STAT243 Lecture 10.4 Matrix factorizations (decompositions) and solving systems of linear equations

1 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*	Γ 正交, Λ (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}

2 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

```python
import scipy as sp
rng = np.random.default_rng(seed=1)
n = 20
X = rng.normal(size=(n,n))
A = X.T @ X

b = rng.normal(size=n)
L = np.linalg.cholesky(A)
U = L.T

out1 = sp.linalg.solve_triangular(L, b, lower=True)
out2 = np.linalg.inv(L) @ b
np.allclose(out1, out2)
# 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

```python
import time

rng = np.random.default_rng(seed=1)
n = 5000
X = rng.normal(size = (n,n))

## R has the `crossprod` function, which would be more efficient
## than having to transpose, but numpy doesn't seem to have an equivalent.
A = X.T @ X
b = rng.normal(size = n)
L = np.linalg.cholesky(A) # L is lower-triangular

t0 = time.time()
out1 = sp.linalg.solve_triangular(L, b, lower=True)
time.time() - t0
# 0.054033756256103516

t0 = time.time()
out2 = np.linalg.inv(L) @ b
time.time() - t0
# 8.393277883529663
```

3 Gaussian elimination (LU decomposition)

Logic

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

3.1 Gaussian elimination

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

$$L_{n-1}\cdots L_1 Ax = Ux = L_{n-1}\cdots L_1 b = b^*$$

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

3.2 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：

$$PA = LU$$

3.3 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

3.4 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
```python
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
```python
np.linalg.inv(A) @ B    # 不推荐
```

---

4 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：

$$A = U^\top U$$

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 $j = i+1, \ldots, n$

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

4.1 Solving with Cholesky

可写为两次 triangular solve：

Python

```python
U = sp.linalg.cholesky(A)
x = sp.linalg.solve_triangular(
        U,
        sp.linalg.solve_triangular(U, b, lower=False, trans='T'),
        lower=False)
```

或使用 SciPy 的封装：

Python

```python
U, lower = sp.linalg.cho_factor(A)
x = sp.linalg.cho_solve((U, lower), b)
```

4.2 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

4.3 Use case: sampling from multivariate normals

Python

```python
L = sp.linalg.cholesky(A, lower=True)
y = L @ rng.normal(size=n)
```

主要计算成本在构造 Cholesky，不在矩阵乘法。

4.4 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 → 不稳定
- 特别在 large correlation、large dimension 时常见

Python

```python
rng = np.random.default_rng(seed=1)
locs = rng.uniform(size = 100)
rho = .1
dists = np.abs(locs[:, np.newaxis] - locs)
C = np.exp(-dists**2/rho**2)
e = np.linalg.eig(C)
np.sort(e[0])[::-1][96:100]
# array([-3.58440747e-16+7.57451730e-17j, -3.58440747e-16-7.57451730e-17j, -4.98458053e-16+1.77854836e-16j, -4.98458053e-16-1.77854836e-16j])

try:
    L = np.linalg.cholesky(C)
except Exception as error:
    print(error)
Matrix is not positive definite

vals = np.abs(e[0])
np.max(vals)/np.min(vals)
# 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

---

5 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]

5.1 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

5.2 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

5.3 Regression and the QR in Python and R

- Python:

Python

```python
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 $Q^\top y$
- 提取 Q, R：

R

```r
X.qr = qr(X)
Q = qr.Q(X.qr)
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()`

5.4 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 ±1
  - 运算稳定、不改变向量长度

5.5 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
$$Q = I - 2uu^\top$$
  性质
  - $Qu = -u$
  - 若 $u^\top v = 0$，则 $Qv = v$
  - Q 对称且正交
- QR 的构造
$$R = Q_p \cdots Q_1 X, \qquad Q = (Q_p \cdots Q_1)^\top$$
- 第一步通过反射将第一列化为
$$(|x|, 0, \ldots, 0)$$
  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：
$$\mathrm{Cov}(\hat{\beta}) \propto (X^\top X)^{-1} = R^{-1} R^{-\top}$$
- $Q^\top Y$ 可分为
  - 前 p 个组成 $z^{(1)}$
  - 后 $n - p$ 个组成 $z^{(2)}$
  - SSR = $||z^{(1)}||^2$
  - 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

5.6 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：
$$R = Q_{pn} \cdots Q_{12} X, \qquad Q = (Q_{pn} \cdots Q_{12})^\top$$
- 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

5.7 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

5.8 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）

6 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

```python
Q, R = qr(A)
```

```python
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

6.1 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）