Anatomy of a Learning Problem

Inputs, outputs, kinds of problems and learning from experience

Last modified: 2026-11-19

```
# The Python chunks on this page import pandas; reticulate provisions it.
reticulate::py_require(c("pandas", "scikit-learn", "scipy", "matplotlib"))
```

> **Source**
>
> Course Overview: What Is Machine Learning? once held the material below as well. It keeps the definition of machine learning, why to learn from data, where machine learning fits, and the course ahead. Both chapters use the Auto and spam data.

1 The anatomy of a learning problem

1.1 Two steps: training and prediction

A machine learning system works in two steps, shown in Figure 1. In the first step, training (Definition 1.3), a learning algorithm (Definition 1.2) reads some training data $\mathcal{D}_{\text{train}}$ (Definition 1.1) and produces a model (Definition 1.5), a function $\hat{y}_{\mathcal{D}_{\text{train}}}$. Training happens once, ahead of time (*train time*). In the second step, prediction, the system feeds a new input x to the model and gets back a prediction $\hat{y}_{\mathcal{D}_{\text{train}}}(x)$. Prediction happens every time the system is used (*run time*). The next section defines each of these terms.

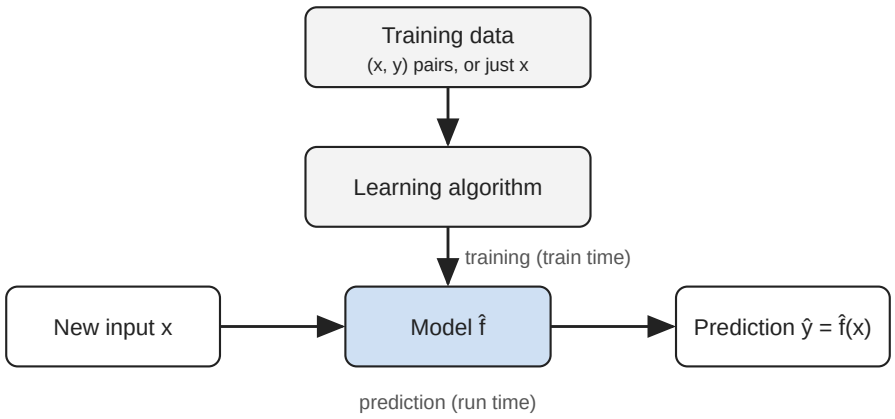


Figure 1: Training turns data into a model once; prediction applies the model to each new input.

1.2 The three parts of a learning problem

Exercise 1.1 (What are the parts of a learning problem?)**.** Figure 1 shows training and then prediction. Name the three objects that training involves:

- what goes in;
- what does the computation;
- what comes out.

Solution

Solution 1.1. Training involves three objects:

- data go in;
- an algorithm does the computation;
- a rule for making predictions comes out.

Definition 1.1 (Training data (training set))**.** The **training data** are inputs x, possibly with outputs y.
Other names: the training data are also called the **training set**.

Definition 1.2 (Learning algorithm)**.** A **learning algorithm** turns the training data (Definition 1.1) into a function h.

Definition 1.3 (Training (fitting, learning, estimation))**.** Running a learning algorithm (Definition 1.2) on the training data is called **training**.
Other names: training is also called **fitting**, **learning** or **estimation**.

Definition 1.4 (Feature (attribute))**.** A **feature** is one number that a model family[a] receives as input [1]: one entry of the output of the preprocessing[b] applied to an input's covariates[c]. With no preprocessing, the features are the covariates.
Other names: a feature is also called an **attribute**. A covariate, a **predictor**, an **input variable** and an **independent variable** are the measured values before preprocessing.

[a] https://morrison-lab.github.io/sds/model-validation.html#def-model-family
[b] https://morrison-lab.github.io/sds/model-validation.html#def-preprocessing
[c] https://morrison-lab.github.io/sds/model-validation.html#def-covariate

Definition 1.5 (Model)**.** A **model**[a] is a set[b] of candidate descriptions of how the outcomes[c] of a data set[d] depend on its inputs[e], or one member of such a set that training has chosen.

[a] https://en.wiktionary.org/wiki/model
[b] https://morrison-lab.github.io/mcs/sets.html#def-set
[c] https://morrison-lab.github.io/sds/model-validation.html#def-outcome-vector
[d] https://morrison-lab.github.io/sds/model-validation.html#def-data-set
[e] https://morrison-lab.github.io/sds/model-validation.html#def-data-set

Remark (A model represents a phenomenon)

Remark 1.1 (A model represents a phenomenon). In science, a model represents an object, a phenomenon or a process, and it makes some part of the world easier to understand, quantify or simulate [2], [3]. The candidates in a model represent how the outcomes[a] of a data set[b] depend on its inputs[c]. Choosing the set[d] of candidates is an assumption: the set rules out every description outside it. George Box summed up what that assumption costs as "all models are wrong but some are useful" [4, p. 202]. He argued that no simple model represents a system in the world exactly, but a well-chosen simple

model can still approximate it usefully, as the ideal gas law $PV = RT$ does for actual gases, none of which obeys it exactly [4, pp. 202–203]. So the question to ask of a model is not whether it is true. In his words, the question is "Is the model illuminating and useful?" [4, p. 203]. Box also noted that a method derived without an explicitly stated model still implies some unstated model [4, p. 201], so even a method that only makes predictions rests on assumptions about the data. A conceptual model[e] is the same idea outside mathematics: a representation of a system, built from concepts, that helps people understand it.

[a] https://morrison-lab.github.io/sds/model-validation.html#def-outcome-vector
[b] https://morrison-lab.github.io/sds/model-validation.html#def-data-set
[c] https://morrison-lab.github.io/sds/model-validation.html#def-data-set
[d] https://morrison-lab.github.io/mcs/sets.html#def-set
[e] https://en.wikipedia.org/wiki/Conceptual_model

Definition 1.6 (Statistical model)**.** A **statistical model** is a model whose candidates are distribution models[a]: either a distribution model family[b] or a fitted distribution model[c].

[a] https://morrison-lab.github.io/sds/model-validation.html#def-distribution-model
[b] https://morrison-lab.github.io/sds/model-validation.html#def-distribution-model-family
[c] https://morrison-lab.github.io/sds/model-validation.html#def-fitted-distribution-model

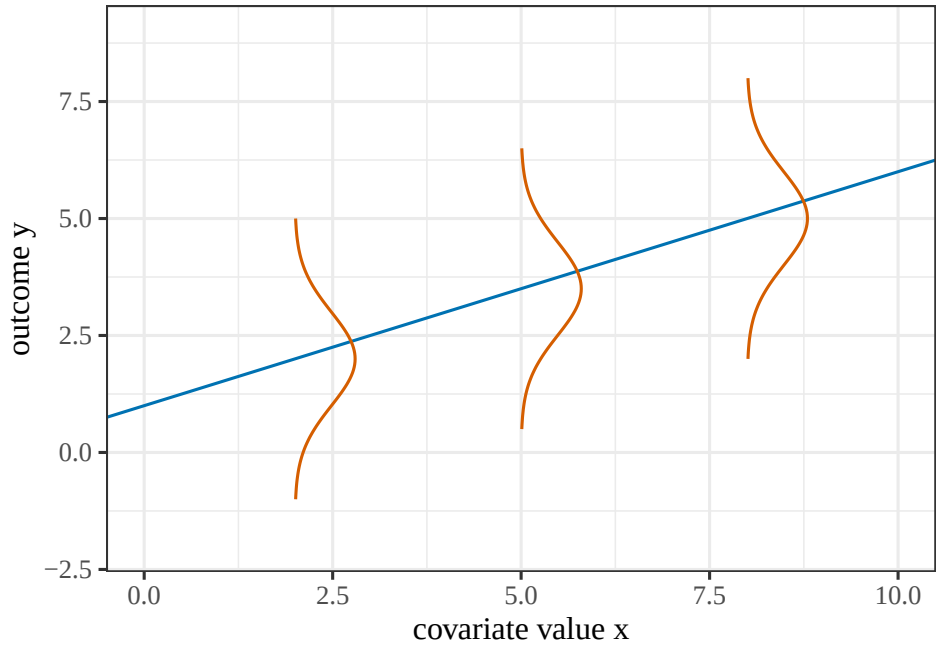
Definition 1.7 (Predictive model)**.** A **predictive model** is a model whose candidates are the members of a prediction function family[a]: either the family itself or one member of the family that training has chosen.

[a] https://morrison-lab.github.io/sds/model-validation.html#def-prediction-function-family

Example 1.1 (Gaussian outcomes with an affine mean: a model of both kinds)**.** Take the set[a] of distribution models[b] that give an outcome[c] Y, given a covariate[d] value x, a Gaussian distribution[e] whose mean[f] is an affine function[g] of x. That set is a statistical model. Each member also gives a mean function[h], and that function predicts the outcome at each covariate value. So the same assumptions also make a predictive model. Figure 2 draws one member.

1.2.1 R

```r
intercept <- 1
slope <- 0.5
noise_sd <- 1
x_at <- c(2, 5, 8)
y_grid <- seq(-3, 3, length.out = 101)
curves <- tidyr::expand_grid(x_at = x_at, dev = y_grid)
curves <- dplyr::mutate(
  curves,
  y = intercept + slope * x_at + dev,
  density = dnorm(dev, sd = noise_sd),
  x = x_at + 2 * density
)
x_limits <- c(0, 10)
y_limits <- c(-2, 9)
curves |>
  ggplot2::ggplot() +
  ggplot2::aes(x = x, y = y, group = x_at) +
  ggplot2::geom_abline(
    intercept = intercept,
    slope = slope,
    colour = "#0072B2"
  ) +
  ggplot2::geom_path(colour = "#D55E00") +
  ggplot2::coord_cartesian(xlim = x_limits, ylim = y_limits) +
  ggplot2::labs(x = "covariate value x", y = "outcome y")
```

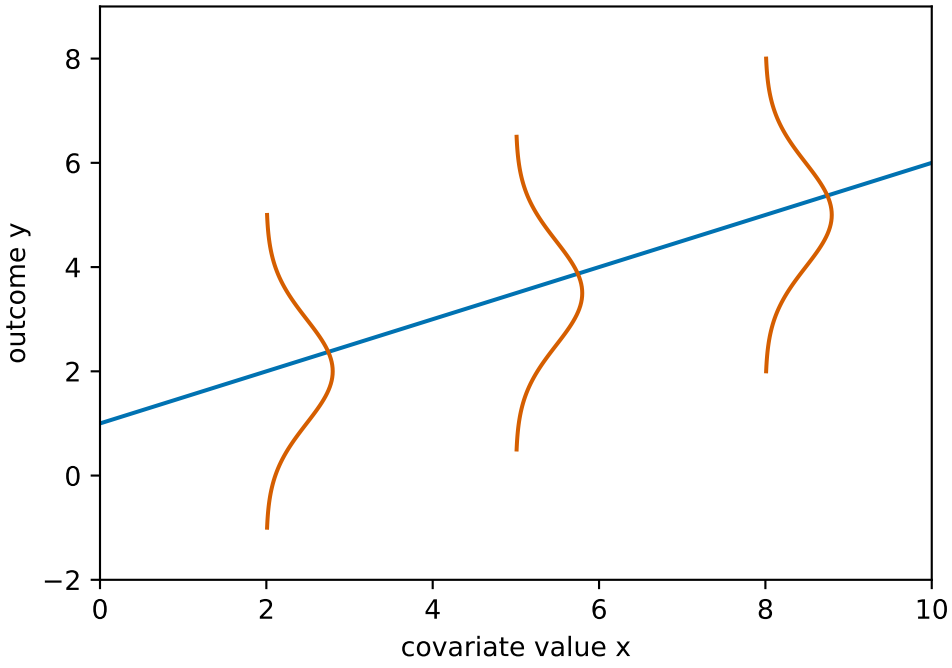


(a)

1.2.2 Python

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

intercept, slope, noise_sd = 1, 0.5, 1
dev = np.linspace(-3, 3, 101)
density = np.exp(-0.5 * (dev / noise_sd) ** 2) / (noise_sd * np.sqrt(2 * np.pi))
x_line = np.array([0, 10])
fig, ax = plt.subplots(figsize=(5, 3.5), layout="constrained")
ax.plot(x_line, intercept + slope * x_line, color="#0072B2");
for x_at in [2, 5, 8]:
    ax.plot(x_at + 2 * density, intercept + slope * x_at + dev, color="#D55E00");
ax.set_xlim(0, 10);
ax.set_ylim(-2, 9);
ax.set_xlabel("covariate value x");
ax.set_ylabel("outcome y");
plt.show()
```



(b)

Figure 2: One member of the statistical model in Example 1.1: the Gaussian distributions[i] of Y given $x = 2$, 5 and 8, centered on the line $y = 1 + 0.5x$, each with standard deviation[j] 1.

[a] https://morrison-lab.github.io/mcs/sets.html#def-set
[b] https://morrison-lab.github.io/sds/model-validation.html#def-distribution-model
[c] https://morrison-lab.github.io/sds/model-validation.html#def-outcome-vector
[d] https://morrison-lab.github.io/sds/model-validation.html#def-covariate
[e] https://morrison-lab.github.io/pds/random-variables.html#def-normal
[f] https://morrison-lab.github.io/pds/expectation.html#def-expectation
[g] https://morrison-lab.github.io/mds/algebra-exponentials.html#def-affine-function
[h] https://morrison-lab.github.io/sds/model-validation.html#def-mean-function
[i] https://morrison-lab.github.io/pds/random-variables.html#def-normal
[j] https://morrison-lab.github.io/pds/variance-covariance.html#def-sd

Remark (Causal models are a third kind)

Remark 1.2 (Causal models are a third kind). A causal model[a] is a further kind of model. Its candidates also describe how the distribution[b] of the data would change under an intervention, such as setting one variable to a chosen value. So a causal model carries a statistical model with it, but a statistical model says nothing about interventions. These notes do not use causal models.

[a] https://en.wikipedia.org/wiki/Causal_model
[b] https://morrison-lab.github.io/pds/random-variables.html#def-cdf

Remark (Which meaning of "model" these notes use)

Remark 1.3 (Which meaning of "model" these notes use). A use of the word model names either a set[a] of candidates or one chosen member, and the surrounding sentence shows which one. These notes usually use the word for a predictive model that training has chosen: the function $\hat{y}_{\mathcal{D}_{\text{train}}}$ that training produces from the training data $\mathcal{D}_{\text{train}}$ and a prediction function family[b]. The chosen function turns a new input[c] value x_* into the prediction $\hat{y}_{\mathcal{D}_{\text{train}}}(x_*)$. The input value x_* holds one value of each feature. The hat in $\hat{y}_{\mathcal{D}_{\text{train}}}$ marks a function estimated[d] from data.

[a] https://morrison-lab.github.io/mcs/sets.html#def-set
[b] https://morrison-lab.github.io/sds/model-validation.html#def-prediction-function-family
[c] https://morrison-lab.github.io/sds/model-validation.html#def-data-set
[d] https://morrison-lab.github.io/sds/estimation.html#def-estimate

Remark ("Predictor" can name a model)

Remark 1.4 ("Predictor" can name a model). Statistics uses predictor for a measured value of an input[a], a covariate[b]. Some machine learning texts use the same word for a fitted model. Hastie, Tibshirani and Friedman call the averaged forest function "the random forest (regression) predictor" [5], equation 15.2. Murphy calls the function obtained by training on a subset of the data "the predictor" [6], equation 16.104. These notes use predictor only for a measured value of an input.

[a] https://morrison-lab.github.io/sds/model-validation.html#def-data-set
[b] https://morrison-lab.github.io/sds/model-validation.html#def-covariate

Example 1.2 (The parts of the spam cutoff problem)**.** Name the parts of the spam cutoff rule on the Spambase data [7]: the training data (Definition 1.1) are the (x, y) pairs of half the messages, the learning algorithm (Definition 1.2) is a function that searches cutoffs, training (Definition 1.3) is running it, and the model (Definition 1.5) is the rule it returns.

1.2.3 Python

```
import numpy as np
import pandas as pd

spam_tp = pd.read_csv("data/spambase.csv")
n_tp = len(spam_tp)
order_tp = (np.arange(n_tp) * 1009) % n_tp  # scrambled row order, as before
train_tp, new_tp = order_tp[: n_tp // 2], order_tp[n_tp // 2 :]

# Training data: x is the share of "!" characters, y the label (1 = spam).
x_tp = spam_tp["char_freq_!"].to_numpy()
y_tp = spam_tp["Class"].to_numpy()

def learn_cutoff_tp(x, y):
    # The learning algorithm: turn training data into a cutoff.
    grid = np.round(np.arange(0, 3.01, 0.05), 2)
    return grid[np.argmin([np.mean((x > c) != (y == 1)) for c in grid])]

cut_tp = learn_cutoff_tp(x_tp[train_tp], y_tp[train_tp])  # training

def model_tp(x):
    # The model: map a new input x to a prediction.
    return (x > cut_tp).astype(int)

print("cutoff:", cut_tp)
```

```
cutoff: 0.1
```

```
print(pd.DataFrame({"x": x_tp[new_tp[:5]], "prediction": model_tp(x_tp[new_tp[:5]]),
                    "label": y_tp[new_tp[:5]]}))
```

```
       x  prediction  label
0  1.103           1      1
1  0.000           0      0
2  0.000           0      0
3  0.633           1      1
4  0.443           1      1
```

1.2.4 R

```r
spam_tp <- read.csv("data/spambase.csv", check.names = FALSE)
n_tp <- nrow(spam_tp)
order_tp <- ((seq_len(n_tp) - 1) * 1009) %% n_tp + 1  # scrambled row order, as before
train_tp <- order_tp[seq_len(n_tp %/% 2)]
new_tp <- order_tp[-seq_len(n_tp %/% 2)]

# Training data: x is the share of "!" characters, y the label (1 = spam).
x_tp <- spam_tp[["char_freq_!"]]
y_tp <- spam_tp[["Class"]]

learn_cutoff_tp <- function(x, y) {
  # The learning algorithm: turn training data into a cutoff.
  grid <- round(seq(0, 3, by = 0.05), 2)
  grid[which.min(sapply(grid, function(c) mean((x > c) != (y == 1))))]
}

cut_tp <- learn_cutoff_tp(x_tp[train_tp], y_tp[train_tp])  # training

# The model: map a new input x to a prediction.
model_tp <- function(x) as.integer(x > cut_tp)

cat("cutoff:", cut_tp, "\n")
```

```
cutoff: 0.1
```

```r
five_tp <- new_tp[1:5]
data.frame(x = x_tp[five_tp], prediction = model_tp(x_tp[five_tp]), label = y_tp[five_tp])
```

```
# A tibble: 5 x 3
      x prediction label
  <dbl>      <int> <int>
1 1.10           1     1
2 0              0     0
3 0              0     0
4 0.633          1     1
5 0.443          1     1
```

Training on 2300 messages produced the model "spam when ! makes up more than 0.1% of the characters". The model never saw the five messages printed, and it gets 5 of them right. The same algorithm, trained on different data, can return a different model.

Watch

Abu-Mostafa names the components of a learning problem in his first Caltech lecture, from 3:06[a], and then formalizes them, from 12:47[b] [8], Lecture 1.

[a] https://www.youtube.com/watch?v=mbyG85GZ0PI&t=186s
[b] https://www.youtube.com/watch?v=mbyG85GZ0PI&t=767s

1.3 Inputs: feature vectors

Exercise 1.2 (How can an email be an input?)**.** A learning algorithm computes with numbers. How could you describe one email, or one car, so that a learning algorithm can use it as an input?

Solution

Solution 1.2. Measure a list of properties, and write each one as a number: for an email, how often each of several words appears; for a car, its horsepower, weight and number of cylinders.

Definition 1.8 (Feature vector)**.** A **feature vector** $x \in \mathbb{R}^p$ describes one input[a] by p features (Definition 1.4), one per measured property. It is an element of the real coordinate space $\mathbb{R}^p$[b].

[a] https://morrison-lab.github.io/sds/model-validation.html#def-data-set
[b] https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-real-coordinate-space

Example 1.3 (A column and a row of Spambase)**.** In the Spambase table [7], each column except `Class` is a feature (Definition 1.4): `word_freq_free` is the percentage of a message's words that are `free`. A message's feature vector is its row of those columns.

1.3.1 Python

```
import pandas as pd

spam_fv = pd.read_csv("data/spambase.csv")
feats_fv = spam_fv.drop(columns="Class")  # the features; Class is the label y

# One feature: a column, with one value per message.
print(feats_fv["word_freq_free"].head(3))
```

```
0    0.32
1    0.14
2    0.06
Name: word_freq_free, dtype: float64
```

```
# One feature vector: a row, with one value per feature.
x1_fv = feats_fv.iloc[0].to_numpy()
print(x1_fv[:6])
```

```
[0.   0.64 0.64 0.   0.32 0.  ]
```

```
print("D =", len(x1_fv))
```

```
D = 57
```

1.3.2 R

```
spam_fv <- read.csv("data/spambase.csv", check.names = FALSE)
feats_fv <- spam_fv[, names(spam_fv) != "Class"]  # the features; Class is the label y

# One feature: a column, with one value per message.
head(feats_fv[["word_freq_free"]], 3)
```

```
[1] 0.32 0.14 0.06
```

```
# One feature vector: a row, with one value per feature.
x1_fv <- unlist(feats_fv[1, ], use.names = FALSE)
x1_fv[1:6]
```

```
[1] 0.00 0.64 0.64 0.00 0.32 0.00
```

```
cat("D =", length(x1_fv), "\n")
```

```
D = 57
```

The feature `word_freq_free` has 4601 values, one per message. The first message's feature vector has $p = 57$ entries, one per feature, so $x \in \mathbb{R}^{57}$ (Definition 1.8).

Watch

Weinberger introduces feature vectors, from 33:27[a] [9], Lecture 1.

[a] https://www.youtube.com/watch?v=MrLPzBxG95I&t=2007s

Definition 1.9 (Feature function (feature map, encoding))**.** A **feature function** $\underset{\sim}{D}$ sends a covariate vector $\underset{\sim}{x}$, whose entries may be numbers or categories, to its feature vector (Definition 1.8) $\underset{\sim}{D}(\underset{\sim}{x}) \in \mathbb{R}^{p \times 1}$.

Example 1.4 (Encoding a shelf location in the Carseats data)**.** In the Carseats data [10], a store's covariate vector holds its shelf location, a category (`Bad`, `Medium` or `Good`), and its price, a number. The feature function replaces the category with one 0/1 feature per category and keeps the price. The first three stores' feature vectors are the rows below, so p is the number of columns.

```
carseats <- read.csv("data/carseats.csv")[1:3, c("ShelveLoc", "Price")]
carseats$ShelveLoc <- factor(carseats$ShelveLoc, levels = c("Bad", "Medium", "Good"))
carseats
```

```
# A tibble: 3 x 2
  ShelveLoc Price
  <fct>     <int>
1 Bad         120
2 Good         83
3 Medium       80
```

```
# One 0/1 column per shelf location, then the price.
data.frame(model.matrix(~ 0 + ShelveLoc + Price, data = carseats))
```

```
# A tibble: 3 x 4
  ShelveLocBad ShelveLocMedium ShelveLocGood Price
         <dbl>           <dbl>         <dbl> <dbl>
1            1               0             0   120
2            0               0             1    83
3            0               1             0    80
```

Definition 1.10 (Output variable (label, target, response, outcome, dependent variable))**.** The **output variable** is the quantity a supervised model predicts for an input. Each example has one value of it, written y once observed; before it is observed, it is a random variable, written Y. In regression the output is a number, and in classification it is a class.
Other names: the output is also called the **label**, **target**, **response**, **outcome** or **dependent variable**. The term *outcome* is common in statistics and the term *target* in machine learning.

Example 1.5 (The output of Spambase)**.** In the Spambase table [7], the `Class` column is the output variable (Definition 1.10): it is 1 for a message that is spam and 0 for one that is not. The other columns are the features (Definition 1.4).

1.3.3 Python

```
import pandas as pd

spam_out = pd.read_csv("data/spambase.csv")
y_out = spam_out["Class"]  # the output variable: one value per message

# The output is a class, so count the messages in each class.
print(y_out.value_counts())
```

```
Class
0    2788
1    1813
Name: count, dtype: int64
```

1.3.4 R

```r
spam_out <- read.csv("data/spambase.csv", check.names = FALSE)
y_out <- spam_out[["Class"]]  # the output variable: one value per message

# The output is a class, so count the messages in each class.
table(y_out)
```

```
y_out
   0    1
2788 1813
```

Of the 4601 messages, 1813 are spam, so the output takes the value 1 for 39.4% of them.

Most often, each input is a feature vector.

Exercise 1.3 (How long is the feature vector?)**.** What is p for each of these inputs[a]?
An email in the spam data, described by the relative frequencies of 57 words and characters.

[a] https://morrison-lab.github.io/sds/model-validation.html#def-data-set

Solution

Solution 1.3. $p = 57$: one relative frequency per word or character.

Exercise 1.4 (How long is the feature vector? (part 2))**.** Use the setup of Exercise 1.3.
A handwritten ZIP-code digit, scanned as a 16-by-16 grid of grayscale pixels.

Solution

Solution 1.4. One intensity per pixel:

$$\begin{aligned} p &= 16 \times 16 \\ &= 256. \end{aligned}$$

Exercise 1.5 (Does every input fit in a row?)**.** Consider four inputs:

- one car, described by its engine size, weight and fuel economy;
- the text of an email;
- a color photograph;
- a recorded spoken sentence.

Which of them arrive as a fixed list of attributes, and which do not?

Solution

Solution 1.5. Only the car arrives as a fixed list of attributes. Tabular data like it is *structured*; text, images and audio are *unstructured*.

Exercise 1.6 (Does every input fit in a row? (part 2))**.** Use the setup of Exercise 1.5.
For each one that does not, name a form in which a model could receive it.

Solution

Solution 1.6. An unstructured input can be presented to a model in several forms:

- flattened into a p-vector for a classical model, as the scanned digit was in Exercise 1.4;
- kept as a *multi-dimensional array* for a model that takes grid-shaped input directly: a matrix $x \in \mathbb{R}^{D_1 \times D_2}$ for a grayscale image or a spectrogram, or a 3-D array for the color photograph;
- given as a *sequence* of inputs, as for the spoken sentence or the email's words;
- structured as an object, such as a parse tree of the email's sentences.

Definition 1.11 (Structured data)**.** Data are **structured** when every observation is described by the same fixed list of attributes, so a dataset of them fits in a table with one row per observation and one column per attribute.
Example: a list of cars, each described by its engine size, weight and fuel economy.

Definition 1.12 (Unstructured data)**.** Data are **unstructured** when they do not come as a fixed list of attributes: a model receives the raw content, such as words, pixels or audio samples, and any attributes have to be derived from it.
Examples: the text of an email, a photograph, a recorded sentence.

Example 1.6 (The numbers and the names in the Auto table)**.**
The Auto data [10] describe each car by the same fixed list of attributes: `mpg`, `cylinders`, `displacement`, `horsepower`, `weight`, `acceleration`, `year` and `origin`. The last column, `name`, is free text:

1.3.5 Python

```
import pandas as pd

auto_sd = pd.read_csv("data/auto.csv")
# Structured: every car has the same attributes, so the data fit a table.
print(auto_sd.drop(columns="name").head(3))
```

```
    mpg  cylinders  displacement  ...  acceleration  year  origin
0  18.0          8         307.0  ...          12.0    70       1
1  15.0          8         350.0  ...          11.5    70       1
2  18.0          8         318.0  ...          11.0    70       1

[3 rows x 8 columns]
```

```
# Unstructured: the text of a name has no fixed attributes. To use it,
# we derive one, here the make, which we take to be the first word.
make_sd = auto_sd["name"].str.split().str[0]
print(auto_sd["name"].head(3).tolist())
```

```
['chevrolet chevelle malibu', 'buick skylark 320', 'plymouth satellite']
```

```
print(make_sd.value_counts().head(3))
```

```
name
ford         48
chevrolet    43
plymouth     31
```

```
Name: count, dtype: int64
```

1.3.6 R

```r
auto_sd <- read.csv("data/auto.csv")
# Structured: every car has the same attributes, so the data fit a table.
head(auto_sd[, names(auto_sd) != "name"], 3)
```

```
# A tibble: 3 x 8
    mpg cylinders displacement horsepower weight acceleration  year origin
  <dbl>     <int>        <dbl>      <int>  <int>        <dbl> <int>  <int>
1    18         8          307        130   3504         12      70      1
2    15         8          350        165   3693         11.5    70      1
3    18         8          318        150   3436         11      70      1
```

```r
# Unstructured: the text of a name has no fixed attributes. To use it,
# we derive one, here the make, which we take to be the first word.
make_sd <- sapply(strsplit(auto_sd$name, " "), `[`, 1)
head(auto_sd$name, 3)
```

```
[1] "chevrolet chevelle malibu" "buick skylark 320"
[3] "plymouth satellite"
```

```r
head(sort(table(make_sd), decreasing = TRUE), 3)
```

```
make_sd
     ford chevrolet  plymouth
       48        43        31
```

The numeric columns of the 392 cars are structured data (Definition 1.11): every row has the same 8 attributes. The `name` text is unstructured (Definition 1.12) until we derive an attribute from it, and the make we derived takes 37 distinct values. A derived attribute inherits the text's quirks: `chevrolet`, `chevy` and `chevroelt` count as three makes.

An unstructured input can reach a model in several forms: flattened into a p-vector, kept as a multi-dimensional array, given as a sequence, or structured as an object such as a parse tree; Solution 1.5 gives an example of each.

Historically, turning unstructured data (Definition 1.12) into good features was a large part of applied machine learning. Hand-engineered features create an information bottleneck: any attribute left out of the feature vector is lost to the model, and the choice of features reflects what its designers overlooked. Deep learning architectures, such as convolutional and recurrent networks, avoid the bottleneck by taking raw inputs and learning feature representations from them. Categorical inputs, such as a color, are turned into numbers too, for example by one-hot encoding (Harris' 2022 weekly notes).

Exercise 1.7 (Explore a real dataset)**.** The plot shows the cars of the Auto data from the *Introduction to Statistical Learning* labs [10], [11]. Pick what to plot. Which variable would you most want to predict from the others, and which variables would you predict it from?

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

In `origin`, 1 is American, 2 is European and 3 is Japanese.

Solution

Solution 1.7. There is no single right answer. Two natural choices:

- Predict `mpg` from `horsepower`, `weight` and the rest. Plotting `mpg` against `horsepower` (Figure 3) shows `mpg` falling as `horsepower` rises, and along a curve rather than a straight line.
- Predict `origin` from the rest. The colors in Figure 3 show that the European and Japanese cars differ from the American ones in both `mpg` and `horsepower`.

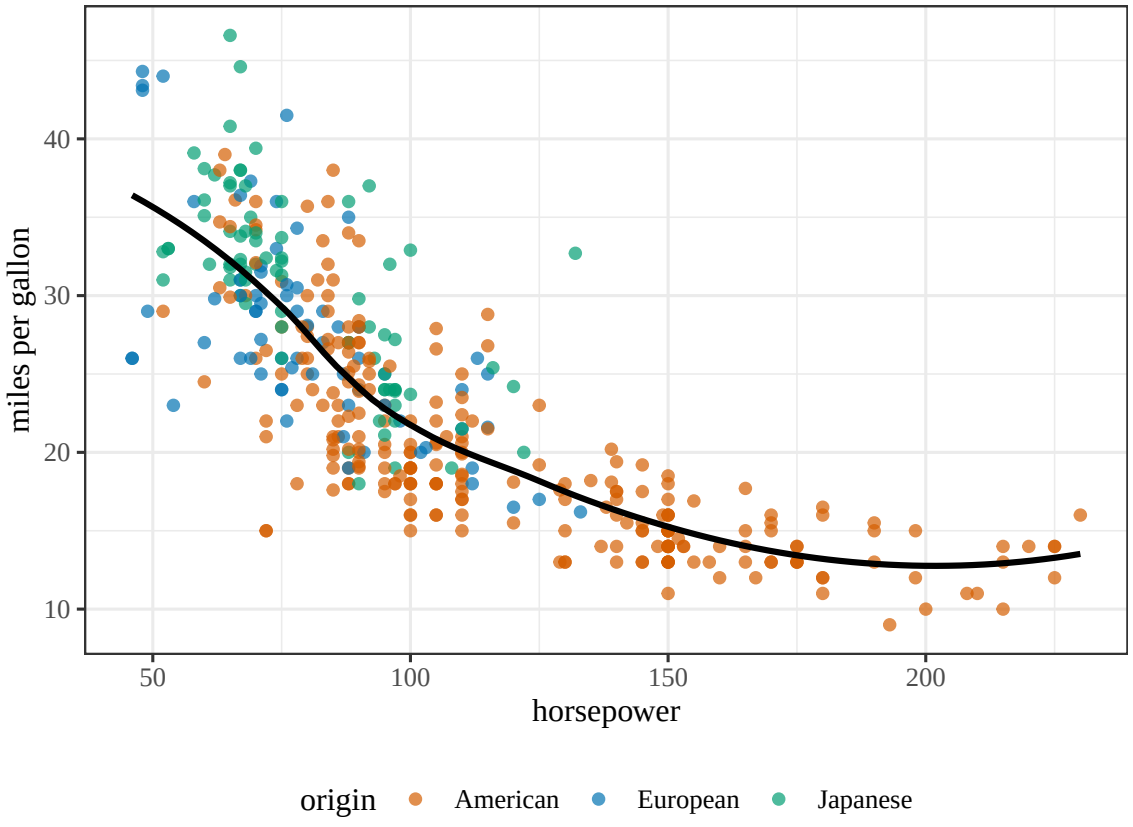


Figure 3: Miles per gallon against horsepower for the cars of the Auto data, colored by `origin`. The black curve is a loess smooth of all the cars.

2 Kinds of learning problems

2.1 Supervised and unsupervised learning

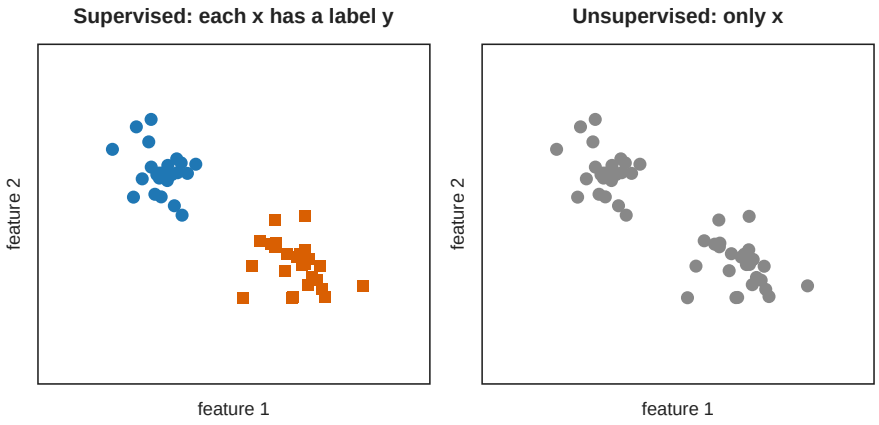


Figure 4: The same fifty points, with labels (left) and without (right). A classifier learns to separate the labeled groups; a clustering algorithm must find the groups without labels.

Exercise 2.1 (What differs between the two panels?)**.** The two panels of Figure 4 show the same fifty points. What does the left panel give a learning algorithm that the right panel does not?

Solution

Solution 2.1. The left panel gives the group of every point, and the right panel gives only the points.

Exercise 2.2 (What differs between the two panels? (part 2))**.** Use the setup of Exercise 2.1. What could the goal be in each of the two panels?

Solution

Solution 2.2. With the group of every point, the goal could be to predict the group of a new point. With only the points, the goal could be to find groups among them.

Definition 2.1 (Supervised learning)**.** In **supervised learning**, every training example is a pair $(\underset{\sim}{x}, y)$: the correct output is given, and the goal is to predict y for new $\underset{\sim}{x}$ [12].
Here y is the output variable (Definition 1.10), which has other names there. A training example is also called an *observation* or an *instance*.

Definition 2.2 (Unsupervised learning)**.** In **unsupervised learning**, the training data are inputs x alone, and the goal is to find structure in them, such as groups, directions of variation, or a density [13].

Example 2.1 (The Spambase table with and without its labels)**.** Take three feature columns of Spambase [7], and the `Class` column that labels each message. With `Class`, the table holds (x, y) pairs. Without it, the table holds inputs x alone.

2.1.1 Python

```
import pandas as pd

spam_su = pd.read_csv("data/spambase.csv")
feats_su = ["word_freq_free", "word_freq_you", "char_freq_!"]

# Supervised: each row is a pair (x, y), and the goal is to predict y from x.
# Comparing the label groups shows which features carry information about y.
print(spam_su[feats_su + ["Class"]].head(3))
```

```
   word_freq_free  word_freq_you  char_freq_!  Class
0            0.32           1.93        0.778      1
1            0.14           3.47        0.372      1
2            0.06           1.36        0.276      1
```

```
print(spam_su.groupby("Class")[feats_su].mean().round(2))
```

```
       word_freq_free  word_freq_you  char_freq_!
Class
0                0.07           1.27         0.11
1                0.52           2.26         0.51
```

```
# Unsupervised: drop the labels and look for structure in x alone,
# here which features rise and fall together.
print(spam_su[feats_su].corr().round(2))
```

```
                word_freq_free  word_freq_you  char_freq_!
```

```
word_freq_free            1.00           0.08         0.10
word_freq_you             0.08           1.00         0.15
char_freq_!               0.10           0.15         1.00
```

2.1.2 R

```r
spam_su <- read.csv("data/spambase.csv", check.names = FALSE)
feats_su <- c("word_freq_free", "word_freq_you", "char_freq_!")

# Supervised: each row is a pair (x, y), and the goal is to predict y from x.
# Comparing the label groups shows which features carry information about y.
head(spam_su[, c(feats_su, "Class")], 3)
```

```
# A tibble: 3 x 4
  word_freq_free word_freq_you `char_freq_!` Class
           <dbl>         <dbl>         <dbl> <int>
1           0.32          1.93         0.778     1
2           0.14          3.47         0.372     1
3           0.06          1.36         0.276     1
```

```r
aggregate(spam_su[, feats_su], by = list(Class = spam_su$Class), FUN = mean)
```

```
# A tibble: 2 x 4
  Class word_freq_free word_freq_you `char_freq_!`
  <int>          <dbl>         <dbl>         <dbl>
1     0         0.0736          1.27         0.110
2     1         0.518           2.26         0.514
```

```r
# Unsupervised: drop the labels and look for structure in x alone,
# here which features rise and fall together.
round(cor(spam_su[, feats_su]), 2)
```

```
               word_freq_free word_freq_you char_freq_!
word_freq_free           1.00          0.08        0.10
word_freq_you            0.08          1.00        0.15
char_freq_!              0.10          0.15        1.00
```

In the supervised view (Definition 2.1), spam averages 0.52% `free` against 0.07% in email, so `free` helps predict the label. In the unsupervised view (Definition 2.2) there is no label to predict: the only finding is structure in the inputs, such as a correlation of 0.08 between `free` and `you`.

Watch

Hastie and Tibshirani present the supervised learning problem[a], then unsupervised learning, from 4:02[b] [14], video 1.2.

[a] https://www.youtube.com/watch?v=B9s8rpdNxU0
[b] https://www.youtube.com/watch?v=B9s8rpdNxU0&t=242s

Read

GeeksforGeeks explains supervised and unsupervised learning[a] [15].

[a] https://www.geeksforgeeks.org/machine-learning/supervised-unsupervised-learning/

Exercise 2.3 (What if labels are scarce, or there are none?)**.** Suppose only a few of the training examples have labels. A harder case: no one gives the correct answer at all, and a game program learns only whether it won. What kind of learning does each case call for?

Solution

Solution 2.3. An algorithm given a few labels could learn from the labeled and the unlabeled examples together. A program given no correct answers could learn from the results of its own moves, counting a win as good and a loss as bad.

Definition 2.3 (Semi-supervised learning)**.** In **semi-supervised learning**, the training data hold a few labeled examples and many unlabeled ones [16]. For a spam filter, the labeled examples are the emails someone has read and marked as *spam* or *email*; the rest of the inbox is unlabeled.

Example 2.2 (A spam filter with 100 labeled messages)**.**
Suppose only 100 of the Spambase messages [7] have been read and marked as spam or email, and the rest of the inbox is unlabeled.

2.1.3 Python

```
import numpy as np
import pandas as pd

spam_ss = pd.read_csv("data/spambase.csv")
n_ss = len(spam_ss)
order_ss = (np.arange(n_ss) * 1009) % n_ss  # scrambled row order, as before

# Hide every label except those of the first 100 messages in the scrambled order.
y_ss = np.array(spam_ss["Class"], dtype=float)  # a writable copy
y_ss[order_ss[100:]] = np.nan
print("labeled:", int(np.sum(~np.isnan(y_ss))), " unlabeled:", int(np.sum(np.isnan(y_ss))))
```

```
labeled: 100  unlabeled: 4501
```

```
# Every message still has its features; only the labeled ones can score a cutoff.
x_ss = spam_ss["char_freq_!"].to_numpy()
lab_ss = ~np.isnan(y_ss)
grid_ss = np.round(np.arange(0, 3.01, 0.05), 2)
errs_ss = [np.mean((x_ss[lab_ss] > c) != (y_ss[lab_ss] == 1)) for c in grid_ss]
cut_ss = grid_ss[np.argmin(errs_ss)]

# Check it against the labels we hid.
truth_ss = spam_ss["Class"].to_numpy()[~lab_ss]
print("cutoff:", cut_ss, " error on the unlabeled:",
      round(np.mean((x_ss[~lab_ss] > cut_ss) != (truth_ss == 1)), 3))
```

```
cutoff: 0.05  error on the unlabeled: 0.217
```

2.1.4 R

```r
spam_ss <- read.csv("data/spambase.csv", check.names = FALSE)
n_ss <- nrow(spam_ss)
order_ss <- ((seq_len(n_ss) - 1) * 1009) %% n_ss + 1  # scrambled row order, as before

# Hide every label except those of the first 100 messages in the scrambled order.
y_ss <- spam_ss$Class
y_ss[order_ss[-seq_len(100)]] <- NA
missing_ss <- is.na(y_ss)
n_unlab_ss <- sum(missing_ss)
n_lab_ss <- length(y_ss) - n_unlab_ss
cat("labeled:", n_lab_ss, " unlabeled:", n_unlab_ss, "\n")
```

```
labeled: 100  unlabeled: 4501
```

```r
# Every message still has its features;
# only the labeled ones can score a cutoff.
x_ss <- spam_ss[["char_freq_!"]]
lab_ss <- !is.na(y_ss)
grid_ss <- round(seq(0, 3, by = 0.05), 2)
errs_ss <- sapply(grid_ss, function(c) mean((x_ss[lab_ss] > c) != (y_ss[lab_ss] == 1)))
cut_ss <- grid_ss[which.min(errs_ss)]

# Check it against the labels we hid.
truth_ss <- spam_ss$Class[!lab_ss]
cat("cutoff:", cut_ss, " error on the unlabeled:",
    round(mean((x_ss[!lab_ss] > cut_ss) != (truth_ss == 1)), 3), "\n")
```

```
cutoff: 0.05  error on the unlabeled: 0.217
```

The training data hold 100 labeled examples and 4501 unlabeled ones (Definition 2.3). This code uses only the labeled 100 to choose the cutoff, so the 4501 unlabeled messages go unused. A semi-supervised method would also learn from their features. The hidden labels were used above only to check the result.

Definition 2.4 (Reinforcement learning)**.** In **reinforcement learning**, an agent acts and learns from rewards rather than from correct answers [17]. For a program learning to play a board game, for example, its actions are its moves, and its reward comes from how its games turn out, not from a label saying which move was right.

Example 2.3 (A filter that learns only from rewards)**.** Let a spam filter act on the Spambase messages [7] one at a time: its action is to send each message to the junk folder or to the inbox, and the only feedback is a reward, +1 for a right call and -1 for a wrong one. Try three cutoff policies, each on its own 500 messages, and keep the one with the highest average reward:

2.1.5 Python

```python
import numpy as np
import pandas as pd

spam_rl = pd.read_csv("data/spambase.csv")
n_rl = len(spam_rl)
order_rl = (np.arange(n_rl) * 1009) % n_rl  # scrambled message order, as before
x_rl = spam_rl["char_freq_!"].to_numpy()
y_rl = spam_rl["Class"].to_numpy()

def reward_rl(cut, rows):
    # Act: junk the message if x > cut. The environment returns +1 or -1;
    # the learner keeps these rewards, not the labels behind them.
    return np.where((x_rl[rows] > cut) == (y_rl[rows] == 1), 1, -1)

for j, cut in enumerate([0.05, 0.5, 1.5]):
    rows = order_rl[500 * j : 500 * (j + 1)]
    print("cutoff", cut, "mean reward:", round(reward_rl(cut, rows).mean(), 3))
```

```
cutoff 0.05 mean reward: 0.576
cutoff 0.5 mean reward: 0.432
cutoff 1.5 mean reward: 0.248
```

2.1.6 R

```r
spam_rl <- read.csv("data/spambase.csv", check.names = FALSE)
n_rl <- nrow(spam_rl)
order_rl <- ((seq_len(n_rl) - 1) * 1009) %% n_rl + 1  # scrambled message order, as before
x_rl <- spam_rl[["char_freq_!"]]
y_rl <- spam_rl$Class

reward_rl <- function(cut, rows) {
  # Act: junk the message if x > cut. The environment returns +1 or -1;
  # the learner keeps these rewards, not the labels behind them.
  ifelse((x_rl[rows] > cut) == (y_rl[rows] == 1), 1, -1)
}

cuts_rl <- c(0.05, 0.5, 1.5)
mean_reward_rl <- sapply(seq_along(cuts_rl), function(j) {
  mean(reward_rl(cuts_rl[j], order_rl[(500 * (j - 1) + 1):(500 * j)]))
})
for (j in seq_along(cuts_rl)) {
  cat("cutoff", cuts_rl[j], "mean reward:", round(mean_reward_rl[j], 3), "\n")
}
```

```
cutoff 0.05 mean reward: 0.576
cutoff 0.5 mean reward: 0.432
cutoff 1.5 mean reward: 0.248
```

The policy with cutoff 0.05 earns the highest average reward, 0.58, so this agent would adopt it (Definition 2.4). This is the simplest case. In a game such as checkers, the reward arrives only when the game ends, and says nothing about which move was the mistake.

Read

GeeksforGeeks explains:

- semi-supervised learning[a] [18]

- reinforcement learning[b] [19]

[a] https://www.geeksforgeeks.org/machine-learning/ml-semi-supervised-learning/
[b] https://www.geeksforgeeks.org/machine-learning/what-is-reinforcement-learning/

Semi-supervised data are common, because labels cost time and money: someone has to read each email to mark it as spam, while unread emails arrive at no cost. The week 10 lecture on reinforcement learning replaces labels with rewards.

Exercise 2.4 (How could a computer find groups?)**.** A dataset has points with no labels, and we want k groups. Propose a procedure that a computer could repeat to split the points into k groups.

Solution

Solution 2.4. Start with a rough guess at the groups, then keep improving the guess:
1. put each point with the group whose middle is closest;
2. recompute each middle;
3. stop once no point changes group.

Definition 2.5 (Clustering)**. Clustering** is the unsupervised task (Definition 2.2) of dividing N inputs $\underset{\sim}{x}_1, \ldots, \underset{\sim}{x}_N$ into groups, called **clusters**, so that inputs in the same cluster are more similar to each other than to inputs in other clusters [20]. The output is a cluster assignment for each input. No labels are given, so the clusters are found, not learned from examples of the right answer.

Example 2.4 (Grouping cars without labels)**.**
Ignore `mpg` and every other column of the Auto data [10] except two: the horsepower and weight of each car. Standardize both, so that neither dominates the distances, and cluster the cars into three groups by two methods, k-means and hierarchical clustering with Ward linkage. Cluster labels are arbitrary, so number the clusters from the lightest cars to the heaviest:

2.1.7 Python

```
import numpy as np
import pandas as pd
from scipy.cluster.hierarchy import fcluster, linkage
from sklearn.cluster import KMeans

auto_cl = pd.read_csv("data/auto.csv").dropna(subset=["horsepower"])
x_cl = auto_cl[["horsepower", "weight"]].to_numpy(float)
x_cl = (x_cl - x_cl.mean(axis=0)) / x_cl.std(axis=0, ddof=1)
w_cl = auto_cl["weight"].to_numpy(float)

def by_weight(labels, weight):
    # Renumber the clusters 0, 1, 2 in order of their mean weight.
    ids, inverse = np.unique(labels, return_inverse=True)
    order = np.argsort([weight[labels == g].mean() for g in ids])
    return np.argsort(order)[inverse]

km_cl = by_weight(KMeans(3, n_init=10, random_state=1).fit(x_cl).labels_, w_cl)
hc_cl = by_weight(fcluster(linkage(x_cl, "ward"), 3, criterion="maxclust"), w_cl)

print(np.bincount(km_cl), np.bincount(hc_cl))
```

```
[159 138  95] [102 187 103]
```

```
print(pd.crosstab(km_cl, hc_cl))
```

```
col_0    0    1   2
row_0
0      102   57   0
1        0  130   8
2        0    0  95
```

2.1.8 R

```
auto_cl <- na.omit(read.csv("data/auto.csv")[, c("horsepower", "weight")])
x_cl <- scale(auto_cl)

# Renumber the clusters 1, 2, 3 in order of their mean weight.
by_weight <- function(labels, weight) {
  rank(tapply(weight, labels, mean))[as.character(labels)]
}

set.seed(1)
km_fit <- kmeans(x_cl, centers = 3, nstart = 10)
km_cl <- by_weight(km_fit$cluster, auto_cl$weight)
hc_cl <- by_weight(cutree(hclust(dist(x_cl), "ward.D2"), k = 3), auto_cl$weight)

rbind(kmeans = table(km_cl), ward = table(hc_cl))
```

```
         1   2   3
kmeans 156 141  95
ward   102 187 103
```

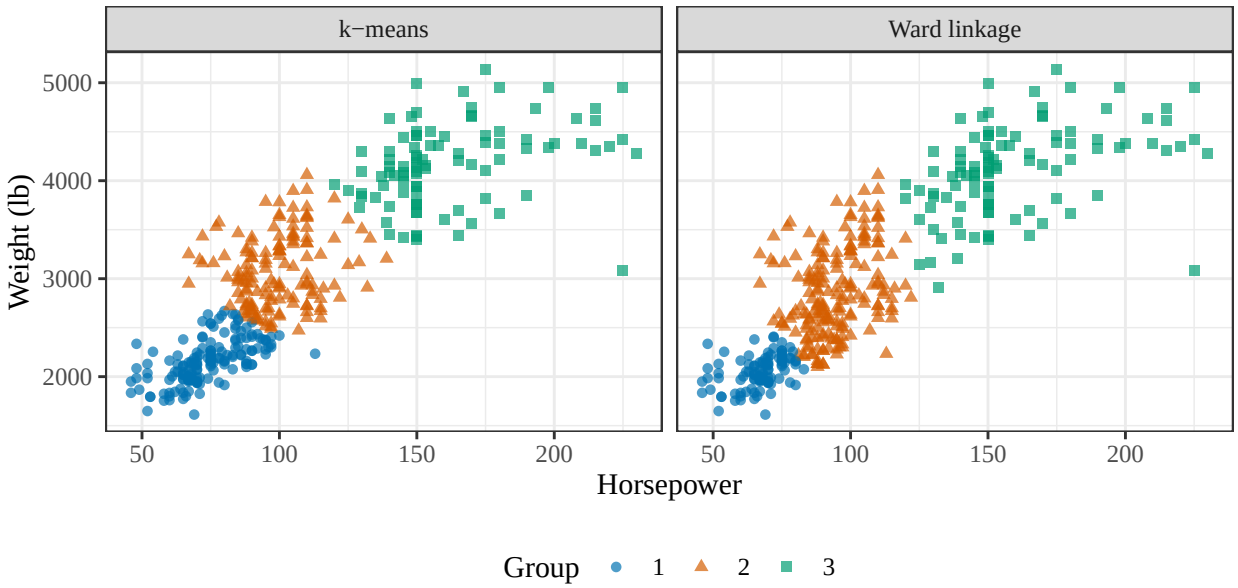
```
table(kmeans = km_cl, ward = hc_cl)
```

```
      ward
kmeans   1   2   3
     1 102  54   0
     2   0 133   8
     3   0   0  95
```

Both methods split the 392 cars into light, middle and heavy groups, and neither was told which cars are which. k-means puts 156, 141 and 95 cars in them, Ward linkage 102, 187 and 103. They agree on 330 cars and differ on the other 62. Each method is an answer to the same task, clustering (Definition 2.5). The two languages start k-means differently, so their counts can differ slightly. The prose gives the R values.

2.1.9 R

```
library(ggplot2)
cl_long <- rbind(
  data.frame(auto_cl, method = "k-means", group = factor(km_cl)),
  data.frame(auto_cl, method = "Ward linkage", group = factor(hc_cl))
)
cl_long$method <- factor(cl_long$method, levels = c("k-means", "Ward linkage"))
ggplot(cl_long, aes(horsepower, weight, colour = group, shape = group)) +
  geom_point(alpha = 0.7) +
  scale_colour_manual(values = c("1" = "#0072B2", "2" = "#D55E00", "3" = "#009E73")) +
  scale_shape_manual(values = c("1" = 16, "2" = 17, "3" = 15)) +
  facet_wrap(~method) +
  labs(x = "Horsepower", y = "Weight (lb)", colour = "Group", shape = "Group")
```

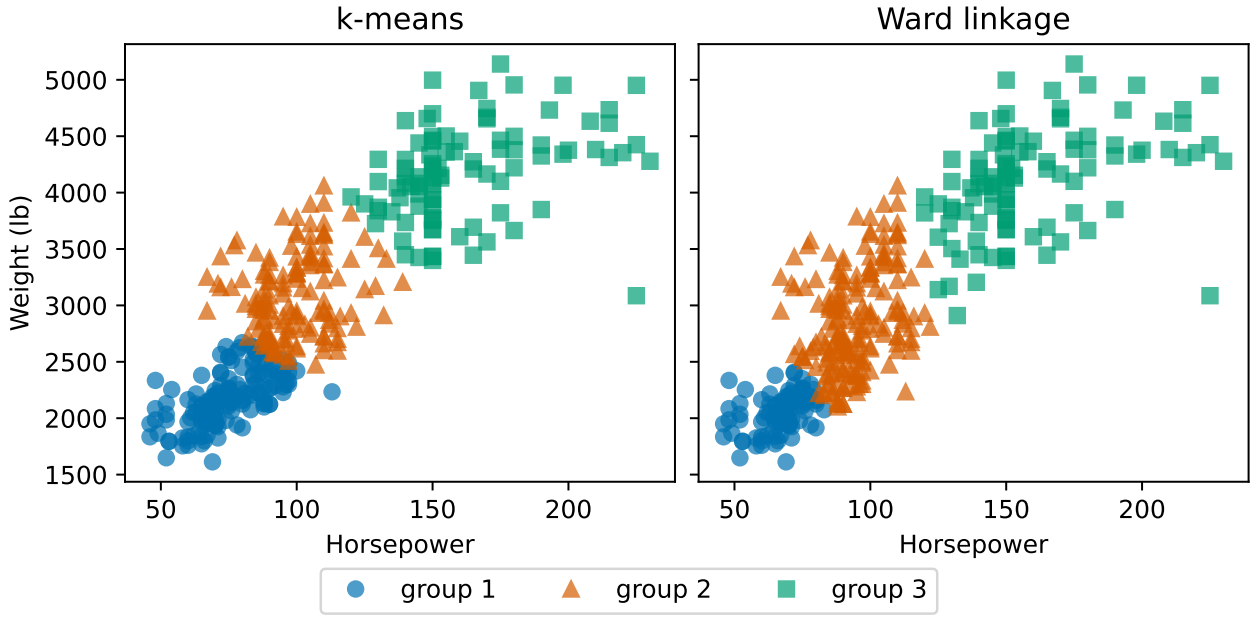

k−means
Ward linkage
Weight (lb)
5000
4000
3000
2000
50
100
150
200
Horsepower
Group
1
2
3

(a)

2.1.10 Python

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(7, 3.5), sharey=True, layout="constrained")
styles_cl = [("#0072B2", "o"), ("#D55E00", "^"), ("#009E73", "s")]
for ax, labels, title in zip(axes, [km_cl, hc_cl], ["k-means", "Ward linkage"]):
    for g, (colour, marker) in enumerate(styles_cl):
        keep = labels == g
        ax.plot(auto_cl["horsepower"][keep], auto_cl["weight"][keep], color=colour,
                marker=marker, linestyle="", alpha=0.7, label=f"group {g + 1}");
    ax.set_title(title);
    ax.set_xlabel("Horsepower");
axes[0].set_ylabel("Weight (lb)");
fig.legend(*axes[0].get_legend_handles_labels(), loc="outside lower center", ncol=3);
plt.show()
```



(b)

Figure 5: The cars of Example 2.4 by horsepower and weight, colored by the group that each method found. Both methods split the cars into light, middle and heavy groups.

> **Remark** (Clustering is a task, not an algorithm)
>
> *Remark* 2.1 (Clustering is a task, not an algorithm). Clustering names the task, not a way of doing it. k-means (Definition 2.6) is one method for it, and hierarchical clustering is another, so the same inputs can be clustered in more than one way.

Definition 2.6 (k-means)**. k-means** is a classic method for clustering (Definition 2.5) the inputs into k groups [21]. It starts with k group centers and repeats two steps until the assignments stop changing:

1. assign each point to its nearest center;
2. move each center to the mean of the points assigned to it.

Example 2.5 (Two groups among the Spambase messages)**.** Run k-means with $k = 2$ on two Spambase features [7], the percentages of `!` and `$` characters, without showing it the labels. Both percentages are heavy-tailed, so the code works with $\log(1 + \text{percentage})$, which keeps a few extreme messages from taking over:

2.1.11 Python

```python
import numpy as np
import pandas as pd

spam_km = pd.read_csv("data/spambase.csv")
feat_km = np.log1p(spam_km[["char_freq_!", "char_freq_$"]].to_numpy())

# Start from the messages with the smallest and the largest feature total.
sums_km = feat_km.sum(axis=1)
centers_km = feat_km[[sums_km.argmin(), sums_km.argmax()]]

for _ in range(100):
    # Step 1: assign each point to its nearest center.
    dist_km = ((feat_km[:, None, :] - centers_km[None, :, :]) ** 2).sum(axis=2)
    group_km = dist_km.argmin(axis=1)
    # Step 2: move each center to the mean of its points; stop when nothing moves.
    new_km = np.array([feat_km[group_km == g].mean(axis=0) for g in range(2)])
    if np.allclose(new_km, centers_km):
        break
    centers_km = new_km

print(centers_km.round(2))
```

```
[[0.06 0.04]
 [0.64 0.16]]
```

```python
# k-means never saw Class; compare its groups with it afterwards.
print(pd.crosstab(group_km, spam_km["Class"], rownames=["group"]))
```

```
Class     0     1
group
0      2652  1025
1       136   788
```

2.1.12 R

```r
spam_km <- read.csv("data/spambase.csv", check.names = FALSE)
feat_km <- log1p(as.matrix(spam_km[, c("char_freq_!", "char_freq_$")]))

# Start from the messages with the smallest and the largest feature total.
sums_km <- rowSums(feat_km)
start_km <- feat_km[c(which.min(sums_km), which.max(sums_km)), ]

# kmeans() repeats the assign and move steps; "Lloyd" is the version
# described in the definition.
fit_km <- kmeans(feat_km, centers = start_km, algorithm = "Lloyd", iter.max = 100)
round(fit_km$centers, 2)
```

```
  char_freq_! char_freq_$
1        0.06        0.04
2        0.64        0.16
```

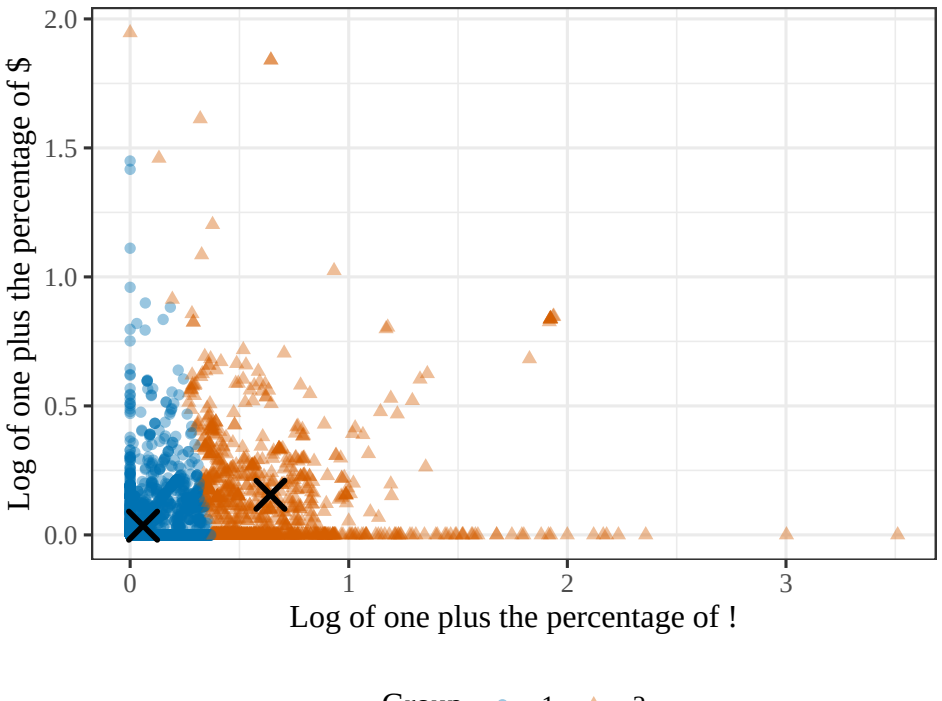
```
# k-means never saw Class; compare its groups with it afterwards.
tab_km <- table(group = fit_km$cluster, Class = spam_km$Class)
tab_km
```

```
     Class
group    0    1
    1 2652 1025
    2  136  788
```

The groups hold 3677 and 924 messages. The method used only the features, as an unsupervised method does (Definition 2.2), yet 85% of the messages in group 2 are spam, against 28% in group 1. Groups found without labels can still line up with the labels.

2.1.13 R

```
library(ggplot2)
km_points <- data.frame(
  excl = feat_km[, 1], dollar = feat_km[, 2],
  group = factor(fit_km$cluster)
)
km_centers <- data.frame(excl = fit_km$centers[, 1], dollar = fit_km$centers[, 2])
ggplot(km_points, aes(excl, dollar, colour = group, shape = group)) +
  geom_point(alpha = 0.4) +
  geom_point(data = km_centers, shape = 4, size = 4, stroke = 1.5, colour = "black") +
  scale_colour_manual(values = c("1" = "#0072B2", "2" = "#D55E00")) +
  scale_shape_manual(values = c("1" = 16, "2" = 17)) +
  labs(
    x = "Log of one plus the percentage of !",
    y = "Log of one plus the percentage of $",
    colour = "Group", shape = "Group"
  )
```

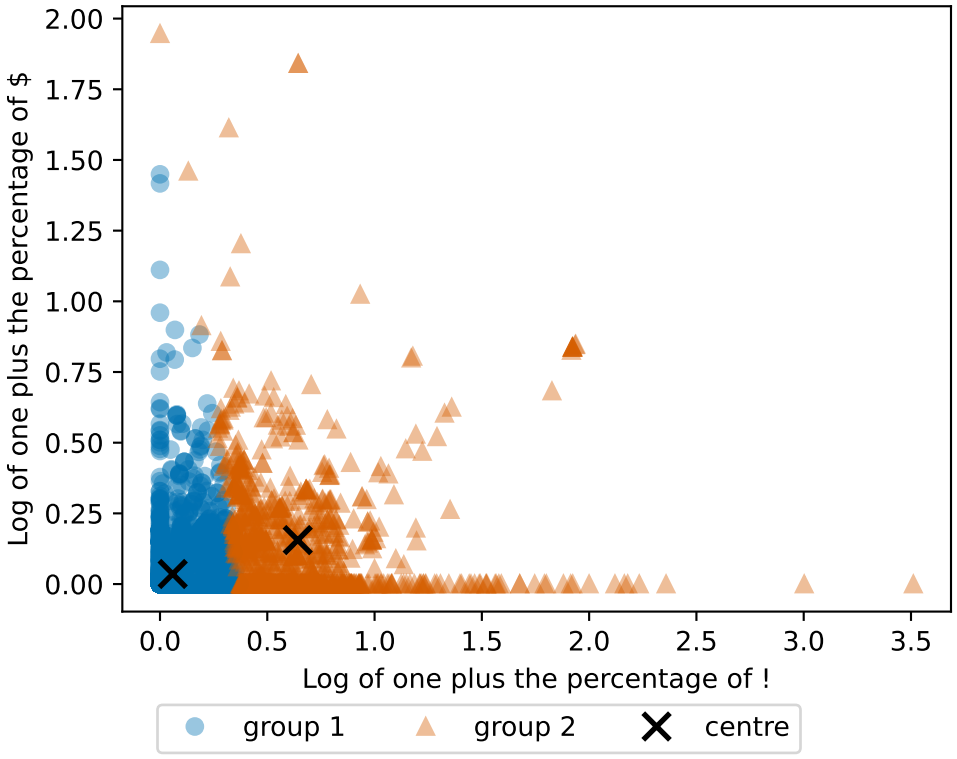


(a)

2.1.14 Python

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(5, 4), layout="constrained")
for g, (colour, marker) in enumerate([("#0072B2", "o"), ("#D55E00", "^")]):
    keep = group_km == g
    ax.plot(feat_km[keep, 0], feat_km[keep, 1], color=colour, marker=marker,
            linestyle="", alpha=0.4, label=f"group {g + 1}");
ax.plot(centers_km[:, 0], centers_km[:, 1], color="black", marker="x", markersize=10,
        markeredgewidth=2, linestyle="", label="centre");
ax.set_xlabel("Log of one plus the percentage of !");
ax.set_ylabel("Log of one plus the percentage of $");
fig.legend(loc="outside lower center", ncol=3);
plt.show()
```



(b)

Figure 6: The two groups that k-means found in Example 2.5, with the two group centers marked by black crosses. The method never saw the class of the messages.

Read

GeeksforGeeks explains k-means clustering[a] [22].

[a] https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

Exercise 2.5 (Let the computer find groups)**.** Here are the cars of the Auto data again, plotted by `weight` and `mpg`, this time with no labels. First, by eye: how many groups do you see, and where are

they?

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

In `origin`, 1 is American, 2 is European and 3 is Japanese.

Solution

Solution 2.5. By eye, most people see two groups: light cars with high `mpg`, and heavy cars with low `mpg`.

Exercise 2.6 (Let the computer find groups (part 2))**.** Use the setup of Exercise 2.5.
Then let k-means (Definition 2.6) look. Choose k, and step through the iterations. At what iteration does it stop changing?

Solution

Solution 2.6. K-means with $k = 2$ agrees and stops changing after one iteration. It finds 217 light cars, averaging 2335 lb and 29 mpg, and 175 heavy ones, averaging 3774 lb and 17 mpg.Larger k also takes longer to stop changing: $k = 3$ is still changing until its ninth iteration.

Exercise 2.7 (Let the computer find groups (part 3))**.** Use the setup of Exercise 2.5.
Then compare its groups with `origin` and with `cylinders`. Which one do the groups match better?

Solution

Solution 2.7. The groups match `cylinders` much better than `origin`. The heavy group holds all 103 eight-cylinder cars, and the light group holds 194 of the 199 four-cylinder cars. By origin, the heavy group is almost all American (163 of 175), but the light group mixes all three: 82 American, 58 European and 77 Japanese cars. European and Japanese cars have much the same weights and `mpg`, so no grouping of these two variables could separate them.
The mismatch with `origin` shows a limitation of unsupervised learning. The algorithm[a] finds structure in the data, and that structure need not match the label we care about. The algorithm always returns k groups, even if the data hold fewer, so choosing k is itself a question. Unsupervised Learning returns to k-means, and to that question.

[a] https://morrison-lab.github.io/mcs/expressions-algorithms.html#def-algorithm

2.2 Regression and classification

Exercise 2.8 (How do outputs differ?)**.** Supervised problems are named for their output. Compare two supervised problems: predicting a house's price, and labeling a message *spam* or *email*. How does the output y differ between them?

Solution

Solution 2.8. A house's price is a number, and a message's label is one of two categories.

Definition 2.7 (Regression)**.** A supervised problem (Definition 2.1) is a **regression** problem when the output y is a quantity, such as a price or a temperature, or a vector of quantities [23].

Definition 2.8 (Classification)**.** A supervised problem (Definition 2.1) is a **classification** problem when y is one of a fixed set of categories, such as *spam* or *email* [24].
With two classes the problem is *binary* classification; with more, it is *multi-class* classification.
Other names: the categories are also called *classes*.

Example 2.6 (Predicting mpg versus predicting spam)**.**
Look at the output y of two problems from this page: a car's `mpg` in the Auto data [10], and a message's `Class` in the Spambase data [7].

2.2.1 Python

```
import pandas as pd

auto_rc = pd.read_csv("data/auto.csv")
spam_rc = pd.read_csv("data/spambase.csv")

# mpg takes many different values along a scale: a quantity.
print(auto_rc["mpg"].describe().round(1))
```

```
count    392.0
mean      23.4
std        7.8
min        9.0
25%       17.0
50%       22.8
75%       29.0
max       46.6
Name: mpg, dtype: float64
```

```
# Class takes just two values, standing for spam and email: a category.
print(spam_rc["Class"].value_counts())
```

```
Class
0    2788
1    1813
Name: count, dtype: int64
```

2.2.2 R

```
auto_rc <- read.csv("data/auto.csv")
spam_rc <- read.csv("data/spambase.csv", check.names = FALSE)

# mpg takes many different values along a scale: a quantity.
summary(auto_rc$mpg)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   9.00   17.00   22.75   23.45   29.00   46.60
```

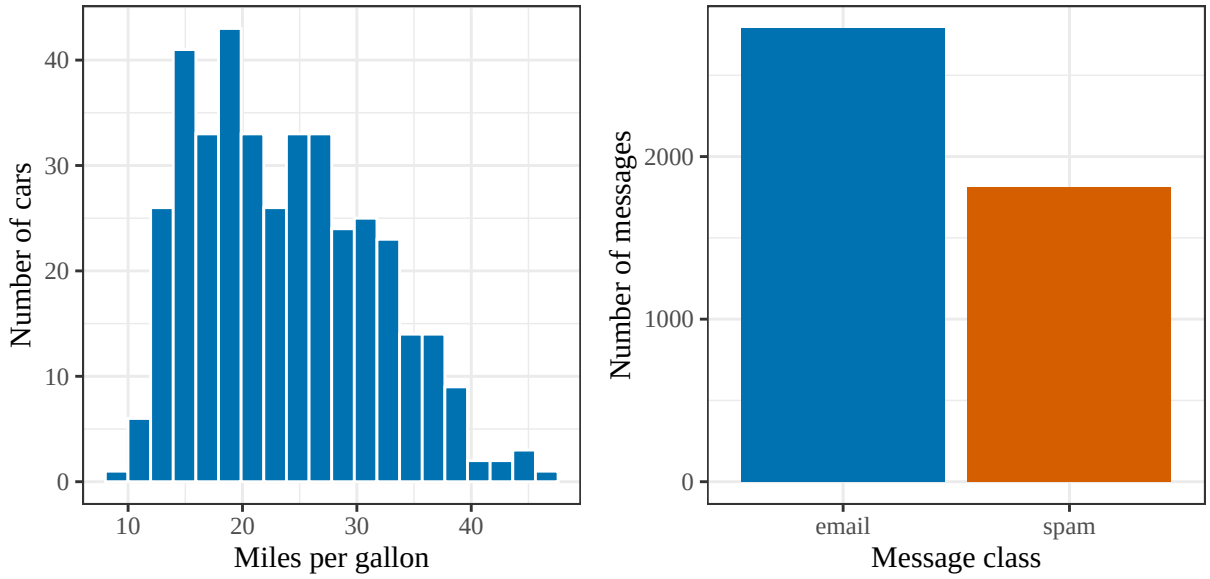
```
# Class takes just two values, standing for spam and email: a category.
table(spam_rc$Class)
```

```
   0    1
2788 1813
```

Predicting `mpg`, which takes 127 distinct values between 9 and 46.6, is a regression problem (Definition 2.7). Predicting `Class`, which takes 2 values, is a classification problem (Definition 2.8), and a binary one because there are two classes. What decides the kind of problem is the output: predicting a car's `origin`, coded 1, 2 or 3 but standing for a country, would be classification too.

2.2.3 R

```
library(ggplot2)
library(patchwork)
rg_mpg <- ggplot(auto_rc, aes(mpg)) +
  geom_histogram(bins = 20, fill = "#0072B2", colour = "white") +
  labs(x = "Miles per gallon", y = "Number of cars")
rg_class <- ggplot(
  data.frame(Class = factor(spam_rc$Class, levels = 0:1, labels = c("email", "spam"))),
  aes(Class, fill = Class)
) +
  geom_bar(show.legend = FALSE) +
  scale_fill_manual(values = c(email = "#0072B2", spam = "#D55E00")) +
  labs(x = "Message class", y = "Number of messages")
rg_mpg + rg_class
```

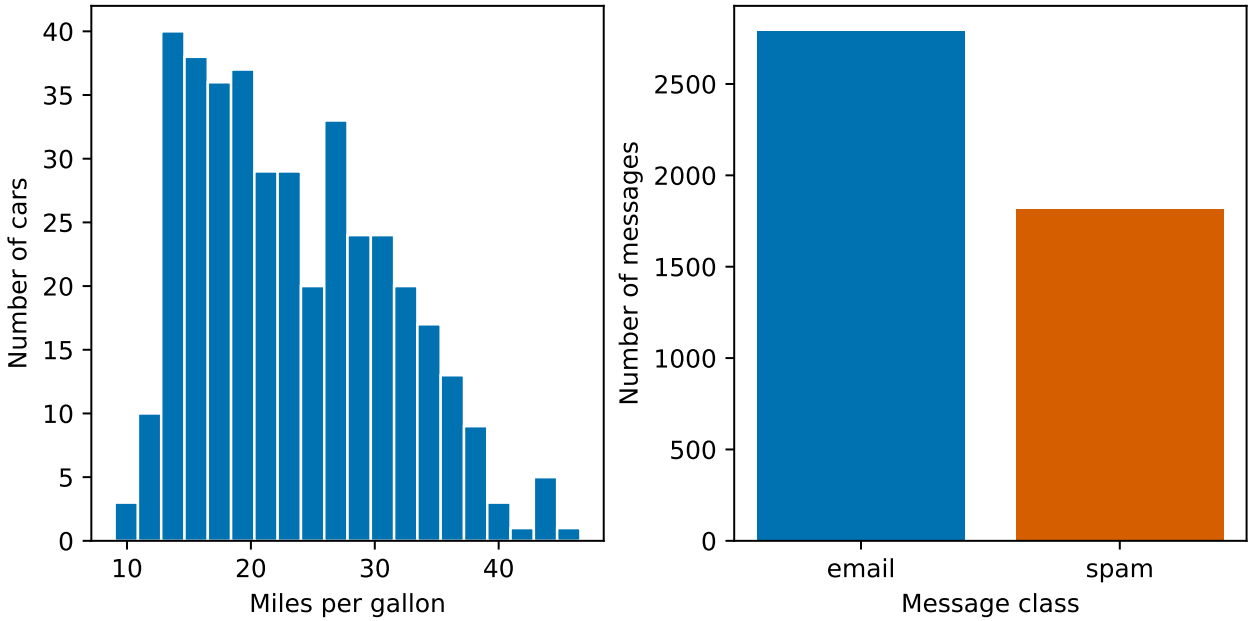


(a)

2.2.4 Python

```
import matplotlib.pyplot as plt

fig, (ax_mpg, ax_class) = plt.subplots(1, 2, figsize=(7, 3.5), layout="constrained")
ax_mpg.hist(auto_rc["mpg"], bins=20, color="#0072B2", edgecolor="white");
ax_mpg.set_xlabel("Miles per gallon");
ax_mpg.set_ylabel("Number of cars");
counts_rg = spam_rc["Class"].value_counts().sort_index()
ax_class.bar(["email", "spam"], counts_rg.to_numpy(), color=["#0072B2", "#D55E00"]);
ax_class.set_xlabel("Message class");
ax_class.set_ylabel("Number of messages");
plt.show()
```



(b)

Figure 7: On the left, the output of a regression takes many values along a scale, such as the `mpg` of cars; on the right, the output of a classification takes just a few, such as the two `Class` values of messages (Example 2.6).

> **Read**
>
> GeeksforGeeks explains classification and regression[a] [25].
>
> [a] https://www.geeksforgeeks.org/machine-learning/ml-classification-vs-regression/

The spam filter is a classification problem with two classes. Predicting the log of a prostate-cancer antigen level in the blood from clinical measurements is a regression problem. So is predicting a house's price from its living area, using 47 houses in Portland, Oregon. Other output types exist, such as a ranking of search results by relevance or a sequence of part-of-speech tags, one for each word of a sentence, but these two are the focus of most of this course.

> **Exercise 2.9** (Regression or classification?)**.** Is predicting `mpg` from the other Auto data variables a regression or a classification problem?

Solution

Solution 2.9. Predicting `mpg` is **regression**, because `mpg` is a quantity.

Exercise 2.10 (Regression or classification? (part 2))**.** Use the setup of Exercise 2.9.
Is predicting `origin` a regression or a classification problem?

Solution

Solution 2.10. Predicting `origin` is **classification**, because `origin` is one of three categories, even though it is stored as the numbers 1, 2 and 3.

2.3 A first regression model

Exercise 2.11 (What is the simplest rule?)**.** We want a rule that takes a car's `horsepower` and returns a predicted `mpg`.
What is the simplest shape such a rule can take?

Solution

Solution 2.11. A straight line,

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x,$$

where - x is a car's horsepower (`horsepower`); - $\hat{y}$ is its predicted `mpg`; - $\hat{\beta}_0$ is the intercept[a]; - $\hat{\beta}_1$ is the slope[b].

[a] https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept
[b] https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope

Exercise 2.12 (What is the simplest rule? (part 2))**.** Use the setup of Exercise 2.11.
How many numbers do we have to choose to specify that rule exactly?

Solution

Solution 2.12. Two: $\hat{\beta}_0$ and $\hat{\beta}_1$. To choose them, we need a way to score a line.

Exercise 2.13 (How could we score a line?)**.** The line's error on each car is $y - \hat{y}$. Propose one number that scores the line on n cars, so that a lower score means a better line.

Solution

Solution 2.13. Square each car's error, so that errors above and below the line both increase its total, and average the squares over the n cars.

Definition 2.9 (Sample mean squared error)**.** The **sample mean squared error** (sMSE) of a fitted model on n examples $(x_1, y_1), \ldots, (x_n, y_n)$ is the average squared error [26],

$$\text{sMSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

where $\hat{y}_i$ is the fitted model's prediction for example i.

Remark (The sMSE estimates the risk)

Remark 2.2 (The sMSE estimates the risk). The sMSE is a sample estimate[a] of the expected[b] squared error on new examples, the risk under squared error.

[a] https://morrison-lab.github.io/sds/estimation.html#def-estimate
[b] https://morrison-lab.github.io/pds/expectation.html#def-expectation

Example 2.7 (A baseline's sample mean squared error on Auto)**.**
The simplest rule for predicting `mpg` in the Auto data [10] ignores the car and always guesses the sample mean `mpg`. Compute its sMSE, and compare it with the sMSE of a line that uses `horsepower`:

2.3.1 Python

```
import numpy as np
import pandas as pd

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

# Baseline: predict the mean mpg for every car.
mse_base_ms = np.mean((y_ms - y_ms.mean()) ** 2)

# Line: predict from horsepower, with the least squares fit.
slope_ms, intercept_ms = np.polyfit(auto_ms["horsepower"], y_ms, 1)
pred_ms = intercept_ms + slope_ms * auto_ms["horsepower"].to_numpy()
mse_line_ms = np.mean((y_ms - pred_ms) ** 2)

print("baseline sMSE:", round(mse_base_ms, 1))
```

```
baseline sMSE: 60.8
```

```
print("line sMSE:    ", round(mse_line_ms, 1))
```

```
line sMSE:     23.9
```

2.3.2 R

```
auto_ms <- read.csv("data/auto.csv")
y_ms <- auto_ms$mpg

# Baseline: predict the mean mpg for every car.
mse_base_ms <- mean((y_ms - mean(y_ms))^2)

# Line: predict from horsepower, with the least squares fit.
fit_ms <- lm(mpg ~ horsepower, data = auto_ms)
mse_line_ms <- mean((y_ms - fitted(fit_ms))^2)

cat("baseline sMSE:", round(mse_base_ms, 1), "\n")
```

```
baseline sMSE: 60.8
```

```
cat("line sMSE:    ", round(mse_line_ms, 1), "\n")
```

```
line sMSE:     23.9
```

On these 392 cars the baseline's sMSE (Definition 2.9) is 60.8, in squared miles per gallon, and the line's is 23.9. The line's sMSE cannot be larger, because the baseline is a line with slope 0. The line's sMSE is smaller by a factor of 2.54.

Definition 2.10 (Sample root mean squared error)**.** The **sample root mean squared error** (sRMSE) of a fitted model on n examples is the square root of its sMSE (Definition 2.9) [27],

$$\text{sRMSE} = \sqrt{\text{sMSE}}.$$

Remark (The sRMSE has the unit of the outcome)

Remark 2.3 (The sRMSE has the unit of the outcome). Squaring the errors squares their unit, and the square root returns the sRMSE to the unit of the outcome. The sRMSE ranks fitted models in the same order as the sMSE (Definition 2.9), because the square root is increasing. We therefore report the sRMSE, and later say more about why.

Example 2.8 (A baseline's sample root mean squared error on Auto)**.** Take the square root of the two sMSEs from Example 2.7:

2.3.3 Python

```
print("baseline sRMSE:", round(np.sqrt(mse_base_ms), 2))
```

```
baseline sRMSE: 7.8
```

```
print("line sRMSE:    ", round(np.sqrt(mse_line_ms), 2))
```

```
line sRMSE:     4.89
```

2.3.4 R

```
rmse_base_ms <- sqrt(mse_base_ms)
rmse_line_ms <- sqrt(mse_line_ms)
cat("baseline sRMSE:", round(rmse_base_ms, 2), "\n")
```

```
baseline sRMSE: 7.8
```

```
cat("line sRMSE:    ", round(rmse_line_ms, 2), "\n")
```

```
line sRMSE:     4.89
```

The baseline's sRMSE (Definition 2.10) is 7.80 miles per gallon, the same unit as `mpg`, and the line's is 4.89. Using `horsepower` shrinks the typical error by a factor of 1.59.

Watch

Hutchinson explains a model, training and a loss step by step, Machine Learning Example[a] (21 min) [28].

[a] https://facultyweb.cs.wwu.edu/~hutchib2/video_lectures/data371/#ml_example

Read

GeeksforGeeks explains mean squared error[a] [29].

[a] https://www.geeksforgeeks.org/maths/mean-squared-error/

Squaring makes every error increase the rule's total, and makes big errors count much more than small ones.

Exercise 2.14 (Score a line by hand)**.** Take the line $\hat{y} = 40 - 0.16x$ and the first three cars in `Auto`:

Table 1: The first three cars, with horsepower and fuel economy.

car	horsepower x	mpg y
chevrolet chevelle malibu	130	18
buick skylark 320	165	15
plymouth satellite	150	18

For each car, find the prediction $\hat{y}$ and the error $y - \hat{y}$. Then find the line's sMSE (Definition 2.9) on these three cars, and take its square root to get the sRMSE (Definition 2.10).

Solution

Solution 2.14.

Table 2: The line's value, its gap and the gap multiplied by itself for each of the three cars.

car	$\hat{y} = 40 - 0.16x$	error $y - \hat{y}$	squared
chevrolet chevelle malibu	$40 - 20.8 = 19.2$	-1.2	1.44
buick skylark 320	$40 - 26.4 = 13.6$	1.4	1.96
plymouth satellite	$40 - 24.0 = 16.0$	2.0	4.00

The average of the squared errors is

$$\frac{1.44 + 1.96 + 4.00}{3} = \frac{7.40}{3} \approx 2.47.$$

That average is the line's sMSE on these three cars (Definition 2.9). Its square root, $\sqrt{7.40/3} \approx 1.57$ miles per gallon, is the line's sRMSE (Definition 2.10).

Exercise 2.15 (Fit a line yourself)**.** Now score lines on all 392 cars at once. Move the intercept and slope to make the sRMSE (Definition 2.10) as small as you can. What is the lowest sRMSE you can reach?

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

Solution

Solution 2.15.
A line fitted by least squares has the lowest possible sRMSE, 4.89 miles per gallon. It is

$$\hat{y} = 39.94 - 0.1578x,$$

the least-squares line (tick *least-squares line* to see it). Each extra unit of horsepower predicts about 0.16 fewer miles per gallon.
The line from Exercise 2.14, $40 - 0.16x$, has an sRMSE of 4.90 on all 392 cars, within 0.1% of the best. Finding the best line does not require a slider: linear regression derives a closed-form solution[a] for it, and gradient descent finds it by repeatedly making small changes that lower the sMSE, much as you just did by hand.

[a] https://morrison-lab.github.io/mcs/expressions-algorithms.html#def-closed-form-solution

Exercise 2.16 (Fit a line yourself (part 2))**.** Use the setup of Exercise 2.15.
How does the lowest sRMSE you reached compare with the sRMSE of ignoring horsepower and pre-

dicting the sample mean[a] `mpg` for every car?

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean

Solution

Solution 2.16.
Predicting the sample mean[a], 23.4 mpg, for every car has an sRMSE of 7.80 miles per gallon. The best line, with an sRMSE of 4.89, cuts the typical error by more than a third.

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean

Exercise 2.17 (Is a straight line the right shape?)**.** Look at where the cars are relative to the least-squares line, at low, middle and high horsepower. Is there a pattern in the errors?

Solution

Solution 2.17. The errors have a pattern. The line predicts too low at both ends, below about 80 and above about 170 horsepower, and too high in the middle, because `mpg` falls steeply at first and then flattens.

Exercise 2.18 (Is a straight line the right shape? (part 2))**.** Use the setup of Exercise 2.17.
Then tick *least-squares curve*, which fits $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2$ by least squares. Should we always prefer the model with the lower sRMSE (Definition 2.10)?

Solution

Solution 2.18.
The curve

$$\hat{y} = 56.90 - 0.4662x + 0.001231x^2$$

follows that bend, and its sRMSE is 4.36, against 4.89 for the line.
But a lower sRMSE on the data we fit is not the goal. A curve with enough terms could pass through every car and reach an sRMSE of zero, and still predict the next car badly. Minimizing sRMSE on the data we fit repeats the mistake of trusting the best spam rule's score on half A. The error that matters is the error on data the rule has not seen, which is generalization, the subject of A First Model and Generalization.

Watch

Abu-Mostafa sorts learning problems into supervised, unsupervised and reinforcement learning in his first Caltech lecture, from 36:04[a] [8], Lecture 1.

[a] https://www.youtube.com/watch?v=mbyG85GZ0PI&t=2164s

3 Learning a game from self-play

3.1 Scoring a checkers position

A program that plays checkers has to choose a move on every turn. One way to choose is to look at the positions each move could lead to, score each position, and pick the move whose position scores best.

Exercise 3.1 (Describe a checkers position with numbers)**.** Suppose the program plays Black. List four numbers you could compute from a checkers board that say whether the position is good for Black.

Hint

Think about what a checkers player counts or looks at before deciding who is ahead.

Solution

Solution 3.1. Four good choices:

- piece advantage: Black's pieces minus White's pieces, with a king counted as more than a man;
- mobility: the number of squares Black could move a piece to;
- center control: the number of Black pieces on the eight central squares;
- advancement: how far Black's men have moved toward the far row, where they become kings.

The IBM checkers program used all four, from a list of 38 such features plus piece advantage [30], "Term replacement".

Exercise 3.2 (What does scoring a position take?)**.** To pick a move, the program scores each position it could reach. What goes into that scoring rule, and what comes out?

Solution

Solution 3.2. A board position goes in, and one number comes out. A bigger number should mean the position is better for Black. So the scoring rule is a function from positions to numbers.

Definition 3.1 (Evaluation function)**.** An **evaluation function** V assigns a number $V(s)$ to each game position s, with a larger number meaning the position is better for the player it scores [31].

Example 3.1 (Scoring a position by who has more pieces)**.** Score a position s from Black's side with

$$V(s) = \begin{cases} 1 & \text{if Black has more pieces than White,} \\ 0 & \text{if the two sides have the same number of pieces,} \\ -1 & \text{if Black has fewer pieces than White.} \end{cases}$$

A position s where Black has 7 pieces and White has 5 scores $V(s) = 1$.

Exercise 3.3 (Combine the features into one score)**.** How could the program combine the numbers from Exercise 3.1 into a single score for a position?

Solution

Solution 3.3. Give each feature a weight, multiply each feature by its weight, and add up the products. A feature that matters more for winning gets a larger weight.

Definition 3.2 (Linear evaluation function)**.** A **linear evaluation function** is an evaluation function (Definition 3.1) that is a weighted sum, the dot product[a] of a weight vector with a feature vector (Definition 1.8) describing the position,

$$V_w(s) = \sum_{j=1}^{p} w_j x_j(s),$$

where $x_1(s), \ldots, x_p(s)$ are the features of s, and the numbers $w_1, \ldots, w_p$ are its weights.

[a] https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-dot-product

Example 3.2 (A two-feature evaluation function)**.** Take two features, $x_1(s)$, Black's piece advantage, and $x_2(s)$, Black's mobility advantage (the number of squares Black can move a piece to, minus White's number), with weights $w_1 = 3$ and $w_2 = 1$. A position s where Black is ahead by one piece and can move to 5 more squares than White scores

$$\begin{aligned} V_w(s) &= 3 \times 1 + 1 \times 5 \\ &= 8. \end{aligned}$$

The features say what the program looks at. The weights say how much each feature matters, and the program can learn the weights.

Exercise 3.4 (Where can the weights come from?)**.** Name three ways to choose the weights $w_1, \ldots, w_p$ of a linear evaluation function for checkers (Definition 3.2).

Solution

Solution 3.4.

1. A person sets them by hand, from experience of the game. No learning happens.
2. The program learns them from games between expert players, choosing weights whose top-scored move tends to match the move the expert made.
3. The program learns them from games it plays against itself.

Exercise 3.5 (Where can the weights come from? (part 2))**.** Use the setup of Exercise 3.4. Which of your three ways count as machine learning?

Solution

Solution 3.5. Ways 2 and 3 are machine learning: the program's play improves with experience, and the experience is data, not rules a person wrote down. The IBM checkers program learned its weights by way 3 in 1959 [30]. A 1967 follow-up reported an improved "book learning" procedure, which Samuel judged "much superior to the linear polynomial[a] method" [32], abstract.

[a] https://morrison-lab.github.io/mds/algebra-exponentials.html#def-polynomial

3.2 Learning from self-play

Nobody tells a program playing against itself how good a position really is, or which move was correct. Learning from self-play is a kind of reinforcement learning (Definition 2.4), and the program has to make its own targets.

Exercise 3.6 (Score a position by looking ahead)**.** The evaluation function scores a position from its features alone. How could the program use the evaluation function to get a more reliable score for a position s, by looking several moves ahead of s?

Solution

Solution 3.6. Look at every position the game could reach from s within a few moves, and score each of those positions with the evaluation function. Then work backwards, assuming each player picks the move that is best for that player.

Definition 3.3 (Backed-up score)**.** The **backed-up score** of a position s looks several moves ahead: it applies the evaluation function to every position reachable from s within a fixed number of moves, then assumes each side picks the move best for itself, working back one move at a time until it reaches s [33].

Example 3.3 (A two-move backed-up score)**.**
In position s, Black is to move and has two moves, A and B. White then has two replies to each. The evaluation function scores the four resulting positions for Black:

Table 3: The four resulting positions of the two-move game, with the value of each for Black.

Black's move	White's reply	score for Black
A	A1	5
A	A2	1
B	B1	3
B	B2	2

White picks the reply that is worst for Black, so move A is worth $\min(5, 1) = 1$ and move B is worth $\min(3, 2) = 2$. Black picks the better of the two moves, so the backed-up score of s is $\max(1, 2) = 2$.

Exercise 3.7 (A target without labels)**.** The program has two ways to score a position s: the evaluation function $V_w(s)$ (Definition 3.2), and the backed-up score of s (Definition 3.3). Which of the two should the program trust more?

Solution

Solution 3.7. The backed-up score is more trustworthy.

Exercise 3.8 (A target without labels (part 2))**.** Use the setup of Exercise 3.7.
Why should the program trust that one more?

Solution

Solution 3.8. The backed-up score has seen more of the game: it scores positions several moves further on, where more of the outcome is already settled.

Exercise 3.9 (A target without labels (part 3))**.** Use the setup of Exercise 3.7.
How could the program use the one it trusts more to improve the weights w?

Solution

Solution 3.9. So the program can treat the backed-up score as the target, and change the weights until $V_w(s)$ comes closer to it.

> we are attempting to make the score, calculated for the current board position, look like that calculated for the terminal board position of the chain of moves which most probably will occur during actual play.

[30], "Polynomial modification procedure"

Exercise 3.10 (Score a position and find the gap)**.** Use the two-feature evaluation function of Example 3.2, with a new position s and these values:

Table 4: The symbols, meanings and values for the position.

symbol	meaning	value
$x_1(s)$	Black's piece advantage in s	2
$x_2(s)$	Black's mobility advantage in s	-1
w_1, w_2	current weights	$1, 1$
t	backed-up score of s (Definition 3.3)	3

Find the score $V_w(s)$ and the gap $\delta = t - V_w(s)$ between the target and the score.

Solution

Solution 3.10.

$$\begin{aligned} V_w(s) &= w_1 x_1(s) + w_2 x_2(s) && \text{(definition of } V_w\text{)} \\ &= 1 \times 2 + 1 \times (-1) && \text{(substitute the values)} \\ &= 1, \\ \delta &= t - V_w(s) && \text{(definition of } \delta\text{)} \\ &= 3 - 1 && \text{(substitute the values)} \\ &= 2. \end{aligned}$$

The score is too low by 2.

Exercise 3.11 (Change the weights slightly)**.** One rule for changing the weights is

$$w_j \leftarrow w_j + \eta\, \delta\, x_j(s),$$

where the step size η controls how far each weight moves. Use $\eta = 0.1$ and your δ from Exercise 3.10 to find the new weights w_1 and w_2.

Solution

Solution 3.11.

$$\begin{aligned} w_1 &\leftarrow w_1 + \eta\, \delta\, x_1(s) && \text{(the update rule, } j = 1\text{)} \\ &= 1 + 0.1 \times 2 \times 2 && \text{(substitute the values)} \\ &= 1.4, \\ w_2 &\leftarrow w_2 + \eta\, \delta\, x_2(s) && \text{(the update rule, } j = 2\text{)} \\ &= 1 + 0.1 \times 2 \times (-1) && \text{(substitute the values)} \\ &= 0.8. \end{aligned}$$

The score was too low ($\delta > 0$), so the weight on the feature that raised the score ($x_1 > 0$) grew, and the weight on the feature that lowered the score ($x_2 < 0$) shrank.

Exercise 3.12 (Check the new score)**.** Find the new score $V_w(s)$ with the new weights from Exercise 3.11.

Solution

Solution 3.12.

$$\begin{aligned} V_w(s) &= w_1 x_1(s) + w_2 x_2(s) && \text{(definition of } V_w\text{)} \\ &= 1.4 \times 2 + 0.8 \times (-1) && \text{(substitute the new weights)} \\ &= 2.8 - 0.8 && \text{(multiply)} \\ &= 2.0. \end{aligned}$$

Exercise 3.13 (Check the new score (part 2))**.** Use the setup of Exercise 3.12. Did the score move toward the target $t = 3$?

Solution

Solution 3.13. The new score, 2.0, is closer to $t = 3$ than the old score of 1.

The IBM program followed the same principle with a different rule. It kept, for each feature, a correlation between the sign of that feature's contribution to the score and the sign of δ, and set the weights to powers of 2 in proportion to those correlations. It also swapped the feature with the weakest correlation for one from a reserve list, about once every eight moves [30], "Polynomial modification procedure" and "Term replacement". The rule in Exercise 3.11 is a later, simpler one.

Exercise 3.14 (Remember positions you have scored)**.** Suppose the program looks 3 moves ahead, and it saves every position it meets in play, together with that position's backed-up score (Definition 3.3). The 3-move lookahead in a later game reaches one of the saved positions. How far ahead does the saved score effectively look?

Solution

Solution 3.14. The saved score already looks 3 moves past the saved position, and the saved position is 3 moves past the current one, so the score in use looks $3 + 3 = 6$ moves ahead. The memory grows with every game, and the effective lookahead grows with the memory.
The IBM program saved over 53,000 positions this way. It learned to play good openings and endgames, though its midgame play barely improved [30], "Rote-learning tests".

4 Going further

The rest of this page is optional: we do it in class if there is time, and otherwise it is here for you to work through on your own.

4.1 More on the spam data

Exercise 4.1 (Look at the spam data)**.** The spam data are the Spambase[a] set, collected at Hewlett-Packard Labs and published in the UCI Machine Learning Repository [7]. The same 4601 messages also ship as `spam` in R's `kernlab` package [34], and Python's `ucimlrepo` package downloads them from UCI. These notes keep a copy of the file, with UCI's column names, at `data/spambase.csv`[b], so each spam exercise in this section that loads the data reads the same file.
Load it.

[a] https://archive.ics.uci.edu/dataset/94/spambase
[b] https://morrison-lab.github.io/lds/chapters/data/spambase.csv

Solution

Solution 4.1.

4.1.1 Python

```
import pandas as pd

spam = pd.read_csv("data/spambase.csv")
```

4.1.2 R

```
spam <- read.csv("data/spambase.csv", check.names = FALSE)
```

Exercise 4.2 (Look at the spam data (part 2))**.** Use the setup of Exercise 4.1.
Look at the `george`, `you`, `free` and `hp` columns.

Solution

Solution 4.2.

4.1.3 Python

```
print(spam[["word_freq_george", "word_freq_you",
            "word_freq_free", "word_freq_hp"]].head())
```

```
   word_freq_george  word_freq_you  word_freq_free  word_freq_hp
0               0.0           1.93            0.32           0.0
1               0.0           3.47            0.14           0.0
2               0.0           1.36            0.06           0.0
3               0.0           3.18            0.31           0.0
4               0.0           3.18            0.31           0.0
```

4.1.4 R

```
spam |>
  dplyr::select(word_freq_george, word_freq_you,
                word_freq_free, word_freq_hp) |>
  str()
```

```
'data.frame':   4601 obs. of  4 variables:
 $ word_freq_george: num  0 0 0 0 0 0 0 0 0 0 ...
 $ word_freq_you   : num  1.93 3.47 1.36 3.18 3.18 0 3.85 0 1.23 1.67 ...
 $ word_freq_free  : num  0.32 0.14 0.06 0.31 0.31 0 0.96 0 0 0 ...
 $ word_freq_hp    : num  0 0 0 0 0 0 0 0 0 0 ...
```

Most entries are 0: in any one message, most of the 57 words never appear.

Exercise 4.3 (Look at the spam data (part 3))**.** Use the setup of Exercise 4.1.
Count how many messages are spam.

Solution

Solution 4.3.

4.1.5 Python

```
print(spam["Class"].value_counts())
```

```
Class
0    2788
1    1813
Name: count, dtype: int64
```

4.1.6 R

```
table(spam$Class)
```

```
   0    1
2788 1813
```

```
n_msg <- nrow(spam)
n_spam <- sum(spam$Class == 1)
n_email <- sum(spam$Class == 0)
```

There are 4601 messages: 1813 spam and 2788 email. The file codes spam as 1 and email as 0.

Exercise 4.4 (Read the file with R's defaults)**.** UCI names six of Spambase's columns after punctuation marks: `char_freq_;`, `char_freq_(` and so on. Read `data/spambase.csv`[a] into R with `read.csv()` and no other arguments, and print the names of columns 49 to 54.

[a] https://morrison-lab.github.io/lds/chapters/data/spambase.csv

Solution

Solution 4.4.

```
default <- read.csv("data/spambase.csv")
names(default)[49:54]
```

```
[1] "char_freq_."   "char_freq_..1" "char_freq_..2" "char_freq_..3"
[5] "char_freq_..4" "char_freq_..5"
```

Exercise 4.5 (Read the file with R's defaults (part 2))**.** Use the setup of Exercise 4.4. What happened to them?

Solution

Solution 4.5. Six names changed: `;`, `(`, `[`, `!`, `$` and `#` cannot appear in an R name, so each became `.`, and the six results were made unique by appending `.1`, `.2` and so on.

Exercise 4.6 (Read the file with R's defaults (part 3))**.** Use the setup of Exercise 4.4. Why did that happen to them?

Solution

Solution 4.6. By default `read.csv` passes every column name through `make.names()`, which turns anything that is not a legal R name into one.

Exercise 4.7 (Read the file with R's defaults (part 4))**.** Use the setup of Exercise 4.4.
How do you keep the names as they are in the file?

Solution

Solution 4.7.

```
kept <- read.csv("data/spambase.csv", check.names = FALSE)
names(kept)[49:54]
```

```
[1] "char_freq_;" "char_freq_(" "char_freq_[" "char_freq_!" "char_freq_$"
[6] "char_freq_#"
```

`check.names = FALSE` keeps the file's names, which must then be quoted with backticks in `$` access: `spam$`char_freq_(`.
Python's `pandas.read_csv` keeps them as they are, and they are used as strings in either case: `spam["char_freq_!"]`.

Exercise 4.8 (Look at four messages)**.** The table of spam and email averages gives the *average* message in each group. Select the same five columns for the first two messages in the file, and for the first two real messages that contain both `george` and `hp`.

Solution

Solution 4.8. Python counts rows from 0 and R from 1, so the same four messages are `[0, 1, 1818, 1823]` in one and `c(1, 2, 1819, 1824)` in the other.

4.1.7 Python

```
import pandas as pd

spam = pd.read_csv("data/spambase.csv")
cols = ["word_freq_george", "word_freq_you", "word_freq_free",
        "char_freq_!", "word_freq_hp", "Class"]
print(spam.loc[[0, 1, 1818, 1823], cols].to_string())
```

```
      word_freq_george  word_freq_you  word_freq_free  char_freq_!  word_freq_hp  Class
0                 0.00           1.93            0.32        0.778          0.00      1
1                 0.00           3.47            0.14        0.372          0.00      1
1818              2.04           0.00            0.00        0.000          4.08      0
1823              0.24           0.24            0.00        0.149          9.33      0
```

4.1.8 R

```r
spam <- read.csv("data/spambase.csv", check.names = FALSE)
cols <- c("word_freq_george", "word_freq_you", "word_freq_free",
          "char_freq_!", "word_freq_hp", "Class")
spam |>
  dplyr::mutate(id = dplyr::row_number()) |>
  dplyr::filter(id %in% c(1, 2, 1819, 1824)) |>
  dplyr::select(id, dplyr::all_of(cols)) |>
  knitr::kable()
```

Table 5: Five of the 57 columns, for two spam messages and two real ones.

id	word_freq_george	word_freq_you	word_freq_free	char_freq_!	word_freq_hp	Class
1	0.00	1.93	0.32	0.778	0.00	1
2	0.00	3.47	0.14	0.372	0.00	1
1819	2.04	0.00	0.00	0.000	4.08	0
1824	0.24	0.24	0.00	0.149	9.33	0

Exercise 4.9 (Look at four messages (part 2))**.** Use the setup of Exercise 4.8.
Does each message look like its group's average?

Solution

Solution 4.9.

```r
you_spam_avg <- mean(spam$word_freq_you[spam$Class == 1])
```

Each message does look like its group. Neither spam message contains `george` or `hp` at all, and both real ones contain both.
An average is not a guarantee, though. Message 1 is spam with `you` at 1.93, below the spam average of 2.26 and well below message 2's 3.47. A rule built on one column will misclassify messages that a rule built on several would classify correctly.

Exercise 4.10 (Check the order of the rows)**.** Is the file shuffled? Look at the `Class` column from top to bottom and say how it is arranged.

Solution

Solution 4.10. Count the runs of a constant label, from the top of the file down:

4.1.9 Python

```python
import pandas as pd

spam = pd.read_csv("data/spambase.csv")
print(spam["Class"].ne(spam["Class"].shift()).sum())
```

```
2
```

```python
print(spam["Class"].head(3).tolist(), spam["Class"].tail(3).tolist())
```

```
[1, 1, 1] [0, 0, 0]
```

4.1.10 R

```
spam <- read.csv("data/spambase.csv", check.names = FALSE)
rle(spam$Class)
```

```
Run Length Encoding
  lengths: int [1:2] 1813 2788
  values : int [1:2] 1 0
```

```
runs <- rle(spam$Class)
n_train <- floor(0.8 * nrow(spam))
```

There are 2 runs, so the label changes exactly once: all 1813 spam messages come first, then all 2788 real ones.

Exercise 4.11 (Check the order of the rows (part 2))**.** Use the setup of Exercise 4.10.
Why would the order of the rows matter?

Solution

Solution 4.11. So taking the first 80% of the rows, 3680 of them, as a training set would put every spam message in training and leave a test set of email alone. Any split of this file has to shuffle first, so the two halves in the held-out exercise were drawn at random.

Exercise 4.12 (How many rules did the computer try?)**.** The *best rule* search in the exercise on beating the all-email rule tried every `%george` threshold from 0 to 5 and every `%you` threshold from 0 to 10, each in steps of 0.05. Count the candidate rules.

Solution

Solution 4.12. From 0 to 5 in steps of 0.05 is $5/0.05 + 1 = 101$ thresholds, and from 0 to 10 is $10/0.05 + 1 = 201$. Every pair is a rule, so the search tried $101 \times 201 = 20301$ rules, each scored on all 4601 emails. A computer does that in much less than a second; a person does not.

4.2 Statistical learning: a short history

The name *statistical learning* is recent, but many of its methods are old [10, Ch. 1]:

- least squares, the first form of linear regression, in the early 1800s;
- linear discriminant analysis in 1936, and logistic regression in the 1940s;
- generalized linear models in the early 1970s;
- classification and regression trees and neural networks in the 1980s, once computers could fit nonlinear models;
- support vector machines in the 1990s.

Statisticians and computer scientists developed these methods at the same time. The same tools led to data mining, machine learning and bioinformatics, fields whose methods "have common underpinnings but are often expressed with different terminology" [5], back cover.

Watch

The Data Science Learning Community's book clubs discuss *An Introduction to Statistical Learning*, Chapter 1, in recorded meetings:

- the R book club, in one meeting[a] [35];
- the Python book club, in one meeting[b] [36].

[a] https://www.youtube.com/watch?v=g1SOI7Q6nEM&list=PL3x6DOfs2NGjlhpKMiWsDcKL2nngwXROq
[b] https://www.youtube.com/watch?v=mbmhUgk9XW0&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp

4.3 Loading the Auto data yourself

Exercise 4.13 (Load the data yourself)**.** The Auto data are available as `data/auto.csv`[a]; the note on the Auto data says where that file comes from. In either language, load the file and report how many rows and columns it has.

[a] https://morrison-lab.github.io/lds/chapters/data/auto.csv

Solution

Solution 4.13.

4.3.1 Python

```
import pandas as pd

# ISLP's load_data("Auto") reads this same file,
# with each car's name as the row index.
Auto = pd.read_csv("data/auto.csv", index_col="name")
print(Auto.shape)
```

```
(392, 8)
```

4.3.2 R

```
# The ISLR2 package holds the same data frame, as `Auto`.
Auto <- read.csv("data/auto.csv", stringsAsFactors = TRUE)
str(Auto)
```

```
'data.frame':   392 obs. of  9 variables:
 $ mpg         : num  18 15 18 16 17 15 14 14 14 15 ...
 $ cylinders   : int  8 8 8 8 8 8 8 8 8 8 ...
 $ displacement: num  307 350 318 304 302 429 454 440 455 390 ...
 $ horsepower  : int  130 165 150 150 140 198 220 215 225 190 ...
 $ weight      : int  3504 3693 3436 3433 3449 4341 4354 4312 4425 3850 ...
 $ acceleration: num  12 11.5 11 12 10.5 10 9 8.5 10 8.5 ...
 $ year        : int  70 70 70 70 70 70 70 70 70 70 ...
 $ origin      : int  1 1 1 1 1 1 1 1 1 1 ...
 $ name        : Factor w/ 301 levels "amc ambassador brougham",..: 49 36 230 14 160 141 54 222 240 2
```

Both find 392 cars. Python reports 8 columns, because the code makes each car's `name` the row index, as `ISLP` does; R reports 9, with `name` as an ordinary column.

Exercise 4.14 (Summarize three variables)**.** Now summarize `mpg`, `horsepower` and `weight`. What is the range of each, and the mean of `mpg`?

Solution

Solution 4.14.

4.3.3 Python

```
import pandas as pd

Auto = pd.read_csv("data/auto.csv", index_col="name")
print(Auto[["mpg", "horsepower", "weight"]].describe())
```

```
              mpg  horsepower       weight
count  392.000000  392.000000   392.000000
mean    23.445918  104.469388  2977.584184
std      7.805007   38.491160   849.402560
min      9.000000   46.000000  1613.000000
25%     17.000000   75.000000  2225.250000
50%     22.750000   93.500000  2803.500000
75%     29.000000  126.000000  3614.750000
max     46.600000  230.000000  5140.000000
```

4.3.4 R

```
Auto <- read.csv("data/auto.csv", stringsAsFactors = TRUE)
Auto |>
  dplyr::select(mpg, horsepower, weight) |>
  gtsummary::tbl_summary(
    type = gtsummary::all_continuous() ~ "continuous2",
    statistic = gtsummary::all_continuous() ~
      c("{mean} ({sd})", "{min}, {max}"),
    digits = gtsummary::all_continuous() ~ 1
  ) |>
  gtsummary::as_kable()
```

Characteristic	**N = 392**
mpg	
Mean (SD)	23.4 (7.8)
Min, Max	9.0, 46.6
horsepower	
Mean (SD)	104.5 (38.5)
Min, Max	46.0, 230.0
weight	
Mean (SD)	2,977.6 (849.4)
Min, Max	1,613.0, 5,140.0

`mpg` runs from 9 to 46.6, with a mean of 23.4; `horsepower` from 46 to 230; and `weight` from 1613 to 5140 pounds.

The chapter 2 lab goes further, in Python (`Ch02-statlearn-lab.ipynb` in the ISLP labs repository[1]) and in R (the chapter 2 files on the book's R resources page[2]).

[1] https://github.com/Morrison-Lab/ISLP_labs
[2] https://www.statlearning.com/resources-second-edition

4.4 Sorting learning problems

Exercise 4.15 (Supervised or unsupervised?)**.** For each task, decide whether it is supervised or unsupervised.
Recognize which digit, 0 to 9, a handwritten image shows.

Solution

Solution 4.15. Supervised: each image comes with its digit.

Exercise 4.16 (Supervised or unsupervised? (part 2))**.** Use the setup of Exercise 4.15.
Predict tomorrow's high temperature in Bellingham.

Solution

Solution 4.16. Supervised: past days come with their recorded highs.

Exercise 4.17 (Supervised or unsupervised? (part 3))**.** Use the setup of Exercise 4.15.
Group news articles by topic, with no topics given in advance.

Solution

Solution 4.17. Unsupervised (clustering): no topics are given.

Exercise 4.18 (Supervised or unsupervised? (part 4))**.** Use the setup of Exercise 4.15.
Predict a house's sale price from its size and location.

Solution

Solution 4.18. Supervised: past sales come with their prices.

Exercise 4.19 (Supervised or unsupervised? (part 5))**.** Use the setup of Exercise 4.15.
Decide whether a credit-card transaction is fraudulent, given past transactions marked as fraud or not.

Solution

Solution 4.19. Supervised: past transactions are marked as fraud or not.
The fraud task can also be an unsupervised problem. A detector could flag transactions unlike the rest even with no labels marking past fraud. That version of the task is called *anomaly detection*. The same goal can lead to different problems depending on what data exist.

Exercise 4.20 (Supervised or unsupervised? (part 6))**.** Use the setup of Exercise 4.15.
Find groups of customers who shop in similar ways.

Solution

Solution 4.20. Unsupervised (clustering): no groups are given.

Exercise 4.21 (Regression or classification, again)**.** For each supervised task in Exercise 4.15, decide whether it is regression or classification.

Solution

Solution 4.21.

- Recognizing a handwritten digit (Exercise 4.15) is classification, with ten classes.
- Predicting tomorrow's high temperature (Exercise 4.16) and a house's sale price (Exercise 4.18) are regression.
- Deciding whether a transaction is fraudulent (Exercise 4.19) is classification, with two classes.

4.5 Asking questions of data

Exercise 4.22 (Asking questions of data)**.** Machine learning starts with a question that data can help answer. A transit agency records card taps at stations, with the time, the line and the weather that day. What patterns might exist in these data?

Solution

Solution 4.22. Many answers work. Some examples:
Public transit: the traffic at a station may depend on the time, the line and the weather, and the traffic at some stations may rise and fall together.

Exercise 4.23 (Asking questions of data (part 2))**.** What would you want a model to predict from the transit data of Exercise 4.22?

Solution

Solution 4.23. **Public transit:** predict how many people will tap in at a station during the next hour from the time, the line and the weather.

Exercise 4.24 (Asking questions of data (part 3))**.** Would the model of Exercise 4.23 be supervised or unsupervised?

Solution

Solution 4.24. **Public transit:** predicting the number of taps is supervised regression, and finding groups of stations whose traffic rises and falls together is unsupervised.

Exercise 4.25 (Asking questions of data (part 4))**.** A farm records soil samples, planting dates, rainfall and harvest weights for many fields. What patterns might exist in these data?

Solution

Solution 4.25. **Farming:** a field's harvest weight may depend on its soil, planting date and rainfall, and fields planted late with little rain may form a group.

Exercise 4.26 (Asking questions of data (part 5))**.** What would you want a model to predict from the farming data of Exercise 4.25?

Solution

Solution 4.26. **Farming:** predict a field's harvest weight from its soil, planting date and rainfall.

Exercise 4.27 (Asking questions of data (part 6))**.** Would the model of Exercise 4.26 be supervised or unsupervised?

Solution

Solution 4.27. **Farming:** predicting the harvest weight is supervised regression, and flagging fields that were planted late and got little rain is unsupervised grouping.

Exercise 4.28 (Asking questions of data (part 7))**.** A public library records each loan, with the book's subject and the borrower's earlier loans. What patterns might exist in these data?

Solution

Solution 4.28. **A public library:** borrowers who read one subject may borrow more books on it, and some borrowers may read alike.

Exercise 4.29 (Asking questions of data (part 8))**.** What would you want a model to predict from the library data of Exercise 4.28?

Solution

Solution 4.29. **A public library:** predict whether a borrower will take out a second book from the same subject.

Exercise 4.30 (Asking questions of data (part 9))**.** Would the model of Exercise 4.29 be supervised or unsupervised?

Solution

Solution 4.30. **A public library:** predicting whether a borrower takes a second book is supervised classification, and grouping borrowers by what they read is unsupervised.

Exercise 4.31 (Your own problem)**.** Brainstorm one use of machine learning not mentioned in this lecture, and name its input features and its output.

Solution

Solution 4.31. Answers will vary; we will share a few and sort them together, as supervised or unsupervised. For a supervised one, check that its output is something you could have recorded for past examples, since the model learns from those recorded outputs.

References

[1] Wikipedia contributors, "Feature (machine learning) — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Feature_(machine_learning)

[2] Wikipedia contributors, "Scientific modelling — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Scientific_modelling

[3] Wikipedia contributors, "Model — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Model

[4] G. E. P. Box, "Robustness in the strategy of scientific model building," in *Robustness in statistics*, R. L. Launer and G. N. Wilkinson, Eds., Academic Press, 1979, pp. 201–236. doi: 10.1016/B978-0-12-438150-6.50018-2[3].

[5] 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/

[6] K. P. Murphy, *Machine learning: A probabilistic perspective*. in Adaptive computation and machine learning. MIT Press, 2012.

[7] M. Hopkins, E. Reeber, G. Forman, and J. Suermondt, "Spambase." UCI Machine Learning Repository, 1999. doi: 10.24432/C53G6X[4].

[8] Y. S. Abu-Mostafa, "Learning from data: CS 156 lectures." Caltech, lecture videos on YouTube, 2012. Available: https://www.youtube.com/playlist?list=PLD63A284B7615313A

[9] K. Weinberger, "CS4780: Machine learning for intelligent systems, spring 2017." Cornell University, lecture videos on YouTube, 2017. Available: https://www.youtube.com/playlist?list=PLl8OlHZGYOQ7bkVbuRthEsaLr7bONzbXS

[10] 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/

[11] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, *An introduction to statistical learning: With applications in Python*. Springer, 2023. doi: 10.1007/978-3-031-38747-0[5].

[12] Wikipedia contributors, "Supervised learning — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Supervised_learning

[13] Wikipedia contributors, "Unsupervised learning — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Unsupervised_learning

[14] 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

[15] GeeksforGeeks, "Supervised and unsupervised learning." GeeksforGeeks, tutorial article, 2025. Available: https://www.geeksforgeeks.org/machine-learning/supervised-unsupervised-learning/

[16] Wikipedia contributors, "Weak supervision — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Weak_supervision

[17] Wikipedia contributors, "Reinforcement learning — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Reinforcement_learning

[18] GeeksforGeeks, "Semi-supervised learning in ML." GeeksforGeeks, tutorial article, 2026. Available: https://www.geeksforgeeks.org/machine-learning/ml-semi-supervised-learning/

[19] GeeksforGeeks, "Reinforcement learning." GeeksforGeeks, tutorial article, 2026. Available: https://www.geeksforgeeks.org/machine-learning/what-is-reinforcement-learning/

[20] Wikipedia contributors, "Cluster analysis — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Cluster_analysis

[21] Wikipedia contributors, "K-means clustering — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/K-means_clustering

[22] GeeksforGeeks, "K means clustering – introduction." GeeksforGeeks, tutorial article, 2026. Available: https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

[23] Wikipedia contributors, "Regression analysis — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Regression_analysis

[24] Wikipedia contributors, "Statistical classification — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Statistical_classification

[25] GeeksforGeeks, "Classification vs regression in machine learning." GeeksforGeeks, tutorial article, 2025. Available: https://www.geeksforgeeks.org/machine-learning/ml-classification-vs-regression/

[3] https://doi.org/10.1016/B978-0-12-438150-6.50018-2
[4] https://doi.org/10.24432/C53G6X
[5] https://doi.org/10.1007/978-3-031-38747-0

[26] Wikipedia contributors, "Mean squared error — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Mean_squared_error

[27] Wikipedia contributors, "Root-mean-square deviation — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Root-mean-square_deviation

[28] 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.wwu.edu/~hutchib2/video_lectures/data371/

[29] GeeksforGeeks, "Mean squared error." GeeksforGeeks, tutorial article, 2026. Available: https://www.geeksforgeeks.org/maths/mean-squared-error/

[30] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959, doi: 10.1147/rd.33.0210[6].

[31] Wikipedia contributors, "Evaluation function — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Evaluation_function

[32] A. L. Samuel, "Some studies in machine learning using the game of checkers. II—recent progress," *IBM Journal of Research and Development*, vol. 11, no. 6, pp. 601–617, 1967, doi: 10.1147/rd.116.0601[7].

[33] Wikipedia contributors, "Minimax — Wikipedia, the free encyclopedia." 2026. Available: https://en.wikipedia.org/wiki/Minimax

[34] A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis, "kernlab – an S4 package for kernel methods in R," *Journal of Statistical Software*, vol. 11, no. 9, pp. 1–20, 2004, doi: 10.18637/jss.v011.i09[8].

[35] 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=PL3x6DOfs2NGjlhpKMiWsDcKL2nngwXROq

[36] 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

[6] https://doi.org/10.1147/rd.33.0210
[7] https://doi.org/10.1147/rd.116.0601
[8] https://doi.org/10.18637/jss.v011.i09