Biostatistics 615/815 Lecture 14:
Implementing Linear Regression

Hyun Min Kang

February 22th, 2011
# Announcements

## Homework #4

- Homework 4 due is March 8th
- Floyd-Warshall algorithm
  - Note that the problem has been changed
  - Read CLRS chapter 25.2 for the full algorithmic detail
- Fair/biased coint HMM
  - Code skeleton has been updated using C++ class

## Midterm

- Midterm is on Thursday, March 10th.
- There will be a review session on Thursday 24th.
Recap - slowPower and fastPower

Function slowPower()

define slowPower( double a, int n ) {
  double x = a;
  for( int i=1; i < n; ++i )
    x *= a;
  return x;
}

Function fastPower()

define fastPower( double a, int n ) {
  if ( n == 1 )
    return a;
  else {
    double x = fastPower(a,n/2);
    if ( n % 2 == 0 )
      return x * x;
    else
      return x * x * a;
  }
}
Recap - ways to matrix programming

- 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
Recap - matrix decomposition to solve linear systems

- **LU decomposition**
  - $A = LU$, where $L$ is lower-triangular and $U$ is upper triangular matrix

- **QR decomposition**
  - $A = QR$ where $Q$ is unitary matrix $Q'Q = I$, and $R$ is upper-triangular matrix
  - $Ax = b$ reduces to $Rx = Q'b$.

- **Cholesky decomposition**
  - $A = U'U$ for a symmetric matrix
Linear Regression

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: $\hat{\beta} = (X^TX)^{-1} X^T y$
- 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 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:  
  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

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:

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>1Q</th>
<th>Median</th>
<th>3Q</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-5.1310</td>
<td>-0.6746</td>
<td>0.0004</td>
<td>0.6747</td>
<td>5.0860</td>
</tr>
</tbody>
</table>

Coefficients:

|                      | Estimate | Std. Error | t value | Pr(>|t|) |
|----------------------|----------|------------|---------|----------|
| (Intercept)          | -0.000513| 0.0004473  | -1.147  | 0.251    |
| x                    | 0.0002359| 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

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

Question of interest

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

Ingredients for simplification

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

Making faster inferences

- $\hat{\beta}_1 = \rho_{xy}\sqrt{\sigma_y^2/\sigma_x^2}$
- $SE(\hat{\beta}_1) = \sqrt{(n - 1)\sigma_y^2(1 - \rho_{xy}^2)/(n - 2)}$
- $t = \rho_{xy}\sqrt{(n - 2)/(1 - \rho_{xy}^2)}$ 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( ( 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 \( 80 \text{GB} \) 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. \( \sigma_x^2 = \text{Var}(x) = (x - \bar{x})^T(x - \bar{x})/(n-1) \)
3. \( \sigma_y^2 = \text{Var}(y) = (y - \bar{y})^T(y - \bar{y})/(n-1) \)
4. \( \sigma_{xy} = \text{Cov}(x, y) = (x - \bar{x})^T(y - \bar{y})/(n-1) \)
Streaming the inputs to extract sufficient statistics

**Sufficient statistics for simple linear regression**

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

**Extracting sufficient statistics from stream**

- \( \sum_{i=1}^{n} x = n\bar{x} \)
- \( \sum_{i=1}^{n} y = n\bar{y} \)
- \( \sum_{i=1}^{n} x^2 = \sigma_x^2(n - 1) + n\bar{x}^2 \)
- \( \sum_{i=1}^{n} y^2 = \sigma_y^2(n - 1) + n\bar{y}^2 \)
- \( \sum_{i=1}^{n} xy = \sigma_{xy}(n - 1) + n\bar{xy} \)
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;
    }
}
```
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 = cor(x,y)

// calculate beta, SE(beta), 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)

```
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))

> 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

\[
X = UDV' \\
\hat{\beta} = (X^TX)^{-1}X^Ty \\
= (VDU^T UDV')^{-1} VDU^Ty \\
= (VD^2V^T)^{-1} VDU^Ty \\
= VD^{-2}V^TVDU^Ty \\
= VD^{-1}U^Ty \\
\text{Cov}(\hat{\beta}) = \hat{\sigma}^2(X'X)^{-1} \\
= \hat{\sigma}^2(VD^{-2}V^T) \\
= \frac{(y - X\hat{\beta})^T(y - X\hat{\beta})}{n - p} (VD^{-1}(VD^{-1})^T)
\]
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.numRows();
    int p = tmpX.numCols();

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

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();  // Cov(\hat{\beta})

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

// 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;
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.
Summary: Part 2 - Matrix Computation

- Understanding the time complexity of matrix computations
- Practical usage of Eigen matrix library
- Brief overview on Matrix decomposition strategies
- C++ implementations of simple and multiple linear regression
Upcoming lectures

Next lecture

- Midterm review session - prepare your questions
- Homework #5 will be announced (due March 15th)

Tuesday March 8th

- More midterm reviews
- Random number generation
- Random sampling from a distribution

Thursday March 10th

- Midterm exam