Biostatistics 615 - Statistical Computing

Lecture 14
Matrix Computations

Jian Kang

Nov 10, 2015

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

- Download at *http://eigen.tuxfamily.org/*
- To install - just uncompress it, no need to build

Using a third party library

Downloading and installing `Eigen` package

- Download at *http://eigen.tuxfamily.org/*
- To install - just uncompress it, no need to build

Using `Eigen` package

- Add `-I ~jiankang/Public/include` option (or include directory containing `Eigen/`) when compile
- No need to install separate library. Including header files is sufficient

Example usages of Eigen library

```
#include <iostream>
#include <Eigen/Dense>// For non-sparse matrix
using namespace Eigen;// avoid using Eigen::
using namespace std;
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;
   Matrix<double, 2, 3> c;
   c << 2, 3, 5, 7, 11, 13;
   cout << "a =\n" << a << endl;
   cout << "b =\n" << b << endl;
   cout << "c =\n" << c << endl;
   cout << "a + b =\n" << a + b << endl; // matrix addition
   cout << "a - b =\n" << a - b << endl; // matrix subtraction
   cout << "a * b =\n" << a * b << endl;//matrix multipication
   cout << "Doing a += b;" << endl;
   a += b;
   cout << "Now a =\n" << a << endl;
   Vector3d v(1,2,3);// vector operations
   Vector3d w(1,0,0);
   cout << "-v + w - v =\n" << -v + w - v << endl;
}
```

More examples

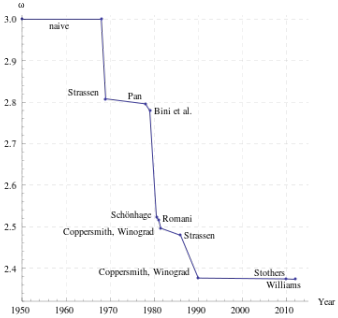
```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;
int main()
{
  Matrix2d mat;                // 2*2 matrix
  mat << 1, 2,
         3, 4;
  Vector2d u(-1,1), v(2,0);  // 2D vector
  cout << "Here is mat*mat:\n" << mat*mat << endl;
  cout << "Here is mat*u:\n"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

More examples

```
#include <Eigen/Dense>
#include <iostream>
using namespace Eigen;
using namespace std;
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;
  cout << "-- Matrix m*n: --" << endl << result << endl << endl;
  result = m.array() * n.array();
  cout << "-- Array m*n: --" << endl << result << endl << endl;
  result = m.cwiseProduct(n);
  cout << "-- With cwiseProduct: --" << endl << result << endl << endl;
  result = (m.array() + 4).matrix() * m;
  cout << "-- (m+4)*m: --" << endl << result << endl << endl;
  return 0;
}
```

Time complexity of square matrix multiplication

- Naive algorithm : $O(n^3)$
- Strassen algorithm (1969): $O(n^{2.807})$ (the fastest practical algorithm)
- Coppersmith-Winograd algorithm (1990): $O(n^{2.376})$
- François Le Gall (2014): $O(n^{2.373})$ (the best known algorithm)
- The best known lower bound: $\Omega(n^2)$ (or $\Omega(n^2 \log n)$ with certain assumptions).



(http://en.wikipedia.org/wiki/Matrix_multiplication#Algorithms_for_efficient_matrix_multiplication)

Strassen algorithm (Volker Strassen, 1969)

Goal: Given A, B, compute $C = AB$, where A, B, C are matrices of size $n \times n$ where $n = 2^k$.

Step 1: Partition A, B, C into submatrices of size $2^{k-1} \times 2^{k-1}$:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}.$$

Step 2: Compute the followings matrices:

$$\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{aligned}$$

(http://en.wikipedia.org/wiki/Strassen_algorithm)

Strassen algorithm (cont.)

Step 3: Compute the followings matrices:

$$
\begin{aligned}
C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} = M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} = M_3 + M_5 \\
C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} = M_2 + M_4 \\
C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} = M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Time complexity analysis

$$T(n) = 7\,T(n/2) + O(n^2)$$

Applying the master theorem, $T(n) = O(n^{\log_2 7}) = O(n^{2.807})$.

Time complexity for matrix inversion

- Matrix inversion can be reduced to matrix multiplication!

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} K^{-1} & -K^{-1}BD^{-1} \\ -D^{-1}CK^{-1} & D^{-1} + D^{-1}CK^{-1}BD^{-1} \end{bmatrix}$$

  where $K = A - BD^{-1}C$.
- Time complexity: $f(n) = 2f(n/2) + 6T(n/2) + O(n^2)$, where $T(n)$ is the time for matrix multiplication.
- Applying the master theorem, $f(n) = \Theta(T(n)) = O(n^{2.373})$.
- Best known lower bound: $\Omega(n^2 \log n)$.

(http://en.wikipedia.org/wiki/Invertible_matrix#Methods_of_matrix_inversion)

Time complexity for matrix determinant

Determinant

- Laplace expansion : $O(n!)$
- LU decomposition : $O(n^3)$
- Bareiss algorithm : $O(n^3)$
- Matrix determinant can also be reduced to matrix multiplication : $O(n^{2.373})$

(http://en.wikipedia.org/wiki/Determinant#Calculation)

Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $\mathbf{u}'AB\mathbf{v}$

Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $\mathbf{u}'AB\mathbf{v}$
- If the order is $(((\mathbf{u}'(AB))\mathbf{v})$
  - $O(n^3) + O(n^2) + O(n)$ operations
  - $O(n^3)$ overall

Computational considerations in matrix operations

Avoiding expensive computation

- Computation of $\mathbf{u}'AB\mathbf{v}$
- If the order is $(((\mathbf{u}'(AB))\mathbf{v})$
  - $O(n^3) + O(n^2) + O(n)$ operations
  - $O(n^3)$ overall
- If the order is $(((\mathbf{u}'A)B)\mathbf{v})$
  - 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 $\mathbf{x}'A\mathbf{y}$.
- $O(n^2) + O(n)$ if ordered as $(\mathbf{x}'A)\mathbf{y}$.
- Can be simplified as $\sum_i \sum_j x_i A_{ij} y_j$

A symmetric case

- Computing $\mathbf{x}'A\mathbf{x}$ where $A = LL'$ (Cholesky decomposition)
- $\mathbf{u} = L'\mathbf{x}$ can be computed more efficiently than $A\mathbf{x}$.
- $\mathbf{x}'A\mathbf{x} = \mathbf{u}'\mathbf{u}$

(http://en.wikipedia.org/wiki/Cholesky_decomposition)

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
- http://en.wikipedia.org/wiki/Invertible_matrix#Methods_of_matrix_inversion

Using matrix decomposition to solve linear systems

LU decomposition

- $A = LU$, making $U\mathbf{x} = L^{-1}\mathbf{b}$
- A needs to be square and invertible.
- Fewer additions and multiplications
- Precision problems may occur
- http://en.wikipedia.org/wiki/LU_decomposition#Algorithms

Cholesky decomposition

- A is a square, symmetric, and positive definite matrix.
- $A = U^T U$ is a special case of LU decomposition
- Computationally efficient and accurate
- http://en.wikipedia.org/wiki/Cholesky_decomposition#Computation

QR decomposition

- $A = QR$ where A is $m \times n$ matrix
- Q is orthogonal matrix, $Q^T Q = I$.
- R is $m \times n$ upper-triangular matrix, $R\mathbf{x} = Q^T\mathbf{b}$.
- http://en.wikipedia.org/wiki/QR_decomposition#Computing_the_QR_decomposition

Solving least square

Solving via inverse

- Most straightforward strategy
- $\mathbf{y} = X\beta + \epsilon$, $\mathbf{y}$ is $n \times 1$, X is $n \times p$.
- $\beta = (X^T X)^{-1} X^T \mathbf{y}$.
- Computational complexity is $O(np^2) + O(np) + O(p^3)$.
- The computation may become unstable if $X^T X$ is singular
- Need to make sure that $rank(X) = p$.
- http://en.wikipedia.org/wiki/Least_squares#Solving_the_least_squares_problem

Singular value decomposition

Definition

A $m \times n (m \geq n)$ 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

- $4m^2 n + 8mn^2 + 9m^3$ for computing U, V,and D.
- $4mn^2 + 8n^3$ for computing V and D only.
- The algorithm is numerically very stable
- http://en.wikipedia.org/wiki/Singular_value_decomposition#Calculating_the_SVD

THE book for matrix computations

Golub, Gene; Van Loan, Charles (2012) Matrix Computations, 4th edition.

Stable inference of least square using SVD

$$
\begin{aligned}
X &= UDV^T \\
\beta &= (X^T X)^{-1} X^T \mathbf{y} \\
&= (VDU^T UDV^T)^{-1} VDU^T \mathbf{y} \\
&= (VD^2 V^T)^{-1} VDU^T \mathbf{y} \\
&= VD^{-2} V^T VDU^T \mathbf{y} \\
&= VD^{-1} U^T \mathbf{y}
\end{aligned}
$$

Stable inference of least square using SVD

```
#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 model

- $\mathbf{y} = X\beta + \epsilon$, 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 : $\hat{\beta} = (X^T X)^{-1} X^T \mathbf{y}$
- Residual variance : $\widehat{\sigma^2} = (\mathbf{y} - X\hat{\beta})^T(\mathbf{y} - X\hat{\beta})/(n-p)$
- Variance/SE of $\hat{\beta}$: $\widehat{\mathrm{Var}}(\hat{\beta}) = \widehat{\sigma^2}(X^T X)^{-1}$
- p-value for testing $H_0 : \beta_i = 0$ or $H_o : R\beta = 0$.

Using R to solve linear model

```
> 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

```
> y = rnorm(5000000)
> x = rnorm(5000000)
> system.time(print(summary(lm(y~x))))

Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-5.2858 -0.6735  0.0004  0.6741  4.9432

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  9.312e-05  4.471e-04   0.208    0.835
x           -2.924e-04  4.471e-04  -0.654    0.513

Residual standard error: 0.9997 on 4999998 degrees of freedom
Multiple R-squared:  8.554e-08, Adjusted R-squared:  -1.145e-07
F-statistic: 0.4277 on 1 and 4999998 DF,  p-value: 0.5131

   user  system elapsed
 20.972   0.402  21.430
```

A case for simple linear regression

A simpler model

- $\mathbf{y} = \beta_0 + \mathbf{x}\beta_1 + \epsilon$
- $X = [1 \;\; \mathbf{x}]$, $\beta = [\beta_0 \;\; \beta_1]^T$.

Question of interest

Can we leverage this simplicity to make a faster inference?

A faster inference with simple linear model

Ingredients for simplification

- $\sigma_y^2 = (\mathbf{y} - \overline{y})^T(\mathbf{y} - \overline{y})/(n-1)$
- $\sigma_x^2 = (\mathbf{x} - \overline{x})^T(\mathbf{x} - \overline{x})/(n-1)$
- $\sigma_{xy} = (\mathbf{x} - \overline{x})^T(\mathbf{y} - \overline{y})/(n-1)$
- $\rho_{xy} = \sigma_{xy}/\sqrt{\sigma_x^2\sigma_y^2}$.

Making faster inferences

- $\hat{\beta}_1 = \rho_{xy}\sqrt{\sigma_y^2/\sigma_x^2}$
- $\mathrm{SE}(\hat{\beta}_1) = \sqrt{\sigma_y^2(1-\rho_{xy}^2)/(n-2)/\sigma_x^2}$
- $t = \rho_{xy}\sqrt{(n-2)/(1-\rho_{xy}^2)}$ follows t-distribution with d.f. $n-2$

A faster R implementation

```r
# note that this is an R function, not C++
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( s2y * ( 1 - rxy * rxy ) / (n-2) / s2x )
  tstat <- rxy * sqrt( ( n - 2 ) / ( 1 - rxy * rxy ) )
  p <- pt( abs(tstat) , n - 2 , lower.tail=FALSE )*2
  return(list( beta = b , se.beta = se.b , t.stat = tstat, p.value = p ))
}
```

Now it became much faster

```
>y = rnorm(5000000)
>x = rnorm(5000000)
> system.time(lm(y~x))
   user  system elapsed
 20.972   0.402  21.430
> system.time(fastSimpleLinearRegression(y,x))
   user  system elapsed
  0.078   0.000   0.078

>y = rnorm(100)
>x = rnorm(100)
>microbenchmark(lm(y~x),fastSimpleLinearRegression(y,x))

Unit: microseconds
                             expr     min       lq      mean  median        uq
                        lm(y ~ x) 876.358 888.8415 1141.2832 894.342 906.7325
 fastSimpleLinearRegression(y, x)  32.645  36.2755   44.0792  42.106  43.7080
       max neval
 18482.746   100
   219.605   100
```

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 $80GB$ 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 $80GB$ 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. $\sigma_x^2 = \widehat{\mathrm{Var}}(x) = (\mathbf{x} - \overline{x})^T(\mathbf{x} - \overline{x})/(n-1)$
3. $\sigma_y^2 = \widehat{\mathrm{Var}}(y) = (\mathbf{y} - \overline{y})^T(\mathbf{y} - \overline{y})/(n-1)$
4. $\sigma_{xy} = \widehat{\mathrm{Cov}}(x, y) = (\mathbf{x} - \overline{x})^T(\mathbf{y} - \overline{y})/(n-1)$

Streaming the inputs to extract sufficient statistics

Sufficient statistics for simple linear regression

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

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

```
#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;                  // temporay 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;
  }
```

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/sqrt(s2x*s2y);                     // rxy = cor(x,y)

// calculate beta, SE(beta), and p-values
double beta = rxy * sqrt(s2y / s2x);
double seBeta = sqrt( s2y / s2x * ( 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)

```
  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

- $\mathbf{y} = X\beta + \epsilon$, 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 : $\hat{\beta} = (X^T X)^{-1} X^T \mathbf{y}$
- Residual variance : $\widehat{\sigma^2} = (\mathbf{y} - X\hat{\beta})^T(\mathbf{y} - X\hat{\beta})/(n-p)$
- Variance/SE of $\hat{\beta}$: $\widehat{\mathrm{Var}}(\hat{\beta}) = \widehat{\sigma^2}(X^T X)^{-1}$
- p-value for testing $H_0 : \beta_i = 0$ or $H_o : R\beta = 0$.

Using lm() function in R

```
> y = rnorm(1000)
> X = matrix(rnorm(5000),1000,5)
> summary(lm(y~X))
Call:
lm(formula = y ~ X)

Residuals:
     Min       1Q   Median       3Q      Max
-2.80084 -0.69271 -0.00114  0.68395  2.98837

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.022873   0.030930   0.740    0.460
X1          -0.048975   0.031194  -1.570    0.117
X2          -0.057141   0.031838  -1.795    0.073 .
X3          -0.016190   0.031910  -0.507    0.612
X4           0.026239   0.031168   0.842    0.400
X5          -0.001209   0.031203  -0.039    0.969
---
Signif. codes:  0 ?***? 0.001 ?**? 0.01 ?*? 0.05 ?.? 0.1 ? ? 1

Residual standard error: 0.9779 on 994 degrees of freedom
Multiple R-squared:  0.007013, Adjusted R-squared:  0.002018
F-statistic: 1.404 on 5 and 994 DF,  p-value: 0.2202
```

Implementing in C++ : Using SVD for increasing reliability

$$
\begin{aligned}
X &= UDV^T \\
\hat{\beta} &= (X^TX)^{-1}X^T\mathbf{y} \\
&= (VDU^TUDV^T)^{-1}VDU^T\mathbf{y} \\
&= (VD^2V^T)^{-1}VDU^T\mathbf{y} \\
&= VD^{-2}V^TVDU^T\mathbf{y} \\
&= VD^{-1}U^T\mathbf{y} \\
\widehat{\mathrm{Cov}}(\hat{\beta}) &= \widehat{\sigma^2}(X^TX)^{-1} \\
&= \widehat{\sigma^2}(VD^{-2}V^T) \\
&= \frac{(\mathbf{y}-X\hat{\beta})^T(\mathbf{y}-X\hat{\beta})}{n-p}(VD^{-1}(VD^{-1})^T)
\end{aligned}
$$

Using Eigen library to implement multiple regression

```
#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)

```
  JacobiSVD<MatrixXd> svd(X, ComputeThinU | ComputeThinV);     // compute SVD
  MatrixXd betasSvd = svd.solve(y);  // solve linear model for computing beta
  // calcuate 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();   // 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 os Cov(\hat{beta})
  std::cout << "----- SE(beta) -----\n"
       << varBetasSvd.diagonal().array().sqrt().format(CleanFmt) << std::endl;
  return 0;
}
```

Copying Matrix615 to MatrixXd objects

```
template <class T>
void Matrix615<T>::cloneToEigen(Eigen::Matrix<T,Eigen::Dynamic,Eigen::Dynamic>& m)
{
  int nr = rowNums();
  int nc = colNums();
  m.resize(nr,nc);
  for(int i=0; i < nr; ++i) {
    for(int j=0; j < nc; ++j) {
      m(i,j) = data[i][j];
    }
  }
}
```

Working examples with $n = 1,000,000$, $p = 6$

Using R and `lm()` routines

```
> 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 in Eigen library: speed-reliability tradeoffs

Decomposition	Method	Requirements on the matrix	Speed	Accuracy
PartialPivLU	partialPivLu()	Invertible	++	+
FullPivLU	fullPivLu()	None	-	+++
HouseholderQR	householderQr()	None	++	+
ColPivHouseholderQR	colPivHouseholderQr()	None	+	++
FullPivHouseholderQR	fullPivHouseholderQr()	None	-	+++
LLT	llt()	Positive definite	+++	+
LDLT	ldlt()	Positive or negative semidefinite	+++	++

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.