Biostatistics 615/815 Lecture 18: Multi-dimensional optimization

Hyun Min Kang

November 15th, 2012
Multidimensional Optimization: A mixture distribution
A general mixture distribution

\[ p(x; \pi, \phi, \eta) = \sum_{i=1}^{k} \pi_i f(x; \phi_i, \eta) \]

- \( x \): observed data
- \( \pi \): mixture proportion of each component
- \( f \): the probability density function
- \( \phi \): parameters specific to each component
- \( \eta \): parameters shared among components
- \( k \): number of mixture components
Problem: Maximum Likelihood Estimation

Finding Maximum-likelihood

Find parameters that maximizes the likelihood of the entire sample

\[ L = \prod_{i} p(x_i | \pi, \phi, \eta) \]

Calculating in log-space

Or equivalently, consider log-likelihood to avoid underflow

\[ l = \sum_{i} \log p(x_i | \pi, \phi, \eta) \]
Gaussian MLE in single-dimensional space

\[ p(x; \mu, \sigma^2) = \mathcal{N}(x; \mu, \sigma^2) \]

Given \( x \), what is the MLE parameters of \( \mu \) and \( \sigma^2 \)?
Gaussian MLE in single-dimensional space

\[ p(x; \mu, \sigma^2) = \mathcal{N}(x; \mu, \sigma^2) \]

Given \( x \), what is the MLE parameters of \( \mu \) and \( \sigma^2 \)?

- Analytical solution does exist
  \[ \hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i \]
  \[ \hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{\mu})^2 \]
### MLE in Gaussian mixture

#### Parameter estimation in Gaussian mixture

- No analytical solution
- Numerical optimization required
- Multi-dimensional optimization problem
  - $\mu_1, \mu_2, \ldots, \mu_k$
  - $\sigma^2_1, \sigma^2_2, \ldots, \sigma^2_k$

#### Possible approaches

- Simplex Method
- Expectation Maximization
- Markov-Chain Monte Carlo
The Simplex Method

- Calculate likelihoods at simplex vertexes
  - Geometric shape with $k + 1$ corners
  - A triangle in $k = 2$ dimensions

- Simplex crawls
  - Towards minimum
  - Away from maximum

- Probably the most widely used optimization method
How the Simplex Method Works
Simplex Method in Two Dimensions

- Evaluate functions at three vertexes
  - The highest (worst) point
  - The next highest point
  - The lowest (best) point

- Intuition
  - Move away from high point, towards low point
Direction for Optimization

- Line through worst point and average of other points
- Average of all points, excluding worst point
Reflection

This is the default new trial point
Reflection and Expansion

If reflection results in new minimum...

Move further along minimization direction.
Contraction (1-dimension)

Try a smaller step

If \( x' \) is still the worst point...
"passing through the eye of a needle"

If a simple contraction doesn't improve things, then try moving all points towards the current minimum
Summary: The Simplex Method
Implementing the Simplex Method

```cpp
template <class F> // F is a function object
class simplex615 { // contains (dim+1) points of size (dim)
  protected:
    std::vector<std::vector<double>> X; // (dim+1)*dim matrix
    std::vector<double> Y; // (dim+1) vector
    std::vector<double> midPoint; // variables for update
    std::vector<double> thruLine; // variables for update
  int dim, idxLo, idxHi, idxNextHi; // dimension, min, max, 2ndmax values
  void evaluateFunction(F& foo); // evaluate function value at each point
  void evaluateExtremes(); // determine the min, max, 2ndmax
  void prepareUpdate(); // calculate midPoint, thruLine
  bool updateSimplex(F& foo, double scale); // for reflection/expansion..
  void contractSimplex(F& foo); // for multiple contraction
  static int check_tol(double fmax, double fmin, double ftol); // check tolerance

public:
  simplex615(double* p, int d); // constructor with initial points
  void amoeba(F& foo, double tol); // main function for optimization
  std::vector<double>& xmin(); // optimal x value
  double ymin(); // optimal y value
};
```
Implementation overview

- Data representation
  - Each $X[i]$ is point of the simplex
  - $Y[i]$ corresponds to $f(X[i])$
  - midPoint is the average of all points (except for the worst point)
  - thruLine is vector from the worse point to the midPoint
Implementation overview

- Data representation
  - Each $X[i]$ is point of the simplex
  - $Y[i]$ corresponds to $f(X[i])$
  - midPoint is the average of all points (except for the worst point)
  - thruLine is vector from the worse point to the midPoint

- Reflection, Expansion and Contraction
  After calculating midPoint and thruLine
  - **Reflection** Call `updateSimplex(foo, -1.0)`
  - **Expansion** Call `updateSimplex(foo, -2.0)`
  - **Contraction** Call `updateSimplex(foo, 0.5)`
Initializing a Simplex

// constructor of simplex615 class : initial point is given

```cpp
template <class F>
simplex615<F>::simplex615(double* p, int d) : dim(d) {
    // set dimension
    // Determine the space required
    X.resize(dim+1);  // X is vector-of-vector, like 2-D array
    Y.resize(dim+1);  // Y is function value at each simplex point
    midPoint.resize(dim);
    thruLine.resize(dim);
    for(int i=0; i < dim+1; ++i) {
        X[i].resize(dim);  // allocate the size of content in the 2-D array
    }
    // Initially, make every point in the simplex identical
    for(int i=0; i < dim+1; ++i)
        for(int j=0; j < dim; ++j)
            X[i][j] = p[j];  // set each simple point to the starting point
    // then increase each dimension by one unit except for the last point
    for(int i=0; i < dim; ++i)
        X[i][i] += 1.;  // this will generate a simplex
}
```

Hyun Min Kang
Biostatistics 615/815 - Lecture 18
November 15th, 2012 18 / 46
Evaluating function values at each simplex point

// simple function for evaluating the function value at each simple point
// after calling this function Y[i] = foo(X[i]) should hold

```cpp
template <class F>
void simplex615<F>::evaluateFunction(F& foo) {
    for(int i=0; i < dim+1; ++i) {
        Y[i] = foo(X[i]); // foo is a function object, which will be visited later
    }
}
```
Determine the best, worst, and the second-worst points

```c++
template <class F>
void simplex615<F>::evaluateExtremes() {
    if ( Y[0] > Y[1] ) {  // compare the first two points
        idxHi = 0; idxLo = idxNextHi = 1;
    } else {
        idxHi = 1; idxLo = idxNextHi = 0;
    }
    // for each of the next points
    for (int i=2; i < dim+1; ++i) {
        if ( Y[i] <= Y[idxLo] ) {  // update the best point if lower
            idxLo = i;
        } else if ( Y[i] > Y[idxHi] ) {  // update the worst point if higher
            idxNextHi = idxHi; idxHi = i;
        } else if ( Y[i] > Y[idxNextHi] ) {  // update also if it is the 2nd-worst point
            idxNextHi = i;
        }
    }
}
```

Hyun Min Kang

Biostatistics 615/815 - Lecture 18
November 15th, 2012 20 / 46
Direction for Optimization

Line through worst point and average of other points

Average of all points, excluding worst point
Determining the direction for optimization

```cpp
template <class F>
void simplex615<F>::prepareUpdate() {
    for(int j=0; j < dim; ++j) {
        midPoint[j] = 0; // average of all points but the worst point
    }

    for(int i=0; i < dim+1; ++i) {
        if ( i != idxHi ) { // exclude the worst point
            for(int j=0; j < dim; ++j) {
                midPoint[j] += X[i][j];
            }
        }
    }

    for(int j=0; j < dim; ++j) {
        midPoint[j] /= dim; // take average
        thruLine[j] = X[idxHi][j] - midPoint[j]; // direction for optimization
    }
}
```
Updating simplex along the line

// scale determines which point to evaluate along the line
// scale = 1 : worse point, scale = 0 : midPoint

```cpp
template <class F>
bool simplex615<F>::updateSimplex(F& foo, double scale) {
    std::vector<double> nextPoint;   // next point to evaluate
    nextPoint.resize(dim);
    for(int i=0; i < dim; ++i) {
        nextPoint[i] = midPoint[i] + scale * thruLine[i];
    }
    double fNext = foo(nextPoint);
    if ( fNext < Y[idxHi] ) { // update only maximum values (if possible)
        for(int i=0; i < dim; ++i) { // because the order can be changed with
            X[idxHi][i] = nextPoint[i]; // evaluateExtremes() later
        }
        Y[idxHi] = fNext;
        return true;
    }
    else {
        return false; // never mind if worse than the worst
    }
}
```
Reflection

This is the default new trial point
Reflection and Expansion

If reflection results in new minimum...

Move further along minimization direction
Contraction (1-dimension)

Try a smaller step

If \( x' \) is still the worst point...
Multiple Contraction

"passing through the eye of a needle"

If a simple contraction doesn't improve things, then try moving all points towards the current minimum.
Updating simplex along the line

// if none of the tried points make things better
// reduce the search space towards the minimum point

template <class F>
void simplex615<F>::contractSimplex(F& foo) {
    for(int i=0; i < dim+1; ++i) {
        if ( i != idxLo ) {  // except for the minimum point
            for(int j=0; j < dim; ++j) {
                X[i][j] = 0.5*( X[idxLo][j] + X[i][j] );  // move the point towards minimum
            }
            Y[i] = foo(X[i]);  // re-evaluate the function
        }
    }
}
template <class F>
void simplex615<F>::amoeba(F& foo, double tol) {
  evaluateFunction(foo);  // evaluate the function at the initial points
  while(true) {
    evaluateExtremes();  // determine three important points
    prepareUpdate();  // determine direction for optimization

    if ( check_tol(Y[idxHi],Y[idxLo],tol) ) break;  // check convergence
    updateSimplex(foo, -1.0);  // reflection
    if ( Y[idxHi] < Y[idxLo] ) {
      updateSimplex(foo, -2.0);  // expansion
    }
    else if ( Y[idxHi] >= Y[idxNextHi] ) {
      if ( !updateSimplex(foo, 0.5) ) {  // 1-d contraction
        contractSimplex(foo);  // multiple contractions
      }
    }
  }
}
A general purpose minimization routine
  - Works in multiple dimensions
  - Uses only function evaluations
  - Does not require derivatives
Checking convergence

// Note that the function is declared as "static" function as
//
// static int check_tol(double fmax, double fmin, double ftol);
//
// because it does not use any member variables

template <class F>
int simplex615<F>::check_tol(double fmax, double fmin, double ftol) {
    // calculate the difference
    double delta = fabs(fmax - fmin);
    // calculate the relative tolerance
    double accuracy = (fabs(fmax) + fabs(fmin)) * ftol;
    // check if difference is within tolerance
    return (delta < (accuracy + ZEPS));
}
Using the Simplex Method Implementation

```cpp
#include <vector>
#include <cmath>
#include <iostream>
#include "simplex615.h"
#define ZEPS 1e-10

int main(int main, char** argv) {
    double point[2] = {-1.2, 1}; // initial point to start

    arbitraryFunc foo; // WILL BE DISCUSSED LATER
    simplex615<arbitraryFunc> simplex(point, 2); // create a simplex
    simplex.amoeba(foo, 1e-7); // optimize for a function

    // print outputs
    std::cout << "Minimum = " << simplex.ymin() << ", at (" <<
        simplex.xmin()[0] << ", " << simplex.xmin()[1] << ")" << std::endl;
    return 0;
}
```

Hyun Min Kang
Biostatistics 615/815 - Lecture 18
November 15th, 2012 32 / 46
// function object used as an argument

class arbitraryFunc {
    public:
        double operator() (std::vector<double>& x) {
            // f(x0,x1) = 100*(x1-x0^2)^2 + (1-x0)^2
            return 100*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0])+(1-x[0])*(1-x[0]);
        }
};
A working example

Minimum = 1.35567e-11, at (0.999999, 0.999997)
Normal Density

Normal density function

\[ f(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{x - \mu}{\sigma} \right)^2 \right] \]

Implementation

class NormMix615 {
public:
    static double dnorm(double x, double mu, double sigma) {
        return 1.0 / (sigma * sqrt(M_PI * 2.0)) *
            exp (-0.5 * (x - mu) * (x-mu) / sigma / sigma);
    }
    ...
};
Gaussian mixture distribution

Density function

\[ p(x|k, \pi, \mu, \sigma) = \sum_{i=1}^{k} \pi_i f(x|\mu_i, \sigma_i) \]

Implementation (within NormMix615)

```cpp
static double dmix(double x, std::vector<double>& pis,
                    std::vector<double>& means, std::vector<double>& sigmas) {
    double density = 0;
    for(int i=0; i < (int)pis.size(); ++i)
        density += pis[i] * dnorm(x,means[i],sigmas[i]);
    return density;
```
Likelihood of multiple observations

Calculating in log-space

\[
L = \prod_i p(x_i|\pi, \pi, \mu, \sigma)
\]
\[
l = \sum_i \log p(x_i|\pi, \mu, \sigma)
\]

Implementation (within NormMix615)

```cpp
static double mixLLK(std::vector<double>& xs, std::vector<double>& pis, std::vector<double>& means, std::vector<double>& sigmas) {
    int i=0;
    double llk = 0.0;
    for(int i=0; i < xs.size(); ++i)
        llk += log(dmix(xs[i], pis, means, sigmas));
    return llk;
}
```
class NormMix615 {
public: // these are internal function
    static double dnorm(double x, double mu, double sigma);
    static double dmix(...);
    static double mixLLK(...);
};

class LLKNormMixFunc {
public: // below are public functions
    LLKNormMixFunc(int k, std::vector<double>& y) :
        numComponents(k), data(y), numFunctionCalls(0) {} 
    // core function - called when foo() is used
    // x is the combined list of MLE parameters (pis, means, sigmas)
    double operator() (std::vector<double>& x);
std::vector<double> data;
    int numComponents;
    int numFunctionCalls;
};
Avoiding boundary conditions

Problem
- The simplex algorithm do not know that \( 0 \leq \pi_i \leq 1 \), and \( \sum_{i=1}^{n} \pi_i = 1 \)
- During the iteration of simplex algorithm, it is possible that \( \pi_i \) goes out of bound

Possible solutions
- Modify simplex algorithm to avoid boundary conditions
- Transform the parameter space to infinite ranges
Transforming the parameter space

**Constraints**

- $0 \leq \pi_i \leq 1$
- $\sum_{i=1}^{n} \pi_i = 1$

**Mapping between the space**

- Given $x \in \mathbb{R}^{n-1}$, for $i = 1, \cdots, n - 1$
- $\pi_i = \frac{1}{1+e^{-x_i}} (1 - \sum_{j=1}^{i-1} \pi_j)$
- $\pi_n = 1 - \sum_{i=1}^{n-1} \pi_i$. 
Implementing likelihood of data

double LLKNormMixFunc::operator() (std::vector<double>& x) {
    // x has (3*k-1) dims
    std::vector<double> priors;
    std::vector<double> means;
    std::vector<double> sigmas;
    // transform (k-1) real numbers to priors
    assignPriors(x, priors);
    for (int i=0; i < numComponents; ++i) {
        means.push_back(x[numComponents-1+i]);
        sigmas.push_back(x[2*numComponents-1+i]);
    }
    return 0-NormMix615::mixLLK(data, priors, means, sigmas);
}
Transforming between bounded and unbounded space of priors

```cpp
void LLKNormMixFunc::assignPriors(std::vector<double>& x, std::vector<double>& priors) {
    priors.clear();
    // convert priors (from [k-1]-d real scale to [k]-d simplex scale)
    double p = 1.;
    for(int i=0; i < numComponents-1; ++i) {
        double logit = 1./(1.+exp(0-x[i]));
        priors.push_back(p*logit);
        p = p*(1.-logit);
    }
    priors.push_back(p);
}
```
Simplex Method for Gaussian Mixture

```cpp
#include <iostream>
#include <fstream>
#include "simplex615.h"
#include "normMix615.h"
#include "llkNormMixFunc.h"
#define ZEPS 1e-10

int main(int main, char** argv) {
    double point[5] = {0, -1, 1, 1, 1}; // 50:50 mixture of N(-1,1) and N(1,1)
    simplex615<LLKNormMixFunc> simplex(point, 5);
    std::vector<double> data; // input data
    std::ifstream file(argv[1]); // open file
    double tok; // temporary variable
    while(file >> tok) data.push_back(tok); // read data from file
    LLKNormMixFunc foo(2, data); // 2-dimensional mixture model
    simplex.amoeba(foo, 1e-7); // run the Simplex Method
    std::cout << "Minimum = " << simplex.ymin() << " , at pi = "
                 << (1./(1.+exp(0.-simplex.xmin()[0]))) << "," << "between N("
                 << simplex.xmin()[1] << "," << simplex.xmin()[3] << ") and N("
                 << simplex.xmin()[2] << "," << simplex.xmin()[4] << ")" << std::endl;
    return 0;
}
```
A working example
A working example

Simulation of data

```r
> x <- rnorm(1000)
> y <- rnorm(500)+5
> write.table(matrix(c(x,y),1500,1),'mix.dat',row.names=F,col.names=F)
```

A Running Example

Minimum = 3043.46, at pi = 0.667271, 
between N(-0.0304604,1.00326) and N(5.01226,0.956009) 
(305 function evaluations in total)
Summary

Today

- Implementation of the Simplex Method
- Application to mixture of normal distributions

Recommended Readings

- Numerical recipes 10.5 - clear description of simplex method
- Subsequent sections contains more sophisticated multivariate normal distribution

Next Lecture

- The Expectation-Maximization Algorithm