Biostatistics 615/815 Lecture 16:
Importance sampling
Single dimensional optimization

Hyun Min Kang

November 1st, 2012

The crude Monte-Carlo Methods

An example problem

Calculating

$$\theta = \int_0^1 f(x) \, dx$$

where \( f(x) \) is a complex function with \( 0 \leq f(x) \leq 1 \).

The problem is equivalent to computing \( E[f(u)] \) where \( u \sim U(0,1) \).

Algorithm

- Generate \( u_1, u_2, \cdots, u_B \) uniformly from \( U(0,1) \).
- Take their average to estimate \( \theta \)

$$\hat{\theta} = \frac{1}{B} \sum_{i=1}^{B} f(u_i)$$

Accept-reject (or hit-and-miss) Monte Carlo method

Algorithm

1. Define a rectangle \( R \) between \((0,0)\) and \((1,1)\)
   - Or more generally, between \((x_m, x_M)\) and \((y_m, y_M)\).
2. Set \( h = 0 \) (hit), \( m = 0 \) (miss).
3. Sample a random point \((x, y)\) \( \in R \).
4. If \( y < f(x) \), then increase \( h \). Otherwise, increase \( m \).
5. Repeat step 3 and 4 for \( B \) times
6. \( \hat{\theta} = \frac{h}{h+m} \).

$$\sigma_{AR}^2 - \sigma_{crude}^2 = \theta(1-\theta) - \frac{1}{B} E[f(u)^2] + \frac{\theta^2}{B}$$
$$= \theta - E[f(u)^2]$$
$$= \frac{1}{B} \int_0^1 f(u)(1-f(u)) \, du \geq 0$$

The crude Monte-Carlo method has less variance then accept-rejection method
Revisiting The Crude Monte Carlo

\[ \theta = E[f(u)] = \int_0^1 f(u) \, du \]
\[ \hat{\theta} = \frac{1}{B} \sum_{i=1}^B f(u_i) \]

More generally, when \( x \) has pdf \( p(x) \), if \( x_i \) is random variable following \( p(x) \),

\[ \theta_p = E_p[f(x)] = \int f(x)p(x) \, dx \]
\[ \hat{\theta}_p = \frac{1}{B} \sum_{i=1}^B f(x_i) \]

Importance sampling

Let \( x_i \) be random variable, and let \( p(x) \) be an arbitrary probability density function.

\[ \theta = E_n[f(x)] = \int f(x) \, dx = \int \frac{f(x)}{p(x)} p(x) \, dx = E_p \left[ \frac{f(x)}{p(x)} \right] \]

\[ \hat{\theta} = \frac{1}{B} \sum_{i=1}^B \frac{f(x_i)}{p(x_i)} \]

where \( x_i \) is sampled from distribution represented by pdf \( p(x) \)

Key Idea

- When \( f(x) \) is not uniform, variance of \( \hat{\theta} \) may be large.
- The idea is to pretend sampling from (almost) uniform distribution.

Analysis of Importance Sampling

Bias

\[ E[\hat{\theta}] = \frac{1}{B} \sum_{i=1}^B E_p \left[ \frac{f(x_i)}{p(x_i)} \right] = \frac{1}{B} \sum_{i=1}^B \theta = \theta \]

Variance

\[ \text{Var} [\hat{\theta}] = \frac{1}{B} \int \left( \frac{f(x)}{p(x)} - \theta \right)^2 p(x) \, dx \]
\[ = \frac{1}{B} E_p \left[ \left( \frac{f(x)}{p(x)} \right)^2 \right] - \theta^2 \]

The variance may or may not increase. Roughly speaking, if \( p(x) \) is similar to \( f(x) \), \( f(x)/p(x) \) becomes flattened and will have smaller variance.
Simulation of rare events

### Problem
- Consider a random variable $X \sim N(0, 1)$
- What is $\Pr[X \geq 10]$?

### Possible Solutions
- Let $f(x)$ and $F(x)$ be pdf and CDF of standard normal distribution.
- Then $\Pr[X \geq 10] = 1 - F(10) = 7.62 \times 10^{-24}$, and we’re all set.
- But what if we don’t have $F(x)$ but only $f(x)$?
  - In many cases, CDF is not easy to obtain compared to pdf or random draws.

### Accept-reject sampling
Sample random variables from $N(0, 1)$, and count how many of them are greater than 10
- How many random variables should be sampled to observe at least one $X \geq 10$?
  - $1/\Pr[X \geq 10] = 1.3 \times 10^{23}$

### Monte-Carlo Integration
- If we have pdf $f(x)$, $\Pr[X \geq 10] = \int_{10}^{\infty} f(x)\,dx$
- Use Monte-Carlo integration to compute this quantity
  1. Sample $B$ values uniformly from $[10, 10 + W]$ for a large value of $W$ (e.g. 50).
  2. Estimate $\hat{\theta} = \frac{1}{B} \sum_{i=1}^{B} f(u_i)$.

### An Example R code
```r
# pnormUpper() function to calculate Pr[x>t] using n random samples
pnormUpper <- function(n, t) {
  lo <- t
  hi <- t + 50  # hi is a reasonably large number

  # accept-reject sampling
  r <- rnorm(n)   # random sampling from N(0,1)
  v1 <- sum(r > t)/n  # count how many meets the condition

  # Monte-Carlo integration
  u <- runif(n,lo,hi)  # uniform sampling [t,t+50]
  v2 <- mean(dnorm(u)*(hi-lo))  # Monte-Carlo integration

  # importance sampling using N(t,1)
  g <- rnorm(n,t,1)  # sample from N(t,1)
  v3 <- sum((g > t) * dnorm(g)/dnorm(g,t,1)) / n;  # take a weighted average

  return (c(v1,v2,v3))  # return three values
}
```
Evaluating different methods

```r
# test pnormUpperTest(n,t) function using r times of repetition
pnormUpperTest <- function(r, n, t) {
gold <- pnorm(t, lower.tail=FALSE)  # gold standard answer
emp <- matrix(rnorm(r*nrow,ncol=3))  # matrix containing empirical answers
for(i in 1:r) { emp[,i] <- pnormUpper(n,t) }  # repeat r times
m <- colMeans(emp)  # get mean of the estimates
s <- apply(emp,2,sd)  # get std of the estimates
print("GOLD :", gold)
print("BIAS (ABSOLUTE) :", m-gold)
print("STDEV (ABSOLUTE) :", s)
print("BIAS (RELATIVE) :", (m-gold)/gold)
print("STDEV (RELATIVE) :", s/gold)
}
```

An example output

```r
> pnormUpperTest(100,10000,10)
[1] "GOLD :
[1] 7.619853e-24
[1] "BIAS (ABSOLUTE) :
[1] "STDEV (ABSOLUTE) :
[1] 0.000000e+00 3.917905e-24 7.559024e-25
[1] "BIAS (RELATIVE) :
[1] -1.00000000 -0.00734439 0.00630843
[1] "STDEV (RELATIVE) :
[1] 0.0000000 0.5141707 0.0992017
```

Another example output

```r
> pnormUpperTest(100,10000,10)
[1] "GOLD :
[1] 7.619853e-24
[1] "BIAS (ABSOLUTE) :
[1] "STDEV (ABSOLUTE) :
[1] 0.000000e+00 3.917905e-24 7.559024e-25
[1] "BIAS (RELATIVE) :
[1] -1.00000000 -0.00734439 0.00630843
[1] "STDEV (RELATIVE) :
[1] 0.0000000 0.5141707 0.0992017
```

1,000 importance sampling gives smaller variance than Monte-Carlo integration with 10,000 random samples.

Integral of probit normal distribution

- Disease risk score of an individual follows \( x \sim N(\mu, \sigma^2) \).
- Probability of disease \( \Pr(y = 1) = \Phi(x) \), where \( \Phi(x) \) is CDF of standard normal distribution.
- Want to compute the disease prevalence across the population.

\[
\theta = \int_{-\infty}^{\infty} \Phi(x) N(x; \mu, \sigma^2) \, dx
\]

where \( N(\cdot; \mu, \sigma^2) \) is pdf of normal distribution.
Monte-Carlo integration using uniform samples

1. Sample $x$ uniformly from a sufficiently large interval (e.g. $[-50, 50]$).
2. Evaluate integrals using

$$\hat{\theta} = \frac{1}{B} \sum_{i=1}^{B} \Phi(x_i) \mathcal{N}(x_i; \mu, \sigma^2)$$

Note that, for some $\mu$ and $\sigma^2$, $[-50, 50]$ may not be a sufficiently large interval.

Monte-Carlo integration using normal distribution

1. Sample $x$ from $\mathcal{N}(\mu, \sigma^2)$
2. Evaluate integrals by

$$\hat{\theta} = \frac{1}{B} \sum_{i=1}^{B} \Phi(x_i)$$

Two distributions are quite different – $\mathcal{N}(x; -8, 1^2)$ may not be an ideal distribution for Monte-Carlo integration.
Monte-Carlo integration by importance sampling

1. Sample $x$ from a new distribution
   - For example, $N(\mu', \sigma'^2)$
   - $\mu' = \frac{\mu}{\sigma}$
   - $\sigma' = \sigma$.

2. Evaluate integrals by weighting importance samples

   \[ \hat{\theta} = \frac{1}{B} \sum_{i=1}^{B} \left[ \Phi(x_i) \frac{N(x; \mu, \sigma^2)}{N(x; \mu', \sigma'^2)} \right] \]

An Example R code

```r
probitNormIntegral <- function(n, mu, sigma) {
  ## integration across uniform distribution
  lo <- -50
  hi <- 50
  u <- runif(n, lo, hi)
  v1 <- mean(dnorm(u, mu, sigma)*pnorm(u))*(hi-lo)

  ## integration using random samples from N(mu, sigma^2)
  g <- rnorm(n, mu, sigma)
  v2 <- mean(pnorm(g))

  ## importance sampling using N(mu', sigma^2)
  adjm <- mu/(sigma^2+1)
  r <- rnorm(n, adjm, sigma)
  v3 <- mean(pnorm(r)*dnorm(r, mu, sigma)/dnorm(r, adjm, sigma))
  return(c(v1, v2, v3))
}
```

Example Output

```r
> probitNormTest(100, 10000, -0.1)
[1] "MEAN :"
[1] 7.643951e-09 6.205931e-09 1.701978e-09
[1] "STDEV :"
[1] 1.579951e-09 1.239459e-08 1.019870e-10
[1] "STDEV (RELATIVE) :"
[1] 0.2869298 1.99721608 0.01324166

Importance sampling shows smallest variance.
```
Summary

- Crude Monte Carlo method
  - Use uniform distribution (or other original generative model) to calculate the integration
  - Every random sample is equally weighted.
  - Straightforward to understand
- Rejection sampling
  - Estimation from discrete count of random variables
  - Larger variance than crude Monte-Carlo method
  - Typically easy to implement
- Importance sampling
  - Reweight the probability distribution
  - Possible to reduce the variance in the estimation
  - Effective for inference involving rare events
  - Challenge is how to find the good sampling distribution.

Specific Objectives

Finding global minimum
- The lowest possible value of the function
- Very hard problem to solve generally

Finding local minimum
- Smallest value within finite neighborhood
- Relatively easier problem

A quick detour - The root finding problem

- Consider the problem of finding zeros for $f(x)$
- Assume that you know
  - Point $a$ where $f(a)$ is positive
  - Point $b$ where $f(b)$ is negative
  - $f(x)$ is continuous between $a$ and $b$
- How would you proceed to find $x$ such that $f(x) = 0$?
A C++ Example: defining a function object

```cpp
#include <iostream>

class myFunc {
  // a typical way to define a function object
  public:
    double operator()(double x) const {
      return (x*x-1);
    }
};

int main(int argc, char** argv) {
  myFunc foo;
  std::cout << "foo(0) = " << foo(0) << std::endl;
  std::cout << "foo(2) = " << foo(2) << std::endl;
}
```

Root Finding with C++

```cpp
// binary-search-like root finding algorithm
double binaryZero(myFunc foo, double lo, double hi, double e) {
  for (int i=0; ++i) {
    double d = hi - lo;
    double point = lo + d * 0.5;  // find midpoint between lo and hi
    double fpoint = foo(point);  // evaluate the value of the function
    if (fpoint < 0.0) {
      d = lo - point; lo = point;
    } else {
      d = point - hi, hi = point;
    }
    // e is tolerance level (higher e makes it faster but less accurate)
    if (fabs(d) < e || fpoint == 0.0) {
      std::cout << "Iteration " << i << ", point = " << point
                 << ", d = " << d << std::endl;
      return point;
    }
  }
}
```

Root Finding Using Linear Interpolation

```cpp
double linearZero (myFunc foo, double lo, double hi, double e) {
  double flo = foo(lo);  // evaluate the function at the end points
  double fhi = foo(hi);
  for (int i=0; ++i) {
    double d = hi - lo;
    double point = lo + d * flo / (flo - fhi);  //
    double fpoint = foo(point);
    if (fpoint < 0.0) {
      d = lo - point;
      lo = point;
      flo = fpoint;
    } else {
      d = point - hi;
      hi = point;
      fhi = fpoint;
    }
    if (fabs(d) < e || fpoint == 0.0) {
      std::cout << "Iteration " << i << ", point = " << point
                 << ", d = " << d << std::endl;
      return point;
    }
  }
}
```
### Performance Comparison

**Finding \( \sin(x) = 0 \) between \(-\pi/4 \) and \( \pi/2 \)**

```cpp
#include <cmath>

class myFunc {
    public:
        double operator()(double x) const { return sin(x); }
    ...

    int main(int argc, char** argv) {
        myFunc foo;
        binaryZero(foo, 0-M_PI/4, M_PI/2, 1e-5);
        linearZero(foo, 0-M_PI/4, M_PI/2, 1e-5);
        return 0;
    }
}
```

**Experimental results**

- `binaryZero()` : Iteration 17, point = -2.99606e-06, d = -8.98817e-06
- `linearZero()` : Iteration 5, point = 0, d = -4.47489e-18

### Summary on root finding

- Implemented two methods for root finding
  - Bisection Method : `binaryZero()`
  - False Position Method : `linearZero()`
- In the bisection method, the bracketing interval is halved at each step
- For well-behaved function, the False Position Method will converge faster, but there is no performance guarantee.

### R example of root finding

```r
> uniroot(sin, c(0-pi/4, pi/2))

$root
[1] -3.531885e-09

$froot
[1] -3.531885e-09

$iter
[1] 4

$estim.prec
[1] 8.719466e-05
```

### Back to the Minimization Problem

- Consider a complex function \( f(x) \) (e.g. likelihood)
- Find \( x \) which \( f(x) \) is maximum or minimum value
- Maximization and minimization are equivalent
  - Replace \( f(x) \) with \(-f(x)\)
Notes from Root Finding

- Two approaches possibly applicable to minimization problems
  - Bracketing
    - Keep track of intervals containing solution
  - Accuracy
    - Recognize that solution has limited precision

Outline of Minimization Strategy

1. Bracket minimum
2. Successively tighten bracket interval

Notes on Accuracy - Consider the Machine Precision

- When estimating minima and bracketing intervals, floating point accuracy must be considered
- In general, if the machine precision is $\epsilon$, the achievable accuracy is no more than $\sqrt{\epsilon}$.
- $\sqrt{\epsilon}$ comes from the second-order Taylor approximation
  \[
  f(x) \approx f(b) + \frac{1}{2} f''(b)(x - b)^2
  \]
- For functions where higher order terms are important, accuracy could be even lower.
  - For example, the minimum for $f(x) = 1 + x^4$ is only estimated to about $e^{1/4}$.

Detailed Minimization Strategy

1. Find 3 points such that
   - $a < b < c$
   - $f(b) < f(a)$ and $f(b) < f(c)$
2. Then search for minimum by
   - Selecting trial point in the interval
   - Keep minimum and flanking points
Part I: Finding a Bracketing Interval

- Consider two points
  - x-values $a$, $b$
  - y-values $f(a) > f(b)$

Part II: Finding Minimum After Bracketing

- Given 3 points such that
  - $a < b < c$
  - $f(b) < f(a)$ and $f(b) < f(c)$
- How do we select new trial point?
What is the best location for a new point $X$?

We want to minimize the size of next search interval, which will be either from $A$ to $X$ or from $B$ to $C$.

Minimizing worst case possibility

- Formulae

\[ w = \frac{b - a}{c - a} \]
\[ z = \frac{x - b}{c - a} \]

Segments will have length either $1 - w$ or $w + z$.

- Optimal case

\[ \begin{cases} 1 - w = w + z \\ \frac{z}{w} = w \end{cases} \]

- Solve It

\[ w = \frac{3 - \sqrt{5}}{2} = 0.38197 \]
The Golden Ratio

Bracketing Triplet

A

B

C

The number 0.38196 is related to the golden mean studied by Pythagoras

Golden Search

- Reduces bracketing by \( \sim 40\% \) after function evaluation
- Performance is independent of the function that is being minimized
- In many cases, better schemes are available
Golden Step

```cpp
#define GOLD 0.38196
#define ZEPS 1e-10 // precision tolerance
double goldenStep (double a, double b, double c) {
    double mid = (a + c) * .5;
    if (b > mid)
        return GOLD * (a - b);
    else
        return GOLD * (c - b);
}
```

Finding minimum of \( f(x) = -\cos(x) \)

```cpp
class myFunc {
public:
    double operator()(double x) const {
        return 0 - cos(x);
    }
};

int main(int argc, char** argv) {
    myFunc foo;
    goldenSearch(foo, 0-M_PI/4, M_PI/4, M_PI/2, 1e-5);
    return 0;
}
```

Results

\( i = 66, b = -4.42163e-09, f(b) = -1 \)
Further improvements

- As with root finding, performance can improve substantially when local approximation is used
- However, a linear approximation won’t do in this case.

Approximation Using Parabola

Summary

Today

- Root Finding Algorithms
  - Bisection Method: Simple but likely less efficient
  - False Position Method: More efficient for most well-behaved function
- Single-dimensional minimization
  - Golden Search

Next Lecture

- More Single-dimensional minimization
  - Brent’s method
- Multidimensional optimization
  - Simplex method