Biostatistics 615/815 Lecture 13: Programming with Matrix

Hyun Min Kang

February 17th, 2011

Annoucements

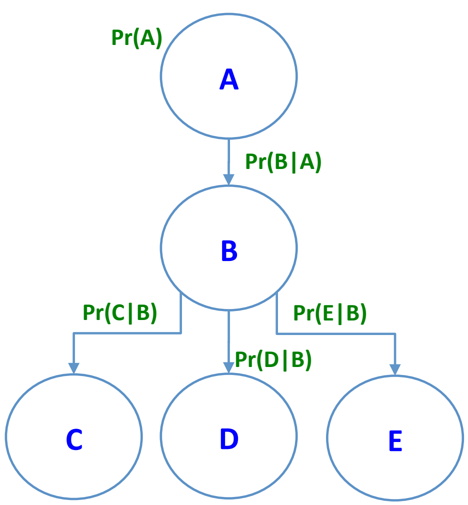
Homework #3

- 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

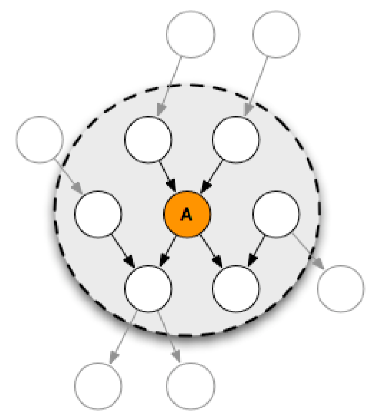
- Homework 4 is out
- Floyd-Warshall algorithm
  - Note that some key information was not covered in the class.
- Fair/biased coint HMM

Last lecture - Conditional independence in graphical models



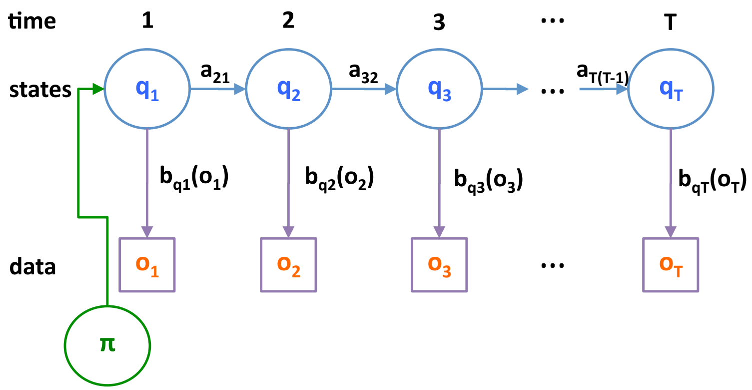
- $\Pr(A, C, D, E|B) = \Pr(A|B)\Pr(C|B)\Pr(D|B)\Pr(E|B)$

Markov Blanket



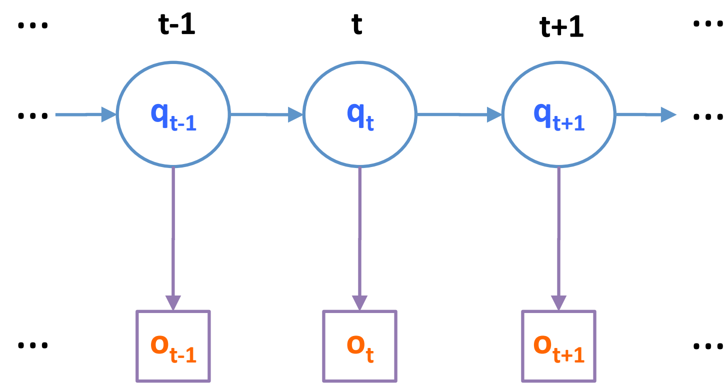
- If conditioned on the variables in the gray area (variables with direct dependency), A is independent of all the other nodes.
- $A \perp (U - A - \pi_A)|\pi_A$

Hidden Markov Models



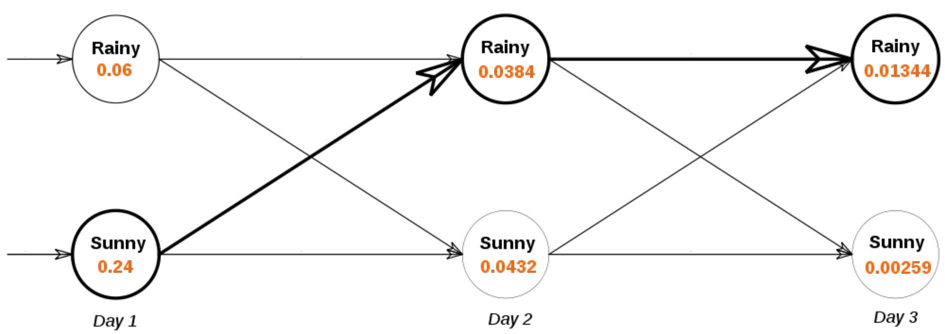
Conditional dependency in forward-backward algorithms

- Forward : $(q_t, o_t) \perp \mathbf{o}_t^- | \mathbf{q}_{t-1}$.
- Backward : $o_{t+1} \perp \mathbf{o}_{t+1}^+ | \mathbf{q}_{t+1}$.



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;
}
```


More efficient computation of power

Function fastPower()

```
double 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;
    }
  }
}
```


Computational time

main()

```
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;
}
```

Running examples

```
slowPower ans = 2.6881e+43, sec = 1.88659
fastPower ans = 2.6881e+43, sec = 3e-06
```


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

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 to Matrix programmming

- 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/index.php?title=3.0_beta*
- To install - just uncompress it

Using `Eigen` package

- Add `-I DOWNLOADED_PATH/eigen` option 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::
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;
}
```

More examples

```
#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;
}
```


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 $\mathbf{u}'AB\mathbf{v}$
- If the order is $(((\mathbf{u}'(AB))\mathbf{v})$
  - $O(n^3) + O(n^2) + O(n)$ operations
  - $O(n^2)$ 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 $A = LL'$
- $\mathbf{u} = L'\mathbf{x}$ can be computed more efficiently than $A\mathbf{x}$.
- $\mathbf{x}'A\mathbf{x} = \mathbf{u}'\mathbf{u}$

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 $U\mathbf{x} = \mathbf{L}^{-1}\mathbf{b}$
- 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, $R\mathbf{x} = Q'\mathbf{b}$.

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
- $\mathbf{y} = X\beta + \epsilon$, $\mathbf{y}$ is $n \times 1$, X is $n \times p$.
- $\beta = (X'X)^{-1}X'\mathbf{y}$.
- 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 $m \times n (m \geq n)$ matrix A can be represented as $A = UDV^T$ such that

- U is $m \times n$ matrix with orthogonal columns $(U'U = I_n)$
- D is $n \times n$ diagnonal matrix with non-negative entries
- V^T is $n \times n$ matrix with orthogonal matrix $(V'V = VV' = I_n)$.

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 inferecne of least square using SVD

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

Stable inferecne 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;
}
```


Summary

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