Matrix Computation in Statistics

Matteo Fasiolo

Introduction

About myself:

- I am Matteo Fasiolo, a statistics lecturer in the School of Mathematics from the University of Bristol
- My research is on non-linear regression modelling with applications in the energy domain
- I attended APTS in 2011 and I enjoyed it very much!

About the unit:

- Lec 1: matrix computation in Statistics
- Lec 2: an introduction to numerical optimisation
- Lec 3: optimisation and numerical differentiation (A. Lee)
- Lec 4 and 5: numerical integration and Monte Carlo (A. Lee)

Lec 1: what can we cover in 90min?

Main objective is to introduce common methods and problems encountered when dealing with matrices in a Statistical context.

The Linear Regression Problem

We will use linear regression as a running example.

It will be used to demonstrate several concepts and methods.

We have a response n-vector $\mathbf{y}$ and $(n \times p)$ design matrix $\mathbf{X}$.

Consider the model

$$y_i = \mathbf{X}_i^{\mathsf{T}} \beta + \epsilon_i,$$

with $\mathbb{E}(\epsilon_i) = 0$, $\text{var}(\epsilon_i) = \sigma^2$ and $\text{cov}(\epsilon_i, \epsilon_j) = 0$ if $i \neq j$.

We can fit it by least squares:

$$\hat{\beta} = \underset{\beta}{\text{argmin}} \, ||\mathbf{y} - \mathbf{X}\beta||^2.$$

The gradient w.r.t. β is

$$\nabla_\beta ||\mathbf{y} - \mathbf{X}\beta||^2 = 2\mathbf{X}^\mathsf{T}(\mathbf{y} - \mathbf{X}\beta),$$

and equating it to zero leads to the **normal equations**

$$\mathbf{X}^\mathsf{T}\mathbf{X}\hat{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{1}$$

which, if $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, has the well-known solution

$$\hat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{2}$$

Why does Equation 1 have its own name, given that Equation 2 gives us the estimator in closed-form?

From a theoretical point of view

$$\mathbf{X}^\mathsf{T}\mathbf{X}\hat{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y}, \quad \text{and} \quad \hat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y},$$

are equivalent (if $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible).

From a computational point of view they are not: the second equations, seems to imply that we need to compute $(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}$.

Key message: Computing matrix inverses should be avoided whenever possible!

We can often avoid inverting and get what we want (here, $\hat{\beta}$) at:

- **lower computational cost** and
- with **better numerical precision**.

Here we will achieve this by using different matrix decompositions.

[1] The Cholesky Decomposition

Define $\mathbf{\Sigma} = \mathbf{X}^\mathsf{T}\mathbf{X}$ and assume that $\mathbf{X}$ is of full-rank p.

Then, $\mathbf{\Sigma}$ is $(p \times p)$ positive-definite (PD) matrix, that is

$$\mathbf{z}^\mathsf{T}\mathbf{\Sigma}\mathbf{z} > 0, \quad \text{for any } \mathbf{z} \in \mathbb{R}^p.$$

PD matrices are the matrix analogue of positive real numbers.

Hence, we look for a matrix square-root of $\mathbf{\Sigma}$, s.t. $\mathbf{\Sigma} = \mathbf{R}^\mathsf{T}\mathbf{R}$.

To guarantee uniqueness we impose that $\mathbf{R}$ is upper-triangular.

Then it's easy to see how to compute $\mathbf{R}$.

Consider the $p = 3$ case

$$\begin{pmatrix} R_{11} & 0 & 0 \\ R_{12} & R_{22} & 0 \\ R_{13} & R_{23} & R_{33} \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12} & A_{22} & A_{23} \\ A_{13} & A_{23} & A_{33} \end{pmatrix}$$

Solving each row in turn:

$$\begin{aligned} A_{11} &= {R_{11}}^2 \\ A_{12} &= R_{11} R_{12} \\ A_{13} &= R_{11} R_{13} \\ A_{22} &= R_{12}^2 + {R_{22}}^2 \\ A_{23} &= R_{12} R_{13} + R_{22} R_{23} \\ &\dots \end{aligned}$$

So it's easy to compute $\mathbf{R}$. See notes for general formulas.

Cost is $O(p^3)$, short for $\frac{1}{3}p^3 + \frac{2}{3}p$ (for instance).

Back to the normal equations

$$\mathbf{X}^\mathsf{T}\mathbf{X}\hat{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y},$$

Plug in $\mathbf{X}^\mathsf{T}\mathbf{X} = \mathbf{R}^\mathsf{T}\mathbf{R}$ to get

$$\mathbf{R}^\mathsf{T}\mathbf{R}\hat{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y},$$

or

$$\hat{\beta} = (\mathbf{R}^\mathsf{T}\mathbf{R})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} = \mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}}\mathbf{X}^\mathsf{T}\mathbf{y}$$

Have we made any progress relative to $\hat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$?

Worst thing would be to compute $\mathbf{R}^{-1}$ (at $O(p^3)$ cost) and then:

1. $\hat{\beta} = (\mathbf{R}^{-1}(\mathbf{R}^{-\mathsf{T}}\mathbf{X}^\mathsf{T}))\mathbf{y}$ at $O(np^2) + O(np^2) + O(np)$ OR
2. $\hat{\beta} = ((\mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}})\mathbf{X}^\mathsf{T})\mathbf{y}$ at $O(p^3) + O(np^2) + O(np)$ OR (default)
3. $\hat{\beta} = \mathbf{R}^{-1}(\mathbf{R}^{-\mathsf{T}}(\mathbf{X}^\mathsf{T}\mathbf{y}))$ at $O(np) + O(p^2) + O(p^2)$ cost.

Instead, we should first compute $\mathbf{z} = \mathbf{X}^\mathsf{T}\mathbf{y}$ and plug in

$$\hat{\beta} = \mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}}\mathbf{X}^\mathsf{T}\mathbf{y} = \mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}}\mathbf{z}.$$

Let's focus on $\tilde{\mathbf{z}} = \mathbf{R}^{-\mathsf{T}}\mathbf{z}$, which is the solution of $\mathbf{R}^\mathsf{T}\tilde{\mathbf{z}} = \mathbf{z}$.

$\mathbf{R}^\mathsf{T}$ is lower-triangular so the system we are trying to solve is:

$$\begin{pmatrix} R_{11} & 0 & 0 \\ R_{12} & R_{22} & 0 \\ R_{13} & R_{23} & R_{33} \end{pmatrix} \begin{pmatrix} \tilde{z}_1 \\ \tilde{z}_2 \\ \tilde{z}_3 \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix}$$

which can be solved at $O(p^2)$ cost by forward-substitution

$$\begin{aligned} R_{11}\tilde{z}_1 &= z_1 \\ R_{12}\tilde{z}_1 + R_{22}\tilde{z}_2 &= z_2 \\ R_{13}\tilde{z}_1 + R_{23}\tilde{z}_2 + R_{33}\tilde{z}_3 &= z_3 \end{aligned}$$

Now we plug $\tilde{\mathbf{z}} = \mathbf{R}^{-\mathsf{T}}\mathbf{z}$ in

$$\hat{\beta} = \mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}}\mathbf{z} = \mathbf{R}^{-1}\tilde{\mathbf{z}},$$

and we use back-substitution to solve $\mathbf{R}\hat{\beta} = \tilde{\mathbf{z}}$ for $\hat{\beta}$.

Summarising the solution to normal equations $\mathbf{X}^{\mathsf{T}}\mathbf{X}\hat{\beta} = \mathbf{X}^{\mathsf{T}}\mathbf{y}$:

1. Compute $\mathbf{X}^{\mathsf{T}}\mathbf{X}$ at $O(np^2)$ and $\mathbf{X}^{\mathsf{T}}\mathbf{y}$ at $O(np)$ cost;
2. Cholesky decomposition $\mathbf{X}^{\mathsf{T}}\mathbf{X} = \mathbf{R}^{\mathsf{T}}\mathbf{R}$ at $O(p^3)$ cost;
3. Forward- and back-substitution at $O(p^2)$ cost each.

In step 2: inverting $\mathbf{X}^{\mathsf{T}}\mathbf{X}$ is also $O(p^3)$ but with **larger constant and less precision** (error in $\hat{\beta}$ is larger).

Objection: for $n \gg p$ computing $\mathbf{X}^{\mathsf{T}}\mathbf{X}$ is dominant cost so choice inversion vs Cholesky unimportant (computationally).

But $\mathbf{X}^\mathsf{T}\mathbf{X}$ is a Lev 3 **Basic Linear Algebra Subprogram (BLAS)**.

Very efficient implementations of Level 3 operations is provided by numerical linear algebra libraries such as LAPACK and OpenBLAS.

To see what you are using in R do:

```
sessionInfo()
Matrix products: default
BLAS:   /lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
LAPACK: /lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1
```

These are called when you do, e.g., `t(X) %*% X` in `R`.

So computing $\mathbf{X}^\mathsf{T}\mathbf{X}$ might be faster than inverting or Cholesky decomposing it.

Further, $\mathbf{X}^\mathsf{T}\mathbf{X}$ is easier to parallelise.

Cholesky Demonstration in R

Simulate data from $y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \epsilon_i$:

```
set.seed(1)
n <- 100
x <- runif(n)
X <- cbind(1, x, x^2)
beta <- c(1, 1, 1)
y <- X %*% beta + rnorm(n)
```

Then, to fit the model we do:

```
XtX <- t(X) %*% X # Tip: crossprod(X) is faster
```

```
( R <- chol(XtX) )
```

```
             x
  10 5.178471 3.3905117
x  0 2.662435 2.6977307
   0 0.000000 0.6541115
```

```r
z <- t(X) %*% y
```

```r
z_tilde <- forwardsolve(t(R), z)
```

```r
beta_hat <- backsolve(R, z_tilde)
```

Compare with `lm` function:

```r
cbind(beta_hat, lm(y ~ X - 1)$coef)
```

```
        [,1]      [,2]
X  0.7843701 0.7843701
Xx 1.5104773 1.5104773
X  0.8044581 0.8044581
```

But do not expect exactly the same results:

```r
max( abs(beta_hat - lm(y ~ X - 1)$coef) )
```

```
[1] 1.354472e-14
```

A Further Example on the Cholesky Decomposition

Consider the squared Mahalanobis distance in p dimensions

$$\text{dist}_{\mathbf{\Sigma}}(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1 - \mathbf{x}_2)^{\mathsf{T}} \mathbf{\Sigma}^{-1} (\mathbf{x}_1 - \mathbf{x}_2).$$

For fixed $\mathbf{\Sigma}$ we want to compute $\text{dist}_{\mathbf{\Sigma}}(\mathbf{x}_1, \mathbf{x}_2)$ for any $\mathbf{x}_1$ and $\mathbf{x}_2$.

Why not just compute $\mathbf{\Sigma}^{-1}$ once, and $\mathbf{\Sigma}^{-1}(\mathbf{x}_1 - \mathbf{x}_2)$ at $O(2p^2)$?

By inverting $\mathbf{\Sigma}^{-1}$ you lose accuracy and Cholesky is more efficient.

Instead, we compute $\mathbf{\Sigma} = \mathbf{R}^{\mathsf{T}}\mathbf{R}$ once, so $\mathbf{\Sigma}^{-1} = \mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}}$, then:

1. Compute $\mathbf{z} = \mathbf{R}^{-\mathsf{T}}(\mathbf{x}_1 - \mathbf{x}_2)$ by forward-substit at $O(\frac{1}{2}p^2)$;
2. Compute $\text{dist}_{\mathbf{\Sigma}}(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{z}^{\mathsf{T}}\mathbf{z}$.

So Cholesky is 4 times faster **under sequential computation**.

Sample applies when computing $\hat{\beta}$ for different $\mathbf{y}$ and fixed $\mathbf{X}$.

[2] Fitting Linear Models via the Eigen-Decomposition

Any symmetric $(p \times p)$ matrix, $\mathbf{A}$ can be written as

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathsf{T}} \tag{3}$$

where $\mathbf{U}^{\mathsf{T}}\mathbf{U} = \mathbf{U}\mathbf{U}^{\mathsf{T}} = \mathbf{I}$ and $\mathbf{\Lambda}$ is a diagonal, with $\mathbf{\Lambda}_{ii} = \lambda_i$.

By convention the eigen-values are decreasing, $\lambda_i \geq \lambda_{i+1}$.

This is the **eigen- or spectral decomposition** of $\mathbf{A}$.

If all eigen-values are $(\geq) > 0$ then $\mathbf{A}$ is positive (semi-)definite and

$$\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x} > 0, \quad \text{for any } \mathbf{x} \in \mathbb{R}^p.$$

Note: eigen-decomposition is complex to compute and costs $O(p^3)$.

For a positive definite $\mathbf{A}$, this provides **a** matrix square-root

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{\mathsf{T}} = \mathbf{U}\sqrt{\boldsymbol{\Lambda}}\sqrt{\boldsymbol{\Lambda}}\mathbf{U}^{\mathsf{T}} = \mathbf{U}\sqrt{\boldsymbol{\Lambda}}\mathbf{U}^{\mathsf{T}}\mathbf{U}\sqrt{\boldsymbol{\Lambda}}\mathbf{U}^{\mathsf{T}} = \sqrt{\mathbf{A}}\sqrt{\mathbf{A}}.$$

The inverse of $\mathbf{A}$ is easily computed

$$\mathbf{A}^{-1} = \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{U}^{\mathsf{T}},$$

assuming no eigen-value is zero.

Number of $\lambda_i \neq 0$ is matrix rank, if none is zero then $\mathbf{A}$ is full-rank.

Back to linear regression problem.

If $\mathbf{X}$ is full-rank, then $\mathbf{X}^\mathsf{T}\mathbf{X}$ is pos. def. and we can compute

$$\hat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}, \quad \rightarrow \quad \hat{\beta} = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^\mathsf{T}\mathbf{z}.$$

Doing $\hat{\beta} = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^\mathsf{T}\mathbf{z}$ involves:

1. $\mathbf{z}' = \mathbf{U}^\mathsf{T}\mathbf{z}$ (rotation);
2. $z_i'' = \lambda_i^{-1} z_i'$ (rescaling);
3. $\hat{\beta} = \mathbf{U}\mathbf{z}''$ (counter-rotation).

We have a problem if $\lambda_i = 0$!!

Not a surprise: An indefinite symmetric matrix is not invertible.

What if $\lambda_i \approx 0$, that is $\mathbf{A} = \mathbf{X}^\mathsf{T}\mathbf{X}$ is almost indefinite?

Assume distinct $\lambda_1, \ldots, \lambda_{p-1} \in [0.5, 1]$, $\lambda_p = O(\epsilon)$ with $0 < \epsilon \ll 1$.

Assume ϵ is precision of our machine.

That is, computer represents numbers one part in ϵ^{-1}.

I.e. 1 and $1 + z$ are the same number if $|z| < \epsilon$.

Similarly, x and $x + z$ are the same number if $|z| < |x|\epsilon$.

In R:

```
( eps <- .Machine$double.eps )
```

```
[1] 2.220446e-16
```

```
1 + eps == 1
```

```
[1] FALSE
```

```
1 + eps/2 == 1
```

```
[1] TRUE
```

Let $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_p$ be the eigen-vectors of $\mathbf{A}$.

Consider simple problem, we want to solve

$$\mathbf{A}\mathbf{x} = \mathbf{u}_1$$

with solution $\mathbf{x} = \mathbf{u}_1$ (but we ignore this!).

We can compute it by

$$\mathbf{x} = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^{\mathsf{T}}\mathbf{u}_1,$$

were first step is $\mathbf{u}_1' = \mathbf{U}^{\mathsf{T}}\mathbf{u}_1 = (1, 0, 0, \ldots, 0)^{\mathsf{T}}$ (in theory).

But, in practice, we obtain

$$\mathbf{u}_1' = \mathbf{U}^{\mathsf{T}}\mathbf{u}_1 = (1 + e_1, e_2, e_3, \ldots, e_p)^{\mathsf{T}},$$

where assume that size of errors $e_1, \ldots, e_p$ is $O(\epsilon)$, or o_ϵ.

Next: How will these errors propagate to the solution $\mathbf{x}$?

When computing

$$\mathbf{x} = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^{\mathsf{T}}\mathbf{u}_1 = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{u}_1',$$

with $\mathbf{u}_1' = (1 + o_\epsilon, o_\epsilon, o_\epsilon, \ldots, o_\epsilon)^{\mathsf{T}}$, next step is

$$\mathbf{u}_1'' = \mathbf{\Lambda}^{-1}\mathbf{u}_1' = \begin{bmatrix} \lambda_1^{-1} & & \\ & \ddots & \\ & & \lambda_p^{-1} \end{bmatrix} \begin{bmatrix} 1 + o_\epsilon \\ o_\epsilon \\ \ddots \\ o_\epsilon \end{bmatrix} = \begin{bmatrix} \lambda_1^{-1}(1 + o_\epsilon) \\ \lambda_2^{-1} o_\epsilon \\ \ddots \\ \lambda_p^{-1} o_\epsilon \end{bmatrix} \approx \begin{bmatrix} \lambda_1^{-1} \\ 0 \\ \ddots \\ 1 \end{bmatrix}.$$

Recall $\lambda_1 = 1$ so

$$\mathbf{x} = \mathbf{U}\mathbf{u}_1'' = \lambda_1^{-1}\mathbf{u}_1 + \mathbf{u}_p = \mathbf{u}_1 + \mathbf{u}_p \neq \mathbf{u}_1, \quad ||\mathbf{x}||^2 = 2 \text{ (very wrong!)}$$

If instead we assume $0 < \lambda_1 \ll 1$ then

$$\mathbf{x} = \lambda_1^{-1}\mathbf{u}_1 + \mathbf{u}_p \approx \lambda_1^{-1}\mathbf{u}_1.$$

Condition number $\kappa = |\lambda_1|/|\lambda_p|$ determines error in solution to:

$$\mathbf{A}\mathbf{x} = \mathbf{u}_1.$$

If $\kappa \approx 1/\epsilon$ then we'll have problems and system is **ill-conditioned**.

Note that problem occurs when solving $\mathbf{A}\mathbf{x} = \mathbf{u}_k$, $k = 1, \ldots, p-1$.

Solution to $\mathbf{A}\mathbf{x} = \mathbf{u}_p$ is fine (check it!).

Problem affects solution to generic system

$$\mathbf{A}\mathbf{x} = \mathbf{z},$$

because we can write $\mathbf{z} = \sum_k w_k \mathbf{u}_k$ and

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{z} = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^{\mathsf{T}} \sum_k w_k \mathbf{u}_k = \sum_k w_k \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^{\mathsf{T}}\mathbf{u}_k.$$

In linear regression case $\mathbf{x} = \beta$, $\mathbf{A} = \mathbf{X}^{\mathsf{T}}\mathbf{X}$ and $\mathbf{z} = \mathbf{X}^{\mathsf{T}}\mathbf{y}$.

Note that high κ does not imply that there is no hope.

Let $\mathbf{D}$ be diagonal with very high κ:

```
( D <- matrix(c(1e16, 0, 0, 1), 2, 2) )
```

```
      [,1] [,2]
[1,] 1e+16    0
[2,] 0e+00    1
```

```
lambda <- eigen(D)$values
```

```
lambda[1]/lambda[2]
```

```
[1] 1e+16
```

```
solve(D)
# Error in solve.default...
# system is computationally singular...
```

But inverse $\mathbf{D}^{-1}$ or solution to $\mathbf{Dx} = \mathbf{y}$ are easy!

Condition number extends beyond square matrices and eigen-values.

In general it quantifies sensitivity of a function to errors in its input.

Large condition number means that the function amplifies errors.

Definition of the condition number depends on the choice of norm.

Looking at κ motivates approach to linear regression that does not compute $\mathbf{X}^\mathsf{T}\mathbf{X}$ in

$$\mathbf{X}^\mathsf{T}\mathbf{X}\hat{\boldsymbol{\beta}} = \mathbf{X}^\mathsf{T}\mathbf{y}.$$

Back to regression problem

```
set.seed(1)
n <- 100
x <- runif(n)
X <-  cbind(1, x, x^2)
```

Let's look at some condition numbers

```
XtX <- crossprod(X)
kappa(XtX)
```

```
[1] 866.1662
```

```
kappa(X)
```

```
[1] 28.86055
```

```
kappa(X)^2
```

```
[1] 832.9311
```

So $\kappa(\mathbf{X}^\mathsf{T}\mathbf{X}) \approx \kappa(\mathbf{X})^2$: Working with $\mathbf{X}^\mathsf{T}\mathbf{X}$ can become a problem.

Let's shift our covariate x:

```
xs <- x + 100
Xs <- cbind(1, xs, xs^2)
XtXs <- crossprod(Xs)
```

Fit should stay the same: $\mathbf{X}\hat{\beta} = \mathbf{X}^S\hat{\beta}^S$ (check it analytically).

But:

```
kappa(Xs)
```

```
[1] 1596137602
```

```
kappa(XtXs)
```

```
[1] 2.225929e+18
```

Q1: How do we compute $\kappa(\mathbf{X})$?

Q2: Can we avoid decomposing $\mathbf{X}^\mathsf{T}\mathbf{X}$, and decompose $\mathbf{X}$ directly?

[3] The Singular Value Decomposition

If $\mathbf{X}$ is $(n \times p)$ matrix then its SVD decomposition is

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathsf{T}}$$

where

- $\mathbf{U}$ is a $(n \times p)$ matrix with orthogonal columns
- $\mathbf{V}$ is a $(p \times p)$ orthogonal matrix
- $\mathbf{D}$ is a $(p \times p)$ diagonal matrix

Singular values $d_1 \geq d_2 \geq \cdots \geq d_p$ are diagonal elements of $\mathbf{D}$.

Each d_i is equal to $\sqrt{\lambda_i(\mathbf{X}^{\mathsf{T}}\mathbf{X})}$.

Condition number of $\mathbf{X}$ is $\kappa(\mathbf{X}) = \frac{d_i}{d_p} = \sqrt{\kappa(\mathbf{X}^{\mathsf{T}}\mathbf{X})}$.

Can we solve least squares problem using SVD?

Back to normal equations

$$
\begin{aligned}
(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} &= (\mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T})^{-1}\mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{y} \\
&= (\mathbf{V}\mathbf{D}^2\mathbf{V}^\mathsf{T})^{-1}\mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{y} \\
&= \mathbf{V}^{-\mathsf{T}}\mathbf{D}^{-2}\mathbf{V}^{-1}\mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{y} \\
&= \mathbf{V}\mathbf{D}^{-2}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{y} \\
&= \mathbf{V}\mathbf{D}^{-1}\mathbf{U}^\mathsf{T}\mathbf{y}
\end{aligned}
$$

where $\mathbf{V}\mathbf{D}^{-1}\mathbf{U}$ has same κ as $\mathbf{X}$:

```
kappa(X)
```

```
[1] 28.86055
```

```
d <- svd(X)$d
d[1] / d[3]
```

```
[1] 26.43029
```

SVD($\mathbf{X}$) more accurate than Cholesky($\mathbf{X}^T\mathbf{X}$), but $O(np^2)$ vs $O(p^3)$.

Objection: $\mathbf{X}^T\mathbf{X}$ costs $O(np^2)$! But recall L3 BLAS efficiency...

```
n <- 100000; p <- 10
X <- matrix(rnorm(n*p), n, p)
XtX <- crossprod(X)
```

```
library(microbenchmark)
microbenchmark(XtX  = crossprod(X),
               chol = chol(XtX),
               eig = eigen(XtX, symmetric = TRUE),
               svd = svd(X),
               times = 10)
```

```
Unit: microseconds
 expr        min         lq       mean     median         uq
  XtX  2453.233  2656.532  2735.1213  2710.5325  2804.938
 chol     7.410    18.030    27.8788    29.9935    33.776
  eig    53.779    62.969   326.4207   109.1420   127.781
  svd 30670.809 32693.951 34450.2833 34556.6020 35719.945
```

In $n = p$ scenario:

```
n <- 1000; p <- 1000
X <- matrix(rnorm(n*p), n, p)
XtX <- crossprod(X)
```

```
microbenchmark(XtX  = crossprod(X),
               chol = chol(XtX),
               eig = eigen(XtX, symmetric = TRUE),
               svd = svd(X),
               times = 10)
```

```
Unit: milliseconds
 expr       min         lq       mean    median        uq
  XtX  29.26937  29.39607  33.81674  29.55714  30.55472  68
 chol  13.45008  13.56710  13.91465  13.71956  13.93872  15
  eig 241.76111 242.77174 245.73050 245.84178 246.54577 252
  svd 606.10923 606.99428 624.72142 618.80716 630.25739 673
```

Can we have the accuracy of $SVD(\mathbf{X})$ without paying the full price?

[4] The QR decomposition

If $\mathbf{X}$ is $(n \times p)$ matrix then its QR decomposition is

$$\mathbf{X} = \mathbf{QR},$$

where

- $\mathbf{Q}$ is $(n \times p)$ with orthogonal columns
- $\mathbf{R}$ is $(p \times p)$ upper triangular

Like SVD, it costs $O(np^2)$ but usually 1/3 of SVD.

To solve least squares do:

$$\begin{aligned}\hat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} &= (\mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{QR})^{-1}\mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{y} \\ &= (\mathbf{R}^\mathsf{T}\mathbf{R})^{-1}\mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{y} \\ &= \mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}}\mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{y} \\ &= \mathbf{R}^{-1}\mathbf{Q}^\mathsf{T}\mathbf{y}.\end{aligned}$$

Note that

$$\mathbf{X}^\mathsf{T}\mathbf{X} = \mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{Q}\mathbf{R} = \mathbf{R}^\mathsf{T}\mathbf{R},$$

so $\mathbf{R}$ factor of QR($\mathbf{X}$) is Cholesky factor of $\mathbf{X}^\mathsf{T}\mathbf{X}$.

What did we gain? Why compute Cholesky via expensive QR? Let:

- $\mathbf{R}$ be the true Cholesky factor
- $\hat{\mathbf{R}}_{Ch}$ be the output of QR on $\mathbf{X}$
- $\hat{\mathbf{R}}_{QR}$ be the output of Chol on $\mathbf{X}^\mathsf{T}\mathbf{X}$.

Each approach has an error, that is

$$\hat{\mathbf{R}}_{Ch}^\mathsf{T}\hat{\mathbf{R}}_{Ch} = \mathbf{X}^\mathsf{T}\mathbf{X} + \mathbf{E}_{Ch},$$

$$\hat{\mathbf{R}}_{QR}^\mathsf{T}\hat{\mathbf{R}}_{QR} = \mathbf{X}^\mathsf{T}\mathbf{X} + \mathbf{E}_{QR},$$

but QR is more accurate, that is $||\mathbf{E}_{QR}|| < ||\mathbf{E}_{Ch}||$.

This is because QR works with $\mathbf{X}$ which has a lower κ.

Smaller error in $\hat{\mathbf{R}}_{QR}$ leads to smaller error in $\hat{\beta}$.

Further, if $\mathbf{X}$ is ill-conditioned, Chol($\mathbf{X}^\mathsf{T}\mathbf{X}$) breaks before QR($\mathbf{X}$).

```
set.seed(1)
n <- 100
x <- runif(n)
xs <- x + 1e7
Xs <- cbind(1, xs, xs^2)
XtXs <- crossprod(Xs)
```

Then

```
C <- chol(XtXs)
# Error in chol.default(XtXs) :
#   the leading minor of order 3 is not positive
```

while

```
QR <- qr(Xs)
```

still works.

```r
n <- 100000; p <- 10  # n >> p
X <- matrix(rnorm(n*p), n, p); XtX <- crossprod(X)
```

```r
microbenchmark(XtX  = crossprod(X),
               chol = chol(XtX),
               eig = eigen(XtX, symmetric = TRUE),
               svd = svd(X),
               qr  = qr(X), times = 10)
```

```
Unit: microseconds
 expr        min         lq       mean     median         uq
  XtX  2733.012  2818.613  2876.6381  2864.1405  2953.997
 chol    10.506    19.770    32.8223    33.8465    39.668
  eig    51.397   111.058   115.0525   129.9285   134.261
  svd 26491.693 28449.205 29363.4988 29053.6610 29985.220 3
   qr 10404.237 10831.122 15913.3023 11125.3920 11770.592 5
```

So QR slower than Cholesky/Eigen but faster than SVD.

QR is what `stats::lm()` uses.

```r
n <- 1000; p <- 1000 # n = p
X <- matrix(rnorm(n*p), n, p); XtX <- crossprod(X)
```

```r
microbenchmark(XtX  = crossprod(X),
               chol = chol(XtX),
               eig = eigen(XtX, symmetric = TRUE),
               svd = svd(X),
               qr  = qr(X), times = 10)
```

```
Unit: milliseconds
 expr       min        lq      mean    median        uq
  XtX  29.29536  29.53522  31.05656  29.88605  30.45376  39
 chol  13.48713  13.52293  18.17425  14.50770  15.03922  51
  eig 238.13905 239.07331 254.09407 244.87891 264.24811 297
  svd 619.60280 623.08426 658.68510 642.44102 710.35909 725
   qr 220.74870 226.27763 240.71120 237.40329 258.84939 274
```

Here QR faster than SVD and competitive with $\mathbf{X}^\mathsf{T}\mathbf{X}$ + eigen.

For $n \approx p$ QR might offer more accuracy with some extra cost.

Conclusions

We used linear example to illustrate matrix computation.

Key message I: don't invert matrices unless you really have to.

In linear regression, you lose accuracy and do more computation.

Key message II: conditional number quantifies error propagation.

Key message III: tradeoff between accuracy and computing time.

On computing time, consider the scenario: $n \gg p$, $n \approx p$...

Order of computation $O(np^2)$ vs $O(p^3)$ is not the whole story.

You need to consider also:

- software (BLAS, compiled vs interpreted language, etc)
- parallelisation opportunities
- hardware (GPU vs CPU)