Biostatistics 615/815 Lecture 20:
Expectation-Maximization (EM) Algorithm

Hyun Min Kang

March 31st, 2011

Recap - The Simplex Method

- General method for optimization
  - Makes few assumptions about function
- Crawls towards minimum using simplex
- Some recommendations
  - Multiple starting points
  - Restart maximization at proposed solution

Recap: Mixture of normals - Avoiding boundary conditions

```cpp
// from class mixLLKFunc...
virtual double 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
    double p = 1.;
    for(int i=0; i < numComponents-1; ++i) {
        double logit = 1./(1.+exp(-x[i]));
        priors.push_back(p*logit);
        p = p*(1.-logit);
    }
    priors.push_back(p);
    for(int i=0; i < numComponents; ++i) {
        means.push_back(x[numComponents-1+i]);
        sigmas.push_back(x[2*numComponents-1+i]);
    }
    return 0-mixLLK(data, priors, means, sigmas);
}
```
### Defining a function using inheritance

```cpp
// this is an abstract base class, which CAN NOT be used as class instance
class optFunc {
    public:
        // 'virtual' means inherited method can be used when
        // optFunc class is used via pointer or reference
        virtual double operator() (std::vector<double>& x) = 0; // function disabled
    };

    // Define a function inherits the function
    // when foo() is called at the simplex, this function is actually called
    class arbitraryOptFunc : public optFunc {
        public:
            virtual double operator() (std::vector<double>& x) {
                // 100*(x[1]-x[0]^2 + (1-x[0])^2
                return 100*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0])+(1-x[0])*(1-x[0]);
            }
    };
```

### Using rectangles and circles

```cpp
void printArea(rectangle& r) {
    std::cout << "Area = " << r.area() << std::endl;
}

void printArea(circle& c) {
    std::cout << "Area = " << c.area() << std::endl;
}

int main(int argc, char** argv) {
    rectangle r(3,4);
    circle c(1);
    printArea(r);
    printArea(c);
    return 0;
}
```

### An appetizer for dynamic polymorphism

```cpp
#include <cmath>
#include <vector>
#include <iostream>

class rectangle {
    public:
        double x;
        double y;
        double area() { return x*y; }
    }

class circle {
    public:
        double r;
        circle(double _r) : r(_r) {}
        double area() { return M_PI*r*r; }
    }

double area() { return M_PI*r*r; }
```

### Avoiding redundancy

```cpp
// We want to do something like this..
void printArea(shape& s) {
    std::cout << "Area = " << s.area() << std::endl;
}

int main(int argc, char** argv) {
    rectangle r(3,4);
    circle c(1);
    printArea(r);
    printArea(c);
    return 0;
}
```
Using class inheritance

```cpp
class shape {
    public:
        double area() { return -1; } // return a dummy value
    }

class rectangle : public shape {
    public:
        double x;
        double y;
        double area() { return x*y; }
    }

class circle : public shape {
    public:
        double r;
        circle(double _r) : r(_r) {}
        double area() { return M_PI*r*r; }
    }
```

What actually happens is..

```cpp
void printArea(shape& s) {
    std::cout << "Area = " << s.area() << std::endl;
}

int main(int argc, char** argv) {
    rectangle r(3,4);
    circle c(1);
    printArea(r); // -1 is printed... why?
    printArea(c); // -1 is printed... why?
    return 0;
}
```

Using 'virtual' to dynamically bind member functions

```cpp
class shape {
    // shape is an abstract class
    public:
        virtual double area() = 0; // shape object will never be created
    }

class rectangle : public shape {
    public:
        double x;
        double y;
        virtual double area() { return x*y; }
    }

class circle : public shape {
    public:
        double r;
        circle(double _r) : r(_r) {}
        virtual double area() { return M_PI*r*r; }
    }
```

A working example

```cpp
int main(int argc, char** argv) {
    rectangle r(3,4);
    circle c(1);
    printArea(r); // 12 is printed
    printArea(c); // 3.14159 is printed

    // must use pointers for referring object using a superclass type
    std::vector<shape*> myShapes; // myShape can store multiple types
    myShapes.push_back(new rectangle(2,3));
    myShapes.push_back(new circle(2));
    for(int i=0; i < (int)myShapes.size(); ++i) {
        printArea( *(myShapes[i]) ); // 6 and 12.5664 is printed
    }
}
```
Our previous examples

```cpp
class optFunc {
public:
    virtual double operator() (std::vector<double>& x) = 0;
};
class arbitraryOptFunc : public optFunc {
public:
    virtual double operator() (std::vector<double>& x) {
        return 100*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0])+(1-x[0])*(1-x[0]);
    }
};
class mixLLKFunc : public optFunc {
    ...
    std::vector<double> data;
    virtual double operator() (std::vector<double>& x) {
        ...
    }
};
```
Some citation records

- **The E-M algorithm**
  - Cited in over 19,624 research articles
- **The Simplex Method**
  - Nelder and Mead (1965) Computer Journal 7:308-313
  - Cited in over 10,727 research articles

**The Basic E-M Strategy**

- \( X = (Y, Z) \)
  - Complete data \( X \) - what we would like to have
  - Observed data \( Y \) - individual observations
  - Missing data \( Z \) - hidden / missing variables
- **The algorithm**
  - Use estimated parameters to infer \( Z \)
  - Update estimated parameters using \( Y \)
  - Repeat until convergence

**E-M strategy in Gaussian Mixtures**

**When are the E-M algorithms useful?**

- Problem is simpler to solve for complete data
  - Maximum likelihood estimates can be calculated using standard methods
  - Estimates of mixture parameters would be obtained straightforwardly
  - if the origin of each observation is known

**Filling in Missing Data in Gaussian Mixtures**

- Missing data is the group assignment of each observation
- Complete data generated by assigning observations to groups 'probabilistically'

**E-M formulation of Gaussian Mixture**

- Gaussian mixture distribution given \( \theta = (\pi, \mu, \sigma) \).

\[
p(x_i) = \sum_{k=1}^{K} \pi_k N(x_i | \mu_k, \sigma_k^2)
\]

- Introducing latent variable \( z \)
  - \( z_i \in \{1, \cdots, K\} \) is class assignment
- The marginal likelihood of observed data

\[
L(\theta; x) = p(x|\theta) = \sum z p(x, z|\theta)
\]

is often intractable
- Use complete data likelihood to approximate \( L(\theta; x) \)
The E-M algorithm

Expectation step (E-step)

- Given the current estimates of parameters $\theta^{(t)}$, calculate the conditional distribution of latent variable $z$.
- Then the expected log-likelihood of data given the conditional distribution of $z$ can be obtained

$$Q(\theta|\theta^{(t)}) = \mathbb{E}_{z|x,\theta^{(t)}} \left[ \log p(x, z|\theta) \right]$$

Maximization step (M-step)

- Find the parameter that maximize the expected log-likelihood

$$\theta^{(t+1)} = \arg \max_{\theta} Q(\theta|\theta^{(t)})$$

Implementing Gaussian Mixture E-M

class normMixEM {
public:
  int k; // # of components
  int n; // # of data
  std::vector<double> data; // observed data
  std::vector<double> pis; // pis
  std::vector<double> means; // means
  std::vector<double> sigmas; // sds
  std::vector<double> probs; // (nk) class probability

  normMixEM(std::vector<double>& input, int _k);
  void initParams();
  void updateProbs(); // E-step
  void updatePis(); // M-step (1)
  void updateMeans(); // M-step (2)
  void updateSigmas(); // M-step (3)
  double runEM(double eps);
};

Implementation of E-step

void normMixEM::updateProbs() {
  for(int i=0; i < n; ++i) {
    double cum = 0;
    for(int j=0; j < k; ++j) {
      probs[i*k+j] = pis[j]*mixLKLFunc::dnorm(data[i],means[j],sigmas[j]);
      cum += probs[i*k+j];
    }
    for(int j=0; j < k; ++j) {
      probs[i*k+j] /= cum;
    }
  }
}
Introduction

Dynamic Polymorphisms

E-M

Summary

Mixture of Normals : The M-step

- Update mixture parameters to maximize the likelihood of the data
- Becomes simple when we assume that the current class assignment are correct
- Simply use the same proportions, weighted means and variances to update parameters
- This step is guaranteed never to decrease the likelihood

Updating Mixture Proportions

\[ \pi_k = \frac{\sum_{i=1}^{n} Pr(z_i = k|x_i, \mu, \sigma^2)}{n} \]

- Count the observations assigned to each group

Updating Mixture Proportions - Implementations

```cpp
void normMixEM::updatePis() {
    for (int j=0; j < k; ++j) {
        pis[j] = 0;
        for (int i=0; i < n; ++i) {
            pis[j] += probs[i*k+j];
        }
        pis[j] /= n;
    }
}
```

Updating Component Means

\[ \hat{\mu}_k = \frac{\sum_{i} x_i Pr(z_i = k|x_i, \mu, \sigma^2)}{\sum_{i} Pr(z_i = k|x_i, \mu, \sigma^2)} \]

- Calculate weighted mean for group
- Weights are probabilities of group membership
### Updating Component Means - Implementations

```c
void normMixEM::updateMeans() {
    for (int j=0; j < k; ++j) {
        means[j] = 0;
        for (int i=0; i < n; ++i) {
            means[j] += data[i] * probs[i*k+j];
        }
        means[j] /= (n * pis[j] + TINY);
    }
}
```

### Updating Component Variances

\[
\sigma_k^2 = \frac{1}{n\pi_k} \sum_{i=1}^{n} (x_i - \mu_k)^2 \Pr(z_i = k | x_i, \mu, \sigma)
\]

- Calculate weighted sum of squared differences
- Weights are probabilities of group membership

### E-M Algorithm for Mixtures

1. Guesstimate starting parameters
2. Use Bayes’ theorem to calculate group assignment probabilities
3. Update parameters using estimated assignments
4. Repeat steps 2 and 3 until likelihood is stable
**Initializing the parameters**

```cpp
void normMixEM::initParams() {
    double sum = 0, sqsum = 0;
    for(int i=0; i < n; ++i) {
        sum += data[i];
        sqsum += (data[i]*data[i]);
    }
    double mean = sum/n;
    double sigma = sqrt(sqsum/n - sum*sum/n/n);
    for(int i=0; i < k; ++i) {
        pis[i] = 1/k; // uniform priors
        means[i] = data[rand() % n]; // pick random data points
        sigmas[i] = sigma; // pick uniform variance
    }
}
```

**Constructing normMixEM object**

```cpp
normMixEM::normMixEM(std::vector<double>& input, int _k) {
    data = input;
    k = _k;
    n = (int)data.size();
    pis.resize(k);
    means.resize(k);
    sigmas.resize(k);
    probs.resize(k * data.size());
}
```

**A working example**

```cpp
int main(int main, char** argv) {
    std::vector<double> data;
    std::ifstream file(argv[1]);
    std::vector<double> data;
    double tok;
    while(file >> tok) data.push_back(tok);
    normMixEM em(data,2);
    double minLLK = em.runEM(1e-6);
    std::cout << "Minimim = " << minLLK << " at pi = " << em.pis[0] << " ";
    return 0;
}
```

**Running example**

```
user@host ~/ ./mixEM ./mix.dat
Minimim = -3043.46, at pi = 0.667842,
between N(-0.0299457,1.00791) and N(5.0128,0.913825)
```
Summary : The E-M Algorithm

- Iterative procedure to find maximum likelihood estimate
  - E-step : Calculate the distribution of latent variables and the expected log-likelihood of the parameters given current set of parameters
  - M-step : Update the parameters based on the expected log-likelihood function
- The iteration does not decrease the marginal likelihood function
- But no guarantee that it will converge to the MLE
- Particularly useful when the likelihood is an exponential family
  - The E-step becomes the sum of expectations of sufficient statistics
  - The M-step involves maximizing a linear function, where closed form solution can often be found

Today
- Dynamic Polymorphisms in C++
- The E-M algorithm

Next lecture
- The Simulated Annealing