

A First Model and Generalization

A worked example, generalization, and bias and variance

Last modified: 2026-10-28

```
# The Python chunks on this page import numpy and pandas; reticulate provisions them.
library(reticulate)
py_require(c("numpy", "pandas", "matplotlib", "SyncRNG"))
```

i Source

Adapted from Brian Hutchinson's Fall 2025 lecture notes, which supplied the crime-and-income example, the five data points, the generalization argument and the bias-variance decomposition. The figure the notes marked *draw figure* is Figure 1; the two the notes marked *plot* are Figure 7 and Figure 8. Every fitted coefficient printed here is computed on the page from those same five points, and `tools/make_big_ideas_figures.py` fits the curves in the figures from them too. Logan Sizemore's Spring 2025 lecture on generalization is the basis of the cross-validation section. The contrast between stochastic and deterministic noise follows the treatment of overfitting in Abu-Mostafa et al. [1, Ch. 4]; the example and its numbers are new.

The bias-variance derivation is rewritten to assume only that the noise has mean zero, finite variance, and is independent of the training set, where the notes assumed it Gaussian; see Theorem 4.4. The wider reading is Hastie et al. [2], Chapter 2, and James et al. [3], Chapter 2.

Course Overview is the companion chapter and comes first: it asks what machine learning *is* and defines the vocabulary, where this one asks how one concrete instance of it works. Where the two meet, this page recaps with a short exercise and links back to the definition.

1 The anatomy of a learning system

Exercise 1.1 (The parts of a learning system). Course Overview named the parts every learning system has. Without looking back, name them, and say which step happens once and which happens every time the system is used.

Solution

Solution 1.1. Figure 1 draws them. The training data go into a learning algorithm (the figure's *training algorithm*), and running it, training, produces the model h . The model then maps each new input x to a prediction $h(x)$.

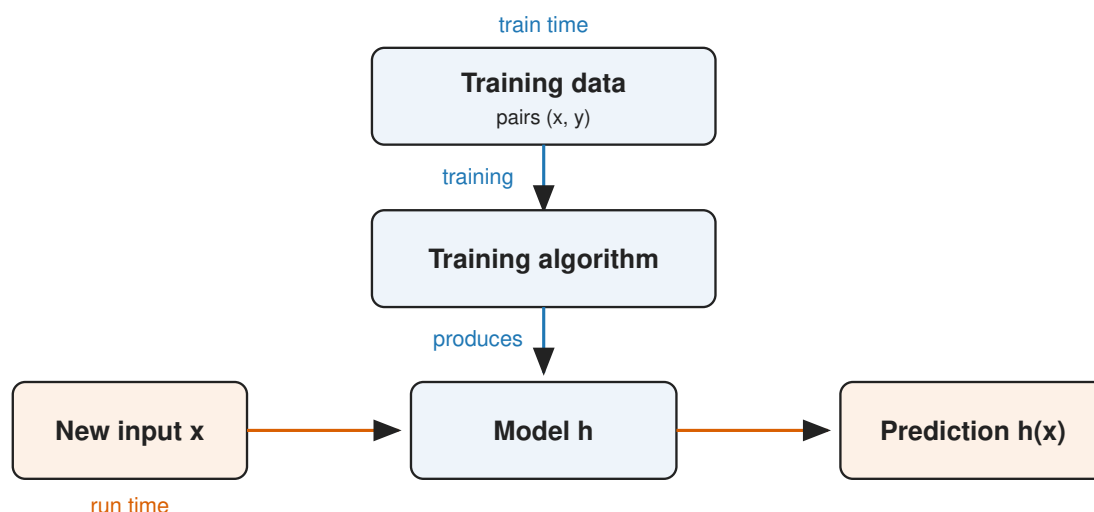


Figure 1: The two kinds of arrows in a learning system. The blue ones happen once, at **train time**; the orange ones happen every time the system is used, at **run time**.

Training, the blue arrows, happens once, at train time. Prediction, the orange arrows, happens every time the system is used, at run time. The difference matters for cost: a cost paid at train time is paid once, and a cost paid at run time is paid on every prediction forever.

2 A first model, end to end

We work with a data set¹, which the statistics notes define as n pairs of an input vector and its outcome. They also define its input matrix² $\mathbf{X}$, whose rows are the input vectors, and its outcome vector³ y , whose entries are the outcomes. As that page notes, the “set” in training set, validation set and test set is short for a subset of the data set⁴.

i Remark (The data set in matrix form)

Remark 2.1 (The data set in matrix form). Row i of $\mathbf{X}$ and entry i of y together are the i -th pair of the data set, so $\mathbf{X}$ and y hold the same information as $\mathcal{D}$ in a tabular form.

Example 2.1 (The Auto data as an input matrix). Take the first five cars of the Auto data [3], with horsepower and weight as the $p = 2$ inputs^a and miles per gallon (mpg) as the outcome. Then $\mathbf{X}$ has five rows and two columns, and y has five entries.

Build the input matrix and outcome vector from the first five cars with a recorded horsepower.

```
import numpy as np
import pandas as pd

auto_five = pd.read_csv("data/auto.csv").dropna(subset=["horsepower"]).iloc[:5]
X_data = auto_five[["horsepower", "weight"]].to_numpy(dtype=float)
y_data = auto_five["mpg"].to_numpy(dtype=float)
print(X_data.shape, y_data.shape)
```

¹<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

²<https://morrison-lab.github.io/sds/model-validation.html#def-input-matrix>

³<https://morrison-lab.github.io/sds/model-validation.html#def-outcome-vector>

⁴<https://morrison-lab.github.io/sds/model-validation.html#rem-data-set-subsets>

(5, 2) (5,)

Figure 2 draws the two arrays side by side. Row i of the left array and entry i of the right array are the same car.

```
import matplotlib.pyplot as plt

def show_array(ax, values, col_labels, title, fmt):
    """Draw a small array as a grid of cells, printing each value inside its cell."""
    # Shade each column on its own scale, because horsepower and weight differ by orders of magnitude
    scaled = (values - values.min(axis=0)) / np.ptp(values, axis=0)
    ax.imshow(scaled, cmap="Blues", aspect="auto", vmin=0, vmax=1.6);
    for i in range(values.shape[0]):
        for j in range(values.shape[1]):
            ax.text(j, i, fmt.format(values[i, j]), ha="center", va="center", fontsize=9);
    ax.set_xticks(range(len(col_labels)), col_labels);
    ax.set_yticks(range(values.shape[0]), [f"{i + 1}" for i in range(values.shape[0])]);
    ax.set_title(title, fontsize=10);

fig, axes = plt.subplots(
    1, 2, figsize=(6, 2.8), layout="constrained",
    gridspec_kw={"width_ratios": [2, 1]},
)
show_array(axes[0], X_data, ["horsepower", "weight"], "input matrix X", "{:.0f}")
show_array(axes[1], y_data.reshape(-1, 1), ["mpg"], "outcome vector y", "{:.1f}")
plt.show()
```

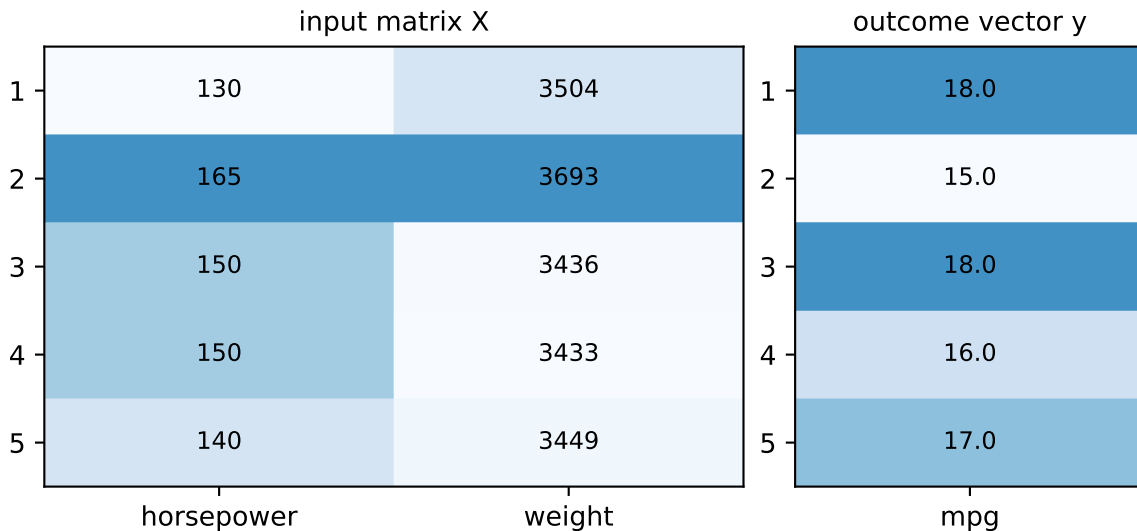


Figure 2: The input matrix of the first five Auto cars (left, one row per car, columns horsepower and weight) next to the outcome vector of their miles per gallon (right). Row i on the left and entry i on the right are the same car.

The first car, for example, has input vector $x_1 = (130, 3504)$ and outcome $y_1 = 18$, and these fill the first row of the matrix and the first entry of the vector.

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

Definition 2.1 (Parameter (weight, coefficient)). A **parameter** of a prediction function family^a is a number that identifies one member of the family.

Other names: parameters are also called **weights**, and the parameters of a linear model are also called **coefficients**.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-prediction-function-family>

Example 2.2 (The parameters of a line). The straight line $h(x) = mx + b$ has two **parameters**, the slope^a m and the intercept^b b . Each pair of values for m and b picks out exactly one line.

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

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

i Remark (Training chooses the parameters)

Remark 2.2 (Training chooses the parameters). Training chooses the values of the **parameters**.

Definition 2.2 (Training set). Given a data set^a $\mathcal{D}$ of n pairs^b, a **training set** is the subset of $\mathcal{D}$ whose pairs are indexed by a set $I_{\text{train}} \subseteq \{1, \dots, n\}$ to which we apply the fitting procedure, that is, the pairs from which the model's **parameters** are estimated^c [4]:

$$\mathcal{D}_{\text{train}} \stackrel{\text{def}}{=} \{(x_i, y_i) : i \in I_{\text{train}}\} \subseteq \mathcal{D}. \quad (1)$$

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

^b<https://morrison-lab.github.io/mds/sets.html#def-ordered-pair>

^c<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Example 2.3 (Estimating a crime rate from income). Suppose a city wants to estimate the crime rate in a neighbourhood from that neighbourhood's per-capita income. An estimate would help it send police, outreach and services before the need appears in the data rather than after. The city's data form a labelled data set^a of n pairs, written

$$\mathcal{D} = \{(x_1, y_1), \dots, (x_n, y_n)\}$$

and here $n = 5$. Until we hold some pairs out, all of them form the **training set**, so $\mathcal{D}_{\text{train}} = \mathcal{D}$:

```
# Read the five neighbourhoods once, from the local copy in data/crime.csv;
# every later example and figure on this page reuses this table.
crime <- read.csv("data/crime.csv")
knitr::kable(
  crime,
  col.names = c("per-capita income $x$", "crime rate $y$"),
  align = "rr", digits = 2
)
```

Table 1: The five sampled neighbourhoods. Both columns are rescaled to run between 0 and 1.

per-capita income x	crime rate y
0.49	0.09
0.18	0.45
0.31	0.23
0.40	0.19
0.24	0.48

```
# The same table for the Python chunks.
import pandas as pd

crime = pd.read_csv("data/crime.csv")
print(crime.shape)
```

(5, 2)

The numbers come from Brian Hutchinson's Fall 2025 notes. They are illustrative rather than measured, and they are the one place in this chapter where invented data is used, because the point being made is about the fitting procedure rather than about crime.

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

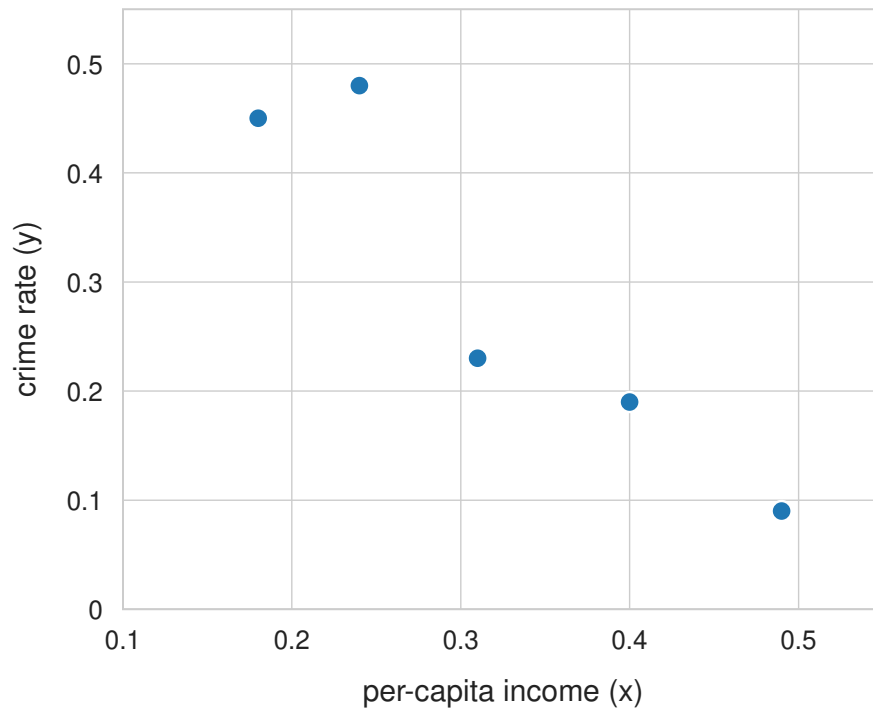


Figure 3: The five neighbourhoods of Table 1.

Exercise 2.1 (What has the city already decided?). Before any model is fitted, the city's setup has already made several choices that are not facts about the world. Name three, and say what each one rules out or risks.

Solution

Solution 2.1.

- The goal is **prediction**, not description. We are not asking whether income *causes* crime, and nothing in this chapter justifies that interpretation. James et al. [3], Section 2.1.1, makes the same distinction between prediction and inference.
- Income is the only input. It is almost certainly not the best one, and the model can only be as good as what it is shown.
- The city sampled five neighbourhoods. Five is a small number, and most of what goes wrong in

this chapter goes wrong because of it.

Exercise 2.2 (What kind of task is this?). The outputs in Table 1 are crime rates. Say whether this is a regression task or a classification task, and say what about the table settles it.

Solution

Solution 2.2. Regression. The column of outputs holds numbers on a continuous scale, and the quantity we want back from the model is a number of the same kind. Had the city instead sorted neighbourhoods into “high” and “low” crime, the same five neighbourhoods would have posed a classification task. The split is about the **output**, not about the inputs and not about the method. The inputs here are numbers too, and that is irrelevant; classification problems routinely have numeric inputs.

2.1 Choosing a model family

Exercise 2.3 (Propose a model). Look at Figure 3. Propose a family of functions h to predict y from x , and say how many numbers have to be chosen to specify one member of your family.

Solution

Solution 2.3. The five points fall roughly along a line, so a reasonable family is the straight lines

$$h(x) = mx + b$$

and two numbers identify exactly one: the slope m and the intercept b .

Nothing yet says which m and b to pick; The next section, “Measuring how well a model fits”, starts on that.

Proposing a family by eye is honest here and does not work for larger problems. With one input we can look at the scatter plot; with fifty inputs there is no plot to look at, and the choice of family becomes a modelling decision argued from what the quantity is rather than from what the data looks like.

Definition 2.3 (Mean-function model family). A **mean-function model family** is a set of mean functions that training chooses from. It is defined in the notes on model validation^a.

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

Example 2.4 (The family of straight lines). The straight lines

$$h(x) = mx + b \tag{2}$$

form a mean-function model family^a, one line for each choice of the slope m and the intercept b . Training on the five points picks one line from this family. No member of the family can bend, so a curved pattern in the data is out of its reach.

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

Definition 2.4 (Parametric model). A **parametric model** is a model family described by a fixed, finite list of parameters (Definition 2.1), whose length does not depend on the number of training points n [5].

Example 2.5 (A line keeps two numbers however many cars it sees). In the `Auto` data [3], fit the straight line of `mpg` on `horsepower` (Equation 2) to the first 20, the first 100 and then all 392 cars. Count the parameters of the fitted model each time:

2.1.1 Python

```
import numpy as np
import pandas as pd

auto_pm = pd.read_csv("data/auto.csv").dropna(subset=["horsepower"])
hp_pm = auto_pm["horsepower"].to_numpy(float)
mpg_pm = auto_pm["mpg"].to_numpy(float)

# np.polyfit returns the slope and the intercept: the line's whole description.
for n_pm in (20, 100, 392):
    coef_pm = np.polyfit(hp_pm[:n_pm], mpg_pm[:n_pm], 1)
    print(n_pm, len(coef_pm), coef_pm.round(3))
```

20 2 [-0.069 27.936]
 100 2 [-0.101 31.46]
 392 2 [-0.158 39.936]

2.1.2 R

```
auto_pm <- na.omit(read.csv("data/auto.csv")[, c("mpg", "horsepower")])

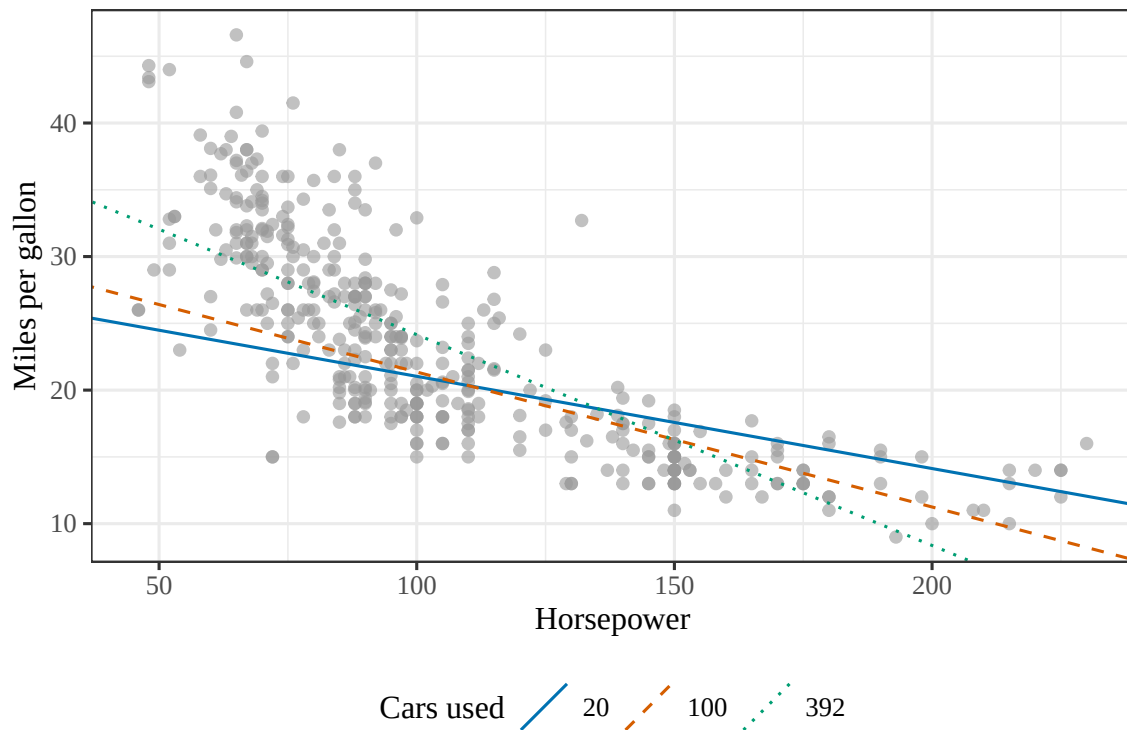
# Each column is one fit: its intercept, then its slope.
fits_pm <- sapply(c(20, 100, 392), function(n) {
  coef(lm(mpg ~ horsepower, data = auto_pm[1:n, ]))
})
colnames(fits_pm) <- c(20, 100, 392)
round(fits_pm, 3)
```

	20	100	392
(Intercept)	27.936	31.460	39.936
horsepower	-0.069	-0.101	-0.158

With 20 cars the slope is -0.069 miles per gallon per horsepower, and with all 392 it is -0.158. The values of the parameters change with the data, but the model keeps 2 of them whether it was fit to 20 cars or to 392. That fixed length makes the family of lines parametric (Definition 2.4).

2.1.3 R

```
library(ggplot2)
pmf_n <- c(20, 100, 392)
pmf_lines <- data.frame(
  cars = factor(pmf_n),
  intercept = fits_pm[1, ],
  slope = fits_pm[2, ]
)
ggplot(auto_pm, aes(horsepower, mpg)) +
  geom_point(colour = "grey60", alpha = 0.6) +
  geom_abline(
    data = pmf_lines,
    aes(intercept = intercept, slope = slope, colour = cars, linetype = cars)
  ) +
  scale_colour_manual(values = c("20" = "#0072B2", "100" = "#D55E00", "392" = "#009E73")) +
  scale_linetype_manual(values = c("20" = "solid", "100" = "dashed", "392" = "dotted")) +
  labs(x = "Horsepower", y = "Miles per gallon", colour = "Cars used", linetype = "Cars used")
```



(a)

2.1.4 Python

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

fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.plot(hp_pm, mpg_pm, color="grey", marker="o", linestyle="", alpha=0.6, markersize=4);
grid_pmf = np.array([hp_pm.min(), hp_pm.max()])
for n_pmf, colour, style in [(20, "#0072B2", "-"), (100, "#D55E00", "--"), (392, "#009E73", ":")]:
    slope_pmf, intercept_pmf = np.polyfit(hp_pm[:n_pmf], mpg_pm[:n_pmf], 1)
    ax.plot(grid_pmf, intercept_pmf + slope_pmf * grid_pmf, color=colour, linestyle=style,
            label=f"{n_pmf} cars");
ax.set_xlabel("Horsepower");
ax.set_ylabel("Miles per gallon");
fig.legend(loc="outside lower center", ncol=3);
plt.show()
```

Example 2.6 (Straight lines are a parametric model). The straight lines of Equation 2 form a [parametric model](#) with two parameters.

i Watch

Hastie and Tibshirani introduce the regression function and how to estimate it, from 6:03^a [6], video 2.1.

^a<https://www.youtube.com/watch?v=ox0cKk7h4o0&t=363s>

2.2 Measuring how well a model fits

Exercise 2.4 (What should a loss function do?). A [loss function](#) $L(y, \hat{y})$ takes the true value y and a prediction $\hat{y}$ and returns one number saying how bad that prediction was. Before choosing a formula, decide what the formula has to satisfy.

Name two properties any such function needs, and for each one, name what goes wrong without it.

Solution

Solution 2.4. **It is never negative.** Without that, errors in opposite directions cancel in the average, and a model can get a lower loss for being wrong in the other direction. A signed error $\hat{y} - y$ does not satisfy this: a model that overshoots one point by 10 and undershoots the next by 10 averages to zero, which is the score a perfect model gets.

It is zero exactly when the prediction is right. Without that, “smaller is better” no longer has its usual meaning, and a perfect prediction can cost more than an imperfect one.

Two functions satisfy both:

$$L_2(y, \hat{y}) \stackrel{\text{def}}{=} (y - \hat{y})^2 \quad \text{and} \quad L_1(y, \hat{y}) \stackrel{\text{def}}{=} |y - \hat{y}| \quad (3)$$

We use squared error, L_2 , throughout this chapter. It has a third property the other lacks, and that property is about *solving* the problem rather than about *stating* it: squared error is differentiable everywhere, while absolute error has no derivative at the one point where the prediction is exact. The methods that search for a minimum step by step, starting with Gradient Descent, need that derivative. The choice has consequences. Squaring an error of 10 costs a hundred times an error of 1, so L_2 gives disproportionate weight to the worst-fitting points and a single outlier can pull the whole fit strongly toward it. L_1 does not, which is exactly why robust regression uses it. Hastie et al. [2], Section 2.4, compares the two.

Definition 2.5 (Loss function). A **loss function** $L(y, \hat{y})$ takes the true value y and a prediction $\hat{y}$ and returns a number, never negative, saying how bad that prediction was [7].

i Remark (Which properties a loss needs, and what a loss scores)

Remark 2.3 (Which properties a loss needs, and what a loss scores). The two losses of Equation 3 also have the second property of Solution 2.4: they are zero exactly when $\hat{y} = y$. The definition of a [loss function](#) asks only for the first, because some losses later in these notes do not have the second: the hinge loss is zero for every score beyond the margin, and the log loss is never exactly zero.

A loss scores one prediction. For the i -th example, that prediction is the **fitted value** $\hat{y}_i = h(x_i)$, the value the model h returns at x_i , so the example’s loss is $L(y_i, \hat{y}_i)$. Other sources call an average of losses over many examples the *cost* or the *objective*.

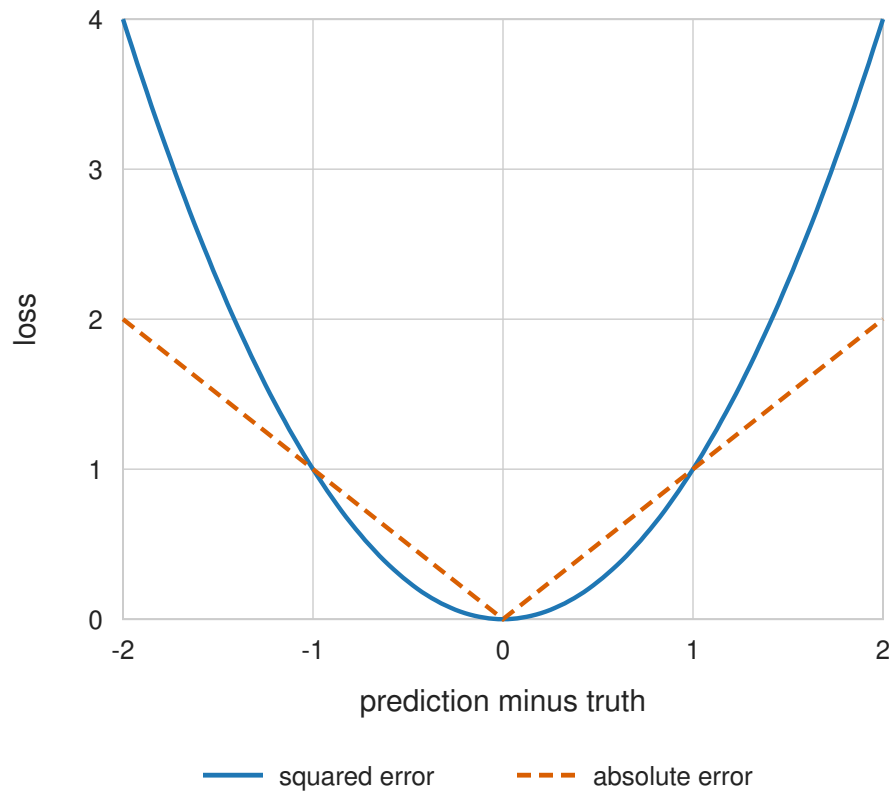


Figure 5: Squared error and absolute error as functions of the error, prediction minus truth. Both are zero only at zero error and never negative. Beyond an error of 1, squared error grows faster; within 1, it is smaller.

Exercise 2.5 (Who chooses the loss?). A frost alarm predicts, each evening, whether tonight will freeze. A false alarm (predict frost, no frost comes) costs one unit, the price of lighting the heaters. A missed frost (predict no frost, frost comes) costs a different amount that depends on what is growing. Over 100 evenings, 10 of which had frost, two alarms scored as follows:

Alarm	Frosts caught	False alarms
Cautious	10 of 10	20 of 90
Relaxed	6 of 10	5 of 90

A commercial orchard loses 50 units to a missed frost. A balcony gardener loses 2. Compute the total cost of each alarm for each owner.

Solution

Solution 2.5. Each alarm's cost is its missed frosts times the miss cost plus its false alarms times one. The Cautious alarm misses none; the Relaxed alarm misses four.

2.2.1 Python

```
import pandas as pd

# Counts from the table: [missed frosts, false alarms] for each alarm.
mistakes_wc = pd.DataFrame({"Cautious": [10 - 10, 20], "Relaxed": [10 - 6, 5]},
                           index=["missed", "false alarm"])

# The only thing that differs between owners is the cost of a missed frost;
# a false alarm costs 1 for both.
miss_cost_wc = {"Orchard": 50, "Balcony": 2}

costs_wc = pd.DataFrame({
    owner: mistakes_wc.loc["missed"] * miss + mistakes_wc.loc["false alarm"] * 1
    for owner, miss in miss_cost_wc.items()
})
print(costs_wc)
```

	Orchard	Balcony
Cautious	20	20
Relaxed	205	13

2.2.2 R

```
# Counts from the table: missed frosts and false alarms for each alarm.
missed_wc <- c(Cautious = 10 - 10, Relaxed = 10 - 6)
false_alarms_wc <- c(Cautious = 20, Relaxed = 5)
# The only thing that differs between owners is the cost of a missed frost;
# a false alarm costs 1 for both.
miss_cost_wc <- c(Orchard = 50, Balcony = 2)

costs_wc <- sapply(miss_cost_wc, function(miss) missed_wc * miss + false_alarms_wc * 1)
costs_wc
```

	Orchard	Balcony
Cautious	20	20
Relaxed	205	13

The orchard prefers the Cautious alarm, at 20 units to the Relaxed alarm's 205. The balcony gardener prefers the Relaxed alarm, at 13 units to the Cautious alarm's 20.

Exercise 2.6 (Who chooses the loss? (part 2)). Use the setup of Exercise 2.5.

The two owners rank^a the alarms differently. What does that say about who should pick the loss function for a model that makes this prediction?

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

Solution

Solution 2.6. The two alarms and their predictions are the same for both owners, so the ranking cannot be a fact about the alarms. It comes from the costs, which belong to whoever lives with the mistakes. That is why a **loss function** is something the user sets: the formula should say how much each kind of mistake matters in the application, and only the user knows that. Accuracy, which counts a missed frost and a false alarm alike, would pick the Relaxed alarm for both owners (91 correct nights against 80), which is the right answer for the balcony and a costly one for the orchard.

Definition 2.6 (Plausible loss). A loss function is **plausible** for a model of the data when it follows from an assumption about how the data arise: minimizing it gives the maximum likelihood^a fit under that assumption.

^a<https://morrison-lab.github.io/sds/intro-MLEs.html#def-mle>

Example 2.7 (Squared error is plausible under Gaussian noise). Suppose the noise terms ε of Equation 20 are independent across observations, each with a Gaussian distribution with mean zero and variance σ^2 . The negative log-likelihood of the observations is then the sum of their squared errors divided by $2\sigma^2$, plus a constant that does not depend on the model. Minimizing the squared error therefore gives the maximum likelihood^a fit, so squared error is plausible under this assumption. For a binary outcome modelled as conditionally Bernoulli, the same maximum-likelihood argument yields the log loss.

^a<https://morrison-lab.github.io/sds/intro-MLEs.html#def-mle>

Definition 2.7 (Friendly loss). A loss function is **friendly** for a model family when the model that minimizes it can be found reliably and cheaply.

Example 2.8 (Squared error and convex losses are friendly). For a linear model, squared error has a closed-form solution^a, so no search is needed. More generally, every local minimum of a convex loss is a global one, so gradient descent cannot get stuck at a worse model. Both cases are friendly in the sense of Definition 2.7.

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

i Remark (When nobody specifies the loss)

Remark 2.4 (When nobody specifies the loss). Often the user cannot say what each mistake costs, or the costs are unknown when the model is fitted. Two other reasons then decide the loss: a **plausible loss** matches an assumption about the data, and a **friendly loss** is easy to optimize. Squared error is both, for the reasons in Example 2.7 and Example 2.8. A loss chosen for these reasons is a convenience, not a statement of what the errors cost, so it is worth checking the finished model against the costs that matter (Exercise 2.5).

This follows the discussion of error measures in Abu-Mostafa et al. [1, Ch. 1].

Exercise 2.7 (Score a model on data it has not seen). The loss function scores a model on one pair at a time. A model is useful only if it also does well on points it has not seen. Suppose a new input–output pair (X, Y) is drawn at random from the same population that produced the training data. For a fixed model h , write the average loss that h would incur over all such new pairs^a, as an expectation^b.

^a<https://morrison-lab.github.io/mds/sets.html#def-ordered-pair>

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

Solution

Solution 2.7. Average the loss over the distribution^a $P(x, y)$ of inputs^b and outputs in the population^c:

$$\mathbb{E}_{(X,Y)}[L(Y, h(X))].$$

^a<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

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

^c<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-population>

Exercise 2.8 (Score a model on data it has not seen (part 2)). Use the setup of Exercise 2.7. Which parts of your expression^a do you know, and which do you not know?

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

Solution

Solution 2.8. The loss function L is known, because we chose it, and the model h is known, because we fitted it. The distribution^a $P(x, y)$ is unknown, so the expectation^b cannot be computed. If we knew $P(x, y)$, we would not need data to build a model.

^a<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

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

Definition 2.8 (Risk (expected loss, population risk, true risk, generalization error, expected prediction error)). The **risk** of a model h is its expected^a loss (Definition 2.5) on a new input–output pair (X, Y) drawn from the joint distribution^b $P(x, y)$ of the population^c:

$$R(h) \stackrel{\text{def}}{=} \mathbb{E}_{(X,Y)}[L(Y, h(X))]. \quad (4)$$

Other names: the risk is also called the **expected loss**, the **population risk**, the **true risk**, or the **generalization error**. *The Elements of Statistical Learning* calls it the **expected prediction error** [2, Sec. 2.4].

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

^b<https://morrison-lab.github.io/pds/random-variables.html#def-joint-distribution>

^c<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-population>

Example 2.9 (Fitting a line to the crime data). Fitting a line to the crime data, for example, is an attempt to pick the line h whose squared error on neighbourhoods we have not yet seen is small on average.

i Remark (The risk cannot be computed)

Remark 2.5 (The risk cannot be computed). The **risk** cannot be computed, because it needs the distribution the data came from, which we do not know. It can only be estimated from the data in hand. The risk measures how well a model predicts neighbourhoods outside the sample.

Every method in these notes tries to make $R(h)$ small.

The probability notes define risk^a for a prediction $\hat{y}$, which can be a constant or a function of X . Here $\hat{y} = h(X)$.

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

Definition 2.9 (Empirical risk). The **empirical risk** of a model h on a data set^a $\mathcal{D}$ is the average **loss** of h 's predictions over the n pairs $(x_{(i)}, y_{(i)})$ of $\mathcal{D}$:

$$\hat{R}(h; \mathcal{D}) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^n L(y_{(i)}, h(x_{(i)})). \quad (5)$$

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

i Remark (The empirical risk names its data set)

Remark 2.6 (The empirical risk names its data set). In the **empirical risk**, $\mathcal{D}$ can be any data set^a: the **training set** the fitted model was fitted to, or data that played no part in the fit. The same model has a different empirical risk on each, so the notation always names the set.

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

Definition 2.10 (Training error). The **training error** [2, Sec. 7.2] of a model h is its **empirical risk** on the **training set**, $\hat{R}(h; \mathcal{D}_{\text{train}})$.

See training error^a in the statistics notes.

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

Example 2.10 (Two models, scored on the same five neighbourhoods). Score two models on the five neighbourhoods of Table 1 under squared error: the least-squares line, and the constant that always predicts the sample mean crime rate.

2.2.3 R

```
# The five neighbourhoods of @tbl-crime-data, read once in @exm-crime-task.
x <- crime$x
y <- crime$y

emprisk_line <- mean(residuals(lm(y ~ x))^2)
emprisk_const <- mean((mean(y) - y)^2)
print(round(c(emprisk_line, emprisk_const), 6))

[1] 0.002721 0.023056
```

2.2.4 Python

```
import numpy as np

# The five neighbourhoods of the crime table, read once in the first example.
x = crime["x"].to_numpy()
y = crime["y"].to_numpy()

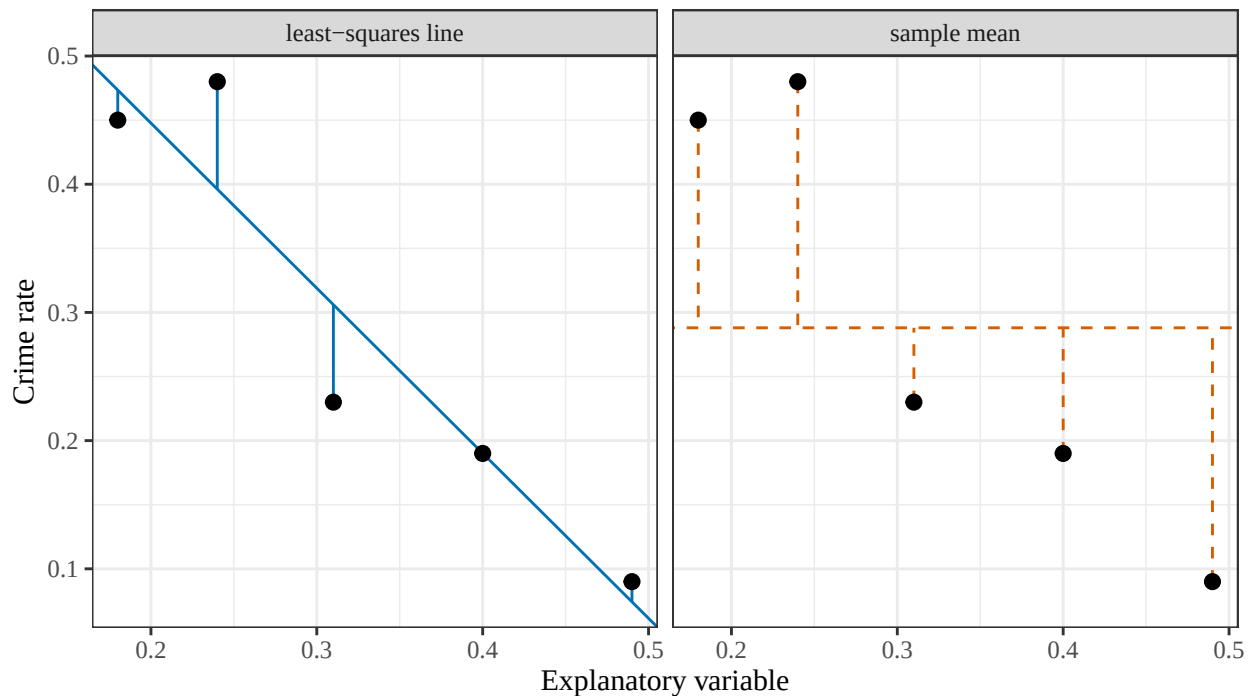
m, b = np.polyfit(x, y, 1)
emprisk_line = np.mean((m * x + b - y) ** 2)
emprisk_const = np.mean((y.mean() - y) ** 2)
print(round(emprisk_line, 6), round(emprisk_const, 6))

0.002721 0.023056
```

The line's empirical risk is 0.002721 and the constant's is 0.023056. The line scores better on these five neighbourhoods. Both models were fitted to them, so these are **training errors**, and the line fits the training set better. It says nothing yet about neighbourhoods outside the sample, which the risk (Definition 2.8) measures.

2.2.5 R

```
library(ggplot2)
erf_fit <- lm(y ~ x, data = crime)
erf_levels <- c("least-squares line", "sample mean")
erf_models <- data.frame(
  model = factor(erf_levels, levels = erf_levels),
  intercept = c(coef(erf_fit)[[1]], mean(crime$y)),
  slope = c(coef(erf_fit)[[2]], 0)
)
erf_points <- merge(crime, erf_models)
erf_points$predicted <- erf_points$intercept + erf_points$slope * erf_points$x
ggplot(erf_points, aes(x, y)) +
  geom_abline(
    data = erf_models,
    aes(intercept = intercept, slope = slope, colour = model, linetype = model)
  ) +
  geom_segment(aes(xend = x, yend = predicted, colour = model, linetype = model)) +
  geom_point(size = 2.5) +
  facet_wrap(vars(model)) +
  scale_colour_manual(values = c("least-squares line" = "#0072B2", "sample mean" = "#D55E00")) +
  scale_linetype_manual(values = c("least-squares line" = "solid", "sample mean" = "dashed")) +
  guides(colour = "none", linetype = "none") +
  labs(x = "Explanatory variable", y = "Crime rate")
```



(a)

2.2.6 Python

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

x_erf = crime["x"].to_numpy()
y_erf = crime["y"].to_numpy()
m_erf, b_erf = np.polyfit(x_erf, y_erf, 1)
grid_erf = np.array([x_erf.min(), x_erf.max()])
fig, axes = plt.subplots(1, 2, figsize=(7, 4), sharey=True, layout="constrained")
for ax, title, colour, style, intercept, slope in [
    (axes[0], "least-squares line", "#0072B2", "-", b_erf, m_erf),
    (axes[1], "sample mean", "#D55E00", "--", erf.mean(), 0),

```

i Remark (Empirical risk estimates the risk)

Remark 2.7 (Empirical risk estimates the risk). The **empirical risk** has a hat because it estimates the risk $R(h)$ of Equation 4 from the sample at hand: the sample average replaces the expectation. That average is over the empirical distribution^a of the pairs in the data set, which is what “empirical” refers to. On the training set the estimate tends to be too optimistic, because the model was fitted to those very pairs; Section 3 estimates the risk from pairs held out of the fit. Exercise 2.9, below, works the case of squared error and five points.

Unlike the risk, the empirical risk can be computed, because it needs only the data in hand and not the distribution they came from.

^a<https://morrison-lab.github.io/sds/nonparametric-models.html#def-empirical-distribution>

Theorem 2.1 (Under squared-error loss, the empirical risk is the sample mean squared error (sMSE)). Let $L(y, \hat{y}) = (\varepsilon)^2$ be the squared-error loss of Equation 3, where $\varepsilon = \hat{y} - y$ is the prediction error. For every model h and every training set $\mathcal{D}_{\text{train}}$ of n pairs, the **empirical risk** of h on $\mathcal{D}_{\text{train}}$ equals the sample mean squared error (sMSE) of h ’s predictions on $\mathcal{D}_{\text{train}}$:

$$\hat{R}(h; \mathcal{D}_{\text{train}}) = \text{sMSE}(h; \mathcal{D}_{\text{train}}). \quad (6)$$

i Proof

Proof. The empirical risk of Equation 5 is the average loss over the pairs of $\mathcal{D}_{\text{train}}$. For example i , the prediction error is $\varepsilon_{(i)} = h(x_{(i)}) - y_{(i)}$. Substituting the squared-error loss into that average gives

$$\hat{R}(h; \mathcal{D}_{\text{train}}) = \frac{1}{n} \sum_{i=1}^n (\varepsilon_{(i)})^2.$$

The sMSE is the average of the squared differences between each label and its prediction, which are the squared prediction errors. The right-hand side above is exactly that average. $\square$

Example 2.11 (The empirical risk and the sMSE of the least-squares line agree). Compute the empirical risk of the least-squares line of Example 2.10 two ways: by averaging the squared-error loss example by example, and by applying the sMSE formula to the vector of residuals.

```
import numpy as np

# The five neighbourhoods of the crime table, read once in the first example.
x = crime["x"].to_numpy()
y = crime["y"].to_numpy()
m, b = np.polyfit(x, y, 1)

# Empirical risk: average the loss of each example separately.
def squared_error(y_true, y_pred):
    return (y_true - y_pred) ** 2

emprisk_by_loss = np.mean([squared_error(y_i, m * x_i + b) for x_i, y_i in zip(x, y)])

# MSE: square the residual vector, then average.
residuals = y - (m * x + b)
mse_by_residuals = np.sum(residuals ** 2) / len(y)

print(round(emprisk_by_loss, 6), round(mse_by_residuals, 6))

0.002721 0.002721
```

The two computations give 0.002721 and 0.002721, as Theorem 2.1 says they must. They differ only in how the arithmetic is organized.

Definition 2.11 (Empirical risk minimization). **Empirical risk minimization** is choosing the h that makes the **empirical risk** smallest [8].

Exercise 2.9 (Score a whole model, not one prediction). Equation 3 scores one prediction. Write an expression that scores a candidate h against the whole training set $\mathcal{D}_{\text{train}}$ of Table 1, using squared error.

Solution

Solution 2.9. Average the loss over the training points. That average is the **empirical risk** (Definition 2.9):

$$\hat{R}(h; \mathcal{D}_{\text{train}}) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^n L(y_{(i)}, h(x_{(i)})) = \frac{1}{5} \sum_{i=1}^5 (h(x_{(i)}) - y_{(i)})^2 \quad (7)$$

Exercise 2.10 (Score a whole model, not one prediction (part 2)). Use the setup of Exercise 2.9. Say which symbols in your expression^a are fixed and which are free to vary.

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

Solution

Solution 2.10. The $x_{(i)}$ and $y_{(i)}$ are fixed: they are the data, and they do not change. What can vary is h — and since Equation 2 says h is fixed by m and b , Equation 7 is a function of those two numbers alone. Writing it that way turns the modelling question into an optimization problem:

$$\hat{m}, \hat{b} \stackrel{\text{def}}{=} \arg \min_{m, b} \hat{R}(h; \mathcal{D}_{\text{train}}) \quad (8)$$

These notes call solving Equation 8 training the model, also called learning.

Exercise 2.11 (Score a whole model, not one prediction (part 3)). Use the setup of Exercise 2.9. Compare your expression^a with the risk of Equation 4. What does it use in place of the expectation^b?

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

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

Solution

Solution 2.11. Compared with the risk of Equation 4, the **empirical risk** replaces the expectation^a over $P(x, y)$ with an average over the n points of $\mathcal{D}_{\text{train}}$. Those points are the only draws from $P(x, y)$ that we have.

The hat on $\hat{R}$ marks it as an estimate^b of the risk. Section 3 says how badly it can mislead.

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

^b<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

i Watch

Hutchinson works one example in which a model, a loss and the empirical risk appear, Machine Learning Example^a (21 min) [9]. The video site requires a login, which enrolled students find in the syllabus.

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

i Read

GeeksforGeeks explains:

- loss functions^a [10]
- empirical risk minimization^b [11]

^a<https://www.geeksforgeeks.org/machine-learning/ml-common-loss-functions/>

^b<https://www.geeksforgeeks.org/machine-learning/empirical-risk-minimization/>

2.3 Learning is optimization

Exercise 2.12 (Fit the line). Solve Equation 8 for the five points of Table 1.

You do not need to do it by hand: every language these notes use has a least-squares routine. Report $\hat{m}$, $\hat{b}$, and the empirical risk the fitted line achieves.

Hint

In Python, `numpy.polyfit(x, y, 1)`. In R, `lm(y ~ x)`.

Solution

Solution 2.12.

2.3.1 R

```
# The five neighbourhoods of @tbl-crime-data, read once in @exm-crime-task.
x <- crime$x
y <- crime$y

fit <- lm(y ~ x)
risk <- mean(residuals(fit)^2)
print(round(c(coef(fit)[["x"]], coef(fit)[["(Intercept)"]], risk), 6))

[1] -1.287671  0.705205  0.002721
```

2.3.2 Python

```
import numpy as np

# The five neighbourhoods of the crime table, read once in the first example.
x = crime["x"].to_numpy()
y = crime["y"].to_numpy()

m, b = np.polyfit(x, y, 1)
risk = np.mean((m * x + b - y) ** 2)
print(round(m, 6), round(b, 6), round(risk, 6))

-1.287671 0.705205 0.002721
Both return the same fit,
```

$$\hat{h}(x) = -1.2877x + 0.7052 \quad (9)$$

at an empirical risk of 0.002721. Figure 8 draws it.

```

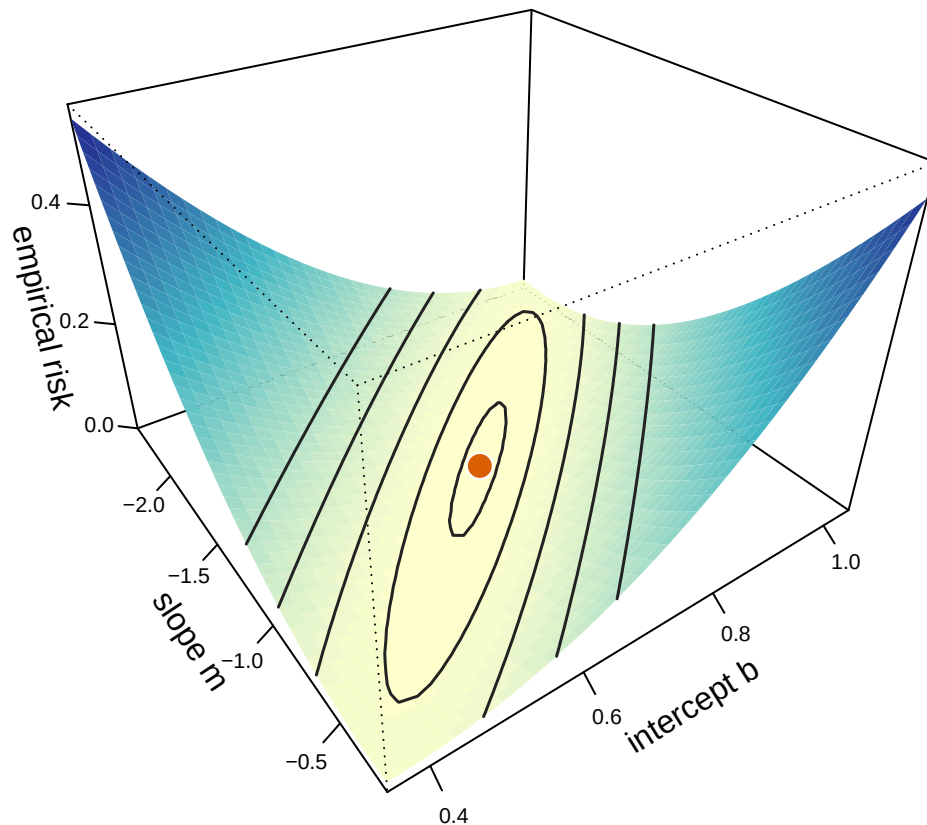
# The empirical risk of the line  $m x + b$  on a grid of slopes and intercepts,
# for the five neighbourhoods the chapter fits. Both drawings below read it.
# The five neighbourhoods of @tbl-crime-data, read once in @exm-crime-task.
x <- crime$x
y <- crime$y

m_grid <- seq(-2.4, -0.2, by = 0.04)
b_grid <- seq(0.35, 1.05, by = 0.02)
risk_of <- function(m, b) mean((m * x + b - y)^2)
risk <- outer(m_grid, b_grid, Vectorize(risk_of))

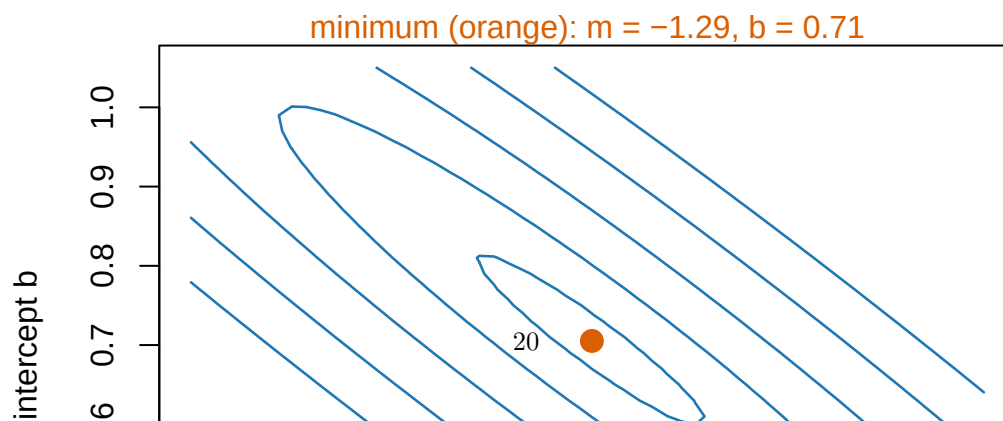
# The least-squares line, the lowest point of the surface.
fit <- lm(y ~ x)
best_m <- coef(fit)[["x"]]
best_b <- coef(fit)[["(Intercept)"]]
best_risk <- risk_of(best_m, best_b)

# The levels of the contour lines, the same on the surface and in the plot.
levels <- c(0.004, 0.012, 0.030, 0.060, 0.100)

```



(a) The surface itself (drag to rotate it on the course website). The black lines on it are the contours of the next panel.



Exercise 2.13 (Read the surface). Equation 8 asks for the lowest point of a surface. For the straight lines of Equation 2 that surface is plotted over two axes, m and b , and Figure 7 draws it. How many points on the surface are lower than every other point?

Solution

Solution 2.13. One. The contours shrink to a single point, and every straight-line slice through that point has its minimum there too. That point has lower risk than every other point in the whole plane, so it is the surface's global minimizer^a, and $(\hat{m}, \hat{b})$ in Equation 8 is the single element of the argmin^b.

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

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

Exercise 2.14 (Read the surface (part 2)). Use the setup of Exercise 2.13. Would a search for the lowest point give a different answer depending on where it started?

Solution

Solution 2.14. No: the surface has no other local minimizer^a that a search could stop at, so wherever a search starts it ends at the same point. So both languages in Solution 2.3.2 agree without anyone choosing a starting guess.

Squared error and a model linear in its parameters give a surface that is a paraboloid^b, and a paraboloid has one minimum, and it has a closed-form solution^c. Neither property holds for most models in the rest of these notes: Gradient Descent starts on what replaces them.

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

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

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

The course website's version of this page has an interactive plot here.

i Watch

Lall treats training as minimizing an empirical risk in EE104, from 3:09^a, and defines the loss, from 5:14^b [12], Lecture 6.

^ahttps://www.youtube.com/watch?v=SwH0rs5TW_Q&t=189s

^bhttps://www.youtube.com/watch?v=SwH0rs5TW_Q&t=314s

2.4 A more flexible model

Exercise 2.15 (Now fit a cubic). Nothing in Equation 5 mentions straight lines. Repeat Exercise 2.12 with the cubic polynomial^a

$$h(x) = m_3x^3 + m_2x^2 + m_1x + b \quad (10)$$

and report its empirical risk beside the line's.

Then continue: fit polynomials of degree^b 2 and 4 as well, and tabulate degree, number of parameters, and empirical risk.

^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>

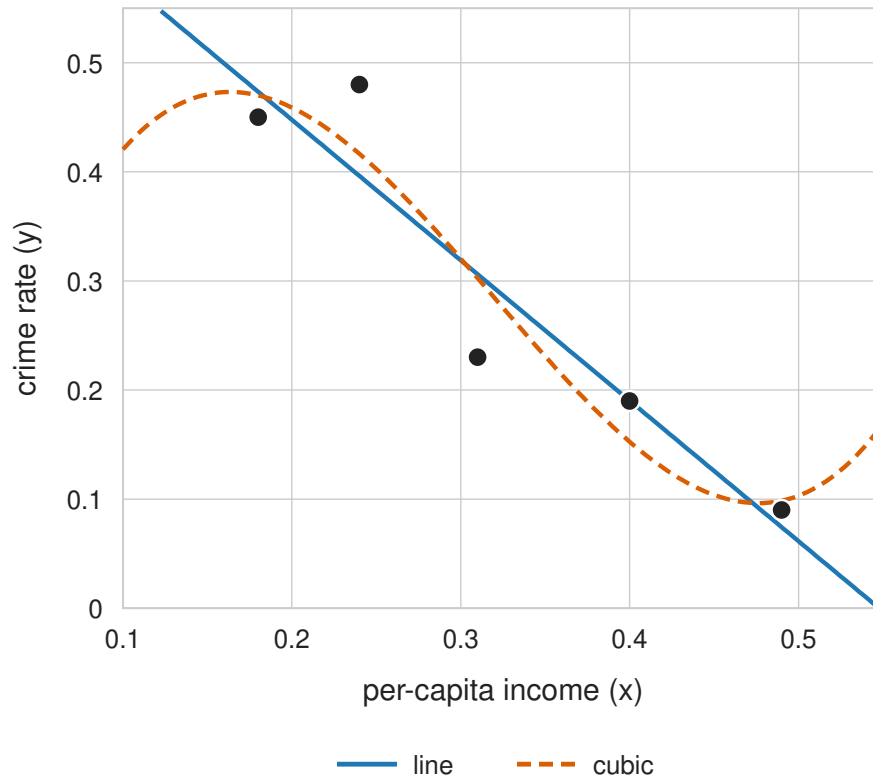


Figure 8: The fitted line of Equation 9 and the cubic fitted in Exercise 2.15.

Solution

Solution 2.15. Change one argument, and run it over a range of values:

2.4.1 R

```
# The five neighbourhoods of @tbl-crime-data, read once in @exm-crime-task.
x <- crime$x
y <- crime$y

for (degree in 1:4) {
  fit <- lm(y ~ poly(x, degree, raw = TRUE))
  risk <- mean(residuals(fit)^2)
  cat(degree, degree + 1, round(risk, 6), "\n")
}
```

```
1 2 0.002721
2 3 0.002608
3 4 0.002234
4 5 0
```

2.4.2 Python

```

import numpy as np

# The five neighbourhoods of the crime table, read once in the first example.
x = crime["x"].to_numpy()
y = crime["y"].to_numpy()

for degree in (1, 2, 3, 4):
    coeffs = np.polyfit(x, y, degree)
    risk = np.mean((np.polyval(coeffs, x) - y) ** 2)
    print(degree, degree + 1, round(risk, 6))

1 2 0.002721
2 3 0.002608
3 4 0.002234
4 5 0.0

# The Python fits again, kept for the equation, the table and the prose.
fits = {degree: np.polyfit(x, y, degree) for degree in (1, 2, 3, 4)}
risks = [float(np.mean((np.polyval(fits[d], x) - y) ** 2)) for d in (1, 2, 3, 4)]
cubic = [float(c) for c in fits[3]]
quartic_max_resid = float(np.max(np.abs(np.polyval(fits[4], x) - y)))

risks <- py$risks
cubic <- py$cubic
# A signed term of the fitted polynomial, for display math: "- 23.6095\,x^2".
term <- function(coef, power) {
    sign <- if (coef < 0) "-" else "+"
    x <- c("", "\\,x", "\\,x^2", "\\,x^3")[power + 1]
    sprintf("%s %.4f%s", sign, abs(coef), x)
}
e6 <- function(v) sprintf("%.6f", v)
risk_cut <- round(100 * (1 - risks[3] / risks[1]))
# The quartic's risk prints as 0.000000 at six decimals; check that its
# largest residual is below the last printed place, so the table's zero is honest.
max_resid <- py$quartic_max_resid
stopifnot(max_resid < 1e-6)

```

The cubic comes out at

$$\hat{h}(x) = 24.6063 x^3 - 23.6095 x^2 + 5.7443 x + 0.0576 \quad (11)$$

and the four fits compare like this:

Table 2: Empirical risk falls monotonically as the model gains parameters. The quartic's risk is not rounded down: its largest residual on these five points is below 0.000001, the last place the table prints.

degree	parameters	empirical risk
1	2	0.002721
2	3	0.002608
3	4	0.002234
4	5	0.000000

Two things in Table 2 are worth examining closely.

Among the fits that do not pass through every point, the cubic has the lowest risk, but it improves on the line by very little: 0.002234 against 0.002721 is a reduction of 18%, obtained by using twice as many parameters. Figure 8 shows what that 18% looks like.

The quartic has the lowest risk, scoring exactly zero. That is not a good model; it is five parameters fitted to five points, so it passes through every one of them by construction and would do so whatever the numbers had been.

Watch

Hastie and Tibshirani compare structured and flexible models^a [6], video 2.2.

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

The course website's version of this page has an interactive plot here.

Exercise 2.16 (Which model was actually better?). Table 2 ranks the four fits, and the quartic comes first. Say why that ranking is not a reason to prefer it, and name what you would have to measure instead.

Solution

Solution 2.16. Because the ranking is computed on the very points the models were fitted to. Any flexibility a model has can be used to fit those points closely, so Equation 5 rewards flexibility and accuracy alike and does not distinguish them. The quartic makes that visible by reaching zero: with as many parameters as data points it interpolates, and interpolation is available to it regardless of whether the data has any pattern at all.

What we want is the loss on points the model has **not** seen. That quantity is the risk (Definition 2.8), and it can only ever be estimated; Section 3 says how.

Every honest estimate of it rests on data held out of the fitting.

This is the whole of the bias-variance story in small form, and it is worth noticing that nothing about it is specific to polynomials. Swap in any family whose members can be made arbitrarily flexible and the same thing happens. Hastie et al. [2], Section 2.9, makes the general argument.

Definition 2.12 (Interpolation). A model h **interpolates** a data set^a $\mathcal{D}$ when it predicts every output in $\mathcal{D}$ exactly: $h(x_{(i)}) = y_{(i)}$ for each of the n pairs^b $(x_{(i)}, y_{(i)})$ of $\mathcal{D}$.

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

^b<https://morrison-lab.github.io/mds/sets.html#def-ordered-pair>

Example 2.12 (The quartic passes through all five neighbourhoods). Predict the five neighbourhoods of Table 1 with the quartic of Table 2, and compare each prediction with the observed crime rate.

```
# fits, x and y are the polynomial fits and the five neighbourhoods of @sol-fit-cubic.
quartic_pred = np.polyval(fits[4], x)
print(np.round(quartic_pred, 6))
```

```
[0.09 0.45 0.23 0.19 0.48]
```

```
print(np.round(y, 6))
```

```
[0.09 0.45 0.23 0.19 0.48]
```

The two rows agree, and the largest residual is 8.7e-14, which is floating-point rounding error. The quartic has five **parameters**, and the five neighbourhoods have distinct inputs, so it can pass through all of them: it **interpolates** the **training set**, and its **training error** under squared-error loss is zero. The line, the quadratic and the cubic have positive **training errors** in Table 2, so none of them interpolates.

Exercise 2.17 (What we made easy). We took three easy steps in the crime example: we found the lowest point of the risk surface, we chose a model family, and we scored the fitted models. For each step, say what about this example made it easy for us, and what would make it hard on a larger problem.

Solution

Solution 2.17. We made each step easy on purpose, and later chapters remove each simplification.

- **We found the minimum exactly.** We can minimize a paraboloid^a by solving a linear system. We cannot do that for a neural network's risk surface, so we reduce the risk step by step instead. That is why the derivative of the loss matters so much. Gradient Descent starts on that.
- **We chose the model by eye.** Five points and one input let us look at the scatter plot. With many inputs there is nothing to look at, so we have to argue for the choice of family and then check it.
- **We computed the score on the training data.** Solution 2.16 says why that is not a measurement of anything we care about. Section 3 says what to do instead.

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

3 Generalization

i Watch

Hutchinson covers generalization, bias, variance and overfitting, Generalization^a (31 min) [9].

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

Definition 3.1 (Objective function (objective, cost function)). An **objective function**, or **objective** for short [13, Sec. 1.1] and 4.1, is the quantity a training procedure minimizes (or maximizes) as the model's parameters vary. It is a function of the parameters alone, with the data held fixed. For a model h_θ with parameters θ and a training set $\mathcal{D}_{\text{train}}$, the objective that is the empirical risk (Equation 5) is

$$J(\theta) \stackrel{\text{def}}{=} \hat{R}(h_\theta; \mathcal{D}_{\text{train}}), \quad (12)$$

and training minimizes $J(\theta)$ over θ .

Other names: the objective is also called the **cost function**, though that name is also used for the loss and for the empirical risk, so it identifies nothing on its own.

Example 3.1 (The objective as a function of the slope and intercept). For the crime data of Table 1, the objective is the empirical risk as a function of the pair (m, b) .

```
# The five neighbourhoods of @tbl-crime-data, read once in @exm-crime-task.
x <- crime$x
y <- crime$y

objective <- function(m, b) mean((m * x + b - y)^2)

fit <- lm(y ~ x)
m_hat <- coef(fit)[["x"]]
b_hat <- coef(fit)[["(Intercept)"]]
print(round(c(objective(0, mean(y)), objective(m_hat, b_hat)), 6))
```

```
[1] 0.023056 0.002721
```

The flat line $(m, b) = (0, \bar{y})$ has objective value 0.023056, and the least-squares pair $(\hat{m}, \hat{b}) = (-1.2877, 0.7052)$ has the smaller value 0.002721. Training searches over (m, b) for the pair with

the smallest value.

i Remark (The empirical risk is one objective among many)

Remark 3.1 (The empirical risk is one objective among many). So far the only **objective** in these notes is the **empirical risk**, which Equation 8 minimizes over the slope and intercept. Later chapters change the objective without changing the loss: whatever is handed to the optimizer is the objective, and the empirical risk is only one thing it can be made from.

Exercise 3.1 (Estimate what cannot be computed). Equation 4 cannot be evaluated and Equation 5 can. Solution 2.16 rules out reporting the second in place of the first. Describe a procedure that uses only the available data and returns an honest estimate of Equation 4, and say what the procedure costs.

Solution

Solution 3.1. Set some data aside. Split $\mathcal{D}_{\text{train}}$ before fitting anything; fit on one part; score on the other. The scored part played no role in choosing the model's parameters, so its average loss is an *unbiased* estimate of Equation 4: over repeated draws of the held-out points, it equals the risk on average, neither systematically above nor below.

The cost is the data itself. Points moved into the held-out set are points the model never learns from, so the model is worse than it needed to be, and the estimate depends on however few points were set aside and is noisy in proportion.

The usual arrangement is three sets rather than two:

1. A training set (Definition 2.2), which the fitting procedure sees.
2. A validation set (Definition 3.2; also called a development or dev set), scored repeatedly while choosing between models, tuning their settings, and deciding when to stop.
3. A test set (Definition 3.4), scored exactly once, at the end, on the model finally chosen. It stays unused until then.

The third set exists because of what the second one does. A validation set consulted a hundred times is a set a hundred decisions have been fitted to, and its score stops being honest about a model those decisions selected. Section 3.1 covers what to do when there is not enough data for all three.

Example 3.2 (Splitting the Auto data). Split the cars of the Auto data [3] at random, 60% for training, 20% for validation and the rest for testing.

```
# Draw the three index sets from one shuffled list of cars, so that they cannot overlap.
import numpy as np
import pandas as pd
```

```
auto_split = pd.read_csv("data/auto.csv").dropna(subset=["horsepower"])
n_cars = len(auto_split)
shuffled = np.random.default_rng(571).permutation(n_cars)
n_train = int(0.6 * n_cars)
n_val = int(0.2 * n_cars)
idx_train = shuffled[:n_train]
idx_val = shuffled[n_train:n_train + n_val]
idx_test = shuffled[n_train + n_val:]
print(n_train, n_val, len(idx_test))
```

235 78 79

Of the 392 cars with a recorded horsepower, 235 form the training set, which is all the fitting procedure sees, and the other 157 are held out of the fit.

Definition 3.2 (Validation set). A **validation set** is a subset of the data set^a, the pairs indexed by a set I_{val} that shares no index with I_{train} , and that we use for model selection^b among candidate models fitted to the **training set** [4]:

$$\mathcal{D}_{\text{val}} \stackrel{\text{def}}{=} \{(x_i, y_i) : i \in I_{\text{val}}\} \subseteq \mathcal{D}, \quad I_{\text{val}} \cap I_{\text{train}} = \emptyset. \quad (13)$$

The model with the smallest **empirical risk** on the validation set is the one we keep.

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

^b<https://morrison-lab.github.io/sds/model-validation.html#def-model-selection>

Definition 3.3 (Validation error). The **validation error** of a model h is its **empirical risk** on the **validation set**, $\hat{R}(h; \mathcal{D}_{\text{val}})$.

See validation error^a in the statistics notes.

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

Example 3.3 (Choosing a polynomial degree). Fit polynomials of degree 1 to 9 to the 235 training cars of Example 3.2, predicting mpg from horsepower, and score each on the 78 validation cars.

```
# Scale horsepower to 0 to 1 so that high degrees are numerically stable.
hp = auto_split["horsepower"].to_numpy(dtype=float)
hp = (hp - hp.min()) / (hp.max() - hp.min())
mpg = auto_split["mpg"].to_numpy(dtype=float)

def mse(coef, idx):
    return np.mean((np.polyval(coef, hp[idx]) - mpg[idx]) ** 2)

degrees = range(1, 10)
fits = {d: np.polyfit(hp[idx_train], mpg[idx_train], d) for d in degrees}
val_mse = {d: mse(fits[d], idx_val) for d in degrees}
train_mse = {d: mse(fits[d], idx_train) for d in degrees}
best_degree = min(val_mse, key=val_mse.get)
train_mse_9 = train_mse[9]
val_mse_9 = val_mse[9]
val_mse_best = val_mse[best_degree]

def mse_ci(coef, idx):
    # Normal-approximation 95% interval for the mean of the squared errors.
    sq = (np.polyval(coef, hp[idx]) - mpg[idx]) ** 2
    se = sq.std(ddof=1) / np.sqrt(len(sq))
    return sq.mean() - 1.96 * se, sq.mean() + 1.96 * se

val_ci_best = mse_ci(fits[best_degree], idx_val)
val_ci_9 = mse_ci(fits[9], idx_val)
print(best_degree, round(val_mse[best_degree], 3), round(train_mse_9, 3), round(val_mse_9, 3))

2 18.006 16.78 19.559

print(np.round(val_ci_best, 3), np.round(val_ci_9, 3))

[12.012 23.999] [11.9 27.218]
```

The degree with the smallest validation error, 2, is the one we keep. The training cars could not make this choice, because the training error keeps falling as the degree grows: the degree-9 fit has training error 16.78 and validation error 19.56. The chosen degree's validation error, 18.01, has a 95% confidence interval of 12.01 to 24.00, and the degree-9 fit's has 11.90 to 27.22. Each interval comes from the 78 validation cars alone, and the chosen degree's is optimistic, because the same cars picked the degree.

Definition 3.4 (Test set). A **test set** is a subset of the data set^a, the pairs indexed by a set I_{test} that shares no index with I_{train} or I_{val} , and that we use only once, after the model is chosen, to estimate the **risk** of the chosen model h by its **empirical risk** on the test set, $\hat{R}(h; \mathcal{D}_{\text{test}})$:

$$\mathcal{D}_{\text{test}} \stackrel{\text{def}}{=} \{(x_i, y_i) : i \in I_{\text{test}}\} \subseteq \mathcal{D}, \quad I_{\text{test}} \cap (I_{\text{train}} \cup I_{\text{val}}) = \emptyset. \quad (14)$$

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

Definition 3.5 (Test error). The **test error** of a model h is its **empirical risk** on the **test set**, $\hat{R}(h; \mathcal{D}_{\text{test}})$. See test error^a in the statistics notes.

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

Example 3.4 (Reporting the chosen degree's error). Having chosen degree 2 on the validation cars of Example 3.3, score that one model on the 79 test cars.

```
# Refit the chosen degree on the training cars of @exm-training-set (scaled horsepower hp, mpg),
# so this chunk does not lean on the fits table of the validation example.
chosen_coef = np.polyfit(hp[idx_train], mpg[idx_train], best_degree)
# A normal-approximation 95% interval: the mean of the squared errors
# plus or minus 1.96 standard errors.
test_sq = (np.polyval(chosen_coef, hp[idx_test]) - mpg[idx_test]) ** 2
test_mse = np.mean(test_sq)
test_se = test_sq.std(ddof=1) / np.sqrt(len(test_sq))
test_lo, test_hi = test_mse - 1.96 * test_se, test_mse + 1.96 * test_se
print(round(test_mse, 3), round(test_lo, 3), round(test_hi, 3))
```

24.232 14.183 34.281

The estimate of the chosen model's risk is 24.23, with a 95% confidence interval of 14.18 to 34.28. The test cars played no part in the fit or the choice, so the choice did not tune this estimate to them. If it disappoints and we go back to choose again, the test cars have become a validation set.

Definition 3.6 (Generalization gap). The **generalization gap** of a model h on a **training set** $\mathcal{D}_{\text{train}}$ is its **risk** minus its **training error** [14, Sec. 5.2]:

$$R(h) - \hat{R}(h; \mathcal{D}_{\text{train}}). \quad (15)$$

Example 3.5 (Estimating the chosen degree's gap). The **risk** in Equation 15 cannot be computed, so estimate^a it by the **test error** of Example 3.4, and subtract the **training error** of the same degree-2 fit.

```
# train_mse holds each degree's training error from @exm-validation-set;
# test_mse is the chosen degree's test error from @exm-test-set.
train_mse_best = train_mse[best_degree]
gap_estimate = test_mse - train_mse_best
print(round(train_mse_best, 3), round(test_mse, 3), round(gap_estimate, 3))
```

17.693 24.232 6.539

The training error is 17.69 and the test error is 24.23, so the estimated^b gap is 6.54. The estimate is noisy: the 95% interval for the test error in Example 3.4 runs from 14.18 to 34.28, and it contains the training error, so these cars alone do not show that the gap is positive.

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^b<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

i Remark (The three sets together)

Remark 3.2 (The three sets together). The three index sets I_{train} , I_{val} and I_{test} are disjoint and together cover $1, \dots, n$, so the training, validation and test sets are disjoint subsets that together make up the data set $\mathcal{D}$, which is why Example 3.2 draws them from one shuffled list. The sets are not interchangeable: the first is for fitting, the second for choosing among fits, and the third for reporting. Data used for one job gives an optimistic answer to the next, as Figure 9 shows. Any pairs kept out of the fit are called *held-out*, so validation and test sets are both held-out sets.

The statistics notes define the same three sets formally: the training set^a, the validation set^b, and the test set^c. They use a different symbol for the training set and a plain observation index, but the same three-way split.

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

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

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

i Remark (What the validation and test errors estimate)

Remark 3.3 (What the validation and test errors estimate). Like the **training error**, the **validation error** and the **test error** are empirical risks, and they differ only in the set that scores the model. The model was not fitted to the validation or test set, so each estimates the **risk** $R(h)$ without the optimism of the training error. We choose the model with the smallest validation error, so the validation error of the chosen model is still optimistic. We compute the test error once, after the choice, so it is not.

Some textbooks use “test error” for the expected error of the fitted model on a new observation, which is the **risk** of the fitted model, not a number computed from a test set [2, Sec. 7.2]. Here the test error is the number computed from the test set, and the risk is what it estimates.

These ideas hold for any loss, not only squared error. Each error is the average loss on the set that scores the model: the training error^a, the validation error^b and the test error^c. The expected loss of the fitted model on a new observation is its generalization error^d, the risk of the fitted model. Averaging the generalization error over training sets gives the expected generalization error^e.

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

^b<https://morrison-lab.github.io/sds/model-validation.html#def-validation-error>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-test-error>

^d<https://morrison-lab.github.io/sds/model-validation.html#def-generalization-error>

^e<https://morrison-lab.github.io/sds/model-validation.html#def-expected-generalization-error>

```
# The constants of the static figure, so its prose and plot agree.
```

```
tv_n <- 10      # training cars
```

```
tv_degree <- 3  # degree of the polynomial drawn on the left
```

```
tv_degrees <- 0:9
```

```
library(ggplot2)
```

```
# The Auto cars with a horsepower, as in the interactive figure:
```

```
# the polynomials are fitted with horsepower rescaled to [0, 1] and mpg standardized,
```

```
# and the plots convert back, so every axis and number is in horsepower and mpg.
```

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

```
auto <- auto[!is.na(auto$horsepower), ]
```

```
hp_min <- min(auto$horsepower)
```

```
hp_range <- diff(range(auto$horsepower))
```

```
mpg_mean <- mean(auto$mpg)
```

```
mpg_sd <- sd(auto$mpg)
```

```
cars <- data.frame(
```

```
  x = (auto$horsepower - hp_min) / hp_range,
```

```
  y = (auto$mpg - mpg_mean) / mpg_sd
```

```
)
```

```
# Back to the original units, for plotting.
```

```

to_hp <- function(x) x * hp_range + hp_min
to_mpg <- function(y) y * mpg_sd + mpg_mean
set.seed(2100)
train_rows <- sample(nrow(cars), tv_n)
train <- cars[train_rows, ]
held <- cars[-train_rows, ]
# Least-squares polynomial of the given degree; degree 0 is the constant fit,
# which poly() does not accept. poly() stops when the degree is not below the number
# of distinct x values, so cars that share a horsepower lower the degree actually fitted.
fit_poly <- function(data, degree) {
  degree <- min(degree, length(unique(data$x)) - 1)
  if (degree == 0) lm(y ~ 1, data = data) else lm(y ~ poly(x, degree), data = data)
}
# Squared errors of a fit on a set of cars. The 95% confidence interval for the
# held-out RMSE is the mean squared error plus or minus 1.96 standard errors
# (central limit theorem), with its endpoints square-rooted.
# The RMSEs are multiplied by the sd of mpg, which puts them in mpg.
sq_errors <- function(fit, data) (predict(fit, data) - data$y)^2
scores <- do.call(rbind, lapply(tv_degrees, function(d) {
  fit <- fit_poly(train, d)
  e <- sq_errors(fit, held)
  half <- 1.96 * sd(e) / sqrt(length(e))
  data.frame(
    degree = d,
    set = c("training points", "held-out points"),
    # Clamped to the bottom of the log axis, so an RMSE of exactly 0 is drawn on the edge.
    rmse = pmax(mpg_sd * c(sqrt(mean(sq_errors(fit, train))), sqrt(mean(e))), 1e-1),
    lower = c(NA, mpg_sd * sqrt(max(mean(e) - half, 0))),
    upper = c(NA, mpg_sd * sqrt(mean(e) + half))
  )
}))
# Left panel: the cars and the fitted curve.
grid_x <- data.frame(x = seq(0, 1, length.out = 200))
grid_x$y <- predict(fit_poly(train, tv_degree), grid_x)
p_fit <- ggplot() +
  geom_point(data = held, aes(to_hp(x), to_mpg(y)), colour = "grey75", size = 0.8) +
  geom_line(data = grid_x, aes(to_hp(x), to_mpg(y)), colour = "#1f77b4", linewidth = 1) +
  geom_point(data = train, aes(to_hp(x), to_mpg(y)), size = 1.8) +
  coord_cartesian(ylim = to_mpg(c(-2, 3.5))) +
  labs(x = "horsepower", y = "mpg")
# Right panel: training and held-out RMSE against degree, on a log scale.
p_err <- ggplot(scores, aes(degree, rmse, colour = set)) +
  geom_vline(xintercept = tv_degree, colour = "grey55") +
  geom_linerange(
    data = scores[scores$set == "held-out points", ],
    aes(ymin = pmax(lower, 1e-1), ymax = upper)
  ) +
  geom_line(linewidth = 0.8) +
  geom_point() +
  scale_x_continuous(breaks = tv_degrees) +
  scale_y_log10(
    limits = c(1e-1, 1e2), oob = scales::squish,
    breaks = 10^(-1:2), labels = c("0.1", "1", "10", "100")
  ) +
  scale_colour_manual(
    values = c("training points" = "#1f77b4", "held-out points" = "#ff7f0e")
  )

```

```
) +
labs(x = "degree", y = "RMSE (mpg, log scale)", colour = NULL) +
theme(legend.position = "bottom")
```

```
# Side by side, with base grid, since no layout package is in the render workflows.
grid::grid.newpage()
grid::pushViewport(grid::viewport(layout = grid::grid.layout(1, 2)))
print(p_fit, vp = grid::viewport(layout.pos.row = 1, layout.pos.col = 1))
print(p_err, vp = grid::viewport(layout.pos.row = 1, layout.pos.col = 2))
```

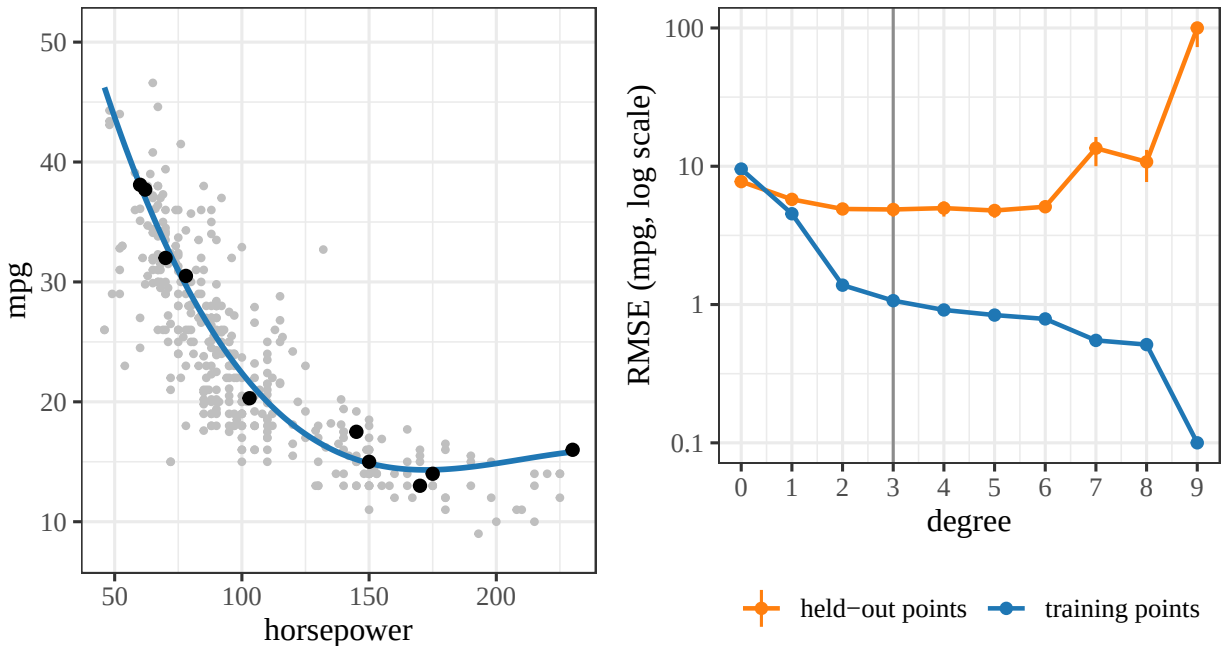


Figure 9: Static version of the interactive plot on the course website: polynomials fitted to a random sample of 10 of the cars in the Auto data [3], mpg against horsepower, scored on the cars in the sample (training) and on the rest (held out), with 95% confidence intervals on the held-out sRMSE. Left, the 10 training cars (black), the held-out cars (grey) and the degree-3 fit (blue). Right, the sRMSE at each degree from 0 to 9, with the degree-3 fit marked by a vertical line. The axes are in horsepower and miles per gallon, and the sRMSE is in miles per gallon.

i Remark (Reading the figure on training and held-out error)

Remark 3.4 (Reading the figure on training and held-out error). The polynomials are fitted with horsepower scaled to run from 0 to 1 and mpg standardized, but every axis and number here is in the original units, horsepower and miles per gallon. The sRMSE is the square root of the sample mean squared error, so it is in the units of the response, here miles per gallon (why we report sRMSE). The vertical bars on the held-out curve are 95% confidence intervals for the held-out sRMSE: the mean of the squared errors plus or minus 1.96 standard errors, from the central limit theorem, with the endpoints square-rooted. It is rough when a high-degree fit makes a few squared errors enormous, and it describes the risk of this fitted polynomial, not of the fitting method. The training error has no interval: the fit was chosen to make those errors small, so they are not independent draws and their mean is biased low as an estimate of the risk, and an interval around it would be centred in the wrong place.

Exercise 3.2 (How far can a held-out score be trusted?). A spam filter h was trained on one set of emails and then scored on a fresh held-out set $\mathcal{V}$ of n emails, drawn independently from the same

distribution as the emails it will face. Its held-out score $\hat{R}(h; \mathcal{V})$ is the fraction of those emails it labels wrong, and its risk $R(h)$ is its error rate on the population. Find an upper bound on the chance that the held-out score misses the risk by more than some $\varepsilon > 0$, using only the mean and variance of the held-out score.

Solution

Solution 3.2. Write Z_i for the loss on the i th held-out email: 1 if h labels it wrong, and 0 if right. The filter was fixed before $\mathcal{V}$ was drawn, so the Z_i are independent, and each is 1 with probability $R(h)$. The held-out score is their average, $\hat{R}(h; \mathcal{V}) = \frac{1}{n} \sum_{i=1}^n Z_i$, so its mean is $R(h)$. Its variance is the variance of one Z_i divided by n . A 0/1 variable that is 1 with probability R has variance $R(1 - R)$, and that is never more than $\frac{1}{4}$:

$$R(1 - R) = \frac{1}{4} - \left(R - \frac{1}{2}\right)^2 \leq \frac{1}{4}.$$

So $\text{Var}(\hat{R}(h; \mathcal{V})) \leq \frac{1}{4n}$, whatever the filter's error rate is. Chebyshev's inequality bounds the chance of a miss by the variance divided by the square of the miss:

$$P(|\hat{R}(h; \mathcal{V}) - R(h)| > \varepsilon) \leq \frac{\text{Var}(\hat{R}(h; \mathcal{V}))}{\varepsilon^2} \leq \frac{1}{4n\varepsilon^2}.$$

Exercise 3.3 (How far can a held-out score be trusted? (part 2)). Use the setup of Exercise 3.2. How does your bound change as n grows?

Solution

Solution 3.3. The bound shrinks in proportion to $1/n$: ten times as many held-out emails make a miss of a given size at most a tenth as likely.

```
n_example <- 1000
eps_example <- 0.05
cheb_example <- 1 / (4 * n_example * eps_example^2)
```

For example, with $n = 1000$ and $\varepsilon = 0.05$, the chance of a miss is at most $1/(4n\varepsilon^2) = 0.1$. Chebyshev uses only the variance^a, and the 0/1 range entered only through the cap of $\frac{1}{4}$ on it. Using the range itself, that each Z_i lies between 0 and 1, gives a much sharper bound, one that shrinks exponentially in n rather than in proportion to $1/n$.

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

Theorem 3.1 (Hoeffding's inequality for a held-out set). *Score a classifier h with the misclassification loss. Let $\mathcal{D}_{\text{val}}$ be a held-out set of n points drawn independently from the same distribution $P(x, y)$, and let $\hat{R}(h; \mathcal{D}_{\text{val}})$ be the fraction of them h gets wrong, as in Equation 5. If h is fixed before $\mathcal{D}_{\text{val}}$ is drawn, then for every $\varepsilon > 0$ [15],*

$$P(|\hat{R}(h; \mathcal{D}_{\text{val}}) - R(h)| > \varepsilon) \leq 2e^{-2\varepsilon^2 n}. \quad (16)$$

i Proof

Proof. The misclassification loss of h on the i th held-out point, $L(y_{(i)}, h(x_{(i)}))$, is 1 when h gets the point wrong and 0 otherwise. Because h is fixed before $\mathcal{D}_{\text{val}}$ is drawn, each loss is the same function of one of n independent draws from $P(x, y)$. So the n losses are IID^a random variables in $[0, 1]$. Their

common mean is $R(h)$ by Equation 4, and their average is $\hat{R}(h; \mathcal{D}_{\text{val}})$ by Equation 5. Hoeffding's inequality for an average of IID variables in $[0, 1]$ ^b bounds the chance that the average misses the mean by at least ε by $2e^{-2\varepsilon^2 n}$. The event “misses by more than ε ” is part of that event, so its chance is at most as large. $\square$

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

^b<https://morrison-lab.github.io/pds/limit-theorems.html#cor-hoeffding-two-sided>

i Remark

Remark. Abu-Mostafa et al. [1, Sec. 1.3.2] states the inequality for a label that is a fixed function of x , and in section 1.4.2 extends its analysis to labels with noise, as here.

i Remark (Reading Hoeffding's inequality)

Remark 3.5 (Reading Hoeffding's inequality). The misclassification loss is 1 when $h(x) \neq y$ and 0 otherwise, so $R(h)$ in Equation 4 is the classifier's error rate on the population. The probability in Equation 16 is over the draw of $\mathcal{V}$; $R(h)$ is unknown but not random.

Exercise 3.4 (How many held-out emails are enough?). A spam filter was trained on one set of emails, and you will estimate its error rate on a fresh held-out set. Say the estimate *misses* when it differs from the filter's true error rate by more than 0.05.

How many held-out emails make the chance of a miss at most 0.05?

Solution

Solution 3.4. In Exercise 3.4, the filter was fixed before the held-out set was drawn, so Equation 16 applies. Write ε for the allowed miss and δ for the allowed chance. The chance of a miss is at most δ whenever the bound is:

$$\begin{aligned} 2e^{-2\varepsilon^2 n} &\leq \delta \\ e^{-2\varepsilon^2 n} &\leq \frac{\delta}{2} \\ -2\varepsilon^2 n &\leq \ln \frac{\delta}{2} \\ 2\varepsilon^2 n &\geq -\ln \frac{\delta}{2} = \ln \frac{2}{\delta} \\ n &\geq \frac{\ln(2/\delta)}{2\varepsilon^2}. \end{aligned}$$

The fourth line multiplies both sides by -1 , which flips the inequality, and the fifth divides by $2\varepsilon^2$.

```
eps <- 0.05
delta <- 0.05
n_one <- ceiling(log(2 / delta) / (2 * eps^2))
```

With $\varepsilon = 0.05$ and $\delta = 0.05$, that is $n \geq 738$ emails. The bound is loose, so fewer emails may well do, but 738 are enough whatever the filter's error rate is.

Exercise 3.5 (How many held-out emails are enough? (part 2)). Use the setup of Exercise 3.4. Suppose instead that 100 different filters, all trained before the held-out set was drawn, are scored on the same held-out set, and you keep the one with the lowest score. How many held-out emails make the chance that the kept filter's estimate^a misses at most 0.05?

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Solution

Solution 3.5. In Exercise 3.5, the kept filter was chosen *by* its held-out score, so it was not fixed before the held-out set was drawn, and Equation 16 does not apply to it directly.

```
n_models <- 100
```

It does apply to each of the $M = 100$ filters $h_1, \dots, h_M$, since each was trained before the held-out set was drawn. The kept filter's estimate^a can miss only if at least one filter's estimate does, and the chance that at least one of several events happens is at most the sum of their chances. So

$$\begin{aligned} P(\text{the kept filter misses}) &\leq P(\text{some } h_m \text{ misses}) \\ &\leq \sum_{m=1}^M P(|\hat{R}(h_m; \mathcal{V}) - R(h_m)| > \varepsilon) \\ &\leq \sum_{m=1}^M 2e^{-2\varepsilon^2 n} \\ &= 2Me^{-2\varepsilon^2 n}, \end{aligned}$$

where the third line applies Equation 16 to each term.

```
n_many <- ceiling(log(2 * n_models / delta) / (2 * eps^2))
ratio_many <- round(n_many / n_one, 1)
```

Repeating the steps of Solution 3.4 with $2M$ in place of 2 gives $n \geq \ln(2M/\delta)/(2\varepsilon^2)$, which is 1659 emails, about 2.2 times as many as for one filter.

```
n_half <- ceiling(log(2 / delta) / (2 * (eps / 2)^2))
```

Both answers are sufficient sizes, not required ones, since they come from an upper bound. Two things still follow for Solution 3.1's three sets. The sufficient n is proportional to $1/\varepsilon^2$, so halving the allowed miss quadruples it: with $\varepsilon = 0.025$, Solution 3.4 needs 2952 emails rather than 738. And a [validation set](#) on which a fixed list of M models is compared needs the size from Solution 3.5 for that M , $\ln(2M/\delta)/(2\varepsilon^2)$, while a test set scored once on one model needs only the size from Solution 3.4. The union bound covers only a finite list of models; tuning a setting that can take infinitely many values needs the further analysis that Abu-Mostafa et al. [1] defers to its chapter 2.

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

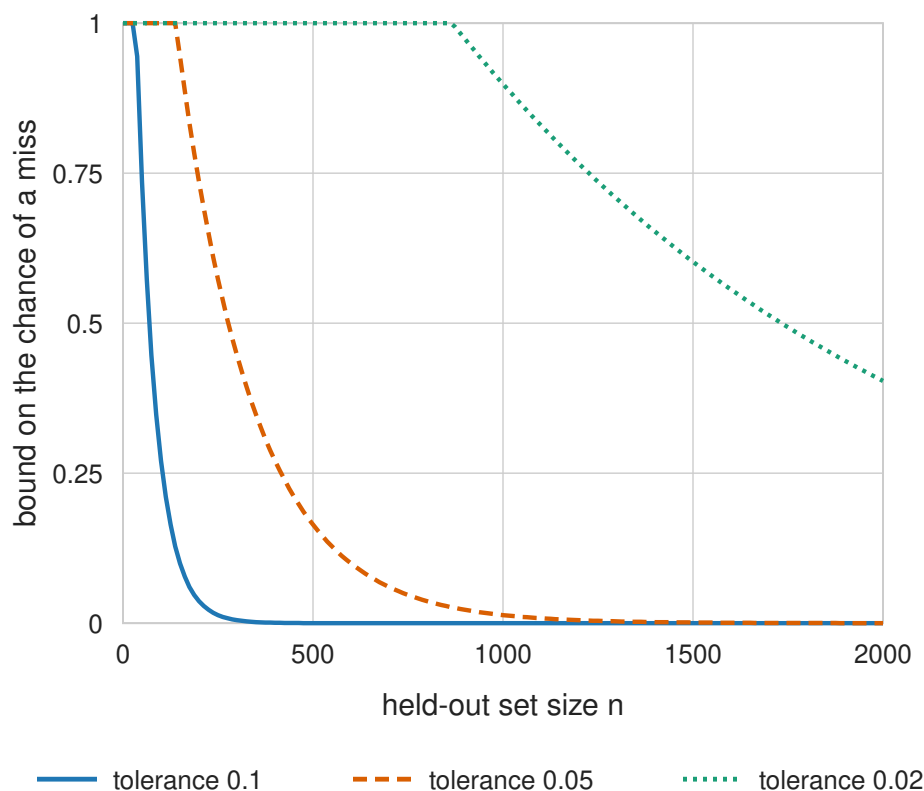


Figure 10: The bound of Equation 16 on the chance that a held-out estimate misses by more than the tolerance, as the held-out set grows, for three tolerances. The bound is capped at 1, since a probability cannot exceed it. A smaller tolerance needs a much larger held-out set before the bound says anything.

3.1 Cross-validation

Exercise 3.6 (Use every point twice). The three sets of Solution 3.1 are practical only when we have a lot of data. Setting aside a tenth of the data for validation and a tenth for testing leaves the model fitting on the remaining eighty percent, and on a small dataset that costs twice: the model is worse for having seen less, and the held-out estimate is noisy because it is based on so few points. Design a procedure in which every point is scored by a model that was not fitted to it, and every point also helps fit a model.

Solution

Solution 3.6. Divide the n points at random into k groups of roughly equal size. For each group in turn, fit the model on the other $k - 1$ groups and score it on the group left out. Average the k scores. This is k -fold cross-validation (Definition 3.8), and the groups are its folds.

Exercise 3.7 (Use every point twice (part 2)). Use the setup of Exercise 3.6. In your procedure, how many times is each point scored?

Solution

Solution 3.7. Each point belongs to exactly one group, so it is scored exactly once, by the one model that did not see it.

Exercise 3.8 (Use every point twice (part 3)). Use the setup of Exercise 3.6. In your procedure, how many times is each point trained on?

Solution

Solution 3.8. Each point is trained on by the other $k - 1$ models. The estimate^a therefore rests on all n points rather than on a held-out fraction of them.

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Exercise 3.9 (Use every point twice (part 4)). Use the setup of Exercise 3.6. What does the procedure cost?

Solution

Solution 3.9. The cost is computation: the model is fitted k times instead of once. Five and ten are the usual values of k . So use a train-validation-test split when there is plenty of data, and this procedure when there is not.

Definition 3.7 (Folds). Given a data set of n pairs, k **folds** $I_1, \dots, I_k$ are a random partition of the indices $\{1, \dots, n\}$ into k index sets of roughly equal size. See folds^a in the statistics notes.

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

Definition 3.8 (k -fold cross-validation). Given a data set of n pairs and a learning algorithm, **k -fold cross-validation** splits the pairs into k **folds** $I_1, \dots, I_k$ [16]. For each fold I_j , the algorithm is fit to the pairs outside that fold, which gives the model $\hat{y}_{-j}$.

Definition 3.9 (k -fold cross-validation estimate). The **k -fold cross-validation estimate** of the **risk** scores each pair by the model $\hat{y}_{-j}$ that did not see it (the model fit outside fold I_j in **k -fold cross-validation**), and averages those scores over all n pairs:

$$\hat{R}_{k\text{-fold}} \stackrel{\text{def}}{=} \frac{1}{n} \sum_{j=1}^k \sum_{i \in I_j} L(y_{(i)}, \hat{y}_{-j}(x_{(i)})). \quad (17)$$

Definition 3.10 (Leave-one-out cross-validation). **Leave-one-out cross-validation** is **k -fold cross-validation** with $k = n$ [16]: each pair is a fold of its own, and the model $\hat{y}_{-i}$ is fitted to the other $n - 1$ pairs. Its estimate of the **risk** is

$$\hat{R}_{\text{LOO}} \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^n L(y_{(i)}, \hat{y}_{-i}(x_{(i)})). \quad (18)$$

Example 3.6 (Leave-one-out on the five neighbourhoods). With $n = 5$ neighbourhoods, k -fold cross-validation with $k = n$ puts each neighbourhood in a fold of its own. This is leave-one-out cross-validation (Definition 3.10): five fits, each on four neighbourhoods and scored on the fifth. Fit the least-squares line to each training set of four, and score it by its squared error on the neighbourhood left out:

3.1.1 Python

```
import numpy as np

# The five neighbourhoods of the crime table, read once in the first example.
x_loo = crime["x"].to_numpy()
y_loo = crime["y"].to_numpy()

loo_py = []
for i_loo in range(5):
    # Train on every neighbourhood except i_loo, so the fit never sees the one it is scored on.
    keep_loo = np.arange(5) != i_loo
    m_loo, b_loo = np.polyfit(x_loo[keep_loo], y_loo[keep_loo], 1)
    # Score the line on the held-out neighbourhood alone.
    loo_py.append((m_loo * x_loo[i_loo] + b_loo - y_loo[i_loo]) ** 2)

loo_py = np.array(loo_py)
# A 95% interval for the mean of the five errors; 2.776 is the t quantile for 4 degrees of freedom,
# which replaces 1.96 because five errors are too few for the normal approximation.
loo_se_py = loo_py.std(ddof=1) / np.sqrt(5)
print(loo_py.round(4), loo_py.mean().round(4), (2.776 * loo_se_py).round(4))

[0.002  0.0026 0.0091 0.      0.015 ] 0.0057 0.0077
```

3.1.2 R

```
# The five neighbourhoods of the crime table, read once in the first example.
crime_loo <- crime
x_loo <- crime$x
y_loo <- crime$y

loo_r <- sapply(1:5, function(i) {
    # Train on every neighbourhood except i, so the fit never sees the one it is scored on.
    fit <- lm(y ~ x, data = crime_loo[-i, ])
    # Score the line on the held-out neighbourhood alone.
    (predict(fit, crime_loo[i, ]) - crime_loo$y[i])^2
})

# The same interval, with the t quantile for 4 degrees of freedom.
loo_se_r <- sd(loo_r) / sqrt(5)
loo_half_r <- qt(0.975, df = 4) * loo_se_r

print(round(c(loo_r, mean = mean(loo_r), half_width = loo_half_r), 4))
```

1	2	3	4	5	mean	half_width
0.0020	0.0026	0.0091	0.0000	0.0150	0.0057	0.0077

Figure 11 shows the five splits. Each panel lists the neighbourhoods in a small table, with the one held out shaded, and plots the same neighbourhoods, with that one in a different colour and shape. The line is fitted to the four blue circles alone.

```

import matplotlib.pyplot as plt

# Okabe-Ito colours, which stay distinct under the common colour-vision deficiencies;
# the circle against the diamond keeps the two groups apart in greyscale too.
train_col, held_col = "#0072B2", "#D55E00"
grid_loo = np.linspace(0.1, 0.55, 50)

fig, axes = plt.subplots(2, 5, figsize=(10, 4.2), layout="constrained",
                        height_ratios=[1.1, 1.4])
for i_loo in range(5):
    keep_loo = np.arange(5) != i_loo
    m_loo, b_loo = np.polyfit(x_loo[keep_loo], y_loo[keep_loo], 1)

    # Top row: the data as a table, with the held-out row shaded and labelled.
    top = axes[0, i_loo]
    top.axis("off");
    top.set_xlim(0, 1);
    top.set_ylim(0, 6.4);
    top.set_title(f"Split {i_loo + 1}", fontsize=10);
    top.text(0.05, 5.7, "x", fontsize=8, fontweight="bold");
    top.text(0.40, 5.7, "y", fontsize=8, fontweight="bold");
    for j in range(5):
        row_y = 4.7 - j
        if j == i_loo:
            top.fill_between([0, 1], row_y - 0.35, row_y + 0.85, color=held_col, alpha=0.25);
            top.text(0.05, row_y, f"{x_loo[j]:.2f}", fontsize=8);
            top.text(0.40, row_y, f"{y_loo[j]:.2f}", fontsize=8);
            top.text(0.72, row_y, "held out" if j == i_loo else "train", fontsize=7,
                    fontweight="bold" if j == i_loo else "normal");

    # Bottom row: the same neighbourhoods as a scatterplot, with the fitted line.
    bot = axes[1, i_loo]
    bot.plot(grid_loo, m_loo * grid_loo + b_loo, color="grey", linewidth=1.5);
    bot.scatter(x_loo[keep_loo], y_loo[keep_loo], color=train_col, marker="o", s=30,
               label="train");
    bot.scatter(x_loo[~keep_loo], y_loo[~keep_loo], color=held_col, marker="D", s=50,
               label="held out");
    bot.set_xlim(0.1, 0.55);
    bot.set_ylim(0, 0.6);
    bot.set_xlabel("per-capita income $x$", fontsize=8);
    if i_loo == 0:
        bot.set_ylabel("crime rate $y$", fontsize=8);
        bot.text(0.12, 0.03, f"error {loo_py[i_loo]:.3f}", fontsize=8);
        bot.tick_params(labelsize=7);

# One key for all five panels, in a row along the bottom of the figure.
fig.legend(*axes[1, 0].get_legend_handles_labels(), fontsize=8,
          loc="outside lower center", ncol=2);
plt.show()

```

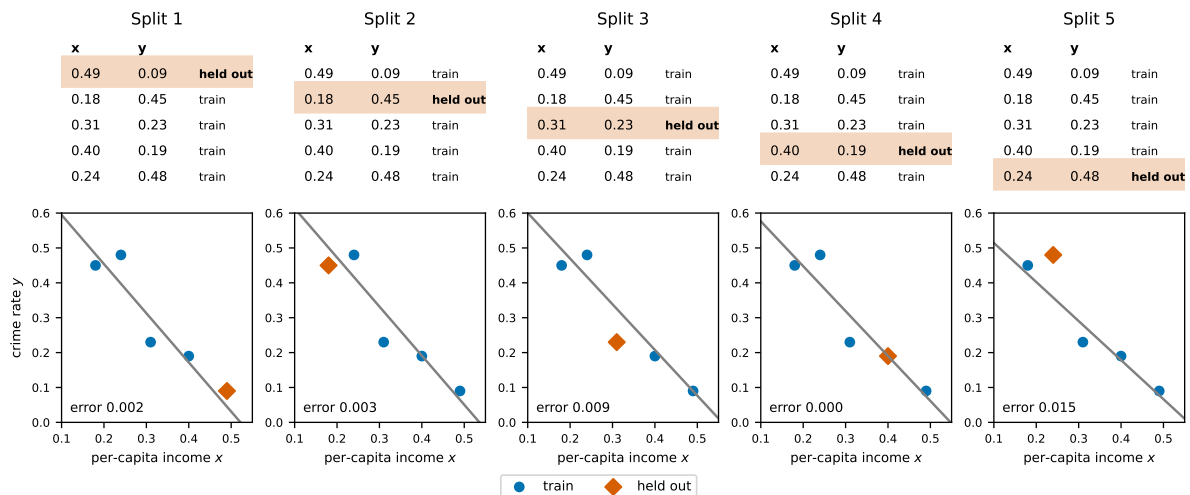


Figure 11: The five leave-one-out splits of the crime data of Table 1. In each panel the table lists the five neighbourhoods and shades the one held out; the scatterplot draws the held-out neighbourhood as an orange diamond, the four training neighbourhoods as blue circles, and the line fitted to the four, with the squared error on the held-out one.

The five squared errors are 0.002, 0.003, 0.009, 0.000, 0.015, and their mean, the leave-one-out estimate of the risk, is 0.006, with a 95% confidence interval of -0.002 to 0.013 . The interval is wide because five errors say little about their mean, and it is not a valid interval: the five errors share training points, so they are not independent, and squared errors are skewed, so a t interval fits them poorly. It reaches below zero, which a mean of squared errors cannot, so treat it as a rough guide to the uncertainty. Every neighbourhood is scored once, by a line that never saw it, and every neighbourhood helps fit four of the five lines. The largest error, 0.015, is in split 5, and the next largest, 0.009, is in split 3. Those two neighbourhoods have incomes 0.24 and 0.31, on either side of the steep drop in crime rate that a straight line fitted without one of them smooths over.

i Watch

Hutchinson covers train, dev and test sets and cross-validation, Generalization, part 2^a (11 min) [9]. The chapter Model Selection puts its cross-validation videos beside each method.

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#generalization_pt2

i Watch

Starmer explains cross-validation^a [17].

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

i Read

GeeksforGeeks explains cross-validation^a [18].

^a<https://www.geeksforgeeks.org/machine-learning/cross-validation-machine-learning/>

```
# The constants of this figure live here, so the widget, the static plot and the prose agree.
cv_n <- 40          # points drawn, and so held-out squared errors
cv_k <- 5           # folds of the static plot
cv_degree <- 3      # degree of the static plot
cv_fresh_n <- 2000  # fresh points the static plot scores
```

```

cv_sd <- 0.1          # standard deviation of the noise in the static plot

# The simulated world of the bias-variance figure: this true mean function plus
# Gaussian noise with standard deviation 0.1, on inputs in [0.05, 0.95].
truth <- function(x) 0.55 - 0.9 * x + 0.6 * sin(6 * x)
draw <- function(n) {
  x <- runif(n, 0.05, 0.95)
  data.frame(x = x, y = truth(x) + rnorm(n, sd = cv_sd))
}
set.seed(3100)
train <- draw(cv_n)
fresh <- draw(cv_fresh_n)
# Deal the points into k folds at random: shuffle, then fold = position mod k.
k <- cv_k
fold <- sample(rep_len(seq_len(k), nrow(train)))
# Least-squares polynomial of the given degree; degree 0 is the constant fit,
# which poly() does not accept.
fit_poly <- function(data, degree) {
  if (degree == 0) lm(y ~ 1, data = data) else lm(y ~ poly(x, degree), data = data)
}
# Held-out root mean squared error of a polynomial fit of the given degree on each fold,
# which is fit on the other k - 1 folds.
fold_rmse <- function(degree) {
  vapply(seq_len(k), function(f) {
    fit <- fit_poly(train[fold != f, ], degree)
    sqrt(mean((predict(fit, train[fold == f, ]) - train$y[fold == f])^2))
  }, numeric(1))
}
# Error of the fit to all the points on the fresh points.
fresh_rmse <- function(degree) {
  fit <- fit_poly(train, degree)
  sqrt(mean((predict(fit, fresh) - fresh$y)^2))
}
# The squared error of every point under the fit that left it out, so each point is scored once.
oof_sq <- function(degree) {
  sq <- numeric(nrow(train))
  for (f in seq_len(k)) {
    fit <- fit_poly(train[fold != f, ], degree)
    sq[fold == f] <- (predict(fit, train[fold == f, ]) - train$y[fold == f])^2
  }
  sq
}
# An approximate 95% confidence interval for the k-fold estimate: the percentile bootstrap
# interval for the mean of the held-out squared errors, with its endpoints square-rooted.
# It runs too narrow, because the errors are not independent (Bates, Hastie and Tibshirani 2024).
cv_ci <- function(degree) {
  sq <- oof_sq(degree)
  roots <- replicate(1000, sqrt(mean(sample(sq, replace = TRUE))))
  unname(quantile(roots, c(0.025, 0.975)))
}
degree <- cv_degree
scores <- fold_rmse(degree)
# The k-fold estimate is the root of the average of the fold MSEs.
kfold <- function(s) sqrt(mean(s^2))
degrees <- 0:8
cv_curve <- vapply(degrees, function(d) kfold(fold_rmse(d)), numeric(1))

```

```

cv_bands <- vapply(degrees, cv_ci, numeric(2))
fresh_curve <- vapply(degrees, fresh_rmse, numeric(1))
cv_now <- cv_ci(degree)
library(ggplot2)
orange <- "#ff7f0e"
est <- sprintf("%d-fold estimate", k)
left <- "Held-out fold"
right <- "Polynomial degree"
panels <- c(left, right)
panel_of <- function(p) factor(p, levels = panels)
# Left panel: each fold's RMSE as a dot, with the k-fold estimate
# and the fresh-point error as lines.
fold_dots <- data.frame(panel = panel_of(left), x = seq_len(k), y = scores)
levels_hline <- data.frame(
  panel = panel_of(left), series = c(est, "fresh points"),
  y = c(kfold(scores), fresh_rmse(degree))
)
# The approximate 95% confidence interval of the k-fold estimate, a pale band.
left_band <- data.frame(
  panel = panel_of(left), xmin = 0.5, xmax = k + 0.5, ymin = cv_now[1], ymax = cv_now[2]
)
# Right panel: the same two quantities at every degree,
# with the same band around the estimate.
curves <- data.frame(
  panel = panel_of(right), x = rep(degrees, 2),
  y = c(cv_curve, fresh_curve),
  series = rep(c(est, "fresh points"), each = length(degrees))
)
right_band <- data.frame(
  panel = panel_of(right), x = degrees, ymin = cv_bands[1, ], ymax = cv_bands[2, ]
)
chosen <- data.frame(panel = panel_of(right), x = degree)
series_colors <- setNames(c(orange, "black"), c(est, "fresh points"))
series_lines <- setNames(c("solid", "dashed"), c(est, "fresh points"))

```

```

ggplot() +
  geom_rect(
    data = left_band, aes(xmin = xmin, xmax = xmax, ymin = ymin, ymax = ymax),
    fill = orange, alpha = 0.2
  ) +
  geom_ribbon(
    data = right_band, aes(x = x, ymin = ymin, ymax = ymax),
    fill = orange, alpha = 0.2
  ) +
  geom_vline(data = chosen, aes(xintercept = x), color = "grey55") +
  geom_hline(
    data = levels_hline, aes(yintercept = y, color = series, linetype = series),
    linewidth = 0.8
  ) +
  geom_line(data = curves, aes(x = x, y = y, color = series), linewidth = 0.8) +
  geom_point(data = curves, aes(x = x, y = y, color = series)) +
  geom_point(data = fold_dots, aes(x = x, y = y), color = orange, size = 2.5) +
  facet_wrap(~panel, scales = "free_x", strip.position = "bottom") +
  scale_x_continuous(breaks = function(lim) seq(ceiling(lim[1]), floor(lim[2]))) +
  scale_y_log10() +
  scale_color_manual(values = series_colors) +
  scale_linetype_manual(values = series_lines) +
  labs(x = NULL, y = "RMSE (log scale)", color = NULL, linetype = NULL) +
  theme(strip.placement = "outside", legend.position = "bottom")

```

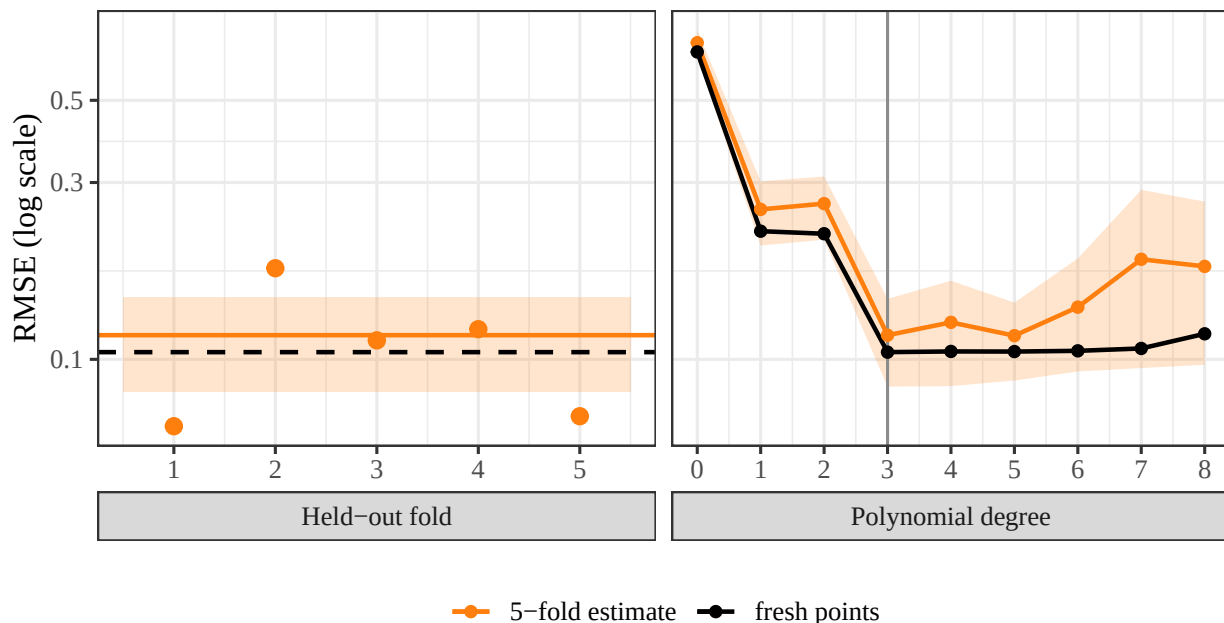


Figure 12: Static version of the interactive plot on the course website: 5-fold cross-validation of a degree-3 polynomial fit to 40 points from a simulated world, a smooth curve plus Gaussian noise with standard deviation 0.1. Left, the sRMSE on each held-out fold (dots), their root-mean-square, the 5-fold estimate (solid line) with its approximate 95% confidence interval (pale band), and the error on 2,000 fresh points (dashed line). Right, the same two quantities for degrees 0 to 8. The band is a bootstrap interval for the mean of the 40 held-out squared errors, square-rooted; it runs too narrow because the errors are not independent.

i Remark (Cross-validation replaces the validation set)

Remark 3.6 (Cross-validation replaces the validation set). Used to choose between models, cross-validation takes the place of the **validation** set, the one used again and again, and not of the final test set, which stays unused. Whichever scheme picked the model, the data used to pick it cannot also be the data used to report how good it is. If cross-validation picks the polynomial degree in Figure 12, for example, the cross-validation error of the winning degree is optimistic, because that degree won partly by fitting this sample's noise well; an honest estimate needs points that took no part in the choice.

4 Bias, variance and residual variance

Exercise 4.1 (What is random when a model predicts). Section 3 says the risk cannot be computed, only estimated. A different question about the same quantity is where it comes from: given a model that predicts badly, what are the distinguishable reasons it could be predicting badly? Start with the pieces. Choose an input value x , fit a model, and predict the output at x . For that process, list each thing that is random.

Solution

Solution 4.1. Three things.

- The output itself.
- The training set $\mathcal{D}_{\text{train}}$.
- The fitted model $\hat{y}_{\mathcal{D}_{\text{train}}}$.

Exercise 4.2 (What is random when a model predicts (part 2)). Use the setup of Exercise 4.1. Say what makes each one random.

Solution

Solution 4.2.

- *The output itself.* The world supplies a **true mean function** μ and noise: an input^a x has output $Y = \mu(x) + \varepsilon$, where ε is mean-zero noise independent^b of the training set (the **noise model**, Definition 4.3).
- The training set $\mathcal{D}_{\text{train}}$. It is a finite sample, and a different sample would have been just as possible.
- The fitted model. The learning algorithm turns $\mathcal{D}_{\text{train}}$ into a model, written $\hat{y}_{\mathcal{D}_{\text{train}}}$. Because $\mathcal{D}_{\text{train}}$ is random, so is $\hat{y}_{\mathcal{D}_{\text{train}}}$.

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

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

Definition 4.1 (True mean function). The **true mean function** μ is the mean function^a of the outcome Y given the input x , taken over the whole population^b and not over a training set: $\mu(x) = \mathbb{E}[Y \mid x]$.

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

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-population>

Example 4.1 (The true mean function of house prices). Let the outcome Y be the sale price of a house, and let the input x be its floor area. The value of the **true mean function** at a floor area x is the average sale price of all houses with that floor area. Two houses with the same floor area can sell for different prices, so the price of one house is not $\mu(x)$ exactly.

Definition 4.2 (Noise). **Noise** ε is the deviation^a of an observed outcome Y from the value $\mu(x)$ of the **true mean function** at the same input^b x :

$$\varepsilon \stackrel{\text{def}}{=} Y - \mu(x).$$

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-deviation>

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

Example 4.2 (The noise of one car). When the 392 cars of the Auto data [3] form the population^a, with input^b x equal to **cylinders** and outcome Y equal to **mpg**, the **true mean function** value at $x = 4$ cylinders is $\mu(4) = 29.28$, the population mean^c of **mpg** for the 199 four-cylinder cars. The 1970 Datsun PL510 has $Y = 27.0$ mpg, so the **noise** for that car is $\varepsilon = 27.0 - 29.28 = -2.28$ mpg. The 1971 Toyota Corolla 1200 has $Y = 31.0$ mpg, so the noise for that car is $\varepsilon = 31.0 - 29.28 = 1.72$ mpg.

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

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

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

Theorem 4.1 (The noise has population mean zero). *At any input^a x , the **noise** ε has conditional expectation^b (population mean^c at x) equal to 0:*

$$\mathbb{E}[\varepsilon \mid x] = 0. \tag{19}$$

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

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

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

i Proof

Proof. For the outcome Y and **noise** ε at input^a x , the **true mean function** value $\mu(x)$ does not vary once x is fixed, so its conditional expectation^b given x is $\mu(x)$.

$$\begin{aligned} \mathbb{E}[\varepsilon \mid x] &= \mathbb{E}[Y - \mu(x) \mid x] && \text{(definition of the noise)} \\ &= \mathbb{E}[Y \mid x] - \mu(x) && \text{(linearity of expectation)} \\ &= \mu(x) - \mu(x) && \text{(definition of the true mean function)} \\ &= 0 && \text{(a number minus itself is zero).} \end{aligned}$$

□

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

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

Example 4.3 (Noise in a count outcome). Suppose the outcome Y at input^a x is a count with a Poisson distribution^b whose population mean^c is the **true mean function** value $\mu(x)$. The **noise** $\varepsilon = Y - \mu(x)$ has population mean 0, by Theorem 4.1. The variance^d of ε at x is $\text{Var}(\varepsilon) = \mu(x)$, so the noise has larger variance at an input x where $\mu(x)$ is larger. The distribution^e of ε at x is also skewed: ε cannot fall below $-\mu(x)$ and has no upper bound. A gamma outcome behaves the same way: the noise is skewed, and the noise variance grows with the population mean $\mu(x)$.

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

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

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

^d<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

^e<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

i Remark (The noise can depend on the input)

Remark 4.1 (The noise can depend on the input). The distribution^a of the **noise** ε at an input^b x can change with x (Example 4.3). That distribution can be skewed, and the variance^c of ε need not be the same at every input. Only the population mean^d of ε at x must be zero (Theorem 4.1).

^a<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

^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/pds/expectation.html#def-expectation>

Definition 4.3 (Noise model). In the **noise model**, an input^a x has outcome Y

$$Y = \mu(x) + \varepsilon, \quad \mathbb{E}[\varepsilon] = 0, \quad \text{Var}(\varepsilon) = \sigma^2, \quad (20)$$

where $\mu(x)$ is the value of the **true mean function** at x , ε is the **noise** with population mean^b $\mathbb{E}[\varepsilon] = 0$, and σ^2 is the finite variance^c of ε at x .

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

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

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

i Remark (Independent noise draws across observations)

Remark 4.2 (Independent noise draws across observations). The **noise** ε is a fresh random draw each time an outcome Y is observed. Each outcome Y_i in the **training set** $\mathcal{D}_{\text{train}}$ at input^a x_i is $Y_i = \mu(x_i) + \varepsilon_i$, with **true mean function** value $\mu(x_i)$ and noise draw ε_i , and a new outcome Y at x has a new draw ε . Once the inputs^b are fixed, $\varepsilon_i = Y_i - \mu(x_i)$ and $\varepsilon = Y - \mu(x)$ shift the outcomes by the fixed values $\mu(x_i)$ and $\mu(x)$, so conditional independence^c of the outcomes given the inputs is equivalent to conditional independence of the noise draws given the inputs. Under that conditional independence, when the distribution^d of ε given x does not change with x , the noise draws are independent^e of the inputs and of one another. When the distribution of ε given x changes with x (Remark 4.1), a new observation drawn independently of $\mathcal{D}_{\text{train}}$ still has a new draw ε at the fixed input x that is independent of $\mathcal{D}_{\text{train}}$.

^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/independence.html#def-cind>

^d<https://morrison-lab.github.io/pds/random-variables.html#def-cdf>

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

i Remark (The noise model in the probability notes)

Remark 4.3 (The noise model in the probability notes). For a single observation of the outcome Y at input^a x with **true mean function** value $\mu(x)$ and finite variance^b σ^2 , Equation 20 is a consequence of Definition 4.2 ($Y = \mu(x) + \varepsilon$ for the **noise** ε) and Theorem 4.1 (conditional expectation^c $\mathbb{E}[\varepsilon | x] = 0$), not an additional assumption. Only the noise's population mean^d and variance matter here. The variance σ^2 is the variance of ε at the input x under discussion, and that variance can change with x (Remark 4.1). The probability notes call Equation 20 the mean-zero noise model^e.

The **noise model** fixes only two components of a distribution model^f for the outcome: the mean function^g is $\mu(x)$, and the variance is σ^2 . Any distribution model with that mean function and that variance satisfies the noise model, such as the normal distribution model^h with mean function $\mu(x)$ and variance σ^2 .

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

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

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

^d<https://morrison-lab.github.io/pds/expectation.html#def-expectation>

^e<https://morrison-lab.github.io/pds/variance-covariance.html#sec-noise-model>

^f<https://morrison-lab.github.io/sds/model-validation.html#def-distribution-model>

^g<https://morrison-lab.github.io/sds/model-validation.html#def-mean-function>

^h<https://morrison-lab.github.io/sds/model-validation.html#exm-distribution-model>

Definition 4.4 (Fitted model). The **fitted model** $\hat{y}_{\mathcal{D}_{\text{train}}}$ is the model that a learning algorithm returns when it is applied to the **training set** $\mathcal{D}_{\text{train}}$. It is a member of the mean-function model family^a, and it depends on $\mathcal{D}_{\text{train}}$, so a different training set gives a different fitted model.

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

Example 4.4 (A line fitted to the five neighbourhoods). Let the learning algorithm fit a straight line by least squares, and let $\mathcal{D}_{\text{train}}$ be the five neighbourhoods of Table 1. The line that results is the fitted model $\hat{y}_{\mathcal{D}_{\text{train}}}$. Five other neighbourhoods would give a different line, which is why the fitted model is random whenever the training set is.

Definition 4.5 (Prediction rule).

A **prediction rule** is a **fitted model** $\hat{y}_{\mathcal{D}_{\text{train}}}$ that is used to predict the outcome at a new covariate value x . Its prediction is $\hat{y}(x)$.

In words, the prediction rule takes a new covariate value and returns the fitted model's value there. See prediction rule^a in the statistics notes.

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

Example 4.5 (The fitted line as a prediction rule). Take the line $\hat{y}_{\mathcal{D}_{\text{train}}}$ **fitted** to the five neighbourhoods of Table 1 (Example 4.4). Using it to predict the crime rate of a sixth neighbourhood makes it a **prediction rule**. Using the same line only to describe the five neighbourhoods makes no prediction, so there it is a **fitted model** and not a prediction rule.

Definition 4.6 (Average fit). The **average fit** at x is the prediction $\hat{y}(x)$ of the **fitted model** at x , averaged over every training set $\mathcal{D}_{\text{train}}$ it might have been fitted to, where $\mathbb{E}_{\mathcal{D}_{\text{train}}}$ denotes that average over training sets. It is the population expectation

$$\mu_h(x) \stackrel{\text{def}}{=} \mathbb{E}_{\mathcal{D}_{\text{train}}} [\hat{y}(x)], \quad (21)$$

a fixed function of x that nothing estimates, so it carries no hat. It is written like the true mean function, with the model as the subscript.

It is not a model anyone ever holds; it is a reference point for measuring how far fits are from the truth and from each other. The statistics notes define it for the fitted model of any fitting procedure⁵.

Example 4.6 (The average fit of a line on cylinders). Treat the 392 cars of the **Auto** data [3] as the whole world of cars. A training set is 30 cars drawn at random from it, and the learning algorithm fits the line of **mpg** on **cylinders**. Draw 2000 training sets, fit a line to each, and record each line's intercept, slope and prediction at 4 cylinders. Both tabs draw with the **SyncRNG** package [19], a random number generator with the same algorithm in R and Python, from the same seed, so they pick the same cars:

4.0.1 Python

⁵<https://morrison-lab.github.io/sds/model-validation.html#def-average-fit>

```

import numpy as np
import pandas as pd
from SyncRNG import SyncRNG

auto_af = pd.read_csv("data/auto.csv").dropna(subset=["horsepower"])
cyl_af = auto_af["cylinders"].to_numpy(float)
mpg_af = auto_af["mpg"].to_numpy(float)

rng_af = SyncRNG(seed=1)
coefs_af = np.empty((2000, 2))
for s_af in range(2000):
    # One training set: the first 30 rows of a shuffle of all the rows (Python counts from 0), and the
    pick_af = rng_af.shuffle(list(range(len(mpg_af))))[:30]
    slope_af, intercept_af = np.polyfit(cyl_af[pick_af], mpg_af[pick_af], 1)
    coefs_af[s_af] = intercept_af, slope_af

preds_af = coefs_af[:, 0] + coefs_af[:, 1] * 4

# The average fit at 4 cylinders is the mean of those predictions.
# The Monte Carlo standard error of that mean is the spread of the predictions over sqrt(2000).
mc_se_af = preds_af.std(ddof=1) / np.sqrt(2000)
print(preds_af.mean().round(2), preds_af.min().round(2), preds_af.max().round(2), mc_se_af.round(3))
28.72 23.89 32.74 0.031

```

4.0.2 R

```

auto_af <- read.csv("data/auto.csv")
auto_af <- na.omit(auto_af[, c("mpg", "cylinders", "horsepower")])

# SyncRNG is a random number generator with the same algorithm in R and Python, so the draws match the
library(SyncRNG)
rng_af <- SyncRNG(seed = 1)
coefs_af <- t(replicate(2000, {
    # One training set: the first 30 rows of a shuffle of all the rows, and the line fitted to it.
    rows <- rng_af$shuffle(seq_len(nrow(auto_af)))[1:30]
    coef(lm(mpg ~ cylinders, data = auto_af[rows, ]))
}))

preds_af <- coefs_af[, 1] + coefs_af[, 2] * 4

# The average fit at 4 cylinders is the mean of those predictions.
# The Monte Carlo standard error of that mean is the spread of the predictions over sqrt(2000).
mc_se_af <- sd(preds_af) / sqrt(2000)
c(round(c(mean = mean(preds_af), min = min(preds_af), max = max(preds_af)), 2), mc_se = round(mc_se_af, 3))

    mean    min    max  mc_se
28.720 23.890 32.740  0.031

```

No single training set produces the average fit. The 2000 lines predict anywhere from 23.9 to 32.7 miles per gallon for a four-cylinder car, and their mean, 28.7, is the average fit $\mu_h(4)$ of Equation 21, with a 95% Monte Carlo confidence interval of 28.66 to 28.78 (1.96 Monte Carlo standard errors either side). The two tabs print the same numbers up to rounding.

Figure 13 plots 100 of the 2000 lines, their pointwise average, and the true mean function:

4.0.3 Python

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

grid_af = np.arange(3, 9)
# The pointwise average fit is the mean of the fitted lines at each cylinder count.
avg_af = (coefs_af[:, [0]] + coefs_af[:, [1]] * grid_af).mean(axis=0)
# The true mean function is the mean mpg of the cars with each cylinder count.
truth_af = auto_af.groupby("cylinders")["mpg"].mean()

fig, ax = plt.subplots(figsize=(6, 3.5), layout="constrained")
for intercept_af, slope_af in coefs_af[:100]:
    ax.plot(grid_af, intercept_af + slope_af * grid_af, color="grey", alpha=0.3, linewidth=0.8);
# One legend entry for the 100 grey lines, with a solid line to stand for them.
ax.plot([], [], color="grey", label="One fitted line (100 shown)");
ax.plot(grid_af, avg_af, color="steelblue", linewidth=2.5, label="Average fit");
ax.plot(truth_af.index, truth_af.values, color="darkorange", linewidth=2.5, marker="o", label="True mean function");
ax.set_xlabel("cylinders");
ax.set_ylabel("miles per gallon");
fig.legend(loc="outside lower center", ncol=3, frameon=False);
plt.show()
```

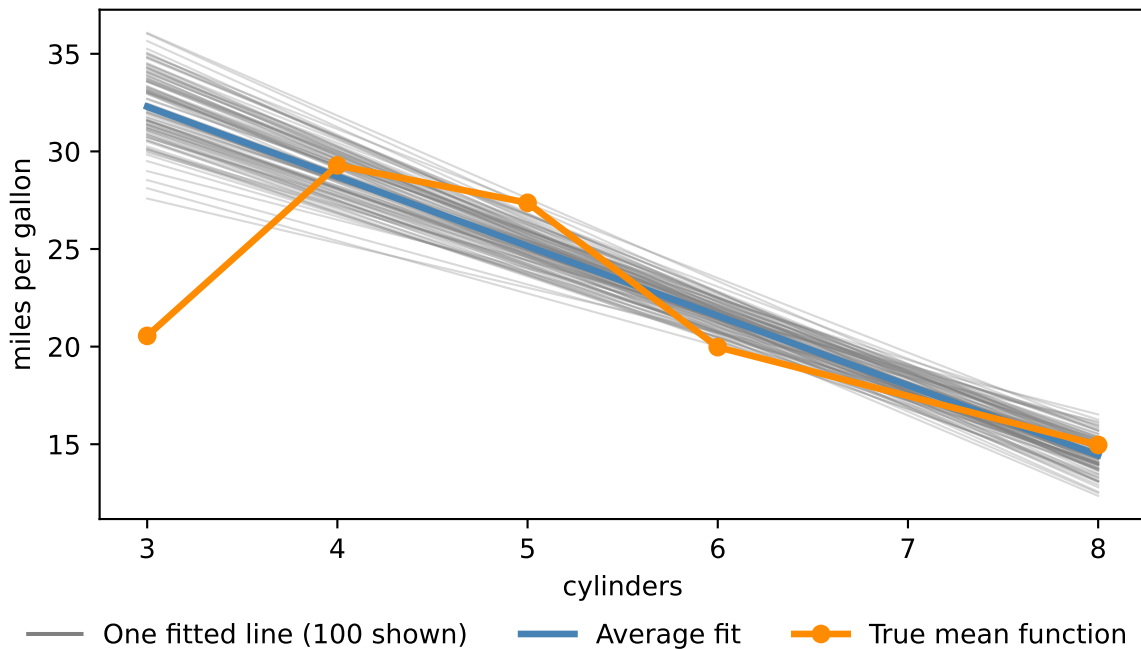


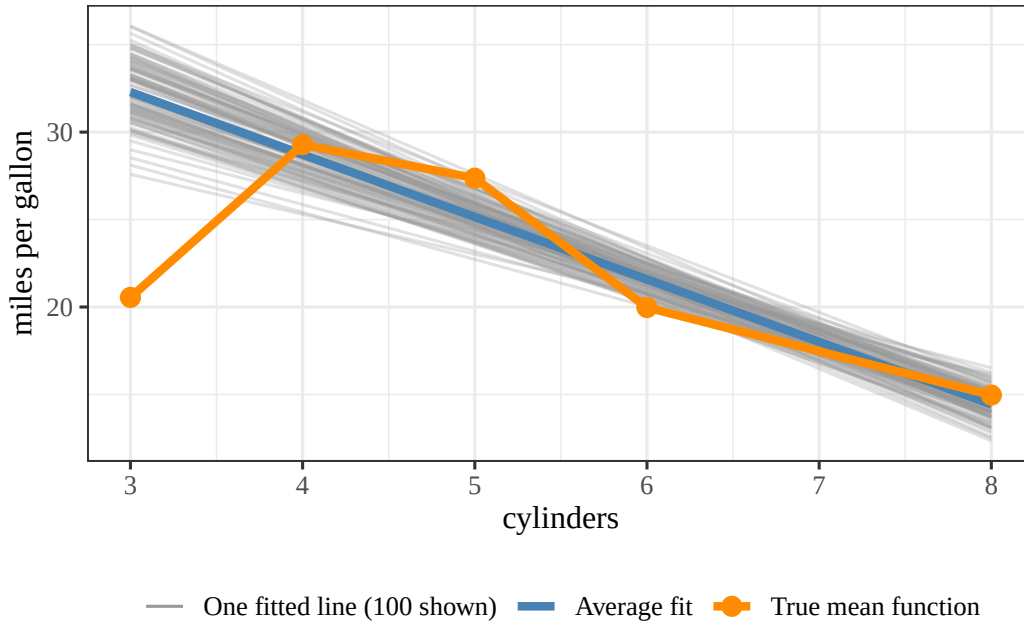
Figure 13: The fitted lines of 100 of the 2000 training sets (grey), their pointwise average fit (blue), and the true mean function (orange), for mpg against cylinders.

4.0.4 R

```
library(ggplot2)

grid_af <- 3:8
# One row per fitted line and cylinder count; the pointwise average fit is the mean over lines.
lines_af <- data.frame(
  line = rep(seq_len(nrow(coefs_af)), each = length(grid_af)),
  cylinders = grid_af,
  fit = as.vector(t(coefs_af[, 1] + outer(coefs_af[, 2], grid_af)))
)
avg_af <- aggregate(fit ~ cylinders, data = lines_af, FUN = mean)
# The true mean function is the mean mpg of the cars with each cylinder count.
truth_af <- aggregate(mpg ~ cylinders, data = auto_af, FUN = mean)

ggplot() +
  geom_line(data = subset(lines_af, line <= 100), aes(cylinders, fit, group = line, colour = "One fitted line (100 shown)"), linewidth = 1) +
  geom_line(data = avg_af, aes(cylinders, fit, colour = "Average fit"), linewidth = 1.5) +
  geom_line(data = truth_af, aes(cylinders, mpg, colour = "True mean function"), linewidth = 1.5) +
  geom_point(data = truth_af, aes(cylinders, mpg, colour = "True mean function"), size = 3) +
  scale_colour_manual(
    values = c("One fitted line (100 shown)" = "grey60", "Average fit" = "steelblue", "True mean function" = "orange"),
    breaks = c("One fitted line (100 shown)", "Average fit", "True mean function")
  ) +
  guides(colour = guide_legend(override.aes = list(alpha = 1))) +
  labs(x = "cylinders", y = "miles per gallon", colour = NULL) +
  theme(legend.position = "bottom")
```



Definition 4.7 (Bias of a prediction). The **bias** of a prediction at x is the expected^a prediction error^b of the prediction at x . The outcome in the error is a new outcome Y at x in the **noise model**, independent^c of the **training set** $\mathcal{D}_{\text{train}}$, so the error is $\varepsilon_{\text{pred}} = (\hat{y} - Y)$. The expectation^d averages over **training sets** $\mathcal{D}_{\text{train}}$ and over the **noise** in Y :

$$\text{Bias}(\hat{y}) \stackrel{\text{def}}{=} \mathbb{E}_{\mathcal{D}_{\text{train}}, Y} [\varepsilon_{\text{pred}}]. \quad (22)$$

^a<https://morrison-lab.github.io/pds/expectation.html#def-expectation>
^b<https://morrison-lab.github.io/sds/prediction.html#def-prediction-error>
^c<https://morrison-lab.github.io/pds/independence.html#def-indpt>
^d<https://morrison-lab.github.io/pds/expectation.html#def-expectation>

i Remark (The bias is signed, not squared)

Remark 4.4 (The bias is signed, not squared). The **bias** is an expected^a prediction error^b, so it is signed, not squared. It is not the expected squared prediction error^c (the **risk**). The risk also adds the **variance** of the prediction and the **noise** variance.

^a<https://morrison-lab.github.io/pds/expectation.html#def-expectation>
^b<https://morrison-lab.github.io/sds/prediction.html#def-prediction-error>
^c<https://morrison-lab.github.io/sds/model-validation.html#def-epe>

Definition 4.8 (Variance of a prediction). The **variance** of a prediction at x is the expected square of the deviation^a of the prediction at x from its population mean over **training sets** $\mathcal{D}_{\text{train}}$, which is the **average fit** $\mu_h(x)$:

$$\text{Var}(\hat{y}) = \mathbb{E}_{\mathcal{D}_{\text{train}}} \left[(\mathfrak{d}_{\hat{y}})^2 \right], \quad \text{where } \mathfrak{d}_{\hat{y}} = \hat{y} - \mathbb{E}_{\mathcal{D}_{\text{train}}} [\hat{y}]. \quad (23)$$

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-deviation>

Example 4.7 (The bias and variance of the cylinders line at four cylinders). Continue Example 4.6 with the same 2000 fitted lines. At $x = 4$ cylinders, the value $\mu(4)$ of the **true mean function** in the **noise model** is the mean mpg of the four-cylinder cars. Compute the bias (Equation 22) and the variance (Equation 23) of the line:

4.0.5 Python

```
import numpy as np

# mu(4) is the mean mpg of the four-cylinder cars.
f4_bv = mpg_af[cyl_af == 4].mean()

# Bias: the average fit minus the truth.
bias_bv = preds_af.mean() - f4_bv
# Variance: how far one fitted line sits from the average fit, on average.
var_bv = np.mean((preds_af - preds_af.mean()) ** 2)
print(np.round([bias_bv, var_bv], 2))

[-0.56  1.86]
```

4.0.6 R

```
# mu(4) is the mean mpg of the four-cylinder cars.
f4_bv <- mean(auto_af$mpg[auto_af$cylinders == 4])

# Bias: the average fit minus the truth.
bias_bv <- mean(preds_af) - f4_bv
# Variance: how far one fitted line sits from the average fit, on average.
var_bv <- mean((preds_af - mean(preds_af))^2)
round(c(bias = bias_bv, variance = var_bv), 2)

bias variance
```

-0.56 1.86

The average fit is 0.56 miles per gallon below the truth, so the bias is -0.56, and one fitted line differs from the average fit by 1.86 squared miles per gallon on average. The two tabs print the same numbers up to rounding.

Figure 14 shows the 2000 predictions with their average and the true mean function at four cylinders:

4.0.7 Python

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6, 3.5), layout="constrained")
ax.hist(preds_af, bins=40, color="lightgrey", edgecolor="white");
ax.axvline(preds_af.mean(), color="steelblue", linewidth=2);
ax.axvline(f4_bv, color="darkorange", linewidth=2);
ax.set_xlabel("predicted miles per gallon at 4 cylinders");
ax.set_ylabel("training sets");
plt.show()
```

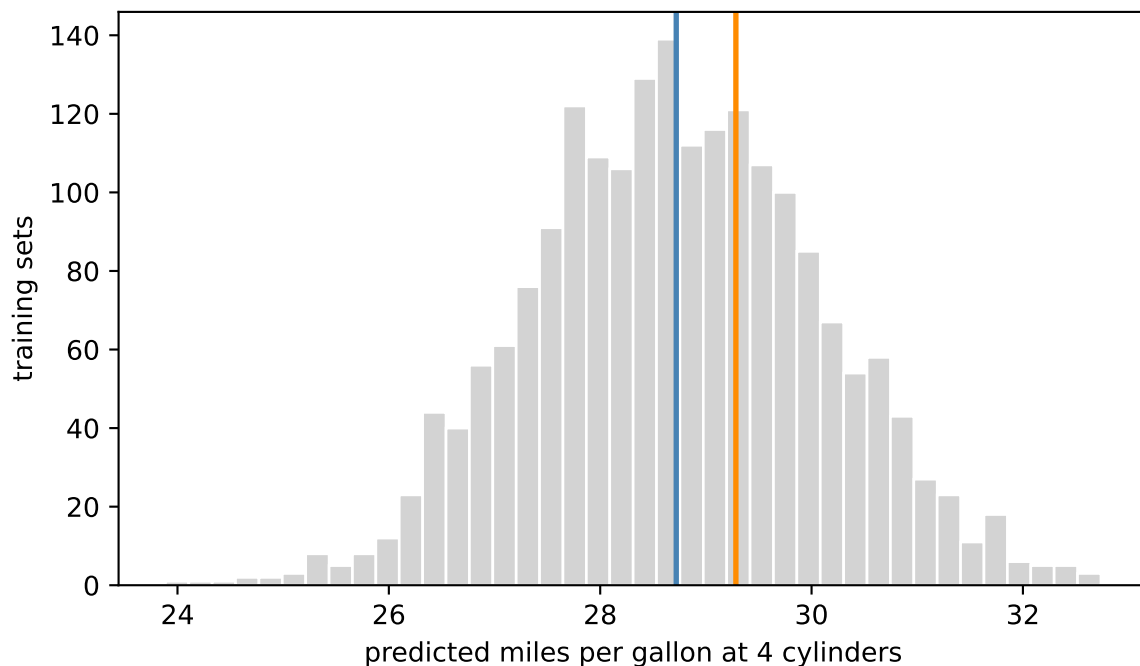
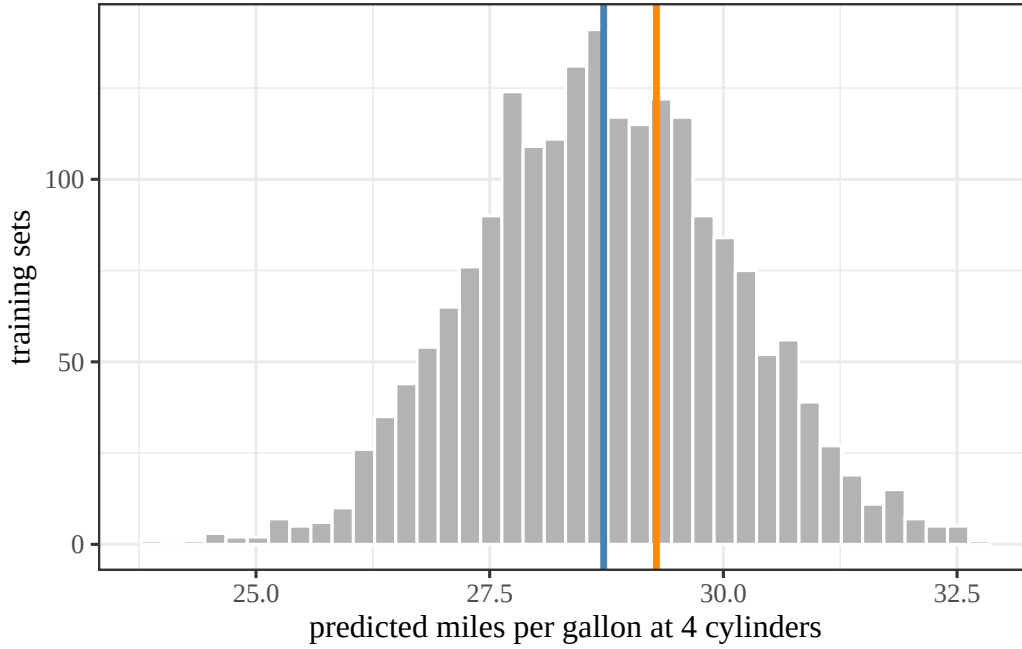


Figure 14: The 2000 predictions at four cylinders, with their average (the average fit, blue) and the true mean function at four cylinders (orange). The gap between the lines is the bias; the spread of the histogram is the variance.

4.0.8 R

```
library(ggplot2)

ggplot(data.frame(prediction = preds_af), aes(prediction)) +
  geom_histogram(bins = 40, fill = "grey70", colour = "white") +
  geom_vline(xintercept = mean(preds_af), colour = "steelblue", linewidth = 1.2) +
  geom_vline(xintercept = f4_bv, colour = "darkorange", linewidth = 1.2) +
  labs(x = "predicted miles per gallon at 4 cylinders", y = "training sets")
```



i Remark (The variance of a fitted model's prediction is the variance of a random variable)

Remark 4.5 (The variance of a fitted model's prediction is the variance of a random variable). The prediction of the **fitted model** at x is a random variable, because it depends on the **training set** $\mathcal{D}_{\text{train}}$. The **variance** of that prediction is therefore the variance^a of that random variable.

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

Theorem 4.2 (The bias as the average fit minus the true mean function). *The **bias** of the prediction at x equals the **average fit** minus the **true mean function** value:*

$$\text{Bias}(\hat{y}) = \mu_h(x) - \mu(x).$$

i Proof

Proof. The new outcome Y is independent^a of $\mathcal{D}_{\text{train}}$, and its mean^b is the **true mean function** value $\mu(x)$ because the **noise** has mean zero by Equation 20.

$$\begin{aligned} \text{Bias}(\hat{y}) &= \mathbb{E}_{\mathcal{D}_{\text{train}}, Y}[\varepsilon_{\text{pred}}] && \text{(definition of bias)} \\ &= \mathbb{E}_{\mathcal{D}_{\text{train}}, Y}[(\hat{y} - Y)] && \text{(definition of prediction error)} \\ &= \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}] - \mathbb{E}_Y[Y] && \text{(linearity of expectation; } Y \text{ is independent of } \mathcal{D}_{\text{train}}) \\ &= \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}] - \mu(x) && \text{(the noise has mean zero)} \\ &= \mu_h(x) - \mu(x) && \text{(definition of the average fit).} \end{aligned}$$

□

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

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

Example 4.8 (The bias as the average fit minus the true mean function at four cylinders). Continue Example 4.7 with the same 2000 fitted lines. The **bias** is the expected^a prediction error^b against a new outcome (Equation 22). The new outcome has mean^c $\mu(4)$, so the bias equals the average estimation error^d of the lines. Compute that average, and compare it with the **average fit** minus the truth:

4.0.9 Python

```
# The bias from its definition: the mean error over the lines, taking the new outcome at its mean.
bias_def_baf = np.mean(preds_af - f4_bv)
# The bias from the theorem: the average fit minus the truth.
bias_thm_baf = preds_af.mean() - f4_bv
print(np.round([bias_def_baf, bias_thm_baf], 2))

[-0.56 -0.56]
```

4.0.10 R

```
# The bias from its definition: the mean error over the lines, taking the new outcome at its mean.
bias_def_baf <- mean(preds_af - f4_bv)
# The bias from the theorem: the average fit minus the truth.
bias_thm_baf <- mean(preds_af) - f4_bv
round(c(definition = bias_def_baf, theorem = bias_thm_baf), 2)
```

definition	theorem
-0.56	-0.56

Both routes give -0.56 miles per gallon, as Theorem 4.2 says.

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

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

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

^d<https://morrison-lab.github.io/sds/estimation.html#def-estimation-error>

Theorem 4.3 (The variance equals the variance of the estimation error). *The **variance** of the prediction of the fitted model at x equals the variance of the estimation error^a of that prediction:*

$$\text{Var}(\hat{y}) = \text{Var}(\varepsilon(\hat{y})).$$

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

i Proof

Proof. The mean^a of the estimation error^b is the **average fit** minus the **true mean function** value, which is the **bias** by Theorem 4.2. Subtracting the bias from the error leaves the fit minus the average fit.

$$\begin{aligned} \varepsilon(\hat{y}) - \text{Bias}(\hat{y}) &= (\hat{y} - \mu(x)) - (\mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}] - \mu(x)) && \text{(definition of estimation error; @thm-bias-average-fit)} \\ &= \hat{y} - \mu(x) - \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}] + \mu(x) && \text{(removing parentheses)} \\ &= \hat{y} - \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}] + (\mu(x) - \mu(x)) && \text{(reordering)} \\ &= \hat{y} - \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}] + 0 && \text{(a number minus itself is zero)} \\ &= \hat{y} - \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}] && \text{(adding zero changes nothing).} \end{aligned}$$

The error's variance is its expected squared deviation from its own mean, and its mean is the bias. So the variance of the error is the expected square of the difference above. By Equation 23, the variance of the fit is the expected square of the same difference. So the two variances are equal. $\square$

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

^b<https://morrison-lab.github.io/sds/estimation.html#def-estimation-error>

Example 4.9 (The variance as the spread around the average fit at four cylinders). Continue Example 4.7 with the same 2000 fitted lines. Compute the **variance** of the lines (Equation 23), and compare it with the variance of their estimation error^a:

4.0.11 Python

```
# The variance from the definition: the spread of the lines around the average fit.
var_def_vaf = np.mean((preds_af - preds_af.mean()) ** 2)
# The variance from the theorem: the spread of the estimation error around its mean, the bias.
err_vaf = preds_af - f4_bv
var_thm_vaf = np.mean((err_vaf - err_vaf.mean()) ** 2)
print(np.round([var_def_vaf, var_thm_vaf], 2))

[1.86 1.86]
```

4.0.12 R

```
# The variance from the definition: the spread of the lines around the average fit.
var_def_vaf <- mean((preds_af - mean(preds_af))^2)
# The variance from the theorem: the spread of the estimation error around its mean, the bias.
err_vaf <- preds_af - f4_bv
var_thm_vaf <- mean((err_vaf - mean(err_vaf))^2)
round(c(definition = var_def_vaf, theorem = var_thm_vaf), 2)

definition    theorem
      1.86      1.86
Both routes give 1.86 squared miles per gallon, as Theorem 4.3 says.
```

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

Five **training sets**, each of five points, each get four **fitted models**, one for each polynomial⁶ degree⁷ from 1 to 4. Both tabs draw the training sets with the SyncRNG package [19] from the same seed, so they draw the same data, and add normal **noise** (standard deviation⁸ 0.13) to the **true mean function**:

4.0.13 Python

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

def truth_bv(x):
    return 0.55 - 0.9 * x + 0.6 * np.sin(6 * x)

x_bv = np.array([0.1, 0.3, 0.5, 0.7, 0.9])
grid_bv = np.linspace(0.05, 0.95, 100)

rng_bv = SyncRNG(seed=37)
# The first draws from a small seed are poorly mixed, so skip the first 100.
# With seed 37 the spread rises with the degree, and the line and quadratic have about the same bias; th
for _ in range(100):
    rng_bv.rand()
```

0.01295785838738086

⁶<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial>

⁷<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

⁸<https://morrison-lab.github.io/pds/variance-covariance.html#def-sd>

0.002371650189161304
0.2968793136533354
0.07564082206226896
3.2246112823486376e-05
0.0008095083758234991
0.13812292250804625
0.5122318337671466
0.020569282583892377
0.5711903637275109
0.4307010236661888
0.3126481119543319
0.8247087667696191
0.25056422734633127
0.01982539240270856
0.656618150183932
0.0035579456016421374
0.2656588805839423
0.005491338204592475
0.148991678375751
0.753784512402491
0.10312855453230457
0.5371983668301263
0.1669946967158469
0.4377464081626392
0.38558150036260547
0.6718784791883091
0.51529568550177
0.14854685007594548
0.754973555216567
0.5450272767338911
0.43828323250636525
0.2673391164280478
0.12641562381759305
0.15407474641688193
0.04990393551997848
0.010626258095726388
0.2968825057614599
0.7246106821112346
0.10357670765370147
0.018643392249941854
0.0019647027365863354
0.7284108900930744
0.1998202099930498
0.03204639488831167
0.5046537269372501
0.14734604465775214
0.7997305849567068
0.0631144754588605
0.2876713678706442
0.12506534811109324
0.7309028850868355
0.8767352423165007
0.919934593373911
0.504620853345842
0.016636913875117922
0.751537843141706

```

0.12165764183737357
0.12537119863554855
0.02983465418219571
0.4856160543859013
0.014024667674675606
0.40144105535000624
0.3228049515746539
0.32815250800922563
0.5642997468821713
0.8135610721074057
0.407059870427475
0.03990194620564586
0.37533342209644677
0.5010020500049003
0.36724307900294717
0.014961027540266536
0.09627288859337584
0.9435006731655463
0.5167076007928707
0.5061757061630495
0.3787356223911053
0.3909605515655136
0.7703797249123466
0.742255591787399
0.8178380983881665
0.17199150472879438
0.9746278689708576
0.8238326623104526
0.8628171815071268
0.020832758862525256
0.1961464814376089
0.6946366745978605
0.2967190681956713
0.07039282610639941
0.2660218474920843
0.6103706422727565
0.8512498410418643
0.5316741552669563
0.022498969919979607
0.5824551463592806
0.008310291450470698
0.3066714736633008
0.881415901705624

```

```

def std_normal_bv():
    # One standard normal draw from two uniform draws (the Box-Muller transform).
    u1, u2 = rng_bv.rand(), rng_bv.rand()
    return np.sqrt(-2 * np.log(u1)) * np.cos(2 * np.pi * u2)

# Five training sets of five points.
samples_bv = [truth_bv(x_bv) + 0.13 * np.array([std_normal_bv() for _ in x_bv]) for _ in range(5)]
titles_bv = ["A line: high bias, low variance", "A quadratic", "A cubic", "A quartic: low bias, high var

fig, axes = plt.subplots(2, 2, figsize=(6.5, 5), sharex=True, sharey=True, layout="constrained")
for degree, ax, title in zip(range(1, 5), axes.flat, titles_bv):
    # One fit per training set, evaluated on the grid.

```

```

fits = np.array([np.polyval(np.polyfit(x_bv, y, degree), grid_bv) for y in samples_bv])
for fit in fits:
    ax.plot(grid_bv, fit, color="#0072B2", linewidth=1);
ax.plot(grid_bv, truth_bv(grid_bv), color="black", linestyle="--", linewidth=1.2);
ax.set_title(title);
for ax in axes[1]:
    ax.set_xlabel("Feature, x");
for ax in axes[:, 0]:
    ax.set_ylabel("Response, y");
fig.legend(
    handles=[
        plt.Line2D([], [], color="#0072B2", label="One fit per training set"),
        plt.Line2D([], [], color="black", linestyle="--", label="The true mean function"),
    ],
    loc="outside lower center",
    ncol=2,
    frameon=False,
);
plt.show()

```

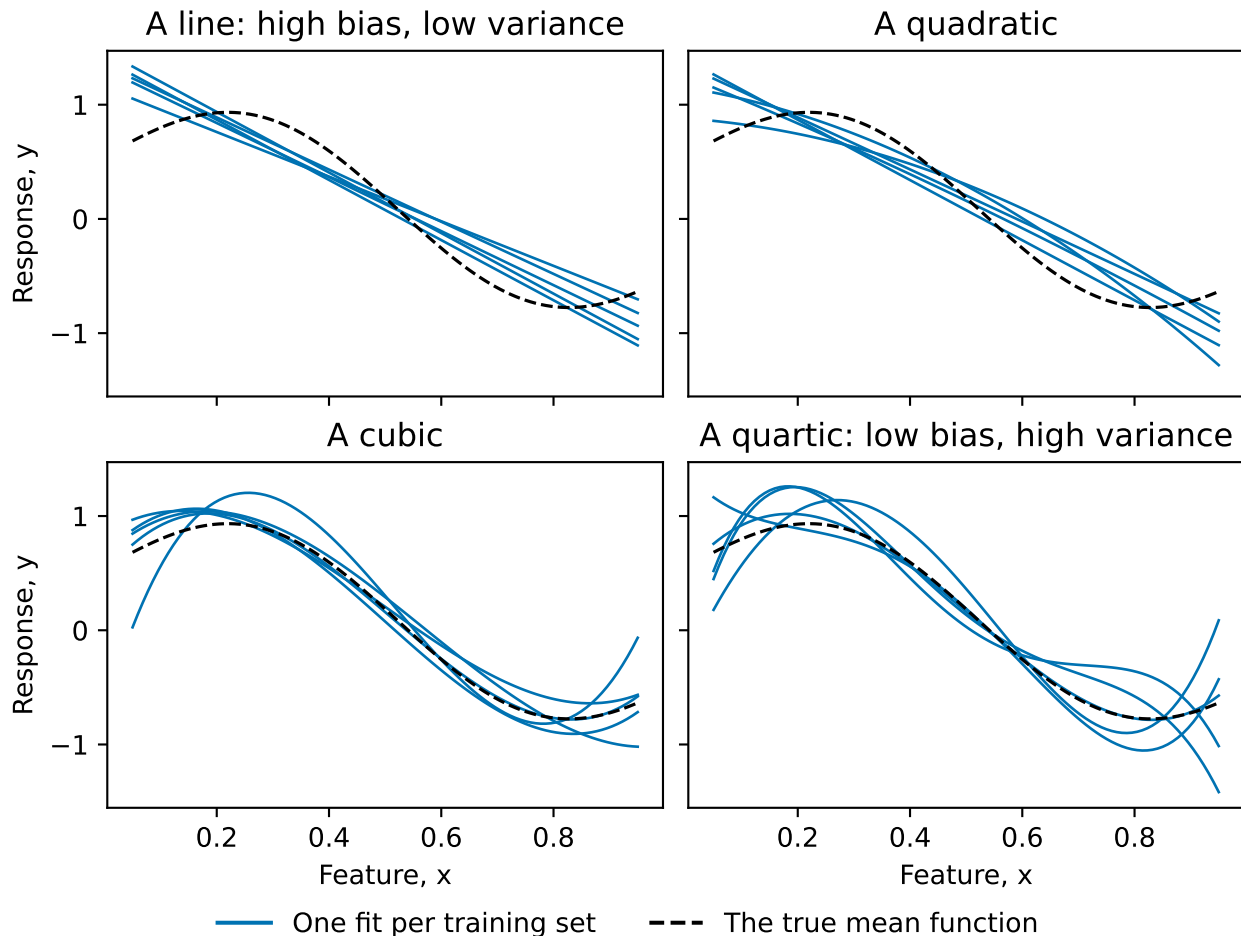


Figure 15: The same five training sets, fitted four times: by a line, a quadratic, a cubic and a quartic. The line and the quadratic lie close together and miss the true mean function (dashed) in the same places. The cubic fits follow it closely except near the ends of the range. The quartic fits, each passing through all five points of its training set, spread widely. Each curve is one training set's fit, so the spread among the five is the variability itself.

4.0.14 R

```
library(ggplot2)
library(SyncRNG)

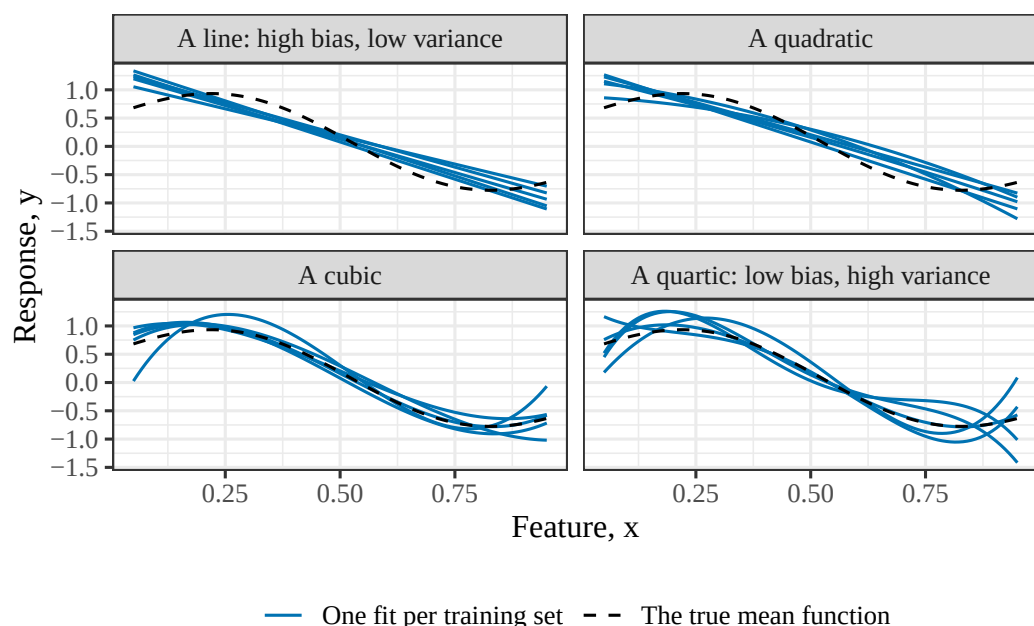
truth_bv <- function(x) 0.55 - 0.9 * x + 0.6 * sin(6 * x)
x_bv <- c(0.1, 0.3, 0.5, 0.7, 0.9)
grid_bv <- seq(0.05, 0.95, length.out = 100)

rng_bv <- SyncRNG(seed = 37)
# The first draws from a small seed are poorly mixed, so skip the first 100.
# With seed 37 the spread rises with the degree, and the line and quadratic have about the same bias; th
for (i in 1:100) rng_bv$rand()
# One standard normal draw from two uniform draws (the Box-Muller transform).
std_normal_bv <- function() {
  u <- c(rng_bv$rand(), rng_bv$rand())
  sqrt(-2 * log(u[1])) * cos(2 * pi * u[2])
}

# Five training sets of five points, each fitted by a polynomial of degree 1 to 4.
fits_bv <- do.call(rbind, lapply(1:5, function(set) {
  y <- truth_bv(x_bv) + 0.13 * replicate(5, std_normal_bv())
  do.call(rbind, lapply(1:4, function(degree) {
    fit <- lm(y ~ poly(x_bv, degree, raw = TRUE))
    data.frame(
      set = set,
      degree = factor(degree, levels = 1:4, labels = c("A line: high bias, low variance", "A quadratic",
    )
  })))
}))
truth_df_bv <- data.frame(x = grid_bv, fit = truth_bv(grid_bv))

# The spread of the five fits: their standard deviation at each x, averaged over the grid.
spread_bv <- sapply(split(fits_bv, fits_bv$degree), function(d) mean(tapply(d$fit, d$x, sd)))

ggplot(fits_bv, aes(x, fit)) +
  geom_line(aes(group = set, colour = "One fit per training set", linetype = "One fit per training set"),
  geom_line(data = truth_df_bv, aes(colour = "The true mean function", linetype = "The true mean function"),
  facet_wrap(~degree, nrow = 2) +
  scale_colour_manual(values = c("One fit per training set" = "#0072B2", "The true mean function" = "black"),
  scale_linetype_manual(values = c("One fit per training set" = "solid", "The true mean function" = "dashed"),
  labs(x = "Feature, x", y = "Response, y", colour = NULL, linetype = NULL)
```



Five points and five [parameters](#): each quartic passes through all five points of its own [training set](#) and swings between them, which is the variability the bottom-right panel shows. The line has two parameters and barely moves, and the quadratic and the cubic sit between the two. The spread of the five fits, averaged over the grid, is 0.08 for the line, 0.10 for the quadratic, 0.13 for the cubic and 0.18 for the quartic, so here it rises with every step in the degree⁹.

Theorem 4.4 (Separating the noise from the expected error). *Choose an input value x , and let the output variable $Y = \mu(x) + \varepsilon$ as in Equation 20, where ε has mean 0 and variance σ^2 and is independent of the training set $\mathcal{D}_{\text{train}}$. Then the expected^a squared error at x is the [noise](#) variance σ^2 plus the model's own expected squared error:*

$$\mathbb{E}_{\mathcal{D}_{\text{train}}, \varepsilon}[(Y - \hat{y})^2] = \sigma^2 + \mathbb{E}_{\mathcal{D}_{\text{train}}}[(\mu(x) - \hat{y})^2]. \quad (24)$$

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

i Proof

Proof. This is the statistics notes' theorem that the expected squared prediction error is the noise variance plus the expected squared estimation error^a, applied to the prediction $\hat{y}$ of the mean $\mu(x)$. $\square$

^a<https://morrison-lab.github.io/sds/model-validation.html#thm-epe-split-noise>

Example 4.10 (Separating the noise at four cylinders). Continue Example 4.6 with the same 2000 fitted lines, at $x = 4$ cylinders, where $\mu(4)$ is the population mean mpg of the four-cylinder cars and the noise is how far a single car sits from that population mean. Compute the left side of Equation 24 directly, and then the noise variance plus the model's own expected squared error:

4.0.15 Python

⁹<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial-degree>

```

y4_sn = mpg_af[cyl_af == 4]
f4_sn = y4_sn.mean()

# The left side: average the squared error over every line and every car.
left_sn = np.mean((y4_sn[None, :] - preds_af[:, None]) ** 2)
noise_sn = np.mean((y4_sn - f4_sn) ** 2)
model_sn = np.mean((f4_sn - preds_af) ** 2)
print(np.round([left_sn, noise_sn, model_sn, noise_sn + model_sn], 2))

[34.18 31.99  2.18 34.18]

```

4.0.16 R

```

y4_sn <- auto_af$mpg[auto_af$cylinders == 4]
f4_sn <- mean(y4_sn)

left_sn <- mean(outer(preds_af, y4_sn, function(h, y) (y - h)^2))
noise_sn <- mean((y4_sn - f4_sn)^2)
model_sn <- mean((f4_sn - preds_af)^2)
round(c(left = left_sn, noise = noise_sn, model = model_sn,
        sum = noise_sn + model_sn), 2)

  left noise model   sum
34.18 31.99  2.18 34.18

```

The expected squared error is 34.18, and the noise variance 31.99 plus the model's own expected squared error 2.18 add to 34.18, as Theorem 4.4 says.

Theorem 4.5 (Separating the bias from the model's own error). *Choose an input value x , let $\mu(x)$ be the true mean function of the outcome variable given the input value x in Equation 20, and let $\mu_h(x)$ be the average fit of Equation 21. Then the expected squared distance between the truth and the fitted model is the squared bias plus the variance:*

$$\mathbb{E}_{\mathcal{D}_{train}}[(\mu(x) - \hat{y})^2] = (\mu(x) - \mathbb{E}_{\mathcal{D}_{train}}[\hat{y}(x)])^2 + \mathbb{E}_{\mathcal{D}_{train}}[(\hat{y} - \mathbb{E}_{\mathcal{D}_{train}}[\hat{y}(x)])^2]. \quad (25)$$

i Proof

Proof. The statistics notes work this as an exercise, separating the bias^a. □

^a<https://morrison-lab.github.io/sds/model-validation.html#sol-epe-split-bias>

Example 4.11 (Separating the bias at four cylinders). Continue Example 4.6 with the same 2000 fitted lines, at $x = 4$ cylinders, where $\mu(4)$ is the population mean mpg of the four-cylinder cars. Compute the left side of Equation 25 directly, and then the squared bias plus the variance:

4.0.17 Python

```

f4_sb = mpg_af[cyl_af == 4].mean()

# The left side: average the squared distance from mu(4) over the lines.
left_sb = np.mean((f4_sb - preds_af) ** 2)
bias2_sb = (f4_sb - preds_af.mean()) ** 2
variance_sb = np.mean((preds_af - preds_af.mean()) ** 2)
print(np.round([left_sb, bias2_sb, variance_sb, bias2_sb + variance_sb], 2))

[2.18 0.32 1.86 2.18]

```

4.0.18 R

```
f4_sb <- mean(auto_af$mpg[auto_af$cylinders == 4])

left_sb <- mean((f4_sb - preds_af)^2)
bias2_sb <- (f4_sb - mean(preds_af))^2
variance_sb <- mean((preds_af - mean(preds_af))^2)
round(c(left = left_sb, bias2 = bias2_sb, variance = variance_sb,
        sum = bias2_sb + variance_sb), 2)
```

left	bias2	variance	sum
2.18	0.32	1.86	2.18

The model's own expected squared error is 2.18, and the squared bias 0.32 plus the variance 1.86 add to 2.18, as Theorem 4.5 says.

Theorem 4.6 (Bias–variance decomposition). *Choose an input value x , and let the output variable $Y = \mu(x) + \varepsilon$ as in Equation 20, where ε , the noise in this new output, has mean 0 and variance σ^2 and is independent of the training set $\mathcal{D}_{\text{train}}$, whose outputs contain their own separate noise draws. Let $\hat{y}$ be the prediction at x of the model a learning algorithm fits to $\mathcal{D}_{\text{train}}$, which varies with $\mathcal{D}_{\text{train}}$, and $\mu_h(x)$ be the *average fit* of that prediction. Under squared-error loss, the expected^a error at x splits into three terms. They are the square of the *bias* $\mathbb{B}[\hat{y}]$ of the prediction, the *variance* $\mathbb{V}[\hat{y}]$ of the prediction, and the variance σ^2 of the noise. Here $\mathbb{B}[\hat{y}]$ and $\mathbb{V}[\hat{y}]$ are short forms of $\text{Bias}(\hat{y})$ and $\text{Var}(\hat{y})$:*

$$\mathbb{E}_{\mathcal{D}_{\text{train}}, \varepsilon}[(Y - \hat{y})^2] = (\mathbb{B}[\hat{y}])^2 + \mathbb{V}[\hat{y}] + \sigma^2. \quad (26)$$

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

i Proof

Proof. By Theorem 4.4 and Theorem 4.5, the expected^a error is the sum of the *variance* σ^2 of the *noise*, $(\mu(x) - \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}(x)])^2$, and $\mathbb{E}_{\mathcal{D}_{\text{train}}}[(\hat{y} - \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}(x)])^2]$. The first is σ^2 itself. The second is the square of the *bias* of the prediction, because that bias is the *average fit* of the prediction minus the value of the *true mean function* of Y given $X = x$, by Theorem 4.2. A square does not change when its base changes sign. The third is the variance of the prediction, by Equation 23. $\square$

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

i Remark (The decomposition holds for any algorithm)

Remark 4.6 (The decomposition holds for any algorithm). Equation 26 holds for any learning algorithm, and ε need not be normal. The statistics notes prove it as the bias, variance and noise decomposition^a of the expected squared prediction error.

^a<https://morrison-lab.github.io/sds/model-validation.html#thm-epe-decomposition>

Example 4.12 (Splitting the error of the cylinders line at four cylinders). Continue Example 4.6 with the same 2000 fitted lines. The input is $x = 4$ cylinders, and the output Y is the *mpg* of a four-cylinder car, so $\mu(4)$ is the population mean *mpg* of the four-cylinder cars and the noise is how far a single car sits from that population mean. Compute the three terms of Equation 26 and, separately, the expected squared error itself:

4.0.19 Python

```
# Four-cylinder cars: their mean mpg is mu(4), and their spread is the noise.
y4_bv = mpg_af[cyl_af == 4]

bias2_bv = (y4_bv.mean() - preds_af.mean()) ** 2
variance_bv = np.mean((preds_af - preds_af.mean()) ** 2)
noise_bv = np.mean((y4_bv - y4_bv.mean()) ** 2)

# The left side: average the squared error over every line and every car.
error_bv = np.mean((y4_bv[None, :] - preds_af[:, None]) ** 2)
# Its Monte Carlo standard error: each line's mean squared error over the cars is one draw.
line_err_bv = np.mean((y4_bv[None, :] - preds_af[:, None]) ** 2, axis=1)
mc_se_bv = line_err_bv.std(ddof=1) / np.sqrt(len(line_err_bv))
print(np.round([bias2_bv, variance_bv, noise_bv, bias2_bv + variance_bv + noise_bv, error_bv], 2), rowsep=1)

[ 0.32  1.86 31.99 34.18 34.18] 0.067
```

4.0.20 R

```
y4_bv <- auto_af$mpg[auto_af$cylinders == 4]

bias2_bv <- (mean(y4_bv) - mean(preds_af))^2
variance_bv <- mean((preds_af - mean(preds_af))^2)
noise_bv <- mean((y4_bv - mean(y4_bv))^2)

# The left side: average the squared error over every line and every car.
err_mat_bv <- outer(preds_af, y4_bv, function(h, y) (y - h)^2)
error_bv <- mean(err_mat_bv)
# Its Monte Carlo standard error: each line's mean squared error over the cars is one draw.
line_err_bv <- rowMeans(err_mat_bv)
mc_se_bv <- sd(line_err_bv) / sqrt(length(line_err_bv))
c(round(c(bias2 = bias2_bv, variance = variance_bv, noise = noise_bv,
          sum = bias2_bv + variance_bv + noise_bv, error = error_bv), 2),
  mc_se = round(mc_se_bv, 3))
```

bias2	variance	noise	sum	error	mc_se
0.320	1.860	31.990	34.180	34.180	0.067

The three terms add to 34.18, which is the expected squared error 34.18 computed directly, as Theorem 4.6 says. That identity is exact for the 2000 lines and the cars used here. As an estimate of the expected squared error over all training sets, 34.18 has a 95% Monte Carlo confidence interval of 34.05 to 34.31. Most of it is the **noise** variance^a, 32.0: even a model with no **bias** and no **variance** would have that expected squared error on a four-cylinder car. The squared bias is only 0.32 and the variance 1.86. The decomposition holds for any learning algorithm, so the same arithmetic would split the error of a tree or a neural network.

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

i Watch

Abu-Mostafa derives the same decomposition in his Caltech lecture on the bias–variance trade-off, from 16:43^a, and he reads bias and variance from learning curves, from 46:16^b [20], Lecture 8.

^a<https://www.youtube.com/watch?v=zrEyxfl2-a8&t=1003s>

^b<https://www.youtube.com/watch?v=zrEyxfl2-a8&t=2776s>

i Watch

Hastie and Tibshirani follow the bias–variance trade-off^a [6], video 2.3.

^a<https://www.youtube.com/watch?v=pvcEQfcO3pk>

i Watch

Weinberger explains the bias–variance decomposition^a [21], Lecture 19.

^a<https://www.youtube.com/watch?v=zUJbRO0Wavo>

i Watch

Starmer explains bias and variance^a [22] without the algebra.

^a<https://www.youtube.com/watch?v=EuBBz3bI-aA>

i Read

GeeksforGeeks explains the bias–variance trade-off^a [23].

^a<https://www.geeksforgeeks.org/machine-learning/ml-bias-variance-trade-off/>

4.1 Reading the decomposition

Exercise 4.3 (Read the three terms). Equation 26 splits the expected squared error at x into three terms. For each one, say what it measures, which panel of Figure 15 shows it large (if any), and whether a different learning algorithm or more data could make it smaller.

Solution

Solution 4.3. Only two of the three can be changed.

The first term, $(\mu(x) - \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}(x)])^2$, is the square of the bias (Definition 4.7) of $\hat{y}$ as an estimate of $\mu(x)$: how far the algorithm’s average fit is from the truth. It is a property of the *family* of models, not of any one fit. A straight line fitted to a curve has bias no amount of data will remove, because the average of many lines is still a line. That is the top-left panel of Figure 15, and the quadratic in the top-right panel behaves the same way: the five fits agree with each other and all differ from the truth in the same way. A different family can shrink it; more data cannot. A model whose error is mostly this term is said to *underfit*.

The second term, $\mathbb{E}_{\mathcal{D}_{\text{train}}}[(\hat{y} - \mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}(x)])^2]$, is the variance (Definition 4.8) of $\hat{y}$: how far a single fit is from the average fit. It is a property of how strongly the algorithm reacts to its training data. The bottom-right panel of Figure 15 shows five quartics fitted to five samples from the same world, tracking the truth on average and differing strongly between the data points. More data or a less flexible family can shrink it. A model whose error is mostly this term is said to *overfit*.

The third term, σ^2 , is the noise in Y itself. No panel of Figure 15 shows it, because the figure plots only fitted curves and the truth, never the noisy data, and σ^2 is the same for every family. It is not a fact about the model or the algorithm, and no method in these notes reduces it. It sets a lower limit: no model, however good, predicts better than σ^2 on average. When a reported error is near that limit, little further improvement is possible; when a reported error is *below* it, something is wrong with how it was measured.

Definition 4.9 (Underfitting). A model is **underfitting** when the bias term of Equation 26 is large enough to dominate its error.

Definition 4.10 (Overfitting). A model is **overfitting** when the variance term of Equation 26 is large enough to dominate its error.

See overfitting^a in the statistics notes.

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

rme's overfitting¹⁰ names the same failure by its symptom: a model that fits its training data well but predicts new observations poorly.

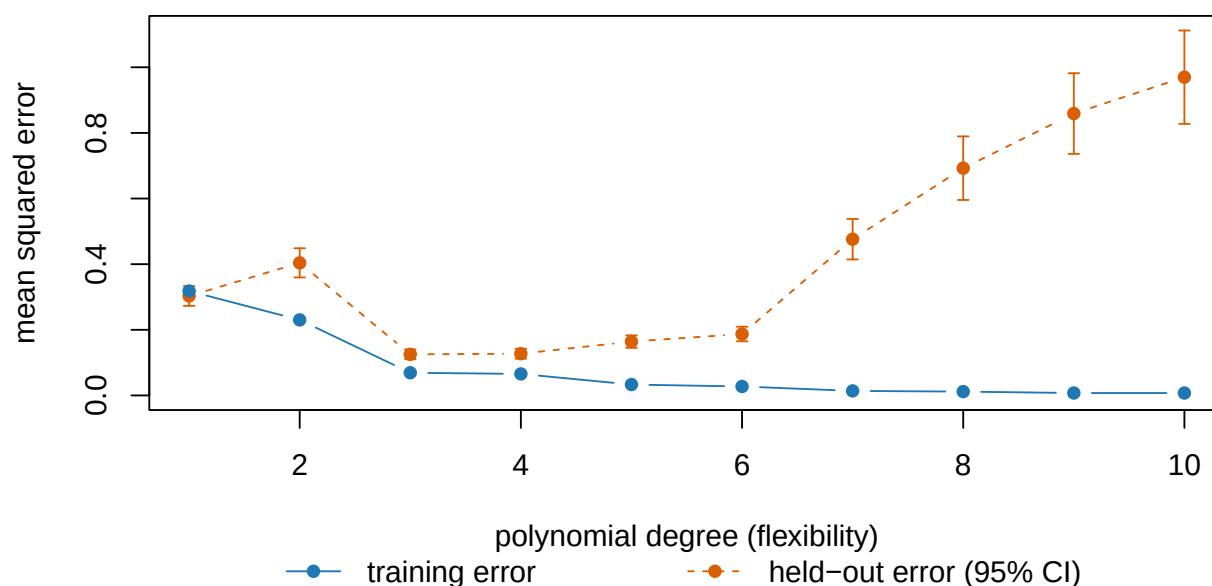


Figure 16: Training error and held-out error of polynomial fits of degree 1 to 10, fitted to 15 noisy points from a smooth curve. Training error only falls as the degree grows. Held-out error, estimated on 500 fresh points with a 95% confidence interval, falls while the fit is underfitting, then rises again as it overfits.

Read

GeeksforGeeks explains underfitting and overfitting^a [24].

^a<https://www.geeksforgeeks.org/machine-learning/underfitting-and-overfitting-in-machine-learning/>

Read

Domingos, a University of Washington professor, gives twelve lessons on generalization, overfitting and feature engineering in A Few Useful Things to Know About Machine Learning^a [25].

^a<https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf>

¹⁰<https://morrison-lab.github.io/rme/chapters/predictor-selection.html#def-overfitting>

Exercise 4.4 (Predict weight from height). Suppose we predict a person’s weight from their height. Could any model, given enough data, drive its expected squared error to zero?

Solution

Solution 4.4. No. The two are related, and yet no function of height alone will ever return weight exactly, because two people of the same height differ in ways that height does not record. That leftover spread is called the *residual variance*: the part of the output that the inputs do not determine.

Exercise 4.5 (Predict weight from height (part 2)). Use the setup of Exercise 4.4. If not, what would lower the minimum possible error?

Solution

Solution 4.5. Residual variance depends on the features and not on the model, so a better model leaves it unchanged. Record frame size too, and some of the spread can move out of the residual variance and into the part a model can learn. Averaged over the inputs, the residual variance does not rise when a feature is added, and it falls exactly when the feature changes the conditional mean of weight given height.

Definition 4.11 (Residual variance). The **residual variance** (also called the **irreducible error**) at x is the variance^a of the **noise** ε in the **noise model**: the variance of the output Y around its **true mean function** at x ,

$$\sigma^2 \stackrel{\text{def}}{=} \text{Var}(\varepsilon) = \text{Var}(Y \mid X = x) = \mathbb{E}[(Y - \mu(x))^2 \mid X = x]. \quad (27)$$

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

i Remark (What “irreducible” means)

Remark 4.7 (What “irreducible” means). The **residual variance** is the part of the expected^a squared error in Equation 26 that no choice of model or algorithm^b changes. It is a variance, a component of the **expected loss**, and not an error.

It is irreducible only given the inputs. It depends on which inputs the model observes. Averaged over new inputs, the residual variance is $\mathbb{E}[\text{Var}(Y \mid X)]$. By the law of total variance^c, adding a feature never raises this average, and lowers it exactly when the feature changes the **conditional mean** of Y given the other inputs. At a single x , the residual variance can rise or fall when a feature is added.

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

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

^c<https://morrison-lab.github.io/pds/variance-covariance.html#thm-total-variance>

i Remark (Estimating the residual variance)

Remark 4.8 (Estimating the residual variance). The **residual variance** is a population^a quantity, which $\hat{\sigma}^2$, computed from the residuals of a **fitted model**, estimates^b.

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

^b<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

i Remark (Residual variance in the statistics notes)

Remark 4.9 (Residual variance in the statistics notes). The statistics notes define the same residual variance^a for the prediction at one covariate value x . It is the σ^2 term in their bias, variance and noise decomposition^b of the expected squared prediction error, which they prove step by step.

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

^b<https://morrison-lab.github.io/sds/model-validation.html#thm-epe-decomposition>

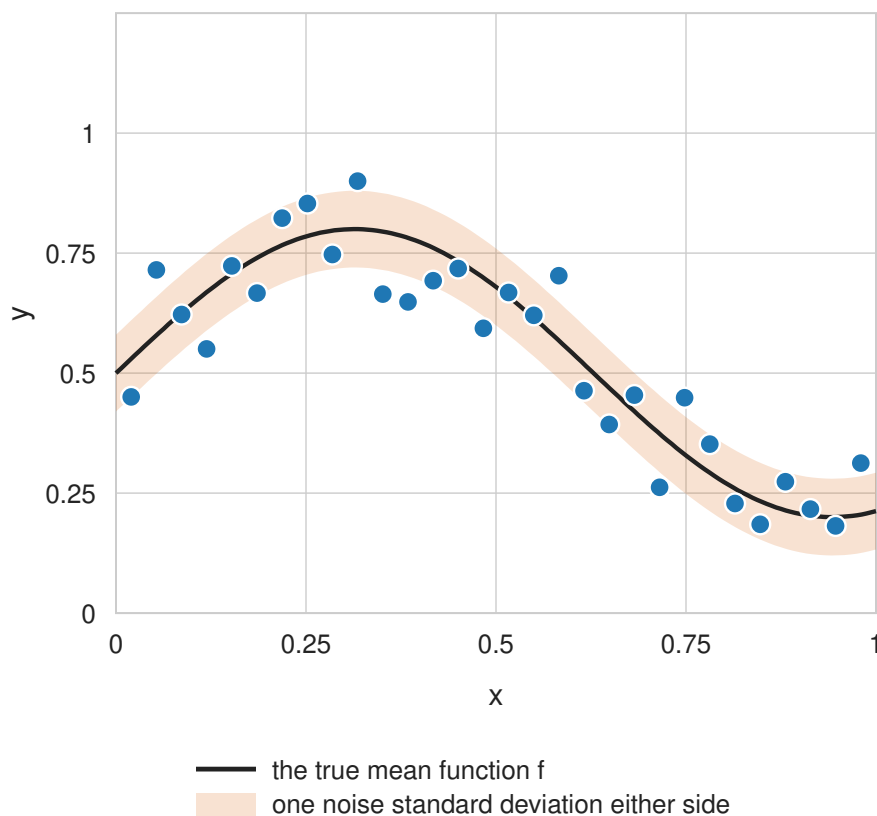


Figure 17: Thirty points drawn from a true mean function plus noise. The band reaches one standard deviation of the noise above and below the true mean function, so its squared half-width is the residual variance σ^2 . Even a model that matches the true mean function exactly misses a typical point by about the half-width of the band, because the noise moves each point off the function. No model can predict the observed values more closely than the spread of the noise allows.

Exercise 4.6 (Error without noise). Imagine a data-generating process with no noise at all: $Y = \mu(x)$ exactly, with $\mu(x) = \sin(2.5\pi x)$ on $-1 \leq x \leq 1$, a curve with two and a half full waves. A student fits polynomials of degree 1, 3, 5 and 7 by least squares to training sets of $n = 8$ points, and tests each fit on new points from the same curve.

The degree-7 polynomial passes through all eight training points. Which polynomial do you expect to predict best on new points?

Solution

Solution 4.6. Fit each degree to 200 training sets of $n = 8$ points, each an equally spaced grid on $[-1, 1]$ shifted a little further along from one set to the next, and split the error on 401 evenly spaced test

points with Equation 26. The grids are fixed, not random, so the R and Python tabs compute the same numbers.

4.1.1 Python

```
import numpy as np
import pandas as pd

# The target, with no noise: the test labels are exactly mu(x).
def f_nn(x):
    return np.sin(2.5 * np.pi * x)

n_nn, sets_nn = 8, 200
x_test_nn = np.linspace(-1, 1, 401)
rows_nn = []
for degree in (1, 3, 5, 7):
    fits, train_err = [], []
    for k in range(sets_nn):
        # Each training set is the same equally spaced grid, shifted a little.
        # Shifting by hand, rather than drawing random points,
        # makes the numbers identical in the R tab.
        x = -1 + 2 / n_nn * (np.arange(n_nn) + (k + 0.5) / sets_nn)
        coef = np.linalg.lstsq(np.vander(x, degree + 1), f_nn(x), rcond=None)[0]
        fits.append(np.vander(x_test_nn, degree + 1) @ coef)
        train_err.append(np.mean((np.vander(x, degree + 1) @ coef - f_nn(x)) ** 2))
    fits = np.array(fits)
    avg_fit = fits.mean(axis=0)
    # The three terms of the decomposition. Noise is zero, so there are only two.
    bias2 = np.mean((avg_fit - f_nn(x_test_nn)) ** 2)
    variance = np.mean(fits.var(axis=0))
    rows_nn.append((degree, np.mean(train_err), bias2, variance, bias2 + variance))
print(pd.DataFrame(rows_nn, columns=["degree", "train", "bias2", "variance", "error"]).round(3))
```

	degree	train	bias2	variance	error
0	1	0.494	0.500	0.006	0.506
1	3	0.430	0.441	0.016	0.457
2	5	0.102	0.743	0.389	1.132
3	7	0.000	1.465	0.862	2.327

4.1.2 R

```
# The target, with no noise: the test labels are exactly mu(x).
f_nn <- function(x) sin(2.5 * pi * x)

n_nn <- 8
sets_nn <- 200
x_test_nn <- seq(-1, 1, length.out = 401)
nn_results <- do.call(rbind, lapply(c(1, 3, 5, 7), function(degree) {
  # Each training set is the same equally spaced grid, shifted a little.
  # Shifting by hand, rather than drawing random points,
  # makes the numbers identical in the Python tab.
  fits <- sapply(seq_len(sets_nn) - 1, function(k) {
    x <- -1 + 2 / n_nn * (seq_len(n_nn) - 1 + (k + 0.5) / sets_nn)
    design <- outer(x, 0:degree, "^")
    coef <- qr.solve(design, f_nn(x))
    c(test = drop(outer(x_test_nn, 0:degree, "^") %*% coef),
      train = mean((design %*% coef - f_nn(x))^2))
  })
  test_fits <- fits[seq_along(x_test_nn), ]
  avg_fit <- rowMeans(test_fits)
  # The three terms of the decomposition. Noise is zero, so there are only two.
  bias2 <- mean((avg_fit - f_nn(x_test_nn))^2)
  variance <- mean(apply(test_fits, 1, var) * (sets_nn - 1) / sets_nn)
  data.frame(degree = degree, train = mean(fits["train", ]),
    bias2 = bias2, variance = variance, error = bias2 + variance)
}))
round(nn_results, 3)
```

```
# A tibble: 4 x 5
  degree train bias2 variance error
  <dbl> <dbl> <dbl>   <dbl> <dbl>
1     1  0.494  0.5    0.006  0.506
2     3  0.43  0.441   0.016  0.457
3     5  0.102  0.743   0.389  1.13
4     7  0     1.46   0.862  2.33
```

The cubic has the lowest error, 0.46, and the line is close behind at 0.51. The septic, which fits all eight training points exactly (training error 0.00), has the highest, 2.33.

Expect the low degrees^a to win, though the degree-7 fit has the best training error. With eight points, the degree-7 fit has no freedom left over, so each shifted training set bends it differently (variance^b 0.86).

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

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

Exercise 4.7 (Error without noise (part 2)). Use the setup of Exercise 4.6.

The residual variance σ^2 of Definition 4.11 is zero here. Where, then, does the error of the best polynomial^a come from?

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

Solution

Solution 4.7. The error comes from the model, not from the world. A line, or even a cubic, cannot follow two and a half waves, so every fit misses the curve in the same way (squared bias^a 0.50 for the line). A flexible polynomial^b passes through the curve at the training points and still goes wrong between them, because eight points are too few to pin it down there. Either way, the part of μ the model cannot learn from this much data acts on the fit like noise in the labels, which the definition of **deterministic noise**

below makes precise.

^a<https://morrison-lab.github.io/sds/estimation.html#def-bias>

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

Figure 18 draws the fits behind the table above: the curves each training set produces, their average, and the error by degree.

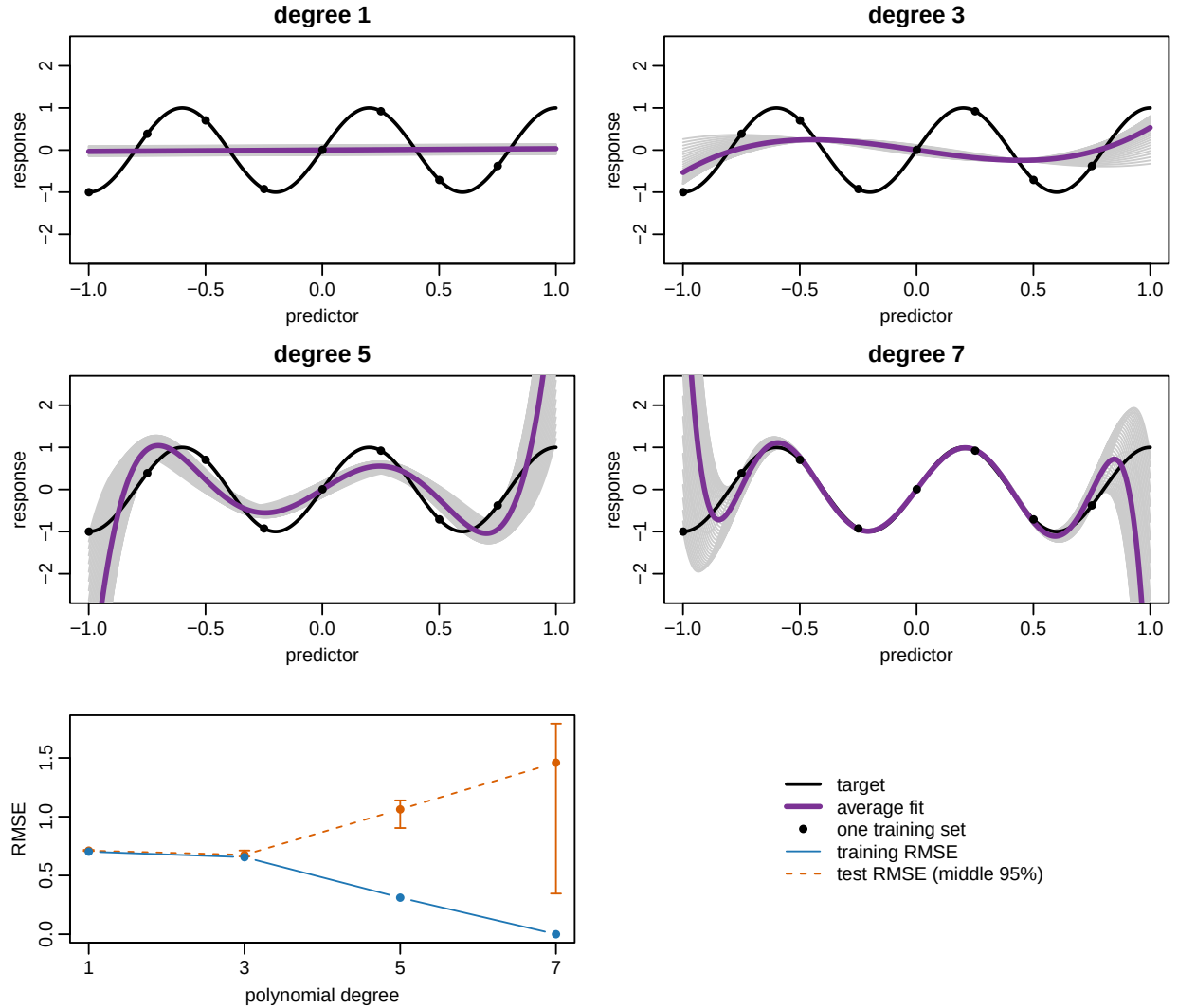


Figure 18: Fits to a noiseless curve, μ , from 200 shifted grids of 8 points. Top: for each polynomial degree, 40 of the 200 fitted curves (grey), their average (colour) and one training set (dots). The line and the cubic miss the curve in the same way every time; the degree-7 polynomial passes through every training set and still swings wildly between the points. Bottom: training and test root mean squared error (RMSE) by degree, with the bars spanning the middle 95% of the 200 training sets.

Definition 4.12 (Best member of a model family). The **best member** h^* of a mean-function model family^a $\mathcal{H}$ for a target function μ is the member h of $\mathcal{H}$ with the smallest mean squared difference from μ , $\mathbb{E}[(\mu(X) - h(X))^2]$, where the random predictor X is a draw of the predictor x from the population^b.

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

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-population>

Example 4.13 (The best line for a curved target). Let the target μ be a sine curve, and let $\mathcal{H}$ be the family of straight lines. No line follows a sine curve, but some lines are closer to it than others, averaging the squared gap over the population of x . The closest line is the best member h^* . A line fitted to a training set is a different object, because it depends on the data and so usually differs from h^* .

Definition 4.13 (Deterministic noise). The **deterministic noise** of a mean-function model family^a $\mathcal{H}$ at x is the part of $\mu(x)$ that the family cannot express: $\mu(x) - h^*(x)$, where h^* is the best member of $\mathcal{H}$ (Definition 4.12).

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

i Remark (Deterministic versus stochastic noise)

Remark 4.10 (Deterministic versus stochastic noise). Deterministic noise differs from the label noise ε of Equation 20, which we call *stochastic noise* when we need to tell the two apart, in two ways:

- Deterministic noise depends on the family: a richer family leaves less of μ unexpressed.
- Deterministic noise is fixed for a given x : drawing a new training set does not change it.

i Watch

Abu-Mostafa separates stochastic from deterministic noise in his Caltech lecture on overfitting [20], Lecture 11; the same chapter of his book, Abu-Mostafa et al. [1, Ch. 4], gives the written treatment.

Exercise 4.8 (Why the ranking was wrong). Table 2 ranked the fitted polynomials by their error on the training set, and the quartic came first. Which of the three terms of Equation 26 does training-set error measure?

Solution

Solution 4.8. None of them. Training-set error is an optimistic estimate of the expected error on the left-hand side of Equation 26.

Exercise 4.9 (Why the ranking was wrong (part 2)). Use the setup of Exercise 4.8. Why does the ranking by training-set error favour the most flexible family?

Solution

Solution 4.9. Training-set error grows more optimistic with flexibility, because a more flexible model can fit the particular noise in the sample it was given. So the quantity Table 2 ranks by improves for exactly the reason the variance^a term worsens.

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

i Remark (The bias-variance tradeoff)

Remark 4.11 (The bias-variance tradeoff). Across the families of Figure 15, more flexibility gives lower **bias** but higher **variance**: the line and the quadratic miss the truth in the same way every time, the cubic follows it closely, and the quartics swing apart from one another. Because of that trade, model selection needs careful study and is not a routine step. For the crime data, ranking the polynomials by training error put the quartic first because it is the most flexible, not because it predicts best (Exercise 4.8), so choosing the degree takes more than comparing the training errors.

Exercise 4.10 (Changes along the tradeoff). List changes you could make to the data or the model that shrink one of the two correctable terms of Equation 26. For each, say in which direction it changes the bias term and in which direction it changes the variance term.

Solution

Solution 4.10. Both correctable terms change in opposite directions as flexibility changes, so the usual changes are:

- more data, which shrinks variance and leaves bias alone;
- a smaller model family, which lowers variance and raises bias;
- a larger model family, which does the reverse;
- keeping the family but penalizing the size of the fitted parameters, which deliberately accepts a little bias in exchange for a larger reduction in variance.

Exercise 4.11 (Why would anyone choose the less flexible model?). A straight line can only tilt; a high-degree polynomial can bend almost any way. Give two reasons the city of Section 2 might still prefer the straight line: one that applies if it wants to *explain* how crime varies with income, and one that applies even if it only wants to *predict*.

Solution

Solution 4.11.

- To explain, the line is readable. Its slope says how much the predicted crime rate changes per unit of income, and its sign says in which direction. The quartic of Exercise 2.16 has no single number like that: each of its five coefficients multiplies a different power of income, so none of them alone says how crime changes with income, even when the curve predicts well. James et al. [26, Sec. 2.1.3] calls this the trade-off between flexibility and interpretability.
- To predict, the line may simply be more accurate. With five neighbourhoods, a flexible fit spends its freedom chasing those five points. Figure 15 shows what that costs: flexible fits to different samples from the same population disagree wildly with each other, so a prediction for a sixth neighbourhood depends on which five happened to be sampled. That is the variance term of Equation 26, so the quartic's perfect training fit in Exercise 2.16 was no evidence that it predicts well. James et al. [26, Sec. 2.1.3] makes the same point: even when only prediction matters, a less flexible method often predicts more accurately, because a very flexible one can overfit (Definition 4.10).

Neither reason makes the line the right choice by default. If the true relationship bends, the line's bias does not shrink with more data.

Definition 4.14 (Regularization). **Regularization** is fitting a model by minimizing its empirical risk plus a penalty on the size of its parameters [27]. With a penalty function $\text{pen}(\theta) \geq 0$ and a penalty weight $\lambda \geq 0$, regularization minimizes the **objective** plus the weighted penalty,

$$J(\theta) + \lambda \text{pen}(\theta). \quad (28)$$

We cover regularization in detail when we study models whose parameters should be penalized.

Example 4.14 (Regularizing the crime-data quartic). For the polynomial fits to the crime data (Table 2), a penalty on the coefficients makes a large coefficient on x^4 costly, so the fitted quartic moves closer to a flatter curve that bends to fit each of the five neighbourhoods less closely.

Definition 4.15 (Penalty weight (regularization strength)). A **penalty weight** $\lambda \geq 0$, also called the **regularization strength**, is the number that scales a penalty $\text{pen}(\theta) \geq 0$, a nonnegative measure of

the size of a model's parameters, when the penalty is added to the training objective in [regularization](#). The value $\lambda = 0$ leaves the objective unchanged, and a larger λ penalizes large parameters more heavily.

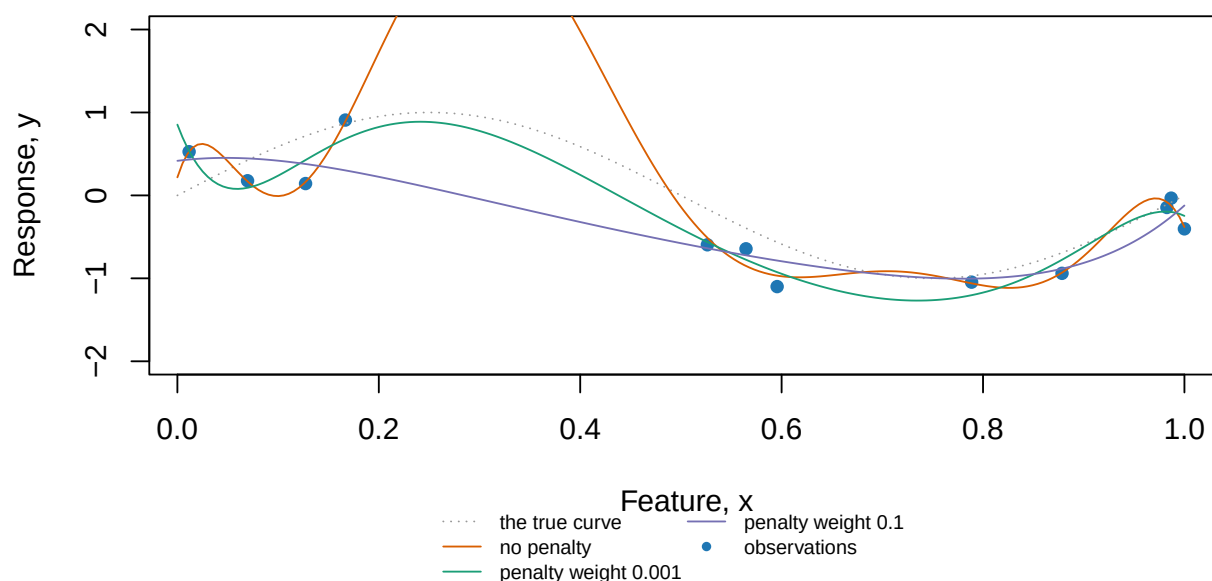


Figure 19: A degree-9 polynomial fitted to 12 noisy points from a smooth curve, with a penalty on the size of its coefficients. With no penalty, the fit passes close to every point and swings wildly between them. A larger penalty weight pulls the fit toward a smoother curve that follows the points less closely.

Read

GeeksforGeeks explains regularization^a [28].

^a<https://www.geeksforgeeks.org/machine-learning/regularization-in-machine-learning/>

Exercise 4.12 (Read the response to a change, name the term). Each of the following describes what happened when you changed one thing about a model and scored it honestly again. For each, say which term of Equation 26 the response points at, and what you would try next. Strengthening the regularization penalty raises training error a little and lowers held-out error.

Solution

Solution 4.12. Variance^a. A stronger penalty limits how closely the fitted model can adapt to the peculiarities of one training set, so it shrinks $\mathbb{E}_{\mathcal{D}_{\text{train}}} \left[\left(\hat{y} - \mathbb{E}_{\mathcal{D}_{\text{train}}} [\hat{y}(x)] \right)^2 \right]$ at the cost of a little bias^b. Held-out error fell, so the variance removed outweighed the bias added. Keep raising the penalty while held-out error keeps falling, and stop where it starts to rise.

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

^b<https://morrison-lab.github.io/sds/estimation.html#def-bias>

Exercise 4.13 (Read the response to a change, name the term (part 2)). Use the setup of Exercise 4.12. Doubling the training set does not change held-out error, but switching to a more flexible model family^a lowers training and held-out error together.

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

Solution

Solution 4.13. **Bias**^a. More data cuts variance^b, because each fit then depends less on the peculiarities of one training set, but it does not move the **average fit** μ_h toward the truth. A held-out error that does not move with twice the data is therefore not mainly coming from variance. A more flexible family can place μ_h closer to the truth, so both errors fell when you switched. Keep adding flexibility, or better features, while held-out error keeps falling.

^a<https://morrison-lab.github.io/sds/estimation.html#def-bias>

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

Exercise 4.14 (Read the response to a change, name the term (part 3)). Use the setup of Exercise 4.12. Training error and held-out error are both small and roughly equal, and collecting ten times as much data changes neither.

Solution

Solution 4.14. **Residual variance**, most likely. Both errors agree, so variance^a is small; both are small, so bias^b is small; and more data moving neither says the remaining error is not a property of the model at all. There is nothing left to try on the modelling side. The remaining options are outside the model: measure the inputs^c more precisely, or find a feature that carries information the current ones do not. Case 3 is the one students most often keep working on. Recognizing a finished model is as much a skill as improving an unfinished one, and σ^2 is unknown, so “most likely” is as strong as the diagnosis gets.

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

^b<https://morrison-lab.github.io/sds/estimation.html#def-bias>

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

Exercise 4.15 (A perfect algorithm). Suppose an algorithm has zero bias and zero variance at every x , so every training set gives it the same fit, and that fit is f . Can the risk of its fitted model still be nonzero? Use Equation 26 to say why or why not, and say what each of the two zeros means in words.

Solution

Solution 4.15. Yes: the expected squared error at x is then σ^2 , because the other two terms of Equation 26 are zero. The risk is nonzero whenever the noise is.

Zero variance means the algorithm returns the same fitted model on every training set, so it ignores its data. Zero bias means that model is the truth, $\mathbb{E}_{\mathcal{D}_{\text{train}}}[\hat{y}(x)] = \mu(x)$. Both zeros together describe an algorithm with nothing left to learn, and even that algorithm cannot predict Y more closely than the noise allows.

Students tend to read “zero bias and zero variance” as “zero error”. Point back to Definition 4.11.

Remark (Bias and variance for a classifier)

Remark 4.12 (Bias and variance for a classifier). The decomposition of Theorem 4.6 is for squared error, and it does not carry over to a classifier’s zero-one loss as an exact sum. For a classifier the words are an analogy. The Bayes error plays the part of the **residual variance**: no classifier using the same features does better. Under zero-one loss, bias and variance combine multiplicatively rather than additively [29,

Sec. 6]. A classifier’s error at x depends on the bias only through the boundary bias, which records how far the average fit lies from the decision threshold and on which side. A fit that is very biased but on the correct side of the threshold is not penalized for that bias. In that case, reducing variance alone moves the error toward the Bayes error. Friedman puts it as “variance tends to dominate bias for classification” [29, Sec. 7], p. 63. He also shows the opposite case: where the average fit is on the wrong side of the threshold, reducing variance makes the error worse [29, Sec. 7], p. 62.

The course website’s version of this page has an interactive plot here.

Reading

- Hastie et al. [2], Chapter 2, covers the same material as this chapter at greater length and with more mathematics. Section 2.9 is the bias-variance trade-off; it is worth reading after Equation 26 rather than before.
- James et al. [3], Chapter 2, covers the same material in a less technical way and is the better first read. Section 2.1.1 is the prediction-versus-inference distinction from Section 2, and Section 2.2.2 is the bias-variance decomposition.
- Mitchell [30], Chapter 1, is the source of the definition of learning that most courses still quote.
- Starmer [31] covers the fundamental concepts of machine learning and cross-validation with pictures and little algebra.
- Stanford University [32], Lectures 3–6, are handwritten notes from a Stanford probability and inference course. They cover concentration inequalities (Hoeffding’s inequality in Lecture 3), empirical risk minimization and overfitting (Lecture 5), and uniform convergence with generalization bounds (Lecture 6), for readers who want the mathematics behind Theorem 3.1 in more detail. The textbook these notes recommend for this material is Shalev-Shwartz and Ben-David [33], Sections 2.1–2.3 and 4.1–4.2, which is free online.

i Watch

The Data Science Learning Community’s book clubs discuss *An Introduction to Statistical Learning*, Chapter 2, in recorded meetings:

- the R book club, in part 1^a and part 2^b [34];
- the Python book club, in part 1^c and part 2^d [35].

^a<https://www.youtube.com/watch?v=xjwAqgqfSw&list=PL3x6DOfs2NGjlhpKMiWsDcKL2nngwXROq>

^b<https://www.youtube.com/watch?v=nknWd1ZOQSO&list=PL3x6DOfs2NGjlhpKMiWsDcKL2nngwXROq>

^c<https://www.youtube.com/watch?v=i0dSfaGIHak&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>

^d<https://www.youtube.com/watch?v=NRb6U9pJq58&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>

i Watch

Andrew Ng’s short lectures for DeepLearning.AI cover the same ideas for neural networks: how to split data into training, validation and test sets [36], how to read underfitting and overfitting as bias and variance [37], and a basic recipe for deciding what to change next [38]. They call the validation set the “dev set”.

References

- [1] Y. S. Abu-Mostafa, M. Magdon-Ismael, and H.-T. Lin, *Learning from data: A short course*. AMLBook, 2012. Available: <https://amlbook.com/>
- [2] 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/>

- [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, “Training, validation, and test data sets — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Training,_validation,_and_test_data_sets
- [5] Wikipedia contributors, “Parametric model — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Parametric_model
- [6] 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=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ>
- [7] Wikipedia contributors, “Loss function — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Loss_function
- [8] Wikipedia contributors, “Empirical risk minimization — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Empirical_risk_minimization
- [9] 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/
- [10] GeeksforGeeks, “ML | common loss functions.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/ml-common-loss-functions/>
- [11] GeeksforGeeks, “Empirical risk minimization.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/empirical-risk-minimization/>
- [12] S. Lall, “EE104: Introduction to machine learning, 2020.” Stanford Online, lecture videos on YouTube, 2020. Available: https://www.youtube.com/playlist?list=PLoROMvodv4rN_Uy7_wmS051_q1d6akXmK
- [13] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge University Press, 2004. doi: 10.1017/CBO9780511804441¹¹.
- [14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016. Available: <https://www.deeplearningbook.org/>
- [15] Wikipedia contributors, “Hoeffding’s inequality — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Hoeffding%27s_inequality
- [16] Wikipedia contributors, “Cross-validation (statistics) — Wikipedia, the free encyclopedia.” 2026. Available: [https://en.wikipedia.org/wiki/Cross-validation_\(statistics\)](https://en.wikipedia.org/wiki/Cross-validation_(statistics))
- [17] J. Starmer, “Machine learning fundamentals: Cross validation.” StatQuest, video on YouTube, 2018. Available: <https://www.youtube.com/watch?v=fSytzGwwBVw>
- [18] GeeksforGeeks, “Cross validation in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/cross-validation-machine-learning/>
- [19] G. van den Burg, *SyncRNG: A synchronized Tausworthe RNG for R and Python*. 2024. doi: 10.32614/CRAN.package.SyncRNG¹².
- [20] Y. S. Abu-Mostafa, “Learning from data: CS 156 lectures.” Caltech, lecture videos on YouTube, 2012. Available: <https://www.youtube.com/playlist?list=PLD63A284B7615313A>
- [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=PLl8OIHZGYOQ7bkVbuRthEsaLr7bONzbXS>
- [22] J. Starmer, “Machine learning fundamentals: Bias and variance.” StatQuest, video on YouTube, 2018. Available: <https://www.youtube.com/watch?v=EuBBz3bI-aA>
- [23] GeeksforGeeks, “Bias-variance trade off - machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/ml-bias-variance-trade-off/>
- [24] GeeksforGeeks, “Underfitting and overfitting in ML.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/underfitting-and-overfitting-in-machine-learning/>
- [25] P. Domingos, “A few useful things to know about machine learning,” *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012, doi: 10.1145/2347736.2347755¹³.

¹¹<https://doi.org/10.1017/CBO9780511804441>

¹²<https://doi.org/10.32614/CRAN.package.SyncRNG>

¹³<https://doi.org/10.1145/2347736.2347755>

- [26] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, *An introduction to statistical learning: With applications in Python*. Springer, 2023. doi: 10.1007/978-3-031-38747-0¹⁴.
- [27] Wikipedia contributors, “Regularization (mathematics) — Wikipedia, the free encyclopedia.” 2026. Available: [https://en.wikipedia.org/wiki/Regularization_\(mathematics\)](https://en.wikipedia.org/wiki/Regularization_(mathematics))
- [28] GeeksforGeeks, “Regularization in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/regularization-in-machine-learning/>
- [29] J. H. Friedman, “On bias, variance, 0/1—loss, and the curse-of-dimensionality,” *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 55–77, 1997, doi: 10.1023/A:1009778005914¹⁵.
- [30] T. M. Mitchell, *Machine learning*. in McGraw-hill series in computer science. New York: McGraw-Hill, 1997.
- [31] J. Starmer, *The StatQuest illustrated guide to machine learning*. StatQuest Publications, 2022. Available: <https://statquest.org/statquest-store/>
- [32] Stanford University, “EE 278, probability and statistical inference: Fall 2024 lecture notes.” Handwritten lecture notes, Lectures 1–17, 2024. Available: <http://web.stanford.edu/class/ee278/lectures.html>
- [33] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014. doi: 10.1017/CBO9781107298019¹⁶.
- [34] 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>
- [35] 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>
- [36] A. Ng, “Train/dev/test sets.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=1waHlpKiNyY>
- [37] A. Ng, “Bias/variance.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=SjQyLhQIXSM>
- [38] A. Ng, “Basic recipe for machine learning.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: https://www.youtube.com/watch?v=C1N_PDHuJ6Q

¹⁴<https://doi.org/10.1007/978-3-031-38747-0>

¹⁵<https://doi.org/10.1023/A:1009778005914>

¹⁶<https://doi.org/10.1017/CBO9781107298019>