

# STAT243 Lecture 10 Linear Algebra

## Logic ▾

许多统计学与机器学习方法都依赖于线性代数操作，例如矩阵乘法或矩阵分解（decomposition）。这些运算构成了模型拟合与分析的基础。常见的应用包括：

- 线性回归与广义回归模型
- 深度神经网络（矩阵乘法是反向传播的核心）
- 主成分分析 (PCA) 及其推广形式（如核 PCA、稀疏 PCA 等）
- 各类矩阵分解（SVD、QR、Cholesky 等）

## 1 Key Principle 关键原则

数学表达式的形式与其在计算机上的实现方式可能完全不同。

良好的计算策略既能加快运算，也能改善数值稳定性。

### 1.1 Example 1

若  $X$ 、 $Y$  为矩阵， $z$  为向量，计算  $X(Yz)$  比  $(XY)z$  更高效。

- $(XY)z$  需要先计算整个矩阵乘法
- 而  $X(Yz)$  只需先计算  $Yz$ （得到一个向量），再进行矩阵向量乘

### 1.2 Example 2

回归估计式  $(X^T X)^{-1} X^T Y$

- 实际实现时并不直接计算  $X^T X$  或其逆矩阵
- 许多算法（如 QR 或 SVD 分解）完全避免了求逆操作

### 1.3 Example 3

若要交换矩阵  $A$  的两行，可引入置换矩阵  $P$ ：

- 理论上  $PAB$  可实现行交换
- 在实际计算中，很多实现仅改变索引，而无需重新存储矩阵元素

## 2 Computational Complexity

### 2.1 基本定义 (Definition)

计算复杂度通过计数加法/减法与乘法/除法操作量来衡量算法效率。

常使用 **大 O 符号**  $O(f(n))$  表示计算量的增长阶：

$$\text{\#operations} \propto f(n)$$

- $O(n^3)$ ：多项式时间（常见于矩阵运算）

- $O(b^n)$ : 指数时间 (极慢)
- $O(\log n)$ : 对数时间 (极快)

#### ⚠ Remark ▾

加法比乘法稍快, 因此很多算法试图用加法替代乘法

## | 2.2 示例: 矩阵乘法

### | 2.2.1 运算速度

若  $A$  为  $a \times b$ ,  $B$  为  $b \times c$ , 则

- 每个输出元素需进行  $b$  次乘法
- 一共  $abc$  次乘法, 因此复杂度为  $O(abc)$

对称  $n \times n$  矩阵乘法为  $O(n^3)$

同样地, Cholesky 或 QR 分解也为  $O(n^3)$  (若矩阵稀疏则可更快)

### | 2.2.2 存储与内存需求

乘法结果需保存  $ab + bc + ac$  个元素

对于对称矩阵, 近似  $3n^2$  个元素

例: 当  $n = 10,000$  且每个元素为 8 字节浮点数时:

$$3 \times 10,000^2 \times 8 / 10^9 = 2.4 \text{ GB}$$

因此大规模矩阵计算既在时间上  $O(n^3)$ , 又在空间上  $O(n^2)$ 。

## | 2.3 多项式、指数与对数时间的意义

- **多项式时间**  $O(n^q)$ : 可接受 (如矩阵分解)
- **指数时间**  $O(b^n)$ : 不可扩展 (如 NP-complete 问题)
- **对数时间**  $O(\log n)$ : 极高效 (如二分查找)

**NP-complete** 问题是无法在多项式时间内求解的难题族。

#### ⚠ Remark ▾

即使复杂度阶相同, 常数项也可能决定算法快慢:

- 例:  $n^2$  操作可能比  $1000(n \log n + n)$  更快
- 因为第二个算法中常数  $c = 1000$  较大, 且存在额外的低阶计算

## | 2.4 FLOPs (Floating Point Operations)

- “flops” 可指:
  1. 浮点运算次数
  2. 每秒浮点运算数 (floating point operations per second)
- 常用于衡量算法计算量或计算机性能

---

## 3 Notation and Dimensions

---

### 3.1 Notations

符号	含义
$A$	矩阵 (matrix)
$x$	向量 (vector)
$x_i$	向量第 $i$ 个元素
$A_{ij}$	矩阵第 $i$ 行、第 $j$ 列元素
$A_{\cdot j}$	第 $j$ 列
$A_{i \cdot}$	第 $i$ 行

默认：

- 向量  $x$  为列向量
- $x^\top$  为行向量

---

### 3.2 Dimensions

检查矩阵 **是否可相乘 (conformable)** 是编程中常见的防错手段：

- 对于  $A + B$ ，要求维度完全相同
- 对于  $AB$ ，要求  $A$  的列数 =  $B$  的行数

例：

$$\text{Cov}(Ax) = A\text{Cov}(x)A^\top$$

而非  $A^\top\text{Cov}(x)A$ ，否则维度不匹配。

---

### 3.3 Inner and Outer Products

- **内积 (Inner Product)**

$$\langle x, y \rangle = x^\top y = \sum_i x_i y_i$$

- **外积 (Outer Product)**

$$xy^\top$$

结果为矩阵，包含  $x$  与  $y$  所有元素的乘积组合。

---

## 4 Norms

---

### 4.1 向量范数 (Vector Norm)

$$|x|_p = \left( \sum_i |x_i|^p \right)^{1/p}$$

常用的欧几里得范数：

$$|x|_2 = \sqrt{x^\top x}$$

## 4.2 矩阵范数 (Matrix Norms)

- Frobenius 范数

$$\|A\|_F = \sqrt{\sum_{i,j} a_{ij}^2}$$

- 诱导范数 (Induced Norm)

$$\|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

对于 2-norm：

$$\|A\|_2 = \sup_{|x|_2=1} \|Ax\|_2$$

等价于矩阵 **最大奇异值** (largest singular value)

## 4.3 范数的几何意义与性质

- $\|A\|$  衡量矩阵对向量的最大拉伸倍数
- 基本性质：
  - 次乘法性： $\|AB\| \leq \|A\| \|B\|$
  - 三角不等式： $\|A + B\| \leq \|A\| + \|B\|$
- 向量单位化 (normalized)：

$$\tilde{x} = \frac{x}{\|x\|}$$

- 向量夹角：

$$\theta = \cos^{-1} \left( \frac{\langle x, y \rangle}{\sqrt{\langle x, x \rangle \langle y, y \rangle}} \right)$$

## 5 Orthogonality

### 5.1 向量正交 (Orthogonal Vectors)

若  $x^\top y = 0$ ，则称  $x$  与  $y$  **正交** (orthogonal)，记作  $x \perp y$ 。

### 5.2 正交矩阵 (Orthogonal Matrix)

一个方阵  $A$  称为正交矩阵若：

$$A^T A = I \quad \text{或等价地} \quad A^{-1} = A^T.$$

正交矩阵的主要性质：

1. 列与行互相正交且单位化

$$A_{\cdot i}^T A_{\cdot j} = 0 \quad (\text{当 } i \neq j) \quad \text{且} \quad |A_{\cdot i}| = 1.$$

2. 满秩 (full rank)

因为列向量线性无关。

3. 行列式 (determinant)

$$\det(A) = \pm 1.$$

4. 乘积封闭性

若  $A$  与  $B$  均为正交矩阵，则

$$(AB)^T AB = B^T A^T AB = B^T B = I$$

因此  $AB$  也是正交的。

## 5.3 置换矩阵 (Permutation Matrices)

- 定义：** 交换矩阵中两行或两列的单位矩阵称为 **基本置换矩阵 (elementary permutation matrix)**。这类矩阵是正交的，且  $\det(P) = -1$ 。
- 性质：**
  - 行置换：** 左乘  $P$ ，即  $PA$ 。
  - 列置换：** 右乘  $P$ ，即  $AP$ 。
  - 计算上不必显式乘法，只需调整索引即可（节省计算与内存）。

## 6 Vector and Matrix Properties

### 6.1 基本代数性质

性质	表达式
非交换性	$AB \neq BA$
可交换性（加法）	$A + B = B + A$
结合律	$A(BC) = (AB)C$

### 6.2 Python 操作回顾

	Python
1	<code>A + B</code> # 矩阵加法
2	<code>np.matmul(A, B)</code> # 矩阵乘法
3	<code>A @ B</code> # 推荐的简写
4	<code>A * B</code> # Hadamard（逐元素）乘法
5	<code>A.dot(B)</code> # 不推荐（NumPy 已弱化）

## 7 Trace and Determinant

## 7.1 矩阵迹 (Trace)

定义：

$$\text{tr}(A) = \sum_i A_{ii}$$

常见性质：

- $\text{tr}(A + B) = \text{tr}(A) + \text{tr}(B)$
- $\text{tr}(A^\top) = \text{tr}(A)$
- 循环不变性：**

$$\text{tr}(ABC) = \text{tr}(BCA) = \text{tr}(CAB)$$

应用意义：

- 可以通过变换矩阵顺序减少计算量。
- 可将标量二次型转化为矩阵迹形式：

$$x^\top Ax = \text{tr}(x^\top Ax) = \text{tr}(xx^\top A)$$

## 7.2 矩阵行列式 (Determinant)

性质：

- $|AB| = |A||B|$
- $|A^{-1}| = 1/|A|$
- $|A| = |A^\top|$  (由 QR 分解与三角矩阵性质可知)

## 8 Transposes and Inverses

### 8.1 转置与逆的关系

若  $A$  可逆，则：

$$(A^{-1})^\top = (A^\top)^{-1}$$

🔗 Proof ▾

$$A^\top (A^{-1})^\top = (A^{-1}A)^\top = I^\top = I$$

### 8.2 矩阵乘法的逆

对于可逆矩阵  $A, B$ ：

$$(AB)^{-1} = B^{-1}A^{-1}$$

验证：

$$B^{-1}A^{-1}AB = I$$

## 8.3 Other Matrix Products

### 8.3.1 Hadamard Product (逐元素乘法)

$$(A \circ B)_{ij} = A_{ij}B_{ij}$$

Python 中对应 `A * B`。

**Challenge:** 如何不计算 `A @ B` 直接求  $\text{tr}(AB)$ ?

解答：

$$\text{tr}(AB) = \sum_{i,j} A_{ij}B_{ji}$$

在 Python 中：

```
Python
1 np.sum(A * B.T)
```

### 8.3.2 Kronecker Product (克罗内克积)

令矩阵  $\mathbf{A} = (a_{i,j}) \in \mathbb{R}^{m \times n}$ ,  $\mathbf{B} = (b_{i,j}) \in \mathbb{R}^{m \times n}$

则**克罗内积**为：

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \cdots & a_{1,n}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \cdots & a_{2,n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}\mathbf{B} & a_{m,2}\mathbf{B} & \cdots & a_{m,n}\mathbf{B} \end{bmatrix}$$

性质：

1.  $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$
2. 计算复杂度显著降低：
  - 直接求逆： $O((nm)^3)$
  - 利用结构： $O(n^3 + m^3)$

## 9 Matrix Decompositions

矩阵分解是将一个复杂矩阵  $A$  表示为若干更“简单”的矩阵乘积：

$$A = BC \quad \text{或} \quad A = BCD$$

### 9.1 常见“简化”形式

- 减少维度：部分行或列
- 特殊结构：对角矩阵、上/下三角矩阵、稀疏矩阵
- 正交矩阵：数值稳定性高

---

## 9.2 优势

- 揭示矩阵内部结构（如方向、方差、相关性）
- 简化计算，例如：
  - 线性系统求解
  - 回归估计
  - 特征值/奇异值分解

---

## 10 Linear Independence, Rank, and Basis

### 10.1 线性无关与秩

一组向量  $v_1, \dots, v_n$  是线性无关的，当：

$$\sum_i c_i v_i = 0 \Rightarrow c_i = 0 \quad \forall i$$

矩阵的秩 (rank) = 线性无关的列（或行）的数量。  
最大值为  $\min(\text{行数}, \text{列数})$ 。

---

### 10.2 基向量与张成空间

任意一组线性无关向量  $v_i$  构成的线性组合：

$$y = \sum_i c_i v_i$$

定义了该组向量张成的空间 (span)。  
 $v_i$  即为该空间的基 (basis)。

若  $v_i$  是正交归一化的，则：

$$c_i = \langle y, v_i \rangle = v_i^\top y$$

---

### 10.3 回归模型中的秩与空间

设计矩阵  $X$  的列为协变量向量：

- 若  $q = \text{rank}(X) < p$   
则有  $p - q$  列是其他列的线性组合
- 空间维数为  $q$ ，对应  $q$  个基向量
- 在回归中：
  - 若  $n = p = q \rightarrow$  唯一且精确解（无残差）
  - 若  $n < p \rightarrow$  欠定系统（多解）
  - 若  $n > p \rightarrow$  过定系统（最小二乘求近似解）



回归的几何解释：  
Y 被投影到由 X 的列张成的空间上。

## 10.4 Invertibility, Singularity, and Positive Definiteness

### 10.4.1 特征分解 (Eigendecomposition)

对于可对角化矩阵：

$$A = \Gamma \Lambda \Gamma^{-1}$$

若 A 对称：

$$A = \Gamma \Lambda \Gamma^\top$$

其中：

- $\Gamma$ ：特征向量矩阵
- $\Lambda$ ：对角特征值矩阵
- 非零特征值数量 = 矩阵秩

### 10.4.2 矩阵分类与性质

性质	定义	推论
非奇异 (nonsingular)	$A^{-1}$ 存在	秩满 (full rank)
正定 (positive definite)	$x^\top A x > 0$	所有 $\lambda_i > 0$
半正定 (positive semi-definite)	$x^\top A x \geq 0$	一部分 $\lambda_i = 0$
奇异 (singular)	不可逆	存在 $\lambda_i = 0$

若 A 为协方差矩阵  $\text{Cov}(y)$ ：

$$x^\top A x = \text{Var}(x^\top y) \geq 0$$

因此协方差矩阵必为半正定矩阵。

### 10.4.3 结论总结

代数性质	统计解释
$A \text{ p.d.} \Leftrightarrow x^\top A x > 0 \Leftrightarrow \lambda_i > 0$	协方差矩阵，方差均为正
$A \text{ p.s.d.} \Leftrightarrow x^\top A x \geq 0$ 且部分 $\lambda_i = 0$	受约束协方差矩阵，部分方向方差为 0
$\lambda_i = 0$	某线性组合无方差信息（不识别）

## 11 Eigen-decomposition Interpretation

生成随机向量  $y$  的思路:

$$y = \Gamma \Lambda^{1/2} z, \quad \text{where } \text{Cov}(z) = I$$

解释：

- $\Gamma_i$  为方差主方向 (特征向量)
- $\sqrt{\lambda_i}$  为对应标准差
- $z_i$  为该方向上的标准化系数

### 11.1.1 问题与解释

1. 若  $\lambda_i = 0$ ：  
该方向上方差为零， $y$  在该方向无变化。
2. 若将极小  $\lambda_i$  设为 0：  
去除对应的低方差分量，相当于“平滑”或“降维”。
3. 若  $(\Lambda^{-1})_{ii}$  很大：  
对应方向方差很小（高精度）。  
若很小  $\rightarrow$  对应方向方差极大（低精度）。

## 12 Generalized (Pseudo) Inverses (optional)

若  $A$  不可逆, 仍可定义广义逆  $A^-$  使:

$$AA^{-}A = A$$

**Moore–Penrose 伪逆 (pseudo-inverse):** 唯一满足附加约束的广义逆, 记为  $A^+$ 。  
其最小化解为:

$$x = A^+b$$

即满足  $Ax = b$  且  $|x|$  最小。

## 12.1 由特征分解求伪逆

$$\mathbf{A}^+ = \mathbf{\Gamma} \mathbf{\Lambda}^+ \mathbf{\Gamma}^\top, \quad \mathbf{\Lambda}_{ii}^+ = \begin{cases} 1/\lambda_i, & \lambda_i > 0 \\ 0, & \lambda_i = 0 \end{cases}$$

统计解释：

若  $\lambda_i = 0$ , 对应方向无信息 (无限方差), 伪逆将该方向方差设为 0。

## 12.2 示例：AR(1) 与 AR(2) 精度矩阵

- 一阶自回归 (AR(1)) 精度矩阵:

$$(1 \quad -1 \quad 0 \quad 0 \quad 0 \quad -1 \quad 2 \quad -1 \quad 0 \quad 0 \quad -1 \quad 2 \quad -1 \quad 0 \quad 0 \quad 0 \quad -1 \quad 2 \quad -1 \quad 0 \quad 0 \quad 0 \quad -1 \quad 1)$$

零特征值对应常数向量（均值不可识别）。

- 二阶自回归 (AR(2)) 矩阵中有两个零特征值, 对应:
  - 总不确定性 (constant component)
  - 线性趋势不确定性 (linear component)

伪逆操作即为：

将  $\lambda_i = 0$  的分量方差设为零，相当于约束  $\sum y_i = 0$  或去除线性趋势。

---

## 13 Regression 中的 Matrices

---

在线性回归中：

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

- $X^T X$ :
  - 对称
  - 半正定 (n.n.d.)
  - 若列向量线性无关  $\rightarrow$  正定 (p.d.)

---

### 13.1.1 Hat Matrix (投影矩阵)

$$H = X(X^T X)^{-1} X^T$$

性质：

1. 投影： $\hat{Y} = HY$
2. 幂等性： $HH = H$
3. 奇异性： $H$  不可逆（因为其秩  $< n$ ）
4. 唯一情况  $H$  可逆： $H = I$ （仅当  $X$  为满秩方阵且  $n = p$ ）

几何解释：

$H$  将  $Y$  投影到由  $X$  列张成的空间；

投影后再投影不改变结果，因此  $HH = H$ 。

---

## 14 Storing matrices

矩阵在内存中的存储方式包括**列优先 (column-major)** 和 **行优先 (row-major)**。

- **列优先**语言（R、Fortran）中，相邻列元素在内存中连续，所以访问同一列或整列更快
- **行优先**语言（Python、C）中，相邻行元素在内存中连续，所以访问同一行或整行更快

某些优化的矩阵分解算法会**将输出结果直接覆盖输入矩阵**以节省内存并提升速度（如某些 in-place factorizations）。

---

## 15 Algorithms

🔗 Logic ▾

优秀算法能将效率提升几个数量级，现代计算速度的主要提升往往来自算法进步而非硬件本身。

### 15.1 一个例子: 向量乘法的两种实现

对于向量乘法  $b = Ax$ ，可使用两种伪代码实现：

### 15.1.1 按行进行内积 (row-wise)

$$b_i = \sum_{j=1}^m a_{ij}x_j$$

```
1 # initialize b[1:n] = 0
2 for(i = 1:n){
3     for(j = 1:m){
4         b_i = b_i + a_{ij} * x_j
5     }
6 }
```

### 15.1.2 按列进行线性组合 (column-wise)

$$b = \sum_{j=1}^m x_j A_{.j}$$

```
1 # initialize b[1:n] = 0
2 for(j = 1:m){
3     for(i = 1:n){
4         b_i = b_i + a_{ij} * x_j
5     }
6 }
```

两种方法的计算量相同，但：

- 方法 1 按行访问  $A$ ，适合 **row-major** (Python、C)
- 方法 2 按列访问  $A$ ，适合 **column-major** (R、Fortran)

## 15.2 General computational issues

许多前面讨论过的计算机浮点限制在矩阵计算中同样适用：

- 小数的舍入误差
- catastrophic cancellation
- 分母接近 0 的不稳定情况

优秀的线性代数库（如 LAPACK）会内部处理这些数值稳定性问题。

## 16 Ill-conditioned problems

Logic ▾

对于矩阵 ill-conditioned problem 的详细论述, 见 [DDA3005 Lecture 6](#)

### 16.1 Basics

若输入的微小扰动导致输出显著变化，则问题是**病态 (ill-conditioned)** 的，一种衡量是否病态的方式为 **condition number**（条件数）

### 16.2 例子：矩阵求解对扰动的敏感性

设

$$A = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}, \quad b = (32, 23, 33, 31)^T$$

求解  $Ax = b$  得到  $x = (1, 1, 1, 1)$

若扰动为

$$\delta b = (0.1, -0.1, 0.1, -0.1)$$

新的解变为

$$x + \delta x = (9.2, -12.6, 4.5, -1.1)$$

可见非常敏感

## 16.3 条件数的公式

对于欧几里得范数  $L_2$  (实际上可以选择任何范数), 定义 condition number 为:

$$\text{cond}(A) = \|A\| \|A^{-1}\|$$

并且 (仅对于  $L_2$  norm) 有:

$$\text{cond}(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$$

其中  $\lambda_{\max}$  和  $\lambda_{\min}$  为最大和最小特征值的绝对值

并且有近似不等式:

$$\frac{\|\delta x\|}{\|x\|} \leq \text{cond}(A) \frac{\|\delta b\|}{\|b\|}$$

解释:

若  $\text{cond}(A) = 10^8$ , 我们在双精度  $10^{-16}$  的基础上会丢失 8 位有效数字

## 16.4 代码验证 (略作整理)



Python

```
1 e = np.linalg.eig(A)
2 evals = e[0]
3 print(evals)
4 # [3.02886853e+01 3.85805746e+00 1.01500484e-02 8.43107150e-01]
5
6 ## relative perturbation in x much bigger than in b
7 norm2(delta_x) / norm2(x)
8 # np.float64(8.19847546803699)
9
10 norm2(delta_b) / norm2(b)
11 # np.float64(0.0033319453118976702)
12
13 ## ratio of relative perturbations
14 (norm2(delta_x) / norm2(x)) / (norm2(delta_b) / norm2(b))
15 # np.float64(2460.567236431514)
16
17 ## ratio of largest and smallest magnitude eigenvalues
18 ## confusingly evals[2] is the smallest, not evals[3]
19 (evals[0]/evals[2])
20 # np.float64(2984.092701676269)
```

## 16.5 Improving conditioning

### Logic ▾

统计问题中病态常由自变量共线性引起。改善方式通常是**重新建模**或对变量**中心化、标准化**。

核心思想：

- **避免输入数值量级差距过大**
- 尽量让变量接近数量级 1

### 16.5.1 以二次回归为例

原始变量  $t = 1990, \dots, 2010$  导致  $X = (1, t, t^2)$  中：

- 1 的量级  $\approx 1$
- $t$  的量级  $\approx 2000$
- $t^2$  的量级  $\approx 4 \times 10^6$

因此  $X^\top X$  条件数巨大，OLS 不稳定。

### 16.6 步骤 1：中心化

定义

$$t_2 = t - 2000$$

### 16.7 步骤 2：进一步缩放

$$t_3 = \frac{t_2}{10}$$

通过中心化和缩放：

- 特征值范围变窄
- 条件数明显下降
- 回归系数更精确

### 16.8 结果总结

- 简单的中心化和标准化往往足以改善病态
- 应始终关注变量的量级
- 最好使每列变量都在数量级  $O(1)$

## 17 Matrix factorizations (decompositions) and solving systems of linear equations

在数值计算中，求解线性系统

$$Ax = b$$

从不通过显式求逆  $A^{-1}$  再相乘。实际做法是使用各种矩阵分解，例如 LU 分解、Cholesky 分解，或利用迭代方法在可接受误差范围内减少计算量。

下表总结了常见分解：

Name	Representation	Restrictions	Properties	Uses
LU	$A_{nn} = L_{nn}U_{nn}$	$A$ 一般为方阵	$L$ 下三角, $U$ 上三角	solving, inversion
QR	$A_{nm} = Q_{nn}R_{nm}$ 或 skinny 形式 $A_{nm} = Q_{nm}R_{mm}$	—	$Q$ 正交, $R$ 上三角	regression
Cholesky	$A_{nn} = U_{nn}^\top U_{nn}$	$A$ positive (semi-) definite	$U$ 上三角	covariance, normals, solving, inversion
Eigen	$A_{nn} = \Gamma \Lambda \Gamma^\top$	$A$ 对称*	$\Gamma$ 正交, $\Lambda$ (非负**) 对角	PCA
SVD	$A_{nm} = UDV^\top$ 或 skinny 形式	—	$U, V$ 正交, $D$ 非负对角	ML, topic models

\* 也存在非对称矩阵的 eigen 形式

\*\* 对于 p.d. 或 p.s.d.  $A_{nn}$

## 18 Triangular systems

### Logic ▾

关于 triangular system 的详细论述, 见 [DDA3005 Lecture 7](#)

若  $A$  为上三角矩阵, 则可直接 backsolve:

1.  $x_n = b_n / A_{nn}$
2. 对  $k < n$ ,

$$x_k = \frac{b_k - \sum_{j=k+1}^n A_{kj}x_j}{A_{kk}}$$

3. 向上重复直到求得所有分量

时间复杂度约为  $O(n^2)$ 。下三角系统同理。

SciPy 使用 `linalg.solve_triangular` 来求解:

```

Python
1  import scipy as sp
2  rng = np.random.default_rng(seed=1)
3  n = 20
4  X = rng.normal(size=(n,n))
5  A = X.T @ X
6
7  b = rng.normal(size=n)
8  L = np.linalg.cholesky(A)
9  U = L.T
10
11  out1 = sp.linalg.solve_triangular(L, b, lower=True)
12  out2 = np.linalg.inv(L) @ b
13  np.allclose(out1, out2)
14  # True

```

**关键点:** 矩阵求逆并不会真的产生  $U^{-1}$ ; 计算机会用更快更稳定的 **backsolve** 来完成该运算。

计时对比显示直接求逆速度更慢且易失稳。



Python

```
1 import time
2
3 rng = np.random.default_rng(seed=1)
4 n = 5000
5 X = rng.normal(size = (n,n))
6
7 ## R has the `crossprod` function, which would be more efficient
8 ## than having to transpose, but numpy doesn't seem to have an equivalent.
9 A = X.T @ X
10 b = rng.normal(size = n)
11 L = np.linalg.cholesky(A) # L is lower-triangular
12
13 t0 = time.time()
14 out1 = sp.linalg.solve_triangular(L, b, lower=True)
15 time.time() - t0
16 # 0.054033756256103516
17
18 t0 = time.time()
19 out2 = np.linalg.inv(L) @ b
20 time.time() - t0
21 # 8.393277883529663
```

## 19 Gaussian elimination (LU decomposition)

🔗 Logic ▾

关于 LU decomposition 的详细论述, 见 [DDA3005 Lecture 8](#)

### 19.1 Gaussian elimination

Gaussian elimination 的前向消元阶段将  $A$  转为上三角  $U$ , 形式为:

$$L_{n-1} \cdots L_1 A x = U x = L_{n-1} \cdots L_1 b = b^*$$

其中每个  $L_j$  为一个简单的行操作矩阵。若仅求解系统而非显式分解矩阵, 可不显式构造全部  $L_j$ , 但数值库会为我们自动完成这一过程。

- LU 分解的复杂度为  $O(n^3)$
- numpy 内部调用 LAPACK 的 `*gesv` (带部分选主元的 LU)
- SciPy 可直接 `scipy.linalg.lu()` 得到  $P, L, U$

### 19.2 Partial pivoting

🔗 Logic ▾

关于 pivoting 的详细论述, 见 [DDA3005 Lecture 9](#)

为了避免除以非常小的值导致数值不稳定, LU 会使用 partial pivoting:

- 在每列选择绝对值最大的元素作为 pivot
- 交换行, 使得计算尽可能稳定
- 这些交换用置换矩阵  $P$  表示

因此 LU 实际满足:



$$PA = LU$$

## 19.3 Determinant from LU

LU 分解还可以用于计算行列式

由于  $|PA| = |P||A| = |L||U| = |U|$ , 有

$$|A| = \frac{|U|}{|P|}$$

由于每个 permutation matrix  $P$  的行列式为  $-1$ , 因此计算时仅需统计交换行的奇偶数

## 19.4 When would we explicitly invert a matrix?

只有在 **最终输出需要逆矩阵本身** (如标准误估计) 时才显式求逆。

若只是想计算  $A^{-1}B$ , 则应使用:



Python

```
1 np.linalg.solve(A, B)
```

因为:

- 求逆成本比分解 + 回代 更高
- 求逆数值更不稳定
- numpy solve 内部使用 LU, 不会构造逆矩阵

绝不应写:



Python

```
1 np.linalg.inv(A) @ B # 不推荐
```

## 20 Cholesky decomposition

🔗 Logic ▾

关于 Cholesky decomposition 的详细论述, 见 [DDA3005 Lecture 9](#)

若  $A$  为正定, 则 Cholesky 分解:

$$A = U^T U$$

其中  $U$  上三角且对角元素为正。

构造  $U$  的算法:

1.  $U_{11} = \sqrt{A_{11}}$
2.  $U_{1j} = A_{1j}/U_{11}$
3. 对一般  $i$ :
  - $U_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} U_{ki}^2}$
  - 若  $i < n$ , 则对于  $j = i + 1, \dots, n$

$$U_{ij} = \frac{A_{ij} - \sum_{k=1}^{i-1} U_{ki} U_{kj}}{U_{ii}}$$

## 20.1 Solving with Cholesky

可写为两次 triangular solve:

```

Python
1 U = sp.linalg.cholesky(A)
2 x = sp.linalg.solve_triangular(
3     U,
4     sp.linalg.solve_triangular(U, b, lower=False, trans='T'),
5     lower=False)

```

或使用 SciPy 的封装:

```

Python
1 U, lower = sp.linalg.cho_factor(A)
2 x = sp.linalg.cho_solve((U, lower), b)

```

## 20.2 Advantages over LU

- 相同  $O(n^3)$  复杂度, 但系数为一半
- 仅需存储  $(n^2 + n)/2$  个数
- 对称正定问题更加稳定、快速

## 20.3 Use case: sampling from multivariate normals

```

Python
1 L = sp.linalg.cholesky(A, lower=True)
2 y = L @ rng.normal(size=n)

```

主要计算成本在构造 Cholesky, 不在矩阵乘法。

## 20.4 Numerical issues

即使矩阵理论上正定, 高维相关矩阵可能会:

- 因舍入误差导致  $U_{ii}^2$  出现负数
- small eigenvalues  $\rightarrow$  不稳定
- 特别在 large correlation、large dimension 时常见

```

Python
1 rng = np.random.default_rng(seed=1)
2 locs = rng.uniform(size = 100)
3 rho = .1
4 dists = np.abs(locs[:, np.newaxis] - locs)
5 C = np.exp(-dists**2/rho**2)
6 e = np.linalg.eig(C)
7 np.sort(e[0])[:, -1][96:100]
8 # array([-3.58440747e-16+7.57451730e-17j, -3.58440747e-16-7.57451730e-17j, -4.98458053e-
9     16+1.77854836e-16j, -4.98458053e-16-1.77854836e-16j])
10
11 try:
12     L = np.linalg.cholesky(C)

```

```

12 except Exception as error:
13     print(error)
14 Matrix is not positive definite
15
16 vals = np.abs(e[0])
17 np.max(vals)/np.min(vals)
18 # np.float64(5.597477620755469e+17)

```

解决办法:

- 使用 eigen/SVD, 将极小的 eigenvalues 截断为 0 (pseudo-inverse)
- R 中有 pivoted Cholesky: `chol(C, pivot=TRUE)`
- Python 默认不提供 pivoted Cholesky

## 21 QR decomposition

🔗 Logic ▾

关于 linear regression 问题的详细论述, 见 [DDA3005 Lecture 10](#)

关于 QR decomposition 的详细论述, 见 [DDA3005 Lecture 11](#)

关于 Householder QR decomposition 的详细论述, 见 [DDA3005 Lecture 12](#)

关于 Givens 和 Gram Schmidt QR decomposition 的详细论述, 见 [DDA3005 Lecture 13](#)

### 21.1 Introduction

- QR decomposition 适用于任意矩阵

$$X = QR$$

其中  $Q$  为正交矩阵,  $R$  为上三角矩阵

- 对于非方阵  $X$  ( $n \times p$ , 且  $n > p$ ):
  - $R$  的前  $p$  行构成一个上三角矩阵  $R_1$
  - $Q$  的前  $p$  列构成  $Q_1$
  - skinny QR:

$$X = Q_1 R_1$$

- 为保证唯一性, 可要求  $R$  的对角元素非负
  - 此时有

$$X^\top X = R^\top R$$

表明  $R$  等于 Cholesky 分解的上三角因子

- 三种常见 QR 计算方法
  - Householder reflections
  - Givens rotations
  - Gram-Schmidt orthogonalization (后文分别说明)
- 对  $n \times n$  的  $X$ :
  - Householder QR 需要约  $2n^3/3$  flops
  - 比 LU 或 Cholesky 慢
- pseudo-inverse 的构造:

$$X^+ = [R_1^{-1} \ 0] Q^\top$$

- 若矩阵不满秩, 可使用带 pivoting 的 QR

- 在  $R_1$  的对角线中会出现额外的零

## 21.2 Regression and the QR

- 回归模型

$$Y = X\beta + \epsilon$$

对应估计

$$\hat{\beta} = (X^\top X)^{-1} X^\top Y$$

- 若使用 skinny QR:

$$X = QR, \quad R^\top \text{ 可逆}$$

则正常方程变为

$$R\beta = Q^\top Y$$

因此求  $\hat{\beta}$  仅需一次 backsolve

- 标准回归对象 (hat matrix, SSE, residuals) 都可用  $Q$  和  $R$  表示  
因为  $Q$  正交、 $R$  上三角, 计算稳定且方便
- 为什么不用 Cholesky 解  $X^\top X$ ?
  - $X^\top X$  的条件数是  $X$  的平方
  - QR 直接分解  $X$ , 条件数好得多
  - 当预测变量高度共线时, QR 更可靠
- 不同最小二乘算法的复杂度
  - Cholesky:

$$np^2 + \frac{1}{3}p^3$$

- Sweeping:

$$np^2 + p^3$$

- QR (Householder):

$$2np^2 - \frac{2}{3}p^3$$

- Modified Gram–Schmidt:

$$2np^2$$

- 若  $n \gg p$ : Cholesky 和 sweeping 更快
- Modified Gram–Schmidt 最稳定, Sweeping 最不稳定
- 回归通常对  $p$  不大, 运算成本一般不是瓶颈

## 21.3 Regression and the QR in Python and R

- Python:



Python

```
1 Q, R = np.linalg.qr(X)
```

- statsmodels 中的 OLS 默认使用 QR
- Python、R 默认都提供 skinny QR:
  - $Q$  的前  $p$  列为列空间基底
  - 剩余列为零空间的正交基底
  - 对应回归中“模型空间”和“残差空间”的分解
- R 的 `qr()` 使用底层 Fortran 库
  - 上三角矩阵  $R$  存储在 `\$qr` 的上三角
  - 下三角 + `\$aux` 用来重建  $Q$
  - `qr.qy()` 可用于  $Qy$
  - `qr.qty()` 可用于  $Q^T y$
- 提取  $Q, R$ :

```

R
1 X.qr = qr(X)
2 Q = qr.Q(X.qr)
3 R = qr.R(X.qr)

```

- R 还提供 QR 基础上的回归计算函数:
  - `qr.resid()`
  - `qr.fitted()`
  - `qr.coef()`

## 21.4 Computing the QR decomposition

QR 的三大方法具备共通特征：都通过一系列正交变换将  $X$  化为上三角。

- 正交变换包括
  - reflections (Householder)
  - rotations (Givens)
- 正交矩阵性质
  - determinant 为  $\pm 1$
  - 运算稳定、不改变向量长度

## 21.5 QR Method 1: Reflections (Householder)

- Householder reflection 基本思想:
  - 若  $x = c_1 u + c_2 v$
  - 则反射为

$$\tilde{x} = -c_1 u + c_2 v$$

- Householder 矩阵

$$Q = I - 2uu^T$$

性质

- $Qu = -u$
- 若  $u^T v = 0$ , 则  $Qv = v$
- $Q$  对称且正交
- QR 的构造

$$R = Q_p \cdots Q_1 X, \quad Q = (Q_p \cdots Q_1)^\top$$

- 第一步通过反射将第一列化为

$$(|x|, 0, \dots, 0)$$

后续 Householder 逐步将下三角消为 0

- 在回归情境中：
  - 对  $X$  和  $Y$  依次应用  $Q_j$
  - 得到  $R$  和  $Q^\top Y$
  - 回代求解  $R\beta = Q^\top Y$
- $\hat{\beta}$  的协方差：

$$\text{Cov}(\hat{\beta}) \propto (X^\top X)^{-1} = R^{-1} R^{-\top}$$

- $Q^\top Y$  可分为
  - 前  $p$  个组成  $z^{(1)}$
  - 后  $n - p$  个组成  $z^{(2)}$
  - $\text{SSR} = \|z^{(1)}\|^2$
  - $\text{SSE} = \|z^{(2)}\|^2$
- 关于最后一步  $Q_p$  的说明：
  - 主要用于符号选择避免数值抵消
  - 若  $n = p$ ,  $Q_p$  不再用于消元，只用于符号调整
  - 若  $n > p$ ,  $Q_p$  仍用于产生 skinny QR 需要的零行

## 21.6 QR Method 2: Rotations (Givens)

- Givens rotation：在二维子空间作旋转以消除某个分量
- 形式：

$$\tilde{x} = Qx$$

且  $Q$  正交但非对称

- QR 过程：通过一系列 Givens rotations 消除下三角元素
- 结果形式：

$$R = Q_{pm} \cdots Q_{12} X, \quad Q = (Q_{pm} \cdots Q_{12})^\top$$

- 适合处理稀疏矩阵，因为只影响局部结构
- 若不 carefully implement，运算量可能比 Householder 更大

## 21.7 QR Method 3: Gram–Schmidt Orthogonalization

- 基本思想：从原矩阵列向量构造一组正交基
- Modified Gram–Schmidt 更稳定
- Algorithm 概要
  - $\tilde{x}_1 = x_1 / \|x_1\|$
  - 对  $k \geq 2$ 
    - 从  $x_k$  中减去其在  $\tilde{x}_1$  上的投影
    - 归一化得到  $\tilde{x}_k$
  - 不断对剩余向量进行正交化与归一化
- $Q$  的列为正交基
- $R = Q^\top X$

- 解释为：将  $X$  的列回归到  $Q$  上
- 上三角结构自然产生

## 21.8 The tall-skinny QR

适用于  $n \gg p$  的超大矩阵，能分布式或流式计算。

- 将  $X$  分块

$$X = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} Q_0 R_0 \\ Q_1 R_1 \\ Q_2 R_2 \\ Q_3 R_3 \end{pmatrix}$$

- 对各分块的  $R_i$  继续做 QR 合并

$$\begin{pmatrix} R_0 \\ R_1 \end{pmatrix} = Q_{01} R_{01}, \quad \begin{pmatrix} R_2 \\ R_3 \end{pmatrix} = Q_{23} R_{23}$$

- 最终得到

$$X = QR$$

- 优点
  - 可用 map-reduce 并行计算
  - 可用于内存无法放下整矩阵的场景（streaming QR）

## 22 Determinants

- 任意分解中若包含上三角矩阵  $R$  和正交矩阵  $Q$ ：

$$|A| = |QR| = |Q||R| = \pm |R|$$

- 对  $A^\top A$ ：

$$|A^\top A| = |R_1^\top R_1| = |R_1|^2$$

- Python 中计算  $\log \det$ ：

```
Python
1 Q, R = qr(A)
2 magn = np.sum(np.log(np.abs(np.diag(R))))
```

- 其他选择：
  - SVD:  $|A| = \prod \sigma_i$
  - eigenvalues:  $|A| = \prod \lambda_i$
  - Cholesky: 正定矩阵中最稳定

### 22.1 Computation notes

- 直接乘对角线会 overflow/underflow  
→ 始终使用 log scale
- 负值对角：取绝对值并记录符号
- `np.linalg.slogdet()` 是推荐方法

- `np.linalg.det()` 不推荐（更不稳定）

## 23 Eigendecomposition

### Logic ▾

关于 Eigendecomposition 的详细论述, 见 [DDA3005 Lecture 15](#)

关于 Power method 的详细论述, 见 [DDA3005 Lecture 16](#)

关于 Inverse iteration, Rayleigh quotient iteration, QR iteration 的详细论述, 见 [DDA3005 Lecture 17](#)

关于对 QR iteration 的改进的详细论述, 见 [DDA3005 Lecture 18](#)

### 23.1 Eigendecomposition 的原理

- Eigendecomposition（谱分解）在算法收敛分析、PCA 等应用中非常重要
  - 将方差结构分解为一组正交方向（eigenvectors）与对应的变异量（eigenvalues）
- 对实对称矩阵  $A$ :

$$A = \Gamma \Lambda \Gamma^\top$$

- $\Gamma$ : 正交矩阵（列为特征向量）
- $\Lambda$ : 对角矩阵（特征值，按降序排列）
- 经典定义:

$$Ax = \lambda x$$

- 逆与平方根:

$$A^{-1} = \Gamma \Lambda^{-1} \Gamma^\top, \quad A^{1/2} = \Gamma \Lambda^{1/2}$$

- Spectral radius:

$$\rho(A) = \max |\lambda_i|$$

对于对称矩阵, spectral radius 即为 L2 norm:

$$\rho(A) = \|A\|_2$$

会在许多迭代算法的 convergence rate 中出现

### 23.2 Eigendecomposition 的 Computation

- 常用方法: QR algorithm
  - 第一步: 将  $A$  化为 upper Hessenberg（对称矩阵化为 tridiagonal）
  - 通过 Householder 或 Givens 完成初步化简
  - 之后反复做 QR 分解并 shift, 对角元素收敛到 eigenvalues
  - eigenvectors 由所有变换矩阵相乘得到
- 只需最大（或前几个最大）特征值时: 使用 power method
  - 通过反复乘以  $A$  并归一化得到最大特征向量

- $$z^{(k)} = \frac{Az^{(k-1)}}{|Az^{(k-1)}|}$$

- $\lambda_1$  可通过

$$\lambda_1 \approx \frac{(Az)_i}{z_i}$$



- 找第二大 eigenvalue:  
使用

$$B = A - \lambda_1 v v^\top$$

对  $B$  使用相同方法

## 23.3 Singular value decomposition

我们考虑一个  $n \times m$  矩阵  $A$ ，假设  $n \geq m$ 。  
矩阵总能分解为

$$A = U D V^\top$$

- $U$ : 左奇异向量，列正交
- $V$ : 右奇异向量，列正交
- $D$ : 对角矩阵，奇异值非负
- 若  $A$  是方阵且对称正定，SVD 与 eigendecomposition 一致

### 23.3.1 Representations

- Rank- $k$  分解:

$$A = \sum_{j=1}^k D_{jj} u_j v_j^\top$$

每项是 rank-one 外积

- 关系:

$$A^\top A = V D^2 V^\top, \quad A A^\top = U D^2 U^\top$$

因此奇异值的平方等于 eigenvalues

- padding 版本 (扩展为方阵)

$$A = U_{n \times n} D_{n \times m} V_{m \times m}^\top$$

### 23.3.2 Uses

- 确定矩阵秩
- 求 pseudo-inverse:

$$A^+ = V D^+ U^\top$$

- 最佳 rank- $p$  近似: Truncated SVD

$$\tilde{A} = \sum_{j=1}^p D_{jj} u_j v_j^\top$$

根据 Eckart–Minsky–Young 定理，这是 Frobenius norm 下的最优 rank- $p$  逼近

- 在图像压缩、聚类、推荐系统中广泛使用 (如 Netflix Prize)

### 23.3.3 Interpretation as basis transformation

- 作用  $U D V^\top x$  可理解为:
  - $V^\top x$ : 在  $V$  的基底中表达  $x$
  - $D$ : 按奇异值缩放
  - $U$ : 将结果映射到  $U$  的列空间
- 是两个正交基之间的线性变换

## 23.4 Computation

### 23.4.1 Golub–Reinsch algorithm (经典 SVD)

- 通过一系列 Householder 变换将  $A$  化为 upper bidiagonal 矩阵  $A^{(0)}$ 
  - 右乘去掉上三角
  - 左乘去掉下三角
- 之后通过一系列 Givens rotations 把  $A^{(j)}$  迭代逼近对角矩阵  $D$

\$\$\$

$$A^{\{(j+1)\}} R^{\text{top}} A^{\{(j)\}} T$$

- $U$  是所有 pre-multiplication 的积
- $V$  是所有 post-multiplication 的积
- 最后调整符号（保证奇异值非负）并按降序排序

## 23.5 Computation for large tall-skinny matrices

- 若  $X$  是  $n \times p$  且  $n \gg p$  可以先做 QR:

$$X = QR$$

- 然后对较小的  $p \times p$  的  $R$  做 SVD:

$$R = UDV^T$$

- 从而

$$X = QUDV^T = U^*DV^T$$

- 优点
  - 最耗时的部分变成 tall-skinny QR, 可高效并行
  - SVD 部分仅对  $p \times p$  矩阵, 计算量大幅减少
  - 适合大规模数据矩阵 (如文本、图像、高维特征)

## 24 Linear algebra in Python

🔗 Logic ▾

在许多矩阵分解中, 你可以选择只返回部分结果而非全部结果, 从而显著提升计算速度。

🔗 Logic ▾

在线性代数是统计与机器学习算法核心的背景下, **大型问题的效率在很大程度上依赖于是否调用了快速的线性代数后端**。这些后端可能同时利用并行 CPU 或 GPU, 从而带来数量级的加速。

### 24.1 BLAS and LAPACK

Python 和 R 的矩阵运算大多直接调用底层 C 或 Fortran 实现, 并不会经过大量解释层的开销。因此速度可以非常快。这些核心线性代数库主要包括:

- **BLAS** (Basic Linear Algebra Subprograms)
- **LAPACK** (Linear Algebra PACKage)

不同系统可选择不同 BLAS 后端，例如：

- OpenBLAS
- Intel MKL
- AMD ACML
- Apple vecLib（在 Apple Silicon 上利用 AMX 协处理器）

在 Conda 环境中，numpy 通常会被构建为使用 MKL 或 OpenBLAS；pip 安装通常使用 OpenBLAS。

BLAS 例程分为三个层级：

- **Level 1**：向量操作
- **Level 2**：矩阵-向量操作
- **Level 3**：矩阵-矩阵操作（最高效、并行化最强）

LAPACK 在 BLAS 之上构建，提供：

- 特征分解
- 线性系统求解
- 多种矩阵分解（QR、Cholesky、SVD 等）

---

## | 24.2 JAX and PyTorch (and GPUs)

JAX 与 PyTorch 提供了替代性的线性代数后端，能够自动检测 CPU 或 GPU。

- JAX 支持 **just-in-time compilation (JIT)**
- PyTorch 和 JAX 均在 GPU 上具备高度并行的矩阵运算能力

对于连续大规模矩阵运算，例如矩阵乘法，它们在 GPU 上的速度可能是 NumPy 的数十倍。

相关示例可参考 [SCF parallelization tutorial](#) 中的 [CPU/GPU 对比演示](#)

---

## | 25 Exploiting known structure in matrices

如果矩阵具有特定结构，则可以通过使用特定格式获得显著速度与存储收益。例子包括：

- 对称矩阵（仅存储半个三角）
- Banded matrices（只有若干条带非零）
- Block diagonal matrices（分块对角）
- 稀疏矩阵（大部分为零）

在 Python 中，可使用 `scipy.sparse` 模块，它支持：

- 结构化稀疏矩阵（如 diagonal、triangular）
- 非结构化稀疏矩阵（CSR、CSC 等格式）

在 R 中对应的工具包括 `Matrix`、`spam` 和 `bdsmatrix`。

---

### | 25.1 Sparse matrix formats

以下示例展示如何通过 CSR (compressed sparse row) 格式高效存储稀疏矩阵：

```
Python
1 import scipy.sparse as sparse
2 mat = np.array([[0,0,1,0,10],
3                [0,0,0,100,0],
4                [0,0,0,0,0],
5                [1000,0,0,0,0]])
6 mat = sparse.csr_array(mat)
7
8 mat.data      # non-zero entries
9 # array([ 1, 10, 100, 1000])
10
11 mat.indices   # column indices
12 # array([2, 4, 3, 0], dtype=int32)
13
14 mat.indptr    # row pointers
15 # array([0, 2, 3, 3, 4], dtype=int32)
```

CSR 格式包含三个数组：

- **data**：按行排列的非零元素
- **indices**：每个非零元素的列索引
- **indptr**：每一行起始元素的位置指针

#### ⚠ Remark ▾

**indptr[i]** = 第 i 行的非零元素在 **data** 里的开始位置  
**indptr[i+1]** = 第 i 行的非零元素在 **data** 里的结束位置

可根据以上三部分直接恢复矩阵：

```
Python
1 mat2 = sparse.csr_array((mat.data, mat.indices, mat.indptr))
2 mat2.toarray()
3 # array([[ 0,  0,  1,  0, 10],
4 #        [ 0,  0,  0, 100,  0],
5 #        [ 0,  0,  0,  0,  0],
6 #        [1000, 0,  0,  0,  0]])
```

## 25.2 Sparse matrix multiplication

CSR 结构可以将矩阵乘法  $x = Ab$  的计算量降至  $O(k)$ ，其中  $k$  为非零元素个数：

```
1 for(i in 1:nrows(A)){
2     x[i] = 0
3     for(j in rowpointers[i] : rowpointers[i+1]-1){
4         x[i] = x[i] + entries[j] * b[colindices[j]]
5     }
6 }
```

对比 dense 乘法的  $O(n^2)$  耗时，提升显著。

稀疏矩阵 Cholesky 分解时，如果稀疏结构固定，可以预先做 fill-reducing reordering（如 AMD, nested dissection），保持稀疏性。

## | 26 Banded matrices

若矩阵具有带状结构：

- 下带宽  $p$ :  $A_{ij} = 0$  当  $i > j + p$
- 上带宽  $q$ :  $A_{ij} = 0$  当  $j > i + q$

则 LU 分解的成本为  $O(npq)$ 。相比 dense LU 的  $O(n^3)$ ，速度提升巨大。

时间序列模型中（如 MA 模型），协方差矩阵常呈带状结构，因此非常适用。

---

## | 27 Low rank updates (optional)

对于 rank-one 更新：

$$\tilde{A} = A - uv^\top$$

更一般的低秩更新：

$$\tilde{A} = A - UV^\top$$

其中  $U, V \in \mathbb{R}^{n \times m}$ 。

著名结果是 **Sherman–Morrison–Woodbury formula**：

$$\tilde{A}^{-1} = A^{-1} - A^{-1}U(I_m - V^\top A^{-1}U)^{-1}V^\top A^{-1}$$

或者等价形式：

$$\tilde{A} = A + UCV^\top$$

则：

$$\tilde{A}^{-1} = A^{-1} - A^{-1}U(C^{-1} + V^\top A^{-1}U)^{-1}V^\top A^{-1}$$

意义：

- 若  $m$  很小（低秩更新），则可避免重新求整个  $A^{-1}$
- 在贝叶斯计算、Gaussian Markov random fields、精度矩阵更新等领域非常常见