

Unsupervised Learning

Last modified: 2026-10-26

```
# The Python chunks on this page import numpy; reticulate provisions it.  
# The Julia chunks run through JuliaCall.  
library(reticulate)  
py_require(c("numpy", "pandas", "scipy"))
```

i Source

Consolidated from Brian Hutchinson’s Fall 2025 topic notes and Kameron Decker Harris’ 2022 weekly notes, and drawing on Logan Sizemore’s Spring 2025 lectures on k-means, on hierarchical clustering and on principal component analysis. The linkage formulas are not in those lectures, which describe the criteria in prose and do not cover Ward’s; they are stated here from the cited sources.

1 Clustering overview

- Clustering is an *unsupervised* task: $\mathcal{X} = \{x_{(1)}, \dots, x_{(N)}\}$
- Goal is to assign points to “clusters” - items in same cluster are similar in some way
- Usually *exploratory*: learn something about the structure of the data (input)

1.1 Why cluster?

- To find some meaningful organization to the data:
Clustering speakers in a talk show based on word frequencies: role? gender? topic?
If features were pitch and height, how might different sets cluster? age, sex, etc.
Don’t expect a perfect clustering - you can only control (features, algorithm) but you can’t provide labels
- To discretize or quantize a continuous value.
“Vector quantization”

i Watch

von Luxburg’s lecture introduces clustering^a [1, Pt. 32].

^a<https://www.youtube.com/watch?v=wYzcnPR8XWU>

i Read

GeeksforGeeks explains clustering^a [2].

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

2 Approaches to clustering

- Flat clustering vs hierarchical clustering (subclusters of subclusters)
- Centroid-based clustering (flat)
 - Each cluster is defined by some centroid, points assigned to closest centroid
 - Example: k-means
- Graph-cut clustering (flat)
 - Each point is a node in a graph
 - Edges denote similarity between nodes (may be omitted if sufficiently small)
 - Find the cheapest edges to cut so that the graph splits into K disconnected parts
 - Example: spectral clustering
- Agglomerative clustering (hierarchical)
 - Start with each node as its own cluster
 - Repeatedly merge the most similar clusters until some stopping criterion
 - Some cluster similarity criteria
 - * Single link: distance between the closest points in each
 - Cheaper than others (N^2 rather than N^3) since it is a spanning tree
 - Leads to non-compact clusters
 - * Complete link: distance between the most distant points in each
 - Leads to compact clusters
 - * Average link: average distance between pairs of nodes (one from each)
 - Produces fairly compact clusters
 - Most common
 - To draw the tree, make each merge a new unlabeled node whose left and right children are the two clusters that were merged.
- Divisive clustering (hierarchical)
 - Start with one large cluster, repeatedly split
 - * Can use another clustering method to split up one cluster
- Often need to compute distance/dissimilarity between points
 - 1) Squared euclidian distance:

$$\begin{aligned}d(x, x') &= \|x - x'\|_2^2 \\ &= \sum_i (x_i - x'_i)^2\end{aligned}$$

- 2) ℓ_1 /city block/manhattan distance:

$$\begin{aligned}d(x, x') &= \|x - x'\|_1 \\ &= \sum_i |x_i - x'_i|\end{aligned}$$

- 3) Hamming distance (for categorical): $d(x, x') = \sum_i I(x_i \neq x'_i)$
- Whatever d you choose, three properties make it behave like a notion of “unlike”
 - Symmetry: $d(x, x') = d(x', x)$. Without it a pair of objects could be close when you read the pair in one order and far apart in the other.
 - Positive separability: $d(x, x') = 0$ if and only if $x = x'$. Without it two genuinely different objects are indistinguishable to the algorithm.
 - Triangle inequality: $d(x, x'') \leq d(x, x') + d(x', x'')$. Without it a detour through a third object could be much shorter than the direct distance.
- A function with all three is a **metric**; a **dissimilarity function** need only have the first two. The distinction matters immediately, because entry 1 of the list of distances is not a metric. Take the collinear points 0, 1 and 2 on the real line: squared Euclidean distance¹ gives $d(0, 2) = 4$ while $d(0, 1) + d(1, 2) = 2$. Squaring is monotone, so it never changes which point is nearest, and algorithms that only ever rank distances are unaffected — but any argument that relies on the triangle inequality does not apply.
- Entry 2 is the $p = 1$ member of the **Minkowski** family, and entry 1 is the square of the $p = 2$ member:

$$\begin{aligned} d_p(x, x') &= \|x - x'\|_p \\ &= \left(\sum_i |x_i - x'_i|^p \right)^{1/p} \end{aligned}$$

- Every d_p with $p \geq 1$ is a metric. Raising p puts more of the weight on the single largest coordinate gap, until $p \rightarrow \infty$ leaves only that gap: $d_\infty(x, x') = \max_i |x_i - x'_i|$.
- $p < 1$ is occasionally used in high dimensions, but it is no longer a metric — the triangle inequality fails.

Exercise 2.1 (Distances between binary vectors). Two binary vectors of length 8 differ in exactly three positions. Compute the squared Euclidean distance, the ℓ_1 distance and the Hamming distance between them.

Solution

Solution 2.1. Each of the three distances adds up a contribution from every position. In a position where the vectors agree, every contribution is 0. In a position where they differ, one entry is 0 and the other is 1, so the contribution is $(1)^2 = 1$ to the squared Euclidean distance, $|1| = 1$ to the ℓ_1 distance, and 1 to the Hamming distance. With three differing positions, all three distances equal 3.

Exercise 2.2 (Distances between binary vectors (part 2)). Use the setup of Exercise 2.1. What does this say about using the three on binary data?

Solution

Solution 2.2. So on binary data the three are the same number: the count of positions where the vectors differ. Changing from one to another changes nothing about which vectors are close to each other, and the plain Euclidean distance^a, $\sqrt{3}$ here, is a monotone function of that count, so it orders the distances the same way.

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

¹<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-euclidean-distance>

3 K-means clustering

3.1 The algorithm

- A very simple iterative flat clustering algorithm
- K is the number of clusters, and is a hyper-parameter
A parameter that controls the structure of the model; we choose it rather than learn it from data
- Initialize the centroids, often picking K datapoints at random (without replacement)
- The algorithm alternates between two steps:
 1. Assign each datapoint to cluster with the closest centroid

$$C(x) = \arg \min_i \|x - c_{(i)}\|$$

2. Update the centroids to be the mean of the points in their cluster

$$c_{(i)} = \frac{1}{N_{(i)}} \sum_{x:C(x)=i} x$$

- We iterate until no assignment changes

Visualization: <https://bits.csb.pitt.edu/kmeans/>

Exercise 3.1 (Trace K -means by hand). Five points on the real line sit at 1, 2, 4, 10 and 11. Run K -means with $K = 2$, starting the centroids at 2 and 4. At each iteration, say which points the assignment step gives to each centroid and where the update step moves the centroids, until nothing changes.

Solution

Solution 3.1. **Iteration 1.** The point 4 is at distance 0 from the second centroid, and 1 and 2 are nearer the first, so the first cluster is $\{1, 2\}$. The other points, 4, 10 and 11, are nearer the second centroid, at 4. The update moves the centroids to the cluster means: $c_1 = 1.5$ and $c_2 = (4 + 10 + 11)/3 = 25/3 \approx 8.33$. **Iteration 2.** The point 4 is 2.5 from c_1 and about 4.33 from c_2 , so it switches to the first cluster. The points 10 and 11 stay with c_2 . The update gives $c_1 = (1 + 2 + 4)/3 = 7/3 \approx 2.33$ and $c_2 = 10.5$. **Iteration 3.** Now 4 is $5/3$ from c_1 and 6.5 from c_2 , so no assignment changes, and the algorithm stops.

3.1.1 Python

```
import numpy as np

# The five points and the starting centroids of the exercise.
pts = np.array([1.0, 2.0, 4.0, 10.0, 11.0])
cent = np.array([2.0, 4.0])

while True:
    # Assignment step: each point goes to its nearest centroid.
    labels = np.argmin(np.abs(pts[:, None] - cent[None, :]), axis=1)
    # Update step: each centroid moves to the mean of its points.
    new_cent = np.array([pts[labels == k].mean() for k in range(2)])
    print("labels", labels, "centroids", new_cent.round(2))
    # Stop when the centroids no longer move, which happens exactly when
    # the assignments stop changing.
    if np.allclose(new_cent, cent):
        break
    cent = new_cent

labels [0 0 1 1 1] centroids [1.5  8.33]
labels [0 0 0 1 1] centroids [ 2.33 10.5 ]
labels [0 0 0 1 1] centroids [ 2.33 10.5 ]
```

3.1.2 R

```
# The five points and the starting centroids of the exercise.
pts <- c(1, 2, 4, 10, 11)
cent <- c(2, 4)

repeat {
    # Assignment step: each point goes to its nearest centroid.
    labels <- apply(abs(outer(pts, cent, "-")), 1, which.min)
    # Update step: each centroid moves to the mean of its points.
    new_cent <- tapply(pts, labels, mean)
    cat("labels", labels, "centroids", round(new_cent, 2), "\n")
    # Stop when the centroids no longer move, which happens exactly when
    # the assignments stop changing.
    if (all(abs(new_cent - cent) < 1e-9)) break
    cent <- as.numeric(new_cent)
}

labels 1 1 2 2 2 centroids 1.5 8.33
labels 1 1 1 2 2 centroids 2.33 10.5
labels 1 1 1 2 2 centroids 2.33 10.5
Both runs end after three iterations at the centroids 2.33 and 10.5.
```

Exercise 3.2 (Trace K -means by hand (part 2)). Use the setup of Exercise 3.1. Report the final clusters and the distortion.

Solution

Solution 3.2. The final clusters are $\{1, 2, 4\}$ and $\{10, 11\}$. The distortion is the sum of squared distances to the centroids:

$$\underbrace{\left(\frac{4}{3}\right)^2 + \left(\frac{1}{3}\right)^2 + \left(\frac{5}{3}\right)^2}_{\text{first cluster}} + \underbrace{0.5^2 + 0.5^2}_{\text{second cluster}} = \frac{42}{9} + 0.5 \approx 5.17.$$

3.1.3 Python

```
# Distortion: squared distance of every point to its own centroid.
print("distortion", round(float(np.sum((pts - cent[labels]) ** 2)), 2))

distortion 5.17
```

3.1.4 R

```
# Distortion: squared distance of every point to its own centroid.
cat("distortion", round(sum((pts - cent[labels])^2), 2), "\n")

distortion 5.17
Both runs report the distortion found by hand.
```

3.2 The k-means objective

- The k-means algorithm tries to solve the following problem (minimize “distortion”)

$$\arg \min_{c_{(1)}, \dots, c_{(K)}, C} \sum_{x \in \mathcal{X}} \|x - c_{(C(x))}\|^2 \quad (1)$$

- *Non-convex*, but converges to a local optimum
- We can produce several random clusterings and take the one with the best objective value.

Exercise 3.3 (Show that K -means never raises the distortion). One iteration of K -means is an assignment step followed by an update step. Explain why the distortion in Equation 1 cannot go up during the assignment step.

Solution

Solution 3.3. Each step minimizes the same distortion over one of its two arguments while the other is held fixed.

The assignment step chooses C with the centroids fixed. The distortion is a sum over points, one term per point, so it is smallest when each point picks the centroid nearest to it, as the step does. No other assignment gives a smaller sum, so the step cannot raise the distortion.

Exercise 3.4 (Show that K -means never raises the distortion (part 2)). Use the setup of Exercise 3.3. Explain why the distortion in Equation 1 cannot go up during the update step.

Solution

Solution 3.4. As in Solution 3.3, the update step chooses the centroids with the assignments fixed. For one cluster, the mean^a is the point that minimizes the sum of squared distances to the cluster’s points, so moving each centroid to its cluster’s mean cannot raise the distortion.

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

Exercise 3.5 (Show that K -means never raises the distortion (part 3)). Use the setup of Exercise 3.3. What does that imply about the algorithm^a stopping?

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

Solution

Solution 3.5. The distortion therefore falls or stays level at every step. Whenever the assignments change, it falls strictly (when a point is equidistant from two centroids, keep it with its current one, so that a tie never counts as a change). A distortion that falls strictly cannot return to a value it has had, so the algorithm^a cannot revisit a set of assignments. The assignments take only finitely many values, so the algorithm must stop, at a local optimum that depends on where the centroids started.

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

Theorem 3.1 (*K*-means never raises the distortion). *Neither the assignment step nor the update step of K-means increases the distortion of Equation 1. The distortion strictly decreases whenever the assignments change, so K-means stops after finitely many iterations, at a local optimum that depends on the starting centroids.*

i Proof

Proof. By Solution 3.3 and Solution 3.4. □

Example 3.1 (The distortion over iterations). Run *K*-means with $K = 2$ on a small simulated data set, and print the distortion after each iteration:

3.3 Python

```
import numpy as np

rng = np.random.default_rng(571)
pts = np.vstack([rng.normal(0, 1, (20, 2)), rng.normal(3, 1, (20, 2))])
cent = pts[[0, 1]].copy()
dist = []
for _ in range(6):
    assign = np.argmax(((pts[:, None] - cent[None]) ** 2).sum(2), axis=1)
    cent = np.array([pts[assign == k].mean(0) for k in range(2)])
    dist.append(((pts - cent[assign]) ** 2).sum())
print(np.round(dist, 3), bool(np.all(np.diff(dist) <= 1e-12)))

[230.171 106.863 68.401 67.592 67.592 67.592] True
```

3.4 R

```
set.seed(571)
pts <- rbind(matrix(rnorm(40, 0, 1), 20), matrix(rnorm(40, 3, 1), 20))
cent <- pts[1:2, ]
dist <- numeric(6)
for (i in 1:6) {
  d2 <- sapply(1:2, function(k) rowSums(sweep(pts, 2, cent[k, ])^2))
  assign <- max.col(-d2)
  cent <- t(sapply(1:2, function(k) colMeans(pts[assign == k, , drop = FALSE])))
  dist[i] <- sum((pts - cent[assign, ])^2)
}
round(dist, 3)

[1] 156.756 83.503 73.213 73.213 73.213 73.213
```

```
all(diff(dist) <= 1e-12)
```

```
[1] TRUE
```

The distortion never rises, and it stops changing once the assignments stop changing.

Exercise 3.6 (Compare the best distortion for different K). Let J_K^* be the smallest distortion over all choices of K centroids and assignments for a fixed data set of $N \geq 4$ distinct points. Is J_4^* always smaller than J_3^* , always larger, or neither?

Solution

Solution 3.6. J_4^* is always smaller than J_3^* , once the data set has at least 4 distinct points. Take the best solution with three centroids. Some cluster holds two or more distinct points, so at least one of them is not at its cluster's centroid. Put the fourth centroid exactly on that point. Its squared distance drops from a positive number to 0, and every other point still has a centroid at least as near as before, so the distortion falls strictly. The best four-centroid solution can only do better still. The same argument shows that J_K^* falls strictly with each added centroid while $K \leq N$.

Exercise 3.7 (Compare the best distortion for different K (part 2)). Use the setup of Exercise 3.6. What is J_N^* ?

Solution

Solution 3.7. With $K = N$, put one centroid on each point. Every squared distance is 0, so $J_N^* = 0$.

Exercise 3.8 (Compare the best distortion for different K (part 3)). Use the setup of Exercise 3.6. Does your answer apply to the distortion a single run of K -means returns?

Solution

Solution 3.8. The answer applies to the best distortion, not to what one run returns. K -means finds a local optimum that depends on the starting centroids, so a single run with $K = 4$ can finish with a larger distortion than a lucky run with $K = 3$. Taking the best of several random starts for each K brings the comparison close to the one about J_K^* . The best distortion keeps falling as K grows, so the lowest distortion cannot pick K by itself. The elbow method instead looks for where the decrease slows.

Exercise 3.9 (Spot a clustering that K -means could not stop at). Six points on the real line sit at 0, 1, 2, 9, 10 and 11. Consider two clusterings for $K = 2$:

- Clustering A: $\{0, 1, 2\}$ and $\{9, 10, 11\}$.
- Clustering B: $\{0, 1, 2, 9\}$ and $\{10, 11\}$.

With centroids at the cluster means^a, what test tells you whether K -means could stop at a clustering?

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

Solution

Solution 3.9. K -means stops when the assignment step no longer changes anything, so a clustering is a possible stopping point only if every point is closest to the mean of its own cluster.

Exercise 3.10 (Spot a clustering that K -means could not stop at (part 2)). Use the setup of Exercise 3.9. With centroids at the cluster means^a, could K -means have stopped at clustering A?

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

Solution

Solution 3.10. Clustering A has centroids 1 and 10. Every point is nearest to the centroid of its own group, so nothing moves: K -means could stop here.

Exercise 3.11 (Spot a clustering that K -means could not stop at (part 3)). Use the setup of Exercise 3.9. With centroids at the cluster means^a, could K -means have stopped at clustering B?

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

Solution

Solution 3.11. Clustering B has centroids 3 and 10.5. The point 9 is 6 from its own centroid but only 1.5 from the other one, so the assignment step would move it into the second cluster. Clustering B is not a stopping point, and K -means would never return it.

3.5 How k-means behaves

- Simple and works fairly well
- Clusters form Voronoi tessellation (or diagram, or partition, or decomposition)
 - Each cluster is the intersection of half-spaces, form *convex polygons*

3.6 When k-means fails

The Voronoi picture is also a list of assumptions. Each assumption names data that a person would cluster without hesitating but k-means will not.

- **Clusters cover equal area.** A cell boundary sits halfway between two centroids, so a large diffuse cluster beside a small tight one loses its outer points to the small cluster's centroid. The objective uses each cluster's mean and ignores its extent.
- **Clusters are spherical, with the same variance in every direction.** Squared Euclidean distance to the centroid treats all directions alike, so k-means cuts an elongated cluster across its long axis instead of recognizing it whole.
- **Clusters are linearly separable.** Intersections of half-spaces are convex, so no value of K and no number of restarts will put a ring and the blob inside it into separate clusters.
- **Every point belongs to some cluster.** K -means cannot leave a point unassigned. It assigns an outlier to the nearest centroid, and the outlier then pulls that centroid toward itself, because it enters the mean with the same weight as any other point.

Every remedy gives the model something the k-means objective, Equation 1, lacks: a per-cluster notion of shape, a notion of adjacency that is not straight-line distance, or the option of assigning nothing at all.

3.7 How to pick the number of clusters

There is no best answer. Usual methods:

- By visual inspection (if low-dimensional).
- By manually inspecting cluster members or statistics (if high-dimensional).

- By expert knowledge (does not use the data).
- By how well it works for some downstream supervised task. Hyper-parameter to tune.
- By the elbow method: plot distortion vs K , look for the “elbow”.
- Techniques out of scope for this class but you can search if interested:
 - By the silhouette method: plot silhouette score vs K , look for the peak.
 - By the eigengap method: plot eigenvalues of the graph Laplacian, look for the gap.

i Watch

Broderick shows how to choose k , from 47:24^a [3].

^a<https://www.youtube.com/watch?v=BaZWcSq3IuI&t=2844s>

i Read

GeeksforGeeks explains the elbow method^a [4].

^a<https://www.geeksforgeeks.org/machine-learning/elbow-method-for-optimal-value-of-k-in-kmeans/>

i Read

The University of Washington’s CSE 446 slides cover K -means^a [5], Lecture 24.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/24-K-means.pdf>

3.8 In code

Both steps are short enough to write out, and doing so shows what Theorem 3.1 states: neither step can increase the distortion, because each is a minimization of that same objective holding the other argument fixed. The assignment step minimizes over C with the centroids fixed, and the update step minimizes over the centroids with C fixed, since the mean minimizes a sum of squared distances. Distortion therefore falls monotonically, and because only finitely many assignments exist, the loop terminates.

Two details in the implementations are not in the algorithm as stated in Section 3.1. Convergence is tested on the *assignments* rather than on the distortion, and because assignments are discrete that test is exact — no tolerance, no stopping rule to tune, unlike gradient descent. And a cluster can lose every one of its points, which the centroid update step does not define; here we leave such a centroid where it is. Library implementations differ on this, and some re-seed it instead.

3.8.1 Python

```
import numpy as np

def kmeans(X, K, rng, max_steps=100):
    c = X[rng.choice(len(X), K, replace=False)].copy()
    h = np.full(len(X), -1)
    for _ in range(max_steps):
        d2 = ((X[:, None, :] - c[None, :, :]) ** 2).sum(axis=2)
        h_new = d2.argmin(axis=1)
        if np.array_equal(h_new, h): # discrete, so this test is exact
            break
        h = h_new
    for i in range(K):
        if np.any(h == i): # an emptied cluster keeps its centroid
```

```

        c[i] = X[h == i].mean(axis=0)
    return c, h, ((X - c[h]) ** 2).sum()

rng = np.random.default_rng(0)
X = np.vstack([rng.normal(m, 0.6, size=(60, 2)) for m in ([0, 0], [4, 0], [2, 3])])

runs = [kmeans(X, 3, rng) for _ in range(10)]
c, h, _ = min(runs, key=lambda r: r[2])
print(round(min(r[2] for r in runs), 2), round(max(r[2] for r in runs), 2))

```

127.01 471.96

```
print(np.round(c[np.argsort(c[:, 0])], 2).tolist())
```

```
[[-0.0, 0.1], [1.99, 2.98], [3.81, 0.03]]
```

3.8.2 R

```

kmeans_fit <- function(X, K, max_steps = 100) {
  c <- X[sample(nrow(X), K), , drop = FALSE]
  h <- rep(-1L, nrow(X))
  for (step in seq_len(max_steps)) {
    d2 <- vapply(seq_len(K), function(i) rowSums(sweep(X, 2, c[i, ]^2,
      numeric(nrow(X)))),
    h_new <- max.col(-d2, ties.method = "first")
    if (identical(h_new, h)) break # discrete, so this test is exact
    h <- h_new
    for (i in seq_len(K)) {
      if (any(h == i)) c[i, ] <- colMeans(X[h == i, , drop = FALSE])
    }
  }
  list(centers = c, h = h, distortion = sum((X - c[h, ])^2))
}

set.seed(0)
means <- rbind(c(0, 0), c(4, 0), c(2, 3))
X <- do.call(rbind, lapply(1:3, function(k)
  cbind(rnorm(60, means[k, 1], 0.6), rnorm(60, means[k, 2], 0.6))))

runs <- lapply(1:10, function(i) kmeans_fit(X, 3))
d <- vapply(runs, function(r) r$distortion, numeric(1))
best <- runs[[which.min(d)]]
cat(round(min(d), 2), round(max(d), 2), "\n")

```

115.06 470.89

```
print(round(best$centers[order(best$centers[, 1]), ], 2))
```

```

      [,1] [,2]
[1,] 0.00 0.01
[2,] 2.02 3.02
[3,] 3.94 0.08

```

3.8.3 Julia

```
using Random, Statistics
```

```
function kmeans_fit(X, K; max_steps = 100)
```

```

c = X[randperm(size(X, 1))[1:K], :]
h = fill(-1, size(X, 1))
for _ in 1:max_steps
    d2 = [sum((X[n, :] .- c[i, :]) .^ 2) for n in axes(X, 1), i in 1:K]
    h_new = [argmin(view(d2, n, :)) for n in axes(X, 1)]
    h_new == h && break # discrete, so this test is exact
    h = h_new
    for i in 1:K
        any(h .== i) && (c[i, :] = vec(mean(X[h .== i, :], dims = 1)))
    end
end
return c, h, sum((X .- c[h, :]) .^ 2)
end;

```

```

Random.seed!(0);
means = [0.0 0.0; 4.0 0.0; 2.0 3.0];
X = vcat([means[k:k, :] .+ 0.6 .* randn(60, 2) for k in 1:3]...);

runs = [kmeans_fit(X, 3) for _ in 1:10];
d = [r[3] for r in runs];
c = runs[argmin(d)][1];
println(round(minimum(d), digits = 2), " ", round(maximum(d), digits = 2))

```

125.05 125.05

```
println(round.(c[sortperm(c[:, 1]), :], digits = 2))
```

[-0.0 0.07; 2.08 3.01; 3.95 0.02]

```
km_d = [r[2] for r in runs]
```

```

# The same seed and data as the Julia tab, restarted two hundred times.
Random.seed!(0);
X200 = vcat([means[k:k, :] .+ 0.6 .* randn(60, 2) for k in 1:3]...);
d200 = [kmeans_fit(X200, 3)[3] for _ in 1:200];

```

```

# The distortions the prose quotes, from the three tabs.
km_py <- range(py$km_d)
km_r <- range(d)
km_jl_d <- JuliaCall::julia_eval("d")
km_jl <- range(km_jl_d)
km_jl200 <- max(JuliaCall::julia_eval("d200"))
stopifnot(JuliaCall::julia_eval("X200 == X"))
fmt2 <- function(v) formatC(v, format = "f", digits = 2)
fmt1 <- function(v) formatC(v, format = "f", digits = 1)
# How many of Julia's runs land on its best value. Which initial centroids
# randperm draws differs between Julia versions, so this is reported rather
# than assumed.
km_jl_n_best <- sum(abs(km_jl_d - km_jl[1]) < 0.005)
km_jl_n_other <- length(km_jl_d) - km_jl_n_best
km_jl_text <- if (km_jl_n_other == 0) {
    paste0(
        "Julia's ten runs all return $", fmt2(km_jl[1]), "$.\n",
        "This result is important, because it is\n",
        "the trap rather than a contradiction --- raise it to two hundred ",
        "restarts on\n",
        "that same seed and a worst of $", fmt2(km_jl200), "$ appears."
    )
}

```

```

} else {
  paste0(
    km_jl_n_best, " of Julia's ten runs return $", fmt2(km_jl[1]), "$, ",
    if (km_jl_n_other == 1) "and the other returns $" else
      paste0("and the other ", km_jl_n_other, " return up to $"),
    fmt2(km_jl[2]), "$.\n",
    "The agreeing majority is important, because it is the trap:\n",
    "had those ten starts not included ",
    if (km_jl_n_other == 1) "the one" else "the ones",
    " that ended at a worse value,\n",
    "nothing in the output would have hinted that such a start exists."
  )
}

```

Each block runs the whole thing ten times and keeps the best run, as Section 3.2 suggests with its “several random clusterings”. Look at the spread. Python reports a best of 127.01 against a worst of 471.96, and R 115.06 against 470.89: the same algorithm on the same kind of data, landing 3.7 and 4.1 times worse depending only on which three points it started from. The spread makes non-convexity concrete.

Julia’s ten runs all return 125.05. This result is important, because it is the trap rather than a contradiction — raise it to two hundred restarts on that same seed and a worst of 577.04 appears. Runs that agree are not evidence that the optimum has been found. They are evidence that those draws happened to land well, and we cannot tell those apart from inside the run.

All three recover centroids close to the (0, 0), (4, 0) and (2, 3) that we generated the data around, once we sort the rows — the labels themselves are arbitrary, since nothing ties cluster 1 to any particular group.

i Watch

- Broderick introduces the k -means algorithm, from 10:20^a, and how to initialize it, from 36:07^b [3].
- Hastie and Tibshirani cover k -means clustering^c [6], video 12.3, and cluster data in their Python lab^d.
- von Luxburg’s lecture on k -means^e [1, Pt. 33] and Ng’s lecture on k -means^f, from 5:37 [7], Lecture 13, give two more views.
- Starmer explains k -means clustering^g [8].

^a<https://www.youtube.com/watch?v=BaZWcSq3IuI&t=620s>

^b<https://www.youtube.com/watch?v=BaZWcSq3IuI&t=2167s>

^chttps://www.youtube.com/watch?v=ded_NQqOe7I

^d<https://www.youtube.com/watch?v=xhLD4nEEVIE>

^e<https://www.youtube.com/watch?v=SQUjEPcm508>

^f<https://www.youtube.com/watch?v=rVfZHWTwXSA&t=337s>

^g<https://www.youtube.com/watch?v=4b5d3muPQmA>

i Read

GeeksforGeeks explains k -means clustering^a [9].

^a<https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/>

4 Spectral clustering

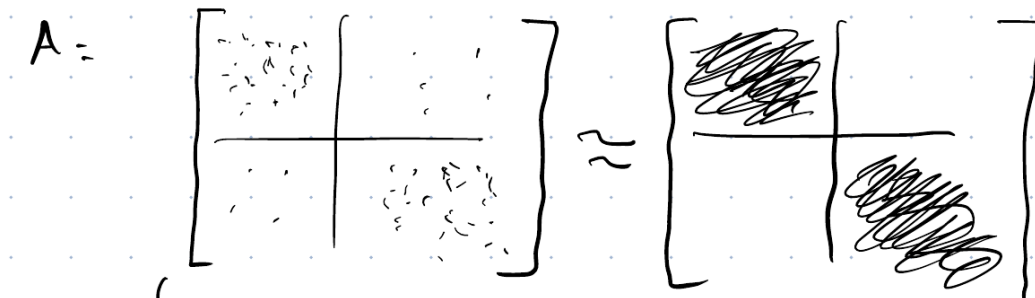
Graph-cut clustering turns the data into a graph and cuts it. Spectral clustering is the standard method [10].

Build a similarity matrix A , where $A_{ij} = \begin{cases} \text{big,} & i \text{ similar to } j \\ \text{small,} & \text{dissimilar} \end{cases}$ then

1. Form the graph Laplacian $\mathcal{L} = \Delta - A$, where $\Delta = \text{diag}(A\vec{1})$ is the diagonal matrix of row sums of A .

The script $\mathcal{L}$ and the Δ keep this apart from the likelihood L and the dimension D , which mean other things elsewhere in this chapter.

2. Find the eigenvectors of $\mathcal{L}$ and keep the K with the smallest eigenvalues.
3. Stack those eigenvectors as columns, giving each original point a row of K numbers, and cluster those rows using k-means.



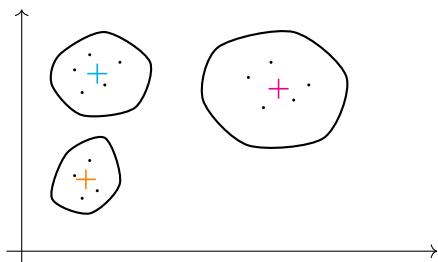
Two common choices of similarity:

$$A_{ij} = \begin{cases} x_i^T x_j \\ e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}} \end{cases} \quad \uparrow \text{scale}$$

The second is the Gaussian kernel that we used for kernel smoothing in Non-Parametric Models, and σ plays the same role.

Compared with k-means, spectral clustering can follow a cluster of any shape. K-means assigns each point to the nearest center, so its clusters are the cells of a Voronoi diagram, which are convex. Spectral clustering never measures distance to a center in the input space. It asks only which points are connected to which, so a long curved band of points that are each close to their neighbors holds together even though its two ends are far apart. It does run k-means at the end, but on the eigenvector embedding rather than on the original coordinates, and the embedding has already straightened the band. Other ways to relax k-means' assumption are to change the distance metric, to build the graph from each point's nearest neighbors rather than from a kernel over every pair, or to fit a mixture of Gaussians, whose components are ellipses rather than Voronoi cells — a mixture-model view that Section 8.1 takes up.

Connecting each point to its k nearest neighbors — a k chosen independently of the number of clusters K — is the usual way to turn a set of points into a graph in the first place, and it makes the similarity matrix A sparse.



i Watch

von Luxburg's lectures cover:

- spectral graph theory^a [1, Pt. 35]
- unnormalized spectral clustering^b [1, Pt. 36]

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

^b<https://www.youtube.com/watch?v=5FQfEjDic84>

i Read

GeeksforGeeks explains spectral clustering^a [11].

^a<https://www.geeksforgeeks.org/machine-learning/spectral-clustering/>

5 Hierarchical clustering

Both algorithms so far need K before they start. Agglomerative clustering does not. It begins with every point in its own cluster, repeatedly merges the two closest clusters, and stops only when one cluster holds everything — producing not a single partition but a nested family of them, one for every number of clusters from N down to 1. The record of which clusters merged, and at what height, is a **dendrogram**.

5.1 Linkage criteria

The algorithm needs a dissimilarity $D(C_i, C_j)$ between *clusters*, built out of the dissimilarity $d(x, x')$ between *points*. That construction is the **linkage criterion**, and given d it is the only choice the algorithm makes.

- **Single linkage** takes the closest cross-cluster pair:

$$D(C_i, C_j) = \min_{x \in C_i, x' \in C_j} d(x, x')$$

- **Complete linkage** takes the farthest:

$$D(C_i, C_j) = \max_{x \in C_i, x' \in C_j} d(x, x')$$

- **Average linkage** takes the mean over all cross-cluster pairs:

$$D(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{x \in C_i} \sum_{x' \in C_j} d(x, x')$$

- **Ward linkage** reduces no set of pairwise distances at all. It asks how far the within-cluster sum of squares would rise if the two clusters merged:

$$D(C_i, C_j) = \frac{|C_i||C_j|}{|C_i| + |C_j|} \|\mu_i - \mu_j\|_2^2$$

where μ_i is the mean of C_i [12]. That quantity is exactly the increase in $\sum_k \sum_{x \in C_k} \|x - \mu_k\|_2^2$ caused by the merge, so greedily merging the cheapest pair is a bottom-up descent on the k-means objective.

Ward's heights therefore sit on a different scale from the other three: for two singletons it returns $\frac{1}{2}d(x, x')^2$ rather than $d(x, x')$. Keep that in mind when comparing heights across linkages.

5.2 A worked merge

We have four weather stations and a dissimilarity for every pair:

	W	X	Y	Z
W	—	0.12	0.30	0.50
X	0.12	—	0.24	0.46
Y	0.30	0.24	—	0.35

	W	X	Y	Z
Z	0.50	0.46	0.35	—

Take average linkage. The smallest entry is $d(W, X) = 0.12$, so W and X merge first, at height 0.12. Rebuild the table against the new cluster:

$$D(W+X, Y) = \frac{0.30 + 0.24}{2} = 0.27, \quad D(W+X, Z) = \frac{0.50 + 0.46}{2} = 0.48$$

	W+X	Y	Z
W+X	—	0.27	0.48
Y	0.27	—	0.35
Z	0.48	0.35	—

Now 0.27 is smallest, so Y joins at that height. One distance remains, and it needs care:

$$\begin{aligned} D(Y+W+X, Z) &= \frac{d(Y, Z) + d(W, Z) + d(X, Z)}{3} \\ &= \frac{0.35 + 0.50 + 0.46}{3} \\ &= \frac{1.31}{3} \\ &\approx 0.437 \end{aligned}$$

This is not $\frac{1}{2}(0.35 + 0.48) = 0.415$. Average linkage averages over the original *point* pairs, so when we rebuild the table, the two old distances enter weighted by cluster size — one point in Y against two in W+X. Single and complete linkage have no such subtlety: a minimum of minima is a minimum, and a maximum of maxima is a maximum, whatever the cluster sizes.

Z joins last, at height $\frac{1.31}{3} \approx 0.437$, and the dendrogram is complete.

5.3 Cutting the dendrogram

A dendrogram is not a clustering; it is every clustering at once. Draw a horizontal line at height h and read off the branches it severs: that is the partition you get by refusing any merge costing more than h . Cutting the tree of Section 5.2 at $h = 0.3$ leaves $\{W, X, Y\}$ and $\{Z\}$, and cutting at $h = 0.2$ leaves $\{W, X\}$, $\{Y\}$ and $\{Z\}$. Asking for exactly K clusters is the same operation counted from the other end: stop the merge sequence with K clusters still standing.

That is the practical appeal. We choose the number of clusters after looking at the tree rather than before running the algorithm, and a tall gap between consecutive merge heights is the dendrogram's version of the elbow.

Exercise 5.1 (Compare single and complete linkage on a line). Five points on the real line sit at 0, 1, 2.5, 4.5 and 6.6.

Cluster them by hand with agglomerative clustering and single linkage. Say which clusters merge at each step and at what height, draw the tree, and give the two clusters you get by stopping at $K = 2$.

Solution

Solution 5.1. The distance between two clusters is the smallest distance between a point in each. On a line, that is the gap between neighbours, and the gaps are 1, 1.5, 2 and 2.1, left to right. So the merges go in that order: $\{0, 1\}$ at height 1, then 2.5 joins at 1.5, then 4.5 at 2, then 6.6 at 2.1. Stopping with two clusters left gives $\{0, 1, 2.5, 4.5\}$ and $\{6.6\}$ (Figure 1).

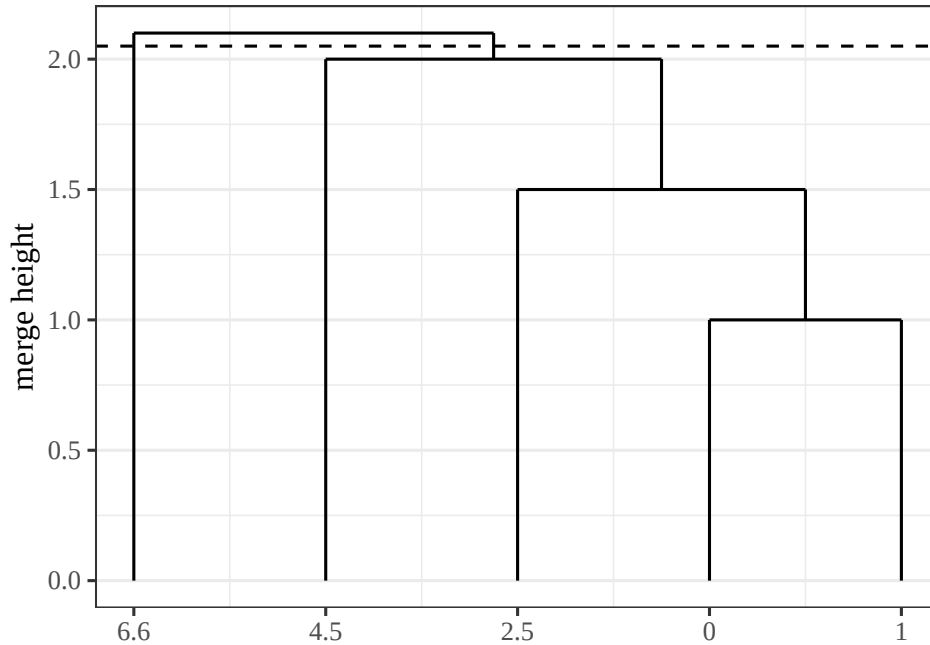


Figure 1: Single-linkage dendrogram of the five points 0, 1, 2.5, 4.5, and 6.6 on a line. The dashed line cuts the tree into two clusters.

Exercise 5.2 (Compare single and complete linkage on a line (part 2)). Use the setup of Exercise 5.1. Cluster them by hand with agglomerative clustering and complete linkage. Say which clusters merge at each step and at what height, draw the tree, and give the two clusters you get by stopping at $K = 2$.

Solution

Solution 5.2. The distance between two clusters is the largest distance between a point in each. The smallest pairwise distance is still $d(0, 1) = 1$, so $\{0, 1\}$ merges first. Next, $d(2.5, 4.5) = 2$ is smaller than $d(4.5, 6.6) = 2.1$, and the complete distance from $\{0, 1\}$ to 2.5 is 2.5, so $\{2.5, 4.5\}$ merges at height 2. Now 6.6 is 4.1 from $\{2.5, 4.5\}$ (its farthest member is 2.5), while $\{0, 1\}$ is 4.5 from $\{2.5, 4.5\}$ (its farthest pair is 0 and 4.5). So 6.6 joins $\{2.5, 4.5\}$ at 4.1, and the last merge, of $\{0, 1\}$ with $\{2.5, 4.5, 6.6\}$, is at height 6.6. Stopping with two clusters left gives $\{0, 1\}$ and $\{2.5, 4.5, 6.6\}$ (Figure 2).

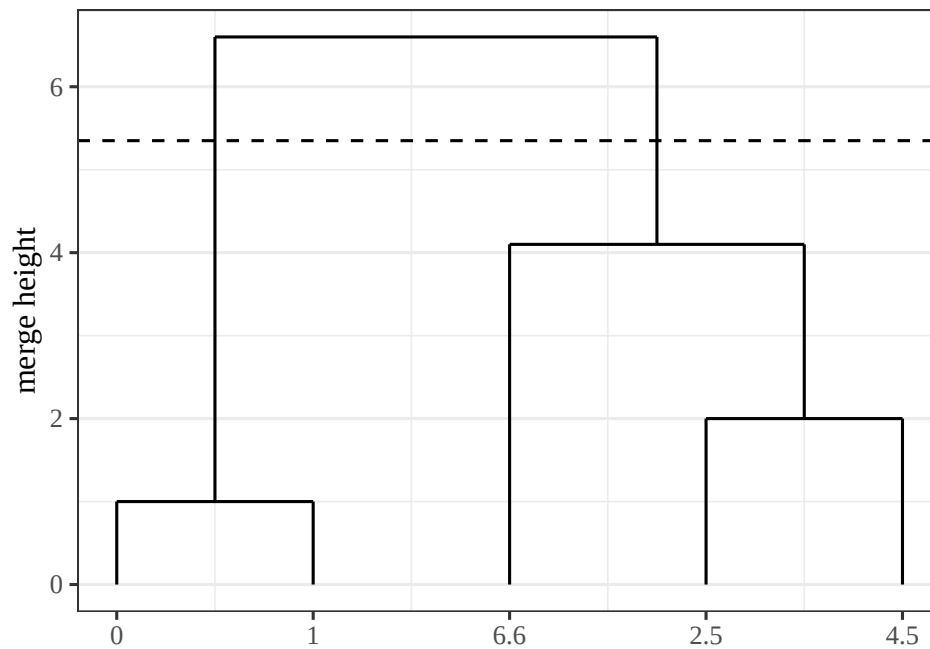


Figure 2: Complete-linkage dendrogram of the five points 0, 1, 2.5, 4.5, and 6.6. The dashed line cuts the tree into two clusters, which differ from the single-linkage ones.

Exercise 5.3 (Compare single and complete linkage on a line (part 3)). Use the setup of Exercise 5.1. Do the two linkages agree?

Solution

Solution 5.3. The linkages disagree. Single linkage looks only at the nearest pair, so it follows the chain of small gaps and cuts off the point with the largest gap, 6.6. Complete linkage penalizes a cluster's whole spread, so it prefers to keep $\{0, 1\}$ compact and groups the three points on the right.

5.3.1 Python

```
import numpy as np
from scipy.cluster.hierarchy import fcluster, linkage

# The five points of the exercise, as a column because linkage() expects
# one row per observation.
x_line = np.array([0, 1, 2.5, 4.5, 6.6]).reshape(-1, 1)

for method in ["single", "complete"]:
    tree = linkage(x_line, method=method)
    # Column 2 of the result is each merge's height; fcluster() cuts the
    # tree at K = 2 and labels each point with its cluster.
    print(method, tree[:, 2], fcluster(tree, 2, criterion="maxclust"))

single [1.  1.5 2.  2.1] [1 1 1 1 2]
complete [1.  2.  4.1 6.6] [1 1 2 2 2]
```

5.3.2 R

The five points of the exercise.

```
x_line <- c(0, 1, 2.5, 4.5, 6.6)
```

```
for (method in c("single", "complete")) {
  tree <- hclust(dist(x_line), method = method)
  # `height` holds each merge's height; cutree() cuts the tree at K = 2 and
  # labels each point with its cluster.
  cat(method, ":", tree$height, "| clusters:", cutree(tree, k = 2), "\n")
}
```

```
single : 1 1.5 2 2.1 | clusters: 1 1 1 1 2
```

```
complete : 1 2 4.1 6.6 | clusters: 1 1 2 2 2
```

Both tools report the heights found by hand, and the two linkages give different two-cluster partitions.

Exercise 5.4 (Build a dendrogram by hand with average linkage). Five points have the pairwise distances below.

	A	B	C	D	E
A	—	1.0	3.0	4.4	9.2
B	1.0	—	2.0	3.4	8.2
C	3.0	2.0	—	1.4	6.2
D	4.4	3.4	1.4	—	4.8
E	9.2	8.2	6.2	4.8	—

Run agglomerative clustering with average linkage by hand. At each step, say which two clusters merge and at what height, and rebuild the distance table against the new cluster.

Solution

Solution 5.4. The smallest entry is $d(A, B) = 1.0$, so A and B merge first, at height 1.0. The next smallest, $d(C, D) = 1.4$, is untouched by that merge, so C and D merge second, at height 1.4. Average linkage averages over the original point pairs, so each new distance is a mean over the pairs that cross the two clusters:

$$\begin{aligned} D(A+B, C+D) &= \frac{3.0 + 4.4 + 2.0 + 3.4}{4} & D(A+B, E) &= \frac{9.2 + 8.2}{2} & D(C+D, E) &= \frac{6.2 + 4.8}{2} \\ &= 3.2, & &= 8.7, & &= 5.5. \end{aligned}$$

The smallest of these is 3.2, so A+B and C+D merge at height 3.2. E is the only cluster left outside, and it joins last, at height

$$\begin{aligned} D(A+B+C+D, E) &= \frac{9.2 + 8.2 + 6.2 + 4.8}{4} \\ &= 7.1. \end{aligned}$$

Both old clusters hold two points, so this happens to equal the plain mean of 8.7 and 5.5. With unequal sizes the two old distances would be weighted by the cluster sizes, as in Section 5.2.

5.3.3 Python

```

import numpy as np
from scipy.cluster.hierarchy import linkage
from scipy.spatial.distance import squareform

# The distance table of the exercise, as a symmetric matrix.
d_five = np.array([
    [0.0, 1.0, 3.0, 4.4, 9.2],
    [1.0, 0.0, 2.0, 3.4, 8.2],
    [3.0, 2.0, 0.0, 1.4, 6.2],
    [4.4, 3.4, 1.4, 0.0, 4.8],
    [9.2, 8.2, 6.2, 4.8, 0.0],
])

# linkage() wants the condensed form (the upper triangle) of the table.
# Each row of the result is one merge: the two clusters joined, the height,
# and the size of the new cluster.
linkage(squareform(d_five), method="average")

array([[0. , 1. , 1. , 2. ],
       [2. , 3. , 1.4, 2. ],
       [5. , 6. , 3.2, 4. ],
       [4. , 7. , 7.1, 5. ]])

```

5.3.4 R

```

# The distance table of the exercise, as a symmetric matrix.
d_five <- matrix(c(
  0.0, 1.0, 3.0, 4.4, 9.2,
  1.0, 0.0, 2.0, 3.4, 8.2,
  3.0, 2.0, 0.0, 1.4, 6.2,
  4.4, 3.4, 1.4, 0.0, 4.8,
  9.2, 8.2, 6.2, 4.8, 0.0
), nrow = 5, dimnames = list(LETTERS[1:5], LETTERS[1:5]))

# hclust() records each merge in `merge` and its height in `height`.
five_tree <- hclust(as.dist(d_five), method = "average")
five_tree$height

[1] 1.0 1.4 3.2 7.1

```

Both tools report the merge heights 1.0, 1.4, 3.2 and 7.1, the heights found by hand.

Exercise 5.5 (Build a dendrogram by hand with average linkage (part 2)). Use the setup of Exercise 5.4. Draw the dendrogram.

Solution

Solution 5.5. The dendrogram joins A and B at 1.0, C and D at 1.4, those two clusters at 3.2, and E at 7.1 (Figure 3).

```

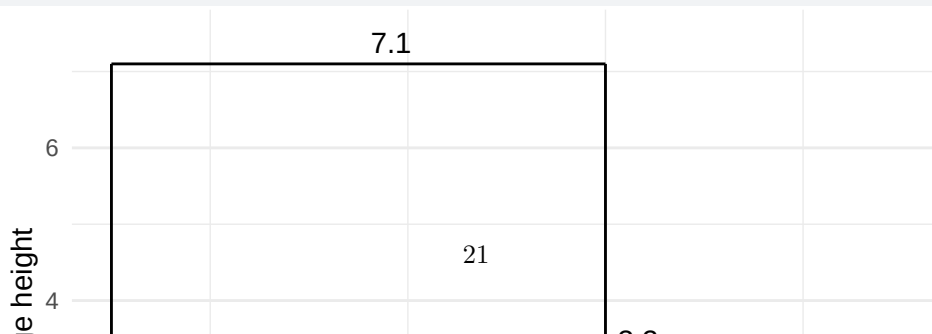
# Each row of `merge` joins two items, a negative index for a point
# and a positive index for the cluster made at that earlier row.
merge_rows <- five_tree$merge
heights <- five_tree$height
leaf_x <- match(seq_along(five_tree$order), five_tree$order)

item_x <- function(item, cluster_x) {
  if (item < 0) leaf_x[-item] else cluster_x[item]
}
item_height <- function(item) if (item < 0) 0 else heights[item]

cluster_x <- numeric(nrow(merge_rows))
bars <- list()
for (i in seq_len(nrow(merge_rows))) {
  left <- merge_rows[i, 1]
  right <- merge_rows[i, 2]
  xs <- c(item_x(left, cluster_x), item_x(right, cluster_x))
  cluster_x[i] <- mean(xs)
  bars[[i]] <- data.frame(
    x = c(xs[1], xs[1], xs[2]),
    xend = c(xs[1], xs[2], xs[2]),
    y = c(item_height(left), heights[i], item_height(right)),
    yend = c(heights[i], heights[i], heights[i])
  )
}
bars <- do.call(rbind, bars)
merge_labels <- data.frame(
  x = cluster_x,
  y = heights,
  label = format(round(heights, 1), nsmall = 1)
)

ggplot2::ggplot() +
  ggplot2::geom_segment(
    data = bars,
    ggplot2::aes(x = x, xend = xend, y = y, yend = yend)
  ) +
  ggplot2::geom_text(
    data = merge_labels,
    ggplot2::aes(x = x, y = y, label = label),
    hjust = -0.3,
    vjust = -0.5
  ) +
  ggplot2::scale_y_continuous(expand = ggplot2::expansion(mult = c(0.02, 0.1))) +
  ggplot2::scale_x_continuous(
    breaks = seq_along(five_tree$order),
    labels = five_tree$labels[five_tree$order]
  ) +
  ggplot2::labs(x = NULL, y = "merge height") +
  ggplot2::theme_minimal() +
  ggplot2::theme(panel.grid.major.x = ggplot2::element_blank())

```



Exercise 5.6 (Cut the dendrogram to get a chosen number of clusters). For the dendrogram of Exercise 5.5, which heights h give two clusters?

Solution

Solution 5.6. A cut at height h refuses every merge that costs more than h . The merges happened at 1.0, 1.4, 3.2 and 7.1 (Solution 5.3.4).

A cut with $3.2 \leq h < 7.1$ keeps the first three merges and refuses the last.

Exercise 5.7 (Cut the dendrogram to get a chosen number of clusters (part 2)). Use the setup of Exercise 5.6.

List the two clusters given by a cut in the range of heights found in Exercise 5.6.

Solution

Solution 5.7. That cut gives **two** clusters: $\{A, B, C, D\}$ and $\{E\}$.

Exercise 5.8 (Cut the dendrogram to get a chosen number of clusters (part 3)). Use the setup of Exercise 5.6.

For the dendrogram of Exercise 5.5, which heights h give three clusters?

Solution

Solution 5.8. A cut at height h refuses every merge that costs more than h , as in Solution 5.6.

A cut with $1.4 \leq h < 3.2$ keeps the first two merges and refuses the last two.

Exercise 5.9 (Cut the dendrogram to get a chosen number of clusters (part 4)). Use the setup of Exercise 5.6.

List the three clusters given by a cut in the range of heights found in Exercise 5.8.

Solution

Solution 5.9. That cut gives **three** clusters: $\{A, B\}$, $\{C, D\}$ and $\{E\}$.

Exercise 5.10 (Cut the dendrogram to get a chosen number of clusters (part 5)). Use the setup of Exercise 5.6.

Does the dendrogram favor two clusters (Exercise 5.6) or three (Exercise 5.8), and why?

Solution

Solution 5.10. The range of heights that gives two clusters is $7.1 - 3.2 = 3.9$ long, and the range that gives three is $3.2 - 1.4 = 1.8$ long. A tall gap between consecutive merge heights shows that the clusters stayed well separated across a wide range of cuts, so the dendrogram favors two clusters. It is the dendrogram's version of the elbow.

5.4 In code

The four criteria differ only in how `height` reduces a pair of clusters, so one implementation can run all four over the same seven points. The code prints each merge as the cluster it produced followed by the height it happened at.

5.4.1 Python

```
import itertools
import math

X = [(0.0, 0.0), (0.9, 0.5), (2.1, 0.1), (3.2, 0.6),
      (6.0, 3.0), (6.8, 3.5), (9.2, 0.2)]
labels = "ABCDEFGF"

def height(ci, cj, linkage):
    if linkage == "ward":
        # rise in within-cluster sum of squares
        mi = [sum(X[k][t] for k in ci) / len(ci) for t in (0, 1)]
        mj = [sum(X[k][t] for k in cj) / len(cj) for t in (0, 1)]
        gap = sum((mi[t] - mj[t]) ** 2 for t in (0, 1))
        return len(ci) * len(cj) / (len(ci) + len(cj)) * gap
    d = [math.dist(X[a], X[b]) for a in ci for b in cj]
    if linkage == "single":
        return min(d)
    if linkage == "complete":
        return max(d)
    return sum(d) / len(d)

def agglomerate(linkage):
    clusters = [[i] for i in range(len(X))]
    steps = []
    while len(clusters) > 1:
        pairs = list(itertools.combinations(range(len(clusters)), 2))
        a, b = min(pairs, key=lambda p: height(clusters[p[0]], clusters[p[1]], linkage))
        h = height(clusters[a], clusters[b], linkage)
        merged = sorted(clusters[a] + clusters[b])
        steps.append("".join(labels[i] for i in merged) + "@" + str(round(h, 3)))
        clusters = [c for k, c in enumerate(clusters) if k not in (a, b)] + [merged]
    print(linkage, " ".join(steps))

for name in ["single", "complete", "average", "ward"]:
    agglomerate(name)
```

```
single EF@0.943 AB@1.03 CD@1.208 ABCD@1.265 ABCDEF@3.688 ABCDEFG@4.08
complete EF@0.943 AB@1.03 CD@1.208 ABCD@3.256 EFG@4.252 ABCDEFG@9.202
average EF@0.943 AB@1.03 CD@1.208 ABCD@2.231 EFG@4.166 ABCDEFG@6.354
ward EF@0.445 AB@0.53 CD@0.73 ABCD@4.85 EFG@11.428 ABCDEFG@63.745
```

5.4.2 R

```
X <- matrix(c(0.0, 0.0, 0.9, 0.5, 2.1, 0.1, 3.2, 0.6,
              6.0, 3.0, 6.8, 3.5, 9.2, 0.2), ncol = 2, byrow = TRUE)
labels <- strsplit("ABCDEFGF", "")[[1]]

height <- function(ci, cj, linkage) {
  if (linkage == "ward") {
    # rise in within-cluster sum of squares
    mi <- colMeans(X[ci, , drop = FALSE])
    mj <- colMeans(X[cj, , drop = FALSE])
    gap <- sum((mi - mj)^2)
    return(length(ci) * length(cj) / (length(ci) + length(cj)) * gap)
  }
  d <- as.vector(outer(ci, cj, Vectorize(function(a, b) sqrt(sum((X[a, ] - X[b, ])^2))))))
```

```

    if (linkage == "single") return(min(d))
    if (linkage == "complete") return(max(d))
    mean(d)
  }

agglomerate <- function(linkage) {
  clusters <- lapply(seq_len(nrow(X)), function(i) i)
  steps <- character(0)
  while (length(clusters) > 1) {
    pairs <- combn(length(clusters), 2)
    hs <- apply(pairs, 2, function(p) height(clusters[[p[1]]], clusters[[p[2]]], linkage))
    k <- which.min(hs)
    a <- pairs[1, k]; b <- pairs[2, k]
    merged <- sort(c(clusters[[a]], clusters[[b]]))
    steps <- c(steps, paste0(paste(labels[merged], collapse = ""), "@", round(hs[k], 3)))
    clusters <- c(clusters[-c(a, b)], list(merged))
  }
  cat(linkage, " ", paste(steps, collapse = " "), "\n", sep = "")
}

for (name in c("single", "complete", "average", "ward")) agglomerate(name)

```

```

single EF@0.943 AB@1.03 CD@1.208 ABCD@1.265 ABCDEF@3.688 ABCDEFG@4.08
complete EF@0.943 AB@1.03 CD@1.208 ABCD@3.256 EFG@4.252 ABCDEFG@9.202
average EF@0.943 AB@1.03 CD@1.208 ABCD@2.231 EFG@4.166 ABCDEFG@6.354
ward EF@0.445 AB@0.53 CD@0.73 ABCD@4.85 EFG@11.428 ABCDEFG@63.745

```

5.4.3 Julia

```

X = [(0.0, 0.0), (0.9, 0.5), (2.1, 0.1), (3.2, 0.6),
      (6.0, 3.0), (6.8, 3.5), (9.2, 0.2)];
labels = "ABCDEFGF";

function height(ci, cj, linkage)
    if linkage == "ward" # rise in within-cluster sum of squares
        mi = (sum(X[k][1] for k in ci) / length(ci), sum(X[k][2] for k in ci) / length(ci))
        mj = (sum(X[k][1] for k in cj) / length(cj), sum(X[k][2] for k in cj) / length(cj))
        gap = (mi[1] - mj[1])^2 + (mi[2] - mj[2])^2
        return length(ci) * length(cj) / (length(ci) + length(cj)) * gap
    end
    d = [sqrt((X[a][1] - X[b][1])^2 + (X[a][2] - X[b][2])^2) for a in ci for b in cj]
    linkage == "single" && return minimum(d)
    linkage == "complete" && return maximum(d)
    return sum(d) / length(d)
end;

function agglomerate(linkage)
    clusters = [[i] for i in 1:length(X)]
    steps = String[]
    while length(clusters) > 1
        pairs = [(i, j) for i in 1:length(clusters) for j in (i + 1):length(clusters)]
        hs = [height(clusters[p[1]], clusters[p[2]], linkage) for p in pairs]
        k = argmin(hs)
        a, b = pairs[k]
        merged = sort(vcat(clusters[a], clusters[b]))
        push!(steps, join(labels[i] for i in merged) * "@" * string(round(hs[k], digits = 3)))
    end
end

```

```

        clusters = [c for (i, c) in enumerate(clusters) if i != a && i != b]
        push!(clusters, merged)
    end
    println(linkage, " ", join(steps, " "))
end;

for name in ["single", "complete", "average", "ward"]
    agglomerate(name)
end

```

```

single EF@0.943 AB@1.03 CD@1.208 ABCD@1.265 ABCDEF@3.688 ABCDEFG@4.08
complete EF@0.943 AB@1.03 CD@1.208 ABCD@3.256 EFG@4.252 ABCDEFG@9.202
average EF@0.943 AB@1.03 CD@1.208 ABCD@2.231 EFG@4.166 ABCDEFG@6.354
ward EF@0.445 AB@0.53 CD@0.73 ABCD@4.85 EFG@11.428 ABCDEFG@63.745

```

```

# The two distances the prose compares, on the R tab's seven points.
pt_dist <- function(a, b) sqrt(sum((X[a, ] - X[b, ])^2))
d_de <- pt_dist(4, 5)
d_fg <- pt_dist(6, 7)
fmt3 <- function(v) formatC(v, format = "f", digits = 3)

```

All four agree on the first four merges. The pairs AB, CD and EF are tight, and ABCD follows. They differ on the fifth merge, and the difference is structural rather than numerical. Single linkage joins ABCD to EF, because the closest single pair between them, $d(D, E) = 3.688$, beats the 4.080 separating F from G. The other three measure the whole spread instead, and by that measure G is the better partner for EF. So cutting at $K = 2$ gives $\{A, \dots, F\}$ and $\{G\}$ under single linkage but $\{A, B, C, D\}$ and $\{E, F, G\}$ under the other three, on identical data with an identical d . Single linkage's willingness to follow one short hop at a time lets it trace out non-convex shapes, and equally lets a single bridging point chain two clusters together.

Exercise 5.11 (Recover two interleaved crescents). A data set has 300 points in the plane, forming two interleaved crescents (the “two moons” shape) with a clear gap between them and little noise. You want the two crescents back as two clusters. Which would you expect to work: K -means with $K = 2$, or single-link agglomerative clustering cut at two clusters?

Solution

Solution 5.11. Single-link clustering works here, and K -means does not.

Exercise 5.12 (Recover two interleaved crescents (part 2)). Use the setup of Exercise 5.11. Say what single-link agglomerative clustering does to the crescents, and why.

Solution

Solution 5.12. Single linkage merges the two clusters with the closest pair of points between them. Within a crescent, neighbouring points are close, so the merges walk along each crescent one short hop at a time. The gap between the crescents is wider than those hops, so it is the last merge, and cutting at two clusters leaves exactly the two crescents.

The answer depends on the gap. If noise put points inside the gap, single linkage could chain the two crescents together through them.

Exercise 5.13 (Recover two interleaved crescents (part 3)). Use the setup of Exercise 5.11. Say what K -means with $K = 2$ does to the crescents, and why.

Solution

Solution 5.13. K -means assigns each point to its nearest centroid, so with $K = 2$ the boundary is the perpendicular bisector of the two centroids: a straight line. Each cluster is therefore a convex region. Neither crescent fits inside a convex region that excludes the other, so the line cuts across both crescents and the clusters mix them.

i Watch

- Hastie and Tibshirani's video on hierarchical clustering^a [6], video 12.4.
- von Luxburg's lecture covers linkage algorithms^b [1, Pt. 34].
- Starmer explains hierarchical clustering^c [13].

^a<https://www.youtube.com/watch?v=yktzn-Mr2Nw>

^b<https://www.youtube.com/watch?v=TQIdgl5SPGk>

^c<https://www.youtube.com/watch?v=7xHsRkOdVwo>

i Read

GeeksforGeeks explains:

- hierarchical clustering^a [14]
- linkage criteria for hierarchical clustering^b [15]

^a<https://www.geeksforgeeks.org/machine-learning/hierarchical-clustering/>

^b<https://www.geeksforgeeks.org/machine-learning/ml-types-of-linkages-in-clustering/>

6 Dimensionality reduction

Clustering assigns each point a label. The other common unsupervised task is to re-describe every point in fewer dimensions [16].

6.1 The curse of dimensionality

A second motive for reducing D has nothing to do with storage. As dimension grows, distances concentrate: the gap between a point's nearest and farthest neighbours shrinks relative to the typical distance between points, so “nearest” carries less and less information. Under broad conditions on the data distribution that relative gap converges to zero, and the effect is measurable in as few as 10 to 15 dimensions [17].

K-means, spectral clustering and hierarchical clustering all compare distances and do nothing else, so they degrade together. The assignment step of k-means shows it most plainly. If every centroid is nearly equidistant from a point, the arg min is settled by noise, and a small change to the data or to the initialization produces a completely different clustering from the same objective.

Two responses are standard. Reduce the dimension before clustering. Or change the metric: within the Minkowski family, smaller p concentrates more slowly, so Manhattan preserves more contrast than Euclidean at the same dimension [18], and cosine similarity discards magnitude entirely, which helps when the structure worth finding lies in a vector's direction rather than its length.

i Read

GeeksforGeeks explains the curse of dimensionality^a [19].

^a<https://www.geeksforgeeks.org/machine-learning/curse-of-dimensionality-in-machine-learning/>

6.2 Low-rank approximation and the SVD

We want $\mathbf{X}_r \approx \mathbf{X}$ with $\text{rank}(\mathbf{X}_r) = r$: a low-rank matrix is a simple matrix. Rank is the number of linearly independent columns or rows, equivalently the *smallest* number of rank-1 matrices that add up to $\mathbf{X}$. The word “smallest” is essential: any rank-1 matrix can be written as a sum of three rank-1 matrices, so without it the count would not be a single number.

The singular value decomposition², or SVD for short, writes $\mathbf{X}$ as exactly such a sum:

$$\begin{aligned}\mathbf{X} &= USV^\top \\ &= \sum_{i=1}^{\text{rank}(\mathbf{X})} \underbrace{u_i \sigma_i v_i^\top}_{\text{rank-1 matrix}} \quad \sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\text{rank}(\mathbf{X})}\end{aligned}$$

Because the singular values are ordered, truncating the sum keeps the largest pieces:

$$\mathbf{X}_r = \sum_{i=1}^r \sigma_i \underbrace{u_i v_i^\top}_{\text{rank-1 term}}$$

This truncation is optimal: no other matrix of rank r or less is closer to $\mathbf{X}$.

$$\mathbf{X}_r \text{ solves } \min_{\substack{Z \\ \text{rank}(Z) \leq r}} \|\mathbf{X} - Z\|_F$$

Here $\|A\|_F = \sqrt{\sum_{i,j} A_{ij}^2}$ is the Frobenius norm³, the entrywise analogue of the Euclidean vector norm⁴. The constraint is $\text{rank}(Z) \leq r$ rather than $= r$, since that is the set the truncation minimizes over.

i Read

GeeksforGeeks explains the singular value decomposition^a [20].

^a<https://www.geeksforgeeks.org/data-science/singular-value-decomposition-svd/>

6.3 Principal component analysis

Principal component analysis, or PCA for short, is that same truncation applied to the **centered** data [21]. Write

$$\mathbf{X}^c = \mathbf{X} - \bar{\mathbf{1}} \bar{x}^\top, \quad \bar{x} = \frac{1}{N} \sum_{i=1}^N x_{(i)}$$

for the data matrix with each column’s mean subtracted, and take its SVD $\mathbf{X}^c = USV^\top$. Then

$$\mathbf{X}^c \approx \sum_{i=1}^r \sigma_i u_i v_i^\top$$

is the best rank- r approximation to the centered data. Its directions v_i are the loadings, or principal components, and the $\sigma_i u_i$ are the scores.

Centering is not a detail to skip. If we run the SVD on $\mathbf{X}$ itself, the leading singular direction is pulled toward the mean of the data, so v_1 would describe where the data sits rather than how it varies — the one thing we ask PCA to describe.

²<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

³<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-frobenius-norm>

⁴<https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-euclidean-norm>

6.4 Maximizing variance

We can arrive at the same $\tilde{v}_i$ a second way, and this way gives the phrase “variance explained” its meaning. Instead of asking for the best rank- r approximation, ask for the single direction along which the data varies most.

Project the centered rows onto a unit vector $\tilde{u}$. The projections $t_i = \tilde{x}_{(i)}^c \top \tilde{u}$ have mean zero, because we centered the data, so their variance is

$$\frac{1}{N} \sum_{i=1}^N t_i^2 = \tilde{u} \top \hat{\Sigma} \tilde{u}, \quad \hat{\Sigma} = \frac{1}{N} \mathbf{X}^c \top \mathbf{X}^c$$

where $\hat{\Sigma}$ is the sample covariance matrix: the estimate, from these N rows, of the population’s variance matrix⁵ Σ . Maximizing that subject to $\|\tilde{u}\|_2 = 1$ gives the Lagrangian $\tilde{u} \top \hat{\Sigma} \tilde{u} - \lambda(\tilde{u} \top \tilde{u} - 1)$, whose stationary condition is

$$\hat{\Sigma} \tilde{u} = \lambda \tilde{u}$$

So $\tilde{u}$ must be an eigenvector of $\hat{\Sigma}$, and since $\tilde{u} \top \hat{\Sigma} \tilde{u} = \lambda$ at any such $\tilde{u}$, the maximizing choice is the one with the largest eigenvalue. Requiring the next direction to be orthogonal to the first and repeating gives the second eigenvector, and so on down.

This is the same answer as before, not a rival one. From $\mathbf{X}^c = USV^\top$,

$$\begin{aligned} \mathbf{X}^c \top \mathbf{X}^c &= VSU^\top USV^\top \\ &= VS^2V^\top \end{aligned} \quad \Rightarrow \quad \hat{\Sigma} = V\left(\frac{1}{N}S^2\right)V^\top$$

The eigenvectors of $\hat{\Sigma}$ are exactly the right singular vectors $\tilde{v}_i$, and the variance along $\tilde{v}_i$ is σ_i^2/N . Dividing by the total,

$$\frac{\sigma_i^2/N}{\sum_j \sigma_j^2/N} = \frac{\sigma_i^2}{\sum_j \sigma_j^2}$$

is the fraction of the total variance that component i accounts for, the quantity usually called the **explained variance ratio**.

Pythagoras explains why the two derivations agree. For a unit $\tilde{u}$, each centered point splits into its projection and what is left over,

$$\|\tilde{x}_{(i)}^c\|_2^2 = t_i^2 + \|\tilde{x}_{(i)}^c - t_i \tilde{u}\|_2^2$$

and summing over i gives $\|\mathbf{X}^c\|_F^2 = \sum_i t_i^2 + \sum_i \|\tilde{x}_{(i)}^c - t_i \tilde{u}\|_2^2$. The left side does not depend on $\tilde{u}$ at all. So maximizing the projected variance is the same as minimizing the squared reconstruction error. Pearson fit the line of closest fit to a cloud of points [22]. Hotelling later developed principal components as the directions of greatest variance [23]. The two formulations therefore describe one method.

Watch

Starmer works through principal component analysis^a [24] step by step.

^a<https://www.youtube.com/watch?v=FgakZw6K1QQ>

⁵<https://morrison-lab.github.io/pds/variance-covariance.html#def-cov-vec-x>

i Read

GeeksforGeeks explains principal component analysis^a [25].

^a<https://www.geeksforgeeks.org/data-analysis/principal-component-analysis-pca/>

i Read

The University of Washington's CSE 446 slides cover PCA^a and, with annotations, the SVD^b [5], Lectures 26 and 27.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/26-PCA.pdf>

^b<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/27-SVD-annotated.pdf>

6.5 In code

PCA is the SVD of the centered matrix, so the implementation is two lines and the interesting part is what happens when you get one of them wrong. Each of the three tabs runs it twice: once properly, and once on the uncentered $\mathbf{X}$, where we deliberately place the cloud a long way from the origin.

6.5.1 Python

```
import numpy as np

rng = np.random.default_rng(0)
n = 200
t = rng.normal(scale=3.0, size=n)
X = np.column_stack([t, 2 * t]) + rng.normal(scale=0.3, size=(n, 2))
X = X + np.array([50.0, 10.0])          # park the cloud away from the origin

Xc = X - X.mean(axis=0)                 # the one line that makes it PCA
U, S, Vt = np.linalg.svd(Xc, full_matrices=False)

print("explained:", (S ** 2 / (S ** 2).sum()).round(4))
```

```
explained: [0.9978 0.0022]
```

```
print("v1:", Vt[0].round(3))
```

```
v1: [-0.441 -0.898]
```

```
print("rank-1 error:",
      round(float(np.linalg.norm(Xc - S[0] * np.outer(U[:, 0], Vt[0]))), 3))
```

```
rank-1 error: 4.326
```

```
V_uncentered = np.linalg.svd(X, full_matrices=False)[2]
print("uncentered v1:", V_uncentered[0].round(3),
      "| mean direction:", (X.mean(0) / np.linalg.norm(X.mean(0))).round(3))
```

```
uncentered v1: [-0.979 -0.205] | mean direction: [0.98 0.198]
```

6.5.2 R

```
set.seed(0)
n <- 200
z <- rnorm(n, sd = 3)
X <- cbind(z, 2 * z) + matrix(rnorm(2 * n, sd = 0.3), ncol = 2)
```

```
X <- sweep(X, 2, c(50, 10), "+")          # park the cloud away from the origin

Xc <- scale(X, center = TRUE, scale = FALSE) # the one line that makes it PCA
s <- svd(Xc)
```

```
cat("explained:", round(s$d^2 / sum(s$d^2), 4), "\n")
```

```
explained: 0.9977 0.0023
```

```
cat("v1:", round(s$v[, 1], 3), "\n")
```

```
v1: -0.445 -0.895
```

```
cat("rank-1 error:",
    round(norm(Xc - s$d[1] * outer(s$u[, 1], s$v[, 1]), "F"), 3), "\n")
```

```
rank-1 error: 4.196
```

```
v_unchentered <- svd(X)$v[, 1]
mu <- colMeans(X)
cat("unchentered v1:", round(v_unchentered, 3),
    "| mean direction:", round(mu / sqrt(sum(mu^2)), 3), "\n")
```

```
unchentered v1: -0.979 -0.202 | mean direction: 0.981 0.195
```

6.5.3 Julia

```
using Random, LinearAlgebra, Statistics
```

```
Random.seed!(0);
n = 200;
z = 3 .* randn(n);
X = [z 2 .* z] .+ 0.3 .* randn(n, 2);
# park the cloud away from the origin
X = X .+ [50.0 10.0];

# the one line that makes it PCA
Xc = X .- mean(X, dims = 1);
U, S, V = svd(Xc);

println("explained: ", round.(S .^ 2 ./ sum(S .^ 2), digits = 4))
```

```
explained: [0.9982, 0.0018]
```

```
println("v1: ", round.(V[:, 1], digits = 3))
```

```
v1: [0.452, 0.892]
```

```
println("rank-1 error: ",
    round(norm(Xc - S[1] * U[:, 1] * V[:, 1]'), digits = 3))
```

```
rank-1 error: 4.033
```

```
V_unchentered = svd(X).V;
mu = vec(mean(X, dims = 1));
println("unchentered v1: ", round.(V_unchentered[:, 1], digits = 3),
    " | mean direction: ", round.(mu ./ norm(mu), digits = 3))
```

```
unchentered v1: [-0.977, -0.215] | mean direction: [0.978, 0.207]
```

```

# What the three tabs print, gathered for the prose, with the claims it
# makes in words checked here.
jl <- function(expr) JuliaCall::julia_eval(expr)
pca <- list(
  py = list(explained = py$S^2 / sum(py$S^2), v1 = py$Vt[1, ],
    unc_v1 = py$V_uncentered[1, ], X = py$X),
  r = list(explained = s$d^2 / sum(s$d^2), v1 = s$v[, 1],
    unc_v1 = v_uncentered, X = X),
  jl = list(explained = jl("S .^ 2 ./ sum(S .^ 2)"), v1 = jl("V[:, 1]"),
    unc_v1 = jl("V_uncentered[:, 1]"), X = jl("X"))
)
truth <- c(1, 2) / sqrt(5)
for (p in pca) {
  mu <- colMeans(p$X)
  mean_dir <- mu / sqrt(sum(mu^2))
  d_unc <- svd(p$X)$d
  stopifnot(
    max(abs(abs(p$v1) - truth)) < 0.01,
    max(abs(abs(p$unc_v1) - mean_dir)) < 0.01,
    d_unc[1]^2 / sum(d_unc^2) > 0.99
  )
}
explained_pct <- unique(vapply(pca, function(p) round(100 * p$explained[1], 1),
  numeric(1)))
explained_text <- if (length(explained_pct) == 1) {
  sprintf("all three put %s\\%%%", explained_pct)
} else {
  sprintf("each puts between %s\\%%$ and %s\\%%%",
    min(explained_pct), max(explained_pct))
}
# Which sign each library returns is an implementation detail, so the prose
# reports whatever the three returned on this render.
v1_signs <- vapply(pca, function(p) sign(p$v1[which.max(abs(p$v1))]),
  numeric(1))
sign_text <- if (length(unique(v1_signs)) > 1) {
  "The signs do not match."
} else {
  "The signs happen to match on this render, and nothing guarantees they will."
}
fmt_v <- function(v, digits = 3) {
  paste0("(", paste(formatC(v, format = "f", digits = digits, flag = "+"),
    collapse = ", "), ")")
}
py_mu <- colMeans(pca$py$X)

```

We built the data to vary along $(1, 2)$, whose unit vector is $(0.447, 0.894)$. All three recover it to within 0.01, and all three put 99.8% of the variance on that one component. The noise we added to each coordinate is isotropic rather than perpendicular to the line, so it lands partly along the line and partly across it; the second singular value measures the part that lands across.

Two things in the printed output matter more than the agreement.

The signs are arbitrary. The signs do not match. Python returns $\tilde{v}_1 = (-0.441, -0.898)$ and R returns $(-0.445, -0.895)$, each drawing its own sample; Julia returns $(+0.452, +0.892)$. All three are correct. If $\tilde{u}_i \sigma_i \tilde{v}_i^\top$ is a term of the decomposition then so is $(-\tilde{u}_i) \sigma_i (-\tilde{v}_i)^\top$, so the SVD fixes the *axis* and not the direction along it, and which one a library returns is a detail of its implementation. Never read a sign, or the sign of a score, as meaning anything; compare $|\tilde{v}_1|$, or fix the convention yourself by flipping so that the

largest-magnitude entry is positive.

Skipping the centering does not degrade the answer, it replaces it. The uncentered run returns $v_1 \approx (-0.98, -0.21)$ in Python, against a mean direction of $\approx (+0.98, +0.20)$: the same axis, and nothing to do with $(0.447, 0.894)$. R and Julia each land on the axis of their own mean direction, to within 0.01. The leading component has stopped describing how the data varies and started describing where it is. Nothing warns you — it still reports over 99% of variance explained, because the mean is genuinely the largest thing in an uncentered matrix. So Section 6.3 insists centering is not a detail to skip, and placing a cloud far from the origin once lets you see it happen.

6.6 PCA on real data

Exercise 6.1 (Find the main directions of variation in car specifications). The Auto data of James et al. [26] describe 392 cars. Download `auto.csv`^a to load it in Python or R. Take five columns: `mpg`, `displacement`, `horsepower`, `weight` and `acceleration`. These are measured in different units, so standardize each column to mean 0 and standard deviation 1. Then run PCA on the standardized matrix with the SVD. Before you run it, say how many components you expect to need to describe most of the variation, and which columns you expect to load together.

^a<https://morrison-lab.github.io/lds/chapters/data/auto.csv>

Solution

Solution 6.1.

6.6.1 Python

```
import numpy as np
import pandas as pd

cars_py = pd.read_csv("data/auto.csv").dropna()
cols = ["mpg", "displacement", "horsepower", "weight", "acceleration"]
Z_py = cars_py[cols].to_numpy(dtype=float)
Z_py = (Z_py - Z_py.mean(axis=0)) / Z_py.std(axis=0, ddof=1)

U_py, S_py, Vt_py = np.linalg.svd(Z_py, full_matrices=False)
```

6.6.2 R

```
cars <- read.csv("data/auto.csv")
cars <- tidy::drop_na(cars)
cols <- c("mpg", "displacement", "horsepower", "weight", "acceleration")
Zc <- cars |>
  dplyr::select(dplyr::all_of(cols)) |>
  as.matrix() |>
  scale()

pca_cars <- svd(Zc)
```

Exercise 6.2 (Find the main directions of variation in car specifications (part 2)). Use the setup of Exercise 6.1. Report the share of variance^a each component explains.

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

Solution

Solution 6.2.

6.6.3 Python

```
pca_cars_py = pd.DataFrame({
    "component": np.arange(1, 6),
    "share": S_py ** 2 / np.sum(S_py ** 2),
})
pca_cars_py
```

	component	share
0	1	0.785351
1	2	0.142401
2	3	0.045124
3	4	0.016575
4	5	0.010549

```
# A bootstrap resamples the cars with replacement and repeats the analysis,
# because no formula gives an interval for a share of variance or a loading.
# Each resample is standardized again, as the original data were.
def boot_pca_py(rng, n_boot=1000):
    shares, loads = [], []
    for _ in range(n_boot):
        zb = Z_py[rng.integers(0, len(Z_py), len(Z_py))]
        zb = (zb - zb.mean(axis=0)) / zb.std(axis=0, ddof=1)
        _, sb, vtb = np.linalg.svd(zb, full_matrices=False)
        shares.append(sb ** 2 / np.sum(sb ** 2))
        # The sign of a component is arbitrary, so point it the way the full-data one points.
        loads.append(vtb[0] * np.sign(vtb[0] @ Vt_py[0]))
    return np.array(shares), np.array(loads)

shares_py, loads_py = boot_pca_py(np.random.default_rng(1))
print(np.round(np.percentile(shares_py, [2.5, 97.5], axis=0), 3))

[[0.763 0.125 0.038 0.014 0.008]
 [0.806 0.161 0.052 0.02  0.013]]
```

6.6.4 R

```
tibble::tibble(component = 1:5, share = pca_cars$d^2 / sum(pca_cars$d^2))
```

```
# A tibble: 5 x 2
  component share
  <int>   <dbl>
1         1 0.785
2         2 0.142
3         3 0.0451
4         4 0.0166
5         5 0.0105
```

```
# A bootstrap resamples the cars with replacement and repeats the analysis,
# because no formula gives an interval for a share of variance or a loading.
# Each resample is standardized again, as the original data were.
set.seed(1)
boot_pca <- replicate(1000, {
  sb <- svd(scale(Zc[sample(nrow(Zc), replace = TRUE), ]))
  # A component's sign is arbitrary: point it the way the full-data one points.
  c(share = sb$d^2 / sum(sb$d^2),
    load = sb$v[, 1] * sign(sum(sb$v[, 1] * pca_cars$v[, 1])))
})

share_r <- pca_cars$d^2 / sum(pca_cars$d^2)
stopifnot(max(abs(share_r - py$pca_cars_py$share)) < 1e-9, share_r[1] > 0.5)
# 95% percentile intervals: the middle 95% of the bootstrap values.
ci_share <- t(apply(boot_pca[1:5, ], 1, quantile, c(0.025, 0.975)))
pct_ci <- function(i) {
  r <- round(100 * ci_share[i, ], 1)
  paste0("[", r[1], ", ", r[2], "]")
}
```

Both languages standardize the same way and use the same SVD^a, so they agree on every share. The first component explains 78.5% of the variance^b (95% bootstrap interval [76.5, 80.7]%) and the second 14.2% ([12.5, 16]%). Each interval comes from 1000 resamples of the 392 cars, drawn with replacement, standardized and decomposed again. It is the middle 95% of the resampled values. The two languages draw different resamples, so their intervals agree only approximately, though their point estimates^c agree exactly. Each interval describes how much the result depends on which cars are in the data.

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

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

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

Exercise 6.3 (Find the main directions of variation in car specifications (part 3)). Use the setup of Exercise 6.1.

Report the loadings of the first component.

Solution

Solution 6.3.

6.6.5 Python

```
pca_load_py = pd.Series(Vt_py[0], index=cols)
pca_load_py

mpg          0.444264
displacement -0.483233
horsepower   -0.484442
weight       -0.471221
acceleration  0.335235
dtype: float64

print(np.round(np.percentile(loads_py, [2.5, 97.5], axis=0), 2))

[[ 0.44 -0.49 -0.49 -0.48  0.3 ]
 [ 0.45 -0.48 -0.48 -0.46  0.36]]
```

6.6.6 R

```
flip <- -sign(pca_cars$v[, 1]) # sign convention: mpg loads negatively
load1 <- setNames(flip * pca_cars$v[, 1], cols)
# 95% percentile intervals: the middle 95% of the bootstrap values.
ci_load1 <- t(apply(flip * boot_pca[6:10, ], 1, quantile, c(0.025, 0.975)))
round(cbind(loading = load1, ci_load1), 2)
```

	loading	2.5%	97.5%
mpg	-0.44	-0.45	-0.44
displacement	0.48	0.48	0.49
horsepower	0.48	0.48	0.49
weight	0.47	0.46	0.48
acceleration	-0.34	-0.36	-0.31

```
stopifnot(max(abs(abs(pca_cars$v[, 1]) - abs(unlist(py$pca_load_py)))) < 1e-9)
```

With the sign chosen so that the mpg loading is negative, the first component has loadings mpg -0.44 [-0.45, -0.44], displacement 0.48 [0.48, 0.49], horsepower 0.48 [0.48, 0.49], weight 0.47 [0.46, 0.48], acceleration -0.34 [-0.36, -0.31], each with its 95% bootstrap interval. The signs of a component are arbitrary (see the sign discussion above), so only the contrast between the two groups of columns is meaningful.

Exercise 6.4 (Find the main directions of variation in car specifications (part 4)). Use the setup of Exercise 6.1.

Plot each car by its first two component scores, using a different symbol for each **origin** (1 is American, 2 is European and 3 is Japanese).

Solution

Solution 6.4. Figure 4 shows the cars by their first two scores.

```
library(ggplot2)
scores <- Zc %>% cbind(flip * pca_cars$v[, 1], pca_cars$v[, 2])
scores_df <- tibble::tibble(
  pc1 = scores[, 1],
  pc2 = scores[, 2],
  weight = cars$weight,
  origin = factor(cars$origin, levels = 1:3,
    labels = c("American", "European", "Japanese"))
)
ggplot(scores_df, aes(pc1, pc2, shape = origin, colour = weight)) +
  geom_point(size = 2) +
  scale_colour_viridis_c(labels = scales::label_comma()) +
  scale_shape_manual(values = c(American = 1, European = 2, Japanese = 3)) +
  labs(x = "component 1 score", y = "component 2 score", shape = NULL,
    colour = "weight (lb)") +
  theme(legend.position = "bottom")
```

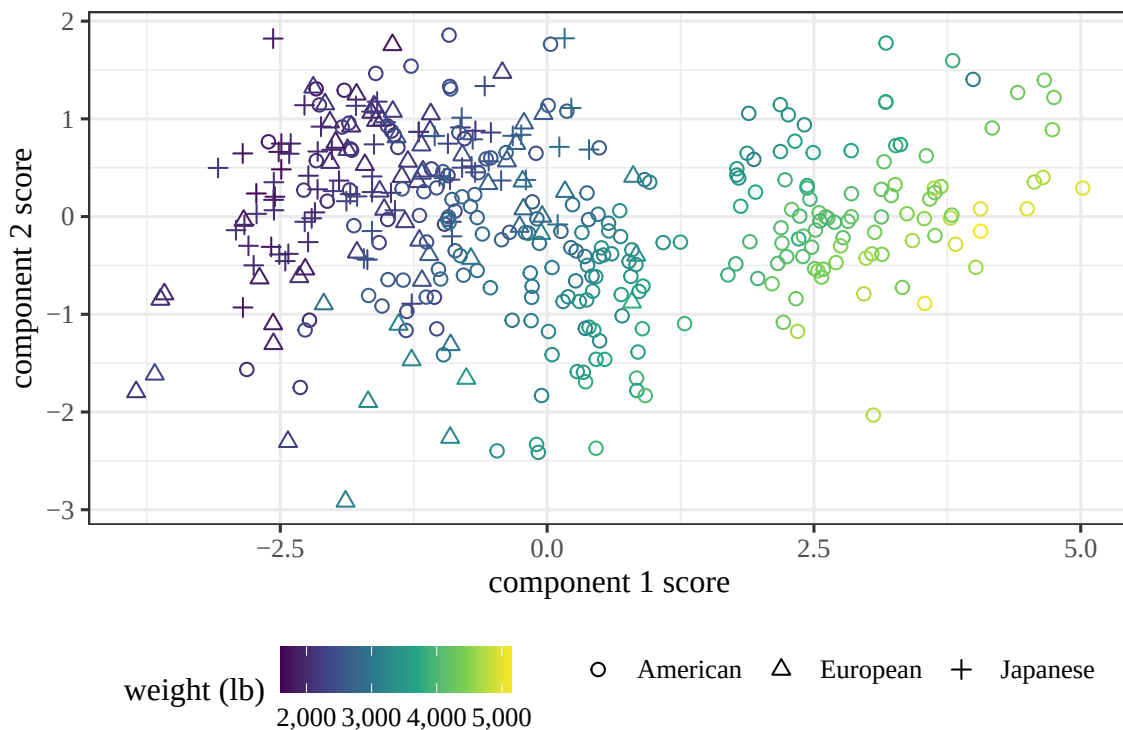


Figure 4: Cars from the Auto data by their first two principal component scores. Circles are American cars, triangles are European and plus signs are Japanese. The scores have no units, so each point is coloured by the car’s weight in pounds. Component 1 separates large cars on the right from small cars on the left, and American cars cluster on the large side.

Exercise 6.5 (Find the main directions of variation in car specifications (part 5)). Use the setup of Exercise 6.1.

How many components are needed to describe most of the variation? Compare with the number you expected^a before running PCA.

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

Solution

Solution 6.5. Two components describe 92.8% of the variation in five columns.

Exercise 6.6 (Find the main directions of variation in car specifications (part 6)). Use the setup of Exercise 6.1.

Which columns load together? Compare with the columns you expected^a to load together.

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

Solution

Solution 6.6. Displacement, horsepower and weight share one sign, and mpg and acceleration have the other: big, powerful, heavy cars have low fuel economy and take longer to reach speed, so the first component measures the size of the car.

Exercise 6.7 (Find the main directions of variation in car specifications (part 7)). Use the setup of Exercise 6.1.

Why must the columns be standardized before PCA here?

Solution

Solution 6.7. Standardizing matters because PCA maximizes variance^a, and a column measured in larger units has larger variance. Here `weight` is in the thousands and `acceleration` is near 15, so without standardizing, the first component would be almost entirely `weight` whatever the other columns did.

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

7 Density estimation

- Goal is to estimate $p(x)$, the probability density function (pdf) of the data
- This is unsupervised, probabilistic, and generative
- Why?
 - Understand the structure of the data / gain insight
 - Anomaly detection
 - Generate new data (see image generation models)
- We are going to focus on Gaussian and Gaussian Mixture Model distributions, as they are the commonly used, flexible, and easy to understand

7.1 The normal and multivariate normal distributions

- The probability behind this section lives in *Probability for Data Science*⁶, which we call “pds” below. This chapter uses that material rather than re-deriving it:
 - the normal distribution⁷ and its mean and variance⁸
 - the multivariate normal distribution⁹ (MVN for short), $N_D(\mu, \Sigma)$, whose mean is μ and variance matrix is Σ ¹⁰
 - the variance matrix¹¹, whose elements are the pairwise covariances¹², and which is always symmetric and positive semidefinite¹³
- For reference, the MVN density over $x \in \mathbb{R}^D$ is

$$p(x) = \frac{1}{\sqrt{(2\pi)^D |\Sigma|}} \exp \left\{ -\frac{1}{2} \underbrace{(x - \mu)^T \Sigma^{-1} (x - \mu)}_{\text{(Squared) Mahalanobis distance } d(x, \mu)} \right\}$$

with Σ positive definite¹⁴, so that Σ^{-1} and $|\Sigma|$ exist and $|\Sigma| > 0$: $|\Sigma|$ is the product of Σ ’s eigenvalues¹⁵, which are all positive¹⁶.

⁶<https://morrison-lab.github.io/pds/>

⁷<https://morrison-lab.github.io/pds/random-variables.html#def-normal>

⁸<https://morrison-lab.github.io/pds/random-variables.html#thm-normal-density>

⁹<https://morrison-lab.github.io/pds/distributions.html#def-mvn>

¹⁰<https://morrison-lab.github.io/pds/distributions.html#thm-mvn-moments>

¹¹<https://morrison-lab.github.io/pds/variance-covariance.html#def-cov-vec-x>

¹²<https://morrison-lab.github.io/pds/variance-covariance.html#thm-vcov-elements>

¹³<https://morrison-lab.github.io/pds/variance-covariance.html#thm-vcov-psd>

¹⁴<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-positive-definite>

¹⁵<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#thm-det-eigenvalues>

¹⁶<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#thm-definite-eigenvalues>

- Why the off-diagonal entries matter: two four-point datasets with the same per-coordinate variances can have variance matrices that differ only in the sign of the covariance¹⁷, tilting their point clouds in opposite directions.
- The density decreases with the Mahalanobis distance¹⁸ from μ (proof¹⁹):
 - special cases²⁰: $\Sigma = I$ gives Euclidean distance (spheres), $\Sigma = cI$ a rescaled one, and a diagonal Σ axis-aligned ellipsoids
 - in general, ellipsoids aligned with the eigenvectors of Σ ²¹, with widths set by the eigenvalues
 - a diagonal Σ means independent normal components²²; for an MVN, uncorrelated components are independent²³
- Σ^{-1} is sometimes called the **precision** or **concentration** matrix; it is positive definite too²⁴.

Exercise 7.1 (Sketch the one-standard-deviation ellipse of a Gaussian). Each of the following covariance matrices defines a 2-dimensional Gaussian with mean $\mu = (0, 0)^T$:

$$\Sigma_1 = \begin{pmatrix} 4 & 0 \\ 0 & 1 \end{pmatrix}, \quad \Sigma_2 = \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix}, \quad \Sigma_3 = \begin{pmatrix} 1 & -0.5 \\ -0.5 & 1 \end{pmatrix}, \quad \Sigma_4 = \begin{pmatrix} 1 & 0.9 \\ 0.9 & 1 \end{pmatrix}.$$

For each, sketch the set of points whose Mahalanobis distance from μ is 1 (the one-standard-deviation ellipse).

Solution

Solution 7.1. The set of points at Mahalanobis distance 1 is $\{x : (x - \mu)^T \Sigma^{-1} (x - \mu) = 1\}$. Its axes point along the eigenvectors of Σ .

- Σ_1 is diagonal, so the axes are the coordinate axes.
- Σ_2 has a positive covariance^a. The ellipse tilts along the line $x_2 = x_1$.
- Σ_3 has a negative covariance, so the long axis now lies along $(1, -1)^T$. A negative covariance tilts the ellipse the other way.
- Σ_4 tilts the same way as Σ_2 . It is a long thin ellipse: the strong covariance says the two coordinates almost move together.

The code below draws the four ellipses in Figure 5.

¹⁷<https://morrison-lab.github.io/pds/variance-covariance.html#exm-vcov-psd>

¹⁸<https://morrison-lab.github.io/pds/distributions.html#def-mahalanobis>

¹⁹<https://morrison-lab.github.io/pds/distributions.html#cor-mvn-mahalanobis>

²⁰<https://morrison-lab.github.io/pds/distributions.html#exm-mahalanobis-special>

²¹<https://morrison-lab.github.io/pds/distributions.html#thm-mahalanobis-eigen>

²²<https://morrison-lab.github.io/pds/distributions.html#thm-mvn-diagonal-indpt>

²³<https://morrison-lab.github.io/pds/distributions.html#cor-mvn-uncorrelated-indpt>

²⁴<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#thm-pd-inverse>

```

covs <- list(
  Sigma_1 = matrix(c(4, 0, 0, 1), 2),
  Sigma_2 = matrix(c(1, 0.5, 0.5, 1), 2),
  Sigma_3 = matrix(c(1, -0.5, -0.5, 1), 2),
  Sigma_4 = matrix(c(1, 0.9, 0.9, 1), 2)
)

# Points on the unit circle, mapped to the one-sd ellipse by the matrix square
# root of Sigma: Sigma = V diag(lambda) V^T, so the map is V diag(sqrt(lambda)).
theta <- seq(0, 2 * pi, length.out = 200)
circle <- rbind(cos(theta), sin(theta))
ellipse_points <- function(S) {
  e <- eigen(S)
  t(e$vectors %*% diag(sqrt(e$values)) %*% circle)
}

op <- par(mfrow = c(1, 4), mar = c(3, 3, 2, 0.5))
for (i in seq_along(covs)) {
  xy <- ellipse_points(covs[[i]])
  plot(xy, type = "l", asp = 1, xlim = c(-2.2, 2.2), ylim = c(-2.2, 2.2),
       xlab = "x1", ylab = "x2", main = names(covs)[i])
  abline(h = 0, v = 0, col = "grey80")
}
par(op)

```

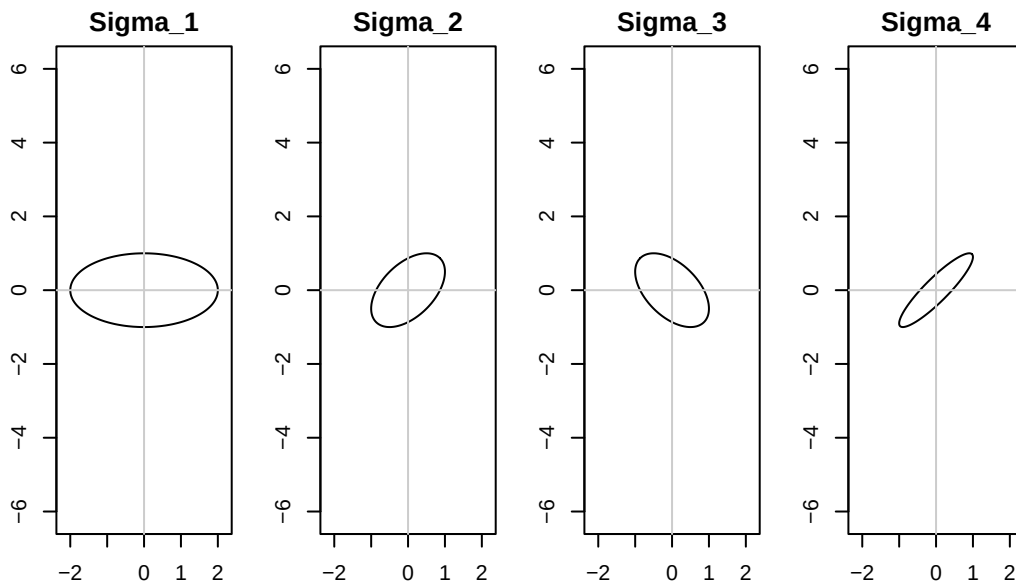


Figure 5: The one-standard-deviation ellipses of the four Gaussians.

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

Exercise 7.2 (Sketch the one-standard-deviation ellipse of a Gaussian (part 2)). Use the setup of Exercise 7.1.

For each ellipse, mark its two axes and say how long each is.

Solution

Solution 7.2. For each matrix of Solution 7.1, the semi-axis along each eigenvector is the square root of the matching eigenvalue.

- Σ_1 has eigenvalues 4 and 1, so the semi-axes are 2 along x_1 and 1 along x_2 .
- Σ_2 has eigenvalues 1.5 (eigenvector $(1, 1)^T$) and 0.5 (eigenvector $(1, -1)^T$), so the semi-axes are $\sqrt{1.5} \approx 1.22$ and $\sqrt{0.5} \approx 0.71$.
- Σ_3 has the same eigenvalues as Σ_2 , so the same semi-axes, with the long one along $(1, -1)^T$.
- Σ_4 has eigenvalues 1.9 and 0.1 along the same directions as Σ_2 . The semi-axes are $\sqrt{1.9} \approx 1.38$ and $\sqrt{0.1} \approx 0.32$.

The code below finds the eigenvalues.

7.1.1 Python

```
import numpy as np

covs_py = {
    "Sigma_1": np.array([[4.0, 0.0], [0.0, 1.0]]),
    "Sigma_2": np.array([[1.0, 0.5], [0.5, 1.0]]),
    "Sigma_3": np.array([[1.0, -0.5], [-0.5, 1.0]]),
    "Sigma_4": np.array([[1.0, 0.9], [0.9, 1.0]]),
}

# eigh() is for symmetric matrices: it returns the eigenvalues in increasing
# order and the eigenvectors as columns. The semi-axes are sqrt(eigenvalues).
for name, S in covs_py.items():
    vals, vecs = np.linalg.eigh(S)
    print(name, "semi-axes:", np.sqrt(vals).round(2))

Sigma_1 semi-axes: [1. 2.]
Sigma_2 semi-axes: [0.71 1.22]
Sigma_3 semi-axes: [0.71 1.22]
Sigma_4 semi-axes: [0.32 1.38]
```

7.1.2 R

```
# eigen() on a symmetric matrix returns the eigenvalues in decreasing order
# and the eigenvectors as columns. The semi-axes are sqrt(eigenvalues).
lapply(covs, function(S) round(sqrt(eigen(S)$values), 2))

$Sigma_1
[1] 2 1

$Sigma_2
[1] 1.22 0.71

$Sigma_3
[1] 1.22 0.71

$Sigma_4
[1] 1.38 0.32
```

Exercise 7.3 (Rank points by Mahalanobis distance). A Gaussian has mean $\mu = (0, 0)^T$ and covariance matrix $\Sigma = \begin{pmatrix} 1 & 0.8 \\ 0.8 & 1 \end{pmatrix}$. Consider the points $P = (1, 1)^T$, $Q = (1, -1)^T$ and $R = (2, 2)^T$. Order them from closest to farthest from μ by Euclidean distance^a.

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

Solution

Solution 7.3. The Euclidean distances are $\sqrt{2}$ for P , $\sqrt{2}$ for Q and $2\sqrt{2}$ for R , so P and Q tie for closest and R is farthest.

Exercise 7.4 (Rank points by Mahalanobis distance (part 2)). Use the setup of Exercise 7.3. Order them from closest to farthest from μ by Mahalanobis distance^a.

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

Solution

Solution 7.4. For the Mahalanobis distance^a, first invert Σ . Its determinant is $1 - 0.8^2 = 0.36$, so

$$\Sigma^{-1} = \frac{1}{0.36} \begin{pmatrix} 1 & -0.8 \\ -0.8 & 1 \end{pmatrix}.$$

The squared Mahalanobis distance of $x = (a, b)^T$ is then $\frac{1}{0.36}(a^2 - 1.6ab + b^2)$, so

$$\begin{aligned} d^2(P) &= \frac{1 - 1.6 + 1}{0.36} = \frac{0.4}{0.36} \approx 1.11, \\ d^2(Q) &= \frac{1 + 1.6 + 1}{0.36} = \frac{3.6}{0.36} = 10, \\ d^2(R) &= \frac{4 - 6.4 + 4}{0.36} = \frac{1.6}{0.36} \approx 4.44. \end{aligned}$$

From closest to farthest the order is P , then R , then Q .

7.1.3 Python

```
import numpy as np

sigma_py = np.array([[1.0, 0.8], [0.8, 1.0]])
pts_py = {"P": np.array([1.0, 1.0]), "Q": np.array([1.0, -1.0]),
          "R": np.array([2.0, 2.0])}

# Squared Mahalanobis distance from the mean 0: x' Sigma^{-1} x.
# solve() applies Sigma^{-1} without forming the inverse.
{k: round(float(x @ np.linalg.solve(sigma_py, x)), 2) for k, x in pts_py.items()}

{'P': 1.11, 'Q': 10.0, 'R': 4.44}
```

7.1.4 R

```
sigma_m <- matrix(c(1, 0.8, 0.8, 1), 2)
pts <- list(P = c(1, 1), Q = c(1, -1), R = c(2, 2))

# Squared Mahalanobis distance from the mean 0: x' Sigma^{-1} x.
# solve() applies Sigma^{-1} without forming the inverse.
sapply(pts, function(x) round(drop(t(x) %*% solve(sigma_m, x)), 2))

      P      Q      R
1.11 10.00  4.44
```

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

Exercise 7.5 (Rank points by Mahalanobis distance (part 3)). Use the setup of Exercise 7.3. Why do the two orders disagree?

Solution

Solution 7.5. The covariance^a is positive, so the Gaussian is stretched along the line $x_2 = x_1$. Points along that line are common, and their Mahalanobis distance^b is small even when their Euclidean distance^c is large, as for R . Q lies across the line, against the direction of greatest spread, so it is far in Mahalanobis distance although it is no farther than P in Euclidean terms.

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

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

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

7.2 Fitting an MVN distribution

- Given a dataset $\mathcal{X} = \{x_{(1)}, \dots, x_{(N)}\}$, how do we estimate μ and Σ ?
- We solve the following optimization problem (maximize likelihood):

$$\begin{aligned} \arg \max_{\mu, \Sigma} p(\mathcal{X}; \mu, \Sigma) &= \arg \max_{\mu, \Sigma} \prod_{i=1}^N p(x_{(i)}; \mu, \Sigma) \\ &= \arg \max_{\mu, \Sigma} \ln \prod_{i=1}^N p(x_{(i)}; \mu, \Sigma) \\ &= \arg \max_{\mu, \Sigma} \sum_{i=1}^N \ln p(x_{(i)}; \mu, \Sigma) \end{aligned}$$

To solve it:

1. plug in the MVN pdf;
 2. take the derivative;
 3. set it to zero;
 4. solve for μ and Σ .
- The solution is:

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_{(i)}, \quad \hat{\Sigma} = \frac{1}{N} \sum_{i=1}^N (x_{(i)} - \hat{\mu})(x_{(i)} - \hat{\mu})^T$$

- This is the **maximum likelihood estimate** (MLE) for μ and Σ
- Works well because many pdfs are **log-concave**, including normal distributions

Exercise 7.6 (Compute the maximum likelihood mean and covariance). Three points in the plane are $x_{(1)} = (1, 2)^T$, $x_{(2)} = (3, 0)^T$ and $x_{(3)} = (5, 4)^T$. Compute the maximum likelihood^a estimate^b $\hat{\mu}$ of the mean^c of a 2-dimensional Gaussian fit to them, by hand.

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

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

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

Solution

Solution 7.6. The mean is the average of the points:

$$\hat{\mu} = \frac{1}{3} \left[\begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 3 \\ 0 \end{pmatrix} + \begin{pmatrix} 5 \\ 4 \end{pmatrix} \right] = \begin{pmatrix} 3 \\ 2 \end{pmatrix}.$$

7.2.1 Python

```
import numpy as np

x_mle = np.array([[1.0, 2.0], [3.0, 0.0], [5.0, 4.0]])
mu_hat = x_mle.mean(axis=0)
print(mu_hat)

[3.  2.]
```

7.2.2 R

```
x_mle <- rbind(c(1, 2), c(3, 0), c(5, 4))
colMeans(x_mle)

[1] 3 2
```

Exercise 7.7 (Compute the maximum likelihood mean and covariance (part 2)). Use the setup of Exercise 7.6.

Compute the maximum likelihood^a estimate^b $\hat{\Sigma}$ of the covariance matrix^c of the same Gaussian, by hand.

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

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

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

Solution

Solution 7.7. The deviations^a $x_{(i)} - \hat{\mu}$ are $(-2, 0)^T$, $(0, -2)^T$ and $(2, 2)^T$. Averaging their outer products^b, with $N = 3$ in the denominator (not $N - 1$, because this is the maximum likelihood^c estimate^d):

$$\begin{aligned}\hat{\Sigma} &= \frac{1}{3} \left[\begin{pmatrix} 4 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 4 \end{pmatrix} + \begin{pmatrix} 4 & 4 \\ 4 & 4 \end{pmatrix} \right] \\ &= \frac{1}{3} \begin{pmatrix} 8 & 4 \\ 4 & 8 \end{pmatrix} \\ &= \begin{pmatrix} 8/3 & 4/3 \\ 4/3 & 8/3 \end{pmatrix}.\end{aligned}$$

7.2.3 Python

```
# bias=True divides by N, the maximum likelihood estimate;
# np.cov divides by N - 1 by default.
print(np.cov(x_mle, rowvar=False, bias=True))

[[2.66666667 1.33333333]
 [1.33333333 2.66666667]]
```

7.2.4 R

```
# cov() divides by N - 1, so rescale by (N - 1) / N to get the
# maximum likelihood estimate, which divides by N.
cov(x_mle) * (nrow(x_mle) - 1) / nrow(x_mle)

      [,1]      [,2]
[1,] 2.666667 1.333333
[2,] 1.333333 2.666667
```

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

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

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

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

8 Mixture models

- Gaussians are useful, but they are **unimodal**
- Often a population is made up of distinct sub-populations
- We can model the overall population as a mixture of subpopulations - this is a **mixture model**²⁵

$$p_{mix}(x) = \sum_{c=1}^M w_c p_c(x)$$

- $w_1, \dots, w_M$ are the mixture weights, non-negative, sum to one
- pds covers the probability of mixtures: the two-stage construction²⁶, the mixture mean²⁷, and which component an observation came from²⁸, with a worked two-component example²⁹

8.1 Gaussian mixture models

- When each model in the mixture is Gaussian, we call it a **Gaussian mixture model** (GMM for short)
- With large enough M , a GMM approximates any continuous density on $\mathbb{R}^d$ arbitrarily well: a narrow Gaussian behaves like a smoothed point mass, so a weighted sum of them is a smoothed histogram, and refining the histogram refines the fit. The claim is about the limit, not about exact representation at any finite M

The underlying generative process for a GMM is

1. Pick a sub-population (also called a component) according to w (also written $P(c)$)
2. Sample from the Gaussian with $\mu_{(c)}$ and covariance matrix $\Sigma_{(c)}$
3. Repeat

Each draw has the mixture density p_{mix} (why³⁰).

i Read

GeeksforGeeks explains Gaussian mixture models^a [27].

^a<https://www.geeksforgeeks.org/machine-learning/gaussian-mixture-model/>

i Read

The University of Washington's CSE 446 slides cover Gaussian mixture models^a [5], Lecture 25.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/25-GMM.pdf>

²⁵<https://morrison-lab.github.io/pds/distributions.html#def-mixture>

²⁶<https://morrison-lab.github.io/pds/distributions.html#thm-mixture-two-stage>

²⁷<https://morrison-lab.github.io/pds/distributions.html#thm-mixture-mean>

²⁸<https://morrison-lab.github.io/pds/distributions.html#thm-mixture-posterior>

²⁹<https://morrison-lab.github.io/pds/distributions.html#exm-mixture-gaussian>

³⁰<https://morrison-lab.github.io/pds/distributions.html#thm-mixture-two-stage>

8.2 Why fitting a GMM is hard

- GMM data log likelihood:

$$\log P(\mathcal{X}|\theta) = \sum_{i=1}^N \ln \left(\sum_{c=1}^M w_c p(x_{(i)}; \mu_{(c)}, \Sigma_{(c)}) \right)$$

This is hard to maximize directly. Taking the gradient is very difficult because the sum is inside the log.

- Some variables are missing: which component generated each datapoint ($z_{(i)} \in \mathbb{R}^M$)
 - $z_{(i)}$ is a one-hot indicator of the component ID

8.3 Fitting a GMM: the idea

- If we knew $z_{(i)}$, the problem would be easy to solve
 - Fit each component to its population

$$\begin{aligned} w_c &= \frac{1}{N} \sum_i z_{(i)c} \\ &= \frac{N_c}{N} \\ \mu_{(c)} &= \frac{\sum_i z_{(i)c} x_{(i)}}{N_c} \\ \Sigma_{(c)} &= \frac{\sum_i z_{(i)c} (x_{(i)} - \mu_{(c)})(x_{(i)} - \mu_{(c)})^T}{N_c} \end{aligned}$$

Here $N_c = \sum_i z_{(i)c}$ is the number of points assigned to component c — a whole number, because each $z_{(i)}$ is one-hot.

This simply takes the single MVN equations and applies them to each sub-population of the data, and then sets the w to the proportion of the data in that sub-population.

- Unfortunately, we do *not* know $z_{(i)}$.
 - But that doesn't mean all components are equally likely to have generated each point
 - Picture two well-separated components with a point sitting deep inside one of them: which component generated it is not in doubt
 - We could pick $z_{(i)}$ to be the component with the highest probability of generating $x_{(i)}$
 - Now picture the two components overlapping, with a point in the overlap: it belongs to no single component
 - In practice, we get better results by treating $z_{(i)}$ as an indicator variable and computing its expected value

$$\begin{aligned}
\underbrace{E[z_{(i)c}]}_{\gamma_{ic}} &= p(z_{(i)c} = 1 | x_{(i)}) \\
&= \frac{p(x_{(i)}, z_{(i)c} = 1)}{p(x_{(i)})} \\
&= \frac{p(x_{(i)}, z_{(i)c} = 1)}{\sum_{k=1}^M p(x_{(i)}, z_{(i)k} = 1)} \\
&= \frac{p(x_{(i)} | z_{(i)c} = 1) p(z_{(i)c} = 1)}{\sum_{k=1}^M p(x_{(i)} | z_{(i)k} = 1) p(z_{(i)k} = 1)} \\
&= \frac{p(x_{(i)}; \mu_{(c)}, \Sigma_{(c)}) w_c}{\sum_{k=1}^M p(x_{(i)}; \mu_{(k)}, \Sigma_{(k)}) w_k}
\end{aligned}$$

- $\gamma_{ic} \in [0, 1]$ is the *responsibility* of component c for point i (the same computation in pds³¹)
 - * Low when $x_{(i)}$ is far from $\mu_{(c)}$ in Mahalanobis distance and/or w_c is low
 - * High when $x_{(i)}$ is close to $\mu_{(c)}$ in Mahalanobis distance and/or w_c is high
- Write $\gamma_c = \sum_i \gamma_{ic}$ for the *effective* number of points belonging to component c . It plays the role N_c plays when $z_{(i)}$ is known, but it is not a count: the responsibilities are fractional, so γ_c is generally not a whole number
- With γ_{ic} defined, we can revisit the rules to compute w , μ and Σ

$$\begin{aligned}
w_c &= \frac{1}{N} \sum_i \gamma_{ic} \\
&= \frac{\gamma_c}{N} \\
\mu_{(c)} &= \frac{\sum_i \gamma_{ic} x_{(i)}}{\gamma_c} \\
\Sigma_{(c)} &= \frac{\sum_i \gamma_{ic} (x_{(i)} - \mu_{(c)})(x_{(i)} - \mu_{(c)})^T}{\gamma_c}
\end{aligned}$$

Exercise 8.1 (Compute a component’s mean and covariance from responsibilities). In a Gaussian mixture, component c has responsibilities $\gamma_{1c} = 1$, $\gamma_{2c} = 0.5$ and $\gamma_{3c} = 0.5$ for the three points of Exercise 7.6. Compute the effective number of points γ_c .

Solution

Solution 8.1. The effective number of points is the sum of the responsibilities, $\gamma_c = 1 + 0.5 + 0.5 = 2$. It is not the number of points, which is 3.

8.3.1 Python

```
import numpy as np

x_w = np.array([[1.0, 2.0], [3.0, 0.0], [5.0, 4.0]])
r = np.array([1.0, 0.5, 0.5])          # responsibilities of one component

r_c = r.sum()                          # effective number of points
print(r_c)

2.0
```

³¹<https://morrison-lab.github.io/pds/distributions.html#thm-mixture-posterior>

8.3.2 R

```
x_w <- rbind(c(1, 2), c(3, 0), c(5, 4))
r <- c(1, 0.5, 0.5) # responsibilities of one component

r_c <- sum(r) # effective number of points
r_c

[1] 2
```

Exercise 8.2 (Compute a component's mean and covariance from responsibilities (part 2)). Use the setup of Exercise 8.1.

Compute the updated mean^a $\mu_{(c)}$.

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

Solution

Solution 8.2. The weighted mean^a divides by γ_c :

$$\mu_{(c)} = \frac{1(1, 2)^T + 0.5(3, 0)^T + 0.5(5, 4)^T}{2} = \frac{(5, 4)^T}{2} = \begin{pmatrix} 2.5 \\ 2 \end{pmatrix}.$$

The point with responsibility 1 counts twice as much as each of the others, so the mean moves toward it, from $(3, 2)^T$ to $(2.5, 2)^T$.

8.3.3 Python

```
mu_c = (r[:, None] * x_w).sum(axis=0) / r_c
print(mu_c)

[2.5 2. ]
```

8.3.4 R

```
mu_c <- colSums(r * x_w) / r_c
mu_c

[1] 2.5 2.0
```

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

Exercise 8.3 (Compute a component's mean and covariance from responsibilities (part 3)). Use the setup of Exercise 8.1.

Compute the updated covariance matrix^a $\Sigma_{(c)}$.

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

Solution

Solution 8.3. The deviations^a from this mean^b are $(-1.5, 0)^T$, $(0.5, -2)^T$ and $(2.5, 2)^T$. Weighting each outer product^c by its responsibility, and dividing by γ_c :

$$\begin{aligned}
\Sigma_{(c)} &= \frac{1}{2} \left[1 \begin{pmatrix} 2.25 & 0 \\ 0 & 0 \end{pmatrix} + 0.5 \begin{pmatrix} 0.25 & -1 \\ -1 & 4 \end{pmatrix} + 0.5 \begin{pmatrix} 6.25 & 5 \\ 5 & 4 \end{pmatrix} \right] \\
&= \frac{1}{2} \begin{pmatrix} 5.5 & 2 \\ 2 & 4 \end{pmatrix} \\
&= \begin{pmatrix} 2.75 & 1 \\ 1 & 2 \end{pmatrix}.
\end{aligned}$$

With every responsibility equal to 1, the formulas reduce to the maximum likelihood^d estimates^e of Solution 7.2.4.

8.3.5 Python

```
dev = x_w - mu_c
# Sum of r_i * (x_i - mu)(x_i - mu)', then divide by r_c.
sigma_c = (r[:, None, None] * dev[:, :, None] * dev[:, None, :]).sum(axis=0) / r_c
print(sigma_c)

[[2.75 1.  ]
 [1.  2.  ]]
```

8.3.6 R

```
dev <- sweep(x_w, 2, mu_c)
# Sum of r_i * (x_i - mu)(x_i - mu)', then divide by r_c.
sigma_c <- crossprod(r * dev, dev) / r_c
sigma_c

      [,1] [,2]
[1,] 2.75   1
[2,] 1.00   2
```

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

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

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

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

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

8.4 Fitting a GMM: the derivation

- Let's introduce z_i variables

$$L(\mathcal{X} | \{\mu_{(c)}\}_{c=1\dots M}, \{\Sigma_{(c)}\}_{c=1\dots M}) = \prod_{i=1}^N \prod_{c=1}^M (w_c p(x_{(i)} | \mu_{(c)}, \Sigma_{(c)}))^{z_{(i)c}}$$

or the **log-likelihood**

$$\ell(\mathcal{X} | \{\mu_{(c)}\}_{c=1\dots M}, \{\Sigma_{(c)}\}_{c=1\dots M}) = \sum_{i=1}^N \sum_{c=1}^M z_{(i)c} \ln(w_c p(x_{(i)} | \mu_{(c)}, \Sigma_{(c)}))$$

- Now, of course, we don't know z . So we'll instead maximize the *expected value*

$$\begin{aligned}
E[\ell(\mathcal{X} | \{\mu_{(c)}\}_{c=1\dots M}, \{\Sigma_{(c)}\}_{c=1\dots M})] &= E \left[\sum_{i=1}^N \sum_{c=1}^M z_{(i)c} \ln(w_c p(x_{(i)} | \mu_{(c)}, \Sigma_{(c)})) \right] \\
&= \sum_{i=1}^N \sum_{c=1}^M \underbrace{E[z_{(i)c}]}_{p(z_{(i)c}=1 | x_{(i)})} \ln(w_c p(x_{(i)} | \mu_{(c)}, \Sigma_{(c)}))
\end{aligned}$$

- We call $p(z_{(i)c} = 1|x_{(i)}) = \gamma_{ic}$ the responsibility of component c for point i

$$\begin{aligned}\gamma_{ic} &= p(z_{(i)c} = 1|x_{(i)}) \\ &= \frac{p(x_{(i)}|z_{(i)c} = 1)p(z_{(i)c} = 1)}{p(x_{(i)})} \\ &= \frac{p(x_{(i)}; \mu_{(c)}, \Sigma_{(c)})w_c}{\sum_k p(x_{(i)}; \mu_{(k)}, \Sigma_{(k)})w_k}\end{aligned}$$

- Given all γ_{ic} , we can now compute new w , $\mu_{(c)}$ and $\Sigma_{(c)}$:

$$\begin{aligned}w_c &= \frac{1}{N} \sum_i \gamma_{ic} \\ &= \frac{\gamma_c}{N} \\ \mu_{(c)} &= \frac{\sum_i \gamma_{ic} x_{(i)}}{\gamma_c} \\ \Sigma_{(c)} &= \frac{\sum_i \gamma_{ic} (x_{(i)} - \mu_{(c)})(x_{(i)} - \mu_{(c)})^T}{\gamma_c}\end{aligned}$$

- The Expectation Maximization algorithm is the training algorithm for GMMs [28]:
 1. Initialize w and all means and covariances
 2. Compute all γ_{ic} (expectation)
 3. Update w and all μ and Σ (maximization)
 4. Repeat steps 2-3 until convergence

One way to initialize is to run k-means and use the cluster assignments as the initial z variables

- See demo: <https://lukapopijac.github.io/gaussian-mixture-model/>
- Note: you can cluster with GMMs, and people often do:
 - Fit a GMM to the data
 - Assign each point to the component with the highest responsibility

Read

GeeksforGeeks explains the expectation-maximization algorithm^a [29].

^a<https://www.geeksforgeeks.org/machine-learning/ml-expectation-maximization-algorithm/>

9 Hard and soft clustering

k-means, spectral clustering and agglomerative clustering all return the same kind of object: a map from points to cluster labels, with every point getting exactly one. That is **hard clustering**, and its limitation shows wherever two clusters overlap. A point in the overlap belongs no more to one than to the other, and a hard assignment records only the winner, discarding the fact that it barely won.

Soft clustering gives each point a distribution over clusters instead, so the overlap is reported rather than resolved. The responsibilities γ_{ic} are exactly that distribution: a point deep inside one component has γ_{ic} near 1 there, while a point in the overlap splits its responsibility between two components. The recipe for clustering with a GMM — fit the mixture, then keep the largest responsibility — is the conversion back, so the soft clustering loses nothing in expressiveness and carries strictly more information.

There is a second distinction, related but not the same. A **probabilistic** model supplies a density $p(x)$ over the whole input space, so you can ask how plausible a new point is under the model at all, rather than only which cluster it falls nearest. A GMM supplies one. k-means does not: it returns centroids and assignments, and there is no $p(x)$ to evaluate. An outlier is therefore invisible to k-means, which assigns it like any other point, and merely improbable under the mixture, which can say so.

10 Further reading

- Hastie et al. [30], Chapter 14
- James et al. [26], Chapter 12
- Zhou [31], Unsupervised Learning³² — slides on principal components, UMAP and clustering, with code in R
- Stanford University [32], Stanford EE 278 (taught by Tsachy Weissman) — handwritten notes on covariance matrices, the spectral decomposition³³, the Karhunen–Loève expansion and optimal linear approximation, and principal component analysis
- Lange [33], Chapter 9 — the singular value decomposition³⁴, its properties, and its applications, with an emphasis on how to compute it

i Watch

The Data Science Learning Community’s book clubs discuss *An Introduction to Statistical Learning*, Chapter 12, in recorded meetings:

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

^a<https://www.youtube.com/watch?v=jE8vyrvwyBE&list=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>

^b<https://www.youtube.com/watch?v=crGNCAO-f7k&list=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>

^c<https://www.youtube.com/watch?v=nNf9Rj-sLec&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>

References

- [1] U. von Luxburg, “Statistical machine learning.” University of Tübingen, lecture videos on YouTube, 2020. Available: <https://www.youtube.com/playlist?list=PL05umP7R6ij2XCvrRzLokX6EoHWaGA2cC>
- [2] GeeksforGeeks, “Clustering in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/clustering-in-machine-learning/>
- [3] T. Broderick, “MIT 6.036, lecture 13: Clustering (fall 2020).” Lecture video on YouTube, uploaded 2021, 2020. Available: <https://www.youtube.com/watch?v=BaZWcSq3IuI>
- [4] GeeksforGeeks, “Elbow method for optimal value of k in KMeans.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/elbow-method-for-optimal-value-of-k-in-kmeans/>
- [5] N. Jaques, “CSE 446/546: Machine learning, lecture slides.” University of Washington, 2026. Accessed: Oct. 12, 2026. [Online]. Available: <https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/>
- [6] T. Hastie, R. Tibshirani, and J. Taylor, “Statistical learning with python.” Stanford Online, lecture videos on YouTube, 2023. Available: <https://www.youtube.com/playlist?list=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ>
- [7] A. Ng *et al.*, “CS229: Machine learning, autumn 2018.” Stanford Online, lecture videos on YouTube, 2018. Available: <https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGzqpfVfbU>

³²<https://ucla-biostat-212a.github.io/2026winter/slides/10-unsupervised/unsupervised.html>

³³<https://morrison-lab.github.io/mds/linear-algebra-diagonalization.html>

³⁴<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-svd>

- [8] J. Starmer, “K-means clustering.” StatQuest, video on YouTube, 2018. Available: <https://www.youtube.com/watch?v=4b5d3muPQmA>
- [9] GeeksforGeeks, “K means clustering – introduction.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/>
- [10] Wikipedia contributors, “Spectral clustering — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Spectral_clustering
- [11] GeeksforGeeks, “Spectral clustering.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/spectral-clustering/>
- [12] J. H. Ward, “Hierarchical grouping to optimize an objective function,” *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, 1963, doi: 10.1080/01621459.1963.10500845³⁵.
- [13] J. Starmer, “Hierarchical clustering.” StatQuest, video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=7xHsRkOdVwo>
- [14] GeeksforGeeks, “Hierarchical clustering in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/hierarchical-clustering/>
- [15] GeeksforGeeks, “Types of linkages in hierarchical clustering.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/ml-types-of-linkages-in-clustering/>
- [16] Wikipedia contributors, “Dimensionality reduction — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Dimensionality_reduction
- [17] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, “When is ‘nearest neighbor’ meaningful?” in *Database theory — ICDT’99*, in Lecture notes in computer science, vol. 1540. Springer, 1999, pp. 217–235. doi: 10.1007/3-540-49257-7_15³⁶.
- [18] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, “On the surprising behavior of distance metrics in high dimensional spaces,” in *Database theory — ICDT 2001*, in Lecture notes in computer science, vol. 1973. Springer, 2001, pp. 420–434. doi: 10.1007/3-540-44503-X_27³⁷.
- [19] GeeksforGeeks, “Curse of dimensionality in machine learning.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/curse-of-dimensionality-in-machine-learning/>
- [20] GeeksforGeeks, “Singular value decomposition (SVD).” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/data-science/singular-value-decomposition-svd/>
- [21] Wikipedia contributors, “Principal component analysis — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Principal_component_analysis
- [22] K. Pearson, “On lines and planes of closest fit to systems of points in space,” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901, doi: 10.1080/14786440109462720³⁸.
- [23] H. Hotelling, “Analysis of a complex of statistical variables into principal components,” *Journal of Educational Psychology*, vol. 24, no. 6, pp. 417–441, 1933, doi: 10.1037/h0071325³⁹.
- [24] J. Starmer, “Principal component analysis (PCA), step-by-step.” StatQuest, video on YouTube, 2018. Available: <https://www.youtube.com/watch?v=FgakZw6K1QQ>
- [25] GeeksforGeeks, “Principal component analysis (PCA).” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/data-analysis/principal-component-analysis-pca/>
- [26] 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/>
- [27] GeeksforGeeks, “Gaussian mixture model.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/gaussian-mixture-model/>
- [28] Wikipedia contributors, “Expectation–maximization algorithm — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm
- [29] GeeksforGeeks, “Expectation-maximization algorithm - ML.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/ml-expectation-maximization-algorithm/>
- [30] 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/>

³⁵<https://doi.org/10.1080/01621459.1963.10500845>

³⁶https://doi.org/10.1007/3-540-49257-7_15

³⁷https://doi.org/10.1007/3-540-44503-X_27

³⁸<https://doi.org/10.1080/14786440109462720>

³⁹<https://doi.org/10.1037/h0071325>

- [31] J. Zhou, “Biostat 212A: Statistical machine learning a, winter 2026.” UCLA, course webpage with lecture slides, 2026. Available: <https://ucla-biostat-212a.github.io/2026winter/>
- [32] Stanford University, “EE 278, probability and statistical inference: Fall 2024 lecture notes.” Handwritten lecture notes, Lectures 1–17, 2024. Available: <http://web.stanford.edu/class/ee278/lectures.html>
- [33] K. Lange, *Numerical analysis for statisticians*, 2nd ed. in Statistics and computing. Springer, 2010. doi: 10.1007/978-1-4419-5945-4⁴⁰.
- [34] Data Science Learning Community, “Introduction to statistical learning using R (islr06).” Book club meeting recordings on YouTube, 2023. Available: <https://www.youtube.com/playlist?list=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>
- [35] Data Science Learning Community, “Introduction to statistical learning using Python (islp03).” Book club meeting recordings on YouTube, 2025. Available: <https://www.youtube.com/playlist?list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp>

⁴⁰<https://doi.org/10.1007/978-1-4419-5945-4>