)

# Each coefficient with its 95% confidence interval
tidy(tm_origin, conf.int = TRUE)

# A tibble: 3 x 7
  term          estimate std.error statistic  p.value  conf.low  conf.high
<chr>          <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 (Intercept)    20.0      0.409     49.0 1.38e-168  19.2      20.8
2 origin_European  7.57     0.877     8.63 1.54e- 16   5.85      9.29
3 origin_Japanese 10.4     0.828    12.6 1.02e- 30   8.79     12.0
```

6.5.40 R (base)

```
fit_origin <- lm(mpg ~ factor(origin), data = auto)
# The table includes each coefficient's 95% confidence interval
parameters::parameters(fit_origin) |>
  parameters::print_md(digits = 4)
```

Parameter	Coefficient	SE	95% CI	t(389)	p
(Intercept)	20.0335	0.4086	(19.2300, 20.8369)	49.0247	< .001
origin (2)	7.5695	0.8767	(5.8458, 9.2932)	8.6339	< .001
origin (3)	10.4172	0.8276	(8.7901, 12.0442)	12.5878	< .001

```
coef_ci_origin <- py$coef_ci(py$beta_origin, py$Phi_origin)
```

1. The intercept is 20.0 (95% confidence interval 19.2 to 20.8), the average mpg of American cars, which have both indicators equal to 0. The European coefficient is 7.6 (5.8 to 9.3), the gap between the European and American averages, and the Japanese coefficient is 10.4 (8.8 to 12.0), the gap between the Japanese and American averages.

Exercise 6.23 (Lab: use a category as a feature (part 2)). Use the setup of Exercise 6.22.

Draw side-by-side box plots of mpg for the three groups. Compute the sample mean^a mpg of each group. How do the sample means relate to the coefficients?

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Solution

Solution 6.23.

6.5.41 Python (statsmodels)

```
print(mpg.groupby(place).mean())
```

```
origin
American    20.033469
European    27.602941
Japanese    30.450633
Name: mpg, dtype: float64
```

```
# The three group averages, each with a 95% confidence interval
group_rows = pd.DataFrame({"const": 1.0, "European": [0.0, 1.0, 0.0],
                           "Japanese": [0.0, 0.0, 1.0]})
print(auto_origin.get_prediction(group_rows).summary_frame(alpha=0.05)
      [["mean", "mean_ci_lower", "mean_ci_upper"]
       .set_axis(["American", "European", "Japanese"])])
```

	mean	mean_ci_lower	mean_ci_upper
American	20.033469	19.230049	20.836890
European	27.602941	26.077936	29.127947
Japanese	30.450633	29.035777	31.865489

6.5.42 Python (NumPy)

```
print(Auto.groupby("origin")["mpg"].mean())
```

```
origin
1    20.033469
2    27.602941
3    30.450633
Name: mpg, dtype: float64
```

```
# The three group averages, each with a 95% confidence interval
Phi_groups = np.array([[1, 0, 0], [1, 1, 0], [1, 0, 1]], float)
group_means = Phi_groups @ beta_origin
group_lower, group_upper = mean_ci(beta_origin, Phi_origin, Phi_groups)
print(np.column_stack([group_means, group_lower, group_upper]))
```

```
[[20.03346939 19.23004902 20.83688976]
 [27.60294118 26.07793562 29.12794673]
 [30.45063291 29.0357766  31.86548922]]
```

```

groups = [Auto.loc[Auto["origin"] == k, "mpg"] for k in (1, 2, 3)]
positions = [1, 2, 3]
fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
_ = ax.boxplot(groups, tick_labels=["American", "European", "Japanese"])
# The averages and their intervals come from the model with the two indicator columns
ax.vlines(positions, group_lower, group_upper, color="C3", linewidth=3);
ax.plot(positions, group_means, "o", color="C3");
ax.set_ylabel("Miles per gallon");
plt.show()

```

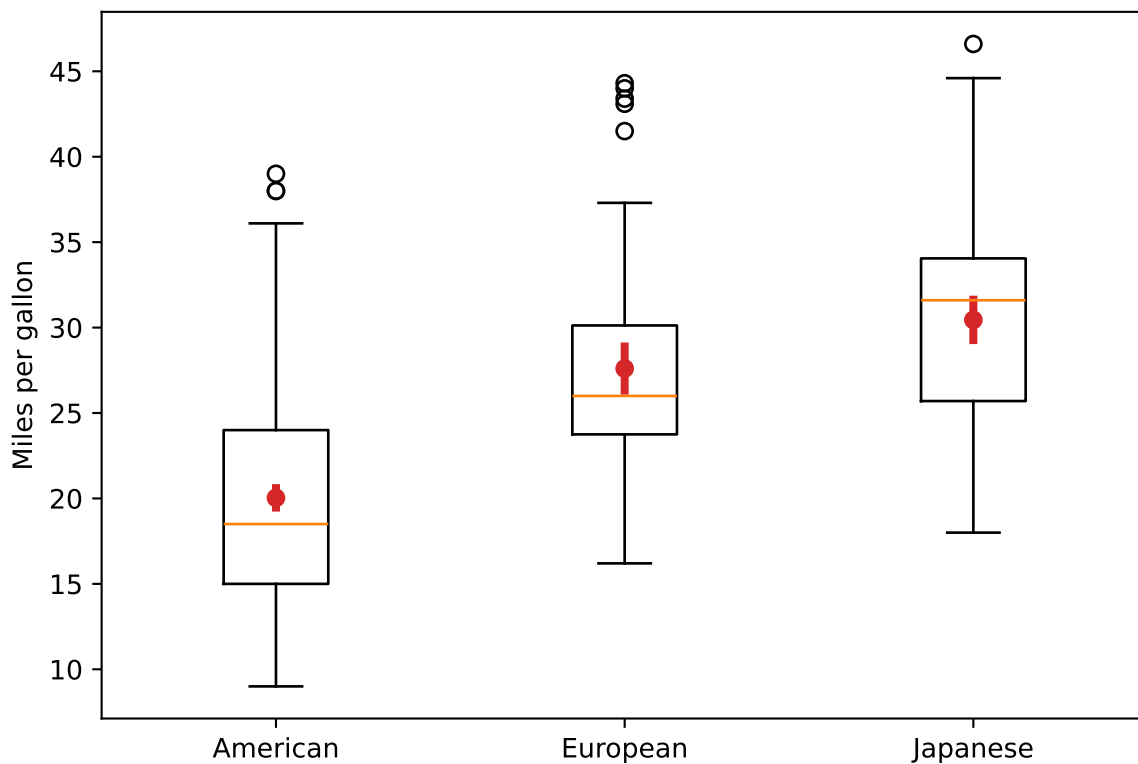


Figure 20: Fuel economy by place of manufacture in the Auto data, with each group's average and its 95% confidence interval in red.

6.5.43 R (tidymodels)

```

auto_origin |> group_by(origin) |> summarise(mean_mpg = mean(mpg))
# A tibble: 3 x 2
  origin mean_mpg
  <fct>    <dbl>
1 American    20.0
2 European    27.6
3 Japanese    30.5

```

```
# The three group averages, each with a 95% confidence interval
origin_groups <- tibble(origin = factor(c("American", "European", "Japanese"),
                                         levels = levels(auto_origin$origin)))

group_means <- bind_cols(origin_groups,
                          predict(tm_origin, new_data = origin_groups),
                          predict(tm_origin, new_data = origin_groups,
                                type = "conf_int"))

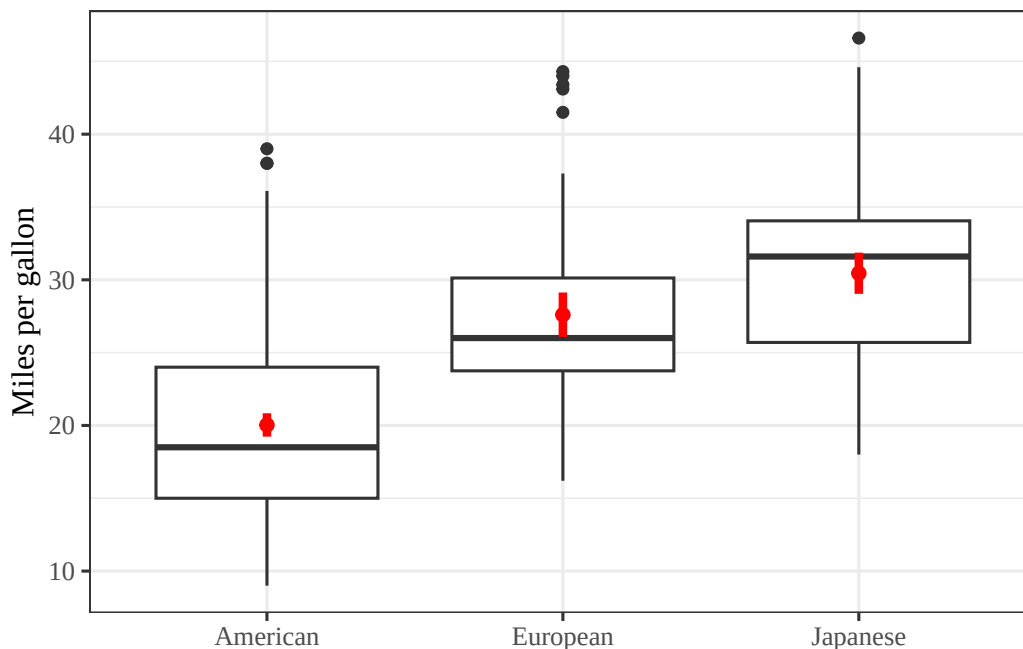
group_means

# A tibble: 3 x 4
  origin   .pred .pred_lower .pred_upper
  <fct>   <dbl>   <dbl>     <dbl>
1 American 20.0     19.2      20.8
2 European 27.6     26.1      29.1
3 Japanese 30.5     29.0      31.9

library(tidymodels)

# One line from the lower to the upper end of each group's interval
interval_ends <- bind_rows(transmute(group_means, origin, mpg = .pred_lower),
                           transmute(group_means, origin, mpg = .pred_upper))

ggplot() +
  geom_boxplot(data = auto_origin, aes(origin, mpg)) +
  geom_line(data = interval_ends, aes(origin, mpg, group = origin),
           colour = "red", linewidth = 1.5) +
  geom_point(data = group_means, aes(origin, .pred), colour = "red", size = 2) +
  labs(x = NULL, y = "Miles per gallon")
```



6.5.44 R (base)

```
tapply(auto$mpg, auto$origin, mean)
```

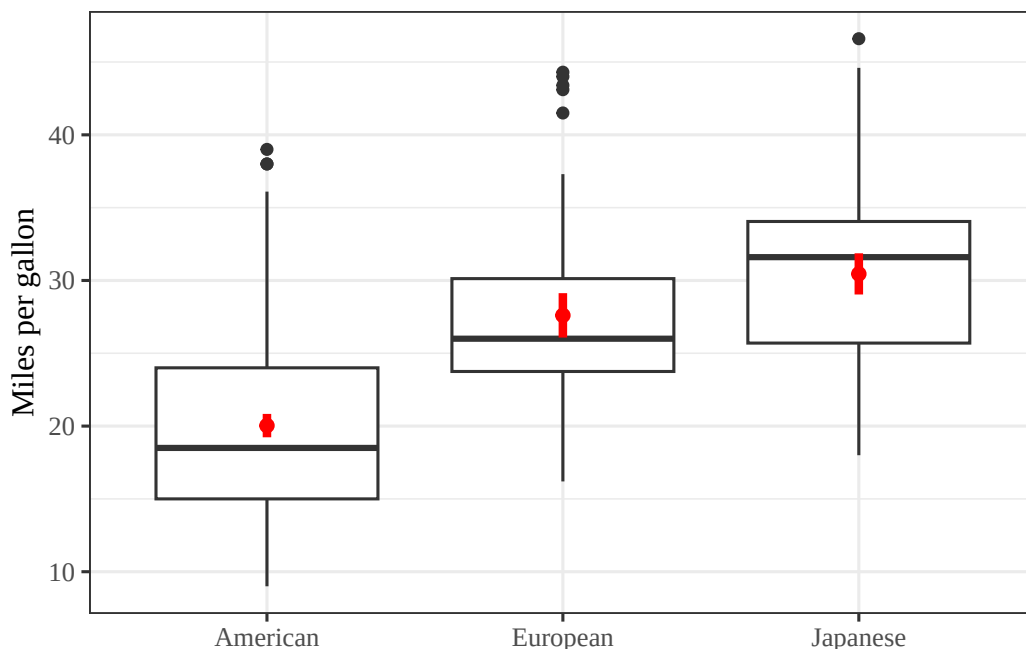
```
      1      2      3
20.03347 27.60294 30.45063
```

```
# The three group averages, each with a 95% confidence interval
origin_groups <- tibble::tibble(origin = 1:3)
group_means_base <- predict(fit_origin, origin_groups, interval = "confidence")
rownames(group_means_base) <- c("American", "European", "Japanese")
group_means_base

      fit      lwr      upr
American 20.03347 19.23005 20.83689
European 27.60294 26.07794 29.12795
Japanese 30.45063 29.03578 31.86549

library(ggplot2)

origin_labels <- c("American", "European", "Japanese")
group_means_plot <- data.frame(
  origin = factor(origin_labels, levels = origin_labels),
  group_means_base
)
# One line from the lower to the upper end of each group's interval
interval_ends <- rbind(
  data.frame(origin = group_means_plot$origin, mpg = group_means_plot$lwr),
  data.frame(origin = group_means_plot$origin, mpg = group_means_plot$upr)
)
ggplot() +
  geom_boxplot(data = auto, aes(factor(origin, labels = origin_labels), mpg)) +
  geom_line(data = interval_ends, aes(origin, mpg, group = origin),
    colour = "red", linewidth = 1.5) +
  geom_point(data = group_means_plot, aes(origin, fit),
    colour = "red", size = 2) +
  labs(x = NULL, y = "Miles per gallon")
```



2. With only indicator columns, least squares reproduces the three group averages. The red intervals in the figure are the 95% confidence intervals^a of those averages, and each coefficient's interval is its t interval, the estimate^b plus or minus a t quantile times its standard error^c.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^c<https://morrison-lab.github.io/sds/estimation.html#def-SE>

Exercise 6.24 (Lab: use a category as a feature (part 3)). Use the setup of Exercise 6.22. Add weight as a third column. What happens to the two origin coefficients, and why?

Solution

Solution 6.24.

6.5.45 Python (statsmodels)

```
X_origin_weight = sm.add_constant(pd.concat([Auto["weight"], origin_columns], axis=1))
auto_origin_weight = sm.OLS(mpg, X_origin_weight).fit()
print(pd.concat([auto_origin_weight.params, auto_origin_weight.conf_int()], axis=1)
      .set_axis(["coef", "lower", "upper"], axis=1))
```

	coef	lower	upper
const	43.732236	41.543128	45.921345
weight	-0.007027	-0.007656	-0.006398
European	0.970906	-0.324295	2.266106
Japanese	2.327150	1.020080	3.634220

6.5.46 Python (NumPy)

```
beta_origin_weight, Phi_origin_weight = fit([weight, european, japanese])
print(np.column_stack([beta_origin_weight,
                       coef_ci(beta_origin_weight, Phi_origin_weight)]))
```

```
[[ 4.37322362e+01  4.15431276e+01  4.59213447e+01]
 [-7.02708331e-03 -7.65634196e-03 -6.39782466e-03]
 [ 9.70905585e-01 -3.24294839e-01  2.26610601e+00]
 [ 2.32714990e+00  1.02008022e+00  3.63421958e+00]]
```

6.5.47 R (tidymodels)

```
origin_weight_recipe <- recipe(mpg ~ weight + origin, data = auto_origin) |>
  step_dummy(origin)
tm_origin_weight <- workflow() |>
  add_recipe(origin_weight_recipe) |>
  add_model(lm_spec) |>
  fit(auto_origin)
tidy(tm_origin_weight, conf.int = TRUE)
```

```
# A tibble: 4 x 7
  term          estimate std.error statistic    p.value conf.low conf.high
<chr>          <dbl>     <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1 (Intercept)   43.7       1.11      39.3 2.89e-137 41.5      45.9
2 weight       -0.00703   0.000320  -22.0 4.96e- 70 -0.00766 -0.00640
3 origin_European 0.971     0.659      1.47 1.41e- 1 -0.324    2.27
4 origin_Japanese 2.33      0.665      3.50 5.18e- 4 1.02      3.63
```

6.5.48 R (base)

```
fit_origin_weight <- lm(mpg ~ weight + factor(origin), data = auto)
parameters::parameters(fit_origin_weight) |>
  parameters::print_md(digits = 4)
```

Parameter	Coefficient	SE	95% CI	t(388)	p
(Intercept)	43.7322	1.1134	(41.5431, 45.9213)	39.2771	< .001
weight	-0.0070	0.0003	(-0.0077, -0.0064)	-21.9559	< .001
origin (2)	0.9709	0.6588	(-0.3243, 2.2661)	1.4738	0.141
origin (3)	2.3271	0.6648	(1.0201, 3.6342)	3.5005	< .001

```
coef_ci_origin_weight <- py$coef_ci(py$beta_origin_weight, py$Phi_origin_weight)
```

3. With `weight` in the model, the origin coefficients become 1.0 (95% confidence interval^a -0.3 to 2.3) and 2.3 (1.0 to 3.6). They are now the gaps between cars of the same weight. American cars are heavier on average, so much of the raw gap was a weight gap, and it shrinks once weight is held fixed.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

6.6 Interaction terms

Exercise 6.25 (Does the horsepower slope depend on weight?). In the model of Exercise 6.12, how much does the predicted `mpg` change per extra horsepower for a light car (2000 pounds), and for a heavy car (4000 pounds)?

Solution

Solution 6.25. The fitted model is $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 \text{hp} + \hat{\beta}_2 w$, where `hp` is horsepower and `w` is weight. Differentiating in `hp` gives $\hat{\beta}_1$, which does not involve `w`. So the change is -0.047 miles per gallon per horsepower for the light car and for the heavy car alike.

The same holds for every pair of features in a model that is a sum of one-feature terms: the effect of one feature on the prediction is the same at every value of the others. James et al. [32, Sec. 3.3.2], p. 94 calls this the *additive* assumption. To let the horsepower slope change with weight, the design matrix needs a column that combines the two features.

Definition 6.7 (Interaction term (product term, cross term)). An **interaction term** (also called a **product term** or a **cross term**) between two features x_1 and x_2 is their product $x_1 x_2$, used as an extra column of the design matrix [33]. With an interaction term, the feature map of Definition 6.4 is $\phi(\tilde{x}) = [1, x_1, x_2, x_1 x_2]^\top$, and the model is

$$h(\tilde{x}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2.$$

The slope of h in x_1 , with x_2 held fixed, is

$$\frac{\partial h}{\partial x_1} = \beta_1 + \beta_3 x_2,$$

so it changes with x_2 . In the same way, the slope in x_2 is $\beta_2 + \beta_3 x_1$. If $\beta_3 = 0$, the model is additive, as in Solution 6.25.

i Other names for interactions

Marketing calls an interaction a *synergy* effect [32, Sec. 3.3.2], p. 95, and epidemiology calls it *effect modification*. These notes say *interaction*.

In R's model formulas, `x1:x2` is the product column alone, and `x1 * x2` is short for `x1 + x2 + x1:x2`.

Example 6.10 (Weight matters more in newer cars). The Auto data record each car's model year, from 1970 to 1982, as 70 to 82. Fit mpg on weight, year and their interaction:

6.6.1 Python

```
year = Auto["year"].to_numpy(float)
beta_weight_year, Phi_weight_year = fit([weight, year, weight * year])
print(beta_weight_year)

[-1.10451919e+02  2.75465227e-02  2.04047640e+00 -4.57932342e-04]

def weight_slope(model_year):
    """Change in predicted mpg per 1000 pounds, in a given model year."""
    return 1000 * (beta_weight_year[1] + beta_weight_year[3] * model_year)

print(weight_slope(70), weight_slope(82))

-4.508741178179798 -10.003929277689418

from scipy import stats

# The covariance of the coefficients is sigma^2 (Phi'Phi)^-1.
# Phi'Phi = R'R for the QR factor R of Phi, so (Phi'Phi)^-1 = R^-1 R^-T,
# which is more accurate than inverting Phi'Phi, since the weight-by-year column is large.
df_wy = n - Phi_weight_year.shape[1]
rse_wy = np.sqrt(np.sum((y - Phi_weight_year @ beta_weight_year) ** 2) / df_wy)
R_inv = np.linalg.inv(np.linalg.qr(Phi_weight_year, mode="r"))
cov_wy = rse_wy ** 2 * R_inv @ R_inv.T
t_wy = stats.t.ppf(0.975, df=df_wy)

def combination_ci(a):
    """95% confidence interval for a'beta: the estimate plus or minus t standard errors."""
    half = t_wy * np.sqrt(a @ cov_wy @ a)
    return np.array([a @ beta_weight_year - half, a @ beta_weight_year + half])

ci_interaction = combination_ci(np.array([0, 0, 0, 1.0]))
ci_slope_70 = combination_ci(1000 * np.array([0, 1, 0, 70.0]))
ci_slope_82 = combination_ci(1000 * np.array([0, 1, 0, 82.0]))
print(ci_interaction, ci_slope_70, ci_slope_82)

[-0.00057407 -0.00034179] [-5.1754346 -3.84204775] [-10.94509211 -9.06276645]

# The ratio of the two slopes is nonlinear in the coefficients, so bootstrap it:
# resample cars with replacement, refit, and recompute the ratio each time.
rng_wy = np.random.default_rng(571)
ratio_wy = np.empty(2000)
for b in range(2000):
    rows = rng_wy.integers(0, n, n)
    beta_b = np.linalg.lstsq(Phi_weight_year[rows], y[rows], rcond=None)[0]
    ratio_wy[b] = (beta_b[1] + 82 * beta_b[3]) / (beta_b[1] + 70 * beta_b[3])
ratio_ci_wy = np.quantile(ratio_wy, [0.025, 0.975])
print(ratio_ci_wy)
```

```
[1.81334968 2.77069719]
```

```
# A function, so the plotting calls' return values are not printed.
def plot_weight_year():
    grid = np.linspace(weight.min(), weight.max(), 100)
    fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
    ax.scatter(weight, y, s=12, alpha=0.4, color="grey");
    for model_year, colour in [(70, "C0"), (82, "C3")]:
        # The design matrix of the fitted line for this model year.
        Phi_grid = np.column_stack([np.ones_like(grid), grid,
                                     np.full_like(grid, model_year), grid * model_year])
        line = Phi_grid @ beta_weight_year
        # Half-width of the 95% confidence band: t * sqrt(phi' cov phi) at each weight.
        half = t_wy * np.sqrt(np.sum((Phi_grid @ cov_wy) * Phi_grid, axis=1))
        ax.fill_between(grid, line - half, line + half, color=colour, alpha=0.3);
        ax.plot(grid, line, color=colour, label=f"19{model_year}");
    ax.set_xlabel("Weight (pounds)");
    ax.set_ylabel("Miles per gallon");
    fig.legend(title="Fitted line", loc="outside lower center", ncol=2);
    plt.show()
```

```
plot_weight_year()
```

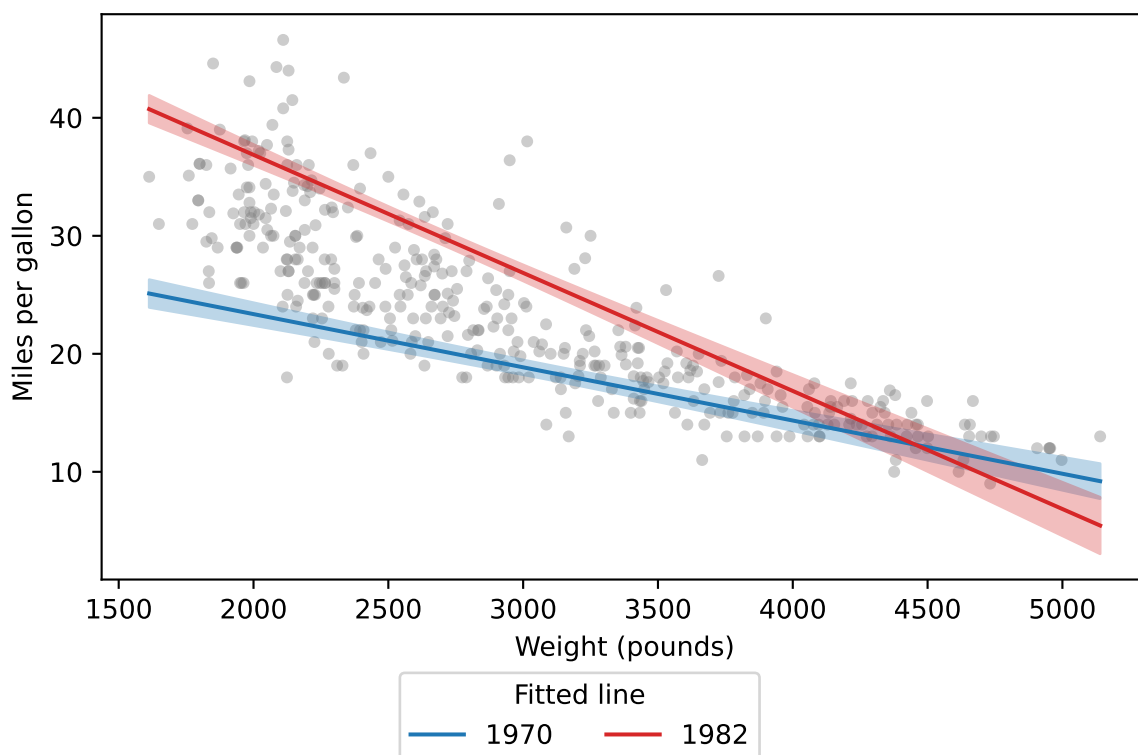


Figure 21: Fuel economy against weight for the Auto data, with the fitted lines of the interaction model for model years 1970 and 1982. Each shaded ribbon is the 95% confidence band for the mean fuel economy of cars of that model year.

6.6.2 R

```
fit_weight_year <- lm(mpg ~ weight * year, data = auto)
parameters::parameters(fit_weight_year) |>
  parameters::print_md(digits = 4)
```

Parameter	Coefficient	SE	95% CI	t(388)	p
(Intercept)	-110.4519	12.9470	(-135.9069, -84.9969)	-8.5311	< .001
weight	0.0275	0.0044	(0.0189, 0.0362)	6.2415	< .001
year	2.0405	0.1718	(1.7027, 2.3783)	11.8756	< .001
weight × year	-0.0005	5.9071e-05	(-0.0006, -0.0003)	-7.7522	< .001

```

weight_slope <- function(model_year) {
  b <- coef(fit_weight_year)
  1000 * (b[["weight"]] + b[["weight:year"]] * model_year)
}
c(weight_slope(70), weight_slope(82))

[1] -4.508741 -10.003929

# 95% confidence interval for the weight slope in a given model year:
# the slope is the linear combination a'b, with variance a' vcov a.
slope_ci <- function(model_year) {
  a <- 1000 * c(0, 1, 0, model_year)
  half <- qt(0.975, df.residual(fit_weight_year)) *
    sqrt(drop(t(a) %*% vcov(fit_weight_year) %*% a))
  sum(a * coef(fit_weight_year)) + c(-1, 1) * half
}
ci_interaction <- confint(fit_weight_year)[["weight:year", ]
rbind(interaction = ci_interaction,
      slope_1970 = slope_ci(70), slope_1982 = slope_ci(82))

              2.5 %      97.5 %
interaction -5.740726e-04 -0.0003417921
slope_1970  -5.175435e+00 -3.8420477515
slope_1982  -1.094509e+01 -9.0627664467

# The ratio of the two slopes is nonlinear in the coefficients, so bootstrap it:
# resample cars with replacement, refit, and recompute the ratio each time.
set.seed(571)
ratio_wy <- replicate(2000, {
  resampled <- auto[sample(nrow(auto), replace = TRUE), ]
  b <- coef(lm(mpg ~ weight * year, data = resampled))
  slope_82 <- b[["weight"]] + 82 * b[["weight:year"]]
  slope_70 <- b[["weight"]] + 70 * b[["weight:year"]]
  slope_82 / slope_70
})
ratio_ci_wy <- quantile(ratio_wy, c(0.025, 0.975))
ratio_ci_wy

      2.5%      97.5%
1.814234 2.777316

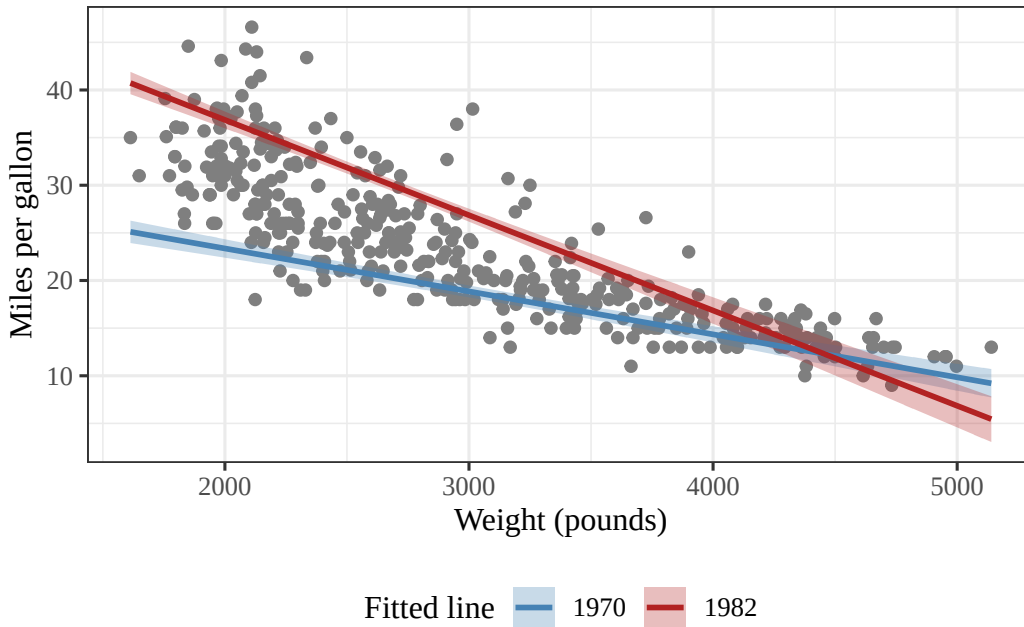
```

```
library(ggplot2)

weight_year_grid <- expand_grid(
  weight = seq(min(auto$weight), max(auto$weight), length.out = 100),
  year = c(70, 82)
)

# predict() returns the fit and the ends of its 95% confidence band.
band <- predict(fit_weight_year, weight_year_grid, interval = "confidence")
weight_year_grid <- cbind(weight_year_grid, band)

ggplot(auto, aes(weight, mpg)) +
  geom_point(colour = "grey50") +
  geom_ribbon(data = weight_year_grid,
            aes(y = NULL, ymin = lwr, ymax = upr, fill = factor(year + 1900)),
            alpha = 0.3) +
  geom_line(data = weight_year_grid,
            aes(y = fit, colour = factor(year + 1900)), linewidth = 1) +
  scale_colour_manual(values = c("steelblue", "firebrick")) +
  scale_fill_manual(values = c("steelblue", "firebrick")) +
  labs(x = "Weight (pounds)", y = "Miles per gallon",
       colour = "Fitted line", fill = "Fitted line")
```



The interaction coefficient is $\hat{\beta}_3 = -4.6 \times 10^{-4}$, with a 95% confidence interval from -5.7×10^{-4} to -3.4×10^{-4} . By Definition 6.7, the slope in weight is $\hat{\beta}_1 + \hat{\beta}_3 \cdot \text{year}$. Per 1000 pounds, it is -4.5 miles per gallon for a 1970 car (interval -5.2 to -3.8) and -10.0 for a 1982 car (interval -10.9 to -9.1). The intervals for the coefficient and the slopes are model-based t intervals. In the fitted model, the estimated slope in weight for 1982 cars is about 2.2 times the estimated slope for 1970 cars (both slopes are negative), with a 95% bootstrap percentile interval for the ratio from 1.8 to 2.8. All of these intervals describe which cars were drawn, and assume the model's form is right.

i Remark (The hierarchy principle)

Remark 6.6 (The hierarchy principle). When a model includes an interaction term $x_1 x_2$, keep the main effects x_1 and x_2 as well. The fitted model then stays the same when either feature is shifted by a constant.

Exercise 6.26 (Lab: let the effect of one feature depend on another). Add the product horsepower $\times$ weight as a third column of the design matrix of Exercise 6.12. This is a feature expansion, like those of [Non-linear regression by feature expansion](#). Fit the model, and report the sMSE and R^2 .

Solution

Solution 6.26.

6.6.3 Python (statsmodels)

```
import statsmodels.api as sm

X_auto_interaction = sm.add_constant(Auto[["horsepower", "weight"]])
X_auto_interaction["horsepower:weight"] = Auto["horsepower"] * Auto["weight"]
auto_interaction = sm.OLS(mpg, X_auto_interaction).fit()

# Each coefficient with its 95% confidence interval
print(pd.concat([auto_interaction.params, auto_interaction.conf_int()], axis=1)
      .set_axis(["coef", "lower", "upper"], axis=1))
```

	coef	lower	upper
const	63.557940	58.951474	68.164406
horsepower	-0.250838	-0.304472	-0.197205
weight	-0.010772	-0.012294	-0.009251
horsepower:weight	0.000054	0.000040	0.000067

```
print(np.sqrt(auto_interaction.ssr / auto_interaction.nobs)) # training sRMSE
```

```
3.9097019687062478
```

```
print(auto_interaction.rsquared)
```

```
0.7484351450253209
```

```
print(bootstrap_rsquared_ci(X_auto_interaction))
```

```
[0.7075474  0.78808742]
```

6.6.4 Python (NumPy)

```
beta_interaction, Phi_interaction = fit([horsepower, weight, horsepower * weight])
print(np.column_stack([beta_interaction, coef_ci(beta_interaction, Phi_interaction)]))
```

```
[[ 6.35579400e+01  5.89514745e+01  6.81644055e+01]
 [-2.50838171e-01 -3.04471555e-01 -1.97204787e-01]
 [-1.07724112e-02 -1.22937951e-02 -9.25102740e-03]
 [ 5.35537776e-05  4.04809307e-05  6.66266245e-05]]
```

```
print(rmse(beta_interaction, Phi_interaction))
```

```
3.9097019687062478
```

```
print(r_squared(beta_interaction, Phi_interaction))
```

```
0.7484351450253209
```

```
r2_ci_interaction = bootstrap_r2_ci(Phi_interaction)
print(r2_ci_interaction)
```

```
[0.7075474  0.78808742]
```

6.6.5 R (tidymodels)

```
library(tidymodels)

interaction_recipe_auto <- recipe(mpg ~ horsepower + weight, data = auto) |>
  step_interact(~ horsepower:weight)
tm_interaction <- workflow() |>
  add_recipe(interaction_recipe_auto) |>
  add_model(lm_spec) |>
  fit(auto)

# Each coefficient with its 95% confidence interval
tidy(tm_interaction, conf.int = TRUE)

# A tibble: 4 x 7
  term                estimate std.error statistic  p.value conf.low conf.high
<chr>                <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 (Intercept)        63.6      2.34e+0   27.1  1.23e-91  5.90e+1  6.82e+1
2 horsepower         -0.251    2.73e-2   -9.20  2.32e-18 -3.04e-1 -1.97e-1
3 weight             -0.0108    7.74e-4  -13.9  5.14e-36 -1.23e-2 -9.25e-3
4 horsepower_x_weight 0.0000536  6.65e-6    8.05  9.93e-15  4.05e-5  6.66e-5

auto_metrics(tm_interaction)

# A tibble: 1 x 2
  rmse r_squared
<dbl> <dbl>
1  3.91  0.748

boot_r2_ci(mpg ~ horsepower * weight, auto)

[1] 0.7083071 0.7911274
```

6.6.6 R (base)

```
fit_interaction <- lm(mpg ~ horsepower * weight, data = auto)

# Each coefficient with its 95% confidence interval
parameters::parameters(fit_interaction) |>
  parameters::print_md(digits = 4)
```

Parameter	Coefficient	SE	95% CI	t(388)	p
(Intercept)	63.5579	2.3429	(58.9515, 68.1644)	27.1273	< .001
horsepower	-0.2508	0.0273	(-0.3045, -0.1972)	-9.1952	< .001
weight	-0.0108	0.0008	(-0.0123, -0.0093)	-13.9213	< .001
horsepower × weight	5.3554e-05	6.6491e-06	(4.0481e-05, 6.6627e-05)	8.0542	< .001

```
c(rmse = rmse(fit_interaction), r_squared = r_squared(fit_interaction))
```

```
rmse r_squared
3.9097020 0.7484351
```

```
boot_r2_ci(mpg ~ horsepower * weight, auto)
```

```
[1] 0.7083071 0.7911274
```

The sample root mean squared error (sRMSE) is 3.9 miles per gallon and $R^2 = 0.75$ (95% bootstrap interval 0.71 to 0.79, from 2,000 refitted resamples of the cars), better than the two-feature fit on the training data. As before, the training sRMSE has no interval.

Exercise 6.27 (Lab: let the effect of one feature depend on another (part 2)). Use the setup of Exercise 6.26.

For the fitted model, what is the change in mpg per extra horsepower for a car of weight w ?

Solution

Solution 6.27. The fitted model is $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 \text{hp} + \hat{\beta}_2 w + \hat{\beta}_3 \text{hp} \cdot w$. Differentiating in hp gives a change of $\hat{\beta}_1 + \hat{\beta}_3 w$ per extra horsepower.

Exercise 6.28 (Lab: let the effect of one feature depend on another (part 3)). Use the setup of Exercise 6.26.

Evaluate that change for a light car (2000 pounds) and a heavy car (4000 pounds).

Solution

Solution 6.28.

6.6.7 Python (statsmodels)

```
# The change per horsepower, beta_1 + beta_3 w, is a linear combination of the coefficients,
# so t_test gives its 95% confidence interval from the coefficients' covariance matrix
for w in [2000, 4000]:
    change = auto_interaction.t_test(np.array([0, 1, 0, w]))
    print(w, change.effect[0], change.conf_int(alpha=0.05)[0])

2000 -0.1437306158206603 [-0.17474791 -0.11271332]
4000 -0.03662306065352933 [-0.05698969 -0.01625644]
```

6.6.8 Python (NumPy)

```
def contrast_ci(beta_hat, Phi, c):
    """95% interval for the linear combination c'beta of the coefficients."""
    df, sigma2, PtP_inv = ols_summary(beta_hat, Phi)
    half = stats.t.ppf(0.975, df) * np.sqrt(sigma2 * c @ PtP_inv @ c)
    return c @ beta_hat - half, c @ beta_hat + half

# The change per horsepower, beta_1 + beta_3 w, is c'beta for c = (0, 1, 0, w)
weights = [2000, 4000]
change_ci = [contrast_ci(beta_interaction, Phi_interaction, np.array([0, 1, 0, w]))
              for w in weights]
for w, interval in zip(weights, change_ci):
    print(w, beta_interaction[1] + beta_interaction[3] * w, interval)
```

```
2000 -0.14373061582060576 (np.float64(-0.17474790679968843), np.float64(-
0.11271332484152308))
4000 -0.03662306065342058 (np.float64(-0.05698968503453583), np.float64(-
0.016256436272305334))
```

6.6.9 R (tidymodels)

```
library(tidymodels)

# lincomb_ci (in the interval helpers) gives the 95% interval
# for a linear combination w'beta of the coefficients

# The change per horsepower, beta_1 + beta_3 w, uses the weights (0, 1, 0, w)
engine_interaction <- extract_fit_engine(tm_interaction)
map(c(2000, 4000), \(w) {
  bind_cols(weight = w, as_tibble(as.list(
    lincomb_ci(engine_interaction, c(0, 1, 0, w))
  )))
}) |>
list_rbind()

# A tibble: 2 x 4
  weight estimate lower upper
  <dbl>   <dbl> <dbl>   <dbl>
1  2000  -0.144 -0.175 -0.113
2  4000  -0.0366 -0.0570 -0.0163
```

6.6.10 R (base)

```
# lincomb_ci (in the interval helpers) gives the 95% interval
# for a linear combination w'beta of the coefficients

# The change per horsepower, beta_1 + beta_3 w, uses the weights (0, 1, 0, w)
t(sapply(c(2000, 4000), \(w) {
  c(weight = w, lincomb_ci(fit_interaction, c(0, 1, 0, w)))
}))
```

	weight	estimate	lower	upper
[1,]	2000	-0.14373062	-0.17474791	-0.11271332
[2,]	4000	-0.03662306	-0.05698969	-0.01625644

At 2000 pounds the change is -0.144 (95% confidence interval^a -0.175 to -0.113), and at 4000 pounds it is -0.037 (-0.057 to -0.016). Each is a 95% confidence interval for a linear combination^b of the coefficients, built from the estimated^c covariance matrix^d of the coefficients. The estimated slope^e in horsepower is more negative for light cars than for heavy ones. The model is still linear in β , so the normal equations of [The normal equations](#) still apply.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-linear-combination>

^c<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^d<https://morrison-lab.github.io/pds/variance-covariance.html#def-cov-vec-x>

^e<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Exercise 6.29 (Lab: let the weight slope depend on a category). Let `american` be the indicator that is 1 for American cars (`origin = 1`) and 0 for the rest. Fit `mpg` on `weight`, `american` and their interaction `weight × american`. What is the fitted change in `mpg` per 1000 pounds for American cars, and for other cars?

Solution

Solution 6.29.

6.6.11 Python (statsmodels)

```
import statsmodels.api as sm

X_american = sm.add_constant(Auto[["weight"]])
X_american["american"] = (Auto["origin"] == 1).astype(float)
X_american["weight:american"] = X_american["weight"] * X_american["american"]
auto_american = sm.OLS(mpg, X_american).fit()

# Each coefficient with its 95% confidence interval
print(pd.concat([auto_american.params, auto_american.conf_int()], axis=1)
      .set_axis(["coef", "lower", "upper"], axis=1))
```

	coef	lower	upper
const	48.771246	44.871650	52.670842
weight	-0.008467	-0.010121	-0.006812
american	-5.622778	-10.173892	-1.071663
weight:american	0.001613	-0.000175	0.003400

```
# Per 1000 pounds, the slope is a linear combination of the coefficients,
# so t_test gives its 95% confidence interval
for label, weights_1000 in [("other", [0, 1000, 0, 0]), ("american", [0, 1000, 0, 1000])]:
    change = auto_american.t_test(np.array(weights_1000))
    print(label, change.effect[0], change.conf_int(alpha=0.05)[0])
```

other	-8.46677756922959	[-10.12122921 -6.81232592]
american	-6.853986380923989	[-7.53124225 -6.17673051]

6.6.12 Python (NumPy)

```
american = (origin == 1).astype(float)
beta_american, Phi_american = fit([weight, american, weight * american])
print(np.column_stack([beta_american, coef_ci(beta_american, Phi_american)]))
```

[4.87712463e+01	4.48716503e+01	5.26708424e+01]
[-8.46677757e-03	-1.01212292e-02	-6.81232592e-03]
[-5.62277783e+00	-1.01738925e+01	-1.07166317e+00]
[1.61279119e-03	-1.74912859e-04	3.40049524e-03]

```
# Per 1000 pounds, the slope is a linear combination c'beta of the coefficients
c_other = np.array([0, 1000, 0, 0])
c_american = np.array([0, 1000, 0, 1000])
slope_ci_other = contrast_ci(beta_american, Phi_american, c_other)
slope_ci_american = contrast_ci(beta_american, Phi_american, c_american)
print(c_other @ beta_american, slope_ci_other)
```

-8.466777569232235	(np.float64(-10.121229213739927), np.float64(-6.812325924724543))
--------------------	---

```
print(c_american @ beta_american, slope_ci_american)
```

-6.853986380922909	(np.float64(-7.5312422523846685), np.float64(-6.17673050946115))
--------------------	--

6.6.13 R (tidymodels)

```
library(tidymodels)

auto_american <- auto |> mutate(american = as.numeric(origin == 1))
tm_american <- lm_spec |> fit(mpg ~ weight * american, data = auto_american)

# Each coefficient with its 95% confidence interval
tidy(tm_american, conf.int = TRUE)

# A tibble: 4 x 7
  term          estimate std.error statistic  p.value  conf.low conf.high
<chr>          <dbl>    <dbl>    <dbl>   <dbl>    <dbl>    <dbl>
1 (Intercept)    48.8      1.98      24.6 3.71e-81  44.9     52.7
2 weight       -0.00847  0.000841   -10.1 2.58e-21 -0.0101  -0.00681
3 american      -5.62     2.31      -2.43 1.56e- 2 -10.2     -1.07
4 weight:american 0.00161  0.000909    1.77 7.69e- 2 -0.000175  0.00340

# Per 1000 pounds, the slope is a linear combination of the coefficients,
# with the weights (0, 1000, 0, 0) for other cars
# and (0, 1000, 0, 1000) for American cars.
# lincomb_ci() is one of the interval helpers.
engine_american <- extract_fit_engine(tm_american)
bind_rows(
  other = as_tibble(as.list(lincomb_ci(engine_american, c(0, 1000, 0, 0)))),
  american = as_tibble(as.list(
    lincomb_ci(engine_american, c(0, 1000, 0, 1000))
  )),
  .id = "cars"
)

# A tibble: 2 x 4
  cars      estimate lower upper
<chr>      <dbl>  <dbl> <dbl>
1 other      -8.47 -10.1  -6.81
2 american   -6.85  -7.53  -6.18
```

6.6.14 R (base)

```
auto <- dplyr::mutate(auto, american = as.numeric(origin == 1))
fit_american <- lm(mpg ~ weight * american, data = auto)

# Each coefficient with its 95% confidence interval
parameters::parameters(fit_american) |>
  parameters::print_md(digits = 4)
```

Parameter	Coefficient	SE	95% CI	t(388)	p
(Intercept)	48.7712	1.9834	(44.8717, 52.6708)	24.5895	< .001
weight	-0.0085	0.0008	(-0.0101, -0.0068)	-10.0616	< .001
american	-5.6228	2.3148	(-10.1739, -1.0717)	-2.4291	0.016
weight × american	0.0016	0.0009	(-0.0002, 0.0034)	1.7737	0.077

```
# Per 1000 pounds, the slope is a linear combination of the coefficients,
# with the weights (0, 1000, 0, 0) for other cars
# and (0, 1000, 0, 1000) for American cars.
# lincomb_ci() is one of the interval helpers.
```

```
rbind(other = lincomb_ci(fit_american, c(0, 1000, 0, 0)),
      american = lincomb_ci(fit_american, c(0, 1000, 0, 1000)))
```

```
      estimate      lower      upper
other    -8.466778 -10.121229 -6.812326
american -6.853986  -7.531242 -6.176731
```

The model is $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 w + \hat{\beta}_2 a + \hat{\beta}_3 wa$, where w is weight and a is the `american` indicator. For other cars, $a = 0$, and the model is the line $\hat{\beta}_0 + \hat{\beta}_1 w$. For American cars, $a = 1$, and the model is the line $(\hat{\beta}_0 + \hat{\beta}_2) + (\hat{\beta}_1 + \hat{\beta}_3)w$. So the interaction gives each group its own intercept and its own slope.

Per 1000 pounds, the fitted change in `mpg` is -8.5 (95% confidence interval -10.1 to -6.8) for other cars and -6.9 (-7.5 to -6.2) for American cars. Each interval is a t interval for a linear combination^a of the coefficients, built from their estimated covariance matrix. Without the product column, the two lines would be parallel, as in part 3 of Exercise 6.22.

^a<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-linear-combination>

7 Inference for the coefficients

This section follows [32, Sec. 3.1.2], and takes the matrix form of the results from the rme notes on linear models⁴.

7.1 How precise is a coefficient?

Exercise 7.1 (How much would the slope change with new data?). Solution 6.5.4 fitted a slope of -0.158 miles per gallon per horsepower to 392 cars. Suppose we measured 392 different cars of the same kinds and fitted the same model. Would we get the same slope?

Solution

Solution 7.1. No. Each response is $y_i = \phi(\tilde{x}_i) \cdot \vec{\beta} + \varepsilon_i$, and the noise terms ε_i differ from one sample to the next. The estimate $\hat{\vec{\beta}} = (D^\top D)^{-1} D^\top \tilde{y}$ is a function of $\tilde{y}$, so it is a random variable too.

Exercise 7.2 (How much would the slope change with new data? (part 2)). Use the setup of Exercise 7.1. What would describe how much the slope^a changes from one sample of cars to another?

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

⁴<https://morrison-lab.github.io/rme/chapters/Linear-models-overview.html#inference-for-individual-predictor-coefficients>

Solution

Solution 7.2. The standard deviation^a of $\hat{\beta}_1$ over repeated samples^b describes how much the slope^c changes. A small standard deviation shows that one sample pins the slope down well.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-sd>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample>

^c<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

7.2 The covariance of the least squares estimate

Exercise 7.3 (The mean and covariance of the least squares estimate). Suppose $y = D\vec{\beta} + \vec{\varepsilon}$, where D is a fixed $n \times p^*$ design matrix with full column rank, $\mathbb{E}[\vec{\varepsilon}] = \mathbf{0}_{n \times 1}$ and $\text{Cov}(\vec{\varepsilon}) = \sigma^2 \mathbf{I}_n$. Find the mean $\mathbb{E}[\hat{\vec{\beta}}]$ and the covariance matrix^a $\text{Cov}(\hat{\vec{\beta}})$ of the least squares estimate $\hat{\vec{\beta}} = (D^\top D)^{-1} D^\top y$.

Hint

First write $\hat{\vec{\beta}} - \vec{\beta}$ as a matrix times $\vec{\varepsilon}$.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-cov-vec-x>

Solution

Solution 7.3. Let $\mathbf{A} = (D^\top D)^{-1} D^\top$, a fixed $p^* \times n$ matrix. Substitute the model for y :

$$\begin{aligned} \hat{\vec{\beta}} &= \mathbf{A}(D\vec{\beta} + \vec{\varepsilon}) && \text{(substitute } y = D\vec{\beta} + \vec{\varepsilon}) \\ &= (D^\top D)^{-1} D^\top D\vec{\beta} + \mathbf{A}\vec{\varepsilon} && \text{(distribute)} \\ &= \vec{\beta} + \mathbf{A}\vec{\varepsilon} && ((D^\top D)^{-1} D^\top D = \mathbf{I}_{p^*}) \end{aligned}$$

The mean follows from linearity of expectation:

$$\begin{aligned} \mathbb{E}[\hat{\vec{\beta}}] &= \vec{\beta} + \mathbf{A} \mathbb{E}[\vec{\varepsilon}] && \text{(linearity; } \mathbf{A} \text{ is fixed)} \\ &= \vec{\beta} && (\mathbb{E}[\vec{\varepsilon}] = \mathbf{0}) \end{aligned}$$

So $\hat{\vec{\beta}}$ is unbiased. For the covariance, $\hat{\vec{\beta}} - \mathbb{E}[\hat{\vec{\beta}}] = \mathbf{A}\vec{\varepsilon}$:

$$\begin{aligned} \text{Cov}(\hat{\vec{\beta}}) &= \mathbb{E}[(\mathbf{A}\vec{\varepsilon})(\mathbf{A}\vec{\varepsilon})^\top] && \text{(definition of covariance)} \\ &= \mathbb{E}[\mathbf{A}\vec{\varepsilon}\vec{\varepsilon}^\top \mathbf{A}^\top] && ((\mathbf{A}\vec{\varepsilon})^\top = \vec{\varepsilon}^\top \mathbf{A}^\top) \\ &= \mathbf{A} \mathbb{E}[\vec{\varepsilon}\vec{\varepsilon}^\top] \mathbf{A}^\top && \text{(linearity; } \mathbf{A} \text{ is fixed)} \\ &= \mathbf{A} (\sigma^2 \mathbf{I}_n) \mathbf{A}^\top && (\mathbb{E}[\vec{\varepsilon}\vec{\varepsilon}^\top] = \text{Cov}(\vec{\varepsilon}), \text{ since } \mathbb{E}[\vec{\varepsilon}] = \mathbf{0}) \\ &= \sigma^2 (D^\top D)^{-1} D^\top D (D^\top D)^{-1} && (\mathbf{A}^\top = D (D^\top D)^{-1}, \text{ as } D^\top D \text{ is symmetric)} \\ &= \sigma^2 (D^\top D)^{-1} && (D^\top D (D^\top D)^{-1} = \mathbf{I}_{p^*}) \end{aligned}$$

Theorem 7.1 (Mean and covariance of the least squares estimate). If $y = D\vec{\beta} + \vec{\varepsilon}$ with D fixed and of full column rank, $\mathbb{E}[\vec{\varepsilon}] = \mathbf{0}_{n \times 1}$ and $\text{Cov}(\vec{\varepsilon}) = \sigma^2 \mathbf{I}_n$, then

$$\mathbb{E}[\hat{\vec{\beta}}] = \vec{\beta}, \quad \text{Cov}(\hat{\vec{\beta}}) = \sigma^2 \underbrace{(D^\top D)^{-1}}_{p^* \times p^*}.$$

In particular, $\text{Var}(\hat{\beta}_j) = \sigma^2 [(D^\top D)^{-1}]_{jj}$.

i Proof

Proof. By Solution 7.3. □

Example 7.1 (Checking the slope's variance by simulation). Take the design matrix of Solution 6.5.4 (an intercept and **horsepower** for the 392 Auto cars), treat its fitted coefficients as the true $\vec{\beta}$, and take σ to be its residual standard error. Simulate 2000 new response vectors $y = D\vec{\beta} + \vec{\varepsilon}$ with $\vec{\varepsilon} \sim N(\mathbf{0}, \sigma^2 \mathbf{I}_n)$, refit each one, and compare the standard deviation of the 2000 slopes with $\sqrt{\sigma^2 [(D^\top D)^{-1}]_{11}}$ from Theorem 7.1:

7.2.1 Python

```
residuals_simple = y - Phi_simple @ beta_simple
sigma_simple = np.sqrt(np.sum(residuals_simple ** 2) / (n - 2))
PtP_inv = np.linalg.inv(Phi_simple.T @ Phi_simple)
sd_theory = sigma_simple * np.sqrt(PtP_inv[1, 1])

rng = np.random.default_rng(571)
n_sims = 2000
sim_slopes = np.empty(n_sims)
sim_ses = np.empty(n_sims)
for sim in range(n_sims):
    y_sim = Phi_simple @ beta_simple + rng.normal(0, sigma_simple, n)
    b_sim = PtP_inv @ Phi_simple.T @ y_sim
    rse_sim = np.sqrt(np.sum((y_sim - Phi_simple @ b_sim) ** 2) / (n - 2))
    sim_slopes[sim] = b_sim[1]
    sim_ses[sim] = rse_sim * np.sqrt(PtP_inv[1, 1])

from scipy import stats

# The simulated slopes are exactly Gaussian, so (N - 1) s^2 / sd^2 follows
# a chi-squared distribution with N - 1 degrees of freedom;
# inverting that statement gives a 95% interval for the true standard deviation.
sd_sim = np.std(sim_slopes, ddof=1)
sd_sim_ci = sd_sim * np.sqrt((n_sims - 1) / stats.chi2.ppf([0.975, 0.025], df=n_sims - 1))
print(sd_theory, sd_sim, sd_sim_ci)

0.006445500517685036 0.006482351093736395 [0.00628751 0.00668974]
```

7.2.2 R

```

design <- model.matrix(fit_simple)
sigma_simple <- summary(fit_simple)$sigma
xtx_inv <- solve(crossprod(design))
sd_theory <- sigma_simple * sqrt(xtx_inv[2, 2])

set.seed(571)
sim_fits <- replicate(2000, {
  y_sim <- fitted(fit_simple) + rnorm(nrow(design), 0, sigma_simple)
  coef(summary(lm(y_sim ~ design - 1)))[2, c("Estimate", "Std. Error")]
})
sd_sim <- sd(sim_fits["Estimate", ])
n_sims <- ncol(sim_fits)
# The simulated slopes are exactly Gaussian, so (N - 1) s^2 / sd^2 follows
# a chi-squared distribution with N - 1 degrees of freedom;
# inverting that statement gives a 95% interval for the true standard deviation.
chi_sq <- qchisq(c(0.975, 0.025), df = n_sims - 1)
sd_sim_ci <- sd_sim * sqrt((n_sims - 1) / chi_sq)
c(theory = sd_theory, simulated = sd_sim,
  lower = sd_sim_ci[1], upper = sd_sim_ci[2])

```

theory	simulated	lower	upper
0.006445501	0.006367362	0.006175978	0.006571075

The formula gives 0.00645, and the 2000 simulated slopes have standard deviation 0.00648, with a 95% confidence interval from 0.00629 to 0.00669. The interval measures only the simulation error of using 2000 draws, and it uses the exact chi-squared distribution of a Gaussian sample's variance. The formula's value lies inside it, so the two agree to within simulation error.

7.3 Standard errors

Definition 7.1 (Standard error of a coefficient). The **standard error** of $\hat{\beta}_j$ is its standard deviation over repeated samples,

$$\text{SE}(\hat{\beta}_j) = \sigma \sqrt{[(D^\top D)^{-1}]_{jj}},$$

by Theorem 7.1.

Example 7.2 (Standard errors when σ is known). In the simulation of Example 7.1, the noise level σ is known, so we can compute the standard errors of Definition 7.1 exactly:

7.3.1 Python

```

se_known_sigma = sigma_simple * np.sqrt(np.diag(PtP_inv))
print(se_known_sigma)

[0.71749866 0.0064455 ]

```

7.3.2 R

```

se_known_sigma <- sigma_simple * sqrt(diag(xtx_inv))
se_known_sigma

```

```

(Intercept) horsepower
0.717498656 0.006445501

```

The slope's standard error is 0.00645. It is one fixed number for the whole simulation, and it matches the spread of the 2000 simulated slopes, whose standard deviation is 0.00648.

Definition 7.2 (Estimated standard error). In practice the noise level σ in Definition 7.1 is unknown. Replacing it by the residual standard error of Definition 6.3, $\hat{\sigma} = \text{RSE}$, gives the **estimated standard error**

$$\widehat{\text{se}}(\hat{\beta}_j) = \hat{\sigma} \sqrt{[(D^\top D)^{-1}]_{jj}}.$$

Software output, and most books, call $\widehat{\text{se}}(\hat{\beta}_j)$ “the standard error” too [32, Sec. 3.1.2], p. 75.

Example 7.3 (The slope’s estimated standard error for the Auto data). For the one-feature fit of Solution 6.5.4, compute $\widehat{\text{se}}(\hat{\beta}_1)$ from Definition 7.2, and compare it with the standard error in the table from R’s parameters package:

7.3.3 Python

```
se_simple = sigma_simple * np.sqrt(np.diag(PtP_inv))
print(se_simple)
```

```
[0.71749866 0.0064455 ]
```

7.3.4 R

```
se_by_hand <- sigma_simple * sqrt(diag(xtx_inv))
se_by_hand
```

```
(Intercept) horsepower
0.717498656 0.006445501
```

```
parameters::parameters(fit_simple) |>
parameters::print_md(digits = 4)
```

Parameter	Coefficient	SE	95% CI	t(390)	p
(Intercept)	39.9359	0.7175	(38.5252, 41.3465)	55.6598	< .001
horsepower	-0.1578	0.0064	(-0.1705, -0.1452)	-24.4891	< .001

The residual standard error is $\hat{\sigma} = 4.91$, and $\widehat{\text{se}}(\hat{\beta}_1) = 0.00645$, the same value as the standard error column of the parameters table. It also equals the standard error of Example 7.2, because that simulation set σ to this same $\hat{\sigma}$. The slope $\hat{\beta}_1 = -0.158$ is about 24 standard errors away from 0.

Unlike the standard error, the estimate changes from one data set to the next. Across the 2000 simulated data sets of Example 7.1, $\widehat{\text{se}}(\hat{\beta}_1)$ ranges from 0.0057 to 0.00727, around the standard error 0.00645 of Example 7.2.

7.4 The t distribution of a coefficient

Theorem 7.2 (The standardized coefficient has a t distribution). *Under the assumptions of Theorem 7.1, suppose also that the noise is Gaussian, $\vec{\varepsilon} \sim N(\mathbf{0}, \sigma^2 \mathbf{I}_n)$. Then for each coefficient,*

$$T_j = \frac{\hat{\beta}_j - \beta_j}{\widehat{\text{se}}(\hat{\beta}_j)} \sim t_{n-p^*},$$

the t distribution^a with $n - p^*$ degrees of freedom [32, Sec. 3.1.2], p. 77.

^a<https://morrison-lab.github.io/sds/inference.html#def-t-dist>

The proof needs the distribution of $\hat{\sigma}^2$ and its independence from $\hat{\beta}$, which these notes do not cover. James

et al. [32, Sec. 3.1.2] states the result for simple linear regression, and the rme notes⁵ state it for multiple regression.

Example 7.4 (Checking the t distribution by simulation). For each of the 2000 simulated samples of Example 7.1, compute $T_1 = (\hat{\beta}_1 - \beta_1) / \widehat{\text{se}}(\hat{\beta}_1)$. By Theorem 7.2, about 5% of them should land outside $\pm t_{n-2}(0.975)$, the 97.5th percentile of t_{n-2} :

7.4.1 Python

```
from scipy import stats

t_sim = (sim_slopes - beta_simple[1]) / sim_ses
t_crit = stats.t.ppf(0.975, df=n - 2)
n_outside = int(np.sum(np.abs(t_sim) > t_crit))
# An exact (Clopper-Pearson) 95% interval for the proportion outside.
outside_ci = stats.binomtest(n_outside, len(t_sim)).proportion_ci(
    confidence_level=0.95, method="exact")
outside_lo, outside_hi = outside_ci.low, outside_ci.high
print(t_crit, n_outside / len(t_sim), outside_lo, outside_hi)

1.9660653313011052 0.0495 0.04041037792654773 0.05993731615842642
```

7.4.2 R

```
t_sim <- (sim_fits["Estimate", ] - coef(fit_simple)[["horsepower"]]) /
  sim_fits["Std. Error", ]
t_crit <- qt(0.975, df = fit_simple$df.residual)
n_outside <- sum(abs(t_sim) > t_crit)
# binom.test gives an exact (Clopper-Pearson) 95% interval
# for the proportion outside.
outside_ci <- binom.test(n_outside, length(t_sim))$conf.int
c(t_crit = t_crit, outside = n_outside / length(t_sim),
  lower = outside_ci[1], upper = outside_ci[2])
```

```
      t_crit      outside      lower      upper
1.96606533 0.04400000 0.03543624 0.05393046
```

With $n - 2 = 390$ degrees of freedom, $t_{n-2}(0.975) = 1.966$, and 5.0% of the simulated T_1 values land outside ± 1.966 , with an exact 95% confidence interval for that proportion from 4.0% to 6.0%. The interval covers the nominal 5%, and it measures only the simulation error of 2000 draws.

7.5 Confidence intervals

Exercise 7.4 (An interval that catches the true coefficient). Use Theorem 7.2 to find an interval, computed from the data, that contains the true β_j with probability 0.95.

Solution

Solution 7.4. Write $c = t_{n-p^*}(0.975)$ for the 97.5th percentile of t_{n-p^*} . By Theorem 7.2, $T_j \sim t_{n-p^*}$, and the t distribution is symmetric about 0, so

⁵<https://morrison-lab.github.io/rme/chapters/Linear-models-overview.html#wald-tests-and-confidence-intervals>

$$\begin{aligned}
0.95 &= \mathbb{P}(-c \leq T_j \leq c) && \text{(definition of } c; \text{ symmetry)} \\
&= \mathbb{P}\left(-c \leq \frac{\hat{\beta}_j - \beta_j}{\widehat{\text{se}}(\hat{\beta}_j)} \leq c\right) && \text{(definition of } T_j) \\
&= \mathbb{P}(-c \widehat{\text{se}}(\hat{\beta}_j) \leq \hat{\beta}_j - \beta_j \leq c \widehat{\text{se}}(\hat{\beta}_j)) && \text{(multiply by } \widehat{\text{se}}(\hat{\beta}_j) > 0) \\
&= \mathbb{P}(\hat{\beta}_j - c \widehat{\text{se}}(\hat{\beta}_j) \leq \beta_j \leq \hat{\beta}_j + c \widehat{\text{se}}(\hat{\beta}_j)) && \text{(rearrange for } \beta_j)
\end{aligned}$$

So the interval $\hat{\beta}_j \pm c \widehat{\text{se}}(\hat{\beta}_j)$ contains β_j with probability 0.95. The interval is random, and β_j is fixed.

Theorem 7.3 (The coefficient interval has probability 0.95 of containing the coefficient). *Let $c = t_{n-p^*}(0.975)$. Under the assumptions of Theorem 7.2,*

$$\mathbb{P}(\hat{\beta}_j - c \widehat{\text{se}}(\hat{\beta}_j) \leq \beta_j \leq \hat{\beta}_j + c \widehat{\text{se}}(\hat{\beta}_j)) = 0.95.$$

The interval is random, and β_j is fixed.

i Proof

Proof. This is the solution to Exercise 7.4: see Solution 7.4. □

Example 7.5 (Coverage of the coefficient interval in simulation). Simulate 5000 data sets of $n = 30$ points from $y = 1 + 2x + \varepsilon$ with $\varepsilon \sim N(0, 1)$. For each, build the 95% interval for the slope of Definition 7.3 and check whether it contains the true slope 2.

```
# Save and restore the random seed, so this example does not change later random output.
ci_old_seed <- if (exists(".Random.seed", globalenv())) get(".Random.seed", globalenv())
set.seed(571)
ci_covered <- replicate(5000, {
  x <- runif(30)
  y <- 1 + 2 * x + rnorm(30)
  ci <- confint(lm(y ~ x))["x", ]
  ci[1] <= 2 && 2 <= ci[2]
})
if (!is.null(ci_old_seed)) assign(".Random.seed", ci_old_seed, globalenv())
mean(ci_covered)
```

```
[1] 0.9532
```

The intervals contain the true slope in 95.3% of the data sets, close to the stated 95%.

Definition 7.3 (Confidence interval for a coefficient). The 95% confidence interval^a for β_j is

$$\hat{\beta}_j \pm t_{n-p^*}(0.975) \widehat{\text{se}}(\hat{\beta}_j).$$

Over repeated samples, 95% of the intervals built this way contain the true β_j , by Theorem 7.3. For large n , $t_{n-p^*}(0.975)$ is close to 2, and James et al. [32, Sec. 3.1.2], p. 75 writes the interval as $\hat{\beta}_j \pm 2 \widehat{\text{se}}(\hat{\beta}_j)$.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

Example 7.6 (A confidence interval for the horsepower slope). Compute the 95% confidence interval for the slope of Solution 6.5.4 from Definition 7.3, and compare it with the 95% confidence interval columns of the table from R's parameters package:

7.5.1 Python

```
ci_slope = beta_simple[1] + np.array([-1, 1]) * t_crit * se_simple[1]
print(ci_slope)

[-0.17051701 -0.14517246]
```

7.5.2 R

```
coef(fit_simple)[["horsepower"]] + c(-1, 1) * t_crit * se_by_hand[[2]]

[1] -0.1705170 -0.1451725
parameters::parameters(fit_simple) |>
  parameters::print_md(digits = 4, ci_digits = 4)
```

Parameter	Coefficient	SE	95% CI	t(390)	p
(Intercept)	39.9359	0.7175	(38.5252, 41.3465)	55.6598	< .001
horsepower	-0.1578	0.0064	(-0.1705, -0.1452)	-24.4891	< .001

The interval runs from -0.171 to -0.145 miles per gallon per horsepower. Every value in the interval is negative, so the interval excludes a slope of zero.

7.6 Testing whether a coefficient is zero

Definition 7.4 (The t -test for a coefficient). To test the null hypothesis^a $H_0 : \beta_j = 0$ (feature j adds nothing to the model that already contains the other features) against $H_A : \beta_j \neq 0$, compute the t -statistic

$$t_j = \frac{\hat{\beta}_j}{\widehat{\text{se}}(\hat{\beta}_j)}.$$

Under H_0 , t_j has the t_{n-p^*} distribution, by Theorem 7.2. The p -value^b is the probability, under H_0 , of a statistic at least as far from 0:

$$\begin{aligned} p &= \mathbb{P}(|T_{n-p^*}| \geq |t_j|) \\ &= 2 \mathbb{P}(T_{n-p^*} \geq |t_j|). \end{aligned}$$

A small p -value is evidence against H_0 [32, Sec. 3.1.2], p. 77.

^a<https://morrison-lab.github.io/sds/inference.html#def-null-hypothesis>

^b<https://morrison-lab.github.io/sds/inference.html#def-p-value>

Example 7.7 (Does horsepower matter once weight is in the model?). For the two-feature fit of Solution 6.5.24, compute the t -statistic and p -value for **horsepower** from Definition 7.4, and compare them with the table from R's parameters package:

7.6.1 Python

```

p_star = Phi_multiple.shape[1]
rse_multiple = np.sqrt(
    np.sum((y - Phi_multiple @ beta_multiple) ** 2) / (n - p_star)
)
se_multiple = rse_multiple * np.sqrt(
    np.diag(np.linalg.inv(Phi_multiple.T @ Phi_multiple))
)
t_multiple = beta_multiple / se_multiple
p_multiple = 2 * stats.t.sf(np.abs(t_multiple), df=n - p_star)
print(t_multiple)

[ 57.5396503   -4.26725251 -11.53463263]

print(p_multiple)

[2.31711323e-192  2.48848204e-005  1.12436236e-026]

```

7.6.2 R

```

parameters::parameters(fit_multiple) |>
parameters::print_md(digits = 4)

```

Parameter	Coefficient	SE	95% CI	t(389)	p
(Intercept)	45.6402	0.7932	(44.0807, 47.1997)	57.5397	< .001
horsepower	-0.0473	0.0111	(-0.0691, -0.0255)	-4.2673	< .001
weight	-0.0058	0.0005	(-0.0068, -0.0048)	-11.5346	< .001

For **horsepower**, $t = -4.27$ with $n - p^* = 389$ degrees of freedom, so $p = 2.5 \times 10^{-5}$. Even with **weight** held fixed, the data are unlikely under $H_0 : \beta_{\text{hp}} = 0$. The t -statistic for **horsepower** is much smaller in size than in the one-feature fit, where it was -24.5 , because **horsepower** and **weight** carry much of the same information (Solution 6.5.24).

7.7 Is any feature useful?

Exercise 7.5 (Why not just check each t -test?). Suppose a model has $p = 100$ features, and none of them is related to the response:

$$\begin{aligned}
 \beta_1 &= \beta_2 \\
 &= \dots \\
 &= \beta_{100} \\
 &= 0.
 \end{aligned}$$

We run the t -test of Definition 7.4 on each coefficient at level 0.05. About how many of the 100 p -values will fall below 0.05?

Solution

Solution 7.5. Under $H_0 : \beta_j = 0$, each p -value falls below 0.05 with probability 0.05. So we expect about $100 \times 0.05 = 5$ of them to fall below 0.05 by chance, even though no feature is related to the response.

Exercise 7.6 (Why not just check each t -test? (part 2)). Use the setup of Exercise 7.5. What does that say about using the separate t -tests to decide whether any feature is related to the response?

Solution

Solution 7.6. If we declared a relationship whenever any one p -value was small, we would almost always find one. We need a single test of all the coefficients together, whose chance of a false alarm is 0.05 no matter how many features there are [32, Sec. 3.2.2], p. 86.

Definition 7.5 (The F -test that any feature is useful). To test

$$H_0 : \beta_j = 0 \text{ for every } j \in \{1, \dots, p\} \quad \text{against} \quad H_A : \text{at least one } \beta_j \neq 0,$$

compute the F -statistic

$$F = \frac{(\text{TSS} - \text{RSS})/p}{\text{RSS}/(n - p - 1)},$$

where $\text{TSS} = \sum_i (y_i - \bar{y})^2$ is the total sum of squares and $\text{RSS} = \sum_i (y_i - \hat{y}_i)^2$ is the residual sum of squares [32, Sec. 3.2.2], p. 84. If H_0 holds and the noise is Gaussian, F has the F distribution^a with p and $n - p - 1$ degrees of freedom [32, Sec. 3.2.2], p. 85. The p -value is $\mathbb{P}(F_{p, n-p-1} \geq F)$.

The numerator is the drop in squared error per feature, and the denominator is $\hat{\sigma}^2$, the squared residual standard error. Under H_0 , both estimate σ^2 , so F is near 1; when some feature is useful, the numerator is larger, and so is F .

^a<https://morrison-lab.github.io/sds/inference.html#def-f-dist>

Example 7.8 (The F -test for horsepower and weight). For the two-feature fit of Solution 6.5.24, compute F and its p -value from Definition 7.5, and compare them with the F -statistic that `broom::glance()` reports in R:

7.7.1 Python

```
tss = np.sum((y - y.mean()) ** 2)
rss_multiple = np.sum((y - Phi_multiple @ beta_multiple) ** 2)
p_features = 2
f_multiple = ((tss - rss_multiple) / p_features) / (
    rss_multiple / (n - p_features - 1)
)
print(f_multiple, stats.f.sf(f_multiple, p_features, n - p_features - 1))
```

467.91015349994683 3.059606299669705e-104

7.7.2 R

```
broom::glance(fit_multiple) |>
  dplyr::select(statistic, p.value, df, df.residual)
```

```
# A tibble: 1 x 4
  statistic p.value    df df.residual
  <dbl>    <dbl> <dbl>    <int>
1    468. 3.06e-104     2        389
```

$F = 467.9$ on 2 and 389 degrees of freedom, far above 1. Its p -value is below 10^{-100} , so at least one of horsepower and weight is related to mpg.

7.8 Testing a group of features

Definition 7.6 (The partial F -test for a group of features). To test whether a group of q features adds anything to a model that already contains the other $p - q$, fit the model twice: once with all p features, with residual sum of squares RSS , and once without the q features, with residual sum of squares RSS_0 . The **partial F -statistic** is

$$F = \frac{(\text{RSS}_0 - \text{RSS})/q}{\text{RSS}/(n - p - 1)}$$

[32, Sec. 3.2.2], p. 85. If the q coefficients are all 0 and the noise is Gaussian, F has the $F_{q, n-p-1}$ distribution. Definition 7.5 is the case $q = p$, where the smaller model has only an intercept and $\text{RSS}_0 = \text{TSS}$. With $q = 1$, the partial F -statistic is the square of that feature's t -statistic.

The rme notes derive this test and connect it to the likelihood ratio test; see the partial F -test⁶.

Example 7.9 (Does origin matter once weight is known?). The model of Exercise 6.22, part 3, fits `mpg` on `weight` and the two origin indicators. Test whether the two indicators together add anything to `weight` alone, so $q = 2$ and $p = 3$:

7.8.1 Python

```
beta_weight, Phi_weight = fit([weight])
rss_0 = np.sum((y - Phi_weight @ beta_weight) ** 2)
rss_1 = np.sum((y - Phi_origin_weight @ beta_origin_weight) ** 2)
f_origin = ((rss_0 - rss_1) / 2) / (rss_1 / (n - 3 - 1))
print(f_origin, stats.f.sf(f_origin, 2, n - 3 - 1))
```

```
6.126802214577667 0.002400578748351605
```

7.8.2 R

```
fit_weight <- lm(mpg ~ weight, data = auto)
anova(fit_weight, fit_origin_weight) |>
  parameters::parameters() |>
  parameters::print_md(footer = "")
```

Parameter	RSS	Sum_Squares	df	df (error)	Mean_Square	F	p
Model 1: mpg ~ weight	7321.23			390			
Model 2: mpg ~ weight + factor(origin)	7097.10	224.14	2	388	112.07	6.13	0.002

$F = 6.13$ on 2 and 388 degrees of freedom, with $p = 0.0024$. The data are unlikely if origin adds nothing once weight is known, even though Solution 6.5.40 showed that weight explains much of the raw gap between origins.

7.9 Confidence and prediction intervals

Exercise 7.7 (Two questions about a 98-horsepower car). Solution 6.5.4 predicted 24.5 miles per gallon for a car with 98 horsepower. Which sources of error make that number uncertain as an estimate^a of the *average* mpg of all 98-horsepower cars?

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

⁶<https://morrison-lab.github.io/rme/chapters/Linear-models-overview.html#sec-partial-F-test>

Solution

Solution 7.7. In Exercise 7.7, the average mpg at 98 horsepower is $\phi(\underline{x}_0) \cdot \vec{\beta}$, with $\phi(\underline{x}_0) = [1, 98]^\top$. Its estimate^a $\phi(\underline{x}_0) \cdot \hat{\vec{\beta}}$ is off only because $\hat{\vec{\beta}}$ differs from $\vec{\beta}$: the reducible error of Proposition 6.1.

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Exercise 7.8 (Two questions about a 98-horsepower car (part 2)). Use the setup of Exercise 7.7. Which sources of error make that number uncertain as a guess at the mpg of *one particular* 98-horsepower car?

Solution

Solution 7.8. One car's mpg is $y_0 = \phi(\underline{x}_0) \cdot \vec{\beta} + \varepsilon_0$. The guess has the same error from $\hat{\vec{\beta}}$, plus the car's own noise ε_0 , whose variance^a is the residual variance.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

Exercise 7.9 (Two questions about a 98-horsepower car (part 3)). Use the setup of Exercise 7.7. Which of the two should we be less sure about?

Solution

Solution 7.9. We should be less sure about the single car, because its error adds the noise ε_0 to the error in $\hat{\vec{\beta}}$ [32, Sec. 3.2.2], p. 90.

Exercise 7.10 (The variance of a prediction). Under the assumptions of Theorem 7.1, let $\phi_0 = \phi(\underline{x}_0)$ be the feature vector of a new input, let $\hat{y}_0 = \phi_0 \cdot \hat{\vec{\beta}}$, and let $y_0 = \phi_0 \cdot \vec{\beta} + \varepsilon_0$, where ε_0 has mean 0 and variance σ^2 and is independent of the training noise $\vec{\varepsilon}$. Find $\text{Var}(\hat{y}_0)$, and

Solution

Solution 7.10. Here $\underline{x}_0$ is the new input^a of Exercise 7.10, ϕ_0 is its feature vector, ε_0 is the noise in the new outcome^b Y_0 at $\underline{x}_0$, and σ^2 is the noise variance^c of Theorem 7.1.

$\hat{Y}_0 = \phi_0 \cdot \hat{\vec{\beta}}$ is a linear combination^d of the entries of $\hat{\vec{\beta}}$ with fixed weights ϕ_0 , and Theorem 7.1 gives its covariance, so

$$\begin{aligned} \text{Var}(\hat{Y}_0) &= \phi_0^\top \text{Cov}(\hat{\vec{\beta}}) \phi_0 && \text{(variance of a fixed linear combination)} \\ &= \phi_0^\top \sigma^2 (D^\top D)^{-1} \phi_0 && \text{(covariance of } \hat{\vec{\beta}} \text{)} \\ &= \sigma^2 \phi_0^\top (D^\top D)^{-1} \phi_0 && \text{(move the noise variance } \sigma^2 \text{ out)} \\ &= \sigma^2 h_0 && \text{(let } h_0 \text{ be the leverage of the new input, } \phi_0^\top (D^\top D)^{-1} \phi_0 \text{)} \end{aligned}$$

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^c<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

^d<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-linear-combination>

Exercise 7.11 (The variance of a prediction (part 2)). Use the setup of Exercise 7.10. $\text{Var}(y_0 - \hat{y}_0)$.

Solution

Solution 7.11. $Y_0 - \hat{Y}_0 = \phi_0 \cdot \vec{\beta} + \varepsilon_0 - \hat{Y}_0$, and $\phi_0 \cdot \vec{\beta}$ is a constant^a. The new noise ε_0 is independent^b of $\hat{\vec{\beta}}$, which depends only on the training noise, so

$$\begin{aligned}\text{Var}(Y_0 - \hat{Y}_0) &= \text{Var}(\varepsilon_0) + \text{Var}(\hat{Y}_0) && \text{(constants add no variance; independent terms)} \\ &= \sigma^2 + \sigma^2 h_0 && \text{(@sol-prediction-variance)} \\ &= \sigma^2(1 + h_0) && \text{(factor out } \sigma^2\text{)}\end{aligned}$$

The extra 1 is the residual variance of Solution 7.8.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/pds/independence.html#def-indpt>

Theorem 7.4 (The variance of a prediction). *Under the assumptions of Theorem 7.1, let $\phi_0 = \phi(\underline{x}_0)$ be the feature vector of a new input and $\hat{y}_0 = \phi_0 \cdot \hat{\vec{\beta}}$. Then*

$$\text{Var}(\hat{y}_0) = \sigma^2 \phi_0^\top (D^\top D)^{-1} \phi_0.$$

i Proof

Proof. This is the solution to Exercise 7.10: see Solution 7.10. □

Example 7.10 (The variance of a prediction at a new input). Fit a line to the inputs $x = 1, 2, 3, 4, 5$ with noise variance $\sigma^2 = 4$, and predict at $x_0 = 3.5$, so $\phi_0 = [1, 3.5]^\top$.

```
pv_Phi <- cbind(1, 1:5)
pv_phi0 <- c(1, 3.5)
pv_sigma2 <- 4
pv_h0 <- drop(t(pv_phi0) %*% solve(t(pv_Phi) %*% pv_Phi) %*% pv_phi0)
c(h0 = pv_h0, var_prediction = pv_sigma2 * pv_h0)
```

```
h0 var_prediction
0.225          0.900
```

By Theorem 7.4, $\text{Var}(\hat{y}_0) = 4 \times 0.225 = 0.9$. The variance is smallest at the center of the inputs, $x_0 = 3$, and grows as x_0 moves away from it.

Theorem 7.5 (The variance of a prediction error). *In the setting of Theorem 7.4, let $y_0 = \phi_0 \cdot \vec{\beta} + \varepsilon_0$, where ε_0 has mean 0 and variance σ^2 and is independent of the training noise. Then*

$$\text{Var}(y_0 - \hat{y}_0) = \sigma^2(1 + \phi_0^\top (D^\top D)^{-1} \phi_0).$$

i Proof

Proof. This is the solution to Exercise 7.11: see Solution 7.10. □

Example 7.11 (The variance of a prediction error). Continue Example 7.10, with $\sigma^2 = 4$ and $\phi_0 = [1, 3.5]^\top$ for the inputs $x = 1, \dots, 5$.

```

pe_Phi <- cbind(1, 1:5)
pe_phi0 <- c(1, 3.5)
pe_sigma2 <- 4
pe_h0 <- drop(t(pe_phi0) %*% solve(t(pe_Phi) %*% pe_Phi) %*% pe_phi0)
c(var_prediction = pe_sigma2 * pe_h0, var_error = pe_sigma2 * (1 + pe_h0))

```

```

var_prediction    var_error
           0.9           4.9

```

By Theorem 7.5, $\text{Var}(y_0 - \hat{y}_0) = 4 \times (1 + 0.225) = 4.9$. Of this, 4 is the new response's own noise, which no amount of training data removes.

Definition 7.7 (Confidence interval for the mean response). The 95% confidence interval for the mean response $\phi_0 \cdot \vec{\beta}$ at a new input x_0 is

$$\hat{y}_0 \pm t_{n-p^*}(0.975) \hat{\sigma} \sqrt{\phi_0^\top (D^\top D)^{-1} \phi_0},$$

where $\hat{y}_0 = \phi_0 \cdot \hat{\vec{\beta}}$ and $\hat{\sigma}$ is the residual standard error. It is Definition 7.3 with the variance of Theorem 7.4 in place of $\text{Var}(\hat{\beta}_j)$.

Example 7.12 (Average fuel economy at 98 horsepower). For the one-feature fit of Solution 6.5.4 and $\phi_0 = [1, 98]^\top$, compute the confidence interval of Definition 7.7, and compare it with R's `predict(..., interval = "confidence")`:

7.9.1 Python

```

# PtP_inv and sigma_simple come from the covariance example,
# and t_crit from the t-distribution example, earlier in this chapter.
phi_0 = np.array([1, 98])
y_hat_0 = phi_0 @ beta_simple
h_0 = phi_0 @ PtP_inv @ phi_0
mean_ci = y_hat_0 + np.array([-1, 1]) * t_crit * sigma_simple * np.sqrt(h_0)
print(y_hat_0, mean_ci)

```

```

24.467077152512417 [23.97307896 24.96107534]

```

7.9.2 R

```

predict(fit_simple, tibble::tibble(horsepower = 98), interval = "confidence")

```

```

      fit      lwr      upr
1 24.46708 23.97308 24.96108

```

The interval runs from 23.97 to 24.96 miles per gallon. It is narrow, because 392 cars pin down the line well near 98 horsepower.

Definition 7.8 (Prediction interval). The 95% **prediction interval** for a single new response y_0 at input x_0 is

$$\hat{y}_0 \pm t_{n-p^*}(0.975) \hat{\sigma} \sqrt{1 + \phi_0^\top (D^\top D)^{-1} \phi_0}.$$

If the noise is Gaussian, it contains the new response y_0 with probability 0.95 [34].

Example 7.13 (One car at 98 horsepower). Compute the prediction interval of Definition 7.8 at 98 horsepower, compare it with R's `predict(..., interval = "prediction")`, and draw both intervals across the range of horsepower:

7.9.3 Python

```
pred_int = y_hat_0 + np.array([-1, 1]) * t_crit * sigma_simple * np.sqrt(1 + h_0)
print(pred_int)
```

```
[14.80939607 34.12475823]
```

```
# A function, so the plotting calls' return values are not printed.
```

```
def plot_prediction_bands():
    grid = np.linspace(horsepower.min(), horsepower.max(), 100)
    Phi_grid = np.column_stack([np.ones_like(grid), grid])
    fitted_grid = Phi_grid @ beta_simple
    h_grid = np.sum(Phi_grid @ PtP_inv * Phi_grid, axis=1)
    half_mean = t_crit * sigma_simple * np.sqrt(h_grid)
    half_pred = t_crit * sigma_simple * np.sqrt(1 + h_grid)
    fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
    ax.scatter(horsepower, y, s=12, alpha=0.4, color="grey");
    ax.plot(grid, fitted_grid, color="C3", label="Fitted line");
    ax.fill_between(grid, fitted_grid - half_mean, fitted_grid + half_mean,
                    color="C3", alpha=0.3, label="95% confidence band");
    ax.plot(grid, fitted_grid - half_pred, color="C0", linestyle="--",
            label="95% prediction band");
    ax.plot(grid, fitted_grid + half_pred, color="C0", linestyle="--");
    ax.set_xlabel("Horsepower");
    ax.set_ylabel("Miles per gallon");
    fig.legend(loc="outside lower center", ncol=3);
    plt.show()
```

```
plot_prediction_bands()
```

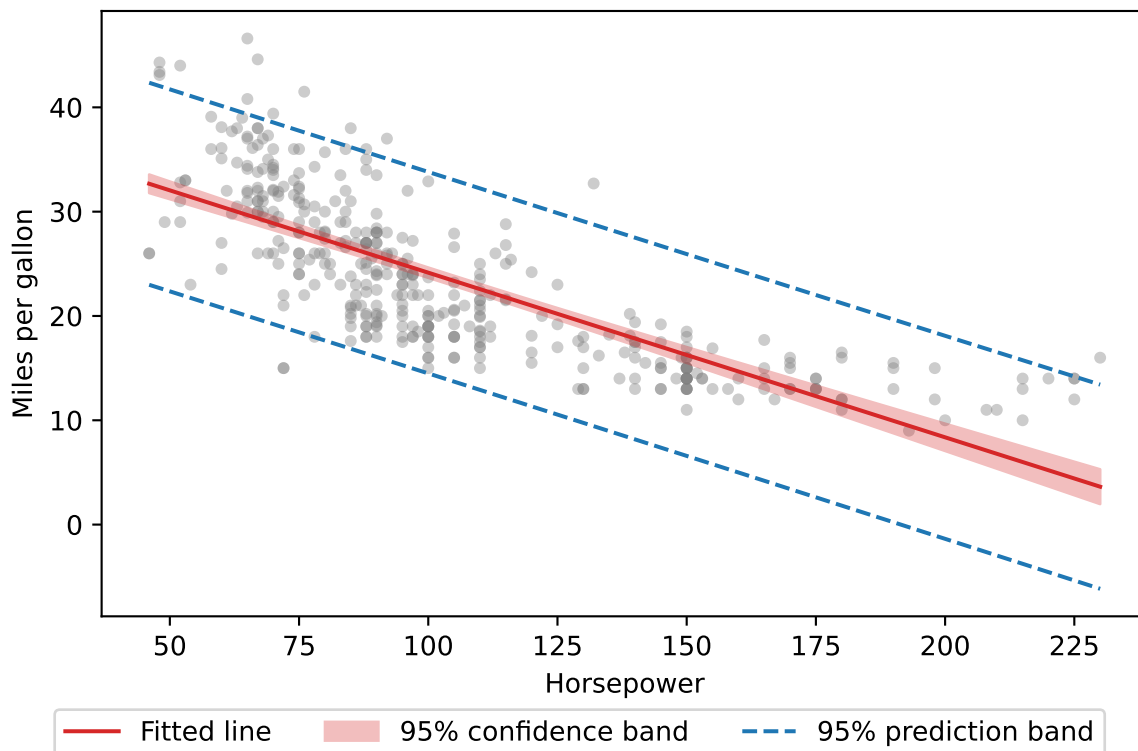


Figure 22: Fuel economy against horsepower for the Auto data, with the least squares line, its 95% confidence band for the mean response, and its 95% prediction band.

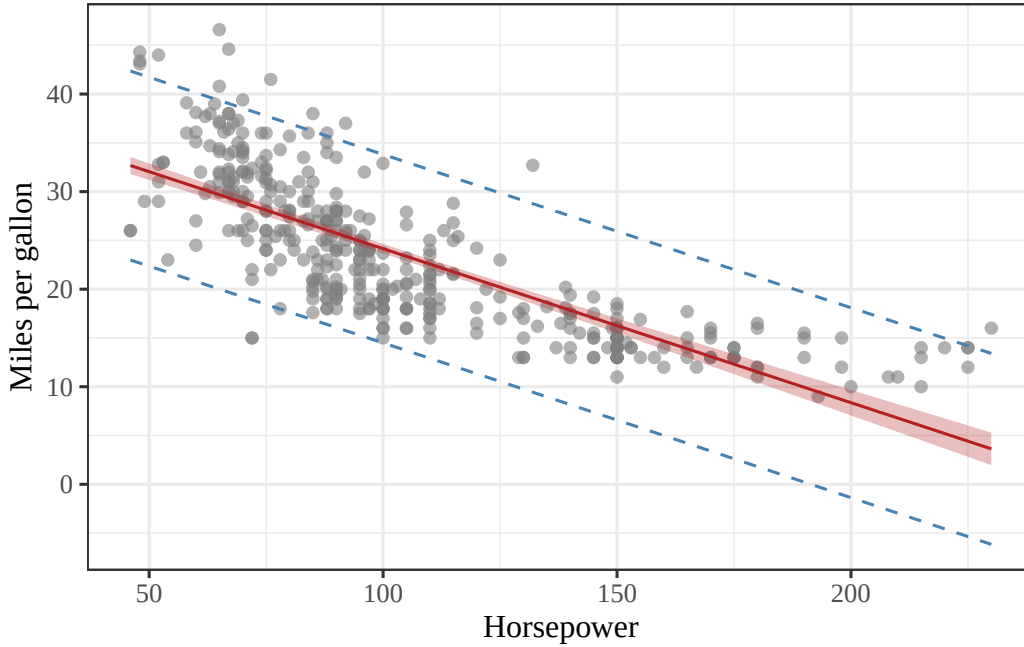
7.9.4 R

```
predict(fit_simple, tibble::tibble(horsepower = 98), interval = "prediction")

      fit      lwr      upr
1 24.46708 14.8094 34.12476

library(ggplot2)

grid <- tibble::tibble(horsepower = seq(min(auto$horsepower),
                                         max(auto$horsepower), length.out = 100))
band_mean <- predict(fit_simple, grid, interval = "confidence")
band_pred <- predict(fit_simple, grid, interval = "prediction")
bands <- tibble::tibble(grid, fit = band_mean[, "fit"],
                        mean_lwr = band_mean[, "lwr"], mean_upr = band_mean[, "upr"],
                        pred_lwr = band_pred[, "lwr"], pred_upr = band_pred[, "upr"])
ggplot(bands, aes(horsepower)) +
  geom_point(data = auto, aes(y = mpg), colour = "grey50", alpha = 0.6) +
  geom_ribbon(aes(ymin = mean_lwr, ymax = mean_upr),
            fill = "firebrick", alpha = 0.3) +
  geom_line(aes(y = fit), colour = "firebrick") +
  geom_line(aes(y = pred_lwr), colour = "steelblue", linetype = "dashed") +
  geom_line(aes(y = pred_upr), colour = "steelblue", linetype = "dashed") +
  labs(x = "Horsepower", y = "Miles per gallon")
```



The prediction interval at 98 horsepower runs from 14.8 to 34.1 miles per gallon, about 20 times as wide as the confidence interval of Example 7.12. The number of cars controls the width of the confidence band, which would shrink toward the line with more data. The prediction band would not shrink below about $\pm 2\hat{\sigma}$, because no amount of data removes a new car's own noise.

Theorem 7.6 (The prediction interval is wider than the confidence interval). *At the same input x_0 and the same level, the prediction interval of Definition 7.8 is wider than the confidence interval for the mean response of Definition 7.7. With $h_0 = \phi_0^\top (D^\top D)^{-1} \phi_0$, the ratio of their widths is $\sqrt{(1 + h_0)/h_0}$ [32, Sec. 3.2.2], p. 90.*

i Proof

Proof. Both intervals have the center $\hat{y}_0$ and use the same $t_{n-p^*}(0.975)$ and $\hat{\sigma}$. Their half-widths are

$$t_{n-p^*}(0.975) \hat{\sigma} \sqrt{h_0} \quad \text{and} \quad t_{n-p^*}(0.975) \hat{\sigma} \sqrt{1 + h_0}.$$

Because $1 + h_0 > h_0$, the second is larger. The ratio of the two is $\sqrt{(1 + h_0)/h_0}$. The extra 1 is the variance of the new response's own noise. $\square$

Example 7.14 (Both intervals at every car). For each car of the `Auto` data, take its own horsepower as x_0 . Compute the width of the confidence interval for the mean response and of the prediction interval, and check that the prediction interval is wider at every car, with the width ratio of Theorem 7.6:

7.9.5 Python

```
# Phi_simple, PtP_inv, sigma_simple and t_crit come from the examples earlier in this chapter.
h_cars = np.sum(Phi_simple @ PtP_inv * Phi_simple, axis=1)
width_mean = 2 * t_crit * sigma_simple * np.sqrt(h_cars)
width_pred = 2 * t_crit * sigma_simple * np.sqrt(1 + h_cars)
ratio_cars = width_pred / width_mean
print(np.all(width_pred > width_mean), ratio_cars.min(), ratio_cars.max())

True 5.883031030489702 19.82234421472044
```

7.9.6 R

```
conf_cars <- predict(fit_simple, auto, interval = "confidence")
pred_cars <- predict(fit_simple, auto, interval = "prediction")
width_conf <- conf_cars[, "upr"] - conf_cars[, "lwr"]
width_pred <- pred_cars[, "upr"] - pred_cars[, "lwr"]
c(all_wider = all(width_pred > width_conf),
  min_ratio = min(width_pred / width_conf),
  max_ratio = max(width_pred / width_conf))
```

```
all_wider min_ratio max_ratio
1.000000  5.883031 19.822344
```

The prediction interval is wider than the confidence interval at 392 of the 392 cars. The ratio of widths runs from 5.9 to 19.8. It is smallest where h_0 is largest, at the extreme horsepower values, where the fitted line is least certain. Even there, the new car's own noise makes the prediction interval wider.

8 Checking the model

This section follows the list of potential problems in [32, Sec. 3.3.3]. The rme notes on diagnostics⁷ go further, with quantile-quantile (QQ) plots, Cook's distance and formal tests.

8.1 Non-linearity

Exercise 8.1 (Plot the residuals). For the one-feature fit of Solution 6.5.4, plot each car's residual $e_i = y_i - \hat{y}_i$ against its fitted value $\hat{y}_i$.

Solution

Solution 8.1.

8.1.1 Python

```
# A function, so the plotting calls' return values are not printed.
def plot_residuals(beta_hat, Phi, label):
    fitted_values = Phi @ beta_hat
    fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
    ax.scatter(fitted_values, y - fitted_values, s=12, alpha=0.5);
    ax.axhline(0, color="grey", linewidth=1);
    ax.set_xlabel(f"Fitted mpg ({label})");
    ax.set_ylabel("Residual");
    plt.show()

plot_residuals(beta_simple, Phi_simple, "straight line")
```

⁷<https://morrison-lab.github.io/rme/chapters/Linear-models-overview.html#sec-diagnose-LMs>

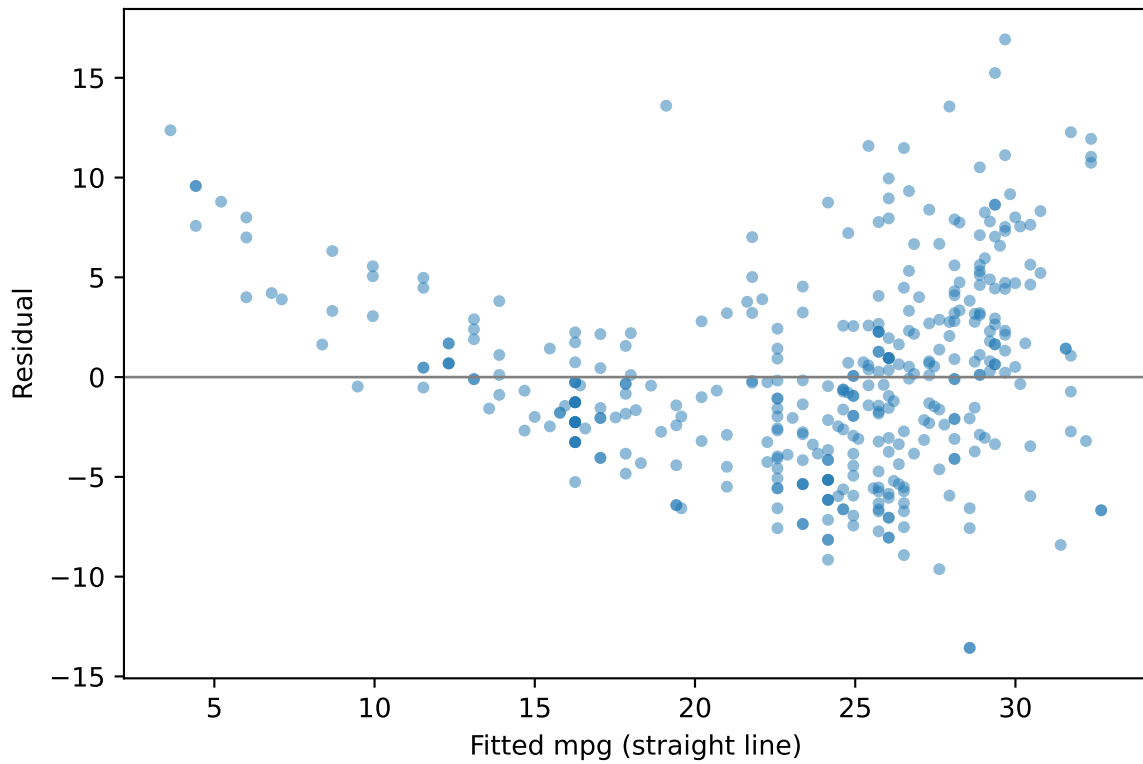


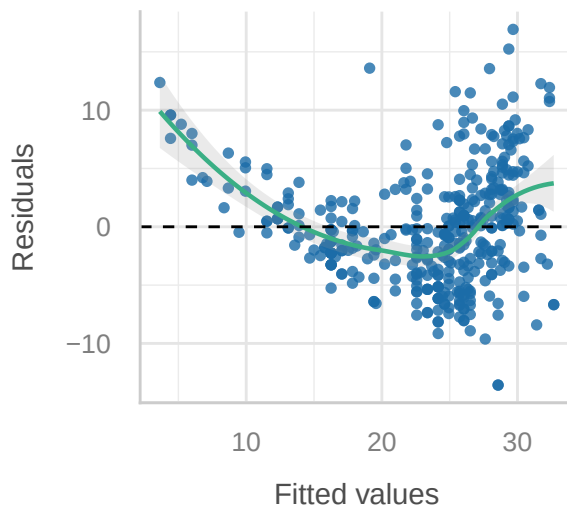
Figure 23: Residuals against fitted values for the straight-line fit of mpg on horsepower.

8.1.2 R

```
performance::check_model(fit_simple, check = "linearity") |>
  plot()
```

Linearity

Reference line should be flat and horizontal



Exercise 8.2 (Plot the residuals (part 2)). Use the setup of Exercise 8.1. If the straight line captured the relationship, what would the plot look like?

Solution

Solution 8.2. If the line captured the relationship, the residuals would scatter evenly above and below 0 at every fitted value, with no pattern.

Exercise 8.3 (Plot the residuals (part 3)). Use the setup of Exercise 8.1. What do you see instead?

Solution

Solution 8.3. Instead they form a U: mostly positive at both ends and mostly negative in the middle. The line is too high in the middle of the horsepower range and too low at both ends, so the relationship is curved [32, Sec. 3.3.3], pp. 100–101.

Definition 8.1 (Residual plot). A **residual plot** shows the residuals $e_i = y_i - \hat{y}_i$ of Definition 2.3 against the fitted values $\hat{y}_i$ (or, in simple linear regression, against the feature). For a model that captures the relationship, the points scatter around 0 with no visible pattern; a curve in the plot is a sign of non-linearity [32, Sec. 3.3.3], p. 101.

Example 8.1 (The residual plot after adding horsepower²). The residual plot for the quadratic fit of Exercise 6.17, part 1:

8.1.3 Python

```
plot_residuals(beta_quadratic, Phi_quadratic, "quadratic")
```

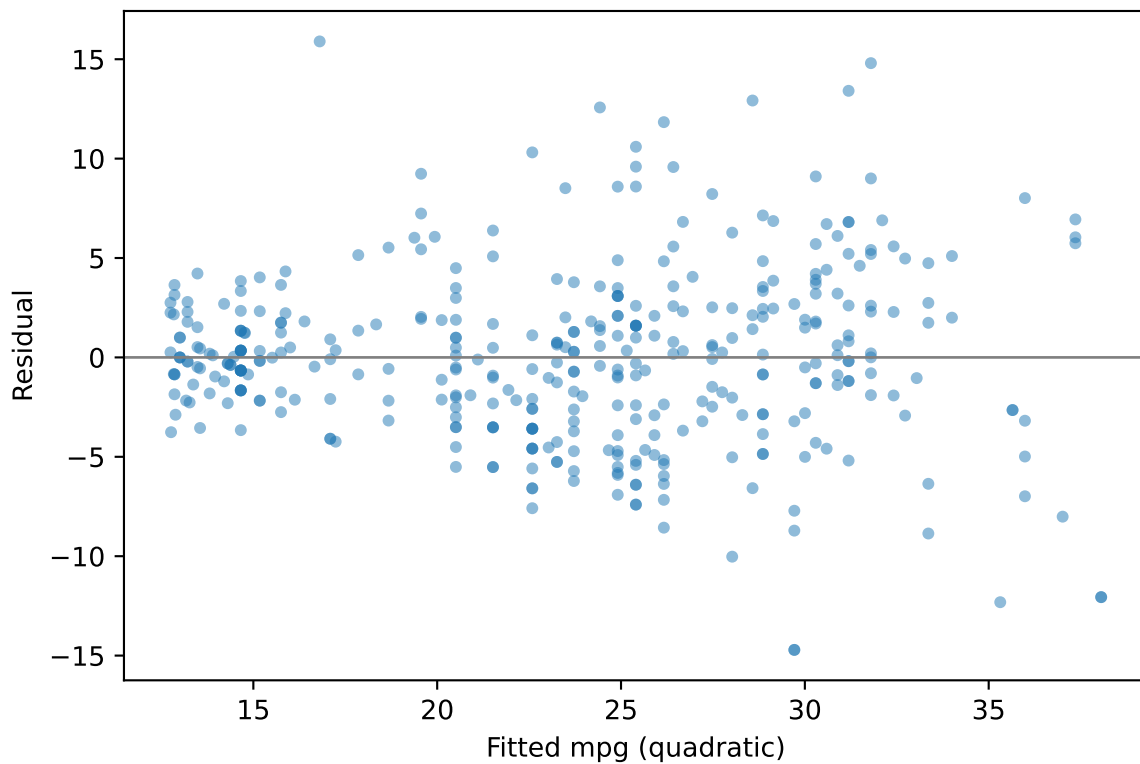


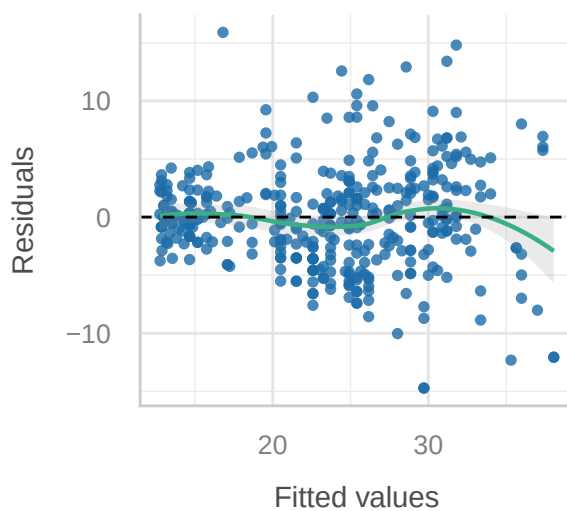
Figure 24: Residuals against fitted values for the quadratic fit of mpg on horsepower.

8.1.4 R

```
performance::check_model(fit_quadratic, check = "linearity") |>
  plot()
```

Linearity

Reference line should be flat and horizontal



The U shape is gone, so the quadratic term captured the curve.

8.2 Non-constant variance

Exercise 8.4 (Does the spread change?). In the residual plot of Example 8.1, split the cars into thirds by fitted value, and compute the standard deviation of the residuals in each third.

Solution

Solution 8.4.

8.2.1 Python

```
def spread_by_third(fitted_values, residuals):
    """Standard deviation of the residuals in each third of the fitted values."""
    cuts = np.quantile(fitted_values, [1 / 3, 2 / 3])
    third = np.searchsorted(cuts, fitted_values)
    return np.array([np.std(residuals[third == k], ddof=1) for k in range(3)])
```

```
fitted_quadratic = Phi_quadratic @ beta_quadratic
spread_quadratic = spread_by_third(fitted_quadratic, y - fitted_quadratic)
print(spread_quadratic)
```

```
[2.85532811 4.46613435 5.36238522]
```

```
# The spreads have no simple interval, so resample the cars.
```

```
# Each resample refits the quadratic, finds its own thirds, and recomputes the spreads.
```

```
def bootstrap_spread_ci(B=2000, seed=571):
```

```
    """95% percentile intervals for the three spreads and for the highest third's spread over the low
```

```
    rng = np.random.default_rng(seed)
```

```
    spreads = np.empty((B, 3))
```

```
    for b in range(B):
```

```
        rows = rng.integers(0, n, n)
```

```
        beta_b = np.linalg.lstsq(Phi_quadratic[rows], y[rows], rcond=None)[0]
```

```
        fitted_b = Phi_quadratic[rows] @ beta_b
```

```
        spreads[b] = spread_by_third(fitted_b, y[rows] - fitted_b)
```

```
    return (np.quantile(spreads, [0.025, 0.975], axis=0),
            np.quantile(spreads[:, 2] / spreads[:, 0], [0.025, 0.975]))
```

```
spread_ci_quadratic, spread_ratio_ci = bootstrap_spread_ci()
```

```
print(spread_ci_quadratic.T)
```

```
[[2.25359029 3.54058612]
```

```
 [3.74873411 5.05624762]
```

```
 [4.56392342 6.10287934]]
```

```
print(spread_quadratic[2] / spread_quadratic[0], spread_ratio_ci)
```

```
1.878027678194057 [1.42727074 2.46554721]
```

8.2.2 R

```
spread_of_thirds <- function(fitted_values, residuals) {
  third <- cut(fitted_values, quantile(fitted_values, c(0, 1 / 3, 2 / 3, 1)),
              include.lowest = TRUE)
  tapply(residuals, third, sd)
}
spread_by_third <- function(model) {
  spread_of_thirds(fitted(model), residuals(model))
}
spread_quadratic_r <- spread_by_third(fit_quadratic)
spread_quadratic_r
```

```
[12.7,20.5] (20.5,26.4] (26.4,38.1]
2.855328    4.504321    5.288733
```

```
# The spreads have no simple interval, so resample the cars.
# Each resample refits the quadratic,
# finds its own thirds, and recomputes the spreads.
```

```
boot_spread_ci <- function(model, data, n_boot = 2000) {
  set.seed(571)
  design <- model.matrix(model)
  y <- data$mpg
  spreads <- vapply(seq_len(n_boot), \(b) {
    rows <- sample(nrow(design), replace = TRUE)
    residuals_b <- lm.fit(design[rows, ], y[rows])$residuals
    spread_of_thirds(y[rows] - residuals_b, residuals_b)
  }, numeric(3))
  list(spreads = apply(spreads, 1, quantile, probs = c(0.025, 0.975)),
       ratio = quantile(spreads[3, ] / spreads[1, ], c(0.025, 0.975)))
}
spread_ci_r <- boot_spread_ci(fit_quadratic, auto)
t(spread_ci_r$spreads)
```

```
                2.5%    97.5%
[13.5,22.4] 2.266060 3.574281
(22.4,26.9] 3.698487 5.030732
(26.9,36.7] 4.588323 6.091378
```

```
spread_quadratic_r[[3]] / spread_quadratic_r[[1]]
```

```
[1] 1.852233
```

```
spread_ci_r$ratio
```

```
        2.5%    97.5%
1.421881 2.447919
```

Exercise 8.5 (Does the spread change? (part 2)). Use the setup of Exercise 8.4. Is the spread of the residuals the same across the range of fitted values?

Solution

Solution 8.5. No. The standard deviation^a grows from 2.9 (95% bootstrap interval 2.3 to 3.5) in the lowest third to 5.4 (4.6 to 6.1) in the highest, so the plot fans out like a funnel. The highest third's spread is 1.9 times the lowest third's (1.4 to 2.5), an interval that excludes 1, the ratio for equal spreads. Each interval is the 2.5th to 97.5th percentile over 2,000 resamples of the cars, refitting the quadratic and recomputing the thirds each time.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-sd>

Exercise 8.6 (Does the spread change? (part 3)). Use the setup of Exercise 8.4. Which assumption of Theorem 7.1 does that bear on?

Solution

Solution 8.6. Theorem 7.1 assumes $\text{Cov}(\vec{\varepsilon}) = \sigma^2 \mathbf{I}_n$, the same variance^a σ^2 for every car. A funnel says the noise variance grows with the fitted value, which pds^b calls *heteroskedasticity*. Then the standard errors^c estimated^d as in Definition 7.2, and the intervals and tests built on them, are no longer right [32, Sec. 3.3.3], p. 103.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

^b<https://morrison-lab.github.io/pds/variance-covariance.html#def-heterosked>

^c<https://morrison-lab.github.io/sds/estimation.html#def-SE>

^d<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Example 8.2 (Fitting $\log(\text{mpg})$ instead). A common remedy for a funnel is to model a concave function of the response, such as $\log y$, which shrinks large responses more than small ones [32, Sec. 3.3.3], p. 103. Refit the quadratic model with $\log(\text{mpg})$ as the response, and repeat Exercise 8.4:

8.2.3 Python

```
log_y = np.log(y)
beta_log = np.linalg.lstsq(Phi_quadratic, log_y, rcond=None)[0]
fitted_log = Phi_quadratic @ beta_log
spread_log = spread_by_third(fitted_log, log_y - fitted_log)
print(spread_log)

[0.15802602 0.18322063 0.18559353]

print(spread_quadratic[2] / spread_quadratic[0], spread_log[2] / spread_log[0])

1.878027678194057 1.1744492221667266

# Bootstrap: resample cars, refit the quadratic model to each response,
# and recompute the ratio of the highest third's spread to the lowest third's.
rng_log = np.random.default_rng(571)
ratio_boot = np.empty((2000, 2))
for b in range(2000):
    rows = rng_log.integers(0, n, n)
    for j, response in enumerate([y, log_y]):
        beta_b = np.linalg.lstsq(Phi_quadratic[rows], response[rows], rcond=None)[0]
        fitted_b = Phi_quadratic[rows] @ beta_b
        spread_b = spread_by_third(fitted_b, response[rows] - fitted_b)
        ratio_boot[b, j] = spread_b[2] / spread_b[0]
# Columns: original scale, log scale; rows: 2.5th and 97.5th percentiles.
ratio_ci = np.quantile(ratio_boot, [0.025, 0.975], axis=0)
print(ratio_ci)

[[1.42727074 0.93581492]
 [2.46554721 1.4846279 ]]
```

8.2.4 R

```
fit_log <- lm(log(mpg) ~ horsepower + I(horsepower^2), data = auto)
spread_by_third(fit_log)

[2.53,2.99] (2.99,3.24] (3.24,3.7]
0.1580260 0.1832827 0.1851136
```

```
# Bootstrap: resample cars, refit the quadratic model to each response,
# and recompute the ratio of the highest third's spread to the lowest third's.
set.seed(571)
ratio_boot <- replicate(2000, {
  boot_cars <- auto[sample(nrow(auto), replace = TRUE), ]
  spread_orig <- spread_by_third(
    lm(mpg ~ horsepower + I(horsepower^2), data = boot_cars))
  spread_log <- spread_by_third(
    lm(log(mpg) ~ horsepower + I(horsepower^2), data = boot_cars))
  c(original = spread_orig[[3]] / spread_orig[[1]],
    log = spread_log[[3]] / spread_log[[1]])
})
# Rows: 2.5th and 97.5th percentiles; columns: original scale, log scale.
apply(ratio_boot, 1, quantile, c(0.025, 0.975))
```

	original	log
2.5%	1.421881	0.9329021
97.5%	2.447919	1.4673721

On the original scale, the highest third's residual spread was 1.88 times the lowest third's, with a 95% bootstrap percentile interval from 1.43 to 2.47. On the log scale, the ratio is 1.17, with an interval from 0.94 to 1.48. Equal spread in the two thirds is a ratio of 1: the original-scale interval excludes it, and the log-scale interval includes it. Each interval comes from 2000 resamples of the cars, refitting the model each time, and describes which cars were drawn. Another remedy, when the variance of each response is known up to a constant, is weighted least squares, which gives each squared residual a weight inversely proportional to that response's variance [32, Sec. 3.3.3], p. 103.

8.3 Correlated errors

Exercise 8.7 (Plot the residuals in time order). The rows of the Auto data are sorted by model year. For the quadratic fit of Exercise 6.17, compute the correlation between each residual and the next one in the file, and the average residual for each model year. Theorem 7.1 assumes the noise terms are uncorrelated. Do the residuals look uncorrelated?

Solution

Solution 8.7.

8.3.1 Python

```
def lag_correlation(residuals):
    """Correlation between each residual and the next one in the file."""
    return np.corrcoef(residuals[:-1], residuals[1:])[0, 1]

residuals_quadratic = y - fitted_quadratic
lag_quadratic = lag_correlation(residuals_quadratic)
year_means = pd.Series(residuals_quadratic).groupby(year).mean()
year_means_min, year_means_max = year_means.min(), year_means.max()
last_negative_year = int(year_means.index[year_means < 0].max())
print(lag_quadratic)

0.4602676830246073

print(year_means)

70.0    -0.616227
71.0    -1.750040
```

```

72.0    -1.938788
73.0    -2.506749
74.0    -2.170697
75.0    -2.926681
76.0    -2.024674
77.0     0.326999
78.0     0.439857
79.0     1.808373
80.0     5.245947
81.0     2.593976
82.0     4.701654

```

```
dtype: float64
```

```

# Each year's average residual, with a 95% t interval from the cars of that year
residuals_by_year = pd.Series(residuals_quadratic).groupby(year)
year_table = pd.DataFrame({"mean": residuals_by_year.mean(),
                           "sem": residuals_by_year.sem(),
                           "df": residuals_by_year.count() - 1})
year_table["half"] = stats.t.ppf(0.975, year_table["df"]) * year_table["sem"]
year_table["lower"] = year_table["mean"] - year_table["half"]
year_table["upper"] = year_table["mean"] + year_table["half"]
year_table["year"] = year_table.index.astype(int)
print(year_table[["year", "mean", "lower", "upper"]].to_string(index=False))

```

year	mean	lower	upper
70	-0.616227	-1.903516	0.671061
71	-1.750040	-3.100018	-0.400061
72	-1.938788	-3.191400	-0.686177
73	-2.506749	-3.560732	-1.452767
74	-2.170697	-3.193823	-1.147571
75	-2.926681	-4.515174	-1.338188
76	-2.024674	-3.075580	-0.973768
77	0.326999	-0.683219	1.337218
78	0.439857	-0.710148	1.589862
79	1.808373	0.316701	3.300046
80	5.245947	3.100536	7.391358
81	2.593976	1.185572	4.002381
82	4.701654	3.459409	5.943899

```

beta_with_year, Phi_with_year = fit([horsepower, horsepower ** 2, year])
lag_with_year = lag_correlation(y - Phi_with_year @ beta_with_year)
print(lag_with_year)

```

```
0.3075231574213514
```

```
# Neighboring residuals are correlated, so resampling single cars would break the order.
# Instead resample blocks of 20 consecutive cars, refit the model, and recompute the correlation.
def block_bootstrap_lag_ci(Phi, block=20, B=20000, seed=571):
    """95% percentile interval for the neighboring-residual correlation from resampled blocks of cars.
    rng = np.random.default_rng(seed)
    n_blocks = int(np.ceil(n / block))
    lag = np.empty(B)
    for b in range(B):
        starts = rng.integers(0, n - block + 1, n_blocks)
        rows = (starts[:, None] + np.arange(block)).ravel()[:n]
        beta_b = np.linalg.lstsq(Phi[rows], y[rows], rcond=None)[0]
        lag[b] = lag_correlation(y[rows] - Phi[rows] @ beta_b)
    return np.quantile(lag, [0.025, 0.975])

lag_ci_quadratic = block_bootstrap_lag_ci(Phi_quadratic)
lag_ci_with_year = block_bootstrap_lag_ci(Phi_with_year)
print(lag_ci_quadratic, lag_ci_with_year)

[0.31733878 0.50875017] [0.1433561 0.38124989]
```

8.3.2 R

```
lag_correlation <- function(e) cor(head(e, -1), tail(e, -1))
lag_correlation(residuals(fit_quadratic))

[1] 0.4602677

tapply(residuals(fit_quadratic), auto$year, mean)

      70      71      72      73      74      75      76
-0.6162271 -1.7500396 -1.9387883 -2.5067494 -2.1706968 -2.9266811 -2.0246741
      77      78      79      80      81      82
0.3269993 0.4398570 1.8083732 5.2459472 2.5939761 4.7016539

# Each year's average residual,
# with a 95% t interval from the cars of that year
sapply(split(residuals(fit_quadratic), auto$year), function(e) {
  half <- qt(0.975, length(e) - 1) * sd(e) / sqrt(length(e))
  c(mean = mean(e), lower = mean(e) - half, upper = mean(e) + half)
}) |>
  t()

      mean      lower      upper
70 -0.6162271 -1.9035155 0.6710612
71 -1.7500396 -3.1000185 -0.4000607
72 -1.9387883 -3.1914000 -0.6861766
73 -2.5067494 -3.5607316 -1.4527671
74 -2.1706968 -3.1938231 -1.1475706
75 -2.9266811 -4.5151741 -1.3381881
76 -2.0246741 -3.0755802 -0.9737680
77 0.3269993 -0.6832193 1.3372179
78 0.4398570 -0.7101477 1.5898617
79 1.8083732 0.3167006 3.3000458
80 5.2459472 3.1005363 7.3913580
81 2.5939761 1.1855717 4.0023805
82 4.7016539 3.4594092 5.9438985

fit_with_year <- lm(mpg ~ horsepower + I(horsepower^2) + year, data = auto)
lag_correlation(residuals(fit_with_year))
```

```

[1] 0.3075232

# Neighboring residuals are correlated,
# so resampling single cars would break the order.
# Instead resample blocks of 20 consecutive cars,
# refit the model, and recompute the correlation.
block_bootstrap_lag_ci <- function(model, block = 20, n_boot = 20000) {
  set.seed(571)
  design <- model.matrix(model)
  y <- model.response(model.frame(model))
  n_blocks <- ceiling(nrow(design) / block)
  lag <- vapply(seq_len(n_boot), \(b) {
    starts <- sample(nrow(design) - block + 1, n_blocks, replace = TRUE)
    rows <- head(as.vector(outer(0:(block - 1), starts, "+")), nrow(design))
    lag_correlation(lm.fit(design[rows, ], y[rows])$residuals)
  }, numeric(1))
  unname(quantile(lag, c(0.025, 0.975)))
}
block_bootstrap_lag_ci(fit_quadratic)

[1] 0.3175578 0.5078073

block_bootstrap_lag_ci(fit_with_year)

[1] 0.1445910 0.3820917

year_table_py <- py$year_table
year_min <- year_table_py[which.min(year_table_py$mean), ]
year_max <- year_table_py[which.max(year_table_py$mean), ]

```

No. Neighboring residuals have correlation 0.46 (95% bootstrap interval 0.32 to 0.51). The interval comes from 20,000 resamples of blocks of 20 consecutive cars, which keep the order of the file, with the quadratic refitted and the correlation recomputed on each resample. The average [residual](#) is negative for every year up to 1976 and positive for every later year (Figure 25), ranging from -2.9 (1975, 95% interval -4.5 to -1.3) to 5.2 (1980, 3.1 to 7.4): the model over-predicts the older cars and under-predicts the newer ones. Each year's interval is a 95% *t* interval for the average residual of the cars of that year, and it excludes 0 in 10 of the 13 years. James et al. [32, Sec. 3.3.3], p. 102 calls runs of same-signed neighboring residuals *tracking*. With correlated errors, the estimated standard errors of Definition 7.2 are too small, and the confidence intervals too narrow [32, Sec. 3.3.3], p. 101.

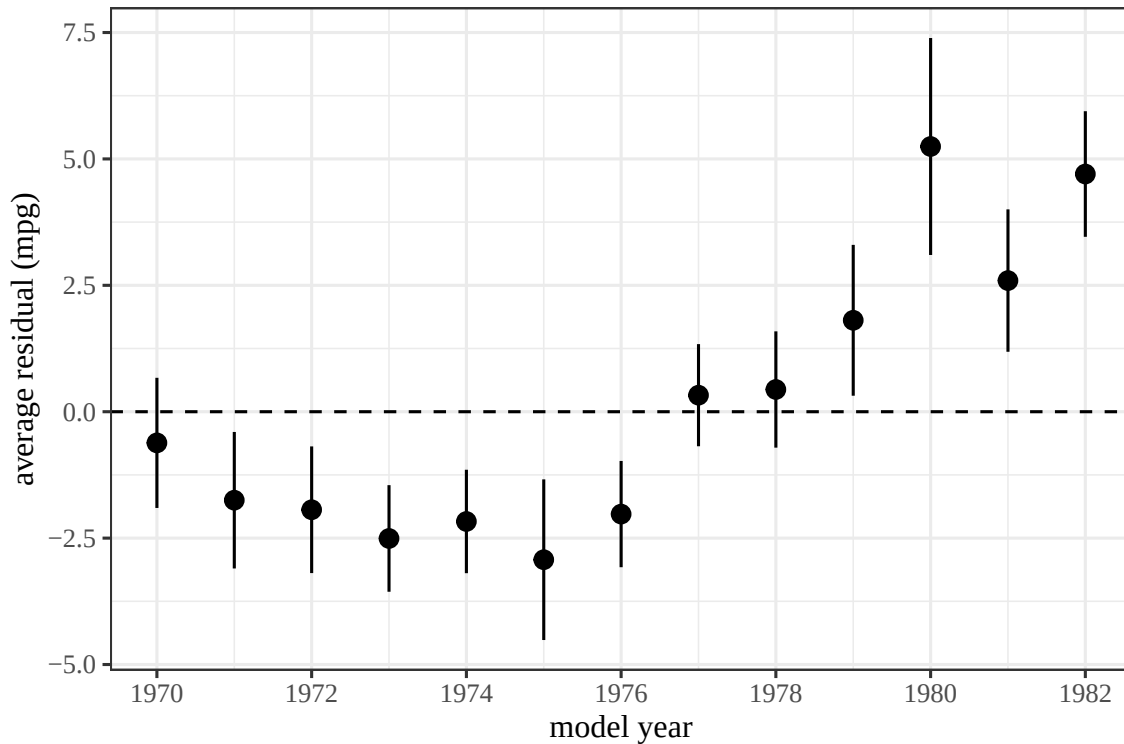


Figure 25: The average residual of the quadratic fit of Exercise 6.17 for each year, with each year's 95% t interval.

Part of the cause is a missing feature. Adding `year` to the model lowers the neighboring correlation to 0.31 (0.14 to 0.38), but does not remove it.

8.4 Outliers

Definition 8.2 (Standardized residual). The **standardized residual** of observation i is its residual divided by its estimated standard deviation,

$$r_i = \frac{e_i}{\hat{\sigma}\sqrt{1-h_i}},$$

where h_i is the leverage of Definition 5.3 and $\hat{\sigma}$ is the residual standard error. Dividing by $\sqrt{1-h_i}$ accounts for high-leverage points having smaller residuals, because they pull the fit toward themselves.

Example 8.3 (Standardized residuals of the quadratic fit). Compute the standardized residuals of the quadratic fit of Exercise 6.17, and look at the car with the highest leverage:

8.4.1 Python

```
PtP_inv_quadratic = np.linalg.inv(Phi_quadratic.T @ Phi_quadratic)
h_quadratic = np.sum(Phi_quadratic @ PtP_inv_quadratic * Phi_quadratic, axis=1)
sigma_quadratic = np.sqrt(np.sum(residuals_quadratic ** 2) / (n - 3))
r_quadratic = residuals_quadratic / (sigma_quadratic * np.sqrt(1 - h_quadratic))
top_leverage = np.argmax(h_quadratic)
print(Auto.loc[top_leverage, ["name", "horsepower", "mpg"]])
```

```

name          pontiac grand prix
horsepower      230
mpg             16.0
Name: 115, dtype: object

```

```

print(h_quadratic[top_leverage], residuals_quadratic[top_leverage],
      r_quadratic[top_leverage])

```

```

0.09359709306399572 1.2281554548362656 0.2949319391422815

```

8.4.2 R

```

r_quadratic <- rstandard(fit_quadratic)
top_leverage <- which.max(hatvalues(fit_quadratic))
auto |>
  dplyr::mutate(id = dplyr::row_number()) |>
  dplyr::filter(id == top_leverage) |>
  dplyr::select(id, name, horsepower, mpg)

```

```

# A tibble: 1 x 4
  id name          horsepower mpg
<int> <chr>          <int> <dbl>
1  116 pontiac grand prix      230  16

```

```

c(leverage = hatvalues(fit_quadratic)[[top_leverage]],
  residual = residuals(fit_quadratic)[[top_leverage]],
  standardized = r_quadratic[[top_leverage]])

```

```

leverage    residual standardized
0.09359709  1.22815545  0.29493194

```

The car with the highest leverage has $h_i = 0.094$, residual $e_i = 1.23$ and standardized residual $r_i = 0.29$. R's `rstandard()` gives the same r_i for every car: the largest difference from the Python values is 4×10^{-13} , which is rounding error.

i Other names for standardized residuals

James et al. [32, Sec. 3.3.3], p. 104 calls r_i a *studentized* residual, and R computes it with `rstandard()`. Other sources keep *studentized* for the version that estimates σ without observation i , which R computes with `rstudent()`. These notes say *standardized*.

Definition 8.3 (Outlier). An **outlier** is an observation whose response is far from what the model predicts. A common rule of thumb flags observation i as a possible outlier when its standardized residual (Definition 8.2) has $|r_i| > 3$ [32, Sec. 3.3.3], p. 104.

Example 8.4 (Outliers in the quadratic fit). List the cars whose standardized residuals in Example 8.3 have $|r_i| > 3$:

8.4.3 Python

```

outliers = np.abs(r_quadratic) > 3
print(Auto.loc[outliers, ["name", "horsepower", "mpg"]].assign(r=r_quadratic[outliers]))

```

```

      name  horsepower  mpg      r
152  mercury monarch      72  15.0 -3.372439
153   ford maverick      72  15.0 -3.372439
320    mazda glc        65  46.6  3.397282

```

```
327 honda civic 1500 gl          67 44.6 3.076382
330      datsum 280-zx          132 32.7 3.646286
```

8.4.4 R

```
auto |>
  dplyr::mutate(id = dplyr::row_number(), r = r_quadratic) |>
  dplyr::filter(abs(r) > 3) |>
  dplyr::select(id, name, horsepower, mpg, r)
```

```
# A tibble: 5 x 5
      id name          horsepower mpg      r
  <int> <chr>          <int> <dbl> <dbl>
1   153 mercury monarch         72  15 -3.37
2   154 ford maverick          72  15 -3.37
3   321 mazda glc             65 46.6  3.40
4   328 honda civic 1500 gl      67 44.6  3.08
5   331 datsum 280-zx          132 32.7  3.65
```

5 of the 392 cars have $|r_i| > 3$, and the largest is $|r_i| = 3.65$. Of these, 3 get more miles per gallon than the model predicts, and 2 get fewer. Under Gaussian noise with constant variance, about 1.1 of 392 cars would have $|r_i| > 3$ by chance, so there are more than chance alone would give. Before removing an outlier, check whether it is a recording error; if it is not, it may be a sign that the model is missing something, such as a feature [32, Sec. 3.3.3], p. 104.

8.5 High-leverage points

Example 8.5 (The highest-leverage car). The leverage h_i of Definition 5.3 measures how unusual car i 's features are. Its average is p^*/n , and James et al. [32, Sec. 3.3.3], p. 106 flags points whose leverage greatly exceeds that average. Find the highest-leverage car in the one-feature fit of Solution 6.5.4:

8.5.1 Python

```
h_simple = np.sum(Phi_simple @ PtP_inv * Phi_simple, axis=1)
top = np.argmax(h_simple)
print(Auto.loc[top, ["name", "horsepower"]], h_simple[top], 2 / n)

name          pontiac grand prix
horsepower          230
Name: 115, dtype: object 0.02975300179092146 0.00510204081632653
```

8.5.2 R

```
h_simple <- hatvalues(fit_simple)
auto |>
  dplyr::mutate(id = dplyr::row_number()) |>
  dplyr::filter(id == which.max(h_simple)) |>
  dplyr::select(id, name, horsepower)

# A tibble: 1 x 3
      id name          horsepower
  <int> <chr>          <int>
1   116 pontiac grand prix      230

c(max = max(h_simple), average = mean(h_simple))

      max      average
0.029753002 0.005102041
```

The pontiac grand prix, with 230 horsepower, has leverage 0.0298, about 5.8 times the average $2/392$. Its horsepower is the largest in the data, far from the sample mean of 104. A high-leverage point is worth checking: since $\hat{y} = Hy$, a change in its response moves its own fitted value by h_i times that change, more than the same change would for any other car.

8.6 Collinearity

Definition 8.4 (Variance inflation factor). Let D have an intercept column and p features $x_1, \dots, x_p$, and let R_j^2 be the coefficient of determination of the regression of feature x_j on the other $p - 1$ features (with an intercept). The **variance inflation factor** (VIF) [35] of feature x_j is

$$\text{VIF}_j \stackrel{\text{def}}{=} \frac{1}{1 - R_j^2}.$$

It is at least 1, and equals 1 exactly when the other features cannot predict x_j at all, in the sample ($R_j^2 = 0$). It grows without bound as x_j approaches a linear combination of the other features ($R_j^2 \rightarrow 1$).

Example 8.6 (Collinearity of horsepower and weight). Definition 8.4 defines the variance inflation factor, $\text{VIF}_j = 1/(1 - R_j^2)$. Compute it for `horsepower` and `weight` in the two-feature fit of Solution 6.5.24:

8.6.1 Python

```
r_hp_weight = np.corrcoef(horsepower, weight)[0, 1]
vif_hp_weight = 1 / (1 - r_hp_weight ** 2)
print(vif_hp_weight)

3.959227866518292

# Bootstrap: resample cars with replacement and recompute the VIF each time.
rng_vif = np.random.default_rng(571)
vif_boot = np.empty(2000)
for b in range(2000):
    rows = rng_vif.integers(0, len(horsepower), len(horsepower))
    r_boot = np.corrcoef(horsepower[rows], weight[rows])[0, 1]
    vif_boot[b] = 1 / (1 - r_boot ** 2)
# The 2.5th and 97.5th percentiles of the resampled VIFs.
vif_ci = np.quantile(vif_boot, [0.025, 0.975])
print(vif_ci)

[3.28696834 4.78555867]
```

8.6.2 R

```
r2_hp <- summary(lm(horsepower ~ weight, data = auto))$r.squared
1 / (1 - r2_hp)

[1] 3.959228

# Bootstrap: resample cars with replacement and recompute the VIF each time.
set.seed(571)
vif_boot <- replicate(2000, {
  rows <- sample(nrow(auto), replace = TRUE)
  1 / (1 - cor(auto$horsepower[rows], auto$weight[rows])^2)
})
# The 2.5th and 97.5th percentiles of the resampled VIFs.
quantile(vif_boot, c(0.025, 0.975))
```

2.5% 97.5%
 3.267544 4.794859

With only two features, R_j^2 is the squared correlation, so both features have the same VIF, 3.96. The variance of each coefficient is 3.96 times what it would be if the two features were uncorrelated. A 95% bootstrap percentile interval for the VIF, from 2000 resamples of the cars, runs from 3.29 to 4.79. The bootstrap fits here because the VIF is a nonlinear function of the correlation, and the interval describes which cars were drawn. The interval lies entirely below the common warning level of 5 to 10 [32, Sec. 3.3.3], p. 108, and the VIF explains why the `horsepower` standard error grows from 0.0064 in the one-feature fit to 0.011 in Example 7.7.

i Remark (Variance inflation from collinear features)

Remark 8.1 (Variance inflation from collinear features). In a model with an intercept, the variance of the least squares estimate of the coefficient of feature x_j is

$$\text{Var}(\hat{\beta}_j) = \frac{\sigma^2}{(n-1)s_{x_j}^2} \text{VIF}_j,$$

where σ^2 is the error variance and $s_{x_j}^2$ is the sample variance of x_j . The formula follows from the covariance of the least squares estimate (Theorem 7.1), and James et al. [32, Sec. 3.3.3] defines the VIF. The factor $\sigma^2/((n-1)s_{x_j}^2)$ is the variance of the slope in a simple linear regression on x_j alone, so VIF_j (Definition 8.4) is the factor by which the other features inflate the variance of $\hat{\beta}_j$. Features that are nearly collinear, such as the powers of one feature in Remark 6.5, have large VIFs and imprecise coefficients.

9 Choosing features

This section follows [32, Sec. 3.2.1] and 3.2.2. Chapter 6 of the same book treats subset selection in more depth.

9.1 A coefficient depends on the other features

Exercise 9.1 (Does faster acceleration raise fuel economy?). In the Auto data, `acceleration` is the time in seconds a car takes to go from 0 to 60 miles per hour, so a larger value means a slower car. Fit `mpg` on `acceleration` alone.

Solution

Solution 9.1.

9.1.1 Python

```
acceleration = Auto["acceleration"].to_numpy(float)
beta_acc, Phi_acc = fit([acceleration])
beta_acc_hp, Phi_acc_hp = fit([acceleration, horsepower])
# Each coefficient with its 95% confidence interval
print(np.column_stack([beta_acc, coef_ci(beta_acc, Phi_acc)]))

[[4.8332498  0.80576513  8.86073448]
 [1.19762419  0.94245663  1.45279175]]

print(np.column_stack([beta_acc_hp, coef_ci(beta_acc_hp, Phi_acc_hp)]))

[[52.55932488  47.47298392  57.64566585]]
```

```

[-0.60977221 -0.8464122 -0.37313222]
[-0.18796643 -0.20492766 -0.17100519]]

print(np.corrcoef(acceleration, horsepower)[0, 1])

-0.6891955103342359

# bootstrap_corr_ci() is one of the interval helpers
corr_ci_acc_hp = bootstrap_corr_ci(acceleration, horsepower)
print(corr_ci_acc_hp)

[-0.73918546 -0.63352643]

```

9.1.2 R

```

# The tables include each coefficient's 95% confidence interval
lm(mpg ~ acceleration, data = auto) |>
  parameters::parameters() |>
  parameters::print_md(digits = 3)

```

Parameter	Coefficient	SE	95% CI	t(390)	p
(Intercept)	4.833	2.048	(0.806, 8.861)	2.359	0.019
acceleration	1.198	0.130	(0.942, 1.453)	9.228	< .001

```

lm(mpg ~ acceleration + horsepower, data = auto) |>
  parameters::parameters() |>
  parameters::print_md(digits = 3)

```

Parameter	Coefficient	SE	95% CI	t(389)	p
(Intercept)	52.559	2.587	(47.473, 57.646)	20.316	< .001
acceleration	-0.610	0.120	(-0.846, -0.373)	-5.066	< .001
horsepower	-0.188	0.009	(-0.205, -0.171)	-21.788	< .001

```

cor(auto$acceleration, auto$horsepower)

[1] -0.6891955

# boot_cor_ci() is one of the interval helpers
boot_cor_ci(auto$acceleration, auto$horsepower)

[1] -0.7378244 -0.6332126

coef_ci_acc <- py$coef_ci(py$beta_acc, py$Phi_acc)
coef_ci_acc_hp <- py$coef_ci(py$beta_acc_hp, py$Phi_acc_hp)

```

Exercise 9.2 (Does faster acceleration raise fuel economy? (part 2)). Use the setup of Exercise 9.1. What is the sign of the slope^a?

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Solution

Solution 9.2. On its own, **acceleration** has slope^a 1.198, so slower cars get more miles per gallon.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Exercise 9.3 (Does faster acceleration raise fuel economy? (part 3)). Use the setup of Exercise 9.1. Is it significant?

Solution

Solution 9.3. The 95% confidence interval^a for the slope^b is 0.942 to 1.453, and $p = 1.8 \times 10^{-18}$, so the slope is significant at the 5% level.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Exercise 9.4 (Does faster acceleration raise fuel economy? (part 4)). Use the setup of Exercise 9.1. Add **horsepower** to the model. What happens to the **acceleration** coefficient?

Solution

Solution 9.4. With **horsepower** in the model, the **acceleration** coefficient is -0.610 (-0.846 to -0.373). It has changed sign. Each coefficient's interval is its t interval, the estimate^a plus or minus a t quantile times its standard error^b.

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^b<https://morrison-lab.github.io/sds/estimation.html#def-SE>

Exercise 9.5 (Does faster acceleration raise fuel economy? (part 5)). Use the setup of Exercise 9.1. Explain the change.

Solution

Solution 9.5. The two features have correlation^a -0.69 (95% bootstrap interval -0.74 to -0.63, from 20,000 resamples of the cars): cars with small engines are slow. In the one-feature fit, **acceleration** stands in for engine power, and gets the credit for the association between engine power and **mpg**. Holding **horsepower** fixed, a slower car gets fewer miles per gallon, not more. James et al. [32, Sec. 3.2.1], p. 83 calls such a feature a *surrogate*. Their example is a regression of shark attacks on ice cream sales at a beach: the slope^b is positive, because hot weather brings both, and ice cream sales are no longer significant once temperature is in the model. As in Solution 6.5.24, a feature's coefficient depends on which other features are in the model.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-correlation>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

9.2 Forward and backward selection

Exercise 9.6 (How many models?). Each model below keeps the intercept and uses some subset of p candidate features.
How many such models are there?

Solution

Solution 9.6. Each feature is either in the model or out of it, so there are 2^p models.

Exercise 9.7 (How many models? (part 2)). Use the setup of Exercise 9.6.
How many are there for 8 features, and for 30?

Solution

Solution 9.7. For $p = 8$ there are $2^8 = 256$ models, few enough to fit them all. For $p = 30$ there are $2^{30} = 1,073,741,824$ models, too many to fit [32, Sec. 3.2.2], p. 87. The procedures that follow fit far fewer.

Definition 9.1 (Forward selection). *Forward selection* starts from the model with only an intercept. At each step, it fits every model that adds one more feature, and keeps the one with the lowest residual sum of squares (RSS). It stops when a stopping rule is met [32, Sec. 3.2.2], p. 87, [36].

Example 9.1 (Forward selection on the Auto data). Run forward selection (Definition 9.1) for mpg over eight candidate features: `cylinders`, `displacement`, `horsepower`, `weight`, `acceleration`, `year`, and the `european` and `japanese` indicators of Exercise 6.22. Record the order in which the features enter and the RSS after each step.

9.2.1 Python

```
candidates = {
    name: Auto[name].to_numpy(float)
    for name in ["cylinders", "displacement", "horsepower", "weight",
                "acceleration", "year"]
}
candidates["european"] = european
candidates["japanese"] = japanese

def rss(names):
    beta_hat, Phi = fit([candidates[k] for k in names])
    return np.sum((y - Phi @ beta_hat) ** 2)

forward_order, forward_rss = [], []
remaining = list(candidates)
while remaining:
    best = min(remaining, key=lambda k: rss(forward_order + [k]))
    forward_order.append(best)
    remaining.remove(best)
    forward_rss.append(rss(forward_order))
print(pd.DataFrame({"added": forward_order, "rss": forward_rss}))
```

	added	rss
0	weight	7321.233706
1	year	4568.952042
2	japanese	4472.554527
3	european	4310.409498
4	displacement	4271.964759
5	horsepower	4221.333733
6	cylinders	4194.483283
7	acceleration	4187.391678

9.2.2 R

```

auto <- dplyr::mutate(auto,
                      european = as.numeric(origin == 2),
                      japanese = as.numeric(origin == 3))
candidates <- c("cylinders", "displacement", "horsepower", "weight",
               "acceleration", "year", "european", "japanese")

rss <- function(names, data) {
  fit <- lm(reformulate(names, response = "mpg"), data = data)
  sum(residuals(fit)^2)
}

forward_order <- character(0)
forward_rss <- numeric(0)
remaining <- candidates
while (length(remaining) > 0) {
  scores <- sapply(remaining, function(k) rss(c(forward_order, k), auto))
  best <- names(which.min(scores))
  forward_order <- c(forward_order, best)
  forward_rss <- c(forward_rss, min(scores))
  remaining <- setdiff(remaining, best)
}
tibble::tibble(added = forward_order, rss = forward_rss)

```

```

# A tibble: 8 x 2
  added      rss
  <chr>    <dbl>
1 weight  7321.
2 year    4569.
3 japanese 4473.
4 european 4310.
5 displacement 4272.
6 horsepower 4221.
7 cylinders 4194.
8 acceleration 4187.

```

weight enters first, with RSS 7321, then year, which brings it down to 4569. After that the gains are small: the last 6 features together lower the RSS only to 4187. The RSS falls at every step, as it must (Solution 6.5.24), so it cannot by itself say when to stop. These are training RSS values, computed on the cars used to fit, so they get no confidence intervals: the residuals are not independent draws, and the values are biased low.

Definition 9.2 (Backward selection). *Backward selection* starts from the model with all p features. At each step, it removes the feature with the largest p -value (Definition 7.4) and refits. It stops when a stopping rule is met, such as every remaining p -value being below a threshold [32, Sec. 3.2.2], p. 87, [36].

Example 9.2 (Backward selection on the Auto data). Run backward selection (Definition 9.2) over the eight features of Example 9.1, stopping when every remaining p -value is below 0.05.

9.2.3 Python

```
def p_values(names):
    beta_hat, Phi = fit([candidates[k] for k in names])
    df = n - Phi.shape[1]
    sigma2 = np.sum((y - Phi @ beta_hat) ** 2) / df
    se = np.sqrt(sigma2 * np.diag(np.linalg.inv(Phi.T @ Phi)))
    return 2 * stats.t.sf(np.abs(beta_hat / se), df)[1:]

backward_kept = list(candidates)
backward_dropped, backward_p = [], []
while np.max(p_values(backward_kept)) >= 0.05:
    p = p_values(backward_kept)
    backward_dropped.append(backward_kept[np.argmax(p)])
    backward_p.append(np.max(p))
    backward_kept.remove(backward_dropped[-1])
print(pd.DataFrame({"dropped": backward_dropped, "p": backward_p}))
```

	dropped	p
0	acceleration	0.421101
1	cylinders	0.117742

```
print(backward_kept)
```

```
['displacement', 'horsepower', 'weight', 'year', 'european', 'japanese']
```

9.2.4 R

```
backward_kept <- candidates
repeat {
  fit <- lm(reformulate(backward_kept, response = "mpg"), data = auto)
  p <- coef(summary(fit))[-1, 4]
  if (max(p) < 0.05) break
  print(c(dropped = names(which.max(p)), p = signif(max(p), 2)))
  backward_kept <- setdiff(backward_kept, names(which.max(p)))
}
```

	dropped	p
	"acceleration"	"0.42"

	dropped	p
	"cylinders"	"0.12"

```
backward_kept
```

```
[1] "displacement" "horsepower"   "weight"       "year"         "european"
[6] "japanese"
```

Backward selection drops acceleration ($p = 0.42$) and then cylinders ($p = 0.12$), and keeps 6 features: displacement, horsepower, weight, year, european, japanese. Forward selection added those same two features last, so here the two procedures agree on which features matter least. They need not agree in general.

i Remark (Mixed selection, and choosing when to stop)

Remark 9.1 (Mixed selection, and choosing when to stop). *Mixed selection* adds features as forward selection does, but after each step it removes any feature whose p -value has risen above a threshold. It can drop a feature that forward selection added early and that later became redundant. Backward selection needs $p < n$, since it starts by fitting every feature; forward selection does not [32, Sec. 3.2.2], p. 87.

None of the three procedures is guaranteed to find the best of the 2^p models. A stopping rule based on training error alone always takes the largest model, as Example 9.1 shows. Model Selection chooses

with cross-validation instead.

Exercise 9.8 (Forward selection on baseball salaries, sized by cross-validation). The `Hitters` data hold the 1986 and career statistics of Major League baseball players, with their 1987 salaries in thousands of dollars. Drop the players whose salary is missing, and code `League`, `Division` and `NewLeague` as 0/1 indicators, which leaves 19 candidate features for the salary. Randomly split the players into a training half and a test half.

On the training half, run forward selection (Definition 9.1) with the RSS as the score, and record the order in which the features enter.

Solution

Solution 9.8.

9.2.5 R

```
hs <- na.omit(read.csv("data/hitters.csv"))
hs_x <- model.matrix(Salary ~ ., data = hs)[, -1]
hs_y <- hs$Salary

set.seed(1)
hs_train <- sample(nrow(hs), nrow(hs) / 2)
xt <- hs_x[hs_train, ]
yt <- hs_y[hs_train]

rss_of <- function(x, y, cols) {
  sum(lm.fit(cbind(1, x[, cols, drop = FALSE]), y)$residuals^2)
}

# Forward selection by the RSS: the features in the order they enter.
hs_forward <- function(x, y) {
  chosen <- integer(0)
  while (length(chosen) < ncol(x)) {
    rest <- setdiff(seq_len(ncol(x)), chosen)
    scores <- sapply(rest, function(j) rss_of(x, y, c(chosen, j)))
    chosen <- c(chosen, rest[which.min(scores)])
  }
  chosen
}

# Predictions of the model with the first `size` features of `entry`.
hs_predict <- function(x, y, entry, size, x_new) {
  cols <- entry[seq_len(size)]
  fit <- lm.fit(cbind(1, x[, cols, drop = FALSE]), y)
  drop(cbind(1, x_new[, cols, drop = FALSE]) %*% fit$coefficients)
}

entry_all <- hs_forward(xt, yt)
colnames(xt)[entry_all]

[1] "CRBI"      "Walks"     "DivisionW" "Years"     "PutOuts"
[6] "CHits"     "CAtBat"    "LeagueN"   "CHmRun"    "Assists"
[11] "AtBat"     "HmRun"     "NewLeagueN" "Errors"    "CRuns"
[16] "CWalks"    "Hits"      "Runs"      "RBI"
```

9.2.6 Python

```
hs_py = pd.read_csv("data/hitters.csv").dropna()
hs_x = pd.get_dummies(hs_py.drop(columns="Salary"), drop_first=True)
hs_names = list(hs_x.columns)
hs_x = hs_x.to_numpy(dtype=float)
hs_y = hs_py["Salary"].to_numpy()

hs_rng = np.random.default_rng(1)
hs_train = hs_rng.permutation(len(hs_y))[: len(hs_y) // 2]
hs_test = np.setdiff1d(np.arange(len(hs_y)), hs_train)
xt = hs_x[hs_train]
yt = hs_y[hs_train]

def hs_design(x, cols):
    return np.column_stack([np.ones(len(x)), x[:, cols]])

def hs_beta(x, y, cols):
    return np.linalg.lstsq(hs_design(x, cols), y, rcond=None)[0]

def hs_rss(x, y, cols):
    r = y - hs_design(x, cols) @ hs_beta(x, y, cols)
    return r @ r

# Forward selection by the RSS: the features in the order they enter.
def hs_forward_py(x, y):
    chosen = []
    while len(chosen) < x.shape[1]:
        rest = [j for j in range(x.shape[1]) if j not in chosen]
        chosen.append(min(rest, key=lambda j: hs_rss(x, y, chosen + [j])))
    return chosen

# Predictions of the model with the first `size` features of `entry`.
def hs_predict_py(x, y, entry, size, x_new):
    cols = entry[:size]
    return hs_design(x_new, cols) @ hs_beta(x, y, cols)

entry_all_py = hs_forward_py(xt, yt)
[hs_names[j] for j in entry_all_py]
```

['CRBI', 'Hits', 'Division_W', 'AtBat', 'PutOuts', 'Years', 'Assists', 'Walks', 'CWalks', 'League_N',

1. The two languages split the players at random in different ways, so their numbers differ. The numbers below are from the R split.

On the training half, the features enter in the order CRBI, Walks, DivisionW, Years, PutOuts, and so on.

Exercise 9.9 (Forward selection on baseball salaries, sized by cross-validation (part 2)). Use the setup of Exercise 9.8.

Randomly assign the training players to ten folds. For each size $m = 1, \dots, 19$, compute the 10-fold cross-validation root mean squared error (RMSE) of the model that has the first m features chosen by [forward selection](#). Run the forward selection again on the other nine folds of each split.

Solution

Solution 9.9.

9.2.7 R

```
# The selection is repeated on the other nine folds of each split.
set.seed(2)
folds <- sample(rep(1:10, length.out = nrow(xt)))
cv_sq <- matrix(NA_real_, nrow(xt), ncol(xt))
for (f in 1:10) {
  held <- folds == f
  entry_f <- hs_forward(xt[!held, ], yt[!held])
  for (m in seq_len(ncol(xt))) {
    pred <- hs_predict(xt[!held, ], yt[!held], entry_f, m, xt[held, ], drop = FALSE)
    cv_sq[held, m] <- (yt[held] - pred)^2
  }
}
cv_rmse <- sqrt(colMeans(cv_sq))
m_best <- which.min(cv_rmse)
data.frame(size = seq_along(cv_rmse), cv_rmse = round(cv_rmse))
```

A tibble: 19 x 2

	size	cv_rmse
	<int>	<dbl>
1	1	350
2	2	330
3	3	320
4	4	322
5	5	330
6	6	330
7	7	335
8	8	338
9	9	347
10	10	348
11	11	346
12	12	350
13	13	350
14	14	347
15	15	349
16	16	351
17	17	352
18	18	351
19	19	351

9.2.8 Python

```
# The selection is repeated on the other nine folds of each split.
folds_py = np.random.default_rng(2).permutation(np.arange(len(yt)) % 10)
cv_sq_py = np.empty((len(yt), xt.shape[1]))
for f in range(10):
    held = folds_py == f
    entry_f = hs_forward_py(xt[~held], yt[~held])
    for m in range(1, xt.shape[1] + 1):
        pred = hs_predict_py(xt[~held], yt[~held], entry_f, m, xt[held])
        cv_sq_py[held, m - 1] = (yt[held] - pred) ** 2
cv_rmse_py = np.sqrt(cv_sq_py.mean(axis=0))
m_best_py = int(np.argmin(cv_rmse_py)) + 1
print(pd.DataFrame({"size": range(1, xt.shape[1] + 1), "cv_rmse": cv_rmse_py.round()}))
```

	size	cv_rmse
0	1	409.0
1	2	420.0
2	3	400.0
3	4	390.0
4	5	383.0
5	6	385.0
6	7	376.0
7	8	392.0
8	9	395.0
9	10	390.0
10	11	376.0
11	12	376.0
12	13	373.0
13	14	372.0
14	15	374.0
15	16	378.0
16	17	378.0
17	18	378.0
18	19	378.0

2. Cross-validation chooses $m = 3$ features, with a cross-validation RMSE of 320 thousand dollars.

Exercise 9.10 (Forward selection on baseball salaries, sized by cross-validation (part 3)). Use the setup of Exercise 9.8.

Choose the size m with the smallest cross-validation RMSE, refit that model on the whole training half, and compute its root mean squared error on the test half.

Solution

Solution 9.10.

9.2.9 R

```
test_rmse_of <- function(m) {
  pred <- hs_predict(xt, yt, entry_all, m, hs_x[-hs_train, ])
  sqrt(mean((hs_y[-hs_train] - pred)^2))
}
c(chosen = test_rmse_of(m_best), all = test_rmse_of(ncol(xt)))

      chosen      all
389.6647 410.6011
```

9.2.10 Python

```
def hs_test_rmse(m):
    pred = hs_predict_py(xt, yt, entry_all_py, m, hs_x[hs_test])
    return float(np.sqrt(np.mean((hs_y[hs_test] - pred) ** 2)))

print("chosen size", m_best_py, "- test RMSE", round(hs_test_rmse(m_best_py)),
      "- all features", round(hs_test_rmse(xt.shape[1])))
```

chosen size 14 - test RMSE 332 - all features 332

The model that cross-validation chooses has a test RMSE of 390 thousand dollars on the test half.

Exercise 9.11 (Forward selection on baseball salaries, sized by cross-validation (part 4)). Use the setup of Exercise 9.8.

Compare it with the test error of the model that uses all 19 features.

Solution

Solution 9.11. The model with all 19 features has a test RMSE of 411 thousand dollars, which is higher than the test RMSE of the model that cross-validation chooses.

Exercise 9.12 (Forward selection on baseball salaries, sized by cross-validation (part 5)). Use the setup of Exercise 9.8.

Why must the [forward selection](#) be run again inside each split, rather than once on the whole training half?

Solution

Solution 9.12. 4. The selection is repeated inside each split because the features chosen on the whole training half have already seen every training player's salary. The held-out fold would then not be new data for the selection, and the cross-validation error would be too small.

This exercise follows the subset selection lab of [3, Sec. 6.5.1], which uses the same `Hitters` data. The R lab picks the size of the model with `regsubsets`, and the Python lab uses a search from the ISLP package [37]. Here we write the [forward selection](#) of this chapter ourselves, and we choose the size by cross-validation.

10 Lab: the book's Chapter 3 lab

These exercises follow the lab of [32, Sec. 3.6], on the Boston and Carseats data, step by step. The Python code follows the book's Chapter 3 lab notebook⁸, which fits models with `statsmodels`. That notebook builds its design matrices with `ModelSpec` from the ISLP package; here we build them with `pandas`, because installing ISLP also installs PyTorch. The R `tidymodels` code follows the Chapter 3 lab of ISLR-`tidymodels-labs`⁹.

Exercise 10.1 (Lab: fit a line to the Boston data). This exercise and the four after it follow the steps of the book's Chapter 3 lab [32, Sec. 3.6]. Each solution shows the code in Python, in R with `tidymodels`, and in base R.

Download the Boston data from `data/boston.csv`^a and save it as `data/boston.csv` next to your code. It has 506 Boston neighborhoods from the 1970 census [38]. The copy here is the one in the Python package ISLP and the R package ISLR2. The response `medv` is the median home value, in thousands of dollars, and `lstat` is the percentage of the population with lower socioeconomic status.

Fit `medv` on `lstat`. Report each coefficient with its standard error^b.

⁸https://github.com/intro-stat-learning/ISLP_labs/blob/main/Ch03-linreg-lab.ipynb

⁹<https://github.com/Morrison-Lab/ISLR-tidymodels-labs/blob/main/03-linear-regression.qmd>

^a<https://morrison-lab.github.io/lds/chapters/data/boston.csv>
^b<https://morrison-lab.github.io/sds/estimation.html#def-SE>

Solution

Solution 10.1.

10.0.1 Python

```
import numpy as np
import pandas as pd
import statsmodels.api as sm

# ISLP's load_data("Boston") reads this same file.
Boston = pd.read_csv("data/boston.csv")
medv = Boston["medv"]

X_simple = pd.DataFrame({"intercept": np.ones(Boston.shape[0]),
                        "lstat": Boston["lstat"]})
boston_simple = sm.OLS(medv, X_simple).fit()
# Each coefficient with its standard error and 95% confidence interval
print(pd.concat([boston_simple.params, boston_simple.bse, boston_simple.conf_int()], axis=1)
      .set_axis(["coef", "std err", "lower", "upper"], axis=1))

           coef    std err   lower    upper
intercept  34.553841  0.562627  33.448457  35.659225
lstat      -0.950049  0.038733  -1.026148  -0.873951

new_X = pd.DataFrame({"intercept": np.ones(3), "lstat": [5, 10, 15]})
new_predictions = boston_simple.get_prediction(new_X)
print(new_predictions.predicted_mean)

[29.80359411 25.05334734 20.30310057]

print(new_predictions.conf_int(alpha=0.05))

[[29.00741194 30.59977628]
 [24.47413202 25.63256267]
 [19.73158815 20.87461299]]

print(new_predictions.conf_int(obs=True, alpha=0.05))

[[17.56567478 42.04151344]
 [12.82762635 37.27906833]
 [ 8.0777421  32.52845905]]
```

```

import matplotlib.pyplot as plt
import numpy as np

lstat_grid = np.linspace(Boston["lstat"].min(), Boston["lstat"].max(), 100)
# The band is the 95% confidence interval for the mean medv at each lstat
band = boston_simple.get_prediction(
    pd.DataFrame({"intercept": 1.0, "lstat": lstat_grid})).conf_int(alpha=0.05)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3.5), layout="constrained")
ax1.scatter(Boston["lstat"], medv, s=10, alpha=0.5);
ax1.fill_between(lstat_grid, band[:, 0], band[:, 1], color="C3", alpha=0.3);
ax1.plot(lstat_grid,
         boston_simple.params["intercept"] + boston_simple.params["lstat"] * lstat_grid,
         color="C3");
ax1.set_xlabel("lstat");
ax1.set_ylabel("medv");
ax2.scatter(boston_simple.fittedvalues, boston_simple.resid, s=10, alpha=0.5);
ax2.axhline(0, color="k", linestyle="--");
ax2.set_xlabel("Fitted value");
ax2.set_ylabel("Residual");
plt.show()

```

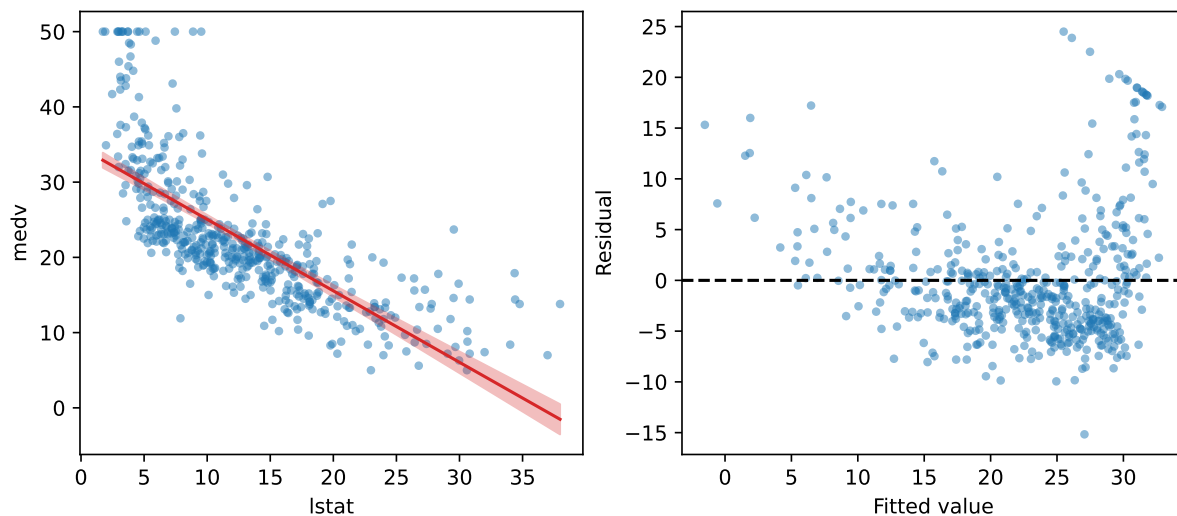


Figure 26: Left: median home value against `lstat` for the Boston data, with the least squares line and its 95% confidence band for the mean. Right: the residual plot of that fit.

10.0.2 R (tidymodels)

```

library(tidymodels)

boston <- read.csv("data/boston.csv")
lm_spec <- linear_reg() |> set_engine("lm")

tm_boston_simple <- lm_spec |> fit(medv ~ lstat, data = boston)
# Each coefficient with its standard error and 95% confidence interval
tidy(tm_boston_simple, conf.int = TRUE)

# A tibble: 2 x 7
  term          estimate std.error statistic    p.value conf.low conf.high
  <chr>          <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>

```

1 (Intercept)	34.6	0.563	61.4	3.74e-236	33.4	35.7
2 lstat	-0.950	0.0387	-24.5	5.08e- 88	-1.03	-0.874

```
new_lstat <- tibble(lstat = c(5, 10, 15))
predict(tm_boston_simple, new_data = new_lstat)
```

```
# A tibble: 3 x 1
  .pred
  <dbl>
1  29.8
2  25.1
3  20.3
```

```
predict(tm_boston_simple, new_data = new_lstat, type = "conf_int")
```

```
# A tibble: 3 x 2
  .pred_lower .pred_upper
  <dbl>         <dbl>
1    29.0         30.6
2    24.5         25.6
3    19.7         20.9
```

```
predict(tm_boston_simple, new_data = new_lstat, type = "pred_int")
```

```
# A tibble: 3 x 2
  .pred_lower .pred_upper
  <dbl>         <dbl>
1    17.6         42.0
2    12.8         37.3
3     8.08        32.5
```

```
library(tidymodels)
```

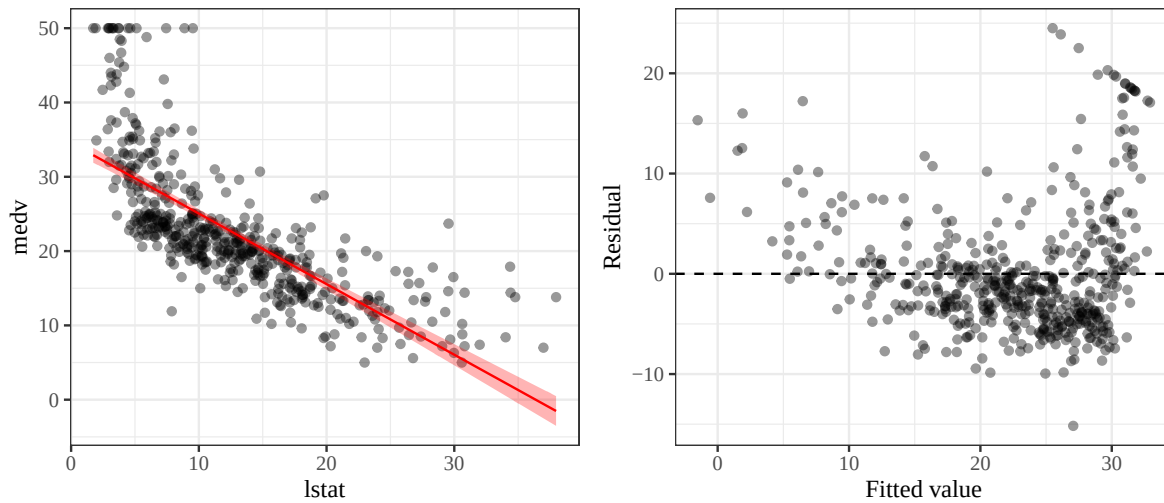
```
# The band is the 95% confidence interval for the mean medv at each lstat
lstat_grid <- tibble(lstat = seq(min(boston$lstat), max(boston$lstat),
                                length.out = 100))
```

```
boston_band <- bind_cols(
  lstat_grid,
  predict(tm_boston_simple, new_data = lstat_grid),
  predict(tm_boston_simple, new_data = lstat_grid, type = "conf_int")
)
```

```
boston_augmented <- augment(tm_boston_simple, new_data = boston)
```

```
p_scatter <- ggplot() +
  geom_point(data = boston, aes(lstat, medv), alpha = 0.4) +
  geom_ribbon(data = boston_band,
            aes(lstat, ymin = .pred_lower, ymax = .pred_upper),
            fill = "red", alpha = 0.3) +
  geom_line(data = boston_band, aes(lstat, .pred), colour = "red")
p_resid <- ggplot(boston_augmented, aes(.pred, .resid)) +
  geom_point(alpha = 0.4) +
  geom_hline(yintercept = 0, linetype = "dashed") +
  labs(x = "Fitted value", y = "Residual")
```

```
p_scatter
p_resid
```



10.0.3 R (base)

```
boston <- read.csv("data/boston.csv")
lm_boston_simple <- lm(medv ~ lstat, data = boston)
# The table includes each coefficient's standard error
# and 95% confidence interval
parameters::parameters(lm_boston_simple) |>
  parameters::print_md(digits = 3)
```

Parameter	Coefficient	SE	95% CI	t(504)	p
(Intercept)	34.554	0.563	(33.448, 35.659)	61.415	< .001
lstat	-0.950	0.039	(-1.026, -0.874)	-24.528	< .001

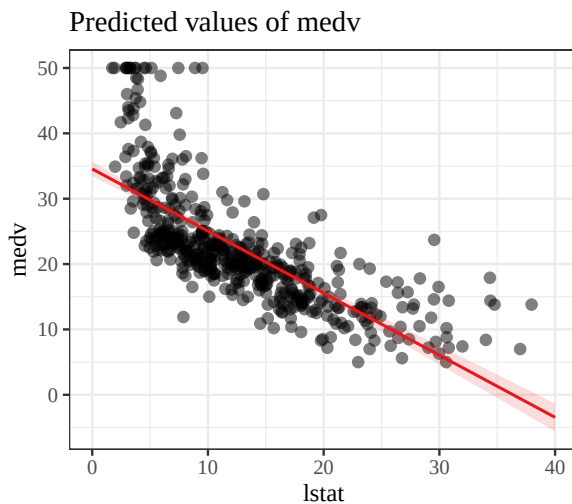
```
new_lstat <- tibble::tibble(lstat = c(5, 10, 15))
predict(lm_boston_simple, new_lstat, interval = "confidence")
```

	fit	lwr	upr
1	29.80359	29.00741	30.59978
2	25.05335	24.47413	25.63256
3	20.30310	19.73159	20.87461

```
predict(lm_boston_simple, new_lstat, interval = "prediction")
```

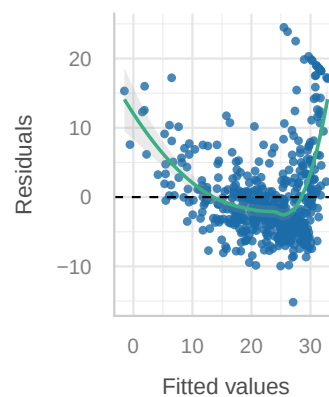
	fit	lwr	upr
1	29.80359	17.565675	42.04151
2	25.05335	12.827626	37.27907
3	20.30310	8.077742	32.52846

```
sjPlot::plot_model(lm_boston_simple, type = "pred", terms = "lstat",
  show.data = TRUE)
performance::check_model(lm_boston_simple, check = "linearity") |>
  plot()
```



Linearity

Reference line should be flat and horizontal



The fitted line is $\widehat{\text{medv}} = 34.55 - 0.95 \times \text{lstat}$, with standard errors 0.563 and 0.039. The 95% confidence intervals, the estimate plus or minus a t quantile times its standard error, are 33.45 to 35.66 for the intercept and -1.026 to -0.874 for the slope. Each extra percentage point of `lstat` goes with a median home value about 950 dollars lower (95% confidence interval 874 to 1026 dollars lower).

Exercise 10.2 (Lab: fit a line to the Boston data (part 2)). Use the setup of Exercise 10.1. Predict `medv` for `lstat` values 5, 10 and 15.

Solution

Solution 10.2. At `lstat` = 10, the predicted `medv` is 25.05.

Exercise 10.3 (Lab: fit a line to the Boston data (part 3)). Use the setup of Exercise 10.1. Give the 95% confidence interval^a for the mean^b response at each value.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Solution

Solution 10.3. The confidence interval^a for the mean^b response at `lstat` = 10 is (24.47, 25.63).

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Exercise 10.4 (Lab: fit a line to the Boston data (part 4)). Use the setup of Exercise 10.1. Give the 95% prediction interval at each value.

Solution

Solution 10.4. The prediction interval at `lstat` = 10 is (12.83, 37.28).

Exercise 10.5 (Lab: fit a line to the Boston data (part 5)). Use the setup of Exercise 10.1. Why is the prediction interval wider?

Solution

Solution 10.5. The confidence interval^a of Definition 7.7 covers only the error in the estimated^b line. The prediction interval of Definition 7.8 also covers the noise of one new neighborhood around that line, so it is much wider.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Exercise 10.6 (Lab: fit a line to the Boston data (part 6)). Use the setup of Exercise 10.1. Draw a scatter plot of `medv` against `lstat` with the fitted line.

Solution

Solution 10.6. The left panel of the figure in Solution 10.0.3 is that scatter plot.

Exercise 10.7 (Lab: fit a line to the Boston data (part 7)). Use the setup of Exercise 10.1. Draw the residual plot of Definition 8.1.

Solution

Solution 10.7. The right panel of the same figure is the residual plot.

Exercise 10.8 (Lab: fit a line to the Boston data (part 8)). Use the setup of Exercise 10.1. What do the plots say about a straight line for these data?

Solution

Solution 10.8. The points curve: they lie above the line at both ends and below it in the middle. The residual plot shows the same pattern as a U shape, the sign of non-linearity in Definition 8.1. Exercise 10.16 adds a squared term for it.

Exercise 10.9 (Lab: many features on the Boston data). Fit `medv` on `lstat` and `age`, the percentage of owner-occupied homes built before 1940. How does the `lstat` coefficient compare with that of Exercise 10.1?

Solution

Solution 10.9.

10.0.4 Python

```
import pandas as pd
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor as VIF
```

```
X_lstat_age = sm.add_constant(Boston[["lstat", "age"]])
boston_lstat_age = sm.OLS(medv, X_lstat_age).fit()
print(boston_lstat_age.summary2().tables[1])
```

	Coef.	Std.Err.	t	P> t	[0.025	0.975]
const	33.222761	0.730847	45.457881	2.943785e-180	31.786872	34.658650
lstat	-1.032069	0.048191	-21.416330	8.419554e-73	-1.126748	-0.937389
age	0.034544	0.012225	2.825605	4.906776e-03	0.010525	0.058564

```
X_all = sm.add_constant(Boston.drop(columns="medv"))
boston_all = sm.OLS(medv, X_all).fit()
print(boston_all.summary2().tables[1])
```

	Coef.	Std.Err.	t	P> t	[0.025	0.975]
const	41.617270	4.936039	8.431308	3.794467e-16	31.919001	51.315539
crim	-0.121389	0.033000	-3.678399	2.605302e-04	-0.186227	-0.056550
zn	0.046963	0.013879	3.383750	7.719966e-04	0.019694	0.074233
indus	0.013468	0.062145	0.216715	8.285201e-01	-0.108633	0.135569
chas	2.839993	0.870007	3.264333	1.173458e-03	1.130614	4.549373
nox	-18.758022	3.851355	-4.870500	1.502073e-06	-26.325116	-11.190928
rm	3.658119	0.420246	8.704714	4.808025e-17	2.832425	4.483813
age	0.003611	0.013329	0.270882	7.865948e-01	-0.022579	0.029800
dis	-1.490754	0.201623	-7.393779	6.170894e-13	-1.886899	-1.094608
rad	0.289405	0.066908	4.325417	1.843595e-05	0.157945	0.420864
tax	-0.012682	0.003801	-3.336504	9.124487e-04	-0.020150	-0.005214
ptratio	-0.937533	0.132206	-7.091443	4.630167e-12	-1.197290	-0.677776
lstat	-0.552019	0.050659	-10.896814	6.392118e-25	-0.651553	-0.452485

```
vif = pd.DataFrame({"vif": [VIF(X_all, i) for i in range(1, X_all.shape[1])]},
                    index=X_all.columns[1:])
```

```
# A VIF has no formula for its interval, so bootstrap_vif_ci
```

```
# (in the interval helpers) resamples the neighborhoods
```

```
vif[["lower", "upper"]] = bootstrap_vif_ci(Boston.drop(columns="medv")).T
print(vif)
```

	vif	lower	upper
crim	1.767486	1.555240	2.553470
zn	2.298459	2.026309	2.745879
indus	3.987181	3.390477	5.125450
chas	1.071168	1.049384	1.158832
nox	4.369093	4.046508	5.051079
rm	1.912532	1.670261	2.353259
age	3.088232	2.755750	3.634929

dis	3.954037	3.566747	4.638972
rad	7.445301	6.198520	9.890067
tax	9.002158	7.247533	12.301629
ptratio	1.797060	1.635727	2.078554
lstat	2.870777	2.453071	3.642755

10.0.5 R (tidymodels)

```
library(tidymodels)
```

```
# Each coefficient with its standard error and 95% confidence interval
tm_boston_lstat_age <- lm_spec |> fit(medv ~ lstat + age, data = boston)
tidy(tm_boston_lstat_age, conf.int = TRUE)
```

```
# A tibble: 3 x 7
```

	term <chr>	estimate <dbl>	std.error <dbl>	statistic <dbl>	p.value <dbl>	conf.low <dbl>	conf.high <dbl>
1	(Intercept)	33.2	0.731	45.5	2.94e-180	31.8	34.7
2	lstat	-1.03	0.0482	-21.4	8.42e-73	-1.13	-0.937
3	age	0.0345	0.0122	2.83	4.91e-3	0.0105	0.0586

```
tm_boston_all <- lm_spec |> fit(medv ~ ., data = boston)
tidy(tm_boston_all, conf.int = TRUE)
```

```
# A tibble: 13 x 7
```

	term <chr>	estimate <dbl>	std.error <dbl>	statistic <dbl>	p.value <dbl>	conf.low <dbl>	conf.high <dbl>
1	(Intercept)	41.6	4.94	8.43	3.79e-16	31.9	51.3
2	crim	-0.121	0.0330	-3.68	2.61e-4	-0.186	-0.0565
3	zn	0.0470	0.0139	3.38	7.72e-4	0.0197	0.0742
4	indus	0.0135	0.0621	0.217	8.29e-1	-0.109	0.136
5	chas	2.84	0.870	3.26	1.17e-3	1.13	4.55
6	nox	-18.8	3.85	-4.87	1.50e-6	-26.3	-11.2
7	rm	3.66	0.420	8.70	4.81e-17	2.83	4.48
8	age	0.00361	0.0133	0.271	7.87e-1	-0.0226	0.0298
9	dis	-1.49	0.202	-7.39	6.17e-13	-1.89	-1.09
10	rad	0.289	0.0669	4.33	1.84e-5	0.158	0.421
11	tax	-0.0127	0.00380	-3.34	9.12e-4	-0.0202	-0.00521
12	ptratio	-0.938	0.132	-7.09	4.63e-12	-1.20	-0.678
13	lstat	-0.552	0.0507	-10.9	6.39e-25	-0.652	-0.452

```
# A VIF has no formula for its interval, so boot_vif_ci
# (in the interval helpers) resamples the neighborhoods

# VIF_j = 1 / (1 - R^2) from regressing feature j on the other features
predictors <- setdiff(names(boston), "medv")
vif_interval <- boot_vif_ci(boston[predictors])
tibble(
  feature = predictors,
  vif = map_dbl(predictors, \(feature) {
    r_squared <- lm_spec |>
      fit(reformulate(setdiff(predictors, feature), feature), data = boston) |>
      glance() |>
      pull(r_squared)
    1 / (1 - r_squared)
  }),
  lower = vif_interval[1, ],
  upper = vif_interval[2, ]
)

# A tibble: 12 x 4
  feature    vif lower upper
  <chr>    <dbl> <dbl> <dbl>
1 crim     1.77  1.55  2.55
2 zn       2.30  2.02  2.74
3 indus    3.99  3.39  5.11
4 chas     1.07  1.05  1.16
5 nox      4.37  4.05  5.05
6 rm       1.91  1.66  2.34
7 age      3.09  2.75  3.64
8 dis      3.95  3.57  4.65
9 rad      7.45  6.18  9.91
10 tax     9.00  7.22 12.2
11 ptratio 1.80  1.63  2.08
12 lstat   2.87  2.45  3.63
```

10.0.6 R (base)

```
# The tables include each coefficient's standard error
# and 95% confidence interval
lm_boston_lstat_age <- lm(medv ~ lstat + age, data = boston)
parameters::parameters(lm_boston_lstat_age) |>
  parameters::print_md(digits = 3)
```

Parameter	Coefficient	SE	95% CI	t(503)	p
(Intercept)	33.223	0.731	(31.787, 34.659)	45.458	< .001
lstat	-1.032	0.048	(-1.127, -0.937)	-21.416	< .001
age	0.035	0.012	(0.011, 0.059)	2.826	0.005

```
lm_boston_all <- lm(medv ~ ., data = boston)
parameters::parameters(lm_boston_all) |>
  parameters::print_md(digits = 3)
```

Parameter	Coefficient	SE	95% CI	t(493)	p
(Intercept)	41.617	4.936	(31.919, 51.316)	8.431	< .001
crim	-0.121	0.033	(-0.186, -0.057)	-3.678	< .001
zn	0.047	0.014	(0.020, 0.074)	3.384	< .001
indus	0.013	0.062	(-0.109, 0.136)	0.217	0.829
chas	2.840	0.870	(1.131, 4.549)	3.264	0.001
nox	-18.758	3.851	(-26.325, -11.191)	-4.870	< .001
rm	3.658	0.420	(2.832, 4.484)	8.705	< .001
age	0.004	0.013	(-0.023, 0.030)	0.271	0.787
dis	-1.491	0.202	(-1.887, -1.095)	-7.394	< .001
rad	0.289	0.067	(0.158, 0.421)	4.325	< .001
tax	-0.013	0.004	(-0.020, -0.005)	-3.337	< .001
ptratio	-0.938	0.132	(-1.197, -0.678)	-7.091	< .001
lstat	-0.552	0.051	(-0.652, -0.452)	-10.897	< .001

```
# A VIF has no formula for its interval, so boot_vif_ci
# (in the interval helpers) resamples the neighborhoods

# VIF_j = 1 / (1 - R^2) from regressing feature j on the other features
predictors <- setdiff(names(boston), "medv")
vif_boston <- sapply(predictors, function(feature) {
  others <- reformulate(setdiff(predictors, feature), feature)
  1 / (1 - summary(lm(others, data = boston))$r.squared)
})
vif_boston
```

```
      crim      zn      indus      chas      nox      rm      age      dis
1.767486 2.298459 3.987181 1.071168 4.369093 1.912532 3.088232 3.954037
      rad      tax ptratio      lstat
7.445301 9.002158 1.797060 2.870777
```

```
vif_boston_ci <- boot_vif_ci(boston[predictors])
t(vif_boston_ci)
```

```
      2.5%      97.5%
crim    1.552895  2.550535
zn       2.020795  2.744689
indus    3.394594  5.108599
chas     1.048955  1.159317
nox      4.045030  5.049063
rm       1.662013  2.343394
age      2.752418  3.635210
dis      3.570608  4.649746
rad      6.181721  9.913714
tax      7.218113 12.244461
ptratio  1.634732  2.081573
lstat    2.446677  3.627501
```

```
# The same resampling gives an interval for the correlation of two features.
# boot_cor_ci() is defined in the solution of the multiple lab.
cor(boston$tax, boston$rad)
```

```
[1] 0.9102282
```

```
boot_cor_ci(boston$tax, boston$rad)
```

```
[1] 0.8804014 0.9349518
```

The `lstat` coefficient moves from -0.950 (95% confidence interval -1.026 to -0.874) to -1.032 (-1.127 to -0.937). It is now the change in `medv` per point of `lstat` among neighborhoods with the same `age`.

Exercise 10.10 (Lab: many features on the Boston data (part 2)). Use the setup of Exercise 10.9. Fit `medv` on all 12 other columns.

Solution

Solution 10.10. The fit has 12 features besides the intercept^a.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept>

Exercise 10.11 (Lab: many features on the Boston data (part 3)). Use the setup of Exercise 10.9. Which coefficients have p -values above 0.05?

Solution

Solution 10.11. `indus` and `age` have p -values above 0.05. With the other features in the model, the data give no clear sign that either one adds anything.

Exercise 10.12 (Lab: many features on the Boston data (part 4)). Use the setup of Exercise 10.9. Compute the variance inflation factor of Definition 8.4 for each feature of the full model.

Solution

Solution 10.12. The variance inflation factors of the full model run from 1.07 to 9.00.

Exercise 10.13 (Lab: many features on the Boston data (part 5)). Use the setup of Exercise 10.9. Which feature has the largest, and what is it?

Solution

Solution 10.13. The largest variance inflation factor is 9.00 (95% bootstrap interval 7.22 to 12.24), for `tax`, the property tax rate. Its coefficient has about 9 times the variance^a it would have if that feature were uncorrelated with the others. The next largest is 7.45 (6.18 to 9.91), for `rad`, the access to radial highways. The two are strongly correlated with each other (correlation^b 0.91, 95% bootstrap interval 0.88 to 0.93). Each bootstrap interval is the 2.5th to 97.5th percentile over 20,000 resamples of the 506 neighborhoods, with the statistic recomputed on each resample. Confidence intervals^c for the coefficients in the tables above are t intervals, the estimate^d plus or minus a t quantile times its standard error^e.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

^b<https://morrison-lab.github.io/pds/variance-covariance.html#def-correlation>

^c<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^d<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^e<https://morrison-lab.github.io/sds/estimation.html#def-SE>

Exercise 10.14 (Lab: an interaction on the Boston data). Fit `medv` on `lstat`, `age` and their product `lstat × age`, the interaction of Definition 6.7. Report the coefficients with their p -values.

Solution

Solution 10.14.

10.0.7 Python

```
import statsmodels.api as sm
```

```
X_interaction = sm.add_constant(Boston[["lstat", "age"]])
X_interaction["lstat:age"] = Boston["lstat"] * Boston["age"]
boston_interaction = sm.OLS(medv, X_interaction).fit()
print(boston_interaction.summary2().tables[1])
```

	Coef.	Std.Err.	t	P> t	[0.025	0.975]
const	36.088536	1.469835	24.552773	4.907116e-88	33.200749	38.976323
lstat	-1.392117	0.167456	-8.313352	8.780730e-16	-1.721117	-1.063117
age	-0.000721	0.019879	-0.036262	9.710878e-01	-0.039777	0.038336
lstat:age	0.004156	0.001852	2.244283	2.524911e-02	0.000518	0.007794

10.0.8 R (tidymodels)

```
library(tidymodels)

interaction_recipe <- recipe(medv ~ lstat + age, data = boston) |>
  step_interact(~ lstat:age)
tm_boston_interaction <- workflow() |>
  add_recipe(interaction_recipe) |>
  add_model(lm_spec) |>
  fit(boston)
# Each coefficient with its standard error and 95% confidence interval
tidy(tm_boston_interaction, conf.int = TRUE)

# A tibble: 4 x 7
  term          estimate std.error statistic  p.value  conf.low conf.high
<chr>         <dbl>     <dbl>     <dbl>   <dbl>   <dbl>    <dbl>
1 (Intercept)  36.1        1.47      24.6   4.91e-88 33.2     39.0
2 lstat       -1.39        0.167    -8.31  8.78e-16 -1.72    -1.06
3 age        -0.000721    0.0199   -0.0363 9.71e- 1 -0.0398  0.0383
4 lstat_x_age  0.00416     0.00185    2.24  2.52e- 2  0.000518 0.00779
```

10.0.9 R (base)

```
lm_boston_interaction <- lm(medv ~ lstat * age, data = boston)
# The table includes each coefficient's standard error
# and 95% confidence interval
parameters::parameters(lm_boston_interaction) |>
  parameters::print_md(digits = 3)
```

Parameter	Coefficient	SE	95% CI	t(502)	p
(Intercept)	36.089	1.470	(33.201, 38.976)	24.553	< .001
lstat	-1.392	0.167	(-1.721, -1.063)	-8.313	< .001
age	-7.209e-04	0.020	(-0.040, 0.038)	-0.036	0.971
lstat × age	0.004	0.002	(5.177e-04, 0.008)	2.244	0.025

The interaction coefficient is 0.0042 (95% confidence interval 0.0005 to 0.0078), with p -value 0.025.

Exercise 10.15 (Lab: an interaction on the Boston data (part 2)). Use the setup of Exercise 10.14. Is there evidence that the slope^a on `lstat` depends on `age`?

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Solution

Solution 10.15.

```
# The slope on lstat, beta_1 + beta_3 age,
# is a linear combination of the coefficients,
# so its 95% interval comes from lincomb_ci(),
# one of the interval helpers,
# with the weights (0, 1, 0, age)
slope_age_20 <- lincomb_ci(lm_boston_interaction, c(0, 1, 0, 20))
slope_age_90 <- lincomb_ci(lm_boston_interaction, c(0, 1, 0, 90))
```

That is modest evidence that the slope^a on `lstat` depends on `age`. The fitted slope on `lstat` is

$\hat{\beta}_1 + \hat{\beta}_3 \cdot \text{age}$. It is about -1.31 (95% confidence interval^b -1.57 to -1.05) in a neighborhood where 20% of homes were built before 1940, and about -1.02 (-1.11 to -0.92) where 90% were. Each interval is a t interval, the estimate^c plus or minus a t quantile times its standard error^d, and the standard error of the slope comes from the coefficients' estimated^e covariance matrix^f. The main effect of **age** has p -value 0.97, but the hierarchy principle of Remark 6.6 keeps it in the model alongside its interaction.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

^b<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^c<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^d<https://morrison-lab.github.io/sds/estimation.html#def-SE>

^e<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^f<https://morrison-lab.github.io/pds/variance-covariance.html#def-cov-vec-x>

Exercise 10.16 (Lab: a curve in `lstat`). Add `lstat`² to the model of Exercise 10.9, so the features are `lstat`, `lstat`² and `age`. Report the coefficients with their p -values.

Solution

Solution 10.16.

10.0.10 Python

```
import statsmodels.api as sm
from statsmodels.stats.anova import anova_lm
```

```
X_quadratic = sm.add_constant(Boston[["lstat", "age"]])
X_quadratic["lstat^2"] = Boston["lstat"] ** 2
boston_quadratic = sm.OLS(medv, X_quadratic).fit()
print(boston_quadratic.summary2().tables[1])
```

	Coef.	Std.Err.	t	P> t	[0.025	0.975]
const	41.288537	0.873212	47.283544	4.859728e-187	39.572938	43.004137
lstat	-2.688300	0.131122	-20.502230	2.569236e-68	-2.945917	-2.430684
age	0.070254	0.010857	6.471052	2.316450e-10	0.048924	0.091585
lstat^2	0.049489	0.003717	13.314608	7.468491e-35	0.042186	0.056791

```
print(anova_lm(boston_lstat_age, boston_quadratic))
```

	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	503.0	19168.128609	0.0	NaN	NaN	NaN
1	502.0	14165.613251	1.0	5002.515357	177.278785	7.468491e-35

```
import matplotlib.pyplot as plt
```

```
fig, ax = plt.subplots(figsize=(5, 3.5), layout="constrained")
ax.scatter(boston_quadratic.fittedvalues, boston_quadratic.resid, s=10, alpha=0.5);
ax.axhline(0, color="k", linestyle="--");
ax.set_xlabel("Fitted value");
ax.set_ylabel("Residual");
plt.show()
```

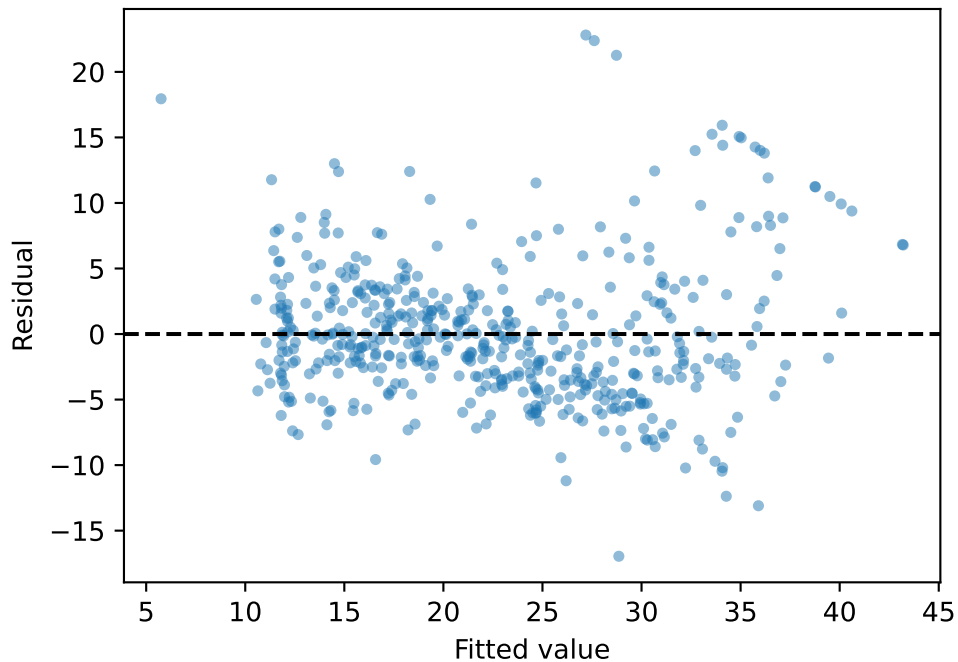


Figure 27: The residual plot of the fit of medv on lstat, lstat² and age.

10.0.11 R (tidymodels)

```
library(tidymodels)

quadratic_recipe <- recipe(medv ~ lstat + age, data = boston) |>
  step_mutate(lstat2 = lstat^2)
tm_boston_quadratic <- workflow() |>
  add_recipe(quadratic_recipe) |>
  add_model(lm_spec) |>
  fit(boston)
# Each coefficient with its standard error and 95% confidence interval
tidy(tm_boston_quadratic, conf.int = TRUE)

# A tibble: 4 x 7
  term      estimate std.error statistic    p.value conf.low conf.high
<chr>      <dbl>     <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1 (Intercept)  41.3       0.873      47.3 4.86e-187  39.6      43.0
2 lstat       -2.69      0.131     -20.5 2.57e- 68  -2.95     -2.43
3 age          0.0703   0.0109      6.47 2.32e- 10   0.0489    0.0916
4 lstat2        0.0495   0.00372    13.3 7.47e- 35   0.0422    0.0568

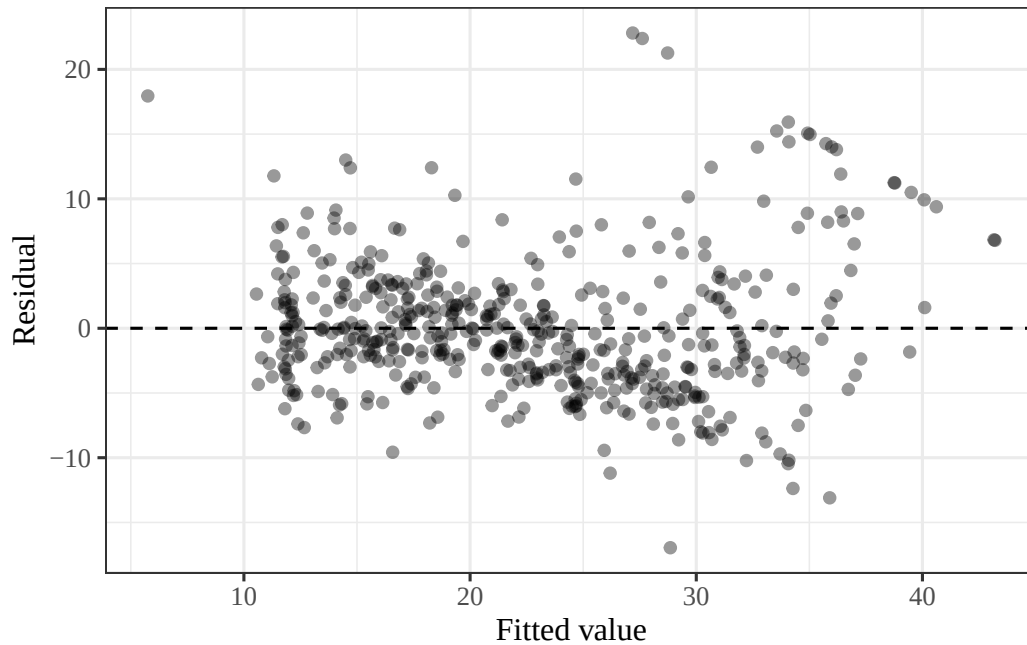
# anova() compares models only when their responses have the same name,
# so fit the smaller model through a workflow too.
tm_boston_lstat_age_wf <- workflow() |>
  add_recipe(recipe(medv ~ lstat + age, data = boston)) |>
  add_model(lm_spec) |>
  fit(boston)
anova(extract_fit_engine(tm_boston_lstat_age_wf),
      extract_fit_engine(tm_boston_quadratic)) |>
  tidy()

# A tibble: 2 x 7
```

term	df.residual	rss	df	sumsq	statistic	p.value
<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1 ..y ~ lstat + age	503	19168.	NA	NA	NA	NA
2 ..y ~ lstat + age + lstat2	502	14166.	1	5003.	177.	7.47e-35

```
library(tidymodels)
```

```
augment(tm_boston_quadratic, new_data = boston) |>
  ggplot(aes(.pred, .resid)) +
  geom_point(alpha = 0.4) +
  geom_hline(yintercept = 0, linetype = "dashed") +
  labs(x = "Fitted value", y = "Residual")
```



10.0.12 R (base)

```
lm_boston_quadratic <- lm(medv ~ lstat + I(lstat^2) + age, data = boston)
# The table includes each coefficient's standard error
# and 95% confidence interval
parameters::parameters(lm_boston_quadratic) |>
  parameters::print_md(digits = 3)
```

Parameter	Coefficient	SE	95% CI	t(502)	p
(Intercept)	41.289	0.873	(39.573, 43.004)	47.284	< .001
lstat	-2.688	0.131	(-2.946, -2.431)	-20.502	< .001
lstat^2	0.049	0.004	(0.042, 0.057)	13.315	< .001
age	0.070	0.011	(0.049, 0.092)	6.471	< .001

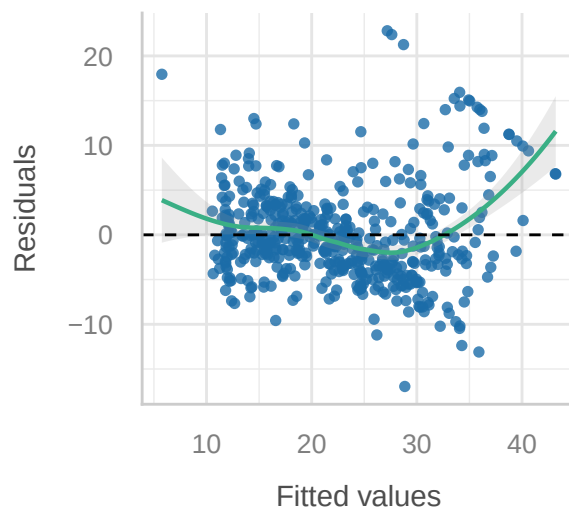
```
anova_boston <- anova(lm_boston_lstat_age, lm_boston_quadratic)
anova_boston |>
  parameters::parameters() |>
  parameters::print_md(footer = "")
```

Parameter	RSS	Sum_Squares	df	df (error)	Mean_Square	F	p
Model 1: medv ~ lstat + age	19168.13			503			
Model 2: medv ~ lstat + I(lstat^2) + age	14165.61	5002.52	1	502	5002.52	177.28	< .001

```
performance::check_model(lm_boston_quadratic, check = "linearity") |>
  plot()
```

Linearity

Reference line should be flat and horizontal



The coefficient of lstat^2 is 0.0495 (95% confidence interval 0.0422 to 0.0568, the estimate plus or minus a t quantile times its standard error), with p -value 7.5×10^{-35} . The positive sign means the fitted curve flattens as lstat grows, as the scatter plot of Solution 10.0.3 suggested.

Exercise 10.17 (Lab: a curve in `lstat` (part 2)). Use the setup of Exercise 10.16. Compare the two models with the partial F -test of Definition 7.6.

Solution

Solution 10.17. The partial F -statistic, with $q = 1$, is 177.3, with p -value 7.5×10^{-35} .

Exercise 10.18 (Lab: a curve in `lstat` (part 3)). Use the setup of Exercise 10.16. What does it say about the squared term?

Solution

Solution 10.18. The squared term clearly improves the fit. Since $q = 1$, the F -statistic is the square of the t -statistic of `lstat`² ($13.31^2 = 177.3$).

Exercise 10.19 (Lab: a curve in `lstat` (part 4)). Use the setup of Exercise 10.16. Draw the residual plot of the new model.

Solution

Solution 10.19. The residual plot of the new model is the one in Solution 10.0.12.

Exercise 10.20 (Lab: a curve in `lstat` (part 5)). Use the setup of Exercise 10.16. Compare it with the residual plot of Exercise 10.1.

Solution

Solution 10.20. The U shape of the first residual plot is gone: the residuals now scatter around 0 across the fitted values, though their spread grows at the highest fitted values.

Exercise 10.21 (Lab: a category with three levels). Download the Carseats data from [data/carseats.csv](https://morrisson-lab.github.io/lds/chapters/data/carseats.csv)^a and save it as `data/carseats.csv` next to your code. It is simulated data on sales of child car seats at 400 stores, from the book [32]. `Sales` is unit sales in thousands. `ShelveLoc` is the quality of the shelf space for the car seats at each store: `Bad`, `Medium` or `Good`. Fit `Sales` on all the other columns, plus the interactions `Income × Advertising` and `Price × Age`.

^a<https://morrisson-lab.github.io/lds/chapters/data/carseats.csv>

Solution

Solution 10.21.

10.0.13 Python

```
import pandas as pd
import statsmodels.api as sm

# ISLP's load_data("Carseats") reads this same file.
Carseats = pd.read_csv("data/carseats.csv")
sales = Carseats["Sales"]

X_carseats = pd.get_dummies(Carseats.drop(columns="Sales"), drop_first=True, dtype=float)
X_carseats["Income:Advertising"] = Carseats["Income"] * Carseats["Advertising"]
X_carseats["Price:Age"] = Carseats["Price"] * Carseats["Age"]
X_carseats = sm.add_constant(X_carseats)
print(X_carseats.columns.tolist())

['const', 'CompPrice', 'Income', 'Advertising', 'Population', 'Price', 'Age', 'Education', 'ShelveLoc',
carseats_fit = sm.OLS(sales, X_carseats).fit()
# The table includes each coefficient's standard error
# and 95% confidence interval
print(carseats_fit.summary2().tables[1])
```

Coef.	Std.Err.	...	[0.025	0.975]
-------	----------	-----	--------	--------

```

const          6.575565  1.008747  ...  4.592239  8.558892
CompPrice      0.092937  0.004118  ...  0.084840  0.101034
Income         0.010894  0.002604  ...  0.005773  0.016015
Advertising    0.070246  0.022609  ...  0.025794  0.114699
Population     0.000159  0.000368  ... -0.000564  0.000883
Price         -0.100806  0.007440  ... -0.115434 -0.086179
Age           -0.057947  0.015951  ... -0.089307 -0.026586
Education     -0.020852  0.019613  ... -0.059414  0.017709
ShelveLoc_Good  4.848676  0.152838  ...  4.548177  5.149175
ShelveLoc_Medium 1.953262  0.125768  ...  1.705986  2.200538
Urban_Yes      0.140160  0.112402  ... -0.080837  0.361156
US_Yes        -0.157557  0.148923  ... -0.450360  0.135245
Income:Advertising 0.000751  0.000278  ...  0.000204  0.001298
Price:Age      0.000107  0.000133  ... -0.000155  0.000369

```

[14 rows x 6 columns]

10.0.14 R (tidymodels)

```
library(tidymodels)
```

```

carseats <- read.csv("data/carseats.csv")
carseats_recipe <- recipe(Sales ~ ., data = carseats) |>
  step_dummy(all_nominal_predictors()) |>
  step_interact(~ Income:Advertising + Price:Age)
carseats_recipe |> prep() |> bake(new_data = NULL) |> names()

```

```

[1] "CompPrice"      "Income"          "Advertising"
[4] "Population"     "Price"           "Age"
[7] "Education"      "Sales"           "ShelveLoc_Good"
[10] "ShelveLoc_Medium" "Urban_Yes"       "US_Yes"
[13] "Income_x_Advertising" "Price_x_Age"

```

```

tm_carseats <- workflow() |>
  add_recipe(carseats_recipe) |>
  add_model(lm_spec) |>
  fit(carseats)
# Each coefficient with its standard error and 95% confidence interval
tidy(tm_carseats, conf.int = TRUE)

```

A tibble: 14 x 7

term <chr>	estimate <dbl>	std.error <dbl>	statistic <dbl>	p.value <dbl>	conf.low <dbl>	conf.high <dbl>
1 (Intercept)	6.58e+0	1.01	6.52	2.22e- 10	4.59e+0	8.56
2 CompPrice	9.29e-2	0.00412	22.6	1.64e- 72	8.48e-2	0.101
3 Income	1.09e-2	0.00260	4.18	3.57e- 5	5.77e-3	0.0160
4 Advertising	7.02e-2	0.0226	3.11	2.03e- 3	2.58e-2	0.115
5 Population	1.59e-4	0.000368	0.433	6.65e- 1	-5.64e-4	0.000883
6 Price	-1.01e-1	0.00744	-13.5	1.74e- 34	-1.15e-1	-0.0862
7 Age	-5.79e-2	0.0160	-3.63	3.18e- 4	-8.93e-2	-0.0266
8 Education	-2.09e-2	0.0196	-1.06	2.88e- 1	-5.94e-2	0.0177
9 ShelveLoc_Good	4.85e+0	0.153	31.7	1.38e-109	4.55e+0	5.15
10 ShelveLoc_Medium	1.95e+0	0.126	15.5	1.34e- 42	1.71e+0	2.20
11 Urban_Yes	1.40e-1	0.112	1.25	2.13e- 1	-8.08e-2	0.361
12 US_Yes	-1.58e-1	0.149	-1.06	2.91e- 1	-4.50e-1	0.135
13 Income_x_Advertisi~	7.51e-4	0.000278	2.70	7.29e- 3	2.04e-4	0.00130
14 Price_x_Age	1.07e-4	0.000133	0.801	4.24e- 1	-1.55e-4	0.000369

10.0.15 R (base)

```
carseats <- read.csv("data/carseats.csv", stringsAsFactors = TRUE)
contrasts(carseats$ShelveLoc)
```

```
      Good Medium
Bad      0      0
Good     1      0
Medium   0      1
```

```
lm_carseats <- lm(Sales ~ . + Income:Advertising + Price:Age, data = carseats)
# The table includes each coefficient's standard error
# and 95% confidence interval
parameters::parameters(lm_carseats) |>
  parameters::print_md(digits = 3)
```

Parameter	Coefficient	SE	95% CI	t(386)	p
(Intercept)	6.576	1.009	(4.592, 8.559)	6.519	< .001
CompPrice	0.093	0.004	(0.085, 0.101)	22.567	< .001
Income	0.011	0.003	(0.006, 0.016)	4.183	< .001
Advertising	0.070	0.023	(0.026, 0.115)	3.107	0.002
Population	1.592e-04	3.679e-04	(-5.640e-04, 8.825e-04)	0.433	0.665
Price	-0.101	0.007	(-0.115, -0.086)	-13.549	< .001
ShelveLoc (Good)	4.849	0.153	(4.548, 5.149)	31.724	< .001
ShelveLoc (Medium)	1.953	0.126	(1.706, 2.201)	15.531	< .001
Age	-0.058	0.016	(-0.089, -0.027)	-3.633	< .001
Education	-0.021	0.020	(-0.059, 0.018)	-1.063	0.288
Urban (Yes)	0.140	0.112	(-0.081, 0.361)	1.247	0.213
US (Yes)	-0.158	0.149	(-0.450, 0.135)	-1.058	0.291
Income × Advertising	7.510e-04	2.784e-04	(2.037e-04, 0.001)	2.698	0.007
Price × Age	1.068e-04	1.333e-04	(-1.554e-04, 3.689e-04)	0.801	0.424

The fit has 14 coefficients: an intercept, the 10 other columns with `ShelveLoc` expanded into two indicators, and the two products.

Exercise 10.22 (Lab: a category with three levels (part 2)). Use the setup of Exercise 10.21. Which indicator columns does the software build for `ShelveLoc`?

Solution

Solution 10.22. The software builds an indicator for `Good` and one for `Medium`.

Exercise 10.23 (Lab: a category with three levels (part 3)). Use the setup of Exercise 10.21. Which level has no column of its own?

Solution

Solution 10.23. `Bad` has no column of its own: it is the baseline, where both indicators are 0. Each tool drops the first level in alphabetical order, and `Bad` comes first.

Exercise 10.24 (Lab: a category with three levels (part 4)). Use the setup of Exercise 10.21. Interpret the coefficients of the two `ShelveLoc` indicators.

Solution

Solution 10.24. The **Good** coefficient is 4.85 (95% confidence interval^a 4.55 to 5.15). A store with a good shelf location is predicted to sell about 4,850 more car seats (95% confidence interval 4,550 to 5,150 seats) than a store with a bad one, with the other features held fixed. The **Medium** coefficient is 1.95 (1.71 to 2.20), so a medium location sits between the two, about 1,950 seats above a bad one (1,710 to 2,200 seats). Each interval is a t interval, the estimate^b plus or minus a t quantile times its standard error^c.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^c<https://morrison-lab.github.io/sds/estimation.html#def-SE>

11 Summary and further reading

11.1 Where this goes next

i Remark (Where linear regression goes next)

Remark 11.1 (Where linear regression goes next). Least squares fitted by the normal equations is the baseline case. Five directions build on it:

1. **Regularization for ill-conditioned or high-dimensional systems:** When $D^\top D$ is singular or near-singular, or when $p \geq n$ (or $p^* > n$), penalty terms stabilize parameter estimates. See Regularization.
2. **Iterative optimization:** When inverting $D^\top D$ is too costly or when minimizing non-quadratic objectives, gradient methods step downhill iteratively. See Gradient Descent.
3. **Representation learning:** When a fixed **feature map** $\tilde{D}(\tilde{x})$ is not known in advance, neural networks learn the representation from data. A dense layer applies an affine map $B^\top \tilde{x}$ followed by a non-linearity. See Neural Networks.
4. **Non-parametric regression and the curse of dimensionality:** Parametric linear regression imposes global structure, avoiding the exponential sample requirements that degrade local non-parametric estimators (e.g. K -nearest neighbors) in high dimensions. See Non-Parametric Models.
5. **Applied statistical modeling and diagnostics:** For applied regression diagnostics (F -tests, analysis of variance (ANOVA) comparisons, and residual plots), see the ISLP Labs^a linear regression materials.

^ahttps://morrison-lab.github.io/ISLP_labs/

11.2 Further reading

- James et al. [3], Chapter 3
- Hastie et al. [39], Chapter 3
- Ma and Ng [40], Chapter 1
- Zhou [41], Linear Regression¹⁰ — slides on *An Introduction to Statistical Learning*, Chapter 3, with code in both R and Python
- Starmer [42], on linear regression, and Starmer [43], on linear and multiple regression — illustrated walk-throughs with little algebra

i Watch

The Data Science Learning Community's book clubs discuss *An Introduction to Statistical Learning*, Chapter 3, in recorded meetings:

- the R book club, in part 1^a and part 2^b [44];
- the Python book club, in part 1^c and part 2^d [45].

¹⁰<https://ucla-biostat-212a.github.io/2026winter/slides/03-linreg/linreg.html>

^a<https://www.youtube.com/watch?v=Qj7Vtu9-TiY&list=PL3x6DOfs2NGjlpKMiWsDcKL2nngwXROq>
^b<https://www.youtube.com/watch?v=jqij1pe8CyU&list=PL3x6DOfs2NGjlpKMiWsDcKL2nngwXROq>
^c<https://www.youtube.com/watch?v=PbTw-SN7-xU&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>
^d<https://www.youtube.com/watch?v=hFfWajZmGio&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>

References

- [1] Wikipedia contributors, “Affine transformation — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Affine_transformation
- [2] Wikipedia contributors, “Linear regression — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Linear_regression
- [3] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning with applications in R*, 2nd ed. Springer, 2021. Available: <https://www.statlearning.com/>
- [4] Wikipedia contributors, “Hyperplane — Wikipedia, the free encyclopedia.” 2026. Available: <https://en.wikipedia.org/wiki/Hyperplane>
- [5] Wikipedia contributors, “Half-space (geometry) — Wikipedia, the free encyclopedia.” 2026. Available: [https://en.wikipedia.org/wiki/Half-space_\(geometry\)](https://en.wikipedia.org/wiki/Half-space_(geometry))
- [6] Wikipedia contributors, “Level set — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Level_set
- [7] Wikipedia contributors, “Regression analysis — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Regression_analysis
- [8] Wikipedia contributors, “Errors and residuals — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Errors_and_residuals
- [9] A. Ng *et al.*, “CS229: Machine learning, autumn 2018.” Stanford Online, lecture videos on YouTube, 2018. Available: <https://www.youtube.com/playlist?list=PLoROMvovd4rMiGQp3WXShtMGgzqpfVfbU>
- [10] Y. S. Abu-Mostafa, “Learning from data: CS 156 lectures.” Caltech, lecture videos on YouTube, 2012. Available: <https://www.youtube.com/playlist?list=PLD63A284B7615313A>
- [11] J. Starmer, “The main ideas of fitting a line to data (the main ideas of least squares and linear regression.)” StatQuest, video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=PaFPbb66DxQ>
- [12] J. Starmer, “Linear regression, clearly explained.” StatQuest, video on YouTube, 2022. Available: <https://www.youtube.com/watch?v=7ArmBVF2dCs>
- [13] GeeksforGeeks, “Linear regression in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/ml-linear-regression/>
- [14] N. Jaques, “CSE 446/546: Machine learning, lecture slides.” University of Washington, 2026. Accessed: Oct. 12, 2026. [Online]. Available: <https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/>
- [15] Wikipedia contributors, “Ordinary least squares — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Ordinary_least_squares
- [16] Wikipedia contributors, “Design matrix — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Design_matrix
- [17] T. Hastie, R. Tibshirani, and J. Taylor, “Statistical learning with python.” Stanford Online, lecture videos on YouTube, 2023. Available: <https://www.youtube.com/playlist?list=PLoROMvovd4rPP6braWoRt5UCXYZ71GZIQ>
- [18] J. Starmer, “Design matrices for linear models, clearly explained.” StatQuest, video on YouTube, 2019. Available: <https://www.youtube.com/watch?v=CqLGvwi-5Pc>
- [19] J. Starmer, “Multiple regression, clearly explained.” StatQuest, video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=zITIFTsivN8>
- [20] Wikipedia contributors, “Gradient — Wikipedia, the free encyclopedia.” 2026. Available: <https://en.wikipedia.org/wiki/Gradient>
- [21] K. Weinberger, “CS4780: Machine learning for intelligent systems, spring 2017.” Cornell University, lecture videos on YouTube, 2017. Available: <https://www.youtube.com/playlist?list=PLI8OIHZGYOQ7bkVbuRthEsaLr7bONzbXS>

- [22] B. Hutchinson, “DATA 471/571 (machine learning) and CSCI 481/581 (deep learning) video lectures.” Western Washington University. Accessed: Sep. 28, 2026. [Online]. Available: https://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/
- [23] GeeksforGeeks, “Normal equation in linear regression.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/ml-normal-equation-in-linear-regression/>
- [24] Wikipedia contributors, “Moore–penrose inverse — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse
- [25] GeeksforGeeks, “Moore - Penrose pseudoinverse | mathematics.” GeeksforGeeks, tutorial article, 2019. Available: <https://www.geeksforgeeks.org/machine-learning/moore-penrose-pseudoinverse-mathematics/>
- [26] Wikipedia contributors, “Condition number — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Condition_number
- [27] L. N. Trefethen and D. Bau III, *Numerical linear algebra*. SIAM, 1997. doi: 10.1137/1.9780898719574¹¹.
- [28] Wikipedia contributors, “Projection matrix — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Projection_matrix
- [29] Wikipedia contributors, “Leverage (statistics) — Wikipedia, the free encyclopedia.” 2026. Available: [https://en.wikipedia.org/wiki/Leverage_\(statistics\)](https://en.wikipedia.org/wiki/Leverage_(statistics))
- [30] GeeksforGeeks, “Regression metrics.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/regression-metrics/>
- [31] GeeksforGeeks, “Polynomial regression for non-linear data - ML.” GeeksforGeeks, tutorial article, 2020. Available: <https://www.geeksforgeeks.org/machine-learning/polynomial-regression-for-non-linear-data-ml/>
- [32] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, *An introduction to statistical learning: With applications in Python*. Springer, 2023. doi: 10.1007/978-3-031-38747-0¹².
- [33] Wikipedia contributors, “Interaction (statistics) — Wikipedia, the free encyclopedia.” 2026. Available: [https://en.wikipedia.org/wiki/Interaction_\(statistics\)](https://en.wikipedia.org/wiki/Interaction_(statistics))
- [34] Wikipedia contributors, “Prediction interval — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Prediction_interval
- [35] Wikipedia contributors, “Variance inflation factor — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Variance_inflation_factor
- [36] Wikipedia contributors, “Stepwise regression — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Stepwise_regression
- [37] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, “ISLP labs: Lab notebooks for an introduction to statistical learning with applications in python.” GitHub repository, BSD 2-Clause license, 2023. Available: https://github.com/intro-stat-learning/ISLP_labs
- [38] D. Harrison and D. L. Rubinfeld, “Hedonic housing prices and the demand for clean air,” *Journal of Environmental Economics and Management*, vol. 5, no. 1, pp. 81–102, 1978, doi: 10.1016/0095-0696(78)90006-2¹³.
- [39] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, 2nd ed. Springer, 2009. Available: <https://hastie.su.domains/ElemStatLearn/>
- [40] T. Ma and A. Ng, “CS229 lecture notes.” Stanford University; revised 23 August 2026, 2026. Available: https://cs229.stanford.edu/main_notes.pdf
- [41] J. Zhou, “Biostat 212A: Statistical machine learning a, winter 2026.” UCLA, course webpage with lecture slides, 2026. Available: <https://ucla-biostat-212a.github.io/2026winter/>
- [42] J. Starmer, *The StatQuest illustrated guide to machine learning*. StatQuest Publications, 2022. Available: <https://statquest.org/statquest-store/>
- [43] J. Starmer, *The StatQuest illustrated guide to statistics: With hands-on examples in Python and R*. 2026. Available: <https://statquest.org/statquest-store/>
- [44] Data Science Learning Community, “Introduction to statistical learning using R (islr06).” Book club meeting recordings on YouTube, 2023. Available: <https://www.youtube.com/playlist?list=PL3x6DOfs2NGjIhpKMjWsDcKL2nngwXROq>

¹¹<https://doi.org/10.1137/1.9780898719574>

¹²<https://doi.org/10.1007/978-3-031-38747-0>

¹³[https://doi.org/10.1016/0095-0696\(78\)90006-2](https://doi.org/10.1016/0095-0696(78)90006-2)

- [45] Data Science Learning Community, “Introduction to statistical learning using Python (islp03).” Book club meeting recordings on YouTube, 2025. Available: <https://www.youtube.com/playlist?list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>