Biostatistics 615/815 Lecture 21:
Linear Algebra with C++

Hyun Min Kang

November 29th, 2011
Programming with Matrix

Why Matrix matters?

- Many statistical models can be well represented as matrix operations
  - Linear regression
  - Logistic regression
  - Mixed models

- Efficient matrix computation can make difference in the practicality of a statistical method

- Understanding C++ implementation of matrix operation can expedite the efficiency by orders of magnitude
Ways for Matrix programming in C++

- Implementing Matrix libraries on your own
  - Implementation can well fit to specific need
  - Need to pay for implementation overhead
  - Computational efficiency may not be excellent for large matrices
Ways for Matrix programming in C++

- Implementing Matrix libraries on your own
  - Implementation can well fit to specific need
  - Need to pay for implementation overhead
  - Computational efficiency may not be excellent for large matrices

- Using BLAS/LAPACK library
  - Low-level Fortran/C API
  - ATLAS implementation for gcc, MKL library for intel compiler (with multithread support)
  - Used in many statistical packages including R
  - Not user-friendly interface use.
  - boost supports C++ interface for BLAS
Ways for Matrix programming in C++

- Implementing Matrix libraries on your own
  - Implementation can well fit to specific need
  - Need to pay for implementation overhead
  - Computational efficiency may not be excellent for large matrices

- Using BLAS/LAPACK library
  - Low-level Fortran/C API
  - ATLAS implementation for gcc, MKL library for intel compiler (with multithread support)
  - Used in many statistical packages including R
  - Not user-friendly interface use.
  - boost supports C++ interface for BLAS

- Using a third-party library, Eigen package
  - A convenient C++ interface
  - Reasonably fast performance
  - Supports most functions BLAS/LAPACK provides
Using a third party library

Downloading and installing Eigen package

- To install - just decompress it, no need to build
Using a third party library

Downloading and installing Eigen package

- To install - just uncompress it, no need to build

Using Eigen package

- Add -I [PARENT_PATH_OF Eigen/] option when compile
- No need to install separate library. Including header files is sufficient
Example usages of Eigen library

```cpp
#include <iostream>
#include <Eigen/Dense>  // For non-sparse matrix
using namespace Eigen;  // avoid using Eigen::

int main()
{
    Matrix2d a;  // 2x2 matrix type is defined for convenience
    a << 1, 2,
              3, 4;
    MatrixXd b(2,2);  // but you can define the type from arbitrary-size matrix
    b << 2, 3,
            1, 4;
    std::cout << "a + b =\n" << a + b << std::endl;  // matrix addition
    std::cout << "a - b =\n" << a - b << std::endl;  // matrix subtraction
    std::cout << "Doing a += b;" << std::endl;
    a += b;
    std::cout << "Now a =\n" << a << std::endl;
    Vector3d v(1,2,3);  // vector operations
    Vector3d w(1,0,0);
    std::cout << "-v + w - v =\n" << -v + w - v << std::endl;
}
```

Hyun Min Kang

Biostatistics 615/815 - Lecture 21
November 29th, 2011 5 / 38
More examples

```cpp
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;

int main()
{
    Matrix2d mat; // 2*2 matrix
    mat << 1, 2,
          3, 4;
    Vector2d u(-1,1), v(2,0); // 2D vector
    std::cout << "Here is mat*mat:\n" << mat*mat << std::endl;
    std::cout << "Here is mat*u:\n" << mat*u << std::endl;
    std::cout << "Here is u^T*mat:\n" << u.transpose()*mat << std::endl;
    std::cout << "Here is u^T*v:\n" << u.transpose()*v << std::endl;
    std::cout << "Here is u*v^T:\n" << u*v.transpose() << std::endl;
    std::cout << "Let's multiply mat by itself" << std::endl;
    mat = mat*mat;
    std::cout << "Now mat is mat:\n" << mat << std::endl;
    return 0;
}
```
More examples

```cpp
#include <Eigen/Dense>
#include <iostream>
using namespace Eigen;

int main()
{
    MatrixXd m(2,2), n(2,2);
    MatrixXd result(2,2);
    m << 1,2,
        3,4;
    n << 5,6,7,8;
    result = m * n;
    std::cout << "-- Matrix m*n: --" << std::endl << result << std::endl << std::endl;
    result = m.array() * n.array();
    std::cout << "-- Array m*n: --" << std::endl << result << std::endl << std::endl;
    result = m.cwiseProduct(n);
    std::cout << "-- With cwiseProduct: --" << std::endl << result << std::endl << std::endl;
    result = (m.array() + 4).matrix() * m;
    std::cout << "-- (m+4)*m: --" << std::endl << result << std::endl << std::endl;
    return 0;
}
```
Time complexity of matrix computation

**Square matrix multiplication / inversion**

- Naive algorithm: $O(n^3)$
- Strassen algorithm: $O(n^{2.807})$
- Coppersmith-Winograd algorithm: $O(n^{2.376})$ (with very large constant factor)

**Determinant**

- Laplace expansion: $O(n!)$
- LU decomposition: $O(n^3)$
- Bareiss algorithm: $O(n^3)$
- Fast matrix multiplication algorithm: $O(n^{2.376})$
Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $u' A B v$
Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $\mathbf{u}' \mathbf{A} \mathbf{B} \mathbf{v}$
  - If the order is $(((\mathbf{u}' \mathbf{A} \mathbf{B}) \mathbf{v})$
    - $O(n^3) + O(n^2) + O(n)$ operations
    - $O(n^2)$ overall
Computational considerations in matrix operations

Avoiding expensive computation

- **Computation of** $u' A B v$
  - If the order is $(((u'(A B))v)$
    - $O(n^3) + O(n^2) + O(n)$ operations
    - $O(n^2)$ overall
  - If the order is $(((u' A)B)v)$
    - Two $O(n^2)$ operations and one $O(n)$ operation
    - $O(n^2)$ overall
Quadratic multiplication

Same time complexity, but one is slightly more efficient

- Computing $x' A y$.
- $O(n^2) + O(n)$ if ordered as $(x' A) y$.
- Can be simplified as $\sum_i \sum_j x_i A_{ij} y_j$

A symmetric case

- Computing $x' A x$ where $A = LL'$
- $u = L' x$ can be computed more efficiently than $A x$.
- $x' A x = u' u$
Solving linear systems

Problem
Find \( \mathbf{x} \) that satisfies \( A\mathbf{x} = \mathbf{b} \)

A simplest approach
- Calculate \( A^{-1} \), and \( \mathbf{x} = A^{-1}\mathbf{b} \)
- Time complexity is \( O(n^3) + O(n^2) \)
- \( A \) has to be invertible
- Potential issue of numerical instability
Using matrix decomposition to solve linear systems

**LU decomposition**

- $A = LU$, making $Ux = L^{-1}b$
- $A$ needs to be square and invertible.
- Fewer additions and multiplications
- Precision problems may occur

**QR decomposition**

- $A = QR$ where $A$ is $m \times n$ matrix
- $Q$ is orthogonal matrix, $Q'Q = I$.
- $R$ is $m \times n$ upper-triangular matrix, $Rx = Q'b$. 
Cholesky decomposition

- $A$ is a square, symmetric, and positive definite matrix.
- $A = U'U$ is a special case of LU decomposition
- Computationally efficient and accurate
Solving least square

Solving via inverse

- Most straightforward strategy
- \( \mathbf{y} = X\mathbf{\beta} + \mathbf{\epsilon}, \mathbf{y} \) is \( n \times 1 \), \( X \) is \( n \times p \).
- \( \mathbf{\beta} = (X'X)^{-1}X'y \).
- Computational complexity is \( O(np^2) + O(np) + O(p^3) \).
- The computation may become unstable if \( X'X \) is singular
- Need to make sure that \( rank(X) = p \).
Singular value decomposition

Definition

A \( m \times n \) matrix \( A \) can be represented as \( A = UDV^T \) such that

- \( U \) is \( m \times n \) matrix with orthogonal columns (\( U^T U = I_n \))
- \( D \) is \( n \times n \) diagonal matrix with non-negative entries
- \( V^T \) is \( n \times n \) matrix with orthogonal matrix (\( V^T V = VV^T = I_n \)).

Computational complexity

- \( 4m^2n + 8mn^2 + 9m^3 \) for computing \( U, V, \) and \( D \).
- \( 4mn^2 + 8n^3 \) for computing \( V \) and \( D \) only.
- The algorithm is numerically very stable.
Stable inference of least square using SVD

\[
X = UDV' \\
\beta = (X'X)^{-1}X'y \\
= (VDU'UDV')^{-1}VDU'y \\
= (VD^2V')^{-1}VDU'y \\
= VD^{-2}V'VDU'y \\
= VD^{-1}U'y
\]
#include <iostream>
#include <Eigen/Dense>

using namespace std;
#using namespace Eigen;

int main()
{
    MatrixXf A = MatrixXf::Random(3, 2);
    cout << "Here is the matrix A:\n" << A << endl;
    VectorXf b = VectorXf::Random(3);
    cout << "Here is the right hand side b:\n" << b << endl;
    cout << "The least-squares solution is:\n"
        << A.jacobiSvd(ComputeThinU | ComputeThinV).solve(b) << endl;
}
Linear Regression

Linear model

- \[ y = X\beta + \epsilon, \text{ where } X \text{ is } n \times p \text{ matrix} \]
- Under normality assumption, \[ y_i \sim N(X_i\beta, \sigma^2). \]

Key inferences under linear model

- Effect size: \[ \hat{\beta} = (X^TX)^{-1}X^Ty \]
- Residual variance: \[ \hat{\sigma}^2 = (y - X\hat{\beta})^T(y - X\hat{\beta})/(n - p) \]
- Variance/SE of \( \hat{\beta} \): \[ \text{Var}(\hat{\beta}) = \hat{\sigma}^2(X^TX)^{-1} \]
- p-value for testing \( H_0 : \beta_i = 0 \) or \( H_o : R\beta = 0 \). 

Hyun Min Kang

Biostatistics 615/815 - Lecture 21
November 29th, 2011 18 / 38
Using R to solve linear model

```r
> y <- rnorm(100)
> x <- rnorm(100)
> summary(lm(y~x))

Call:
  lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-2.15759 -0.69613  0.08565  0.70014  2.62488

Coefficients:                  Estimate   Std. Error  t value  Pr(>|t|)
(Intercept)            0.02722     0.10541    0.258   0.797
x                            -0.18369   0.10559   -1.740  0.085   .  
---
Signif. codes:  ***

Residual standard error: 1.05 on 98 degrees of freedom
Multiple R-squared: 0.02996, Adjusted R-squared: 0.02006
F-statistic: 3.027 on 1 and 98 DF,  p-value: 0.08505
```
Dealing with large data with `lm`

```r
> y <- rnorm(5000000)
> x <- rnorm(5000000)
> system.time(print(summary(lm(y~x))))

Call:
`lm(formula = y ~ x)`

Residuals:
```
       Min      1Q  Median      3Q     Max
-5.1310  -0.6746   0.0004   0.6747   5.0860
```

Coefficients:
```
                Estimate Std. Error   t value  Pr(>|t|)
(Intercept)   -0.000513 0.0004473   -1.147   0.251
         x        0.000236 0.0004473    0.527   0.598
```

Residual standard error: 1 on 4999998 degrees of freedom
Multiple R-squared: 5.564e-08, Adjusted R-squared: -1.444e-07
F-statistic: 0.2782 on 1 and 4999998 DF, p-value: 0.5979

user  system elapsed
57.434  14.229 100.607
```
A case for simple linear regression

A simpler model

- \( y = \beta_0 + x_1 \beta_1 + \epsilon \)
- \( X = [1 \; x], \quad \beta = [\beta_0 \; \beta_1]^T. \)

Question of interest

Can we leverage this simplicity to make a faster inference?
A faster inference with simple linear model

Ingredients for simplification

- \[ \sigma_y^2 = \frac{(y - \bar{y})^T(y - \bar{y})}{n - 1} \]
- \[ \sigma_x^2 = \frac{(x - \bar{x})^T(x - \bar{x})}{n - 1} \]
- \[ \sigma_{xy} = \frac{(x - \bar{x})^T(y - \bar{y})}{n - 1} \]
- \[ \rho_{xy} = \frac{\sigma_{xy}}{\sqrt{\sigma_x^2 \sigma_y^2}} \]

Making faster inferences

- \[ \hat{\beta}_1 = \rho_{xy} \sqrt{\frac{\sigma_y^2}{\sigma_x^2}} \]
- \[ \text{SE}(\hat{\beta}_1) = \sqrt{(n - 1)\sigma_y^2(1 - \rho_{xy}^2)/(n - 2)} \]
- \[ t = \frac{\rho_{xy} \sqrt{(n - 2)/(1 - \rho_{xy}^2)}} \] follows t-distribution with d.f. \( n - 2 \)
# note that this is an R function, not C++

```r
fastSimpleLinearRegression <- function(y, x) {
  y <- y - mean(y)
  x <- x - mean(x)
  n <- length(y)
  stopifnot(length(x) == n)              # for error handling
  s2y <- sum( y * y ) / ( n - 1 )       # \sigma_y^2
  s2x <- sum( x * x ) / ( n - 1 )       # \sigma_x^2
  sxy <- sum( x * y ) / ( n - 1 )       # \sigma_{xy}
  rxy <- sxy / sqrt( s2y * s2x )       # \rho_{xy}
  b <- rxy * sqrt( s2y / s2x )
  se.b <- sqrt( ( n - 1 ) * s2y * ( 1 - rxy * rxy ) / (n-2) )
  tstat <- rxy * sqrt( ( n - 2 ) / ( 1 - rxy * rxy ) )
  p <- pt( abs(t) , n - 2 , lower.tail=FALSE )*2
  return(list( beta = b , se.beta = se.b , t.stat = tstat, p.value = p ))
}
```
Now it became must faster

```r
> system.time(print(fastSimpleLinearRegression(y,x)))
$beta
[1] 0.0002358472

$se.beta
[1] 1.000036

$t.stat
[1] 0.5274646

$p.value
[1] 0.597871

user  system elapsed
0.382  1.849  3.042
```
Dealing with even larger data

Problem

- Supposed that we now have 5 billion input data points
- The issue is how to load the data
- Storing 10 billion double will require $80\, GB$ or larger memory
Dealing with even larger data

Problem

- Supposed that we now have 5 billion input data points
- The issue is how to load the data
- Storing 10 billion double will require 80 GB or larger memory

What we want

- As fast performance as before
- But do not store all the data into memory
- R cannot be the solution in such cases - use C++ instead
Streaming the inputs to extract sufficient statistics

**Sufficient statistics for simple linear regression**

1. \( n \)
2. \( \sigma_x^2 = \text{Var}(x) = (\mathbf{x} - \overline{x})^T(\mathbf{x} - \overline{x})/(n - 1) \)
3. \( \sigma_y^2 = \text{Var}(y) = (\mathbf{y} - \overline{y})^T(\mathbf{y} - \overline{y})/(n - 1) \)
4. \( \sigma_{xy} = \text{Cov}(x, y) = (\mathbf{x} - \overline{x})^T(\mathbf{y} - \overline{y})/(n - 1) \)
Streaming the inputs to extract sufficient statistics

**Sufficient statistics for simple linear regression**

1. \( n \)
2. \( \sigma_x^2 = \text{Var}(x) = (\mathbf{x} - \overline{x})^T(\mathbf{x} - \overline{x})/(n - 1) \)
3. \( \sigma_y^2 = \text{Var}(y) = (\mathbf{y} - \overline{y})^T(\mathbf{y} - \overline{y})/(n - 1) \)
4. \( \sigma_{xy} = \text{Cov}(x, y) = (\mathbf{x} - \overline{x})^T(\mathbf{y} - \overline{y})/(n - 1) \)

**Extracting sufficient statistics from stream**

- \( \sum_{i=1}^n x = n\overline{x} \)
- \( \sum_{i=1}^n y = n\overline{y} \)
- \( \sum_{i=1}^n x^2 = \sigma_x^2(n - 1) + n\overline{x}^2 \)
- \( \sum_{i=1}^n y^2 = \sigma_y^2(n - 1) + n\overline{y}^2 \)
- \( \sum_{i=1}^n xy = \sigma_{xy}(n - 1) + n\overline{xy} \)
Implementation: Streamed simple linear regression

```cpp
#include <iostream>
#include <fstream>
#include <boost/math/distributions/students_t.hpp>
using namespace boost::math; // for calculating p-values from t-statistic

int main(int argc, char** argv) {
    std::ifstream ifs(argv[1]); // read file from the file arguments
    double x, y; // temporary values to store the input
    double sumx = 0, sumsqx = 0, sumy = 0, sumsqy = 0, sumxy = 0;
    int n = 0;

    // extract a set of sufficient statistics
    while (ifs >> y >> x) { // assuming each input line feeds y and x
        sumx += x;
        sumy += y;
        sumxy += (x*y);
        sumsqx += (x*x);
        sumsqy += (y*y);
        ++n;
    }
    // ...
```

---

Hyun Min Kang
Biostatistics 615/815 - Lecture 21
November 29th, 2011
Streamed simple linear regression (cont’d)

// convert the set of sufficient statistics to
double s2y = (sumsqy - sumy*sumy/n)/(n-1);  // s2y = \sigma_y^2
double s2x = (sumsqx - sumx*sumx/n)/(n-1);  // s2x = \sigma_x^2
double sxy = (sumxy - sumx*sumy/n)/(n-1);   // sxy = \sigma_{xy}
double rxy = sxy/(s2x*s2y);                 // rxy = \text{cor}(x,y)

// calculate beta, SE(b), and p-values
double beta = rxy * s2y / s2x;
double seBeta = s2y * sqrt((n-1) * (1 - rxy*rxy) / (n-2));
double t = rxy * sqrt((n-2)/(1-rxy*rxy));       // t-statistics

students_t dist(n-2);  // use student's t-distribution to compute p-value
double pvalue = 2.0*cdf(complement(dist, t > 0 ? t : (0-t)));
Streamed simple linear regression (cont’d)

```cpp
std::cout << "Number of observations = " << n << std::endl;
std::cout << "Effect size - beta = " << beta << std::endl;
std::cout << "Standard error - SE(beta) = " << seBeta << std::endl;
std::cout << "Student's-t statistic = " << t << std::endl;
std::cout << "Two-sided p-value = " << pvalue << std::endl;
return 0;
}
Summary - Simple Linear Regression

- A linear regression with one predictor and intercept
- `lm()` function in R may be computationally slow for large input
- Faster inference is possible by computing a set of summary statistics in linear time
- Streaming via C++ programming further resolves the memory overhead
- The idea can be applied in more sophisticated, large-scale analyses.
Multiple regression - a general form of linear regression

Recap - Linear model

- \( y = X\beta + \epsilon \), where \( X \) is \( n \times p \) matrix
- Under normality assumption, \( y_i \sim N(X_i\beta, \sigma^2) \).

Key inferences under linear model

- Effect size: \( \hat{\beta} = (X^TX)^{-1}X^Ty \)
- Residual variance: \( \hat{\sigma}^2 = (y - X\hat{\beta})^T(y - X\hat{\beta})/(n - p) \)
- Variance/SE of \( \hat{\beta} \): \( \text{Var}(\hat{\beta}) = \hat{\sigma}^2(X^TX)^{-1} \)
- \( p \)-value for testing \( H_0: \beta_i = 0 \) or \( H_o: R\beta = 0 \).
Using `lm()` function in R

```r
> y <- rnorm(1000)
> X <- matrix(rnorm(5000),1000,5)
> summary(lm(y~X))

.....

Coefficients:

|            | Estimate | Std. Error | t value | Pr(>|t|) |
|------------|----------|------------|---------|----------|
| (Intercept)| 0.010934 | 0.031597   | 0.346   | 0.729    |
| X1         | 0.026340 | 0.031886   | 0.826   | 0.409    |
| X2         | -0.025339| 0.031789   | -0.797  | 0.426    |
| X3         | -0.036607| 0.031739   | -1.153  | 0.249    |
| X4         | -0.002549| 0.031467   | -0.081  | 0.935    |
| X5         | 0.050064 | 0.031665   | 1.581   | 0.114    |

Residual standard error: 0.9952 on 994 degrees of freedom
Multiple R-squared: 0.004966, Adjusted R-squared: -3.948e-05
F-statistic: 0.9921 on 5 and 994 DF, p-value: 0.4213
Implementing in C++: Using SVD for increasing reliability

\[ X = UDV' \]
\[ \hat{\beta} = (X^T X)^{-1} X^T y \]
\[ = (VDU^T UDV')^{-1} VDU^T y \]
\[ = (VD^2 V^T)^{-1} VDU^T y \]
\[ = VD^{-2} V^T VDU^T y \]
\[ = VD^{-1} U^T y \]
\[ \text{Cov}(\hat{\beta}) = \hat{\sigma}^2 (X' X)^{-1} \]
\[ = \hat{\sigma}^2 (VD^{-2} V^T) \]
\[ = \frac{(y - X\hat{\beta})^T (y - X\hat{\beta})}{n - p} (VD^{-1} (VD^{-1})^T) \]
Using Eigen library to implement multiple regression

```cpp
#include "Matrix615.h" // The class is posted at the web page
   // mainly for reading matrix from file
#include <iostream>
#include <Eigen/Core>
#include <Eigen/SVD>
using namespace Eigen;

int main(int argc, char** argv) {
    Matrix615<double> tmpy(argv[1]); // read n * 1 matrix y
    Matrix615<double> tmpX(argv[2]); // read n * p matrix X
    int n = tmpX.rowNums();
    int p = tmpX.colNums();

    MatrixXd y, X;
    tmpy.cloneToEigen(y); // copy the matrices into Eigen::Matrix objects
    tmpX.cloneToEigen(X); // copy the matrices into Eigen::Matrix objects
```
Implementing multiple regression (cont’d)

```cpp
JacobiSVD<MatrixXd> svd(X, ComputeThinU | ComputeThinV); // compute SVD
MatrixXd betasSvd = svd.solve(y); // solve linear model for computing beta
// calculate \( VD^{-1} \)
MatrixXd ViD = svd.matrixV() * svd.singularValues().asDiagonal().inverse();
double sigmaSvd = (y - X * betasSvd).squaredNorm()/(n-p); // compute \( \sigma^2 \)
MatrixXd varBetasSvd = sigmaSvd * ViD * ViD.transpose(); // \( \text{Cov}(\hat{\beta}) \)

// formatting the display of matrix.
IOFormat CleanFmt(8, 0, ",", ",\n", ",[", "]");

// print \( \hat{\beta} \)
std::cout << "----- beta -----\n" << betasSvd.format(CleanFmt) << std::endl;
// print SE(\( \hat{\beta} \)) -- diagonals of Cov(\( \hat{\beta} \))
std::cout << "----- SE(\beta) -----\n"
    << varBetasSvd.diagonal().array().sqrt().format(CleanFmt) << std::endl;
return 0;
```

Hyun Min Kang  
Biostatistics 615/815 - Lecture 21  
November 29th, 2011  
35 / 38
Working examples with $n = 1,000,000$, $p = 6$

Using R and `lm()` routines

```r
> system.time(y <- read.table('y.txt'))
user  system elapsed
 4.249   0.079   4.345
> system.time(X <- read.table('X.txt'))
user  system elapsed
62.013   0.658  62.314
> system.time(summary(lm(y~X)))
user  system elapsed
 5.849   1.228   7.703
```

Using C++ implementations

Elapsed time for matrix reading is 23.802
Elapsed time for computation is 1.19252
### Alternative implementations: speed-reliability tradeoffs

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>Method</th>
<th>Requirements on the matrix</th>
<th>Speed</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartialPivLU</td>
<td>partialPivLu()</td>
<td>Invertible</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>FullPivLU</td>
<td>fullPivLu()</td>
<td>None</td>
<td>-</td>
<td>+++</td>
</tr>
<tr>
<td>HouseholderQR</td>
<td>householderQr()</td>
<td>None</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>ColPivHouseholderQR</td>
<td>colPivHouseholderQr()</td>
<td>None</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>FullPivHouseholderQR</td>
<td>fullPivHouseholderQr()</td>
<td>None</td>
<td>-</td>
<td>+++</td>
</tr>
<tr>
<td>LLT</td>
<td>llt()</td>
<td>Positive definite</td>
<td>+++</td>
<td>+</td>
</tr>
<tr>
<td>LDLT</td>
<td>ldlt()</td>
<td>Positive or negative semidefinite</td>
<td>+++</td>
<td>++</td>
</tr>
</tbody>
</table>
Summary - Multiple regression

- Multiple predictor variables, and a single response variable.
- A reliable C++ implementation of linear model inference using SVD
- Eigen library provides a convenient and reasonably fast way to implement sophisticated matrix operations in C++
- C++ implementations may have advantages in both speed and memory in large-scale data analyses.