Biostatistics 615/815 Lecture 13: Programming with Matrix

Hyun Min Kang

February 17th, 2011

---

Last lecture - Conditional independence in graphical models

Pr(A)  
\[ \begin{align*} 
A & \quad \text{Pr}(B|A) \\
B & \quad \text{Pr}(C|B) \quad \text{Pr}(E|B) \\
C & \quad \text{Pr}(D|B) \\
D & \\
E & 
\end{align*} \]

- \( \text{Pr}(A, C, D, E|B) = \text{Pr}(A|B) \text{Pr}(C|B) \text{Pr}(D|B) \text{Pr}(E|B) \)

Markov Blanket

- If conditioned on the variables in the gray area (variables with direct dependency), A is independent of all the other nodes.
- \( A \perp (U - A - \pi_A)|\pi_A \)

---

Announcements

Homework #3
- Homework 3 is due today
- If you’re using Visual C++ and still have problems in using boost library, you can ask for another extension

Homework #4
- Homework 4 is out
- Floyd-Warshall algorithm
  - Note that some key information was not covered in the class.
- Fair/biased coint HMM

---

Last lecture - Conditional independence in graphical models

Pr(A)  
\[ \begin{align*} 
A & \quad \text{Pr}(B|A) \\
B & \quad \text{Pr}(C|B) \quad \text{Pr}(E|B) \\
C & \quad \text{Pr}(D|B) \\
D & \\
E & 
\end{align*} \]

- \( \text{Pr}(A, C, D, E|B) = \text{Pr}(A|B) \text{Pr}(C|B) \text{Pr}(D|B) \text{Pr}(E|B) \)

Markov Blanket

- If conditioned on the variables in the gray area (variables with direct dependency), A is independent of all the other nodes.
- \( A \perp (U - A - \pi_A)|\pi_A \)
Hidden Markov Models

![Diagram of Hidden Markov Models]

Conditional dependency in forward-backward algorithms

- Forward: $(q_t, o_t) \perp o_{t-1} | q_{t-1}$.
- Backward: $o_{t+1} \perp o_{t+1}^+ | q_{t+1}$.

Viterbi algorithm - example

- When observations were (walk, shop, clean)
- Similar to Dijkstra’s or Manhattan tourist algorithm

Today’s lecture

- Calculating Power
- Linear algebra 101
- Using Eigen library for linear algebra
- Implementing a simple linear regression
Calculating power

**Problem**
- Computing $a^n$, where $a \in \mathbb{R}$ and $n \in \mathbb{N}$.
- How many multiplications would be needed?

**Function slowPower()**

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

**Computational time**

```cpp
int main(int argc, char** argv) {
    double a = 1.0000001;
    int n = 1000000000;
    clock_t t1 = clock();
    double x = slowPower(a,n);
    clock_t t2 = clock();
    double y = fastPower(a,n);
    clock_t t3 = clock();
    std::cout << "slowPower ans = " << x << " sec = " << (double)(t2-t1)/CLOCKS_PER_SEC << std::endl;
    std::cout << "fastPower ans = " << y << " sec = " << (double)(t3-t2)/CLOCKS_PER_SEC << std::endl;
}
```

**Summary - fastPower()**
- $\Theta(\log n)$ complexity compared to $\Theta(n)$ complexity of slowPower()
- Similar to binary search vs linear search
- Good example to illustrate how the efficiency of numerical computation could change by clever algorithms

**Running examples**
- slowPower ans = 2.6881e+43, sec = 1.88659
- fastPower ans = 2.6881e+43, sec = 3e-06
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

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;
}
```
# 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**: $O(n^{2.376})$

---

### More examples

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

using namespace Eigen;

int main()
{
    Matrix2d mat; // 2x2 matrix
    mat << 1, 2,
           3, 4;
    Vector2d u(-1.1), v(2.0); // 2D vector
    std::cout << "Here is mat: " << mat
               << std::endl;
    std::cout << "Here is mat*u: " << mat*u
               << std::endl;
    std::cout << "Here is u^T*mat: " << u.transpose()*mat
               << std::endl;
    mat = mat*mat;
    std::cout << "Now mat is mat: " << mat
               << std::endl;
    return 0;
}
```

---

### Computational considerations in matrix operations

#### Avoiding expensive computation

- Computation of $u^T A B v$:
  - If the order is $((u^T (AB))v)$
    - $O(n^3) + O(n^2) + O(n)$ operations
    - $O(n^2)$ overall
  - If the order is $((u^T A)B)v$
    - 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 $x'Ay$.
  - $O(n^2) + O(n)$ if ordered as $(x'A)y$.
  - Can be simplified as $\sum_i \sum_j x_i A_{ij} y_j$

#### A symmetric case

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

Problem
Find \( x \) that satisfies \( Ax = b \)

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

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

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, \( Q^T Q = I \).
- \( R \) is \( m \times n \) upper-triangular matrix, \( Rx = Q^T b \).

Solving least square

Solving via inverse
- Most straightforward strategy
  - \( y = X\beta + \epsilon \), \( y \) is \( n \times 1 \), \( X \) is \( n \times p \).
  - \( \beta = (X^T X)^{-1} X^T y \).
  - 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 \( \text{rank}(X) = p \).
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 (\( U^TU = I_n \))
- \( D \) is \( n \times n \) diagonal matrix with non-negative entries
- \( V^T \) is \( n \times n \) matrix with orthogonal matrix (\( V^TV = VV^T = I_n \)).

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

\[
X = UDV^T
\]
\[
\beta = (X'X)^{-1}X'y
\]
\[
= (VDU'UDV)^{-1} VDU'y
\]
\[
= (VD'^{-1}V^{-1}V^{-1}U'y
\]
\[
= VD^{-2}V'DU'y
\]
\[
= VD^{-1}U'y
\]

Summary

- Calculating Power
- Linear algebra 101
- Using Eigen library for linear algebra
- Implementing a simple linear regression

```cpp
#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;
}```