
NumPy and Numerical Computation for PCA

David I. Inouye

Dynamic Content · Last Updated: September 17, 2026

| The PCA equations give us a computation to implement

Let $\widetilde{X} \in \mathbb{R}^{n \times d}$ contain the original observations as rows.

Operation	Mathematics	Shape
Center	$X = \widetilde{X} - \mathbf{1}\mu^T$	$n \times d$
Choose directions	$X = U\Sigma V^T, W = V_k$	$d \times k$
Encode	$Z = XW$	$n \times k$
Reconstruct	$\widehat{\widetilde{X}} = ZW^T + \mathbf{1}\mu^T$	$n \times d$

Route: represent the objects, implement the operations, check the numerical result.

| Lists and arrays give different meanings to *

```
1 import numpy as np
2
3 values = [2, 5, 8]
4 x = np.array(values, dtype=np.float64)
5
6 print("type(values):", type(values))
7 print("type(x):", type(x))
8 print(f"values * 2: {values * 2!r}")
9 print(f"x * 2: {x * 2!r}")
10 print("x.shape:", x.shape)
11 print("x.dtype:", x.dtype)
```

```
type(values): <class 'list'>
type(x): <class 'numpy.ndarray'>
values * 2: [2, 5, 8, 2, 5, 8]
x * 2: array([ 4., 10., 16.])
x.shape: (3,)
x.dtype: float64
```

- A list is a Python sequence. Multiplication by an integer repeats it.
- An array stores values with a common data type. Here multiplication scales each entry.
- `shape` describes the axes. `dtype` describes how entries are stored.

| A one-dimensional array has **no row or column axis to swap**

Shapes: $(d,)$ = one observation; (n, d) = observations $\times$ features; (b, n, d) adds a batch axis.

```
1 x = np.array([2., 5., 8.])
2 print("x.shape:", x.shape)
3 print("x.T.shape:", x.T.shape)
4 print("x[:, None].shape:", x[:, None].shape)
5 print("x[None, :].shape:", x[None, :].shape)
```

```
x.shape: (3,)
x.T.shape: (3,)
x[:, None].shape: (3, 1)
x[None, :].shape: (1, 3)
```

- `None` (equivalently `np.newaxis`) **inserts a length-one axis** at that position.
- `x[:, None]` makes a column; `x[None, :]` makes a row.
- For 2D arrays, `.T` swaps the row and column axes.

Indexing: `A[i, :]` selects observation i ; `A[:, j]` selects feature j .

| * multiplies entries; @ implements matrix multiplication

```
1 P = np.array([[1., 2.],
2               [3., 4.]])
3 Q = np.array([[2., 0.],
4               [1., 2.]])
5 print("P * Q (entrywise):\n", P * Q)
6 print("P @ Q (matrix product):\n", P @ Q)
```

P * Q (entrywise):

[[2. 0.]

[3. 8.]]

P @ Q (matrix product):

[[4. 4.]

[10. 8.]]

For PCA, $Z = X @ W$ maps $(n, d) @ (d, k)$ to (n, k) .

The upper-left entries are $1 \cdot 2 = 2$ and $1 \cdot 2 + 2 \cdot 1 = 4$. Both results have shape $(2, 2)$. Shape alone cannot tell us which operation we intended.

| Centering averages across observations for each feature

For each feature, $\mu_j = \frac{1}{n} \sum_{i=1}^n \widetilde{X}_{ij}$.

```
1 A = np.array([[2., 10.],
2               [4., 14.], [6., 18.]])
3 mu = A.mean(axis=0)
4 X = A - mu
5 print("mu (feature means):", mu)
6 print("X (centered data):\n", X)
```

```
mu (feature means): [ 4. 14.]
X (centered data):
[[-2. -4.]
 [ 0.  0.]
 [ 2.  4.]]
```

Axis rule: count positions in the shape tuple from zero. Aggregate over the chosen dimension and remove it by default. The resulting shapes are:

```
1 T = np.zeros((2, 3, 4))
2 print("T.shape:", T.shape)
3 print("axis=0:", T.mean(axis=0).shape)
4 print("axis=1:", T.mean(axis=1).shape)
5 print("axis=2:", T.mean(axis=2).shape)
```

```
T.shape: (2, 3, 4)
axis=0: (3, 4)
axis=1: (2, 4)
axis=2: (2, 3)
```

Why does $X = A - \mu$ work? Ordinary matrix subtraction requires equal shapes. A matrix minus a vector is undefined. NumPy supplies **broadcasting**.

| Broadcasting compares dimensions **from right to left**

At each position, **either** of two rules must hold:

1. The two dimension sizes are **equal**: keep that size.
2. One size is **1**: reuse its value along the other dimension's size.

Missing dimensions count as 1s (marked * below). If neither rule holds at any position, broadcasting fails.

Vector (4,)

```
(3, 4)
(*, 4)
-----
(3, 4)
```

Both inputs expand

```
(3, 1)
(1, 4)
-----
(3, 4)
```

Vector (3,): incompatible

```
(3, 4)
(*, 3)
-----
Error
```

Matrix (3, 1)

```
(2, 3, 4)
(*, 3, 1)
-----
(2, 3, 4)
```

Scalar ()

```
(2, 3, 4)
(*, *, *)
-----
(2, 3, 4)
```

Rightmost sizes **4 and 3** are neither equal nor 1.

| Mathematics makes the repeated mean **an explicit matrix**

$$\underbrace{X}_{n \times d} = \underbrace{\widetilde{X}}_{n \times d} - \underbrace{\mathbf{1}_n \boldsymbol{\mu}^T}_{(n \times 1)(1 \times d) = n \times d}.$$

In NumPy, `A - mu` produces the same entries: $X_{ij} = \widetilde{X}_{ij} - \mu_j$.

```
1 ones = np.ones((A.shape[0], 1))
2 mu_matrix = ones @ mu[None, :]
3 print("mu_matrix:\n", mu_matrix)
4 print("A - mu:\n", A - mu)
5 print("Same result:",
6       np.allclose(A - mu_matrix, A - mu))
```

```
mu_matrix:
[[ 4. 14.]
 [ 4. 14.]
 [ 4. 14.]]
A - mu:
[[-2. -4.]
 [ 0.  0.]
 [ 2.  4.]]
Same result: True
```

`A: (3, 2); mu: (2, )`. **Broadcasting avoids copying the repeated means**, saving allocation and memory traffic. The output still needs storage.

| `keepdims=True` preserves reduced axes **with length one**

For `T.shape == (2, 3, 4)`, reducing `axis=1` gives `(2, 4)` by default. With `keepdims=True`, it gives `(2, 1, 4)`: the reduced position stays explicit.

```
1 reduced = T.mean(axis=1)
2 kept = T.mean(axis=1, keepdims=True)
3 restored = reduced[:, None, :]
4 print("T.shape:", T.shape)
5 print("reduced.shape:", reduced.shape)
6 print("kept.shape:", kept.shape)
7 print("restored.shape:", restored.shape)
8 print("(T - kept).shape:", (T - kept).shape)
```

```
T.shape: (2, 3, 4)
reduced.shape: (2, 4)
kept.shape: (2, 1, 4)
restored.shape: (2, 1, 4)
(T - kept).shape: (2, 3, 4)
```

- `keepdims` preserves a reduced axis; `None` inserts a new axis where specified.
- `(2, 1, 4)` broadcasts back against `(2, 3, 4)`.
- `(2, 4)` would align as `(1, 2, 4)` and **fail** at the middle position.

| Division broadcasts **one standard deviation per feature**

For positive feature standard deviations s_j , $H_{ij} = (\widetilde{X}_{ij} - \mu_j) / s_j$. In matrix notation, $H = X \text{diag}(1/s_1, \dots, 1/s_d)$.

```
1 std = A.std(axis=0)
2 H = (A - mu) / std
3 H_mul = (A - mu) * (1 / std)
4 print("std:", std.round(3))
5 print("H.round(3):\n", H.round(3))
6 print("H.std(axis=0):", H.std(axis=0))
7 print("Division equals multiplication:",
8       np.allclose(H, H_mul))
```

```
std: [1.633 3.266]
```

```
H.round(3):
```

```
[[ -1.225 -1.225]
```

```
 [ 0.      0.   ]
```

```
 [ 1.225  1.225]]
```

```
H.std(axis=0): [1. 1.]
```

```
Division equals multiplication: True
```

$(3, 2) / (2,)$ reuses each feature's scale across observations. The same shape rules apply to elementwise $+$, $-$, $*$, and $/$. **Scaling changes the relative weight of features in PCA's reconstruction error.**

Code can run successfully while centering the wrong objects

```
1 A = np.array([[2., 10.],  
2               [4., 14.], [6., 18.]])  
3 B = A - A.mean(axis=1, keepdims=True)  
4 print("B:\n", B)  
5 print("B.mean(axis=1):", B.mean(axis=1))  
6 print("B.mean(axis=0):", B.mean(axis=0))
```

```
B:  
[[-4.  4.]  
 [-5.  5.]  
 [-6.  6.]  
 B.mean(axis=1): [0.  0.  0.]  
 B.mean(axis=0): [-5.  5.]
```

What does this code subtract? Which averages become zero? Would this prepare the data for the PCA problem we defined? Repair the expression and explain why.

| NumPy returns the right singular vectors as rows of `Vt`

```
1 U, s, Vt = np.linalg.svd(  
2     X, full_matrices=False)  
3 W = Vt[:k, :].T
```

`full_matrices=False` omits extra directions multiplied by zero rows or columns of rectangular Σ . This **reduced SVD** keeps $m = \min(n, d)$ singular values; selecting the first k is a separate truncation.

Returned array	Shape	Mathematical role
<code>U</code>	<code>(n, m)</code>	Left singular vectors
<code>s</code>	<code>(m,)</code>	Singular values, largest first
<code>Vt</code>	<code>(m, d)</code>	Rows are right singular vectors transposed
<code>W</code>	<code>(d, k)</code>	First k right singular vectors as columns

| Encoding and reconstruction follow the same equations as before

`Xraw` stores $\widetilde{X}$, the original observations.

```
1 Xraw = np.array([[2., 10.], [4., 15.],  
2                 [6., 17.], [8., 22.]])  
3 mu = Xraw.mean(axis=0)  
4 X = Xraw - mu  
5 U, s, Vt = np.linalg.svd(  
6     X, full_matrices=False)  
7 k = 1  
8 W = Vt[:k, :].T  
9 Z = X @ W  
10 Xraw_hat = Z @ W.T + mu  
11 print("Z.round(3):\n", Z.round(3))
```

```
Z.round(3):  
[[-6.708]  
 [-1.347]  
 [ 1.347]  
 [ 6.708]]
```

For $X \in \mathbb{R}^{n \times d}$ and $W \in \mathbb{R}^{d \times k}$, $\widehat{X} = (XW)W^T = X(WW^T)$.

Centered reconstruction	Arithmetic cost	Intermediate
<code>Xhat = (X @ W) @ W.T</code>	$O(ndk)$	$n \times k$
<code>Xhat = X @ (W @ W.T)</code>	$O(d^2k + nd^2)$	$d \times d$

| The computed arrays show PCA's shapes and reconstruction

```
1 print("X.shape:", X.shape)
2 print("U.shape:", U.shape)
3 print("s.shape:", s.shape)
4 print("Vt.shape:", Vt.shape)
5 print("W.shape:", W.shape)
6 print("Z.shape:", Z.shape)
7 print("Xraw_hat.shape:", Xraw_hat.shape)
8 print("s.round(4):", s.round(4))
9 print("Xraw_hat.round(3):\n",
10       Xraw_hat.round(3))
11 error = np.sum((Xraw - Xraw_hat)**2)
12 print(f"error (squared): {error:.6f}")
```

```
X.shape: (4, 2)
U.shape: (4, 2)
s.shape: (2,)
Vt.shape: (2, 2)
W.shape: (2, 1)
Z.shape: (4, 1)
Xraw_hat.shape: (4, 2)
s.round(4): [9.6755 0.6201]
Xraw_hat.round(3):
[[ 1.923 10.04 ]
 [ 4.382 14.803]
 [ 5.618 17.197]
 [ 8.077 21.96 ]]
error (squared): 0.384552
```

We retained one direction, but `full_matrices=False` still computed **both** singular values.

| New observations use **the fitted mean and directions**

```
1 # Apply the representation fitted on Xraw
2 Xraw_new = np.array([[5., 16.], [7., 20.]])
3 Z_new = (Xraw_new - mu) @ W
4 Xraw_new_hat = Z_new @ W.T + mu
5 print("Z_new.round(3):\n", Z_new.round(3))
6 print("Xraw_new_hat.round(3):\n",
7       Xraw_new_hat.round(3))
```

```
Z_new.round(3):
[[0.   ]
 [4.472]]
Xraw_new_hat.round(3):
[[ 5.    16.   ]
 [ 7.051 19.974]]
```

- **Fitting** estimates μ and W from a chosen dataset.
- **Applying** uses those fixed values for other observations with the same feature definitions.
- Re-estimating the mean or directions creates a different fitted representation.

PCA relies on centering and **then** measuring variance. **What would happen if we computed variance directly, without centering first?**

| Computing spread for PCA starts with **an exact identity**

For one feature's values $x_1, \dots, x_n$, let $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$. The mean $\bar{x}$ is constant with respect to i .

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 &= \frac{1}{n} \sum_{i=1}^n (x_i^2 - 2x_i\bar{x} + \bar{x}^2) && \text{(Expand the square inside the sum)} \\ &= \frac{1}{n} \sum_{i=1}^n x_i^2 - \frac{2\bar{x}}{n} \sum_{i=1}^n x_i + \frac{1}{n} \sum_{i=1}^n \bar{x}^2 && \text{(Distribute the sum; pull out constants)} \\ &= \frac{1}{n} \sum_{i=1}^n x_i^2 - \frac{2\bar{x}}{n} (n\bar{x}) + \frac{1}{n} (n\bar{x}^2) && (\sum_{i=1}^n x_i = n\bar{x}; n \text{ copies of } \bar{x}^2) \\ &= \frac{1}{n} \sum_{i=1}^n x_i^2 - 2\bar{x}^2 + \bar{x}^2 && \text{(Cancel the factors of } n) \\ &= \frac{1}{n} \sum_{i=1}^n x_i^2 - \bar{x}^2 && \text{(Combine terms)} \end{aligned}$$

| Will mathematically equivalent computations give the same answer?

Take the five numbers $10^9 + 1, \dots, 10^9 + 5$. Their mean is $10^9 + 3$. **What variances will the two computations report?** Predict before revealing.

```
1 v = 1e9 + np.arange(1., 6.)
2 centered = v - v.mean()
3 center_first = np.mean(centered**2)
4 subtract_totals = np.mean(v**2) - v.mean()**2
5 print("Centered:", center_first)
6 print("Raw squares:", subtract_totals)
```

Centered: 2.0
Raw squares: 0.0

Choice	Centered	Raw squares
A	2	2 (up to a tiny rounding error)
B	2	0
C	2	Overflow: the squared values are too large

| Numerical errors can erase the entire answer

```
1 print("Centered values:", centered)
2 print("Mean of squares:", np.mean(v**2))
3 print("Square of mean:", v.mean()**2)
4 print("Centered:", center_first)
5 print("Raw squares:", subtract_totals)
```

```
Centered values: [-2. -1.  0.  1.  2.]
Mean of squares: 1.000000006e+18
Square of mean: 1.000000006e+18
Centered: 2.0
Raw squares: 0.0
```

- Centering first gives $(-2, -1, 0, 1, 2)$: the mean square is $(4 + 1 + 0 + 1 + 4)/5 = 2$.
- The other formula subtracts rounded values near 10^{18} . The small difference is lost: **catastrophic cancellation**.
- This is **100% relative error**: zero instead of two, not just a changed last decimal.

Numerical stability: choose computations that control the effect of rounding errors. A correct mathematical formula alone does not ensure a reliable computation.

Later: log probabilities help avoid tiny probabilities rounding to zero; softmax needs careful implementation to avoid overflow.

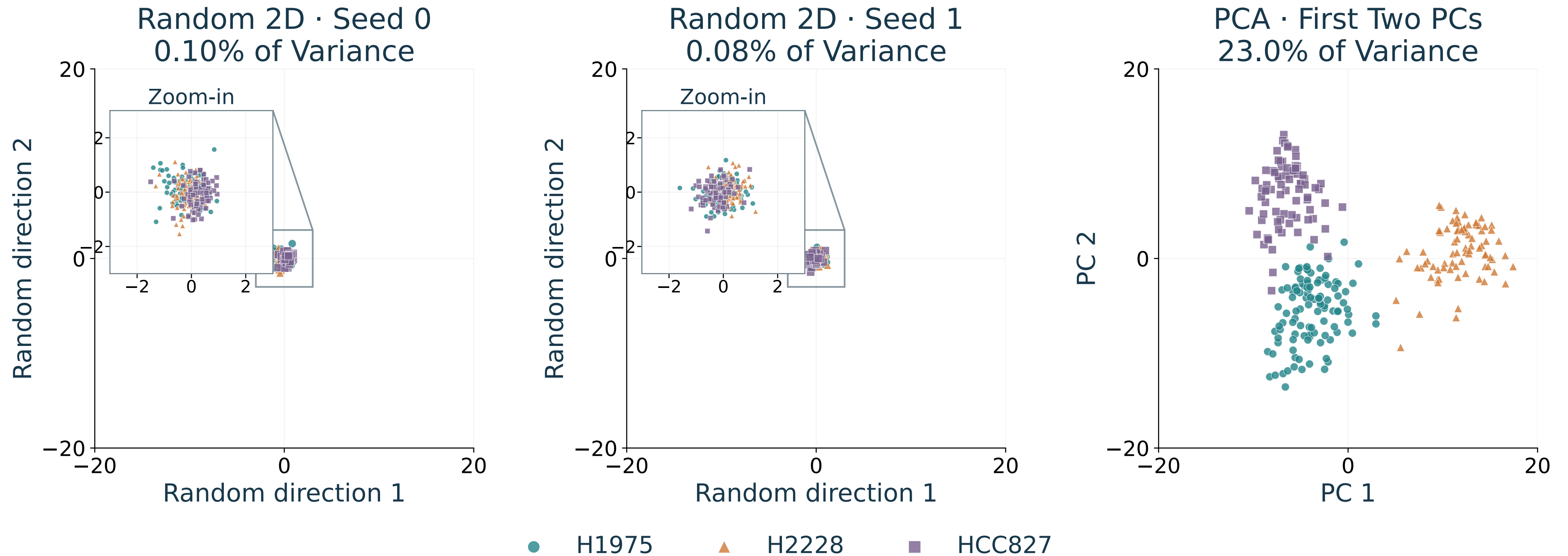
From computing PCA to interpreting real data

Back to our original question: what should a useful representation preserve?

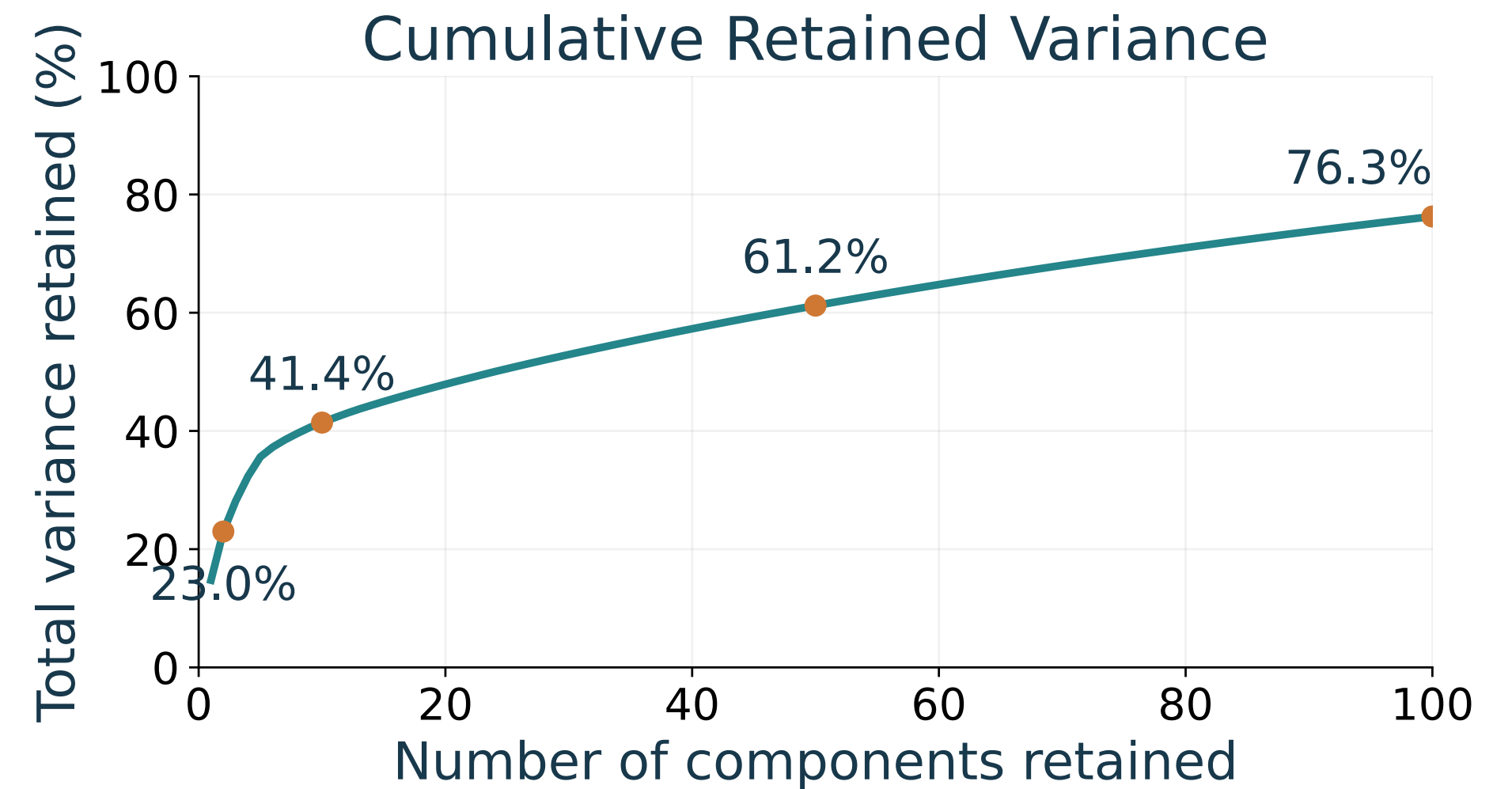
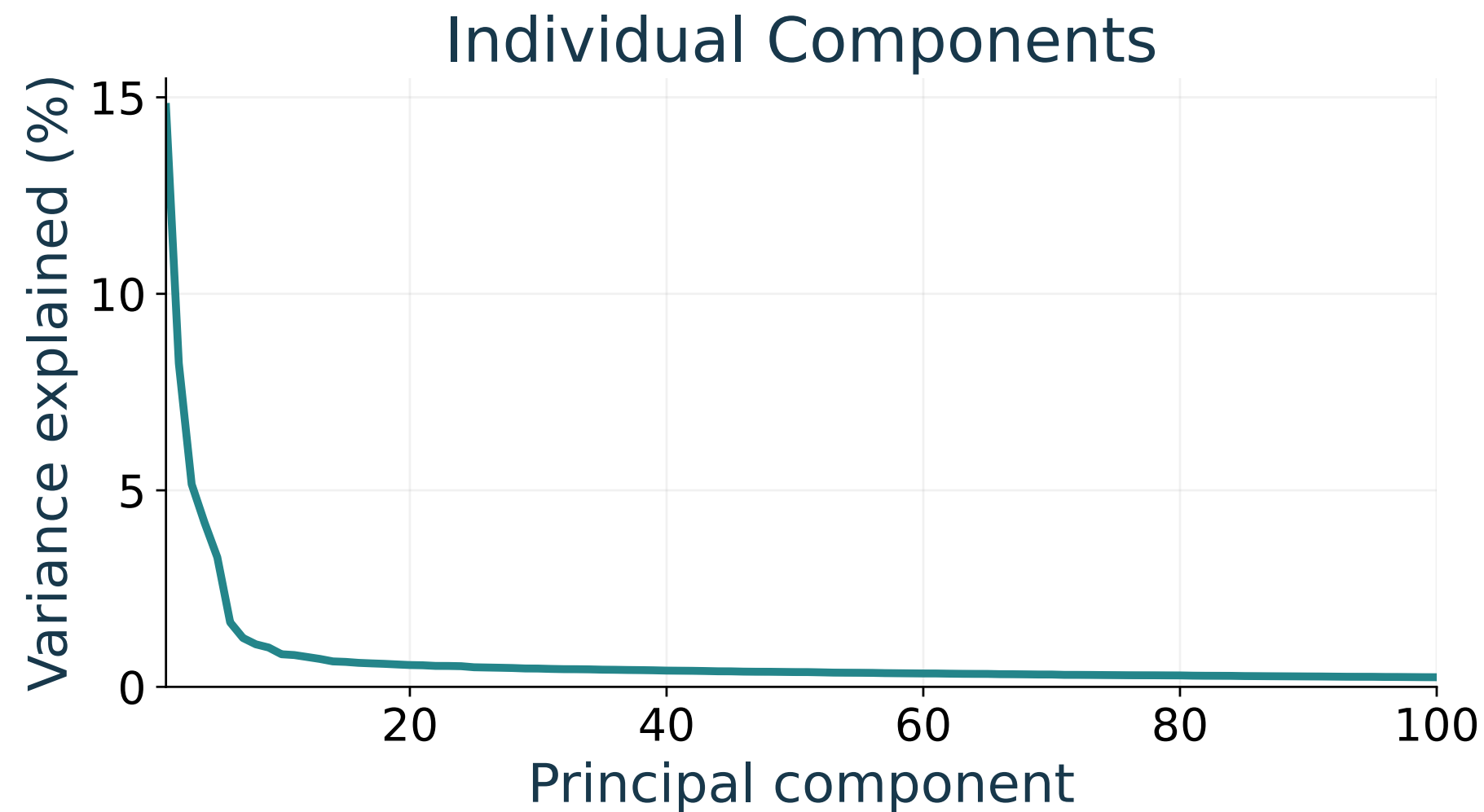
We can now compute PCA. Does it reveal useful structure in real gene-expression data?

| PCA reveals known cell-line structure **without seeing the labels**

Real gene expression: 274 cells from three lung-cancer cell lines; 2,000 selected genes per cell.



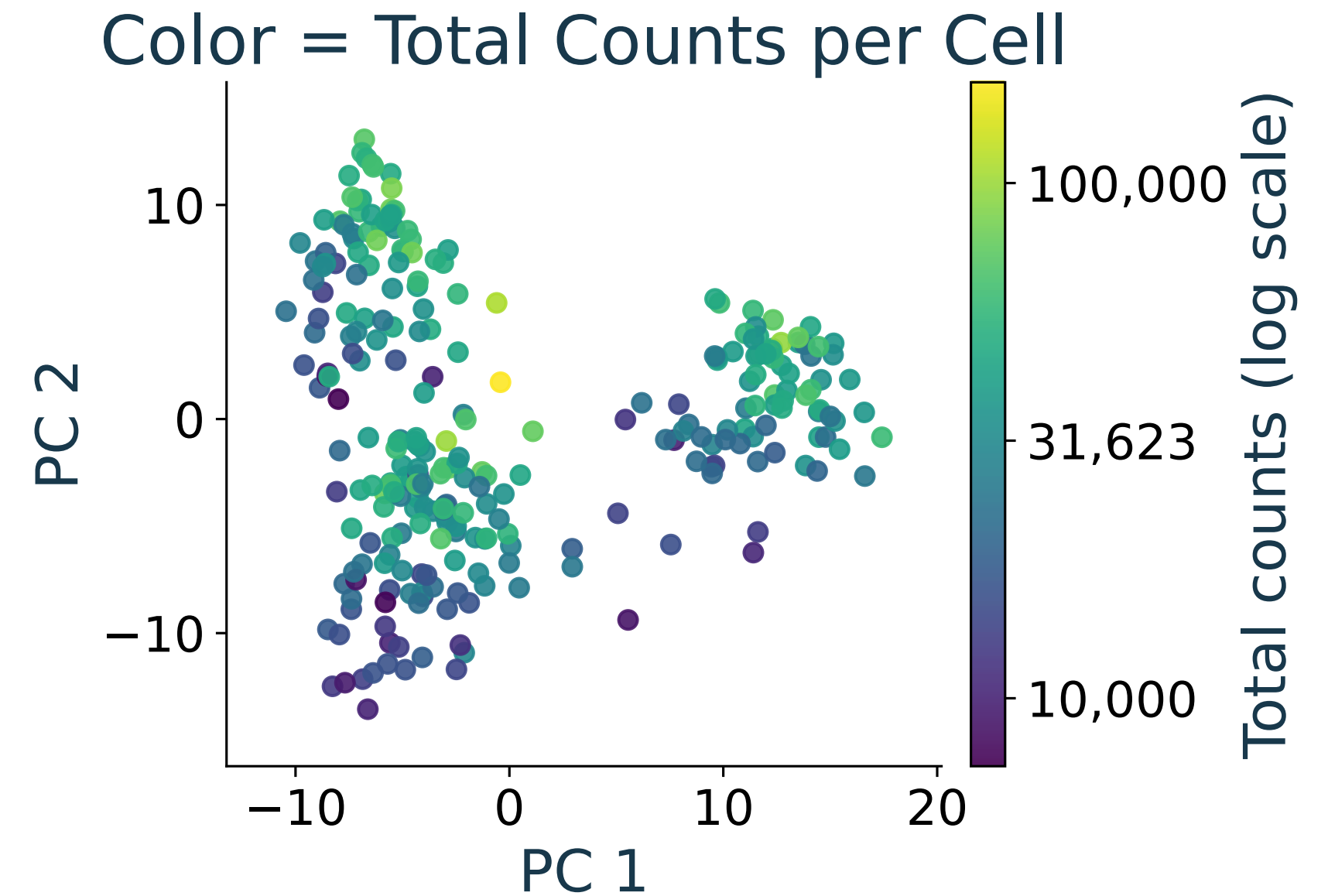
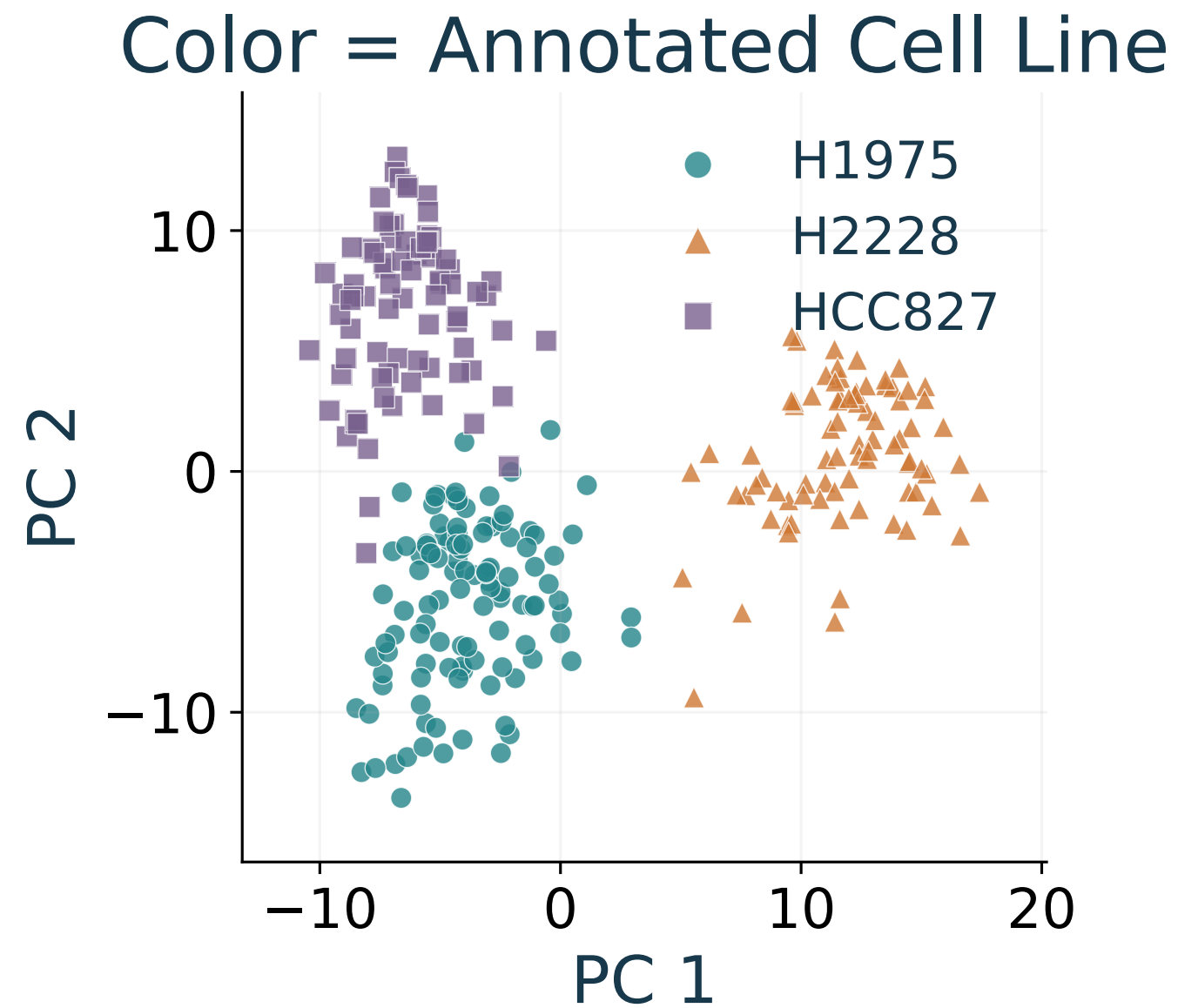
| Two PCs reveal groups but retain **only 23% of the variance**



$$\text{Retained fraction}(k) = \frac{\sum_{j=1}^k \sigma_j^2}{\sum_{j=1}^r \sigma_j^2}, \quad r = \text{rank}(X).$$

- **Visualization:** two coordinates already reveal cell-line structure.
- **Reconstruction:** 100 PCs retain 76.3%; reaching 90% requires 173 PCs.

| The same PCA plot supports different questions about usefulness



Discuss: Do these plots justify saying that PCA preserved the biology we care about?

- What does agreement with known cell-line labels support?
- What might total counts per cell explain—and what evidence would you want next?

| We fitted these observations; **new cells need new evidence**

From the original question...	...to what we established
Make thousands of measurements easier to inspect	Choose what the representation should preserve
Preserve observations under a squared-error criterion	PCA gives the best rank- k linear reconstruction
Compute that representation	Center, use SVD, and attend to finite precision
Inspect this cell dataset	Two PCs reveal known groups, but retain only 23% of variance

For new cells: keep the fitted preprocessing, gene set, mean, and directions fixed.

Before moving on: what general lessons does this case teach us about building useful representations?

| A useful solution begins with **deciding what to preserve**

Gene-expression measurements → desired representation → objective → constraints → solution

Choice we made	What it allowed us to specify
Which measurements, preprocessing, and units?	The observations and geometry
What should the representation preserve?	Squared reconstruction error
Which transformations are allowed?	A linear representation with k coordinates
What is the best solution under those choices?	Leading right singular vectors of centered data

Formalization makes a vague goal solvable. Choosing the goal remains our responsibility.

| Representations preserve **some structure and discard other structure**

In our cell-expression example	What it teaches us
Two PCs reveal known cell-line groups	A small representation can expose useful structure
Those two PCs retain only 23% of variance	Visible groups do not imply faithful reconstruction
100 PCs retain 76.3% of variance	The number of coordinates depends on the purpose

- Rank and discarded singular values describe **linear information loss**.
- Low-dimensional structure need not be linear: recall the curved-data example.
- Large variance is not automatically the variation that matters for our task.

Ask what was preserved, what was lost, and whether that matches the intended use.

A proof, a working computation, and useful evidence are **different achievements**

Question	What this arc established
Did we solve the stated problem?	SVD gives the optimal linear PCA reconstruction
Did we compute it correctly and efficiently?	Shapes, broadcasting, operation order, and finite precision matter
Does the result accomplish our purpose?	Real-data evidence supports particular claims, with limits

A mathematical guarantee does not establish every practical claim we might want to make.

Next question: What evidence would justify trusting a learned rule on new observations?