Chapter 2: Linear Algebra for Machine Learning

Master in Computer Science (M1) — Applications with Real Data in Python

Youssef SALMAN

Academic Year 2025–2026

1 Introduction

Machine learning is the field of computer science that focuses on building algorithms capable of learning patterns from data and making predictions or decisions without being explicitly programmed. From recommending movies on streaming platforms to detecting anomalies in network traffic or recognizing faces in images, machine learning is now embedded in daily technology.

Linear algebra, on the other hand, is a branch of mathematics concerned with vectors, matrices, and linear transformations. It provides the formal language to describe high-dimensional data and the operations we perform on them. Whenever we work with a dataset of thousands of features, compute a weighted sum of inputs, or rotate data into a new coordinate system, we are using the principles of linear algebra.

These two worlds come together naturally: machine learning relies heavily on linear algebra as its computational backbone. Training a regression model involves solving systems of linear equations; deep learning requires millions of matrix multiplications; dimensionality reduction methods such as Principal Component Analysis (PCA) depend on eigenvalues and eigenvectors. Without linear algebra, machine learning would lack the tools to represent, manipulate, and transform data efficiently.

Real applications illustrate this connection clearly. Search engines use matrix decompositions (such as the singular value decomposition) to index billions of documents. Image compression standards rely on linear transformations to reduce file size without losing essential details. Recommendation systems use vector similarity to suggest products or movies that are close to a user's past choices. In each of these cases, linear algebra not only underlies the algorithm but also determines its performance and scalability.

This chapter is therefore devoted to exploring the linear algebra concepts most relevant for machine learning. We begin with the basic objects, vectors and matrices, and study their arithmetic. We then interpret matrices as linear transformations and examine their structure through eigenvalues and eigenvectors. Finally, we introduce matrix decompositions, including the singular value decomposition, and show how these ideas culminate in one of the most widely used techniques in data science: Principal Component Analysis (PCA).

2 Vectors and Matrices

2.1 Vectors

A vector is an ordered collection of numbers. In machine learning, a data point with d features is naturally represented as a vector in $\mathbb{R}^d$.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}.$$

The length (Euclidean norm) of $\mathbf{x}$ is

$$\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_d^2}.$$

Two important operations are the **dot product**, $\mathbf{x}\cdot\mathbf{y} = \sum_i x_i y_i$, which measures similarity, and the **cosine similarity**, $\cos\theta = \frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}$, which is widely used in information retrieval.

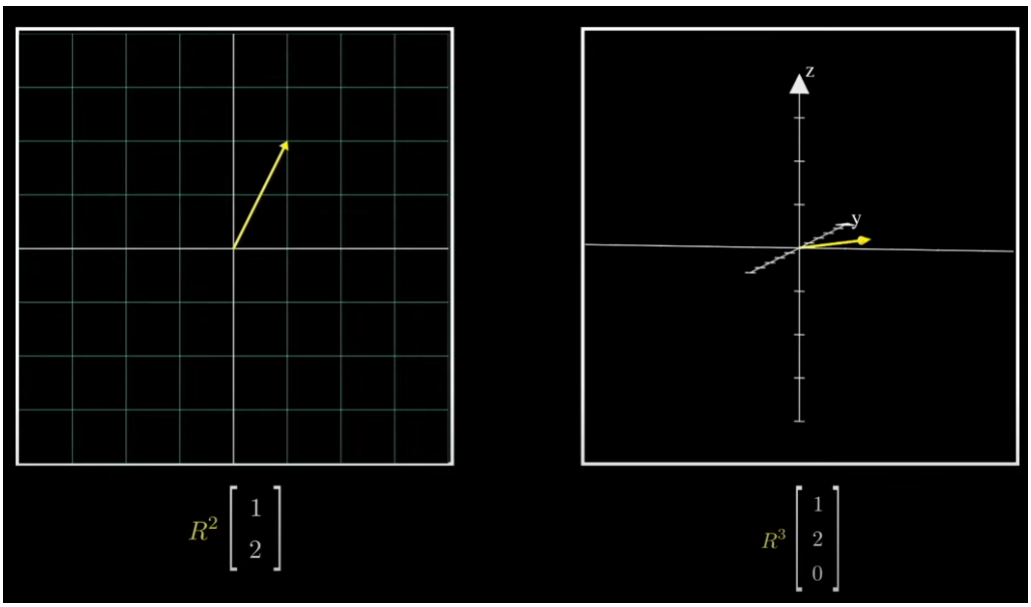


Figure 1: Point of view for 2 spaces

2.2 Matrices

A matrix is a rectangular array of numbers with m rows and n columns:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

Matrices naturally represent datasets (rows = observations, columns = features) or linear transformations.

The **matrix-vector product** $A\mathbf{x}$ maps $\mathbb{R}^n \to \mathbb{R}^m$, combining features through weighted sums.

3 Matrix Arithmetic

3.1 Addition and Scalar Multiplication

If A and B have the same size, then $(A+B)_{ij} = a_{ij} + b_{ij}$. For a scalar λ, $(\lambda A)_{ij} = \lambda a_{ij}$.

3.2 Matrix Multiplication

Given A $(m \times n)$ and B $(n \times p)$, their product $C = AB$ is $m \times p$ with entries

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Matrix multiplication encodes composition of linear transformations. It is associative but not commutative.

3.3 Transpose and Inverse

Transpose of a matrix

The transpose of a matrix A, denoted A^T, is obtained by interchanging rows and columns. If

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix},$$

then

$$A^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix}.$$

Geometrically, transposition reflects a matrix along its diagonal. In machine learning, transposes appear when switching between column-oriented and row-oriented representations of datasets, and when forming the normal equations $X^T X$.

For conformable matrices,

$$(AB)^T = B^T A^T, \qquad (A+B)^T = A^T + B^T.$$

Inverse of a matrix

The **inverse** of a square matrix A, denoted A^{-1}, is defined (if it exists) as the matrix that satisfies

$$AA^{-1} = A^{-1}A = I,$$

where I is the identity matrix. If A is invertible, we call it **nonsingular**; otherwise, it is **singular**.

Existence of the inverse. Not every square matrix has an inverse. A matrix A is invertible if and only if:

- A is square $(n \times n)$, and

- $\det(A) \neq 0$, which is equivalent to saying A has full rank (its rows and columns are linearly independent).

If $\det(A) = 0$, then some linear dependence exists, and no unique inverse can be found.

Example.

$$B = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}, \quad \det(B) = (1)(4) - (2)(2) = 0.$$

Thus B is singular and non-invertible. Geometrically, B maps the plane into a line, collapsing information and making the transformation irreversible.

Properties of the inverse. For invertible matrices A and B of the same size:

$$\begin{aligned} (AB)^{-1} &= B^{-1}A^{-1}, \\ (A^T)^{-1} &= (A^{-1})^T, \\ (A^{-1})^{-1} &= A. \end{aligned}$$

These rules are essential in algebraic manipulations involving matrices.

Application in machine learning. The matrix inverse is central in deriving the closed-form solution of the **ordinary least squares** (OLS) estimator. If X is the design matrix and y the vector of outputs, the regression coefficients are formally given by

$$\hat{\beta} = (X^T X)^{-1} X^T y,$$

provided $X^T X$ is invertible. In practice, $X^T X$ may be close to singular (multicollinearity), which leads to numerical instability. This motivates techniques such as **regularization** (Ridge regression), replacing $(X^T X)^{-1}$ with $(X^T X + \lambda I)^{-1}$ for greater stability.

Numerical note. In implementations, $\hat{\beta}$ is not computed by explicitly inverting $X^T X$, but through more stable decompositions:

- **QR decomposition:** factorizes X into $X = QR$, with Q orthogonal and R upper triangular. The least-squares problem reduces to solving $R\hat{\beta} = Q^T y$, which is stable and efficient.
- **Singular Value Decomposition (SVD):** factorizes $X = U\Sigma V^T$, where Σ contains singular values. SVD directly reveals the rank and allows computing the Moore–Penrose pseudoinverse X^+, giving the *minimum-norm* solution $\hat{\beta} = X^+ y$ even when X is not full rank.

Thus, QR is preferred for well-conditioned problems, while SVD is more robust for ill-conditioned or rank-deficient cases.

4 Linear Transformations

Linear transformation

A **linear transformation** is a mapping $T : \mathbb{R}^n \to \mathbb{R}^m$ that preserves both vector addition and scalar multiplication:

$$T(\mathbf{x} + \mathbf{y}) = T(\mathbf{x}) + T(\mathbf{y}), \quad T(\alpha\mathbf{x}) = \alpha T(\mathbf{x}),$$

for all vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and scalars $\alpha \in \mathbb{R}$.

4.1 Matrix Representation

Every linear transformation can be represented as multiplication by a matrix. That is, for some $A \in \mathbb{R}^{m \times n}$ we can write

$$T(\mathbf{x}) = A\mathbf{x}.$$

Conversely, every matrix defines a linear transformation. This equivalence allows us to study transformations through the properties of their matrices.

4.2 Geometric Interpretations in $\mathbb{R}^2$

Multiplying by a matrix can produce different geometric effects:

- **Scaling:** $T(\mathbf{x}) = \alpha\mathbf{x}$ enlarges or shrinks vectors.
- **Reflection:** $T(x, y) = (x, -y)$ reflects points across the x-axis.
- **Rotation:**
$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$
rotates vectors counterclockwise by angle θ.
- **Shear:**
$$S = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$
shears the plane by shifting points horizontally.
- **Projection:**
$$P = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$
projects vectors onto the x-axis, collapsing all y components to zero.

4.3 Kernel and Image

Given a linear transformation $T : \mathbb{R}^n \to \mathbb{R}^m$, two fundamental subspaces describe its behavior.

Kernel

The **kernel** (or null space) of T is the set of all input vectors that are mapped to zero:

$$\ker(T) = \{\mathbf{x} \in \mathbb{R}^n \mid T(\mathbf{x}) = 0\}.$$

If the kernel is only the zero vector, then T is one-to-one (injective).

Image

The **image** (or range) of T is the set of all possible outputs:

$$\mathrm{Im}(T) = \{T(\mathbf{x}) \mid \mathbf{x} \in \mathbb{R}^n\}.$$

If the image is the whole $\mathbb{R}^m$, then T is onto (surjective).

Together, the kernel and image describe how much information is lost or preserved by the transformation.

4.4 Rank and Invertibility

Rank and Invertibility

The **rank** of T is the dimension of its image. It measures how many independent directions are preserved by the transformation.
If $T : \mathbb{R}^n \to \mathbb{R}^n$ and $\mathrm{rank}(T) = n$, then the transformation is both injective and surjective, hence invertible. Its inverse T^{-1} is also linear.

Nullity Theorem.

For every linear transformation $T : \mathbb{R}^n \to \mathbb{R}^m$,

$$\dim(\ker T) + \dim(\mathrm{Im}\, T) = n.$$

This fundamental identity links the null space and the image. For example, if the kernel is large, the image must be correspondingly smaller.

4.5 Composition of Transformations

The composition of two linear transformations corresponds to matrix multiplication. If $T_1(\mathbf{x}) = A\mathbf{x}$ and $T_2(\mathbf{x}) = B\mathbf{x}$, then

$$T_2(T_1(\mathbf{x})) = B(A\mathbf{x}) = (BA)\mathbf{x}.$$

This shows that matrix multiplication encodes the chaining of transformations in the correct order.

Example (Projection). Consider the projection

$$P = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

It satisfies $P^2 = P$, meaning repeated application does nothing new (idempotent). Moreover, $P^T = P$, so P is an **orthogonal projection** onto the x-axis.

4.6 Applications in Machine Learning

Linear transformations are everywhere in machine learning:

- In **neural networks**, each layer applies $T(\mathbf{x}) = W\mathbf{x} + b$, a linear transformation followed by a nonlinearity.
- In **dimensionality reduction**, PCA projects data onto a lower-dimensional subspace using a linear transformation defined by principal components.
- In **computer vision**, rotations, scalings, and reflections are modeled by linear transformations on pixel vectors.
- In **feature engineering**, normalization and standardization can be expressed as linear transformations.

5 Eigenvalues and Eigenvectors

Eigenvector

Let A be a square $n \times n$ matrix. A nonzero vector $\mathbf{v} \in \mathbb{R}^n$ (or $\mathbb{C}^n$) is called an **eigenvector** of A if there exists a scalar λ such that

$$A\mathbf{v} = \lambda\mathbf{v}.$$

The scalar λ is the corresponding **eigenvalue**.

Geometrically, eigenvectors point in directions that are preserved under the transformation A, while eigenvalues indicate the factor by which these directions are stretched or shrunk.

5.1 Existence

Not every real matrix has real eigenvalues, but every square matrix has eigenvalues if we allow complex numbers. This is a consequence of the Fundamental Theorem of Algebra: the characteristic polynomial of degree n has n (possibly complex) roots.

- Real symmetric (or Hermitian) matrices have n real eigenvalues.
- Orthogonal matrices have eigenvalues of modulus 1.
- Diagonal matrices have eigenvalues equal to their diagonal entries.

5.2 Computation

To find eigenvalues, we solve

$$(A - \lambda I)\mathbf{v} = 0 \quad \iff \quad \det(A - \lambda I) = 0.$$

This determinant equation is called the **characteristic equation**. Its roots are the eigenvalues, and the null space of $(A - \lambda I)$ gives the eigenvectors.

Example 1 (real eigenvalues).

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}.$$

The characteristic polynomial is $(2-\lambda)(3-\lambda)$, giving eigenvalues $\lambda_1 = 2, \lambda_2 = 3$ with eigenvectors along the x- and y-axes.

Example 2 (complex eigenvalues).

$$B = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.$$

This is a 90° rotation. Its characteristic polynomial is $\lambda^2 + 1 = 0$, so eigenvalues are $\lambda = i, -i$. No real eigenvectors exist, but over $\mathbb{C}$ we can find eigenvectors.

5.3 Multiplicity and Diagonalizability

Each eigenvalue has an **algebraic multiplicity** (how many times it is a root of the characteristic polynomial) and a **geometric multiplicity** (dimension of its eigenspace).

$$\text{Always: geometric multiplicity } \leq \text{ algebraic multiplicity.}$$

A matrix is **diagonalizable** if there exists a basis of eigenvectors. Symmetric matrices are always diagonalizable.

5.4 Properties

- The sum of eigenvalues (with multiplicity) equals the trace: $\operatorname{tr}(A) = \sum \lambda_i$.
- The product of eigenvalues equals the determinant: $\det(A) = \prod \lambda_i$.
- Eigenvectors associated with distinct eigenvalues of a symmetric matrix are orthogonal.
- If A is invertible, its eigenvalues are all nonzero, and the eigenvalues of A^{-1} are λ_i^{-1}.
- If A is triangular, its eigenvalues are simply the diagonal entries.

5.5 Applications

Eigenvalues and eigenvectors appear throughout machine learning and computer science:

- **Principal Component Analysis (PCA):** eigenvectors of the covariance matrix give directions of maximum variance.
- **Spectral clustering:** eigenvectors of graph Laplacians identify community structure in networks.
- **PageRank:** the importance of web pages is given by the dominant eigenvector of a stochastic matrix.
- **Stability analysis:** in dynamical systems, signs of eigenvalues of the system matrix determine stability.

6 Matrix Decompositions

6.1 Eigen-decomposition

If A is diagonalizable, it can be written as

$$A = PDP^{-1},$$

where D is diagonal with eigenvalues of A, and P contains eigenvectors as columns. This decomposition simplifies powers of A and differential equations.

6.2 Singular Value Decomposition (SVD)

Every $m \times n$ matrix A can be decomposed as

$$A = U\Sigma V^T,$$

where U and V are orthogonal matrices and Σ is diagonal with non-negative entries (singular values).

SVD is fundamental in numerical stability, dimensionality reduction, and data compression. Truncating Σ gives a low-rank approximation of A.

Exercise (QR, SVD, singular values, inverse).

(a) Let

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

Compute a *thin* QR factorization $X = QR$ with $Q \in \mathbb{R}^{3\times 2}$ orthonormal and $R \in \mathbb{R}^{2\times 2}$ upper triangular. Then solve the least-squares problem $\min_\beta \|X\beta - y\|_2$ using $R\hat{\beta} = Q^T y$.

(b) Let

$$B = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}.$$

Compute its **singular values** from $B^T B$ and give a short SVD description $B = U\Sigma V^\top$ (numerical values are fine). Decide if B is invertible; if yes, compute B^{-1}; otherwise state the Moore–Penrose pseudoinverse B^+.

(c) (Rank-deficient contrast) Consider $C = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$. Use the singular values to decide invertibility and say what C^+ provides when you solve $Cx = b$.

Solution (summary).

1. **(a) QR and LS via back-substitution).**:
   Columns $a_1 = (1, 1, 0)^\top$, $a_2 = (1, 0, 1)^\top$.

$$r_{11} = \|a_1\| = \sqrt{2}, \qquad q_1 = \tfrac{1}{\sqrt{2}}(1, 1, 0)^\top.$$

Orthogonalize a_2 against q_1: $r_{12} = q_1^\top a_2 = \frac{1}{\sqrt{2}}$, $u_2 = a_2 - r_{12}q_1 = \left(\frac{1}{2}, -\frac{1}{2}, 1\right)^\top$, $r_{22} = \|u_2\| = \sqrt{\frac{3}{2}}$. Hence $q_2 = u_2/r_{22} \approx (0.4082, -0.4082, 0.8165)^\top$.

Therefore

$$Q = \begin{bmatrix} 0.7071 & 0.4082 \\ 0.7071 & -0.4082 \\ 0 & 0.8165 \end{bmatrix}, \qquad R = \begin{bmatrix} 1.4142 & 0.7071 \\ 0 & 1.2247 \end{bmatrix}.$$

Solve $R\hat{\beta} = Q^\top y$. Since $Q^\top y = (0.7071,\ 1.2247)^\top$, back-substitution gives

$$\hat{\beta}_2 = \frac{1.2247}{1.2247} = 1, \qquad \hat{\beta}_1 = \frac{0.7071 - 0.7071 \cdot 1}{1.4142} = 0.$$

$$\boxed{\hat{\beta} = (0,\ 1)^\top}$$

Interpretation: the best LS fit uses only the second column of X.

2. **(b) SVD, singular values, and inverse of B).**:

$$B^T B = \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix}.$$

Its eigenvalues solve $\lambda^2 - 6\lambda + 1 = 0 \Rightarrow \lambda_{1,2} = 3 \pm 2\sqrt{2}$. Hence the singular values are

$$\sigma_1 = \sqrt{3 + 2\sqrt{2}} = \sqrt{2} + 1 \approx 2.4142, \qquad \sigma_2 = \sqrt{3 - 2\sqrt{2}} = \left|\sqrt{2} - 1\right| \approx 0.4142.$$

Since both $\sigma_i > 0$, B is invertible. A short SVD is $B = U\Sigma V^\top$ with $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2)$, and V formed by the orthonormal eigenvectors of $B^T B$ (any numerically computed U, V with these σ_i is acceptable).

The (true) inverse is easy to check directly:

$$B^{-1} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}.$$

(Because B is full rank, the pseudoinverse equals the inverse: $B^+ = B^{-1}$.)

3. **(c) Rank-deficient C and the pseudoinverse).**:
For $C = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$, we have $\mathrm{rank}(C) = 1$. Thus one singular value is positive and the other is $0 \Rightarrow C$ is *not* invertible. The SVD-based pseudoinverse $C^+ = V\Sigma^+ U^\top$ (with Σ^+ inverting only the nonzero singular value) returns the *minimum-norm* solution of $Cx = b$ when the system is consistent, and the minimum-norm least-squares solution when it is not.

```
import numpy as np

# ---------- Helpers ----------
def back_substitute(R, b):
    """Solve R x = b for upper-triangular R (n x n)."""
    R = np.asarray(R)
    b = np.asarray(b)
```

```
    n = R.shape[0]
    x = np.zeros(n, dtype=float)
    for i in range(n-1, -1, -1):
        x[i] = (b[i] - R[i, i+1:] @ x[i+1:]) / R[i, i]
    return x

# ============================================================
# (a) QR least-squares for X beta ~= y
# ============================================================
X = np.array([[1., 1.],
              [1., 0.],
              [0., 1.]]) # 3x2
y = np.array([1., 0., 1.])

# Thin (reduced) QR
Q, R = np.linalg.qr(X, mode="reduced") # Q: 3x2, R: 2x2, X = Q R
beta_via_qr = back_substitute(R, Q.T @ y)

# Cross-check with np.linalg.lstsq
beta_lstsq, residuals, rank, svals = np.linalg.lstsq(X, y, rcond=None)

print("----- (a) QR Least Squares -----")
print("Q =\n", Q)
print("R =\n", R)
print("Q^T y =", Q.T @ y)
print("beta (via QR) =", beta_via_qr)
print("beta (np lstsq) =", beta_lstsq)
print("residual norm =", np.linalg.norm(X @ beta_via_qr - y))
print()

# ============================================================
# (b) SVD, singular values, and inverse for B
# ============================================================
B = np.array([[1., 2.],
              [0., 1.]])

# Singular values from B^T B
BtB = B.T @ B
eigs = np.linalg.eigvalsh(BtB) # sorted eigenvalues
sing_from_BtB = np.sqrt(eigs) # singular values

# Full SVD (U, Sigma, Vt)
U, s, Vt = np.linalg.svd(B, full_matrices=True)

# Inverse (exists because rank(B)=2)
B_inv = np.linalg.inv(B)
I_check = B @ B_inv

print("----- (b) SVD & Inverse of B -----")
print("B^T B =\n", BtB)
print("Eigenvalues(B^T B) =", eigs)
```

```
print("Singular values (sqrt eigs) =", sing_from_BtB)
print("SVD singular values =", s)
print("V (right-singular vecs) =\n", Vt.T)
print("B^{-1} =\n", B_inv)
print("B @ B^{-1} ~ I ? ->", np.allclose(I_check, np.eye(2)))
# Pseudoinverse equals inverse for full-rank square B
print("pinv(B) == inv(B) ? ->", np.allclose(np.linalg.pinv(B), B_inv))
print()

# ============================================================
# (c) Rank-deficient C: pseudoinverse and min-norm solutions
# ============================================================
C = np.array([[1., 2.],
              [2., 4.]]) # rank 1

rank_C = np.linalg.matrix_rank(C)
U_C, s_C, Vt_C = np.linalg.svd(C, full_matrices=True)
C_pinv = np.linalg.pinv(C)

# Choose a consistent b (in column space of C): [3,6] = 3 * [1,2]
b_consistent = np.array([3., 6.])
x_min_norm_consistent = C_pinv @ b_consistent # exact solution with minimum
    norm

# Choose an inconsistent b' (not in column space): [1,0]
b_inconsistent = np.array([1., 0.])
x_min_norm_ls = C_pinv @ b_inconsistent # min-norm least-squares solution

print("----- (c) Rank-deficient C -----")
print("rank(C) =", rank_C)
print("Singular values of C =", s_C)
print("C^+ (pseudoinverse) =\n", C_pinv)

print("\nConsistent system C x = b with b=[3,6]^T:")
print("x_min_norm =", x_min_norm_consistent)
print("Check Cx =", C @ x_min_norm_consistent, " (should equal b)")

print("\nInconsistent system C x = b' with b'=[1,0]^T (least squares):")
print("x_min_norm_ls =", x_min_norm_ls)
print("Residual norm =", np.linalg.norm(C @ x_min_norm_ls - b_inconsistent))
```

Takeaways.

- **QR** solves LS via $R\hat{\beta} = Q^\top y$ (no explicit inverse).
- **SVD** exposes rank/conditioning; singular values $\downarrow \Rightarrow$ ill-conditioning.
- **Pseudoinverse** (X^+) gives the minimum-norm solution when X is rank-deficient.

7 Application: Principal Component Analysis (PCA)

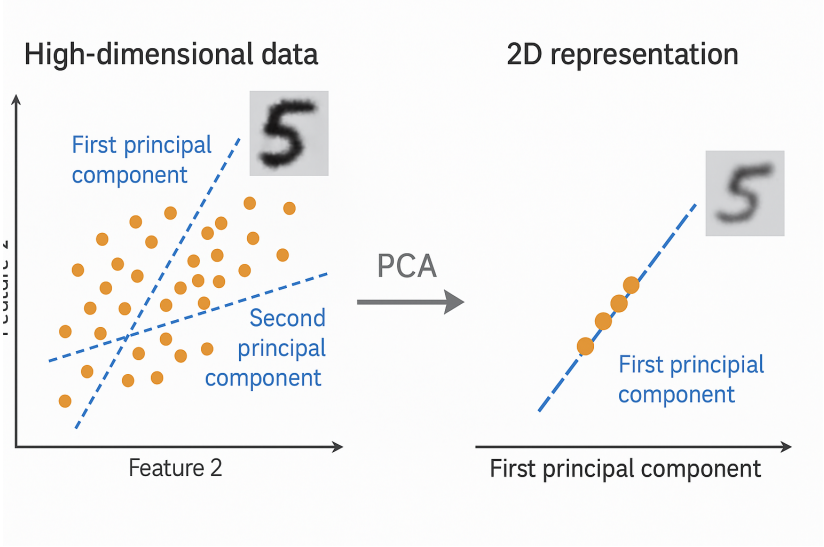
Modern datasets are often high-dimensional: an image with 100×100 pixels has 10,000 features, and a text document represented by word counts may have tens of thousands of dimensions. High-dimensional data are difficult to visualize, computationally expensive to process, and often contain redundancy because many features are correlated.

Principal Component Analysis (PCA) is a linear transformation technique that reduces dimensionality by finding new orthogonal axes (principal components) along which the data vary the most. Instead of keeping all original features, PCA projects the data onto the leading components, preserving as much information (variance) as possible.

7.1 Geometric and Statistical Intuition

The first principal component is the direction in feature space along which the variance of the data is maximized. The second principal component is orthogonal to the first and captures the largest remaining variance, and so on.

Equivalently, PCA finds the linear transformation $Y = WX$ (with W built from eigenvectors of the covariance matrix) that maximizes variance in the projected data while minimizing information loss.



7.2 Algorithmic Steps

Given a centered data matrix X (n samples, d features):

1. Compute the covariance matrix
$$C = \frac{1}{n-1} X^T X.$$
2. Solve the eigenvalue problem $C\mathbf{v} = \lambda\mathbf{v}$ to obtain eigenvalues λ_i and eigenvectors $\mathbf{v}_i$.
3. Sort eigenvectors in decreasing order of eigenvalues (largest variance first).
4. Choose the first k eigenvectors and project data:
$$Z = XW_k, \quad W_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k].$$

7.3 Worked Numerical Example (2D Data)

Suppose we have three 2D points:

$$(2,0), \quad (0,2), \quad (3,3).$$

Step 1. Center the data by subtracting the mean $(1.67, 1.67)$. The centered points are:

$$(0.33, -1.67), \quad (-1.67, 0.33), \quad (1.33, 1.33).$$

Step 2. Compute the covariance matrix:

$$C = \frac{1}{3-1}\begin{bmatrix} 0.33 & -1.67 & 1.33 \\ -1.67 & 0.33 & 1.33 \end{bmatrix}\begin{bmatrix} 0.33 & -1.67 & 1.33 \\ -1.67 & 0.33 & 1.33 \end{bmatrix}^T = \begin{bmatrix} 2.33 & 1.17 \\ 1.17 & 2.33 \end{bmatrix}.$$

Step 3. Solve $\det(C - \lambda I) = 0$:

$$\det\begin{bmatrix} 2.33-\lambda & 1.17 \\ 1.17 & 2.33-\lambda \end{bmatrix} = 0 \quad \Rightarrow \quad (\lambda - 3.5)(\lambda - 1.17) = 0.$$

Eigenvalues: $\lambda_1 = 3.5$, $\lambda_2 = 1.17$.

Step 4. Eigenvectors: For $\lambda_1 = 3.5$, eigenvector $\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ (direction along the diagonal). For $\lambda_2 = 1.17$, eigenvector $\mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$.

Step 5. Interpretation: The data vary most along the direction $(1,1)$, which explains about $\frac{3.5}{3.5+1.17} \approx 75\%$ of the variance. Projecting onto $\mathbf{v}_1$ reduces 2D data to 1D while keeping most information.

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

# ------------------------------------------------------------
# Data: three 2D points
# ------------------------------------------------------------
X = np.array([[2.0, 0.0],
              [0.0, 2.0],
              [3.0, 3.0]]) # shape (3, 2)
n = X.shape[0]

# 1) Center the data
mu = X.mean(axis=0)
Xc = X - mu

print("Mean (should be close to [1.6667, 1.6667]):", mu)
print("Centered data:\n", Xc)

# 2) Sample covariance (divide by n-1)
# np.cov with rowvar=False returns the (biased/unbiased) covariance across
    columns
C = np.cov(X, rowvar=False, bias=False) # same as (Xc.T @ Xc)/(n-1)
print("Sample covariance matrix C =\n", C)
```

```
# 3a) PCA via eigen-decomposition of covariance
evals, evecs = np.linalg.eigh(C) # returns ascending eigenvalues
idx = np.argsort(evals)[::-1] # sort descending
evals = evals[idx]
evecs = evecs[:, idx]
print("Eigenvalues (descending) =", evals)
print("Eigenvectors (columns) =\n", evecs)

# 3b) PCA via SVD of centered data
# Xc = U S Vt, PCs are columns of V, singular values s relate to variance by s
    ^2/(n-1)
U, s, Vt = np.linalg.svd(Xc, full_matrices=False)
V = Vt.T
explained_var = (s**2) / (n - 1)
print("Singular values =", s)
print("Explained variances via SVD (s^2/(n-1)) =", explained_var)
print("Principal directions via SVD (columns of V) =\n", V)

# 4) Project onto first principal component (PC1)
pc1 = evecs[:, 0] # same direction as V[:, 0] up to sign
X_proj_1d = Xc @ pc1 # coordinates along PC1
X_proj_2d = np.outer(X_proj_1d, pc1) # back to 2D in centered coords

print("1D coordinates on PC1:", X_proj_1d)

# ------------------------------------------------------------
# 5) Plots
# (a) Centered points with principal axes
# (b) Projections onto PC1 (line)
# ------------------------------------------------------------

# Helper: draw a principal axis as a segment centered at origin
def draw_axis(ax, direction, length=2.0, **kwargs):
    d = direction / np.linalg.norm(direction)
    p1 = -length * d
    p2 = length * d
    ax.plot([p1[0], p2[0]], [p1[1], p2[1]], **kwargs)

# (a) Centered data with PCs
fig, ax = plt.subplots(figsize=(5, 5))
ax.scatter(Xc[:, 0], Xc[:, 1], s=60, label="Centered points")

# Principal axes (scale by sqrt of eigenvalues to suggest variance)
draw_axis(ax, evecs[:, 0] * np.sqrt(evals[0]), length=2.0, color="C1",
    linewidth=2, label="PC1")
draw_axis(ax, evecs[:, 1] * np.sqrt(evals[1]), length=2.0, color="C2",
    linewidth=2, label="PC2")

# Coordinate axes for reference
ax.axhline(0, color="gray", linewidth=0.8)
```

```
ax.axvline(0, color="gray", linewidth=0.8)

ax.set_aspect("equal", adjustable="box")
ax.set_title("PCA on centered data: points and principal axes")
ax.set_xlabel("x (centered)")
ax.set_ylabel("y (centered)")
ax.legend(loc="best")
plt.show()

# (b) Projections on PC1 (show original centered points and their drops to PC1
    line)
fig, ax = plt.subplots(figsize=(5, 5))
ax.scatter(Xc[:, 0], Xc[:, 1], s=60, label="Centered points")
ax.scatter(X_proj_2d[:, 0], X_proj_2d[:, 1], s=60, marker="x", label="
    Projections on PC1")

# Draw PC1 axis line
draw_axis(ax, pc1, length=3.0, color="C1", linewidth=2, label="PC1 line")

# Connect each point to its projection
for i in range(n):
    ax.plot([Xc[i, 0], X_proj_2d[i, 0]],
            [Xc[i, 1], X_proj_2d[i, 1]],
            linestyle="--", color="gray", linewidth=1)

ax.axhline(0, color="gray", linewidth=0.8)
ax.axvline(0, color="gray", linewidth=0.8)
ax.set_aspect("equal", adjustable="box")
ax.set_title("Projection onto PC1 (1D)")
ax.set_xlabel("x (centered)")
ax.set_ylabel("y (centered)")
ax.legend(loc="best")
plt.show()

# ------------------------------------------------------------
# 6) Variance explained
# ------------------------------------------------------------
total_var = evals.sum()
explained_pc1 = evals[0] / total_var
explained_pc2 = evals[1] / total_var
print("Explained variance ratio: PC1 =", explained_pc1, ", PC2 =",
    explained_pc2)
```

7.4 Interpretation in Machine Learning

PCA is widely used:

- **Visualization:** Reducing thousands of features to 2 or 3 components makes it possible to plot and explore patterns.
- **Noise reduction:** By discarding components with small eigenvalues, we remove

directions dominated by noise.

- **Preprocessing:** PCA is often applied before clustering or regression to avoid multicollinearity.
- **Face recognition:** Images are represented as high-dimensional vectors. PCA extracts "eigenfaces," which capture the main modes of variation, enabling fast and efficient recognition.
- **Recommender systems:** User–item matrices can be compressed using PCA or SVD, revealing latent factors that drive preferences.

8 Exercice - Blood Pressure Data: PCA Interpretation

We consider the following dataset of blood pressure measurements (systolic and diastolic):

$$X = \begin{bmatrix} 126 & 78 \\ 128 & 80 \\ 128 & 82 \\ 130 & 82 \\ 130 & 84 \\ 132 & 86 \end{bmatrix}$$

(a) **Center the data.** Compute the column means of systolic and diastolic blood pressure, then subtract them to obtain the centered data matrix.

(b) **Covariance matrix.** Compute the covariance matrix

$$C = \frac{1}{n-1} X_c^\top X_c$$

where X_c is the centered data.

(c) **Eigen-decomposition.** Find the eigenvalues and eigenvectors of C.

(d) **Interpretation.** - Which eigenvalue is largest? What percentage of the total variance does it explain? - What does the associated eigenvector represent in terms of blood pressure? - Relate your findings to the correlation between systolic and diastolic pressure.

9 Conclusion

This chapter has established the linear algebraic foundations of machine learning. Vectors and matrices provide the language for representing data and transformations. Matrix arithmetic describes how features combine. Eigenvalues and eigenvectors reveal invariant directions, while matrix decompositions such as the SVD give powerful tools for analyzing structure and compressing data. PCA illustrates the culmination of these ideas in a widely used method for dimensionality reduction.

Together, these concepts form a crucial bridge between mathematics and practical machine learning algorithms, preparing students to implement and understand them in Python.